Propagation d'informations sémantiques dans un pipeline de compilation vérifié

Propagation of semantic information in a verified compilation pipeline

Benjamin Bonneau
Version for the jury

Version with corrections suggested by the reviewers

This document contains links to our Rocq development, which is publicly available at:

https://gricad-gitlab.univ-grenoble-alpes.fr/these-bbonneau/artifacts/-/tree/thesis

This manuscript is meant to be self-contained. Specific proficiency in Rocq is not a prerequisite. Some names used in this document differ from the ones used in the development. The repository contains a correspondance with the manuscript.

Table of contents

Résumé en français 7

I Background and motivations 18

1 Introduction 19

1.1 Compilation ... 19
1.2 Formal verification ... 20
1.2.1 Properties of programs ... 20
1.2.2 Correctness properties of compilers ... 22
1.2.3 Formally verified programs and compilers ... 23
1.3 Propagation of information in compilation pipelines ... 24
1.3.1 The need for semantic information during compilation ... 24
1.3.2 Sources of information ... 25
1.3.3 Encoding information in intermediate representations ... 26
1.3.4 The case of formally verified programs and compilers ... 26
1.4 Contributions ... 27
1.5 Conventions ... 28
1.5.1 Link with the development ... 28
1.5.2 Notations ... 28
1.5.3 Standard definitions ... 29
1.6 Outline ... 30

2 State of the art 31

2.1 Undefined Behaviors and their limitations ... 31
2.1.1 Encoding information in the semantics ... 31
2.1.2 Implicit dependencies of UBs ... 32
2.1.3 Mitigating restrictions of UBs with poison values ... 32
2.2 Memory models ... 33
2.3 Representation of aliasing information ... 35
2.4 Verified compilers and intermediate representations ... 36
2.5 Verifying programs with memory ... 38

3 Frag IR: an intermediate representation to propagate information 40

3.1 Tracking assumptions with dependent types ... 40
3.1.1 Static justification of information with type systems ... 40
3.1.2 Propositional variables with dependent types ... 41
3.1.3 Deriving properties with lemmas ... 42
3.1.4 Assertions to recover information using Undefined Behaviors ... 43
3.2 Controlling aliasing with memory fragments ... 44
3.2.1 Usage of memory fragments ... 44
3.2.2 Separating disjoint parts of the memory with heaplets ... 45
3.2.3 Storing the state of the memory in memory fragments ... 46
3.2.4 Operations on memory fragments ... 47
3.2.5 Linearity condition ... 48
3.2.6 Non-disjointness of the fragments ... 49
3.2.7 Ghost transformations on memory fragments with generalized lemmas ... 50
3.2.8 Transformations on programs with memory fragments ... 53

3.3 Pipeline 55
3.4 Related works 56

II Modular Intermediate Representations and passes 58

4 `SMonad`: a library to define denotational semantics 59

4.1 Semantics 60
4.1.1 Formal definitions of semantics 60
4.1.2 Operational semantics 61
4.1.3 Denotational semantics 64
4.2 Axiomatization 68
4.2.1 Domain 68
4.2.2 Operators 70
4.2.3 Refinement and axioms 71
4.2.4 Additional effects 73
4.3 Using the theory to prove transformations 78
4.3.1 Relational reasoning 78
4.3.2 Using relation on events for inter-procedural transformations 79
4.3.3 Derived unary predicates 81
4.4 Derived properties 81
4.4.1 Proving loop transformations using the codiagonal property 81
4.4.2 Function inlining 86
4.4.3 Inconsistent properties 88
4.5 Transformers 91
4.5.1 Adding external events 92
4.5.2 Adding a state 93
4.6 Related works 94
4.6.1 Semantics in verified compilers 94
4.6.2 Axiomatization of denotational domains 95
4.6.3 Abstraction of effects 95
4.6.4 Relations on events 96
4.6.5 Transformers 96
4.6.6 Undefined Behaviors 96
4.6.7 Non-determinism 97

5 A LTS-based model 98

5.1 Definition of the domain 98
5.2 Operators 100
5.3 Simulations 103
5.4 Behaviors of LTSs 104
5.4.1 Set of behaviors of a LTS 104
5.4.2 Comparison with the formalization used by CompCert 105
5.5 Connection with operational semantics 106
5.5.1 `FromEval`: equivalences with big-step semantics 107
5.5.2 `FromOperational`: equivalences with small-step semantics 108
5.5.3 Application: a library to define interpreters 110
5.5.4 Example: a denotational semantics for Cminor 114
5.6 Related works: simulations 118

6 A parameterized representation in functional SSA 120
6.1 Static Single Assignment ... 120
6.1.1 Common properties of SSA ... 121
6.1.2 Standard SSA ... 121
6.1.3 Functional SSA ... 121
6.2 Our functional SSA IR ... 122
6.3 Two syntactic representations in Rocq ... 124
6.3.1 Well-formed, Dependently-Typed Syntax (DTS) ... 124
6.3.2 Extracted, Algebraic Syntax (AS) ... 127
6.3.3 Formal implementation relation between the syntaxes ... 128
6.3.4 Comparison with the use of a single syntax ... 129
6.4 Semantics ... 129
6.5 Typing ... 131
6.5.1 Typing judgment ... 131
6.5.2 Type preservation ... 132
6.6 Related works ... 134
6.6.1 Formalizations of SSA ... 134
6.6.2 Representation of variables ... 135
6.6.3 Modular compilers ... 135
7 Passes on the parameterized representation 136
7.1 A simplification pass ... 136
7.2 Inter-procedural Dead Code Elimination ... 139
7.2.1 Removable operations and variables ... 139
7.2.2 Analysis ... 140
7.2.3 Erasure transformation ... 141
7.3 Block-sinking ... 144
7.3.1 Motivation and effects ... 145
7.3.2 Syntactic specification ... 147
7.3.3 Implementation ... 151
7.4 Type-checker ... 158
III Propagation of information 159
8 Definition of Frag IR 160
8.1 Instantiation of the parameterized IR ... 160
8.1.1 Types and values ... 160
8.1.2 Domain ... 166
8.1.3 Operators ... 167
8.2 Side effects of the operators ... 174
8.3 Instantiation of the generic passes ... 175
8.3.1 Type-checker ... 176
8.3.2 DCE ... 177
8.4 Examples ... 177
8.4.1 Writing programs in Frag IR ... 178
8.4.2 Example: arrays ... 178
8.4.3 Example: singly linked lists ... 186

9 Compilation to a lower-level language 190

9.1 Flat IR ... 191
9.1.1 Flat memory ... 191
9.1.2 Domain ... 191
9.1.3 Operators ... 192
9.2 A pass to remove optional values ... 194
9.3 Translation from Frag IR to Flat IR ... 197
9.3.1 Linearity condition ... 197
9.3.2 Transformation of the program ... 199
9.4 Erasure and simplification ... 202
9.5 Verified interpreter ... 203
9.6 Correctness of the pipeline ... 204

10 Using Vélus as a front-end 206

10.1 Vélus and its Clight back-end ... 206
10.1.1 Lustre: a synchronous source language ... 207
10.1.2 The Obc intermediate representation ... 208
10.1.3 The existing Clight back-end ... 210
10.2 Motivations ... 211
10.3 Translation of Obc to Frag IR ... 212
10.3.1 Types and signatures ... 213
10.3.2 Generalized variables ... 214
10.3.3 Expressions ... 215
10.3.4 Statements ... 215
10.3.5 Methods ... 220
10.3.6 Main loop ... 220
10.4 Correctness of the translation ... 221
10.4.1 Alternative monadic semantics for Obc ... 221
10.4.2 Specifications of the translation functions ... 224
10.4.3 Correctness of the main loop ... 229

Conclusion 235

References 239

Résumé en français

– This thesis is written in English and starts on page 18. This part summarizes its content in French.

Un compilateur est un logiciel qui traduit des programmes écrits dans des langages haut niveau vers des représentations bas niveau. Durant ce processus, les compilateurs peuvent effectuer des transformations destinées à optimiser le programme généré. Cependant, ces transformations doivent être correctes, c'est-à-dire, doivent préserver les propriétés désirées du programme. Ces propriétés dépendent en général de l'exécution du programme, on parle alors de propriétés dynamiques. Elles incluent notamment l'absence d'erreurs ainsi que les sorties du programme. Pour ces propriétés dynamiques, les transformations doivent être valides pour toutes les exécutions du programme. Vérifier une transformation peut alors nécessiter des *informations sémantiques*, valides pour toute exécution du programme.

On s'intéresse en particulier à des programmes utilisant la mémoire temporaire de la machine. Les accès à cette mémoire sont indirects. En effet, plutôt que de fixer l'emplacement des données dans la mémoire, les programmes manipulent des pointeurs. Ces pointeurs peuvent être stockés dans des variables syntaxiques et transmis entre composantes du programme. Leurs valeurs sont dynamiques et décrivent l'emplacement des données sur lequel le programme doit opérer. Il est possible que, lors de l'exécution du programme, plusieurs variables syntaxiques contiennent des pointeurs vers le même emplacement de la mémoire. On parle alors d'*aliasing*. Cette situation invalide certaines transformations, en introduisant des dépendances entre des opérations manipulant des variables syntaxiquement distinctes. Par exemple, la figure 1 décrit une transformation dans la syntaxe du langage C, qui peut sembler à première vue légitime.

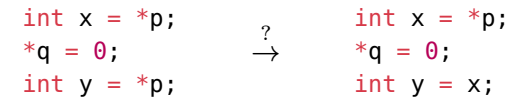


Figure 1: Une transformation sur les accès mémoire

Dans cet exemple, le programme source lit deux fois la valeur stockée dans la mémoire à l'emplacement décrit par le pointeur `p` (noté `*p`), et stocke les résultats dans les variables `x` et `y`. Ces deux lectures sont séparées par une écriture, à l'emplacement de la mémoire `*q` spécifié par `q`. Les variables `p` et `q` étant syntaxiquement distinctes, en l'absence d'aliasing, l'écriture via `q` ne peut pas changer la valeur stockée en `*p`. La seconde lecture produit alors le même résultat que la première. Il serait donc désirable d'optimiser ce programme en réutilisant le résultat `x` de la première lecture, afin d'éviter un accès à la mémoire. Cependant, cette transformation est invalide pour une exécution avec aliasing, où les variables `p` et `q` contiennent le même pointeur. En effet, dans ce cas, `*p` et `*q` désignent le même emplacement, donc la seconde lecture de `*p` renvoie la valeur `0` écrite via `q`, a priori distincte de la première lecture. Il n'est donc pas correct d'effectuer cette transformation durant la compilation, à moins d'avoir l'information sémantique que `p` et `q` n'aliasent jamais.

Pour déterminer la validité de transformations potentielles, les compilateurs ont donc besoin d'informations sémantiques sur les programmes. Il existe plusieurs sources d'informations :

- Des analyses peuvent inférer automatiquement des invariants du programme [89]. Par exemple, selon le contexte définissant `*p` et `*q`, une analyse d'alias pourrait permettre la transformation décrite en figure 1. Cependant, ces analyses peuvent être trop imprécises, ce qui empêche d'effectuer des transformations pourtant valides. En particulier, lors d'une compilation séparée d'une partie d'un programme (par exemple d'une fonction), les analyses n'ont accès qu'à des invariants restreints sur les interactions avec les autres composantes. Elles doivent donc sur-approximer les comportements des autres composantes (par exemple les résultats des fonctions appelées) ainsi que le contexte dans lequel la composante compilée s'exécute (par exemple les valeurs données en arguments).

- Une alternative est d'utiliser des informations déjà connues sur le programme source. Certains langages fournissent des annotations permettant de transmettre des informations au compilateur. Par exemple en C, si `p` et `q` sont déclarés avec le qualificatif `restrict`, alors le compilateur peut supposer qu'il n'y a pas d'alias et effectuer la transformation de la figure 1. Certains langages possèdent de plus des conventions restreignant les opérations pouvant être effectuées par les programmes, comme les règles de « strict aliasing » en C. Ces conventions fournissent des informations au compilateur, sans nécessiter d'annotations supplémentaires. Qu'elles soient explicites ou implicites, ces informations sur la source sont formalisées en ajoutant des comportements indéfinis (*undefined behaviors* [64]) à la sémantique du langage, qui signalent dynamiquement les violations des propriétés affirmées sur le programme source. Le compilateur n'a aucune garantie à fournir sur le comportement du programme cible à partir du moment où la source a un comportement indéfini. Formellement, cela lui permet de supposer la validité des informations encodées dans la source. L'utilisation de ces informations augmente cependant le risque de générer un code bas niveau incorrect. En particulier, une différence d'interprétation du langage source entre le producteur du programme et le compilateur peut causer des comportements indéfinis inattendus.
  Les vérifications automatiques sur le programme source sont une source plus fiable d'informations. Ainsi, certains langages sont équipés de systèmes de types statiques. La cohérence des types est assurée automatiquement par un vérificateur de types au début de la compilation. Ces types fournissent notamment des informations sur les valeurs que peuvent prendre les variables. Un cas particulièrement intéressant est le « borrow checker » du langage Rust, qui garantit des propriétés de non-aliasing [50].
- Les compilateurs effectuent des traductions afin d'encoder des abstractions haut niveau avec les primitives disponibles dans le langage cible. Le code généré par ces traductions peut vérifier des propriétés structurelles. Par exemple, lors de la compilation d'un langage fonctionnel comme OCaml, le compilateur implémente les fonctions de première classe à l'aide de clôtures. Bien que ces clôtures soient stockées en mémoire, les accès mémoires du programme source n'interfèrent pas avec cette implémentation. Ainsi, la transformation de la figure 1 serait valide si `p` pointe vers les données d'une clôture alors que l'écriture de `*q` correspond à un accès du programme source.

En pratique, les compilateurs sont implémentés comme des *pipelines* appliquant successivement des passes, qui transforment graduellement le programme source jusqu'au langage cible. Durant ce processus, les programmes sont décrits syntaxiquement dans plusieurs langages, incluant les langages source et cibles ainsi que des *représentations intermédiaires*. Ces représentations intermédiaires facilitent les transformations à divers niveaux d'abstraction. Comme les langages source et cibles, elles doivent décrire précisément le comportement du programme. Il est aussi utile qu'elles puissent encoder les informations connues sur le programme, afin de permettre leur propagation dans le pipeline. Cette propagation est même nécessaire pour utiliser les informations connues sur le programme source ou sur le code généré par une traduction. En effet, ces informations ne sont fournies que sur un programme particulier du pipeline, mais peuvent être utiles sur les programmes obtenus après des transformations ultérieures.

Le sujet de cette thèse est cette propagation d'information. On s'intéresse particulièrement aux informations d'alias. De plus, on considère le cas où le programme source comme le compilateur sont vérifiés formellement. C'est-à-dire, on suppose l'existence d'une preuve informatisée de la correction du programme source, et on souhaite avoir une preuve que le compilateur préserve cette correction jusqu'au langage cible. Cette situation ne concerne qu'une fraction des cas pratiques, mais est intéressante pour plusieurs raisons. D'une part, la combinaison de la correction du compilateur avec celle du programme source implique formellement la correction du programme cible généré. D'autre part, on argumente que la propagation d'information est particulièrement adaptée à cette situation. En effet :

- Propager des informations évite l'utilisation d'analyses. Dans le cas d'un compilateur vérifié, la correction de ces analyses doit souvent être prouvée.
- Vérifier des propriétés sur le programme source demande la formalisation d'invariants. Ces invariants fournissent en particulier des informations sur les interactions du programme (ou de la composante considérée) avec son contexte. Notamment, prouver des propriétés non triviales nécessite de contrôler les effets de tous les accès mémoires. On peut donc supposer des informations plus précises que celles qui pourraient être obtenues via des analyses. Dans le cas d'une preuve informatisée du programme source, une représentation formelle de ces invariants est déjà disponible et pourrait être donnée en entrée au compilateur.
- De façon similaire, les propriétés structurelles du code généré par les passes de traduction sont vraisemblablement déjà nécessaires pour prouver leur correction.
- Prouver à la fois la correction des programmes et la validité des informations encodées empêche l'introduction de bogues lors de l'utilisation de ces dernières.

Enfin, la présence d'informations influence la complexité des preuves sur les programmes sources et le compilateur, et ce, dans plusieurs directions.

- D'une part, il est nécessaire de prouver que le programme source et la sortie des passes satisfont les propriétés affirmées. Cela génère a priori des obligations de preuve supplémentaires.
- Les passes peuvent supposer la validité des informations encodées dans le programme reçu en entrée. La justification formelle de la correction des transformations les utilisant dépendra alors de la manière d'associer une sémantique aux informations.
- La sémantique des programmes doit tenir compte des informations encodées. Si c'est a priori une source de complexité, on montrera que les informations peuvent en fait simplifier certains aspects de la sémantique.

Cette thèse comprend deux axes majeurs. D'une part, on définit une nouvelle représentation intermédiaire, appelée Frag IR, et on l'utilise pour propager des informations dans un pipeline de compilation vérifié. D'autre part, on cherche à augmenter la modularité de l'implémentation et la preuve de ce pipeline. L'élément central de ce deuxième axe est la définition d'une bibliothèque `SMonad`, qui facilite l'utilisation de sémantiques dénotationnelles. On se sert de cette bibliothèque dans la formalisation de Frag IR, mais elle peut être réutilisée pour d'autres développements. `SMonad`, Frag IR ainsi que le pipeline l'entourant sont implémentés et prouvés corrects au sein de l'assistant de preuve Rocq.

Dans la suite, on détaille ces deux axes, avant de résumer le contenu des chapitres rédigés en anglais.

Frag IR : une représentation intermédiaire pour propager des informations sémantiques

Encodage des informations Frag IR utilise deux moyens pour encoder des informations :

- Des propositions avec des types dépendants indiquent des relations entre les variables du programme. Ces propositions peuvent notamment décrire des pré- et post-conditions d'opérations ou de fonctions. Par exemple, dans la figure 2, l'opérateur `Frag.read` a une post-condition `POST : *c = r` et la fonction `array_extract_begin` une pré-condition `LT : i < len`. Cette dernière est définie par un programme et utilisée pour accéder à l'élément `i` d'un tableau de longueur `len`. La pré-condition `i < len` est standard pour un tel accès. L'originalité de Frag IR est de représenter ces pré-conditions dans la syntaxe du programme, plutôt qu'uniquement via des comportements indéfinis dans la sémantique. Cette approche permet d'expliciter les dépendances des informations, alors que les comportements indéfinis dépendent de l'ensemble de l'exécution les précédant (comme les branchements) du fait de leur nature sémantique. En second recours, Frag IR permet la récupération d'informations à partir de comportements indéfinis via une instruction `assert`.

```
Frag.read{t}(p: ptr, ghost c: cell{t} p)
  : (r: t, ghost POST: *c = r)
Frag.write{t}(p: ptr, ghost c: cell{t} p, v: t)
  : (ghost c': cell{t} p, ghost POST: *c' = v)
```

a: Sous-ensemble des opérateurs primitifs pour accéder aux cellules mémoires

```
fn array_extract_begin(
     p : ptr, ghost len : i64, i : i64,
     ghost a0 : mem{i64_array} p len,
     ghost LT : i < len)
  : (p1 : ptr,
     ghost c : cell{i64} p1, ghost a1 : mem{i64_array_without} p len p1)
{
  let SIZE_INT : i64, ghost EQ_SIZE_INT = const{8}();
  let ofs      : i64, ghost EQ_ofs      = mul{i64}(i, SIZE_INT);
  let p1       : ptr, ghost EQ_p1       = add{ptr}(p, ofs);
  let ghost c : cell{i64} p1, ghost a1 : mem{i64_array_without} p len p1 = lemma {
      (p' : ptr, len' : i64, i' : i64, SIZE_INT' : i64, ofs' : i64, p1' : ptr,
       a0' : mem{i64_array} p' len',
       LT'           : i' < len',
       EQ_SIZE_INT' : SIZE_INT' = const{8},
       EQ_ofs'       : ofs'      = mul{i64} i' SIZE_INT',
       EQ_p1'        : p1'       = add{ptr} p' ofs')
    : (c' : cell{i64} p1', a1' : mem{i64_array_without} p' len' p1')
      consume (a0')
    }(p, len, i, SIZE_INT, ofs, p1, a0, LT, EQ_SIZE_INT, EQ_ofs, EQ_p1);
  return (p1, c, a1)
}
```

b: Une fonction définie par un programme pour accéder à un élément d'un tableau

Figure 2: Des composantes de programmes Frag IR utilisant la mémoire

- Des fragments mémoires représentent des zones disjointes de la mémoire, et dont les accès n'interfèrent pas. La figure 2b implique trois fragments. Avant la fonction, un unique fragment `a0 : mem{i64_array} p len` décrit une région commençant à l'adresse `p` et contenant un tableau de `len` entiers. La fonction `array_extract_begin` n'accède pas elle-même à l'élément `i`. Elle calcule seulement un pointeur `p1` vers cet élément et le sépare du reste du tableau. Ainsi, elle renvoie deux fragments, à savoir une cellule `c : cell{i64} p1` contenant l'élément `i`, et un fragment `a1 : mem{i64_array_without} p len p1` contenant les autres éléments.
  Frag IR fournit des opérateurs primitifs réalisant les équivalents des accès mémoires standards. La figure 2a décrit la signature des opérateurs de lecture et d'écriture, pour un type scalaire `t` arbitraire (e.g., `i64` pour les entiers 64 bits). Comme leurs versions standards, ces opérateurs prennent en argument un pointeur `p : ptr` décrivant l'adresse où l'accès doit être effectué. Cependant, ils prennent aussi en argument un fragment `c : cell{t} p` représentant syntaxiquement l'emplacement accédé. De plus, les fragments sont immuables, et `Frag.write` renvoie donc un fragment modifié.

Représentation syntaxique Ces deux représentations d'information sont encodées via les variables du programme. Les propositions et les fragments mémoires sont ainsi considérés comme des variables standards (mais avec des types dépendants) dans la syntaxe, le jugement de type ou encore la sémantique. Éviter l'introduction de nouveaux concepts simplifie aussi les passes. Ces variables utilisées pour encoder des informations ne sont cependant pas utiles pour exécuter le programme. Dans notre pipeline, elles sont donc supprimées avant la génération du programme cible. L'annotation `ghost` prépare cette suppression.

```
let (x : i64,           _) = Frag.read{i64}(p, c0);
let (c1' : cell{i64} q, _) = Frag.write{i64}(q, c1, 0);
let (y : i64,           _) = Frag.read{i64}(p, c0) → x;
                                              ↓
```

Figure 3: Transformation équivalente à la figure 1 en Frag IR

Dépendances explicites Un objectif commun aux propositions et aux fragments est de représenter syntaxiquement les dépendances sémantiques entre les opérations du programme. Cela permet aux transformations de détecter syntaxiquement l'indépendance d'instructions. Par exemple, la figure 3 décrit l'équivalent de la transformation de la figure 1. En Frag IR, cette transformation est toujours valide (on suppose l'absence de masquage de noms de variables, ici `c0` et `c1'` sont distincts). En fait, l'immutabilité des fragments mémoires implique que cette transformation est sémantiquement valide quelles que soient les instructions séparant les deux lectures.

Lemmes Une opération `lemma` permet d'effectuer des transformations sur les propositions et les fragments. Pour les propositions, elle permet de dériver des conséquences logiques des relations déjà connues entre les variables. Par exemple, un `lemma` peut dériver une nouvelle proposition `x = z` à partir de deux propositions `x = y` et `y = z` existantes. Pour les fragments, `lemma` permet de séparer et de regrouper des fragments disjoints. Par exemple, l'opération `lemma` à la line 11 de la figure 2b sépare en deux le fragment contenant le tableau. Dans tous les cas, les lemmes sont décrits par une signature close donnée en paramètre (entre accolades `{…}`). Cette signature décrit les types des arguments et résultats. L'opération doit être appliquée à des arguments ayant des types correspondants, et renvoie des propositions ou fragments avec les types indiqués dans la signature. Dans la figure 2b, on différencie explicitement les variables du contexte des variables formelles du lemme en ajoutant un symbole prime (`'`) aux secondes. Afin d'assurer la représentation des dépendances, les lemmes doivent prendre en arguments toutes les hypothèses permettant de produire leur résultat.
Les lemmes sont sujets à une condition de correction. Cette condition requiert que les propositions renvoyées soient des conséquences logiques des arguments, et que les fragments en sortie puissent être obtenus en réorganisant les fragments en entrée. Les lemmes ayant une signature close, cette condition ne dépend pas de leur position dans le programme. Notre compilateur suppose que le programme source ne contient que des lemmes corrects, et maintient cet invariant tout au long du pipeline. C'est donc au producteur du programme source de garantir la validité des lemmes.
Afin de représenter des relations ou régions mémoires complexes, le programme source peut contenir des propositions `prop` ou des fragments mémoire `mem` abstraits. La figure 2b utilise ainsi des fragments abstraits `mem{i64_array}` et `mem{i64_array_without}` pour représenter des tableaux (qui n'ont pas de types ou d'opérations prédéfinis dans Frag IR). Le producteur du programme source doit choisir une interprétation de ces types, et prouver la correction des lemmes vis-à-vis de cette interprétation. Le compilateur n'a pas accès à cette interprétation, mais doit préserver les dépendances matérialisées par les variables ayant des types abstraits.

Absence de mémoire globale Dans la sémantique de Frag IR, ce sont les valeurs des fragments qui propagent les données stockées en mémoire. Par exemple, une variable de type `cell{i64} p` contient directement l'entier stocké à l'adresse `p`. Les opérations sur les cellules (figure 2a) ont une sémantique fonctionnelle : `Frag.read` renvoie le contenu de la cellule passée en argument et `Frag.write` une cellule contenant la valeur écrite. Les sémantiques de ces deux opérations ignorent les autres arguments (le pointeur, ainsi que l'ancienne cellule pour `Frag.write`). Ces arguments ne sont utilisés que plus tard dans le pipeline, lors du remplacement des opérateurs par leurs versions standards.
Frag IR n'a ainsi aucun état mémoire global. Cela évite des dépendances implicites entre les accès mémoires. En particulier, les allocations mémoires sont indépendantes des zones de la mémoire déjà allouées, et peuvent donc renvoyer des pointeurs déjà alloués. Ce comportement non standard facilite les transformations des allocations. La correction de certains programmes repose cependant

sur l'absence d'intersection entre les domaines de différentes unités d'allocation. Pour ces cas, Frag IR fournit un opérateur `Frag.joinable`, qui fournit au programme une proposition assurant l'absence d'intersection entre les domaines de fragments donnés en arguments. Là encore, cet opérateur signale syntaxiquement des dépendances qui seraient sinon implicites.

Linéarité et traduction On implémente une procédure traduisant les programmes Frag IR en une représentation Flat IR utilisant une mémoire globale. Flat IR est aussi une nouvelle représentation, définie spécifiquement pour servir de cible à la traduction de Frag IR. Contrairement à Frag IR, elle est similaire à des représentations standards existantes, et pourrait servir d'intermédiaire entre Frag IR et ces représentations dans des extensions futures. Syntaxiquement, cette traduction remplace les accès aux cellules mémoires par les opérations correspondantes sur la mémoire globale. Les propositions, fragments ainsi que les opérations `lemma` et `Frag.joinable` sont supprimées du programme.
Cependant, cette transformation n'est sémantiquement correcte que si le programme utilise les fragments de manière linéaire. Cette *condition de linéarité* interdit notamment l'utilisation des fragments ayant déjà été donnés en arguments à `Frag.write`, `lemma` ou à une fonction. Cette condition est un invariant informel de notre pipeline. Autrement dit, nos passes doivent le préserver, mais on ne prouve pas formellement que c'est le cas. À la place, la passe de traduction vérifie défensivement que le programme qu'elle reçoit en entrée utilise les fragments linéairement. Dans le cas contraire, elle signale une erreur de compilation.
La condition de linéarité sur Frag IR ne concerne que les fragments. Les autres variables peuvent être utilisées librement. En particulier, grâce à l'utilisation de variables distinctes pour les pointeurs et les fragments, les premiers peuvent être aliasés.

Connexion avec Vélus Dans la direction opposée, on prolonge notre pipeline en le connectant au compilateur vérifié Vélus[1]. On implémente pour cela une passe de traduction entre sa représentation Obc et Frag IR. Outre la possibilité de générer des programmes pour notre pipeline, cette connexion peut être comparée à la traduction standard de Vélus vers Clight. En particulier, on vérifie la capacité de Frag IR à encoder des informations. En effet, les programmes Obc opèrent sur une mémoire globale, mais structurée sous forme d'arbre, qui assure par construction des propriétés de séparation. Ces propriétés sont encodées dans Frag IR à l'aide de fragments, alors qu'elles nécessiteraient une analyse inter-procédurale pour être retrouvées en Clight. Un autre critère de comparaison est la simplicité de la traduction vers Frag IR, à la fois pour l'implémentation et la preuve. L'implémentation de la traduction vers Frag IR est ainsi plus volumineuse que celle vers Clight. Cela s'explique à la fois par l'émission des représentations des informations et par des différences de structures entre Frag IR et Obc (notamment l'utilisation d'une forme SSA fonctionnelle pour Frag IR). En revanche, les deux preuves ont des tailles similaires. En particulier, on évite une preuve de logique de séparation spécifique à la traduction d'Obc. Cette preuve est en fait remplacée par la validation défensive de la linéarité lors de la traduction vers Flat IR.

`SMonad` : raisonner sur les programmes à l'aide de sémantiques dénotationnelles

Axiomatisation La bibliothèque `SMonad` est centrée sur une axiomatisation de domaines[2] pouvant être utilisés pour définir des sémantiques dénotationnelles. Un tel domaine est formellement une famille de types `S`. Un élément du type `S E A` est une sémantique interagissant avec son contexte en émettant des évènements de la famille `E` et en terminant avec un résultat de type `A`. Notre axiomatisation fixe la signature d'opérateurs permettant de construire des sémantiques. On suppose notamment

[1] Vélus compile des programmes Lustre (pour systèmes embarqués critiques) en C. Voir https://velus.inria.fr/.

[2] La notion de « domaine » utilisée dans cette thèse diffère de celle des ensembles partiellement ordonnés utilisés en théorie des domaines.

que `S` est équipé d'opérateurs `ret` et `bind` faisant de `S E` une monade (pour toute famille d'évènements `E`). Deux opérateurs d'itération, `iter` et `mrec`, permettent de définir la sémantique de boucles ou de systèmes de fonctions mutuellement récursives. Notre axiomatisation est paramétrée par une relation de raffinement `sem_ref` entre les sémantiques. Cette relation est hétérogène, autrement dit, elle permet de comparer deux sémantiques u_0 : `S E`$_0$ `A`$_0$ et u_1 : `S E`$_1$ `A`$_1$ interagissant avec leurs contextes via des types d'évènements (`E`$_0$ et `E`$_1$) et de résultats (`A`$_0$ et `A`$_1$) différents. Pour cela, `sem_ref` prend en argument des relations sur les évènements et sur les résultats. On capture ensuite le comportement attendu des opérateurs construisant les sémantiques. Pour cela, on axiomatise leur interaction avec `sem_ref` et une notion d'équivalence dérivée.
L'axiomatisation de `SMonad` est inspirée de la théorie existante autour des « interaction trees » [102]. On généralise cependant cette théorie dans plusieurs directions :

- Notre axiomatisation permet l'utilisation d'autres modèles.
- On généralise la signature du raffinement pour introduire une notion de relation sur les évènements.
- On axiomatise les comportements indéfinis ainsi que le non-déterminisme.

Définition et raisonnement sur les sémantiques Un domaine `S` équipé des opérateurs de `SMonad` peut être utilisé pour définir des sémantiques dénotationnelles. Dans cette approche, on associe une sémantique (c'est-à-dire un élément de `S`) à chaque catégorie syntaxique du programme. Pour cela, on suit la structure syntaxique du programme, et on définit la sémantique d'un élément en combinant les sémantiques de ses composantes grâce aux opérateurs axiomatisés par `SMonad`. Par exemple, la figure 4a est extraite de la définition de la sémantique de Frag IR. Elle dépend de notions qui ne sont pas introduites dans ce résumé. On peut cependant observer que la sémantique des instructions $[\![.]\!]_{\text{instr}}$ et des programmes $[\![.]\!]_{\text{prog}}$ sont toutes les deux des fonctions dont le co-domaine est `FragIR.S`. Il s'agit du modèle de `SMonad` (c'est-à-dire, l'instance du paramètre `S`) utilisé pour Frag IR. De plus, ces fonctions sont définies en utilisant les opérateurs de `SMonad` (implémentés par le domaine `FragIR.S`), et en suivant la syntaxe des programmes. En particulier, la sémantique $[\![.]\!]_{\text{instr}}$ est définie par récursion sur la syntaxe. Ainsi, la sémantique d'une séquence `let xs = lhs; rhs` est définie en combinant les sémantiques de ses deux sous-instructions (`lhs` et `rhs`) à l'aide de l'opérateur `bind`. De même, la sémantique $[\![.]\!]_{\text{prog}}$ du programme est dérivée de la sémantique $[\![.]\!]_{\text{func}}$ de ses fonctions grâce à `mrec`.
La structure de ces sémantiques dénotationnelles permet de prouver des transformations par induction sur la syntaxe du programme source. Le caractère hétérogène de `sem_ref` permet de plus une forme de raisonnement relationnel, qui tolère des changements dans les interactions avec le contexte entre la source et la cible. Par exemple, la figure 4b est extraite d'un invariant utilisé pour prouver la correction d'une passe d'effacement. Ce raffinement décrit la sémantique d'une instruction cible u_1 obtenue en transformant une instruction source u_0. Informellement, cet invariant exprime qu'étant donné des valuations ρ_0 et ρ_1 correspondantes de leurs variables, la sémantique de u_1 raffine celle de u_0. Cependant, certains résultats pouvant être effacés, donc une relation `Erasure.res_match` tient compte du fait que u_1 ne renvoie qu'un sous-ensemble des résultats de u_0. De plus, la transformation étant inter-procédurale, des arguments et résultats de fonctions appelées par les instructions peuvent aussi être effacés. Ces appels de fonctions sont représentés par des évènements internes, et c'est une relation `Erasure.fcall_match` qui tient compte des différences entre les appels sources et cibles.

Modèle basé sur des systèmes de transitions `SMonad` étant inspirée des interaction trees, ceux-ci sont un modèle de notre axiomatisation. On fournit un autre modèle, basé sur des systèmes de transitions (LTSs). Contrairement aux interaction trees, ce modèle supporte nativement les comportements indéfinis et le non-déterminisme. Les instances des opérateurs de `SMonad` permettent la construction de systèmes complexes en combinant des LTSs. On instancie `sem_ref`, par une notion de simulation avec piétinement, de la cible par la source. On prouve que ce raffinement satisfait tous les axiomes de `SMonad`.

$$[\![\texttt{u} : \texttt{Instr.s E r}]\!]_{\text{instr}}\ (\rho : \texttt{E.var} \to \texttt{val}) : \texttt{FragIR.S}\ (\texttt{Fct.Call.ev E.fct})\ (\texttt{Res.t E.knt})$$
$$[\![\texttt{pr} : \texttt{Prog.s a}, \texttt{f}_0 : \texttt{pr.F a}_0]\!]_{\text{prog}}\ (\texttt{varg}_0 : \texttt{val}^{\texttt{a}_0.\texttt{arg}}) : \texttt{FragIR.S}\ \varnothing\ \texttt{val}^{\texttt{a}_0.\texttt{res}}$$
$$[\![\texttt{let xs = lhs; rhs}]\!]_{\text{instr}}\ \rho := \texttt{bind}\ ([\![\texttt{lhs}]\!]_{\text{instr}}\ \rho)\ \left(\lambda \texttt{r}.\ \begin{cases} [\![\texttt{rhs}]\!]_{\text{instr}}\ [\rho, \texttt{xs} \mapsto \texttt{vs}] & \text{if } \texttt{r} = \texttt{Res.Ret vs} \\ \texttt{ret}\ (\texttt{Res.Knt c}) & \text{if } \texttt{r} = \texttt{Res.Knt c} \end{cases}\right)$$
$$[\![\texttt{pr}, \texttt{f}_0]\!]_{\text{prog}}\ \texttt{varg}_0 := \texttt{mrec}\ \left(\lambda(\texttt{Fct.Call.mk f varg}).\ [\![\texttt{pr.func f}]\!]_{\text{func}}\ \texttt{varg}\right)\ (\texttt{Fct.Call.mk}\ \texttt{f}_0\ \texttt{varg}_0)$$

a: Extrait de la définition de la sémantique dénotationnelle (figure 6.6)

$$\begin{aligned}&\texttt{Erasure.venv_match eE}\ \rho_0\ \rho_1 \Rightarrow \\ &\texttt{sem_ref}\ (\texttt{Re_ty}\ \Gamma_0; \texttt{Erasure.fcall_match eE})\ (\texttt{Erasure.res_match eE er}) \\ &\quad \left([\![\texttt{u}_0]\!]_{\text{instr}}\ \rho_0\right)\ \left([\![\texttt{u}_1]\!]_{\text{instr}}\ \rho_1\right)\end{aligned}$$

b: Un raffinement utilisé dans la preuve d'une passe d'effacement (lemma 7.2)

Figure 4: Extraits d'utilisation de `SMonad` pour la formalisation de Frag IR

On fournit de plus des théories spécifiques aux LTSs. On définit les comportements des LTSs et on prouve la correction de la simulation par rapport à l'inclusion des comportements. Cela permet de dériver de `sem_ref` des propriétés de correction similaires à celle donnée par CompCert [69, définition 3]. On fournit également une théorie permettant de prouver l'équivalence entre des sémantiques dénotationnelles (dans le domaine LTS) et des sémantiques opérationnelles à petit pas. Comme application, on implémente une bibliothèque permettant la définition d'interpréteurs vérifiés.

Représentation paramétrée en SSA fonctionnel `SMonad` fournit des éléments pouvant être réutilisés pour plusieurs langages : le domaine LTS ainsi que des propriétés dérivées de l'axiomatisation. Avec ce même objectif de modularité, on définit une représentation syntaxique paramétrée des programmes. Les représentations introduites dans notre pipeline, Frag IR et Flat IR, sont toutes deux des instances de cette représentation. Cela permet de factoriser la définition de leur syntaxe, mais aussi de leur méta-théorie (sémantique et jugement de type).
Notre représentation est paramétrée par l'ensemble des types et des opérateurs, mais définit la structure des instructions et des programmes. Grâce à `SMonad`, sa sémantique est paramétrée par le domaine utilisé ainsi que par la sémantique des opérateurs. En particulier, les instances peuvent avoir des effets différents. Ainsi, Flat IR a un état global, mais pas Frag IR.
Notre représentation paramétrée utilise une structure de SSA fonctionnel [7]. La propriété SSA évite des invalidations des types dépendants, utilisés par Frag IR. De plus, les programmes en SSA fonctionnel utilisent des instructions inductives. Celles-ci possèdent une structure syntaxique plus riche que des représentations basées sur des graphes de contrôle (CFG). Cette structure est reflétée par leur sémantique dénotationnelle. Nos instructions inductives encodent des graphes de flot de contrôle arbitraires grâce à des continuations, introduites localement par des instructions `krec`. Comme illustré en figure 5, ces continuations permettent notamment la définition de boucles.

Passes génériques On définit plusieurs passes sur la représentation paramétrée :

- Une simplification linéarisant les séquences d'instructions et éliminant les copies de variables qu'elles contiennent.
- Une élimination de code mort inter-procédurale. On implémente pour cela une analyse de « liveness », qui détecte automatiquement les variables et opérations inutilisées. La structure SSA permet d'accroitre l'efficacité de cette analyse.
- Une transformation de « block-sinking », qui déplace la définition des continuations au sein des instructions. Elle permet entre autres de refléter syntaxiquement la relation de dominance entre les points de programme.
- Un vérificateur de type, qui permet de garantir formellement que les programmes sont bien typés au sein du pipeline de compilation.

```
for (i = 0; i < 2; ++i) {
  if (i == 0) {
    x = 2 * i;
    f(x);
  }
  g(2 * i);
}
```

a: En C, sans SSA

```
let c0 = 0; let c1 = 1; let c2 = 2;
krec {
  loop(i):
    let b0 = u_lt(i, c2);
    if b0 _ then {
      let b1 = u_eq(i, c0);
      if b1 _ then {
        let x = mul(c2, i); fcall f(x)
      } else return ();
      let y  = mul(c2, i); fcall g(y);
      let i1 = add(i, c1); kcall loop(i1)
    } else return ()
}(c0)
```

b: Dans nos représentations avec SSA

Figure 5: Encodage d'une boucle dans nos représentations SSA fonctionnelles

Une grande partie de la correction de ces passes est prouvée de façon générique, indépendamment de l'instance de la représentation paramétrée. On instancie ces passes sur Frag IR et Flat IR. Pour l'élimination de code mort et le vérificateur de type, il est pour cela nécessaire de définir des paramètres des passes (par exemple, la signature des opérateurs), et de prouver des propriétés de correction sur ces définitions. Dans tous les cas, la preuve paramétrée permet de dériver des garanties au niveau du programme à partir de ces éventuelles preuves locales.

Résumé des chapitres

La partie I présente à haut niveau notre approche.

Chapitre 1 - Introduction

On donne du contexte sur la compilation, son objectif et ses interactions avec les méthodes formelles. On décrit ensuite plus spécifiquement le problème de la propagation d'informations. On liste enfin nos contributions et les conventions utilisées dans le manuscrit.

Chapitre 2 - État de l'art

On décrit les approches et travaux existants liés à la propagation d'informations. On rappelle en particulier les utilisations classiques des comportements indéfinis et leurs limitations. On liste différentes formalisations de modèles mémoires, ainsi que différentes méthodes utilisées pour propager des informations d'alias, y compris dans des compilateurs non vérifiés. On mentionne certaines formalisations liées à la compilation vérifiée, ainsi que des approches utilisées pour vérifier des programmes utilisant la mémoire.

Chapitre 3 - Frag IR : une représentation intermédiaire pour propager des informations

On décrit informellement Frag IR. On détaille en particulier les deux mécanismes (types dépendants et fragments mémoires) utilisés pour propager les informations. On explique l'impact de ces annotations sur la sémantique et quelles transformations elles permettent. On donne également une description haut niveau du pipeline de compilation vers Flat IR.

On décrit ensuite les différentes composantes du développement, en suivant l'ordre de leurs dépendances. La partie II traite ainsi des aspects liés à la modularité, en lien avec les sémantiques dénotationnelles.

Chapitre 4 - `SMonad` : une bibliothèque pour définir des sémantiques dénotationnelles

On commence par motiver l'utilisation de sémantiques dénotationnelles, en les comparant sur un exemple aux approches opérationnelles. On décrit ensuite l'axiomatisation utilisée par `SMonad`, à savoir les opérateurs construisant des sémantiques et leur interaction avec une relation de raffinement hétérogène. On donne des exemples de raisonnements relationnels utilisant ces axiomes. On discute ensuite certaines propriétés et transformations impliquées par l'axiomatisation. On montre en particulier qu'elle permet de justifier des transformations de boucle et d'inlining de

fonctions. Cependant, certaines propriétés sont inconsistantes avec l'axiomatisation de `SMonad`. C'est notamment le cas du déplacement des choix non-déterministes en dessous des évènements. Ces propriétés ne peuvent donc ni être dérivées des axiomes, ni satisfaites par des modèles non triviaux. Ce second point empêche d'utiliser `SMonad` pour prouver certaines transformations. Les modèles de `SMonad` peuvent supporter des effets différents, et des transformateurs permettant d'ajouter a posteriori des effets à des domaines.

Chapitre 5 - Un modèle à base de systèmes de transition

Notre principal modèle de `SMonad` est basé sur des systèmes de transitions étiquetés (LTSs), comparés avec une notion de simulation. On formalise plusieurs théories spécifiques à ce modèle, comme la définition des comportements des systèmes. Il est possible de prouver l'équivalence de nos sémantiques dénotationnelles avec des sémantiques opérationnelles à petit pas. Notre modèle peut ainsi être connecté avec les définitions existantes de CompCert.

Chapitre 6 - Une représentation paramétrée en SSA fonctionnel

On rappelle les propriétés des représentations SSA ainsi que les spécificités de l'approche fonctionnelle. On décrit ensuite la syntaxe de notre représentation paramétrée et son implémentation en Rocq. On définit une sémantique dénotationnelle et un jugement de type sur cette représentation. On utilise `SMonad` pour paramétrer la sémantique par son domaine et par la sémantique des opérateurs.

Chapitre 7 - Passes sur la représentation paramétrée

On implémente quatre passes sur la représentation paramétrée : une simplification des séquences, une élimination de code mort, une transformation de block-sinking et un vérificateur de type. Certaines de ces passes prennent des paramètres spécifiques à l'instance de la représentation paramétrée. On décrit les hypothèses sur ces paramètres, ainsi que les résultats prouvés sur les résultats des passes.

La partie III est consacrée à notre pipeline vérifié propageant des informations.

Chapitre 8 - Définition de Frag IR

On obtient Frag IR comme une instance de la représentation paramétrée. Pour cela, on définit ses types et opérateurs ainsi qu'un domaine utilisé pour sa sémantique. Ce domaine est dérivé du modèle LTS. On revient plus formellement sur les propriétés de sa sémantique. On instancie les passes génériques sur Frag IR. On teste la représentation en écrivant des petits programmes directement dans Frag IR.

Chapitre 9 - Compilation vers un langage bas-niveau

On définit une procédure de compilation de Frag IR vers une représentation plus standard. Le langage cible de cette procédure, Flat IR, est une autre instance de la représentation paramétrée. Il étend Frag IR avec une mémoire concrète globale. Une première étape de ce pipeline supprime des « variables optionnelles », une notion introduite dans Frag IR pour représenter des variables non initialisées. La compilation de Frag IR vers Flat IR est effectuée par deux passes successives. Dans un premier temps, une passe de traduction introduit les accès à la mémoire globale correspondant à ceux effectués sur les fragments. Dans un second temps, une passe d'effacement supprime les annotations (propositions et fragments mémoires). Notre pipeline s'arrête actuellement à Flat IR, mais un interpréteur permet d'exécuter les programmes qu'il produit. On dérive la correction du pipeline en composant les preuves des différentes passes.

Chapitre 10 - Connexion avec Vélus

On connecte notre pipeline comme une sortie alternative pour le compilateur vérifié Vélus. Pour cela, on implémente une traduction de la représentation Obc de Vélus vers Frag IR. On rappelle les concepts clés de Vélus, et en particulier ceux impliqués dans Obc et sa traduction standard vers Clight. On décrit ensuite syntaxiquement notre traduction vers Frag IR, et présente les principaux invariants de sa preuve de correction. Notre traduction génère une boucle réactive similaire à

celle de Clight. En revanche, on donne un résultat de correction différent pour tenir compte de la présence de non-déterminisme dans Frag IR et Flat IR.

Part I
Background and motivations

1

Introduction

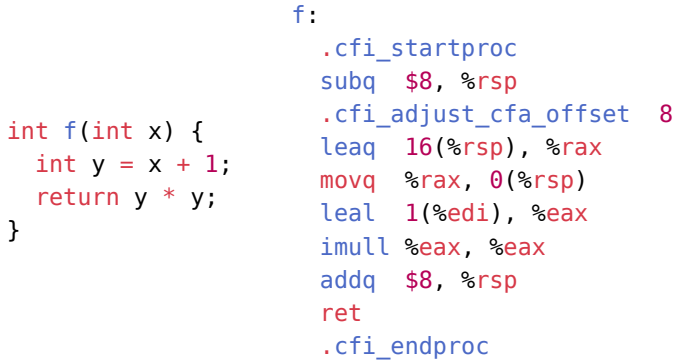
This thesis lies at the interface between compilation and formal methods. Section 1.1 gives a general background on compilation. Section 1.2 explains how it interacts with formal verification. Section 1.3 then focuses on the main topic of this thesis, namely the propagation of information in verified compilation pipelines. Section 1.4 lists our contributions and section 1.6 describes the structure of the following chapters.

1.1 Compilation

Computers are controlled by programs. A program is a precise description of the actions that must be performed by the machine. Processors execute programs written in specific formats, called *binaries*. In those formats, the behavior of a program is specified using elementary operations provided by the processor. Those operations perform interactions with the environment or update the state of the machine. Binary formats are designed to be easily executable by the machine. However, they are less adequate for writing programs. Programs are thus rarely written directly in binary format, or even in its textual assembly representation (that we identify with binary in this introduction). They are instead expressed in *programming languages*. There exist many different programming languages: C, Rust, OCaml, Python… An informal notion of level is associated with languages and concepts to express how much they differ from what is actually executed by the machine. Binary is the lowest level. There are several motivations for not writing programs directly in binary format:

Abstraction Programming languages abstract low-level details of the computer. For instance, to store temporary values at the binary level, programs only have a direct access to a bounded number of "registers" (e.g., `eax` in figure 1.1b). More data can be remembered using indirect accesses to the memory of the machine. Higher-level languages instead provide an unbounded number of "variables" (e.g., `y` in figure 1.1a), so that programmers do not have to manage which variables are available directly. Programming languages are also more concise and easier to understand for humans.

Independence with respect to the architecture There are multiple binary formats, depending on the architecture of the processor (e.g. x86 or riscV). The expected behavior of programs is often independent of the specificity of the processor. Programming languages offer a way of expressing those behaviors independently of the architecture.



```c
int f(int x) {
  int y = x + 1;
  return y * y;
}
```

a: Source C

```asm
f:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
  leal  1(%edi), %eax
  imull %eax, %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
```

b: Target x86–64 assembly

Figure 1.1: Fragments of the compilation of a program by CompCert[3]

Modularity Large programs are defined by composing several components. In particular, they use libraries: collections of program parts which are used by several programs. Different components can be interfaced at the binary level. For instance, function calls enable the execution of code defined in another component. However, this approach often has a cost on the efficiency of programs. Programming languages may offer other mechanisms to compose components, for instance traits in Rust or templates in C++.

Those higher-level program representations cannot be executed directly by processors. There are two ways of executing a program `p` written in a programming language:

By compilation This approach consists in generating a binary which describes the same behavior as the original program `p`. The binary is then executed directly by the processor.

By interpretation This approach relies on another program, called interpreter. The interpreter is executed by the processor. Its behavior is to read the program `p` and to perform the actions it specifies.

This thesis is about the first approach. The process of generating a program in a lower-level language corresponding to a program expressed in a higher-level language is called *compilation*. This process is performed automatically by programs called *compilers*. A compiler takes as input a program in its source language and outputs a program in its lower-level target language. They are often decomposed into several passes which gradually transform the program. Those passes operate on representations of the programs in several languages: the source and target languages and *Intermediate Representations* (IRs). There are two kinds of passes:

Translation passes Those passes replace abstractions by their implementations.

Optimization passes Those passes transform programs to improve their efficiency.

1.2 Formal verification

Computers are expected to satisfy specifications, which translate to properties on binary programs (section 1.2.1). When a binary is produced by compilation, ensuring those properties involves both the source program and the compiler (section 1.2.2). Formal methods are approaches to obtain strong guarantees on programs, including on compilers (section 1.2.3).

1.2.1 Properties of programs

Computers are used to perform various tasks: controlling physical devices, storing and transmitting data, running numerical simulations… For those goals to be achieved, the behavior of the computer must satisfy some properties. There are different kinds of properties, depending on the application. Those properties include:

[3]CompCert is an industrial-strength formally verified C compiler. See https://compcert.org/.

Absence of crashes Processors consider that some operations are errors under particular conditions. For instance, accesses to non-existing or restricted parts of the memory or some arithmetic operations such as division by zero. Processors provide mechanisms for software to react to those errors. In particular, most computers run a software called operating system (OS), which, among other things, limits the effects of errors of a program on other programs running concurrently. However, computers sometimes perform erroneous operations that were not anticipated. In those cases, mechanisms to recover from error states may be missing or unable to handle the errors. Those situations lead to abrupt and undesired termination of programs.

Functional correctness Computers output some results, by transmitting commands to actuators, writing data on external devices or on the network or by displaying information to users. Those outputs must satisfy some specifications, which often depend on the inputs received by the computer. Those specifications sometimes fully determine the expected outputs, for instance "the output is the result of a given mathematical formula" or "if a key is pressed on the keyboard, the corresponding character is displayed on the screen". However, contrary to programs, those specifications do not describe how to generate the outputs using the operations of the processor. Moreover, they can be intentionally vague and non-deterministic in order to provide freedom to the implementation.

Termination Computers sometimes get stuck in infinite loops: they execute repeatedly some sequences of operations. Such infinite loops are undesirable if they prevent the execution of operations of interest, such as the output of the expected results or reaction to new inputs.

Time bounds Termination properties can be strengthened by requiring not only the computer to eventually perform some action, but also that those actions happen quickly. In some applications (numerical simulations, data processing…) it is desirable to obtain the result as soon as possible, but there are no precise bounds on the time available. On the contrary, in cyber-physical systems, it is necessary to react to inputs in a precisely bounded time for the system to reach its goal. Reducing delays below the fixed limit is however not necessarily useful.

Secret preservation Computers sometimes perform tasks involving secret data: cryptographic keys, private data… It is desirable that adversaries cannot recover those secrets by observing the behavior of the system. In such adversarial settings, the behaviors include not only the public outputs of programs (for instance messages on the network) but also any physical property that adversaries have access to. Those physical properties are called *side channels*; they include execution times and power consumption of processors. Defending against those attacks requires the publicly observable properties of the computer to be independent of the value of the secrets it receives.

Because programs control computers, all those properties on computers translate to desirable properties on the binaries. Additional properties arise from the interaction with the OS and processor:

Application Binary Interface (ABI) Binaries must adhere to conventions on the way they interact with the processor, the system and other programs. The set of conventions that must be respected by a binary, called ABI, includes the set of operations supported by the processor and ways to perform calls to the system and libraries.

Memory usage Programs need to store data in order to perform their tasks. Computer systems have both persistent and temporary memories. Programs may dynamically require access to some quantities of memory during their execution. However, the amount of memory available on a system is limited and shared between multiple programs. It is thus desirable to bound the amount of memory used by programs.

A *bug* is a behavior of the computer which does not satisfy an expected property. The root cause of some bugs is sometimes found in the processor or more generally the physical implementation of the system. However, most bugs are caused by software errors. We focus on this second category.

1.2.2 Correctness properties of compilers

For compiled programs originally written in higher level languages, the objective is still that the binary representation satisfies expected properties. However, the way properties are enforced is influenced by the compilation to binary.

Properties guaranteed by the compiler The compilation process ensures that the generated binaries satisfy some properties, independently of the source program (or at least for a subset of the source language). This is often the case for the respect of ABIs since those low-level conventions are handled by the compiler. Some languages, such as OCaml (without `Obj.t`) or the safe fragment of Rust, rule out some memory errors by performing automatic verifications (type checking, borrow checking) on the programs. Some Domain Specific Languages (DSLs) only provide access to a restricted subset of the capabilities of the processor to enforce some properties. For instance, the programs written in languages without loops or recursive calls are necessarily terminating.

Properties preserved by the compiler Some properties on binaries have equivalent properties on the source language. The compilation process can then be designed to preserve them. Hence, one can enforce a property on the executions of the binary by enforcing a corresponding property on the source programs. Contrary to the previous category, those properties are not enforced unconditionally by the compiler. This category includes more expressive properties, in particular the absence of any kind of crashes, functional correctness and termination.

Properties required by the compiler In order to perform transformations while ensuring the two previous categories of properties, compilers sometimes make assumptions on the behaviors of programs. For instance, C programs can read and write the memory state of the machine (in order to store temporary data), but are only allowed to access a subset of this memory. Indeed, the compiler also introduces structures in this memory to implement some abstractions (e.g., function calls). Arbitrary memory accesses in the C source could thus not only trigger errors at the binary level, but also introduce new behaviors by altering the implementations of abstractions.

Properties enforced at the binary level Because of abstraction, some properties desirable on the binary have no equivalent at the source level. Other properties are not preserved by compilers. This is often the case for strict time and memory bounds and the absence of side channel attacks. In those cases, one either needs to write parts of programs directly at a lower level, or to check that the output of the compiler satisfies properties strong enough for the application.

The guarantees given by a compiler depend on the source and target languages and on the compiler used. Those guarantees are often described as *refinements* between the source and the target programs. The definition of the notion of refinement depends on the guarantees considered. For instance, the preservation of functional correctness can be expressed as an inclusion of the finite behaviors (that is, the possible interaction of the program with its environment) of the target program in those of the source program. Another approach is to describe the specification of a compiler as a function from source-level to target-level properties [1]. Such a function describes the guarantees provided by the compiler on the target program, given a specification of the source program.

In this thesis, we will build a compiler which, under some assumptions on source programs, ensures:

- The absence of errors in the target. In particular, we will guarantee the absence of memory errors, while providing total memory operations (but with additional type constraints) in the source.
- The preservation of functional correctness, more precisely of the interaction of the program with its environment.
- The preservation of termination.

In particular, this specification does not require the preservation of the accesses to the temporary memory. The compiler is thus allowed to optimize the program by changing the way it uses the memory. Currently, our implementation only removes some unused reads, but we show on manual transformations that our framework enables larger changes, including optimizing memory allocations.

1.2.3 Formally verified programs and compilers

Formal methods Formal methods are mathematical approaches to prove the absence of bugs in programs expressed at various levels of abstraction. There are multiple approaches, which vary on several aspects:

- The properties that can be proved. For instance, some method may prove the absence of errors, but not functional correctness.
- The set of programs on which they can be applied. For instance some verification tools for Rust [37; 47] only support the safe fragment of the language. More subtly, most methods fail to prove properties on programs if their proof is too complex. In particular, proving the absence of errors sometimes requires proving intermediate functional correctness results.
- The level of automation of the verification tools. Some fully automatic procedures such as type checkers can validate restricted properties on programs using few annotations.

In this thesis, we will use the Rocq [46], which is a general-purpose prover. It enables stating properties on programs or more general mathematical objects. Proofs of those properties are then built interactively, using tactics described by the user.

Verified compilers One ultimately wants to obtain guarantees on binary programs, but formal methods are often applied at the source level. Indeed, some approaches have dedicated handling for high-level abstractions, which enable probing more properties than at the binary level. Moreover, it is easier for users to provide guidance at the source level than on a corresponding binary generated by a compiler. However, those source-level guarantees only imply properties on the binary if they are preserved by the compiler. One thus has to check that the specification of the compiler ensures the preservation of those properties. Moreover, the compiler is itself a program, which may itself have bugs. To increase the level of trust in the results claimed on the binary program, one can apply formal methods to verify the compiler. The desired property on a compiler is mainly functional correctness. That is, one wishes to prove that, on a correct source program, if the compiler successfully generates a target program, then it is also correct. By "correct", we mean here that the program satisfies some of the properties described in section 1.2.1. As described in section 1.2.2, the notion of program correctness is also not necessarily the same at the source and target level. Even the statement of the correctness of a compiler is thus a complex statement. For instance, the preservation of termination involves reasoning on infinite executions of an arbitrary program received as input. This motivates the use of expressive general provers to verify compilers. Another benefit is that those provers can also check the composition of the statements of the correctness of the compilers and of the properties of the source program. If both of those components are verified within the same prover, their composition may even yield a mechanically checked proof of properties at low level, without references to the source program.

Semantics Programs are defined syntactically, for instance in a textual representation or as Abstract Syntax Trees (AST) reflecting their structure. However, reasoning on programs and transformations requires defining the meaning of those syntaxes. This meaning is called the *semantics* of the programs. Such semantics is a mathematical object which captures the relevant properties of the program. It may abstract away some details of the syntax. For instance the expressions `x + 1` and `1 + x` have distinct syntaxes, but may have the same semantics. However, the semantics must be precise enough to define whether properties of interest are satisfied. For instance, giving the same semantics to `x + 1 + 2` and `x + 3` is acceptable if one only cares about functional correctness, but not if one is concerned about side channel attacks, since it could be observed that the second syntax performs one less addition.
In practice, one defines the semantics of languages, that is, the semantics of all programs written in a given language. There are several approaches to obtain such definition. In this thesis, we use a denotational approach.

1.3 Propagation of information in compilation pipelines

Some transformations performed by compilers are only valid under semantic conditions on programs (section 1.3.1). Those conditions may be checked using several sources of information (section 1.3.2). It is thus useful to propagate information from high-level to low-level representations in the compilation pipeline (section 1.3.3). This propagation is the subject of this thesis. We focus on cases where both the source program and the compiler are verified, and argue that propagating information could be particularly useful in this setting (section 1.3.4).

1.3.1 The need for semantic information during compilation

Compilers may optimize programs, for instance to reduce their execution time, memory usage or size in binary format. Those optimizations are often performed by dedicated passes, which transform programs without changing the language in which they are expressed. In all cases, the optimizations can only be performed if they preserve the properties of interest of the program. In this thesis, we thus consider transformations that preserve the outputs of the program and do not introduce crashes or divergence. However, the validity of some transformations depends on semantic properties of the program.

As a first example, consider the transformation of figure 1.2a. Here and often in this manuscript, we use the syntax of C to illustrate transformations, even when they are actually performed at a lower level. In this case, the transformation depicted is an integer promotion. It increases the number of bits used to encode the integer variable `i` from 32 to 64 bits. As explained by Lee [64, §2.4], this transformation can reduce the execution time by avoiding a conversion within the loop. However, this transformation is not always valid. Indeed, because integers are represented using a fixed number of bits, machine operations on them actually implement a modulo arithmetic. This means that there is a maximal representable integer ($2^{s-1}-1$, where `s` is the number of bits of the encoding). Adding 1 to this value yields the minimal representable integer (-2^{s-1}), a phenomenon called *integer overflow*. In our example, if `n` is larger than $2^{31}-1$ (which is possible if it is encoded using 64 bits), then in the source, `++i` overflows and the loop never terminates. On the contrary, the target loop always terminates, because `i` can increase above 2^{31} and up to `n`. In this case, the transformation would thus change the behavior of the program (depending on the effects of `foo`), which is undesirable.

As a second example, figure 1.2b describes a Common Sub-expression Elimination (CSE), which factorizes the two computations of the expression `*p`. This program involves explicit accesses to the memory of the machine. Informally, this memory can be thought of as a map from *pointers* (also called addresses) to values. Programs access the values stored in the memory via their addresses, which may be manipulated as variables of the program. For instance, figure 1.2b involves two variables `p` and `q` containing pointers to some locations in the memory. In the general case, pointers act as indirections between the program syntax and the memory of the machine (figure 1.3). Since their values are dynamic (that is, depend on the execution of the program), pointer variables enable writing code independent of the region of the memory accessed. For instance, the standard `memcpy(void* p, void* q, size_t n)` function copies `n` bytes from the memory region starting at the address given in the argument `q` to the region starting at the address given in the argument `p`.
In our example, the factorized expression `*p` loads the value stored at address `p` (that is, at a pointer itself stored in the variable `p`) in the memory. It thus depends on the state of the memory, and the transformation is only valid if this state contains the same value during the two source occurrences of `*p`. However, the program contains another memory access between them: a write `*q = 0` at address `q`. Although `p` and `q` are syntactically distinct variables, they could contain the same address, a situation called *aliasing*, illustrated in figure 1.3b. If this is the case, then in the source, the second `*p` loads the value stored through `q`, so the result `y` is 0 (figure 1.3c). On the contrary, in the target, `y` is always the

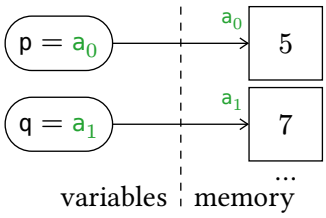
```
for(int32_t i = 0; i < n; ++i) {
  foo((int64_t)i);
}
```
→ if ++i does not overflow →
```
for(int64_t i = 0; i < n; ++i) {
  foo(i);
}
```

a: Integer promotion, in the absence of integer overflow

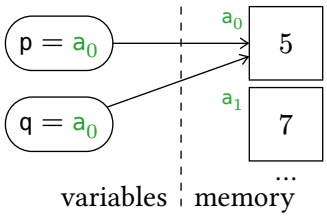
```
int x = *p;
*q = 0;
int y = *p;
```
→ if p and q do not alias →
```
int x = *p;
*q = 0;
int y = x;
```

b: CSE, under non-aliasing conditions

Figure 1.2: Examples of conditional transformations



a: non-aliased



b: aliased pointers

	Source		Target
non-aliased	5	→	5
aliased	0	↛	5

c: result in y

Figure 1.3: Examples of initial memory configuration, and effect of the transformation of figure 1.2b

value initially stored in the memory. This factorization is thus only valid if p and q do not alias. Those aliasing properties are a major focus of this thesis.

1.3.2 Sources of information

In order to determine whether a transformation is valid, compilers thus need information about the program given as argument to the pass. There are several ways of obtaining that information.

Analyses Compilers contain analyses, which are fully automated procedures establishing some classes of properties on programs. For instance, in figure 1.2a, an interval analysis may find bounds on i and n, valid for any execution of the source program. If they are low enough, those bounds can ensure the absence of overflow and thus guarantee the validity of the transformation.
In fact, prior to this thesis, I worked during an internship at Verimag on the verified compiler Chamois (a fork of CompCert). I implemented an integer promotion pass to improve the results of a later optimization [45, §6.5]. I relied on an interval analysis to decide which variables could be promoted. In a joint work with Léo Gourdin, we also extended a Lazy Code Motion (LCM) pass [45, §5] to enable the use of aliasing information. In this case, information is provided by the standard `ValueAnalysis` of CompCert.
However, analyses are not always precise enough and may miss assumptions such as function pre- and post-conditions. Moreover, in verified compilation pipelines, the analyses need to be verified [17; 89].

Information available on the source Some information is already known on source programs. Programmers maintain invariants, at least informally, to ensure the correctness of their programs. In particular, interactions between components (e.g., at function calls) are subject to conventions between the client and the implementation of the component. Languages sometimes provide explicit annotations to transmit that information to the compiler. For instance, in C, the `restrict` qualifier indicates that accesses to a set of pointers (typically given as function arguments) do not alias. Some languages also put restrictions on the operations that source programs are allowed to perform. Those restrictions reflect the conventional usage of the language, and allow compilers to obtain information without additional annotations. They are formalized through Undefined Behaviors (UBs), some kinds of source-level errors, which cancel some or all guarantees provided by the compiler. For instance, integer overflow is UB in C when signed variables are used, so the compiler is always allowed to perform the transformation of figure 1.2a. Still in C, the strict-aliasing rules ban some cases of aliasing

between pointers, depending on the types of objects referenced.
The producer of the source program is responsible for the validity of the semantic information it contains, both explicitly and implicitly. Invalid information may lead to apparent miscompilations. This can happen if the producer of the program does not respect the conventions expected by the compiler. For this reason, the use of that information by compilers is sometimes deliberately restricted. For instance, the strict-aliasing rules are disabled when compiling the Linux kernel [101]. Even verified compilers like CompCert avoid using UBs which are likely to happen in real-world unverified source code[4].
Automated source-level checks provide more reliable sources of information. Languages with a static type system may thus retrieve information about the contracts of functions, including external functions with an unknown implementation. Recently, the borrow system of Rust has also been identified as a source of non-aliasing information [50; 99].

Structures of translations Translation passes emit low-level code implementing high-level abstraction. The correctness of such translation relies both on invariants between its source and target, but also on properties of the code it generates. For instance, a register allocation is a pass from an IR with an unbounded number of variables (sometimes called pseudo-registers) to an IR with a bounded number of registers. To do so, it reserves some space in the memory to store the values of some variables when there are not enough registers. This implementation relies on non-aliasing properties between the storage locations of some variables, and with other structures stored in the memory (e.g., the parts accessed explicitly in the source program). In the general case, passes occurring after translations may optimize the generated code, and can benefit from the properties it satisfies.

1.3.3 Encoding information in intermediate representations

Except for the analyses, the sources described in section 1.3.2 provide information at a fixed level of the compilation pipeline, for instance on the source program or on the output of a translation pass. However, that information can be useful later in the pipeline, after the application of several passes. One thus needs ways to propagate information in the compilation pipeline simultaneously with the representation of the program itself. In fact, this may also be useful for information obtained through analyses, because they can be costly to run or only available on particular representations.

This information needs to be maintained by the passes. The inputs and outputs of passes are thus no longer only programs, but pairs of programs and information on them. More pragmatically, information can be encoded directly in the representation of programs, to avoid updating auxiliary data-structures. In all cases, the specification of passes now has to take into account this additional information.

- On one side, passes can rely on the information accompanying their input program.
- On the other side, passes have to guarantee the validity of information encoded with their output program.

Similarly to the source level, semantics of intermediate representations can be altered (or even designed) to take encoded information into account. For instance, LLVM IR features optional attributes representing information. The semantics of those attributes is formalized using UBs [64].

1.3.4 The case of formally verified programs and compilers

In this thesis, we focus on the case of formally verified compilation applied to verified programs. We argue that propagating information could be especially useful in this setting:

- The source-level invariants are likely expressed in some formal language, that can be taken as input by the compiler. This includes both invariants written directly by the user and those inferred by

[4] For instance, https://github.com/AbsInt/CompCert-small-tests/blob/4df7312a/regression/alias.c tests that the `ValueAnalysis` remains conservative on liberal low-level patterns.

automatic passes. This contrasts with unverified programs, whose invariants are likely to be written as documentation in informal language.
- Some of the properties satisfied by the output of verified translation passes are already formalized. To propagate those properties, one still has to encode them in the target. However, proving the validity of this encoding could be achieved without additional invariants.
- Formally verifying a program often requires finding precise invariants. One can thus expect more information on those programs. In particular, we will assume precise memory properties, with bounds on all accesses performed by programs. On the contrary, alias analyses may fail to provide information about some accesses, such as those performed by external functions. Our goal is thus not to use the results of such analyses.
- Guaranteeing the validity of information with formal proofs dramatically reduces the risk of introducing miscompilation when using it.

Because we aim to use verified programs as sources, a point of attention is the effort needed to verify them. In particular, using solely UBs to encode information can only make the verification task harder. Indeed, UBs yield additional side-conditions that must be proved on the source program. This also applies to semantics of IRs, since one must formally justify that passes generate UB-free programs. We instead aim to design semantics that leverage encoded information to reduce the verification effort. In particular, we draw inspiration from verification tools on the safe fragment of Rust [37; 47], which leverage non-aliasing properties enforced by the borrow checker to simplify the verification of functional properties.

1.4 Contributions

The first contribution of this thesis is a new intermediate representation, called Frag IR. This IR is designed to support the propagation of information within verified compilation pipelines. It is implemented within the Rocq prover [46], and its meta-theory is formalized, including its semantics and typing judgment. We implemented and proved correct a compilation pipeline to another representation, called Flat IR. This second IR is also new, but is similar to standard low-level languages. Although our compilation pipeline currently does not generate binaries, an interpreter on Flat IR enables executing the programs it generates. Passes on Frag IR preserve information encoded in the programs. To demonstrate the possibility of using Frag IR as a back-end, we connected our pipeline with the verified compiler Vélus[5]. This connection encodes properties enforced by construction in the IRs of Vélus into our more general representation.

Our second contribution was motivated by the implementation and formalization of Frag IR and its pipeline in Rocq. We implemented a library `SMonad` centered on an axiomatization of monads used to define denotational semantics. This axiomatization is inspired by Interaction Trees [102]. We generalize the relational approach used to reason on this domain, and show that it enables proving multiple transformations, including inter-procedural ones and loop transformations. `SMonad` abstracts over the effect of semantics. It is thus possible to perform a single proof for the same transformation on semantics with different effects. We give a model of our axiomatization, based on transition systems and supporting several effects: divergence, events, non-determinism, and undefined behaviors. We use `SMonad` to define a parameterized intermediate representation in functional Static Single Assignment. We define both Frag IR and Flat IR as instances of this representation, despite the different effects of their semantics. We implemented several verified passes on the parameterized representation, including a type-checker, a Dead Code Elimination, and a block-sinking transformation. Those passes have themselves a few parameters, corresponding to parts specific to the instances. By implementing those parameters, we obtain those passes on Frag IR and Flat IR. Both `SMonad` and our parameterized

[5] Vélus transpiles Lustre programs (for safety-critical embedded systems) to C. See https://velus.inria.fr/.

SSA representation are independent of Frag IR, and could be reused to define other languages and passes of verified compilation pipelines.

1.5 Conventions

We describe the conventions used in the present manuscript. Section 1.5.1 explains how it relates to the Rocq formalization. Section 1.5.2 lists some notations, and section 1.5.3 defines standard concepts that we will use in the following.

1.5.1 Link with the development

Unless signaled, all definitions and proofs described in this manuscript have been formalized in Rocq. However, this document is not a formal documentation of the development. Our goal is instead to describe the high-level ideas, including its structure and why the properties proved hold. For conciseness and to ease the reading, we took some liberties with respect to the formal definitions. For instance, we generally identify textual descriptions of programs with their ASTs, use partial functions as if they were total, and omit well-formedness conditions on the objects. The manuscript also introduces notations for some of our concepts, whereas we usually prefer names in the development. For conciseness, we reuse the same unqualified names or notations in different sections.

1.5.2 Notations

We use informal notations, in between the syntax of Rocq and more standard mathematical notations.

Definitions and variables In this manuscript, we differentiate the names used for variables from those used for defined objects. The latter notion includes definitions, lemmas, axioms and modules. We write them in `blue`. In the digital versions, those names are links to the point where they are introduced. The variables are written in `green`, and often introduced implicitly.

Inductive types and propositions Rocq features a unified notion of "inductives", which enable defining both types and propositions. In this manuscript, we use different notations for the two situations. For types, we use a BNF-like presentation, similar to the one of Rocq or of the algebraic datatypes of OCaml. For instance, the standard family of types `option` is defined by:

```
option (A : Type) := Some (x : A) | None
```

This family is indexed by a type parameter `A`, and has two constructors `None` and `Some`. This second constructor has an argument `x` of type `A`. We use the same notation to define the syntax of languages, and identify this syntax with their ASTs.
For propositions, we instead list their constructors as inference rules. For instance, the relation less or equal on natural numbers is defined in Rocq by two constructors `le_n` and `le_S`:

$$\frac{}{n \leq n}\,\texttt{le_n} \qquad \frac{n \leq m}{n \leq m + 1}\,\texttt{le_S}$$

The relation $n \leq m$ holds if it is derivable by a proof tree without infinite branches and whose rules are instances of those two constructors. For instance, the proposition $1 \leq 2$ holds because of the following tree:

$$\dfrac{\dfrac{}{1 \leq 1}\,\texttt{le_n}}{1 \leq 2}\,\texttt{le_S}$$

We also use inference rules to present sets of lemmas or axioms. The specificity of inductive definitions is that the set of constructors is described exhaustively. We will thus say that an inductive proposition is "generated" by its inference rules.

We will also occasionally use coinductive types and propositions. There is no finiteness condition on their trees. We signal those occurrences explicitly.

Functions Given two types `A` and `B`, we note `A → B` the type of total functions from `A` to `B`, and `A ⇀ B` for the partial functions. The latter is encoded in Rocq as a total function with an `option` type as co-domain (`A → option B`). We note `dom f` the domain of a partial function `f`. Like in functional languages, functions are applied without parentheses. For instance, given `f : A → B` and `x : A`, the application `f x` yields an element of type `B`. By default, functions with multiple arguments are curried. For instance, $f : A_0 \to A_1 \to B$ is a function with two arguments and is applied with $f\ x_0\ x_1$.
Rocq features dependent arrow types. For instance, `Option.map [A B : Type] (f : A → B) : option A → option B` describes the signature of a function where some arguments (`f` and the anonymous `option A`) and the result types depend on the arguments `A` and `B`. We also use this syntax when we want to name the arguments, or mark some of them as implicit (within brackets [...]). Like in Rocq, the notation ∀ also describes dependent function types, for instance `∀A. option A → option A`.
The notation `λ(args...). body` defines an anonymous function. We extend or update a function with the notation `[f, x ↦ y]`. We generalize this notation to multiple updates, either by providing multiple bindings ($[f, x_0 \mapsto y_0, x_1 \mapsto y_1]$), corresponding lists of source and target elements (`[f, xs ↦ ys]`, where `xs` and `ys` have the same length), or another function (`[f, g]` `x` is equal to `g x` if `x` is in its domain, and to `f x` otherwise). We omit the function `f` to define a new function by enumeration, e.g. `dom` $[x_0 \mapsto y_0, x_1 \mapsto y_1] = \{x_0, x_1\}$.
The composition of two functions `f : A → B` and `g : B → C` is written `g ∘ f : A → C`. We may apply a function `f : A → B` directly on an object `u : T A` containing elements of type `A`, if `T` has a functor structure (i.e., a "map" function, like `Option.map` for `T = option`). We signal those applications using parentheses, `f(u) : T B`. For instance, on a list, $(\lambda n.\ n + 1)([1; 2; 3]) = [2; 3; 4]$

Records Records are types (or propositions) gathering several named fields. Those types are declared with the syntax $\{\texttt{field}_0 : \texttt{ty}_0; \texttt{field}_1 : \texttt{ty}_1\}$. Record values have shape $\{\texttt{field}_0 := \texttt{v}_0; \texttt{field}_1 := \texttt{v}_1\}$. We access the fields of such value `a` using projections, e.g. $\texttt{a.field}_0$. Record types may also contain propositions as fields, which is useful for enforcing well-formedness conditions.

Substitutions We write `a[b/x]` for the capture-avoiding substitution of a free variable `x` by an element `b` in a syntactic element `a`. For instance, for the syntactic expressions `a := (x = y)` and `b := (z + 1)`, the substitution `a[b/x]` yields `z + 1 = y`. Similarly to function extensions, we generalize the notation to multiple parallel substitutions and to lists of variables and substituted elements.

1.5.3 Standard definitions

Propositions and Booleans Rocq distinguishes the propositions (`Prop`) from the Booleans (`bool`). The former are used to state properties, while the latter are used within executable implementations, like in programming languages. The proposition `True` (resp. `False`) is always (resp. never) satisfied. The type `bool` is defined as an inductive datatype with two constructors: `bool ::= true | false`. We use the same notations for the operators on `Prop` and `bool` (e.g., ¬, ∨).

Types and sets The sort `Type` of types is itself a type in Rocq. This manuscript omits all details related to the universe system, which ensures the soundness of this feature. We often use the word "set" instead of "type" to avoid confusion with the types of our languages (e.g., our "sets of types" are actually `Type`s in Rocq). We occasionally use set-theoretic notations, for instance enumerations "$\{x_0, x_1\}$". The equivalent of a singleton set in `Type` is an inductive `unit ::= tt`. The disjoint union of two types `A` and `B` is always defined, by a sum type `A + B ::= inl (x : A) | inr (y : B)`. The inhabitants of a product type `A × B` are pairs `(x : A, y : B)`. Rocq also features dependent pairs (called sigma types) whose types are written `{x : A & B x}` (where `B : A → Type` is a dependent type). Their inhabitants

are normally written `existT B (x : A) (y : B x)`, but we will also use the notation `(x, y)`.
Sets are also sometimes encoded as predicates, `A → Prop`, describing a subset of a carrier type `A`.

Optional results We already described the standard `option` type. We use the notations `⌊x⌋` for `Some x` and `∅` for `None`. We also define a usual result type, which provides more information about errors: `res [E : Type] (A : Type) := ROk (x : A) | RErr (e : E)`.

Numbers Several types represent numbers in Rocq. The type `nat` encodes natural numbers (ℕ) in a unary fashion. This is convenient for proofs, but not efficient when used in implementations. The type `N.t` represents the same set, but with a more efficient binary encoding. Positive and relative numbers (ℤ) are also encoded in binary in the types `positive` and `Z`. The dependent type `Fin.t n` represents the `n` naturals between 0 (included) and `n` (excluded). We also reuse the formalization of bounded machine integers of CompCert, which is based on `Z`.
In this manuscript, we generally do not specify which representation is used in the development: we use the operations for all types of numbers, and omit the conversions between them.
We never use rational or real numbers, so intervals `[n; m)` are always interpreted as integer intervals.

Lists The standard representation of lists is defined inductively `list A := [] | (hd : A) :: (tl : list A)`. The length of a list is given by `List.length l`. We note $\texttt{l}_\texttt{i}$ the $\texttt{i}^{\text{nth}}$ element of `l`, for $0 \leq \texttt{i} < \texttt{List.length l}$. The consing operation, `hd :: tl`, adds a single element at the beginning of a list. The concatenation $\texttt{l}_0 +\!\!+ \texttt{l}_1$ is defined for lists $\texttt{l}_0, \texttt{l}_1$: `list A` containing the same type of elements. The operator `List.combine [A B : Type] : list A → list B → list (A × B)` pairs in a pointwise fashion the elements of two lists, with the same length but different types of content.
Our development also uses the dependent type `Vector.t A n` of lists of fixed length `n`. In this manuscript, we abbreviate this type as $\texttt{A}^\texttt{n}$. However, we often omit the dependence in `n` and replace it with `list A`. We also use the notations and operations of lists for their equivalents on `Vector.t`.

1.6 Outline

Chapter 2 discusses related works on the propagation of information in compilation pipelines, but also more generally on verified compilers. Chapter 3 then describes informally the main ideas of the design of Frag IR. We then present more formally our contributions, in the order of their dependencies in our development.
Part II thus starts by presenting our use of denotational semantics and our parameterized SSA IR. Chapter 4 describes our library `SMonad`, and chapter 5 its LTS-based model. Chapter 6 presents our parameterized SSA IR and its meta-theory. Chapter 7 lists our parameterized passes and explains their proofs.
Part III describes the components of our pipelines around Frag IR. Chapter 8 starts by detailing the implementation of Frag IR. Chapter 9 presents the pipeline from Frag IR to Flat IR. Finally, Chapter 10 describes the connection with Vélus.

2

State of the art

This chapter describes related works relevant to our main goal, namely the propagation of semantic information in a verified compilation pipeline. Other chapters contain sections comparing our approach with existing works. Section 3.4 is centered on techniques similar to our encoding of information. Section 4.6 compares our reasoning on semantics with other approaches. Section 6.6 focuses on several aspects of our representation of programs. This chapter discusses the following topics: the representation of information using undefined behaviors (section 2.1), memory models used in formal semantics (section 2.2), the representation of aliasing information in verified and unverified compilers (section 2.3), existing formalizations related to verified compilation (section 2.4) and approaches to verify programs involving a memory state (section 2.5).

2.1 Undefined Behaviors and their limitations

Undefined Behaviors (UBs) are specific events that may occur in the semantics of programs. Correct programs should not contain UBs. Compilers assume this semantic property on the source program, and maintain it on all generated representations, including the target. One of the classical uses of UBs is to encode properties known about a program (section 2.1.1). However, this semantic approach does not encode *why* the properties are satisfied. By default, compilers must thus consider that information provided by a UB depends on the whole execution that leads to it (section 2.1.2), which restricts the possible transformations. Deferred undefined behaviors are a weaker kind of UBs which allows some of the transformations prohibited by standard UBs (section 2.1.3). However, they require modifying the values used by programs and may prohibit other transformations.

2.1.1 Encoding information in the semantics

There are several formalizations of UBs, some of which may enable more transformations. We discuss more precisely those differences in section 4.6.6. All those formalizations have in common that, if the source program has an execution which emits a UB, then corresponding target executions are allowed to do anything after this point. This license given to the compiler is justified by the fact that we expect the source program to be safe, that is, that its executions never emit UBs (at least for the interactions with the environment we are interested in). With this point of view, UBs are assumptions provided to the compiler about the semantics of the source program. Intuitively, the compiler assumes that

the semantics of the source program does not contain UBs[6]. The absence of UBs is an invariant of compilation pipelines. That is, this property holds on the source program (by assumption) and each transformation must ensure that the program it generates does not have UBs if its input does not.

The classical way of encoding information using UB leverages this invariant. With this approach, the fact that a property holds at a program point is encoded in the semantics, by raising a UB when the program reaches the point without satisfying the property [64]. For instance the C standard considers overflows of signed integers as UBs, enabling compilers to obtain bounds on the values and to perform transformations. Intermediate representations can also have UBs. This approach thus allows both recovering information on the source program and propagating information in the compilation pipeline. Formally, the addition of UBs has the following consequences:

- The producer of the source program must ensure that it does not contain UBs. Formal proofs of the correctness of the source program thus contain corresponding obligations.
- During proofs of transformations, if the source program reaches a point where it has a UB if a property P is not satisfied, we obtain the assumption that P holds. However, we also have to prove that the generated target program does not have UBs.

2.1.2 Implicit dependencies of UBs

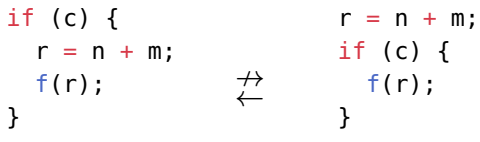
When propagating information using UBs, we encode in the semantics that some properties are satisfied at some points during program executions. However, since UBs are defined semantically, they implicitly depend on the whole execution that leads to them, in particular on the control flow. For instance in figure 2.1, if one chooses to consider integer overflow as UB, then the absence of UB in program (a) ensures that the addition does not overflow, but only if the condition `c` is true. The program (b) is obtained by moving the addition above the conditional. This is an instance of a speculation: in some executions the target performs an operation that the source does not. This kind of transformation is useful to avoid multiple computations of the same value within a loop. However, the transformation is invalid if integer overflows are UB. Indeed, the absence of UBs in (b) implies the absence of overflow independently of the value of `c`, which cannot be justified using the information provided by (a). The reverse transformation is valid, but cannot be reverted. This example illustrates not only that UBs introduce dependencies that are not reflected by the syntax, but also that they can prevent transformations. For this reason, intermediate representations designed for optimizations avoid assigning UB to instructions that are implemented with total operations in the target language. For instance, the addition of two integers never has UB in the RTL representation of CompCert, but information that was ensured by the absence of overflow in the C source is lost at this level.

2.1.3 Mitigating restrictions of UBs with poison values

A compromise between retaining information and speculation is the use of *deferred undefined behaviors* [64]. Contrary to immediate (i.e. standard) UBs, deferred UBs are associated to an instruction instead of a program point. They may ultimately trigger standard UBs, but only if the result of the instruction affects the observable behavior of the program. It thus allows speculative transformations, like figure 2.1. Indeed, if we consider integer overflow as a deferred UB, then figure 2.1(b) does not have a UB if `c` is false and the addition overflows since its result is not used. Information encoded using deferred UB can be used to transform the instruction associated to the UB and the computations which depend on it. For instance, in figure 2.1, a compiler could assume the absence of overflow when transforming `n + m`. However, the information cannot be used to transform other parts of the program.

Formally, deferred UBs are encoded by extending the set of values with a dedicated *poison* value (called `Vundef` in CompCert and poison in LLVM [64, §2.4] and VIR [14; 105]). The semantics handle this

[6] In some formalizations (including ours), the compiler still has to generate a target program which behaves like the source until the emission of UBs. In this case, the absence of UBs cannot be given as a global assumption like we do here.



(a) (b)

Figure 2.1: A transformation invalid if integer overflow is UB

poison value in a way that ensures that it cannot influence the observable behaviors of the program without raising UB. In particular, using the poison value in external events or for the control flow yields an immediate UB. Moreover, it is propagated by computations. That is, operations applied to a poisoned argument return poison as a result. Note that this formalization strictly over-approximates the idea of "not influencing the observable behaviors". For instance, a conditional on poison always raises an immediate UB, even if its branches only compute values that do not influence the observable behaviors. In formal proof, the fact that poisoned values do not affect the behavior of programs enables replacing the source poison values by arbitrary values in the target. In CompCert, this is technically achieved using "`lessdef`" invariants [69, §11.3] between the source and the target: corresponding source and target data are equal unless the source is poison. Such principles enable using information encoded with deferred UBs. For instance, consider a transformation of `x = e0` into `x = e1`, where `e0` and `e1` are pure expressions (in particular without immediate UBs). If `e0` returns poison when some property P is not satisfied, then the transformation is valid if, *under the assumption* that P is satisfied, `e0` and `e1` evaluate to the same result.

As another application, poison values can be associated to uninitialized variables and memory locations. This prevents programs from depending on uninitialized values. We come back to this topic in sections 8.1.1.4 and 8.1.3.2.

A drawback of deferred UBs is that they require extending the set of values. Poison values must be taken into account in the whole semantics, for instance in the memory model (e.g. [62, §2]). Those additions can break some reasoning properties. For instance a division by `2 * n + 1` becomes a partial operation since it is not defined if `n` is a poison value.

2.2 Memory models

Defining the semantics of a language involving a memory requires a formal model for the memory. The choice of model influences the properties provable on programs and which transformations are valid.

Concrete models A first approach is to mimic the low-level interface of the memory. Assembly programs have access to a sequence of memory bytes. Each byte is accessed through its address, which is a machine integer. Those addresses thus belong to the range $[0; 2^s)$, where `s` is a constant corresponding to the number of bits used to encode them, typically 32 or 64. In concrete memory models [52, §2.1] (sometimes called "flat" [65, §2.1]), pointers are thus machine integers and states of the memory are maps from pointers to bytes. Those models have the advantage of being simple and of allowing programs to perform operations using the low-level representation of pointers (e.g., bit stealing [10, §2.3.1] or using them in hashes). In particular, they are well-suited for local transformations at the assembly level [79].
However, some higher-level program transformations are hard to justify in those models, because they constrain too loosely the usage of pointers by programs. For instance, in figure 2.2, it is tempting to consider that `foo` does not modify `x`, since it does not have access to its address. In practice, LLVM `-O2` may indeed replace following occurrences of `x` by `0`, without knowing the implementation of `foo`. However, justifying such a transformation requires proving that `foo` never performs a store at the addresses where `x` is stored. In a concrete model, those addresses are integers. One must thus justify that `foo` cannot build those addresses from all the information it has access to (in particular the pointer

```
int x = 0, y;          int x = 0, y;
foo(&y);        →      foo(&y);
bar(x+1);              bar(1);
```

Figure 2.2: A transformation performed by LLVM and CompCert

to y, which could be stored near x) or even by guessing the address of x in some executions. More precisely, the twin allocation technique [65, §3.2] shows that it is enough to justify that `foo` cannot build *reliably* the address of x. That is, the transformation is indeed valid if `foo` may access x in some executions but also has UB in others.
Another drawback of concrete models is that the set of addresses is finite. This can invalidate some compiler optimizations [14; 52; 65], such as the removal of dead allocations. Although this happens only in some artificial examples where the set of addresses is nearly exhausted, this may prevent formal proofs of transformations, since they must cover all cases.
To summarize, concrete models enjoy simple definitions but are hard to reason about.

Logical models To enable proofs of transformations, CompCert introduced a more abstract memory model [68; 69]. In this logical model, the memory contains an infinite number of blocks. Each block is indexed by a natural number and contains itself a sequence of bytes indexed by machine integers. A pointer is a pair $(\mathtt{b}, \mathtt{ofs})$ of a block number $\mathtt{b} \in \mathbb{N}$ and of an offset $\mathtt{ofs} \in [0; 2^s)$ within this block. A block becomes available after a corresponding memory allocation (either stack frames and variables or explicit calls to `malloc`), which return a pointer to the base (offset 0) of the allocated block. Because there is an infinite number of blocks, allocation is a deterministic function which selects the next block available. In CompCert, memory allocation thus never fails. Moreover, different allocations return distinct blocks.
Contrary to concrete models, logical models only support a subset of the operations available at low-level on pointers. This restricts both source programs and intermediate programs generated by the compiler in semantics with logical memory models. In particular, they cannot be converted to machine integers. The supported operations include pointer arithmetics (adding an integer offset, subtracting pointers) and comparison. Except for equality tests, those operations are UB if they are applied to pointers referencing different blocks. Another drawback of logical pointers is that they cannot be encoded to sequences of bytes (since they contain unbounded block identifiers). As a consequence, the memory does not simply store bytes, but either directly values (for the first version of the memory model of CompCert [62]) or generalized memory bytes (`memval` in CompCert memory model v2 [68], "symbolic bytes" in VIR [14]).
This model supports memory transformations which remove or merge blocks [62, §5.4]. Moreover, it prevents programs from *forging* pointers. That is, the only way of obtaining a pointer within an allocated block is to derive it from another pointer within the block. Combined with the fact that allocations return distinct blocks, this sometimes bounds the set of accesses to subsets of the memory. For instance, in figure 2.2, the variables x and y are allocated separately in the semantics of the C formalized in CompCert. They are thus stored in distinct blocks. Since the address of x has not been taken before the call, `foo` has no way of obtaining a pointer to x. The constant propagation of CompCert[7] is thus able to perform the transformation of figure 2.2.
To summarize, logical memory models are slightly more complex than concrete models, but enable more transformations. However, they only support a subset of the operations on pointers.

Hybrid models Several formalisms attempt to recover more operations on pointers in logical models. Those hybrid models involve both logical and concrete pointers.

[7] Actually, this transformation is the combination of two optimizations. First, since the address of x is not taken, the pass `SimplLocals` turns x into a temporary variable, which is not stored in the global memory. This later allows the actual constant propagation `Constprop` to propagate its value independently of memory accesses.

In the model of Kang et al. [52], pointers are by default logical. However, it supports casts from pointers to integers by allocating concrete pointers during casts. The drawback of this model is that those cast operations become effectful.
CompCertS [10] is a variant of CompCert which supports more low-level operations on pointers. Its semantics is still based on logical pointers, but values are replaced by expressions describing their computation as applications of operators on standard values (including logical pointers). Low-level operations on pointers are allowed by replacing the logical pointers by concrete ones in an operation called *normalization*. This operation is only performed in some effectful instructions (e.g., for the addresses of loads and stores). The program has UB if those normalizations depend on the mapping from logical to concrete pointers. Despite this restriction, this model enables giving formal guarantees for the compilation of programs involving bit stealing.

Enforcing language-specific constraints Some memory models enforce language-specific disciplines on memory accesses. By assigning UB to violation of those constraints, semantics provide assumptions to the program analysis which assume the absence of UB (which is the case when they are used during compilation) and may enable more transformations. Detecting those violations may require additional data-structures in the memory model.
Krebbers [58] proposes a formalization of the C standard in Rocq, using a memory model [57] enriched with trees corresponding to the shape of C objects (e.g., `struct`s and `union`s). This enables the use of strict-aliasing rules to recover non-aliasing properties between objects with disjoint types. Klappe [56] studies formally the `restrict` qualifier, another source of aliasing information in C.
The Rust compiler features a static verification algorithm, called the *borrow checker*. Among other things, this check enforces an aliasing discipline at the source level. Programs written in an "unsafe" fragment of the language can bypass this verification, but are expected to respect the aliasing discipline when interfacing with safe code. Stacked borrows [50] and their improvements tree borrows [99] formalize the aliasing discipline of Rust, by enriching the memory model with stack or tree structures. Those formalizations enable requiring semantically the unsafe code to conform to the aliasing discipline. Moreover, by capturing within the semantics properties enforced by the borrow checker, they could propagate this information in compilation pipelines. Both models can justify some program transformations, notably within the framework Simuliris [41] for stacked borrows.

2.3 Representation of aliasing information

Some memory models already propagate aliasing information. This is especially the case for those enforcing language-specific constraints. We now discuss existing approaches to propagate more generic non-aliasing properties, that is, not specific to a source language. Those approaches rely on various kinds of annotations in intermediate representations. Those annotations describe the known non-aliasing properties. In this section, we consider languages with formal semantics as well as mechanisms used in unverified compilers.

Classes of addresses A first family of approaches [44; 82] associates "classes" to each syntactic address. For instance, when compiling C programs, those classes can distinguish accesses to the stack, the heap and global variables. More generally, classes may encode properties derived from the structure of the source language, results of analyses, but also annotations provided by the programmer. For instance, in LLVM [71], the `noalias` and `noalias.scope` metadata enable propagating information provided by the `restrict` qualifier of C in a way compatible with function inlining. This is also the approach I followed during my internship to propagate transitorily non-aliasing information in the BTL representation of Chamois [45, appendix B]. In this case, the classes were the abstract addresses inferred by the `ValueAnalysis` of CompCert.
In the general case, a possible interpretation for classes is to see them as constraints on the dynamic

values of pointers. Optimizations can then test if two syntactic addresses may overlap by checking whether the sets of pointers allowed by their respective constraints intersect.

Dependencies between instructions Another approach is to maintain a graph of dependencies between syntactical memory accesses [48]. The nodes of this graph are instructions. If an instruction `j` depends on an earlier instruction `i`, then there is a path from `i` to `j` in the graph. This includes the implicit dependencies between memory accesses. That is, there is a dependency between two memory accesses `i` and `j` if they access addresses that are not known to be disjoint, and at least one of them is a store. This structure is used by some program transformations, notably instruction scheduling. However, maintaining those graphs is non-trivial when transforming loops. In practice, they are rarely kept within intermediate representations and instead built by the passes that need them.

Variables representing fragments of the memory Steensgaard [96] proposed partitioning the memory into several fragments, using the results of alias analyses. Those fragments are represented within programs using objects called "functional stores". A `lookup` operator takes as argument both the address accessed and the functional store in which it is stored. Similarly, an `update` operator takes as argument a pointer, a functional store and the value to be written. It returns a new store, which must be used in following operations instead of the store given as argument. The partition of the memory evolves during the execution of the program. For this purpose, an operator `merge` takes as argument several stores and returns a new store containing their union. A second operator, `restrict`, returns a sub-part of a given store, corresponding to some abstract addresses coming from a points-to analysis.
Firm [11] uses a similar representation, where the fragments of the memory are called "memory values". Its `Store` node has the same signature as the `update` of Steensgaard, and a `Sync` node plays the same role as `merge`. However, the `Load` node returns a memory value as additional result, and there is no equivalent to the `restrict` operation. Instead, several operations on disjoint parts of the memory may directly take as argument the same memory value. It is thus unclear whether the semantics of Firm could be formalized by assigning actual values to its "memory values", or if they should only be interpreted as a representation of dependencies between instructions.
Jung [51] proposes a formal model for an intermediate language where memory variables partition the memory. His language is inspired both by the functional stores of Steensgaard and by Firm. It features a `load` operator with the same signature as the `lookup` of Steensgaard, and a `split` operator with the same purpose as `restrict`, but which returns two memories partitioning the one given as argument. A notion of *realizability* relates semantically programs with several memories to their normalization as standard programs with a single memory. This semantic property is implied by a typing judgment.
Some SSA IRs [32] introduce variables representing memory locations. This may encode aliasing information if partitioning the memory is allowed. As a counter-example, the MemorySSA[8] virtual IR of LLVM only considers versions of the entire memory.

Logical invariants OptiTrust [15] is a framework to perform source-to-source transformations. In order to validate transformations relying on invariants on the shape of the memory, programs contain separation logic annotations. Those annotations describe both invariants (e.g., function contracts) and ghost transformations reorganizing the available resources. A typing algorithm checks the coherence of those annotations. Program transformations are applied interactively, using a set of trusted core transformations implemented in OCaml. Those transformations maintain the annotations.

2.4 Verified compilers and intermediate representations

Over the last two decades, there have been multiple formalizations of intermediate representations, machine-checked proofs of transformations and even implementations of formally verified compilers.

[8] https://releases.llvm.org/22.1.0/docs/MemorySSA.html

Those developments differ on several aspects, including their source languages, the guarantee provided, and the compilation algorithms formalized. This section is not exhaustive.

CompCert CompCert [69] is a formally verified C compiler. It supports a large subset of C, called CompCert C and whose semantics has been formalized during the development of CompCert. It can now compile programs to several assembly languages. It is defined as a pipeline composed of several passes between multiple intermediate languages. Those transformations include instruction selection, the translation of structured instructions into control flow graphs, a register allocation and the implementation of functions using a stack. CompCert also has optional optimizations. Most of them are performed on an intermediate representation called RTL (Register Transfer Language), where the bodies of functions are represented as control flow graphs of instructions.
CompCert formalizes the semantics of all its languages, in a small-step fashion. All semantics use the same logical model and interact with the environment using the same interface (external events and result code). The main correctness theorem relies on this second common point. It states that all behaviors of the target can be observed on the source (unless the source has undefined behaviors). This main theorem is actually obtained by combining simulations proved for each pass. To this end, each intermediate representation has a semantics, defined in a small-step fashion. Thanks to the determinism of the target assembly languages, the simulations are proven in a forward style. That is, CompCert proves for each pass that the target program simulates the source. This proof approach was considered easier than the use of backward simulations. We give a more detailed comparison with our approach for proving semantics refinement in section 4.6 and section 5.6. We also interface our semantic formalism with the small-step semantics of CompCert in section 5.5.4.

Variants of CompCert Multiple groups have forked CompCert, in order to study other compilation aspects while leveraging its infrastructure and support for the C language. Among those variants:

- CompCertSSA [12] adds a SSA representation at the level of RTL. It includes SSA construction and destruction passes and optimizations leveraging SSA invariants. We discuss SSA in more detail in section 6.1.
- CompCertS [10] gives a defined semantics to more source programs, notably those relying on the low-level representation of pointers. It also provides guarantees on the memory usage of the target.
- CompCertO [59] tackles the separate compilation of multiple components. They introduce a notion of "simulation conventions", which is close to the "relations on events" we will define to reason on semantics (section 4.6.4).
- CompCertTSO [92] gives formal guarantees on the compilation of concurrent programs relying on the relaxed shared memory model of x86, based on Total Store Ordering (TSO). A crucial difference with standard CompCert is the non-determinism of the semantics of concurrent programs. Yet, most proofs of CompCert were reused with only minor changes.
- Peek [79] is a framework for performing additional peephole optimizations at the level of assembly. Those transformations are performed in a semantics with a concrete memory model, different from the one used by standard CompCert.
- Chamois [45] is a variant of CompCert developed at Verimag. It supports additional back-ends and transformations. It also defines new IRs, in particular BTL (Block Transfer Language), which is interconvertible with RTL. BTL provides a more structured representation of programs, thanks to blocks containing sequences and conditionals. Chamois leverages defensive passes to simplify the proofs. That is, the compiler uses untrusted oracles, but their results are checked by formally verified validators.

  During my internship, I added annotations within BTL to propagate results of static analyses (either an interval analysis or an alias analysis). I amended the semantics to take those annotations into account using UBs. The information propagated is used both by untrusted oracles (so that they avoid describing incorrect transformations), and by a formally verified validator. The proof of the latter

ensures that it only accepts target programs which semantically refine the source. This refinement thus depends on the information encoded in the source and target, so the validator can use them to accept transformations that rely on that information. As a by-product, it checks that the properties encoded in the target are implied by those of the source.
This approach enabled the validation of several conditional transformations [45]: lazy loads and stores motions, an integer promotion triggering a later strength reduction and superblock scheduling. More recently, it also enabled optimizing memcopies [16]. However, we faced some of the limitations described in section 2.1. With immediate UBs we could no longer move annotated instructions above branches. We partially solved this issue by using deferred UBs, but this raised questions about situations where multiple occurrences of the same instruction (e.g. a load at a given address) had different annotations. The oracles (loads and stores motions) had to conservatively consider those instructions as distinct, meaning that they would miss optimizations that they would perform if no information was present.
This first attempt at propagating information, and more generally this experience on a verified compiler, motivated the subject of this thesis.

CompCert as back-end The pipeline of CompCert handles the compilation to assembly of multiple languages, with varying levels of abstraction. It is thus possible to use one of those languages as target to compile other source languages. Correctness properties are then derived by combining the proofs of CompCert with the correctness of the translation to the chosen representation of CompCert.

- Vélus [21] translates synchronous programs to the high-level representation Clight. We connect our development with Vélus, and will describe its relevant aspects in section 10.1.
- Capla [75] is a domain specific language for algebra libraries involving arrays. It features a verified back-end to another high-level representation of CompCert, namely Cminor. This language is also relevant for our work because it enforces a simplified version of the non-aliasing discipline of Rust.
- Schneider [91] defines a compilation pipeline using the low-level representation Linear of CompCert. This pipeline is centered around an IR supporting SSA, with a formalization based on a notion of *coherence*. Another aspect relevant for us is the use of instructions defined inductively, instead of CFGs like in most representations of CompCert. This enables proofs by induction on the syntax of programs, an approach we use in all our proofs.
- Barrière [8] implemented a formally verified Just In Time (JIT) compiler. This execution engine interleaves interpretation of the source program with more efficient native execution of parts compiled using CompCert (with a connection at the level of RTL [9]). The primitives inserted by the engine to enable speculative execution are formalized using a non-deterministic semantics.

Vellvm Vellvm is a formalization of some parts of the LLVM compiler [71], including LLVM IR, its SSA structure and its memory model. Its first iteration [106; 107] uses a non-deterministic operational semantics. Its second iteration [14; 105] instead uses a more modular denotational semantics based on Interaction Trees [102]. Section 4.6.1.3 compares this second approach to defining denotation semantics with our own.

CakeML CakeML[54; 60] is a verified compilation pipeline from a ML-like functional source language to assembly. This pipeline involves changes of the formal model of values and memory [54, D5]. CakeML uses big-step semantics and leverages the determinism of its languages to obtain termination-sensitive correctness properties.

2.5 Verifying programs with memory

Verifying programs written in standard stateful programming languages (e.g., C or OCaml) requires particular care about the memory. Indeed, any component of the program (e.g., function) can a priori access any part of the memory. One thus has to prove that they do not alter memory invariants

expected by other, distant components. One also has to take into account the possibility of aliased pointers. Multiple approaches tackle those challenges.

Memory as an array A first approach is to directly reason on the memory, as a map from pointers to objects of the language. For instance, this is the approach used by Dafny [67]. With this approach, specifications do not constrain by default the memory accesses. Instead, they must be manually strengthened to provide enough guarantees. A way of obtaining strong enough, yet modular specifications is to describe the set of memory locations modified by program components. By additionally maintaining non-aliasing invariants, one can then deduce the preservation of memory invariants of client codes. In Dafny, those two aspects are implemented by `modifies` clauses and disjointness invariants between the memory footprint of structures.

Separation logic In verification tools based on separation logic [87], the memory is split dynamically into several, disjoint "heaplets". Program specifications are then given using predicates on those heaplets. A separating conjunction operator ("$\star$") combines several predicates using a partition. That is, the composed predicate $p_0 \star p_1$ holds on a heaplet h, if and only if there exists a partition of h into two heaplets h_0 and h_1 such that p_0 holds on h_0 and p_1 on h_1. Program components implicitly preserve any heaplet disjoint with their pre-conditions. By default, separation logics thus restrict the set of memory locations that can be accessed by components (those in heaplets appearing in their pre-conditions, plus the locations they allocate), and also enforce non-aliasing properties between data-structures (thanks to the disjointness of the sub-heaplets in the separating conjunction).
Modern separation logic frameworks actually enable a finer-grained control. In particular, the notion of "disjointness" between heaplets is generalized to the validity of compositions in partial commutative monoids [49]. Those monoids can encode protocols controlling the access to shared resources by concurrent components. As a particular case, fractional permissions enable separating a heaplet into several heaplets providing read-only access to all its locations. For instance, this enables passing aliased data-structures as arguments to a function which only reads them.
The VST (Verified Software Toolchain [5]) defines a separation logic on top of the memory model of CompCert. It enables formal proofs of properties on C programs, with guarantees that those specifications are also satisfied by the assembly programs generated by CompCert. Concurrent programs are supported, although CompCert has only a semantics of sequential programs. This is possible thanks to the representation of synchronization primitives using external calls in the semantics of CompCert. An "oracle semantics" [5, chapter 44] abstracts the effects of other threads during those external calls. Those external calls also change the memory permissions of the thread, to ensure that CompCert does not introduce accesses interfering with resources owned by other threads. Those permissions are chosen using angelic non-determinism, because they depend on invariants of the separation logic proof.

Translation to functional languages Creusot [37; 77] and Aeneas [47] are verification tools which leverage the non-aliasing guarantees of the safe fragment of Rust to avoid reasoning on a global state. References are instead directly represented as values. Those verification frameworks thus encode programs written in an imperative language with a global memory into functional languages. This encoding is the inverse of the compilation described by Wadler [100] from functional to imperative languages.

3

Frag IR: an intermediate representation to propagate information

We define a new intermediate representation, called Frag IR, which enables propagating semantic information in a verified compilation pipeline. This chapter describes the main ideas used for this purpose, and how they influence the language. We implemented the language, its meta-theory and all compilation passes in Rocq. For now, we stay at an informal level of description, and only give the formal definition of the language in chapter 8. In particular, our examples can be read as if they were written in a typed functional language where operators and function calls are applied to types and variables and may return several values.

Frag IR features two mechanisms to represent information:
- Propositional variables with dependent types represent relations between the values of several program variables (section 3.1).
- Memory fragments syntactically reflect the structure of the memory and the dependencies between the accesses (section 3.2).

We use Frag IR as a middle-end representation in a compilation pipeline whose target is a low level language executable by interpretation (section 3.3).

3.1 Tracking assumptions with dependent types

Examples such as figure 2.1 motivate us to search for a representation of information which specifies the hypotheses under which it is valid. In particular, we avoid a purely semantic approach based on UBs in order to avoid implicit dependencies on the control flow. Instead, we rely on a static approach which uses ad hoc dependent types. Section 3.1.1 recalls the key ideas behind static type systems and their interest for propagating information in a compilation pipeline. Section 3.1.2 introduces the notion of propositional variables, which are used to represent information in Frag IR. Dependencies of information are represented syntactically thanks to "`lemma`" instructions (section 3.1.3). We still provide an "`assert`" instruction as a way of encoding information using UBs (section 3.1.4).

3.1.1 Static justification of information with type systems

In static type systems, types are annotations attached to static elements, such as variables or functions. They describe properties of the dynamic elements associated with them. For instance, we will give the type `i64` to variables which only contain 64-bit integers during executions. Types thus provide infor-

mation which can be useful for transformations. For instance, the RTL intermediate representation of CompCert has a type system which assigns types (e.g. `int` or `float`) to registers. Those types give information about the size of the values the registers may contain. This information is used by program transformations to know which number of bytes (specified by a "chunk" in CompCert) must be used to store the value in memory without loss. This notably allows the register allocation pass [69, §8] to spill registers to the memory.

Static type systems come with a *typing judgment*. This judgment is a static criterion on programs which ensures that information described by the types is valid during its executions. For instance, if an assignment `n = e` syntactically occurs in the program, then the judgment will require the variable `n` to have a type containing all values that can be produced by the expression `e`. The typing judgment is an invariant of compilation pipelines. Transformations can thus assume that the information described by the types is true. Like the absence of UBs, they must preserve this property. However, contrary to UBs, the typing judgment is defined statically and thus does not depend implicitly on the control flow. Another advantage of some type systems compared to UBs is that verifying that a program is well-typed is sometimes a decidable problem. That is, one can implement a type-checking procedure which accepts its input program if and only if it is well-typed. Type-checking the programs increases the confidence in the information provided by the types. In a formally verified setting, a type-checker may replace a formal proof that the transformation preserves the typing judgment.

Dependent type systems allow the types to refer to the value of program variables. The dynamic properties represented by those types depend on the values taken by the variables during the executions considered. For instance, in F* [84], a variable `i` with type `i:uint32{i < n}` may only contain integers smaller than the value stored in the variable `n`. Dependent types increase the expressivity of type systems. Indeed, they enable the representation of properties involving several variables.

3.1.2 Propositional variables with dependent types

In Frag IR, each variable has a type. Programs perform computations using scalar variables. Those scalar variables have standard, non-dependent types: machine integers (e.g. `i64`), floating-point numbers (e.g. `float32`), Booleans (`bool`) and pointers (`ptr`). More precise information known about variables is represented using separate variables, called *propositional variables*. The types of those propositional variables are called *propositions*. Those types depend on one or several variables, and describe a property between their values. For instance, in figure 3.1, the propositional variable `PRE` has type `n > 0`. This variable represents the information that during any execution, the variable `n` stores a positive integer. We interpret the types of propositional variables using the Curry-Howard isomorphism: a proposition is a type inhabited if and only if the property it describes is satisfied. If they are satisfied, our propositions have exactly one inhabitant. We are not interested in the value of propositional variables, but having such variables in the context provides information about the values of the other variables they depend on. They can thus be used instead of UBs. For instance, figure 3.2 gives the signature of an unsigned division operator `pre_u_div`. This operator takes an additional argument `PRE` with a proposition `not_error` which ensures that the division is well-defined (that is, that the divisor `m` is not zero). Thanks to this argument, the division becomes a total operator, but is still never performed with a zero divisor. In order to reflect in the type system information about the results of computations, operations return an additional result `POST` with a proposition type. For the same reason, a branch on a guard `b` introduces a variable (e.g. `Hb` in figure 3.3) with type `b = true` in the `then` branch and `b = false` in the `else` branch. Note that the information represented by a propositional variable is only available at program points where the propositional variable is in the context. In other words, the information is only valid in the scope of the propositional variable. For instance in figure 3.3, `Hb : b = true` can only be used in the `then` branch.

```
fn foo(n: i64, ghost PRE:  n > 0) : (m: i64, ghost POST: m > 0) {…}
fn bar(n: i64, ghost H0: n > 1) : (m: i64) {
  (* fcall foo(n, H0) is ill-typed *)
  let ghost H1: n > 0 = lemma{ (n': i64, H0': n' > 1) : (H1': n' > 0) }(n, H0);
  let m, _ = fcall foo(n, H1);
  return (m)
}
```

Figure 3.1: A program involving propositional variables (section 3.1.2) and an implication between pre-conditions with `lemma` (section 3.1.3) [compiler/test/src/example/lemma_impl_pre.frag]

```
add{i64}(n: i64, m: i64)
  : (r: int, ghost POST: r = n + m)

pre_u_div{i64}(n: i64, d: i64, ghost PRE: not_error (u_div{i64} n d))
  : (r: i64, ghost POST: r = u_div{i64} n d)
```

Figure 3.2: Signatures of two operators of Frag IR

Propositional variables only represent information, they do not influence the computations performed by programs. They are thus an instance of *ghost variables*. Our compilation pipeline ultimately removes them. To guide this transformation, we annotate propositional variables with an attribute "`ghost`". Despite their particular usage, propositional variables are treated exactly like standard variables in the syntax, typing and semantics. In particular, they can appear as arguments or results of functions (e.g. `foo` in figure 3.1). They then act as pre- and post-conditions and propagate inter-procedural information.

3.1.3 Deriving properties with lemmas

Typing judgments for programs in Frag IR do not take into account the semantics of propositions; like all types, propositions are compared using syntactical equality. For instance, in figure 3.1 the propositional variable `H0 : n > 1` cannot be given as an argument to the function `foo` which expects a pre-condition `n > 0`. Instead we provide an operation `lemma`, which derives new propositions from existing ones. A lemma is described by a closed signature, which defines the type of its formal arguments (`n'` and `H0'` in figure 3.1) and of its formal results (`H1'`). It is applied to variables defined in the context (`n` and `H0`) and whose types match the types of the formal arguments. Its actual results (`H1`) add derived information to the context. To avoid stuck lemmas while ensuring the validity of the information represented in the type system, the propositions returned by a lemma must be satisfied whenever the lemma is executed. To ensure this, we associate a *soundness condition* to each lemma.

Definition 3.1 A lemma is sound if for any well-typed values for its formal arguments, there exists values having the types of its formal results.

Since propositions are inhabited if and only if they are satisfied, definition 3.1 describes an implication between the propositions taken as arguments and those returned, with a universal quantification on the values of the other arguments. For instance, the soundness condition of figure 3.1 is equivalent to:

$$\forall n'.\, n' > 1 \Rightarrow n' > 0$$

Note that in definition 3.1, only the propositions taken as arguments by a lemma can be used to justify its soundness condition. For instance, the lemma of figure 3.1 would become invalid if we removed its argument, `H0'`, even if `H0` was still in the context. In fact, since lemmas have closed signatures, their soundness is independent of the context where they are used and of the semantics of the program. The soundness of all lemmas is a well-formedness property of programs, and we maintain it as a formal invariant of the compilation pipeline. That is, the producer of a program must ensure that all lemmas are sound and transformations assume and preserve this property.

```
predicate P
fn foo(b: bool, n: i64, ghost H0: prop{P} b n) : () {
  if b Hb then {
    let ghost H1: n < 9 = lemma{…}(b, n, H0);
    let ghost H2: n > 0 = lemma{…}(b, n, H0, Hb);
    …
  }
}
```

Figure 3.3: Combination of `lemma` instructions and abstract `prop` types [compiler/test/src/transform/lemma_branch_dep.frag:6]. The dependencies of the lemmas reflect that `n < 9` is true regardless of the value of `b` whereas `n > 0` may depend on hypothesis `Hb : b = true`.

Isolating with lemmas the derivation of information has several benefits:

- Since lemmas can only use the propositions taken as arguments, the dependencies of information are explicit. In particular, lemmas do not depend implicitly on the control flow. For instance in figure 3.3, a compiler could move the first `lemma` above the branch to obtain the information $n < 9$ earlier.
- The undecidable implications between propositions are captured with the soundness condition on lemmas. The typing judgments rely only on syntactical equality. This allowed us to implement a type-checker. Producers of programs can call it instead of proving that the programs they generate are well-typed.

Since type-checking does not depend on the semantics of propositions, we increase their expressivity by allowing abstract propositions, whose semantics is not expressed with the syntax of our types and unknown to the compiler. Those abstract propositions are syntactically represented as `prop{P} args` where `P` is a predicate identifier (declared at top-level) and `args` are expressions which have access to the variables in the context. Formally, the semantics of those predicates are chosen by the producer of the program, as functions of the evaluations of their arguments. The soundness of the lemmas must be justified with respect to the chosen semantics of the predicates. Abstract propositions thus enable the use of arbitrary logical predicates, while keeping a syntactical representation of the dependences of information. For instance in figure 3.3, the type of `H0` is an abstract proposition. To justify the validity of the program, we interpret the predicate `P` as:

$$\texttt{P b n} \quad := \quad \texttt{n} < 9 \wedge (\texttt{b} = \texttt{true} \Rightarrow \texttt{n} > 0)$$

The soundness conditions of the two lemmas are then:

$$\forall \texttt{b n}.\, (\texttt{n} < 9 \wedge (\texttt{b} = \texttt{true} \Rightarrow \texttt{n} > 0)) \Rightarrow \texttt{n} < 9$$
$$\forall \texttt{b n}.\, (\texttt{n} < 9 \wedge (\texttt{b} = \texttt{true} \Rightarrow \texttt{n} > 0)) \wedge \texttt{b} = \texttt{true} \Rightarrow \texttt{n} > 0$$

and are satisfied.

3.1.4 Assertions to recover information using Undefined Behaviors

In some cases, one may want to justify information semantically instead of deriving it statically using lemmas. Either because it is too complex to express the necessary invariants or because one only has access to a semantic justification, for instance when compiling a language with UBs. For this purpose, Frag IR includes an operation `assert`, which reflects in the type system a property *semantically* known to be valid at some program point. Given a proposition `P`, `assert<P>()` returns a propositional variable of type `P`. However, this operator must only be used at program points where `P` holds during any execution. The semantics of `assert<P>()` is to return the value inhabiting `P` if `P` is satisfied and to abort the execution by raising a UB otherwise. Like in some verification tools [66], `assert` is not a dynamic check and instead requires the producer of the program to prove that `P` always holds at the program point. In particular, the compiler can remove assertions if their result is not used, although this erases information.

```
let ghost H: not_error(u_div{i64} n d) =
  assert<not_error (u_div{i64} n d)>();
let r: i64, _ = pre_u_div{i64}(n, d, H)
if c _
then fcall f(r)
```

↔

```
let ghost H: not_error(u_div{i64} n d) =
  assert<not_error (u_div{i64} n d)>();
if c _ then {
  let r: i64, _ = pre_u_div{i64}(n,d,H);
  fcall f(r)
}
```

Figure 3.4: Reversible move of a division below a branch [compiler/test/src/transform/div_branch_comm.frag]

Figure 3.4 illustrates the encoding of a partial division operation using `assert`. Even if `assert` relies on UBs, this encoding prevents the loss of information when performing transformations compared to a standard approach where the division itself has UB. Indeed, our encoding separates the point where the information is obtained with `assert` from its use by `pre_u_div`. As a consequence, it is possible to move the division operation within a branch while keeping the information that it is well-defined unconditionally. This transformation is thus reversible, contrary to the transformation of figure 2.1. However, contrary to `lemma` or deferred UBs, `assert` cannot be moved above branches.

3.2 Controlling aliasing with memory fragments

We now focus on information useful to transform memory accesses. Two kinds of information are relevant:

- Because programs are only allowed to access parts of the memory that they allocated, most memory operations are partial. Speculating them thus requires information about the validity of accesses.
- Because of the indirection provided by pointers, non-aliasing information is needed to reason on memory operations and to transform them.

Information about memory accesses is particularly valuable because it often relies on inter-procedural invariants and can only partially be found by alias analyses.

Aliasing information is challenging to encode because it relates many dynamic addresses. Steensgaard [96] showed that it could be encoded by fragmenting the memory using functional stores. Following this approach, programs in Frag IR operate over multiple fragments which are explicitly represented with variables instead of a single global memory. We describe the main ideas of this encoding on an example (section 3.2.1) before specifying the semantics of the fragments and of operations on them.

3.2.1 Usage of memory fragments

Figure 3.5a describes a function which increments the values stored at two non-aliasing memory locations. The function takes as arguments not only pointers containing the addresses of the locations but also cells representing the fragments of the memory where the integers are stored. Memory fragments store values and permissions for a subset of the memory. For instance, the variables with type `cell{i64} p0` associate values to the addresses in the range $[\texttt{p0}; \texttt{p0} + \texttt{sizeof int64})$; where `sizeof t` is the number of bytes used to encode a value of type `t`. Instead of accessing a global state, all memory operations take as argument or return memory fragments. Memory fragments are immutable, hence the write operation `Frag.write` creates a new cell storing the value given as argument and the function `incr2` returns the cells containing the updated values.

Transformations enabled by memory fragments A key property of Frag IR is that, thanks to the absence of a global state, memory operations have explicit dependencies. Indeed, syntactically distinct memory fragments do not alias. For instance in figure 3.5a, the accesses at addresses `p0` and `p1` operate on distinct memory fragments. They are thus semantically independent. In particular, one could commute the write on `p0` with the read on `p1` to parallelize the accesses. In the equivalent C function (figure 3.5b), this transformation would require the inter-procedural information that pointers

```
fn incr2(p0: ptr, p1: ptr,
         ghost c0a: cell{i64} p0,
         ghost c1a: cell{i64} p1)
      : (ghost c0b: cell{i64} p0,
         ghost c1b: cell{i64} p1) {
  let       C,   _ = 1;
  let       x0a, _ = Frag.read{i64}(p0, c0a);
  let       x0b, _ = add(x0a, C);
  let ghost c0b, _ = Frag.write{i64}(p0, c0a, x0b);
  let       x1a, _ = Frag.read{i64}(p1, c1a);
  let       x1b, _ = add(x1a, C);
  let ghost c1b, _ = Frag.write{i64}(p1, c1a, x1b);
  return (c0b, c1b)
}
```

a: Frag IR

```
void incr2(int* p0, int* p1)
{


  int C = 1;
  int x0a = *p0;
  int x0b = x0a + C;
  *p0 = x0b;
  int x1a = *p1;
  int x1b = x1a + C;
  *p1 = x1b;

}
```

b: C

Figure 3.5: Functions which update two non-aliasing memory cells

do not alias. Another interesting property is that `Frag.read` is total. It is thus possible to speculate it. We give more examples of transformations involving memory fragments in section 3.2.8.

Implementation with a global memory Memory fragments are ultimately implemented with a global memory. Our compilation pipeline contains a translation pass, which replaces all operations on memory fragments by corresponding operations on the global memory. We need to add restrictions to Frag IR to ensure that it is possible (section 3.2.5). Indeed, the translation requires a linear usage of memory fragments [100]. In particular, the fragments given as arguments to `Frag.write` are considered as "consumed" and must not be used afterwards. Note that this condition is respected for `c0a` and `c1a` in figure 3.5a.

3.2.2 Separating disjoint parts of the memory with heaplets

To define the semantics of Frag IR and to reason on it, we need a formal representation of "subsets" of the memory. Drawing inspiration from separation logic [87], we use heaplets. Before defining them, we introduce our representation of pointers and related definitions:

Definition 3.2

- Our *pointers* are 64-bit machine integers, that is, natural numbers in the range $[0; 2^{64})$. Their set is `Ptr.t`.
- The *null* pointer, `Ptr.null` : `Ptr.t` is the machine integer 0. No data is ever allocated at this address, so programs use it to represent errors or the absence of values.
- A range with base pointer `p` : `Ptr` and size `n` : $\mathbb{N}$ is *valid* if `p` is non-null and $\texttt{p} + \texttt{n} < 2^{64}$. We write `valid_range p n` if it is the case. The second condition ensures that the addition does not wrap around in the modular arithmetic on pointers. Our development currently does not take into account alignment constraints.

We now define our type of heaplets:

Definition 3.3 A *heaplet* is either invalid ($\bot$) or defined by a partial function from pointers to bytes and a set of permissions to deallocate memory ranges:

$$\texttt{heaplet} ::= \bot \mid (\texttt{data}, \texttt{perm})$$

where `data` : `Ptr.t` $\rightharpoonup$ `Byte.t` and `perm` : `Ptr.t` $\rightharpoonup \mathbb{N}$ are partial functions. The partial function `perm` associates deallocation permissions to base pointers returned by our `malloc` primitive, that is, to pointers that can be given as arguments to `free`. A heaplet contains the permission to deallocate a range $[\texttt{p}; \texttt{p} + \texttt{n})$ if and only if $\texttt{perm p} = \texttt{n}$.

Compared to the structures used in modern separation logics (section 2.5) this type of heaplet is rather simple. In particular, it does not allow representing read-only permissions. Permissions to deallocate memory ranges are stored in a partial map; this enforces that for any address `p`, a heaplet contains at most one deallocation permission for a range starting at `p`.

Example 3.1

- `emp` $:= (\varnothing, \varnothing)$ is the empty heaplet containing no data or permissions. Note that it is distinct from the invalid heaplet $\bot$.
- If `p` is a pointer, and $(\texttt{bs}_i)_{0 \leq i < n}$ a list of `n` bytes, then `hrange p bs` is:
  - invalid if `valid_range p n` does not hold
  - otherwise, equal to $(\texttt{data}, \varnothing)$, where `data` has domain $[\texttt{p}; \texttt{p} + \texttt{n})$ and is defined by $\texttt{data}\ (\texttt{p} + \texttt{i}) = \texttt{bs}_i$.

Heaplets that describe distinct parts of the memory can be merged using a disjoint union operation:

Definition 3.4 Two valid heaplets $(\texttt{data}_0, \texttt{perm}_0)$ and $(\texttt{data}_1, \texttt{perm}_1)$ are *disjoint* if their partial functions are, that is if:

$$\texttt{dom data}_0 \cap \texttt{dom data}_1 = \varnothing \quad \wedge \quad \texttt{dom perm}_0 \cap \texttt{dom perm}_1 = \varnothing$$

The invalid heaplet is not disjoint with any heaplet.
Disjoint union is a total binary operation on heaplets, defined from the disjoint union of partial functions by:

$$\begin{aligned} \texttt{data}_0, \texttt{perm}_0 \uplus \texttt{data}_1, \texttt{perm}_1 &:= (\texttt{data}_0 \uplus \texttt{data}_1, \texttt{perm}_0 \uplus \texttt{perm}_1) && \text{if the heaplets are disjoint} \\ \texttt{h}_0 \uplus \texttt{h}_1 &:= \bot && \text{otherwise} \end{aligned}$$

The purpose of the invalid heaplet ($\bot$) is to define the disjoint union as a total function. It is the absorbing element of the operation: $\bot \uplus \texttt{h} = \texttt{h} \uplus \bot = \bot$. The empty heaplet is the neutral element: $\texttt{emp} \uplus \texttt{h} = \texttt{h} \uplus \texttt{emp} = \texttt{h}$.

3.2.3 Storing the state of the memory in memory fragments

Rather than having a single mutable global memory, Frag IR programs use several disjoint, immutable memory fragments. Memory fragments are values stored in variables but with a footprint representing their implementation in the memory. Formally, this footprint is given as a heaplet and we define a total function `footprint` : `val` $\to$ `heaplet`. Values which are not memory fragments are mapped to the empty heaplet. We generalize `footprint` to lists of values, as the disjoint union of their footprints. There are two kinds of memory fragments:

- Unstructured fragments are directly defined by a heaplet. Some primitive types are inhabited by them. For instance `mrange p sz` is inhabited by memory fragments containing exactly the addresses $[\texttt{p}; \texttt{p} + \texttt{sz})$. Similarly to abstract propositions, abstract types of memory fragments may be defined by predicates chosen by the producer of the program (e.g., `mem{linked_list}` in figure 3.11). Those predicates take as argument a heaplet and thus correspond to the notion of assertion in separation logic. More generally, any closed type `T` defines a set of heaplets, as the footprints of its inhabitants: $\{\texttt{footprint v} \mid \texttt{v}$ inhabits $\texttt{T}\}$.
- To avoid reasoning on the partial maps contained in heaplets and to simplify the semantics, Frag IR also provides structured fragments. Those values are trees whose leaves store scalar values or padding heaplets. Their purpose is similar to the `struct`s of C. They inhabit types which describe their memory layout. Structured fragments are shaped according to their type. For instance, figure 3.6 represents the shape of inhabitants of the type `{32 : 0 cell{i64}; 8 {16 : 0 cell{i64}; 8 cell{i64}}} p`. This type describes a 32-byte-long structure at base address `p`. At offset 0, it contains a field storing an `i64`. At offset 8 (i.e. right after its first field), it contains a 16-byte-long

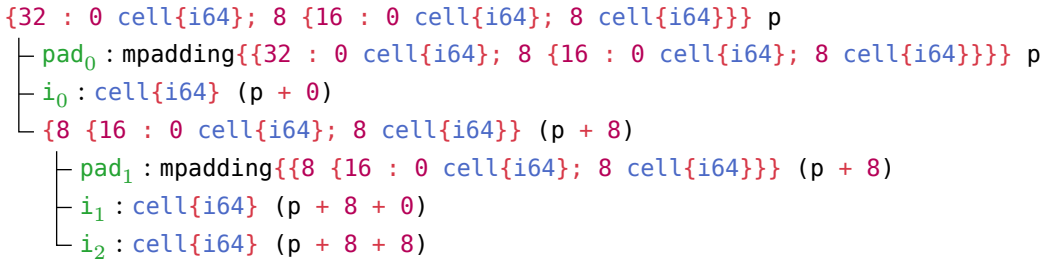


Figure 3.6: Shape of a value inhabiting the type `{32 : 0 cell{i64}; 8 {16 : 0 cell{i64}; 8 cell{i64}}} p`. The leaves i_0, i_1 and i_2 are 8-byte-long encodings of 64-bit machine integers. The paddings are heaplets: pad_0 contains values for the addresses $[p + 24; p + 32)$ and pad_1 is empty.

sub-structure of type `{16 : 0 cell{i64}; 8 cell{i64}}` $(p + 8)$. Each internal node also has exactly one padding heaplet as child. This heaplet contains the bytes not covered by its fields. For instance, pad_0 contains bytes for the addresses $[p + 24; p + 32)$. In the general case, those "`mpadding`" heaplets are not continuous and can be empty (like pad_1 in figure 3.6). For instance, there are two padding segments $[1; 3] \cup [5; 7]$ in `{8: 0 cell{i8}; 4 cell{i8}}`.

The types `cell{t} p` are particular cases of structured types, where the trees are reduced to a leaf containing a scalar of type `t`. The footprint of a cell is a heaplet `hrange p bs` containing a byte encoding `bs` of the scalar. In the general case, the footprint of a tree is the union of the footprints of its leaves. The paddings ensure that, for a structure of size `n` at address `p`, this union yields a heaplet which contains data for the addresses $[p; p + n)$.

In Frag IR, programs operate both over pointers and memory fragments. Most operations on structured fragments (which will be described in section 3.2.4) could be defined without the pointers. In the semantics of Frag IR, pointers are used to describe transformations involving unstructured fragments (section 3.2.7). In all cases, pointers are needed to translate the operations on memory fragments into operations on a global memory. The dependent types of the memory fragments link them to the pointers used to access them.

Like for the propositional variables, there is no special handling of variables storing memory fragments in the semantics and typing of Frag IR. Contrary to the values of propositional variables, memory fragments are used in the semantics of Frag IR. However, once the operations on memory fragments are translated into operations on the global memory, the variables storing memory fragments no longer influence the computations performed by the program. We thus erase them at the same time as the propositional variables. In Frag IR, we already annotate variables storing memory fragments as `ghost`. Note however, that those variables are not ghosts at this level. Indeed, they influence the semantics of the programs.

3.2.4 Operations on memory fragments

Frag IR provides the equivalents of standard memory operations: read, write and dynamic allocation. Figure 3.7 describes their signatures. Instead of interacting with a global state, they take as argument and return memory fragments. Like for the functional stores used by Steensgaard [96], read and write operations take as arguments both a pointer `p` and a memory fragment `c` which represent a cell at this address. The operation `Frag.read` retrieves the value `r` from its cell argument, its post-condition reflects the fact that `r` is the content `*c` of the cell. Since memory fragments are immutable, `Frag.write` returns a new cell `c'` at the same address `p` but which now stores the value `v` given as argument. Programs allocate memory fragments with `Frag.malloc`, which takes as argument the size `sz` of the range to allocate and returns a pointer `p` and an unstructured memory fragment of type `malloc p sz`. Its semantics is non-deterministic, it either fails and returns the null pointer and an empty fragment,

```
Frag.read{t}(p: ptr, ghost c: cell{t} p)
  : (r: t, ghost POST: *c = r)
Frag.write{t}(p: ptr, ghost c: cell{t} p, v: t)
  : (ghost c': cell{t} p, ghost POST: *c' = v)
Frag.malloc(sz: i64) : (p: ptr, ghost m: malloc p sz)
Frag.free(p: ptr, ghost sz: i64, ghost m: malloc p sz) : ()
```
section 3.2.4

```
Frag.joinable(ghost args…) : (ghost POST: joinable args…)
```
section 3.2.6

```
Strc.make{{sz : ofs_i field_i;…}}
    (ghost p : ptr, ghost pad : mpadding{{sz : ofs_i field_i;…}} p,
     ghost f_i : field_i (p + ofs_i),… )
  : (ghost s : {sz : ofs_i field_i;…} p,
     ghost POST : s = (mkstrc pad f_i…))
Strc.elim{{sz : ofs_i field_i;…}}
    (ghost p : ptr, ghost s : {sz : ofs_i field_i;…} p)
  : (ghost pad : mpadding{{sz : ofs_i field_i;…}} p,
     ghost f_i : field_i (p + ofs_i),…
     ghost POST : s = (mkstrc pad f_i…))
```
section 3.2.7

Figure 3.7: Signatures of some operations on memory fragments

```
1 let c1 : cell{i64} p, _ = Frag.write{i64}(p, c0, 1);
2 let c2 : cell{i64} p, _ = Frag.write{i64}(p, c1, 2);
3 let r  : i64,         _ = Frag.read{i64}(p, c1);
4 output{i64}(r);
```
output: 1
a: Frag IR

↛

```
Flat.write{i64}(p, 1);
Flat.write{i64}(p, 2);
let r = Flat.read{i64}(p);
output{i64}(r);
```
output: 2
b: Flat IR

Figure 3.8: A program which does not satisfy the linearity condition, and whose behavior is altered by the implementation of fragments with a global memory. [compiler/test/src/pass/translation/ko_4_overwritten.frag]

or returns an arbitrary non-null pointer with a fragment containing arbitrary values for the range $[p; p + sz)$ and the permission to deallocate this range. The semantics of the type `malloc` reflects those two cases. Conversely, `Frag.free` represents the deallocation of a fragment of type `malloc p sz`, semantically it is a no-op.

TODO (rocq)

utilisation d'un type malloc sans sz

3.2.5 Linearity condition

Our compilation pipeline ultimately implements the memory fragments using a global memory. We thus generate target programs in another representation, called Flat IR. This IR extends Frag IR with a global memory and standard operations to access it (in particular `Flat.read` and `Flat.write`). We replace the operations `Frag.read`, `Frag.write`, `Frag.alloc` and `Frag.free` by corresponding operations on the global memory. However, without restrictions, this translation is unsound. Consider the translation in figure 3.8. In the source Frag IR program (a), memory fragments are immutable. At line 2 `Frag.write` thus does not change the value stored in `c1`. At line 3, `c1` thus still contains the value written at line 1, so `Frag.read` returns `1`. However, in the target Flat IR program (b), at line 2 `Flat.write` overwrites the value that was written in the global memory at line 1. At line 3, `Flat.read` thus returns the value written at line 2, that is `2`. The translation thus changes the behavior of this program.

```
let c1 : cell{i64} p, _ = Frag.write{i64}(p, c0, 0);
let c2 : cell{i64} q, _ = lemma {
    (p : ptr, q : ptr, EQ : p = q, c : cell{i64} p)
  : (c' : cell{i64} q)
    consume (c)
  }(p, q, EQ, c1)
let c3 : cell{i64} q, _ = Frag.write{i64}(q, c2, 1);
```

a: Frag IR

→

```
Flat.write{i64}(p, 0);
Flat.write{i64}(q, 1);
```

b: Flat IR

Figure 3.9: A program involving aliased pointers. The generalized `lemma` operation (described in section 3.2.7) is used to cast the fragment between equivalent types. [compiler/test/src/pipeline/aliased_cells]

To ensure the soundness of the translation, we require Flat IR programs to use memory fragments linearly [100]. Our linearity condition considers that the fragments given as arguments to `Frag.write`, `Frag.free` and function calls are consumed and must not be used later. Reading a cell does not consume it, but consumed cells can no longer be read. Formally, we express this condition by associating to each program point a static set of *active* variables. This set contains the variables storing memory fragments that are available in the context and which have not yet been consumed. For instance in figure 3.8a, assuming that at the beginning of line 1 `c0` is the only active variable, then at line 3 the set of active variables is {`c2`}. Since `c1` is not active, our linearity criterion rejects the `Frag.read`.

We emphasize that the linearity criterion constrains the memory fragments, not the pointers used to access them. It is thus possible to encode programs with aliasing, where several pointers are used to access the same data. For instance, in figure 3.9, `p` and `q` point to the same address. This is reflected in the type system using a proposition `EQ : p = q`. This proposition is used in Frag IR to change the type of the fragment which stores the data at their common address (using a generalization of `lemma`, described in section 3.2.7). At any program point there is only one active fragment containing the value of this address: `c0` then `c1`, `c2` and `c3`.

The linearity criterion does not affect the semantics or typing of programs. It is thus possible to reason about programs that do not use fragments linearly. Our approach is to maintain linearity as an informal invariant during the compilation pipeline and to validate it defensively during the translation to a global memory. Since memory fragments do not alias in the semantics of Frag IR, formally verified transformations are still able to rely on the separation properties encoded by the fragments.

3.2.6 Non-disjointness of the fragments

Since `Frag.malloc` does not access a global state, the non-deterministic choice of the returned address is not constrained by other existing allocations. That is, in figure 3.10, although `p1` is allocated line 2 after `p0` and before its deallocation, the semantics allows `p1` to take the same value as `p0`. This may be surprising, but it avoids implicit dependencies between allocations and provides opportunities for optimizations. For instance, if `m0` and `m1` are used in non-overlapping ranges, they could be allocated at the same location. In section 3.2.8, we will also see that this semantics enables removing unused allocations despite our use of concrete pointers.

Memory accesses still have a well-defined semantics when several fragments contain the target address, because the operators take as argument the fragment used. However, we cannot always track statically the source of each cell fragment. That is, for most data-structures, we actually need the pointers to find which data is accessed. Concretely, the structured fragments only represent the simplest memory layout that programs may use. More complex layouts must be encoded by storing several pointer-indexed components of a data-structure in the same unstructured fragment (that is, heaplet). Recovering individual components is thus only possible if they have distinct addresses. For

```
1 let p0, ghost m0 = Frag.malloc(sz);
2 let p1, ghost m1 = Frag.malloc(sz);
3 – This point can be reached with p0 = p1 ≠ null
4 let ghost H = Frag.joinable(m0, m1);
5 – At this point p0 = p1 = null ∨ p0 ≠ p1
6 Frag.free(p0, sz, m0); Frag.free(p1, sz, m1);
```

Figure 3.10: A Frag IR program which performs two memory allocations with overlapping live ranges [compiler/test/src/transform/malloc.frag:43]

instance, all cells of a linked-list will at some point be stored in the same memory fragments. They thus need to have distinct addresses, even if they are allocated using several `Frag.mallocs`.

We thus provide a way for programs to require the disjointness of memory fragments, from which pointer inequalities can be deduced. We make such requirements explicit, with operations `Frag.joinable` (figure 3.7). This operator takes arbitrary arguments `args` and ensures that their footprints are disjoint (or equivalently that their union yields a valid heaplet, hence its name). This property is reflected with the proposition `joinable args` as post-condition. The semantics of `Frag.joinable` is a particular case of `assume` instructions that can be found in verification languages [66], and of a semantics `any` we formalized in `SMonad` (section 4.2.4.1). That is, it is the opposite of a UB: it is a guarantee provided by the compiler. Formally, executions that reach a `Frag.joinable` operation with non-disjoint arguments are ignored when considering the set of behaviors of a program. However, it does not cancel previous UBs or external events. For instance in figure 3.10, an assertion `p0 = p1 = null ∨ p0 ≠ p1` would still generate a UB line 3, but not line 5. Hence, operations `Frag.joinable` cannot be commuted below instructions which may have UB, like `assert`. Moreover, `Frag.joinable` restricts the set of valid compilations of a program. In particular, it could prevent the elimination of dead allocations.

In the linearity criterion, the arguments of `Frag.joinable` must be active, but are not consumed by the operation. We justify the disjointness of the fragments during the translation to a global memory, using the fact that they are simultaneously active. The operator `Frag.joinable` is thus propagated in the compilation pipeline until this pass. Although we do not currently have automatic optimizations on memory allocations, such transformations would have to take `Frag.joinable` into account to avoid breaking the linearity condition. We expect that this would restrict the optimizations, when needed to preserve the disjointness assumptions requested by the source program.

3.2.7 Ghost transformations on memory fragments with generalized lemmas

We need a way to split and merge memory fragments. Steensgaard [96] introduces two operations for this purpose: `merge` and `restrict`. For instance, when encoding linked-lists, we need to merge several list cells to obtain an unstructured fragment containing a whole list. Conversely, to access a specific cell (e.g. the head of the list), we need to split it from the rest of the list, that is, obtain two fragments: one containing only the accessed cell and another containing the rest of the list.

Merging several fragments is easily done in a deterministic way using the disjoint union of heaplets. Splitting memory fragments is less obvious. Indeed, we need to choose which part of the original fragment goes into each resulting fragment. This choice depends on how the program will use those results. We ruled out the use of angelic non-determinism [39] because, like UBs, they would implicitly depend on the whole program execution that precedes them. In particular, they would not commute with demonic non-deterministic choices, that are used in the semantics of `Frag.malloc`. To avoid angelic non-determinism, we assume that the types of the memory fragments resulting from a split are restrictive enough. That is, that the continuation of the program accepts any choice of results compatible with those types. This enables using demonic non-determinism.

```
let ghost JOINABLE : joinable hd tl = Frag.joinable(hd, tl);
lemma {
    (p0 : ptr, p1 : ptr, p2 : ptr,
     hd : mem{list_cell} p0 p1, tl : mem{linked_list} p1 p2,
     JOINABLE : joinable hd tl)
  : (ll : mem{linked_list} p0 p2)
    consume (hd, tl)
  }(p0, p1, p2, hd, tl, JOINABLE)
```

Figure 3.11: Snippet of a toy program which uses singly linked lists [compiler/test/src/example/singly_linked_list.frag:179], also described in section 8.4.3. The lemma merges two memory fragments representing a list cell `hd : mem{list_cell} p0 p1` at address `p0` pointing to `p1` and a linked list segment `tl : mem{linked_list} p1 p2` starting at `p1` into a linked list at address `p0`.

In Frag IR, programs use the same operation to merge and split fragments. This operation is a generalization of the `lemma` presented in section 3.1.3. We generalize its semantics so that it can consume a set of memory fragments to produce an equivalent set of new ones. For instance a memory range can be split using a `lemma` with the following signature [compiler/test/src/example/range.frag:8]:

```
    (p: ptr, sz: i64, a: i64, r: mrange p sz, H: a ≤ sz)
  : (r0 : mrange p a, r1 : mrange (p + a) (sz - a))
    consume (r)
```

We also added a field `consume` in the signature to specify the subset of the arguments which are consumed to produce the outputs. This field enables taking fragments as arguments without consuming them, in order to use their value in the specification of the lemma. The semantics of `lemma` is a demonic non-deterministic choice of well-typed results whose combined footprint (that is, the disjoint union of their individual footprints) is equal to the footprint of the consumed arguments. To ensure that such choice is always possible and meaningful, we extend the soundness condition of lemmas.

Definition 3.5 A lemma is sound if for any well-typed values for its formal arguments,
1. the footprints of the consumed arguments are disjoint
2. there exists values having the types of its formal results and whose combined footprint is equal to the combined footprint of the consumed arguments

Note that this semantics and generalized soundness property coincides with the presentation of section 3.1.3 when no argument is consumed and all results are propositions. Indeed, inhabitants of propositions have an empty footprint so the additional conditions on footprints are satisfied. Moreover, when they have at most one inhabitant, there is no non-deterministic choice in the semantics of `lemma` in this case. In practice, we thus use the same `lemma` operator for proposition and memory fragments. We also use it to cast memory fragments between different types, like we did in figure 3.9.

This idea of using `lemma` both for deriving propositions and transforming memory fragments is inspired by the entailments of separation logic. Indeed, when mapping types into assertions of separation logic with the `footprint` function, the soundness condition resembles an entailment. There is a difference however: the disjointness of the premises is given as an assumption in entailments, whereas the corresponding disjointness property of the consumed arguments is a proof obligation in definition 3.5. The reason is that this disjointness property is usually justified with the existence of a global state satisfying the premises of the entailment. In our semantics, there is no such state and variables can have non-disjoint footprints. We instead require lemmas to prove the validity of the footprint of their arguments from the types of the arguments. A `Frag.joinable` operation can always be used before a lemma to ensure this disjointness. For instance the `lemma` of figure 3.11 merges the head of a linked list with its tail using the assumption that their footprints are disjoint. Separating `Frag.joinable` from

```
let ghost m0 : mem{mtrue}, ghost m1 : mem{mtrue} = lemma{
    (p : ptr, c : cell{i64} p)
  : (m0 : mem{mtrue}, m1 : mem{mtrue})
    consume (c)
  }(p, c);
let ghost IS_CELL : prop{is_cell} p m0 = assert<prop{is_cell} p m0>();
```

Figure 3.12: A program with a UB due to the non-deterministic semantics of `lemma` [compiler/test/src/example/lemma_nondet_ub.frag:10]

```
let ghost c0 : cell{i64} p = Frag.write{i64}(p, c, 0);
let ghost c1 : cell{i64} p = Frag.write{i64}(p, c, 1);
let ghost m00 : mem{P0} p, ghost m01 : mem{P1} p = lemma{…}(p, c0);
let ghost m10 : mem{P0} p, ghost m11 : mem{P1} p = lemma{…}(p, c1);
let ghost c2 : cell{i64} p = lemma{…}(p, m00, m11);
let v : i64 = Frag.read{i64}(p, c2);
output{i64}(v);
```

Figure 3.13: A program whose semantics depend on the interpretation of abstract predicates [compiler/test/src/pass/translation/ko_5_sem_dep.frag]

`lemma` ensures that there always exist values that satisfy our requirements on the results of lemmas. Contrary to `Frag.joinable`, lemmas can thus be commuted with UB.

Since they have a non-deterministic semantics, lemmas do not require an effective way of transforming the memories (for instance using a set of addresses for each output fragment). However, the types of the results must be restrictive enough to ensure that the chosen fragments are the expected ones. As a counterexample, the program of figure 3.12 uses a lemma to split a cell `c` into two memory fragments `m0` and `m1` of type `mem{mtrue}` (interpreted as a predicate satisfied by all memory fragments). It then uses an assertion to check the value of `m0`. We interpret the abstract predicate `is_cell` so that `prop{is_cell} p m0` is satisfied if and only if `m0` contains exactly the bytes in range $[\texttt{p}; \texttt{p} + \texttt{sizeof i64}]$. This program has a UB. Indeed, `lemma` is allowed to put some of the bytes of the cell in the second fragment. In this case, the assertion will fail. This example could be fixed by adding a post-condition `prop{is_cell} p m0` within the lemma statement.

Another consequence of the semantics of lemmas is that some Frag IR programs cannot be executed. Indeed, the non-deterministic choice performed by a lemma is constrained by the types of its results. With abstract predicates, those types can be arbitrarily complex and their definition not represented in the program text. Two different choices of interpretations of the abstract predicates may lead to incompatible semantics. The actual semantics of the program is thus selected by its producer. As an example, consider the program of figure 3.13. We start by creating two cells `c0` and `c1` at the same address `p` but with different contents (respectively `0` and `1`). We then use two lemmas to split each cell into two unstructured fragments described by some abstract predicates `P0` and `P1`. We use a third lemma to merge the fragment `mem{P0} p` obtained from `c0` with the fragment `mem{P1} p` obtained from `c1`. The result of this lemma is a cell `c2` at address `p`. We observe its content by outputting the value it contains. We cannot determine a correct value for this output without knowing the interpretations of `P0` and `P1`. The issue is that the bytes of `c2` may come from `c0` or `c1`. A first valid interpretation of the abstract predicate is to have `mem{P0} p` contain a cell at address `p` (so that it behaves like `cell{i64} p`) and `mem{P1} p` only contain the empty heaplet. All lemmas are then valid, because the combined footprints of their arguments and the combined footprints of their results are cells at address `p`. Moreover, they have deterministic semantics: the first two lemmas put all bytes in the fragments `mem{P0} p` and the third lemma takes all bytes from the fragment coming from `c0`. The semantics of the program is thus to output `0`. However, if we swap the interpretations of `P0` and `P1`, we obtain a semantics which outputs

1. There is thus no semantics which refines all semantics that the producer of the program is allowed to choose.

Notice however, that the program of figure 3.13 does not use fragments linearly: the original cell `c` at address `p` is consumed twice, by the two `Frag.write`. For programs satisfying the linearity criterion, we obtain an executable refinement independent of the interpretation of predicates by translating the operations on memory fragments by operations on a global memory. On some artificial examples (of program and interpretation of predicates), this refinement is strict, but on useful programs we expect the transformation to produce equivalent programs.

When structured fragments are used, Frag IR provides dedicated operations to transform the fragments. Given a structure description `{sz : ofs_i field_i;…}` with `n` fields of types `field_1`…`field_n` at offsets `ofs_1`…`ofs_n`, the operations `Strc.make` and `Strc.elim` build or destruct the first level of structured memory fragment with this layout. Both of them take as argument or return a list of variables containing the fields of the structure. This list includes the implicit, unstructured memory fragment `pad` containing the padding bytes. Its type `mpadding` is parameterized by the layout of the structure and its base pointer `p` to ensure that it contains the expected addresses. Both operations also return a post-condition `POST` which provides an equality between the value `s` of the structure and the combination of its fields with an operator `mkstrc`. The interest of those operations compared to using `lemma` is that they have simpler semantics, operating directly on the trees defining the structured fragments instead of their footprint.

The operations `lemma`, `Strc.make` and `Strc.elim` change the fragmented representation of the memory, but do not change its content. The translation into a global memory thus does not generate any instructions to implement them. As a consequence, the pointer arguments of `Strc.make` and `Strc.elim` do not need to be computed, hence the `ghost` attribute in figure 3.7.

3.2.8 Transformations on programs with memory fragments

Memory fragments enable program transformations that would be harder to detect or justify on a global memory. That is, with memory fragments:

- the validity of the transformations is ensured by a syntactic criterion rather than analyses
- proving the soundness of the transformations only requires reasoning on the modified parts

Figure 3.14 reproduces examples of transformations enabled by memory fragments. We manually performed and justified those transformations. Although we expect that memory fragments could be used to perform transformations in automated optimization passes, we did not implement such passes.

Value propagation Since memory fragments are immutable, information known about a variable storing a memory remains valid at any later point. For instance, in figure 3.14a, after the first `Frag.read`, we know that the content of `c` is equal to `v0`. We can thus use `v0` to replace a later read, independently of the instructions between the two reads. The reverse transformation is also always semantically valid, but may break the linearity criterion if `c` is consumed between the two accesses. A syntactic check could ensure that it is not the case.

Commutation of memory accesses Thanks to the absence of a global state, in most cases, instructions that operate on syntactically distinct variables are independent. In figure 3.14b, we are thus able to commute two `Frag.write` using only the fact that they operate on syntactically distinct cells. We emphasize that the correctness of this translation does not require any reasoning on the values of the pointers `p` and `q`. In fact, the transformation would be semantically valid even if `p` and `q` were equal, but the linearity criterion ensures that this situation cannot happen. The commutation of figure 3.14b is generalizable by replacing the write on `q` by an arbitrary instruction which does not syntactically depend on `c` or `c1`.

Although Frag IR does not have any global state, its semantics involves several effects, including

```
let v0, _ = Frag.read{i64}(p, c);          let v0, _ = Frag.read{i64}(p, c);
…                                    →     …
let v1, _ = Frag.read{i64}(p, c);          let v1 = v0;
```

a: Value propagation [compiler/test/src/transform/memory_rw.frag:3]

```
let c1, _ = Frag.write{i64}(p, c, v);   ↔   let d1, _ = Frag.write{i64}(q, d, w);
let d1, _ = Frag.write{i64}(q, d, w);       let c1, _ = Frag.write{i64}(p, c, v);
```

b: Write commutation [compiler/test/src/transform/memory_rw.frag:18]

```
if b Hb then {                              let v, _ = Frag.read{i64}(p, c);
  let v, _ = Frag.read{i64}(p, c);    ↔     if b Hb then {
  …                                           …
}                                           }
```

c: Load speculation [compiler/test/src/transform/memory_rw.frag:32]

```
let p, m = Frag.malloc(8);
if (p == null) _
then { … (* error handling *) }      →      ∅
Frag.free(p, 8, m);
```

d: Removal of unused allocation [compiler/test/src/transform/malloc.frag:24]

Figure 3.14: Transformations enabled by memory fragments

non-determinism and undefined behaviors. If pure operations (like `Frag.read` or `Frag.write`) may always be commuted with arbitrary instructions that use distinct variables, effectful operations require more care. We already mentioned in section 3.2.6 that `Frag.joinable` could not be commuted below `assert`. We also have technical difficulties to commute non-deterministic choices (`Frag.malloc` and `lemma`) below diverging loops, external events and function calls (section 8.2).

Load speculation The operation `Frag.read` is total. We may thus add loads during target executions, without risking introducing memory errors. Figure 3.14c uses this property to speculate a load above a branch. In the general case, this transformation is possible when the arguments of `Frag.load` (the pointer and the memory cell) are already available at the point where it is speculated. The example is valid because `p` and `c` are already defined before the conditional. It would not be possible if `c` was obtained using a `lemma` depending on the branch hypothesis `Hb : b = true`. This would signal that the validity of the load depends on the guard of the conditional.

Removal of dead allocations The semantics of Frag IR uses concrete pointers. A pointer is a 64-bit machine integer. There is thus only a finite number of them. In models based on a global memory, this property prevents the removal of unused allocations (section 2.2). In Frag IR, allocations are a priori independent of each other thanks to the absence of any global state. In figure 3.14d, we are thus able to remove a `Frag.malloc` and the corresponding `Frag.free`. We justify the soundness of the target using a source execution where the allocation succeeds. That is, the transformation is performed by replacing `Frag.malloc` with an arbitrary valid pointer and corresponding fragment. Such valid pointer exists since it is not constrained by other existing allocations and because the amount of bytes allocated (8) is not too large. If there was a `Frag.joinable` between `m` and other fragments, this justification would become invalid. Indeed, the choice of `p` would have to take into account other existing allocations. Note that using the error case of `Frag.malloc` (that is, choosing `p = null`) would lead to a target which executes the error handling part. This would amount to replacing calls to `malloc` with allocation failures, which would be semantically correct but undesirable.

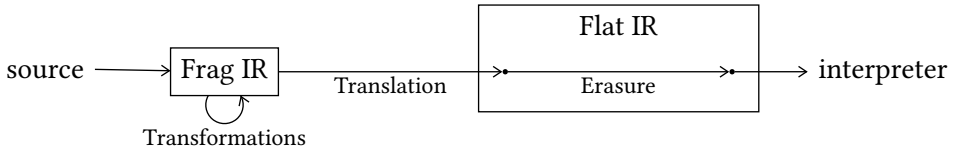


Figure 3.15: Overview of our compilation pipeline around Frag IR

3.3 Pipeline

This section explains how we use Frag IR in a verified compilation pipeline. Figure 3.15 gives an overview of the different IRs and passes involved.

Source programs The first step in this pipeline is to generate a program in Frag IR. We implemented a parser which enables writing programs directly in Frag IR. However, this approach is only feasible for toy examples. Indeed, Frag IR is designed to be an intermediate representation rather than a source language. It is verbose because of the syntactical representation of information with dependent types and memory fragments. Some toy Frag IR programs are an order of magnitude longer than equivalent C programs. In particular, we did not implement any type inference on Frag IR.

We instead intend Frag IR to be used as a middle-end representation. That is, source Frag IR programs should be generated automatically from higher-level representations. As a case study, chapter 10 uses Vélus [21] as a front-end. This enables generating Frag IR programs from synchronous programs, including existing ones. This connection with Vélus is implemented by a translation pass from Obc (a sequential, imperative IR of Vélus) to Frag IR.

Transformations Our pipeline then contains several transformation passes on Frag IR. Some transformations are optional and optimize or simplify the program. We implemented a Dead Code Elimination (DCE), a simplification pass and a block-sinking transformation. Those passes are standard, and do not leverage the information encoded with propositions and memory fragments. We also perform a translation pass at this level (section 9.2), which removes uses of some primitive operations.

Translation to Flat IR We ultimately want to obtain programs that are efficiently executable on a real machine. We explained in section 3.2.7 that in general, Frag IR programs are not even executable. However, programs satisfying the linearity criterion can always be refined into an executable representation, by replacing memory fragments with a global memory. To perform this transformation, we defined another representation, called Flat IR, which extends Frag IR with a global memory. We define an "executable" subset of Flat IR, which does not contain proposition types or memory fragments. This subset resembles existing low-level imperative languages (e.g. Cminor in CompCert or VIR). In particular, we implemented an interpreter for programs written in this subset. We compile Frag IR programs which use fragments linearly into the executable subset of Flat IR. This compilation involves two steps. The first one is a translation from Frag IR to Flat IR. It leverages the linearity to replace the operations on memory fragments by corresponding operations on the global memory. After this transformation, we obtain a Flat IR program containing memory fragments and propositions only as ghost variables. The second step is to erase them.

Propagation of information In our pipeline, semantic information about the program is encoded in Frag IR using the proposition types and memory fragments. This information is given by the producer of the source program. When it is written manually, it is specific to the program at hand. When it is generated automatically, this information may reflect the encoding of the front-end language. For instance, with the Vélus front-end, memory fragments define a tree-shaped memory corresponding to the nodes of the source program.

Although none of our automated passes use the information encoded in Frag IR, they have to preserve

it. In particular, they must ensure the correctness of the information represented in the program they generate. Ensuring this preservation only required small specific logic within our passes. Syntactically, the most important consequence of the encoding of information is the use of dependent types. We also had to restrict the DCE to avoid removals of some memory fragments that would introduce memory leaks or break the linearity condition. The formal proofs of those passes are also largely unaffected by the presence of propositions and memory fragments. We only had to give a single additional global invariant, in the proof of typing preservation of the DCE. This addition is related to the presence of dependent types, which requires an injective relation between the source and target variables. The fact that we are able to propagate information with little effort is linked to two aspects of our approach:

- Information is encoded using standard variables (propositions and memory fragments).
- We rely on defensive passes to avoid formally proving invariants of the pipeline. We sometimes use a type-checker when we formally need typing assumptions. The linear usage of memory fragments is only needed when translating them using a global memory, and we defensively check it at this point.

Memory fragments are kept as ghost variables during the translation to Flat IR. Some information about the memory is thus still available at this point. It is discarded when the propositions and memory fragments are erased.

Formal properties of the pipeline We define a well-formedness property on Frag IR and Flat IR programs, which include the fact that they only contain sound lemmas. It is a formal invariant of our compilation pipeline. The producer of the source Frag IR program must ensure that it is well-formed and we formally prove that each pass preserves the well-formedness.
For each pass, we also prove a semantic refinement between its source and target programs. By composing those refinements, we obtain a refinement between the source Frag IR program and the target Flat IR program. This semantic refinement relates the observable behaviors of the source and target. It enables deriving properties of the target program from similar properties on the source. When combined with a verified front-end like Vélus, we obtain formal guarantees with respect to the source program of the front-end. Another benefit of having a semantic refinement is that it enables the use of UBs (with `assert` in Frag IR) to propagate information.

Informal invariants There are two informal invariants in our pipeline: the fact that programs are well-typed and the linear use of fragments. Those properties must be satisfied by the source program and preserved by the passes. We say that they are informal because we do not always give a formal proof that they hold. Instead, they are checked defensively (by the type-checker or the translation to Flat IR) when we need them to justify some formal properties. If, because of a bug, they would not hold at some point in the compilation pipeline, then either:

- a defensive pass would detect the error and abort the compilation
- or the error could go unnoticed if the invariant is reestablished by a later transformation, for instance if a faulty part is optimized away

Using informal invariants lightens the proof burden in our verified compilation pipeline, but increases the risk of compilation failures.

3.4 Related works

Proof Carrying Code enables verification of properties of low-level code from an untrusted source or compiler [6; 81]. This approach relies on accompanying machine-checkable proofs. A compiler may produce such a certificate by translating a source-level proof. Our encoding of information with dependent types is motivated by a similar need for expressiveness. However, our goal is not to check the information encoded in a program. In particular, we do not carry a representation of the proofs of `lemma` instructions. Moreover, we prove once and for all that our compilation passes generate target code which semantically refines the source, according to a notion of backward simulation. One can thus establish complex properties on the behavior of the target without encoding them in the type

system. This refinement property also enables the use of the `assert` instruction since it implies that the target does not have UB if the source does not.

Dependent types are a way of encoding invariants of a program. They are also used in some proof assistants, like Rocq [46], Lean [78] or Agda [3]. Our proposition types and lemmas on them are largely inspired by the concepts with the same names of the verification language F* [97]. However, our typing system is significantly more restricted, although we recover some expressiveness thanks to the abstract predicates.
We only use dependent types for propositions and memory fragments. Other approaches attach invariants directly to the data manipulated by programs (e.g. with refinement types in F* or sigma types in Rocq).
Since one of our goals is to track the dependencies of information, we only attempt to encode intrinsic proofs. In this style of proofs, properties about a component of a program (e.g. a function) must be derived within the component and declared in its signature. Results of extrinsic proofs (reasoning on components after defining them) may still be reflected in the type system using `assert`.
In our system, types are only compared syntactically, but reasoning about them is possible with the `lemma` operation. In Rocq, types are compared modulo a conversion relation, which captures a part of the semantics of types directly in the typing judgment. F* even includes non-decidable sub-typing conditions, and checks them using SMT queries.

Representation of the memory with values Our compilation pipeline performs the inverse of the transformations of Creusot [37] and Aeneas [47] (section 2.5). That is, we use the direction originally studied by Wadler [100]. The source language of our pipeline, Frag IR, has a semantics where the memory fragments are handled like values. The global memory is only introduced later in the pipeline and the encoding is performed by the compiler, thanks to a defensive linearity check.
Capla [75] is a domain-specific language for algebra libraries where arrays are stored in mutable variables and passed by copy-restore. In Frag IR, fragments are immutable. Functions thus need to return syntactically distinct fragments instead of modifying their arguments. Similarly to our linearity criterion, the type system of Capla prevents aliasing between a mutable variable and other variables. Contrary to Frag IR, Capla supports read-only variables, which may be aliased. Another difference is that the semantics of Capla does not involve pointers, only the content of arrays. During the compilation to C, variables with array types are replaced with pointers. Dependent types are used to track the sizes of the arrays. This enables emitting out-of-bound checks instead of using UB.

Part II
Modular Intermediate Representations and passes

4

SMonad: a library to define denotational semantics

This chapter describes a Rocq library, SMonad, which provides tools to define and reason on denotational semantics. The core of the library is an axiomatization of operators to build semantics and of their interaction with a refinement relation. This axiomatization generalizes relational reasoning principles which were identified on Interaction Trees [102]. The axiomatization is satisfied by several models, which support different effects. We derive reasoning principles from the axioms. Those principles may be used on any model satisfying the axioms. We also define models of the axiomatization and transformers to derive more complex models. The most developed model is based on Labelled Transition Systems (LTSs), and described in chapter 5.

This chapter uses as running example a toy imperative language called MiniBTL. It is inspired by the Block Transfer Language (BTL) of Chamois [45], a variant of CompCert developed at Verimag (section 2.4). Figure 4.1 describes the syntax of MiniBTL. A program is a list of mutually recursive functions. The body of each function is a Control Flow Graph (CFG) of blocks, identified by their `label`. Each block is defined by a loop-free inductive instruction (`instr`). Those instructions operate over variables (`var`) local to the enclosing function. In particular, instructions may assign to a variable the result of the evaluation of an operation or of a function call.

Section 4.1 recalls standard notions on semantics, and motivates our choice of using denotational ones. Section 4.2 describes the axiomatization used by SMonad. Section 4.3 explains how this axiomatization is generally used, for relational reasoning. Section 4.4 mentions properties derived from the axiomatization, and their consequences. Section 4.5 explains how we derive new models using transformers. Section 4.6 concludes with existing works related to semantics.

```
op       ::= Const (c : ℕ) | Add                pure operations
           | Div                                 partial division
           | Pick                                non-deterministic choice
           | Input | Output                      external events
           | Read | Write                        memory accesses

instr    ::= skip                                no operation
           | return var                          terminate the function
           | goto label                          jump to a label
           | var = (op|fct_name)(var*)           operation or function call
           | instr; instr                        sequence
           | if var {instr} else {instr}         conditional

function ::= fn fct_name(var*) {(label:instr)*}

program  ::= function*
```

Figure 4.1: Syntax of a toy MiniBTL language [SMonadExamples.MiniBTL.Syntax]

4.1 Semantics

Section 4.1.1 recalls standard notions about semantics. We then motivate our choice of using denotational semantics, by comparing on MiniBTL an operational semantics in the style of CompCert (section 4.1.2) and a denotational semantics defined using SMonad (section 4.1.3).

4.1.1 Formal definitions of semantics

The semantics (sometimes called "dynamic semantics" [69, §4.2]) of a language L is a mathematical definition of the executions admitted by programs written in L. It thus captures precisely the interpretation of the syntax of L. In verified compilation, this semantics is used to specify the correctness of passes that take as input or produce programs in L. It acts as a common ground between consecutive passes. Semantics are thus key ingredients, which enable proofs of compilation pipelines by composition of the proofs of individual passes.

It is also possible to define the semantics of *components* of a program. In this case, the semantics must account for the interactions with other parts of the program. In this thesis, we emphasize those additional interactions using the following vocabulary:

Definition 4.1

- The external *environment* (or simply environment) contains everything that is not part of the program. In this thesis, we will often model the interaction of the program with the environment using toy input/output operations. In reality, the environment would include other programs, the system and the physical world.
- The *context* of a component is the union of the external environment and of the other parts of the program. Components can interact with the environment, but also with other components of the program (executed concurrently or sequentially) through language-specific mechanism (e.g., function call or a state).

A challenge in defining semantics is that they involve notions that are not natively supported in mathematical logics, in particular in Rocq.

Definition 4.2 A semantics is *pure* if it is represented by a total function. That is, if it is fully characterized by an association of a unique result to every input. Semantics that are not pure are called *effectful*. An *effect* is an aspect of a semantics which makes it effectful.

Example 4.1 In general, the semantics of the execution of a program on a real computer is not pure. Examples of effects which may appear at this level are:

Errors programs may stop on errors instead of generating a result.
In MiniBTL, the division operator `Div` has an error if its second argument is 0. During compilation, errors can be interpreted as UBs. This allows removing operations, even if they could raise errors.

Divergence the executions of some programs are infinite and thus never produce a final result.
There are two sources of divergence in MiniBTL: loops in CFGs and recursion between functions.

Interactions programs may interact with their environment during their execution.
MiniBTL features some `Input` and `Output` operators.

Example 4.1 only mentions effects directly observable during concrete execution of whole programs. We will introduce other effects to define the semantics of components of programs (for instance access to a state, e.g., memory `Read` and `Write` in MiniBTL), and of intermediate representations which are not directly executable. Internal non-determinism is an effect useful for such IRs. It must be distinguished from external non-determinism:

Definition 4.3 A semantics has *external non-determinism* if its outputs depend on its interaction with its context. A semantics has *internal non-determinism* if it admits zero, or several outputs for the same set of inputs from its context.

Example 4.2 In this thesis, we consider that the following functions both feature non-determinism:

```
bool nd_ext() { int r = input(); return r == 0; }
bool nd_int() { void* p = malloc(8); return p == NULL; }
```

`nd_ext` has external non-determinism, because its result depends on the answer to an external event `input`. On the contrary, `nd_int` has internal non-determinism, because its result depends on the choice performed by the implementation of `malloc` (which can return `NULL`).

MiniBTL has a `Pick` operator, as an artificial source of internal non-determinism. In this thesis, we consider *demonic* non-determinism. That is, we consider that a program is correct if *all* outputs it admits are allowed by its specification. In practice, internal demonic non-determinism is a way of abstracting choices made later in the compilation pipeline (e.g., the implementation of `malloc`). The program must be correct independently of them. This allows transforming those choices during the compilation. In particular, they can be replaced with deterministic implementations. But they can also be optimized while keeping non-determinism in the target (e.g., merging several `malloc`).

The logic of Rocq is based on pure concepts. Giving a formal definition to a semantics thus requires encoding its effects. For instance, a semantics which returns values of type `A` but can also stop with an error can be represented using a total function with co-domain `option A`. For more general sequential programs, semantics can often be defined as a set of "behaviors" (or "traces") observable when executing the program.
There are then two approaches to reason on semantics:
- either rely on the chosen encoding to reason on pure objects.
- or abstract this encoding and reason using properties of the effects.

We follow the second approach.

4.1.2 Operational semantics

Operational semantics describe the executions of programs by modelling an idealized machine. Figure 4.2 is an extract of a definition of an operational semantics for MiniBTL. This semantics only supports a subset of the language. We do not handle events (`Input`, `Output`), nor non-determinism (`Pick`). Moreover, we assume that function calls are only performed within blocks with shape "`x = f(args); goto l`". The original BTL language has similar restrictions, but supports events and

$$\texttt{final} ::= \texttt{FRet}\ (\texttt{x} : \texttt{var}) \mid \texttt{FGoto}\ (\texttt{trg} : \texttt{label})$$
$$(\texttt{me}_0 : \texttt{Mem.t}), (\texttt{ve}_0 : \texttt{VEnv.t}), (\texttt{u} : \texttt{instr}) \Rightarrow (\texttt{me}_1 : \texttt{Mem.t}), (\texttt{ve}_1 : \texttt{VEnv.t}), (\texttt{f} : \texttt{option final})$$

$$\frac{}{\texttt{me}_0, \texttt{ve}_0, (\texttt{return r}) \Rightarrow \texttt{me}_0, \texttt{ve}_0, \lfloor(\texttt{FRet r})\rfloor} \qquad \frac{}{\texttt{me}_0, \texttt{ve}_0, (\texttt{goto l}) \Rightarrow \texttt{me}_0, \texttt{ve}_0, \lfloor(\texttt{FGoto l})\rfloor}$$

$$\frac{\texttt{me}_0, \texttt{ve}_0, \texttt{lhs} \Rightarrow \texttt{me}_1, \texttt{ve}_1, \lfloor\texttt{f}\rfloor}{\texttt{me}_0, \texttt{ve}_0, (\texttt{lhs; rhs}) \Rightarrow \texttt{me}_1, \texttt{ve}_1, \lfloor\texttt{f}\rfloor} \qquad \frac{\texttt{me}_0, \texttt{ve}_0, \texttt{lhs} \Rightarrow \texttt{me}_1, \texttt{ve}_1, \varnothing \qquad \texttt{me}_1, \texttt{ve}_1, \texttt{rhs} \Rightarrow \texttt{me}_2, \texttt{ve}_2, \texttt{f}}{\texttt{me}_0, \texttt{ve}_0, (\texttt{lhs; rhs}) \Rightarrow \texttt{me}_2, \texttt{ve}_2, \texttt{f}}$$

a: Big-step semantics of instructions (extract)

$$\texttt{stackframe} := \{\texttt{func} : \texttt{function}; \texttt{ret_var} : \texttt{var}; \texttt{succ} : \texttt{label}; \texttt{venv} : \texttt{VEnv.t}\}$$
$$\texttt{state} := \{\texttt{stack} : \texttt{list stackframe}; \texttt{func} : \texttt{function}; \texttt{pc} : \texttt{label}; \texttt{mem} : \texttt{Mem.t}; \texttt{venv} : \texttt{VEnv.t}\}$$
$$\texttt{init}\ (\texttt{p} : \texttt{program})\ (\texttt{vargs} : \texttt{list val}) : \texttt{state} :=$$
$$\{\texttt{stack} := []; \texttt{func} := \texttt{p@main}; \texttt{pc} := \texttt{entry}; \texttt{mem} := \texttt{Mem.init}; \texttt{venv} := [\texttt{xargs} \mapsto \texttt{vargs}]\}$$
$$\text{if } \texttt{p@main} = (\texttt{fn main(xargs) \{entry:...\}})$$
$$(\texttt{p} : \texttt{program}) : (\texttt{pre} : \texttt{state}) \Rightarrow (\texttt{post} : \texttt{state} + \texttt{val})$$

$$\frac{\texttt{st.mem}, \texttt{st.venv}, (\texttt{st.func})@(\texttt{st.pc}) \Rightarrow \texttt{me}_1, \texttt{ve}_1, \lfloor\texttt{FGoto l}\rfloor}{\texttt{p} : \texttt{st} \Rightarrow \texttt{inl}\ \{\texttt{stack} := \texttt{st.stack}; \texttt{func} := \texttt{st.func}; \texttt{pc} := \texttt{l}; \texttt{mem} := \texttt{me}_1; \texttt{venv} := \texttt{ve}_1\}}$$

$$\frac{\texttt{st.mem}, \texttt{st.venv}, (\texttt{st.func})@(\texttt{st.pc}) \Rightarrow \texttt{me}_1, \texttt{ve}_1, \lfloor\texttt{FRet x}\rfloor \qquad \texttt{st.stack} = []}{\texttt{p} : \texttt{st} \Rightarrow \texttt{inr}\ (\texttt{ve}_1\ \texttt{x})}$$

$$\frac{\texttt{st.mem}, \texttt{st.venv}, (\texttt{st.func})@(\texttt{st.pc}) \Rightarrow \texttt{me}_1, \texttt{ve}_1, \lfloor\texttt{FRet x}\rfloor \qquad \texttt{st.stack} = [\texttt{cr}; \texttt{tl}]}{\texttt{p} : \texttt{st} \Rightarrow \texttt{inl}\ \left\{\begin{array}{l}\texttt{stack} := \texttt{tl}; \texttt{func} := \texttt{cr.func}; \texttt{pc} := \texttt{cr.succ};\\ \texttt{mem} := \texttt{me}_1; \texttt{venv} := [\texttt{cr.venv}, \texttt{cr.ret_var} \mapsto \texttt{ve}_1\ \texttt{x}];\end{array}\right\}}$$

$$\frac{(\texttt{st.func})@(\texttt{st.pc}) = (\texttt{x = f(args); goto l}) \qquad \texttt{p@f} = (\texttt{fn f(xargs) \{entry:...\}})}{\texttt{p} : \texttt{st} \Rightarrow \texttt{inl}\ \left\{\begin{array}{l}\texttt{stack} := \left\{\begin{array}{l}\texttt{func} := \texttt{st.func}; \texttt{ret_var} := \texttt{x};\\ \texttt{succ} := \texttt{l}; \texttt{venv} := \texttt{st.venv};\end{array}\right\} :: \texttt{st.stack};\\ \texttt{func} := \texttt{p@f}; \texttt{pc} := \texttt{entry}; \texttt{mem} := \texttt{st.mem}; \texttt{venv} := [\texttt{xargs} \mapsto \texttt{st.venv(args)}];\end{array}\right\}}$$

b: Small-step semantics of programs

Figure 4.2: Extract of an operational semantics for a subset of MiniBTL

function calls at the end of arbitrary blocks. Our definition has the same structure as its semantics[9], which itself follows several patterns used in the semantics of CompCert's languages [69, §3.5]. Like all languages of CompCert, BTL has a small-step semantics. However, it also uses big-step semantics for its loop-free instructions.

Small-step semantics We first focus on the small-step part, which defines the semantics of the whole program (figure 4.2b). It defines a transition system, by a set of states (`state`), an initial state (`init`) and a transition relation. This relation has shape `p : pre ⇒ post`, where `p` is the whole program considered, `pre` a state and `post` either a successor state or a final program result. The type `state` contains the current active function (`func`), the label of the next block to execute (`pc`), the state of the global memory (`mem : Mem.t`) and values (`val`) of the local variables of the function (`venv : VEnv.t`, where `VEnv.t := var → val`). The state also includes a call stack (`stack`). Each frame describes the state of the caller, and how to continue its execution after the callee terminates. The type `stackframe`

[9] https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert/-/blob/f07b18203421f1732d7167f510bd5c94d920640d/scheduling/BTL.v#L230

thus contains the syntactic definition of the caller (`func`), the variables to which the result of the callee is assigned (`ret_var`), the label of the block following the call (`succ`) and the values of the local variables of the caller (`venv`).
The initial state `init` has an empty stack and starts at the `main` function (whose declaration in `p` is given by `p@main`), with an initial memory (`Mem.init`), and the values given as parameters assigned to the formal arguments.
The step relation handles both transitions within and between functions. It is defined inductively, with a constructor for each kind of transition. Most cases involve the big-step semantics `st.mem`, `st.venv`, `(st.func)@(st.pc)` $\Rightarrow$ $\texttt{me}_1$, $\texttt{ve}_1$, `f` of the current block `(st.func)@(st.pc)`. This premise defines the states of the memory ($\texttt{me}_1$) and of the local variables ($\texttt{ve}_1$) after the block, as well as its terminator. This terminator is described by the type `final`, and either returns a variable (`FRet`) or jumps to another block (`FGoto`). For a jump, the corresponding rule of the step relation only updates the part of the state related to the active function. For a return, the program either terminates or gives back the control to the caller, depending on whether the stack is empty. Finally, a call block saves the state of the caller in a new stack frame, and initializes the callee.

Non modularity This small-step semantics handles simultaneously the CFGs of functions and the recursion between functions. That is, there is no semantics of a function in isolation, only of functions within a whole program. As a consequence, the `state` and the step relation mix concepts local to functions (e.g., `pc`, `venv`) with those implementing the recursion between function (`stack`). Most IRs of the back-end of CompCert feature a similar stack-based representation of recursion, but each small-step semantics implements it from scratch.

Proofs by simulation In CompCert, the correctness of passes is proved using simulations, that is, by coinduction on the execution of the program. Because there is no semantics of functions in isolation, those simulations are proved at the level of programs, even for intra-procedural transformations. In particular, a simulation invariant must be defined for the stack. For example, if local variables are reorganized during an intra-procedural transformation (e.g. dead code elimination or code motion), because their state is saved in the stack during function calls, the simulation invariant needs to express how this reorganization transforms the stacks. Moreover, the preservation of this simulation invariant requires a formal proof, even if, intuitively, the semantics (e.g. for BTL) ensures that callees cannot corrupt the local variables of callers.
Another issue with direct proofs of simulation is that they involve the notion of steps of the programs. Small-step semantics involves intermediate steps, which do not interact with the environment (e.g., by emitting events), and are called τ-steps. All but the last of the steps of our current subset of MiniBTL fall in this category. Silently diverging programs are encoded by infinite sequences of τ-steps. However, semantics differing by finite sequences of τ-steps admit the same behaviors, and should be considered equivalents. Some "stuttering" simulations relate semantics which differ by such finite sequences [69, §3.7]. However, they can be harder to use in practice than "lock-step" simulations, which are more restricted one-to-one refinements between source and target steps. In the formalism of CompCert, simulations where the target program performs fewer steps than the source involve an auxiliary measure to avoid infinite stuttering. For this reason, some transformations insert no-op instructions (later removed by a separate pass [69, §9]) to add artificial steps in the target.
Moreover, the notion of step of the semantics is not always the most convenient for the proofs. For instance, at the beginning of a function call, the semantics steps within the callee, and may only return to the caller once it terminates. This mismatches with the way programs are usually transformed, especially in intra-procedural passes. A more natural proof flow would consider directly what happens after the call returns.

Effects Small-step semantics encode divergence by infinite sequences of τ-steps. They can also support interaction with the environment through external events, by adding a corresponding label on the steps (we do not do it in figure 4.2b). In the framework of CompCert, errors (that is, UBs) are represented implicitly, by stuck states without outgoing transitions. Since transitions are defined by a relation, there can be several transitions leaving a given state. This models both internal and external non-determinism. However, allowing internal non-determinism has several implications. Firstly, we may want to consider that states without out-going transitions do not admit any behaviors. Such states would then be interpreted as situations that the program is allowed to ignore, because they are handled at a lower level. In Frag IR, we use this notion to model the guarantees given by the compiler when `Frag.joinable` is used section 3.2.6. The absence of behaviors is a concept distinct from the UBs associated to states without out-going transitions in CompCert (which can be implemented by anything). Secondly, considering small-step semantics without internal non-determinism can simplify proofs. In particular, most IRs of CompCert adhere to this restriction, to enable the use of forward simulations (section 4.6.1.1).

Big-step semantics Some issues of small-steps semantics can be solved by using big-step operational semantics (sometimes called "natural semantics"). In MiniBTL, this is the approach followed for the semantics of instructions (figure 4.2a). This semantics is defined as an inductive predicate $\mathtt{me}_0, \mathtt{ve}_0, \mathtt{u} \Rightarrow \mathtt{me}_1, \mathtt{ve}_1, \mathtt{f}$, which describes the memory ($\mathtt{me}_1$), local states ($\mathtt{ve}_1$) and optional terminator ($\mathtt{f}$) after the execution of an instruction ($\mathtt{u}$) from given initial states ($\mathtt{me}_0$, $\mathtt{ve}_0$).
This semantics does not depend on the context of the instruction (function, program). It thus enables specifying transformations directly on isolated instructions. Such specification can be described as implications between the source and target big-step semantics, and proved by induction on the source semantics.
The notion of step in this semantics follows the syntactic structure of the program. In particular, sequences are executed by a single application of a rule, which directly has as premise on the right-hand side when the left-hand side does not end with a terminator. This contrasts with the function calls in the small-step semantics.
However, some effects are harder to represent with big-step semantics. In particular, divergence requires switching to coinductive predicates.

4.1.3 Denotational semantics

The idea of denotational semantics is to associate a semantics to each syntactic element of programs (expressions, instructions, functions…). The semantics of composite elements is defined by composing the semantics of their components. For instance, we define the semantics of programs by composing the semantics of their functions. This approach requires mathematical objects to represent not only the semantics of whole programs, but also of their components in isolation. In this thesis, we will call *domains*[10] such sets of objects. In order to be used to define the semantics of a language, a domain must be expressive enough to capture the properties of interest and in particular the effects of programs. The semantics of each kind of syntactic elements is defined as a denotation function. Such functions map syntactic elements to the representation of their semantics, as elements of the domain. For instance in figure 4.3, the denotation function $[\![.]\!]^0_{\mathrm{instr}}$ maps syntactic instructions (in the type `instr`) into a domain $\mathtt{S}^0$. The domain is used as a formal language to describe the semantics. It must provide mathematical operators to build semantics. In particular, the definition of composite elements relies on combinators on the domain (e.g. `seq` in figure 4.3).

[10]The partially ordered sets of domain theory are examples of such sets, but we use more generally the name "domain" for any representation of semantics.

The denotational approach has several benefits:

- The mathematical domain is independent of the syntax of the language. The same domain can be used to define the semantics of several languages.
  This enables factorizing definitions and proofs between languages. For instance, we will consider a generic operator `mrec`, which defines semantics of systems of recursive functions. This operator comes with reasoning principles, which notably facilitate proofs of intra-procedural transformations. It is also easier to compare programs defined in different languages if their denotations use the same domain.
- Isolated syntactic elements have a semantics.
  This enables decomposing proofs of transformations by specifying their effects on the syntactic elements. Those proofs are then composed using properties of the operators of the domain.
  It is also possible to quantify over the semantics of some elements. For instance, we will define in section 6.4 a semantics of programs parameterized by the semantics of the operators.
- Denotation functions do not depend on the definition of the domain. They only depend on the signature of the operators which build elements of the domain. In particular, they do not depend on the way effects are represented.
  As an example, consider the semantics of straight-line blocks defined in figure 4.3. We first give a semantics to the subset of the language containing only the pure operators `Const` and `Add`. In this setting, the semantics does not involve a global memory. The big-step operational semantics $\rho_0, a \Rightarrow^0 \rho_1$ of an instruction a only depends on the values of the variables before (ρ_0) and after (ρ_1) the execution. Similarly, we define a domain S^0 with the variables as only state. The denotational semantics $[\![.]\!]^0_{\text{instr}}$ is then defined by recursion using the operators of S^0. In order to add the memory operators `Read` and `Write`, the signature of the big-step semantics needs to be updated. The extended $m_0, \rho_0, a \Rightarrow^1 m_1, \rho_1$ takes as additional arguments the state of the memory before (m_0) and after (m_1) the instruction. This change affects all rules, not only the case of assignment which directly uses the new operators. On the contrary, with the denotational approach, the additional effect only affects the definition of the domain and the case of operators. Indeed, the new domain S^1 supports all operators of S^0. The presence of an additional effect only adds new operators `get_mem` and `set_mem`. The signature of the new denotation function is thus updated by replacing S^0 with S^1. Cases of the denotation function not related to the new effect simply use the operators of S^1 instead of those of S^0.
  As illustrated, some domains share common operators. Those operators may also have similar properties. By axiomatizing such an interface, we enable quantifying over domains both in definitions and reasoning. We thus obtain theories (section 4.4), semantics of languages (section 6.4) and proofs of transformations (chapter 7) generic with respect to the effects.

Without memory	**With memory**

Semantics of the operators

$[\![\texttt{o} : \texttt{op}]\!]^0_{\text{op}} : \texttt{list val} \to \texttt{val}$

$[\![\texttt{Const n}]\!]^0_{\text{op}}\ [] := \texttt{n}$

$[\![\texttt{Add}]\!]^0_{\text{op}}\ [\texttt{n}_0; \texttt{n}_1] := \texttt{n}_0 + \texttt{n}_1$

$[\![\texttt{o} : \texttt{op}]\!]^1_{\text{op}} : \texttt{Mem.t} \to \texttt{list val} \to \texttt{val} \times \texttt{Mem.t}$

$[\![\texttt{Const n}]\!]^1_{\text{op}}\ \texttt{m}\ [] := (\texttt{n}, \texttt{m})$

$[\![\texttt{Add}]\!]^1_{\text{op}}\ \texttt{m}\ [\texttt{n}_0; \texttt{n}_1] := (\texttt{n}_0 + \texttt{n}_1, \texttt{m})$

$[\![\texttt{Read}]\!]^1_{\text{op}}\ \texttt{m}\ [\texttt{p}] := (\texttt{m}\ \texttt{p}, \texttt{m})$

$[\![\texttt{Write}]\!]^1_{\text{op}}\ \texttt{m}\ [\texttt{p}; \texttt{v}] := (0, [\texttt{m}, \texttt{p} \mapsto \texttt{v}])$

Big-step semantics

$$\frac{}{\rho, \texttt{skip} \Rightarrow^0 \rho}$$

$$\frac{[\![\texttt{o}]\!]^0_{\text{op}}\ \rho(\texttt{ys}) = \texttt{v}}{\rho, (\texttt{x} = \texttt{o(ys)}) \Rightarrow^0 [\rho, \texttt{x} \mapsto \texttt{v}]}$$

$$\frac{\rho_0, \texttt{a}_0 \Rightarrow^0 \rho_1 \qquad \rho_1, \texttt{a}_1 \Rightarrow^0 \rho_2}{\rho_0, (\texttt{a}_0;\ \texttt{a}_1) \Rightarrow^0 \rho_2}$$

$$\frac{}{\texttt{m}, \rho, \texttt{skip} \Rightarrow^1 \texttt{m}, \rho}$$

$$\frac{[\![\texttt{o}]\!]^0_{\text{op}}\ \texttt{m}_0\ \rho(\texttt{ys}) = (\texttt{v}, \texttt{m}_1)}{\texttt{m}_0, \rho, (\texttt{x} = \texttt{o(ys)}) \Rightarrow^1 \texttt{m}_1, [\rho, \texttt{x} \mapsto \texttt{v}]}$$

$$\frac{\texttt{m}_0, \rho_0, \texttt{a}_0 \Rightarrow^1 \texttt{m}_1, \rho_1 \qquad \texttt{m}_1, \rho_1, \texttt{a}_1 \Rightarrow^1 \texttt{m}_2, \rho_2}{\texttt{m}_0, \rho_0, (\texttt{a}_0;\ \texttt{a}_1) \Rightarrow^1 \texttt{m}_2, \rho_2}$$

Denotation domain

$\texttt{S}^0 := \texttt{VEnv.t} \to \texttt{VEnv.t}$

$\texttt{skip} : \texttt{S}^0 := \lambda\rho.\ \rho$

$\texttt{get_venv}\ (\texttt{u} : \texttt{VEnv.t} \to \texttt{S}^0) : \texttt{S}^0 := \lambda\rho.\ \texttt{u}\ \rho\ \rho$

$\texttt{set_venv}\ (\rho : \texttt{VEnv.t}) : \texttt{S}^0 := \lambda_.\ \rho$

$\texttt{seq}\ (\texttt{u}_0\ \texttt{u}_1 : \texttt{S}^0) : \texttt{S}^0 := \texttt{u}_1 \circ \texttt{u}_0$

$\texttt{S}^1 := \texttt{Mem.t} \times \texttt{VEnv.t} \to \texttt{Mem.t} \times \texttt{VEnv.t}$

$\texttt{skip} : \texttt{S}^1 := \lambda(\texttt{m}, \rho).\ (\texttt{m}, \rho)$

$\texttt{get_venv}\ (\texttt{u} : \texttt{VEnv.t} \to \texttt{S}^1) : \texttt{S}^1 := \lambda(\texttt{m}, \rho).\ \texttt{u}\ \rho\ (\texttt{m}, \rho)$

$\texttt{set_venv}\ (\rho : \texttt{VEnv.t}) : \texttt{S}^1 := \lambda(\texttt{m}, _).\ (\texttt{m}, \rho)$

$\texttt{seq}\ (\texttt{u}_0\ \texttt{u}_1 : \texttt{S}^1) : \texttt{S}^1 := \texttt{u}_1 \circ \texttt{u}_0$

$\texttt{get_mem}\ (\texttt{u} : \texttt{Mem.t} \to \texttt{S}^1) : \texttt{S}^1 := \lambda(\texttt{m}, \rho).\ \texttt{u}\ \texttt{m}\ (\texttt{m}, \rho)$

$\texttt{set_mem}\ (\texttt{m} : \texttt{Mem.t}) : \texttt{S}^1 := \lambda(_, \rho).\ (\texttt{m}, \rho)$

Denotational semantics

$[\![\texttt{a} : \texttt{instr}]\!]^0_{\text{instr}} : \texttt{S}^0$

$[\![\texttt{skip}]\!]^0_{\text{instr}} := \texttt{skip}$

$[\![\texttt{x} = \texttt{o(ys)}]\!]^0_{\text{instr}} := \texttt{get_venv}\ (\lambda\rho.\ \texttt{set_venv}\ ([\rho, \texttt{x} \mapsto [\![\texttt{o}]\!]^0_{\text{op}}\ (\rho(\texttt{ys}))]))$

$[\![\texttt{a}_0; \texttt{a}_1]\!]^0_{\text{instr}} := \texttt{seq}\ [\![\texttt{a}_0]\!]^0_{\text{instr}}\ [\![\texttt{a}_1]\!]^0_{\text{instr}}$

$[\![\texttt{a} : \texttt{instr}]\!]^1_{\text{instr}} : \texttt{S}^1$

$[\![\texttt{skip}]\!]^1_{\text{instr}} := \texttt{skip}$

$[\![\texttt{x} = \texttt{o(ys)}]\!]^1_{\text{instr}} := \texttt{get_mem}\ (\lambda\texttt{m}.\ \texttt{get_venv}\ (\lambda\rho.\ \texttt{set_result}\ \rho\ \texttt{x}\ ([\![\texttt{o}]\!]^1_{\text{op}}\ \texttt{m}\ (\rho(\texttt{ys})))))$

$[\![\texttt{a}_0; \texttt{a}_1]\!]^1_{\text{instr}} := \texttt{seq}\ [\![\texttt{a}_0]\!]^1_{\text{instr}}\ [\![\texttt{a}_1]\!]^1_{\text{instr}}$

where:

$\texttt{set_result}\ \rho\ \texttt{x}\ (\texttt{v}, \texttt{m}) : \texttt{S}^1 := \texttt{seq}\ (\texttt{set_mem}\ \texttt{m})\ (\texttt{set_venv}\ [\rho, \texttt{x} \mapsto \texttt{v}])$

Figure 4.3: Four toy semantics of straight-line MiniBTL instructions. We consider two restrictions of the operators (with and without a memory state) and two approaches to define the semantics (operational big-step and denotational). [SMonadExamples.MiniBTL.BlockSemantics]

Domain

```
BTL_S : sem_s
IO : EventFam.t := EInput : IO ℕ | EOutput (o : ℕ) : IO unit
external [E A] (e : IO A) : BTL_S E A
get_mem [E] : BTL_S E Mem.t
set_mem [E] (m : Mem.t) : BTL_S E unit
```

Semantics of operators

$$[\![\texttt{o : op}]\!]_{\text{op}}\ \texttt{(args : list val) : BTL_S } \varnothing\ \texttt{val}$$

$$[\![\texttt{Const c}]\!]_{\text{op}}\ \texttt{[]} := \texttt{ret c} \qquad [\![\texttt{Add}]\!]_{\text{op}}\ \texttt{[m; n]} := \texttt{ret (m + n)}$$

$$[\![\texttt{Div}]\!]_{\text{op}}\ \texttt{[m; n]} := \begin{cases} \texttt{ub} & \text{if } \texttt{n} = 0 \\ \texttt{ret m/n} & \text{otherwise} \end{cases} \qquad [\![\texttt{Pick}]\!]_{\text{op}}\ \texttt{[]} := \texttt{b} \leftarrow \texttt{any bool; ret} \begin{cases} 1 \text{ if b} \\ 0 \text{ otherwise} \end{cases}$$

$$[\![\texttt{Input}]\!]_{\text{op}}\ \texttt{[]} := \texttt{external EInput} \qquad [\![\texttt{Output}]\!]_{\text{op}}\ \texttt{[n]} := \texttt{external (EOutput n);; ret } 0$$

$$[\![\texttt{Read}]\!]_{\text{op}}\ \texttt{[p]} := \texttt{m} \leftarrow \texttt{get_mem; ret (m p)} \qquad [\![\texttt{Write}]\!]_{\text{op}}\ \texttt{[p; v]} := \begin{pmatrix} \texttt{m} \leftarrow \texttt{get_mem;} \\ \texttt{set_mem [m, p} \mapsto \texttt{v];;} \\ \texttt{ret } 0 \end{pmatrix}$$

Semantics of instructions

```
CallE : EventFam.t := mk_call (f : fct_name) (args : list val) : CallE val
outcome : Type := OContinue (ve : VEnv.t) | ORet (r : val) | OGoto (trg : label) (ve : VEnv.t)
```

$$[\![\texttt{a : instr}]\!]_{\text{instr}}\ \texttt{(ve : VEnv.t) : BTL_S CallE outcome}$$

$$[\![\texttt{skip}]\!]_{\text{instr}}\ \texttt{ve} := \texttt{ret (OContinue ve)}$$

$$[\![\texttt{return r}]\!]_{\text{instr}}\ \texttt{ve} := \texttt{ret (ORet (ve r))}$$

$$[\![\texttt{goto l}]\!]_{\text{instr}}\ \texttt{ve} := \texttt{ret (OGoto l ve)}$$

$$[\![\texttt{dst = op(args)}]\!]_{\text{instr}}\ \texttt{ve} := \texttt{v} \leftarrow \texttt{lift_ev_emp } ([\![\texttt{op}]\!]_{\text{op}}\ \texttt{ve(args)})\texttt{; ret (OContinue [ve, dst} \mapsto \texttt{v])}$$

$$[\![\texttt{dst = f(args)}]\!]_{\text{instr}}\ \texttt{ve} := \texttt{v} \leftarrow \texttt{trigger (mk_call f ve(args)); ret (OContinue [ve, dst} \mapsto \texttt{v])}$$

$$[\![\texttt{a}_0\texttt{; a}_1]\!]_{\text{instr}}\ \texttt{ve} := \texttt{o} \leftarrow [\![\texttt{a}_0]\!]_{\text{instr}}\ \texttt{ve;} \begin{cases} [\![\texttt{a}_1]\!]_{\text{instr}}\ \texttt{ve'} & \text{if } \texttt{o = OContinue ve} \\ \texttt{ret o} & \text{otherwise} \end{cases}$$

$$[\![\texttt{if c \{a}_0\texttt{\} else \{a}_1\texttt{\}}]\!]_{\text{instr}}\ \texttt{ve} := [\![\texttt{a}_0 \text{ if } \texttt{ve c; a}_1 \text{ otherwise}]\!]_{\text{instr}}\ \texttt{ve}$$

Semantics of isolated functions

$$[\![\texttt{fn f(xargs) \{body\} : function}]\!]_{\text{func}}\ \texttt{(vargs : list val) : BTL_S CallE val} :=$$
$$\texttt{iter} \left(\lambda(\texttt{l, ve}).\ \texttt{o} \leftarrow [\![\texttt{body@l}]\!]_{\text{instr}}\ \texttt{ve;} \begin{cases} \texttt{ub} & \text{if } \texttt{o = OContinue _} \\ \texttt{ret (Break r)} & \text{if } \texttt{o = ORet r} \\ \texttt{ret (Continue (l', ve'))} & \text{if } \texttt{o = OGoto l' ve'} \end{cases} \right)$$
$$(\texttt{entry}, [\texttt{xargs} \mapsto \texttt{vargs}])$$

Semantics of program

$$[\![\texttt{pr : program}]\!]_{\text{prog}}\ (\texttt{vargs}_0\ \texttt{: list val}) : \texttt{BTL_S } \varnothing\ \texttt{val} :=$$
$$\texttt{mrec } (\lambda(\texttt{mk_call f vargs}).\ \texttt{map_ev Ev.inl } ([\![\texttt{pr@f}]\!]_{\text{func}}\ \texttt{vargs})) \ (\texttt{mk_call main vargs}_0)$$

Figure 4.4: Denotational semantics of MiniBTL. The domain `BTL_S` is defined in section 4.5. The present definition only relies on the fact that it implements the interface of `SMonad`, and has some additional operators `external`, `get_mem` and `set_mem`. [SMonadExamples.MiniBTL.Semantics]

4.2 Axiomatization

The SMonad library is centered around an interface satisfied by several domains suitable to define denotational semantics. The interface fixes the signatures of domains (section 4.2.1) and of some core operators (section 4.2.2). Those first two sections describe intuitively the semantics represented by the operators. We capture formally those intuitions by axiomatizing the interaction of the operators with a heterogeneous refinement relation (section 4.2.3). We also axiomatize some optional operators to expose additional effects in domains which support them (section 4.2.4).

Figure 4.4 illustrates the use of the library with a denotational definition of the semantics of MiniBTL. The definition and concepts it depends on are introduced in the following sections. Its domain `BTL_S` has the signature described in section 4.2.1, and will be defined in section 4.5. This domain implements the axiomatization of SMonad. In particular, it provides the operators described in section 4.2.2. Figure 4.4 uses those operators to define the semantics of the components of MiniBTL. We proved the equivalence of this denotational semantics with the operational one (figure 4.2), using tools described in section 5.5.

4.2.1 Domain

The interface of SMonad is parameterized by a family `S` of types used to represent semantics. For instance, we use some domain `BTL_S` (defined in section 4.5) for MiniBTL. We need to represent not only the semantics of full programs but also the semantics of their components. Semantics may thus interact with a context. The context contains both the external environment of the program and other components of the program. The family `S` is thus indexed by elements describing how the semantics interact with their context. We distinguish two kinds of interactions, namely return values and events. We describe those interactions in the next paragraphs. Formally, the sort `sem_s` of `S` has thus two indexes (with types described below) representing the kind of interactions used by semantics:

$$\texttt{sem_s} := \texttt{EventFam.t} \to \texttt{Type} \to \texttt{Type}$$

Given a family of events `E` and a type `A`, an inhabitant of `S E A` represents a semantics which emits events of `E` and returns values of type `A`.

Returning values When reaching the end of their execution, semantics may provide a result to their context. We describe the sets of values that a semantics can return using a `Type`. For instance in figure 4.4:

- $[\![.]\!]_{\text{op}}$ returns the value the operator evaluates to.
- $[\![.]\!]_{\text{instr}}$ returns a `outcome` describing both the values of the variables after the instruction and the way the instruction terminated.

Semantics without meaningful result (such as `set_mem`) have a return type `unit` with a single inhabitant. We use the empty type, $\varnothing$, for semantics which never terminate by returning to their context.

Emitting events Semantics can only return a value at the end of their execution. Using events, we more generally allow interactions in the middle of the execution of semantics. We use events to represent function calls and interactions with the external environment. When emitting an event, a semantics provides a value to its context and waits for its answer. The semantics then continues its execution, which depends on the answer provided by the context. We represent the set of events that a semantics can emit using a family with sort $\texttt{EventFam.t} := \texttt{Type} \to \texttt{Type}$. We index the events by the type of the answer expected from the context. That is, given a family $\texttt{E} : \texttt{EventFam.t}$ of events and a type `A`, the type `E A` contains the events of `E` for which the context replies with a value of type `A`. The sort `EventFam.t` has structures similar to `Type`. In particular, it has categories of functions and relations, described in figure 4.5 and the next sections.

Following an idea of McBride [74], we represent internal function calls using events. In

	Type	**EventFam.t := Type → Type**
empty	$\varnothing$	$\varnothing := \lambda_.\, \varnothing$
sum	$\mathtt{A} + \mathtt{B} := \mathtt{inl}\ (\mathtt{x} : \mathtt{A}) \mid \mathtt{inr}\ (\mathtt{y} : \mathtt{B})$	$\mathtt{D} + \mathtt{E} := \lambda \mathtt{A}.\, \mathtt{D}\ \mathtt{A} + \mathtt{E}\ \mathtt{A}$
	Functions (section 4.2.2)	
morphisms	$\mathtt{Fun}\ \mathtt{A}\ \mathtt{B} := \mathtt{A} \to \mathtt{B}$	$\mathtt{EventFam.Fun.t}\ \mathtt{D}\ \mathtt{E} := \forall [\mathtt{A}]\ (\mathtt{d} : \mathtt{D}\ \mathtt{A}).\ \{\mathtt{B} : \mathtt{Type}; \mathtt{e} : \mathtt{E}\ \mathtt{B}; \mathtt{f} : \mathtt{Fun}\ \mathtt{B}\ \mathtt{A}\}$
identity	$\mathtt{id} := \lambda \mathtt{x}.\, \mathtt{x}$	$\mathtt{id} := \lambda \mathtt{e}.\, \{\mathtt{e} := \mathtt{e}; \mathtt{f} := \mathtt{id}\}$
composition	$\mathtt{f}; \mathtt{g} := \lambda \mathtt{x}.\, \mathtt{g}\ (\mathtt{f}\ \mathtt{x})$	$(\mathtt{fe}; \mathtt{ge}) := \lambda \mathtt{e}.\ \left\{\begin{array}{l}\mathtt{e} := (\mathtt{ge}\ (\mathtt{fe}\ \mathtt{e}).\mathtt{e}).\mathtt{e} \\ \mathtt{f} := (\mathtt{ge}\ (\mathtt{fe}\ \mathtt{e}).\mathtt{e}).\mathtt{f}; (\mathtt{fe}\ \mathtt{e}).\mathtt{f}\end{array}\right\}$
	Relations (section 4.2.3)	
morphisms	$\mathtt{Rel}\ \mathtt{A}\ \mathtt{B} := \mathtt{A} \to \mathtt{B} \to \mathtt{Prop}$	$\mathtt{EventFam.Rel.t}\ \mathtt{D}\ \mathtt{E} := \forall [\mathtt{A}\,\mathtt{B} : \mathtt{Type}]\ (\mathtt{d} : \mathtt{D}\ \mathtt{A})\ (\mathtt{e} : \mathtt{E}\ \mathtt{B})\ (\mathtt{P} : \mathtt{Rel}\ \mathtt{A}\ \mathtt{B}).\ \mathtt{Prop}$
identity	$\mathtt{id} := \lambda \mathtt{x}\,\mathtt{y}.\, (\mathtt{x} = \mathtt{y})$	$\dfrac{}{\mathtt{id}\ \mathtt{e}\ \mathtt{e}\ \mathtt{id}}$
composition	$\mathtt{R}_0; \mathtt{R}_1 := \lambda \mathtt{x}\,\mathtt{z}.\, \exists \mathtt{y}.\, \mathtt{R}_0\ \mathtt{x}\ \mathtt{y} \wedge \mathtt{R}_1\ \mathtt{y}\ \mathtt{z}$	$\dfrac{\mathtt{Re}_0\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{P}_0 \qquad \mathtt{Re}_1\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{P}_1}{(\mathtt{Re}_0; \mathtt{Re}_1)\ \mathtt{e}_0\ \mathtt{e}_2\ (\mathtt{P}_0; \mathtt{P}_1)}$
order	$\mathtt{R}_0 \leq \mathtt{R}_1 := \forall \mathtt{x}\,\mathtt{y}.\, \mathtt{R}_0\ \mathtt{x}\ \mathtt{y} \Rightarrow \mathtt{R}_1\ \mathtt{x}\ \mathtt{y}$	$\mathtt{Re}_0 \leq \mathtt{Re}_1 := \forall \mathtt{d}\,\mathtt{e}\,\mathtt{P}_0.\, \mathtt{Re}_0\ \mathtt{d}\ \mathtt{e}\ \mathtt{P}_0 \Rightarrow \exists \mathtt{P}_1.\, \mathtt{Re}_1\ \mathtt{d}\ \mathtt{e}\ \mathtt{P}_1 \wedge \mathtt{P}_1 \leq \mathtt{P}_0$
empty	$\bot := \lambda _\,_.\, \mathtt{False}$	$\bot := \lambda ___.\, \mathtt{False}$
sum	$\dfrac{\mathtt{R}_0\ \mathtt{x}\ \mathtt{y}}{(\mathtt{R}_0 + \mathtt{R}_1)\ (\mathtt{inl}\ \mathtt{x})\ (\mathtt{inl}\ \mathtt{y})}$ $\dfrac{\mathtt{R}_1\ \mathtt{x}\ \mathtt{y}}{(\mathtt{R}_0 + \mathtt{R}_1)\ (\mathtt{inr}\ \mathtt{x})\ (\mathtt{inr}\ \mathtt{y})}$	$\mathtt{Re}_0 + \mathtt{Re}_1 := \lambda \mathtt{d}_2\,\mathtt{e}_2\,\mathtt{P}.\, ((\lambda \mathtt{d}\,\mathtt{e}.\, \mathtt{Re}_0\ \mathtt{d}\ \mathtt{e}\ \mathtt{P}) + (\lambda \mathtt{d}\,\mathtt{e}.\, \mathtt{Re}_1\ \mathtt{d}\ \mathtt{e}\ \mathtt{P}))\ \mathtt{d}_2\ \mathtt{e}_2$
converse	$\dagger \mathtt{R} := \lambda \mathtt{x}\,\mathtt{y}.\, \mathtt{R}\ \mathtt{y}\ \mathtt{x}$	$\dfrac{\mathtt{Re}\ \mathtt{e}\ \mathtt{d}\ \mathtt{P}}{(\dagger \mathtt{Re})\ \mathtt{d}\ \mathtt{e}\ \dagger \mathtt{P}}$
of function	$\mathtt{of_fun}\ \mathtt{f} := \lambda \mathtt{x}\,\mathtt{y}.\, (\mathtt{y} = \mathtt{f}\ \mathtt{x})$	$\dfrac{}{(\mathtt{of_fun}\ \mathtt{fe})\ \mathtt{e}\ (\mathtt{fe}\ \mathtt{e}).\mathtt{e}\ \ \dagger(\mathtt{of_fun}\ (\mathtt{fe}\ \mathtt{e}).\mathtt{f})}$

Figure 4.5: Mathematical structures on Type and families of events. [SMonad.Event]

figure 4.4, the semantics of instruction $[\![.]\!]_{\text{instr}}$ thus emits events in the family `CallE`. An event `mk_call f args` in this family contains the name `f` of the callee and the values `args` given as arguments. It has `val` as index and is thus answered with a single value, the value returned by the callee.

External calls are also represented by a family of events (`IO` for MiniBTL). However, in figure 4.4, those events are part of the effects of the domain `BTL_S`, and they do not appear in the index of `BTL_S` in the signature of the semantics. More generally and contrary to other approaches based on Interaction Trees [102; 105], we do not use the event family index of the domain to represent effects. Instead, we directly denote programs in a domain `S` which support all effects we are interested in. Our approach enables transforming effectful computations, at the cost of working with an axiomatic theory (section 4.6.3).

We could have simplified `sem_s` by considering the end of the execution as a dedicated event with empty answer type. However, in practice, return values are used more often than events when defining semantics. The signature with a distinguished return type simplifies the definitions.

4.2.2 Operators

The `SMonad` library fixes the signatures of some operators that build elements of the domain `S`. We will use figure 4.4 to illustrate how they are used to define semantics. Most of those operators were already defined on Interaction Trees [102]. In Rocq, the operators are defined as typeclasses. In this manuscript, we thus use the same names (e.g., `ret`) for the operators on an abstract domain (`S` here), and for the instances on a particular domain (e.g., `BTL_S` in figure 4.4).

The operators `ret` and `bind` define monads `S E` for each event family `E`:

`ret` `[E A] (x : A) : S E A` : return a value without any side effect. It is used to define the semantics of pure operators, such as $[\![\texttt{Add}]\!]_{\text{op}}$ in figure 4.4.

`bind` `[E A B] (u : S E A) (k : A → S E B) : S E B` : execute `u` then `k`. The second semantics (`k`) depends on the value returned by the first semantics (`u`). This dependency is represented using a function of Rocq (`A → _`). This enables the use of all pure functions of the host language to define semantics. For instance, the "$\texttt{a}_0$ if `ve c`; $\texttt{a}_1$ otherwise" of the semantics of conditionals is defined in Rocq by pattern matching. We use the notations "`x ← u; k`" for `bind u (λx. k)` and "`u;; k`" for `bind u (λ_. k)`. The operator `bind` is used to define the semantics of sequences ($[\![.\,;\,.]\!]_{\text{instr}}$). The value returned by the semantics of the left-hand side instruction is used both to express data dependencies (the values of variables) and for the control flow.

Two operators `trigger` and `interp` operate on events in a way similar to how `ret` and `bind` operate over return values. Like Interaction Trees, we use *event handlers*, to associate semantics to a family of events. Given two families of events `D` and `E`, an event handler `h` of `D` into `E` is a function with type:

$$\texttt{handler D E} := \forall [\texttt{A}]\,(\texttt{d} : \texttt{D A}).\,\texttt{S E A}$$

That is, for any event `d` of `D`, `h d` is a semantics which returns an answer for `d`, and which emits events in `E`. A particular case of event handlers is one where the semantics associated to any event of `D` consists in emitting an event of `E`. Those handlers are useful to insert injections in semantics, between formally distinct event families. They correspond to functions `EventFam.Fun.t D E` between event families (figure 4.5). A function `f : EventFam.Fun.t D E` maps any event `d` in `D` to:

- An event `(f d).e` of `E`, with an answer type `(f d).B` a priori different from the answer type of `d`.
- A function `(f d).f` which maps answers to `(f d).e` into answers to `d`.

A particular case of functions between event families is that of functions that map events into events with the same answer type, so that the function between answers is the identity. Such functions are described more concisely with types $\forall$`A. D A → E A`. In particular, `Ev.inl : EventFam.Fun.t D (D + E)`

and `Ev.inr : EventFam.Fun.t E (D + E)` define the injections into disjoint unions of families of events. We sometimes convert implicitly functions `∀A. D A → E A` into `EventFam.Fun.t D E`.

`trigger [E A] (e : E A) : S E A` : emit an event `e` and return its answer. Since internal calls are represented by events, we use it to define the semantics of function calls.

`interp [D E] (h : handler D E) [A] (u : S D A) : S E A` : replace the events of `u` by their semantics defined by `h`. The particular case where `h` is a function `f : EventFam.Fun.t D E` is noted `map_ev f u`. We also define `lift_ev_emp [E A] (u : S ∅ A) : S E A` for the particular case where the signature of `u` guarantees that it does not emit any events. This operator enables the use of the semantics of operators (which do not emit any events) in the semantics of instructions (which may emit call events).

The core axiomatization of `SMonad` includes the effect of divergence, that is, infinite executions. Divergence may be introduced by two operators:

`iter [E A B] (f : A → S E (A + B)) (ini : A) : S E B` : repeatedly execute `f`. This operator represents a loop with a state of type `A`. Its initial value is `ini`. At each iteration, `f` is executed with the current state as argument. If it returns a value of type `B`, then `iter` returns this value. Otherwise, the iteration continues from the value of type `A` returned by `f`. We use aliases `Continue` and `Break` for `inl` and `inr` to emphasize the semantics of the two kinds of results. In MiniBTL, `iter` is used to define the semantics of CFGs. The state is composed of the label of the next block to execute and of the current values of the variables.

`mrec [D E] (f : handler D (D + E)) [Aini] (ini : D Aini) : S E Aini` : set of mutually recursive functions. In `iter`, the semantics `f` must have terminated for the next iteration to start. This is enough to represent loops or terminal calls, but not general function calls, which return to the caller. The operator `mrec` overcomes this limitation by using events to represent function calls. It is parameterized by a family of events `D` which represents function calls. It takes as argument a handler of `D` which defines the semantics of each function call. Those semantics may themselves perform calls by emitting events in `D`. The operator `mrec` then defines the semantics of an initial call `ini` by recursively replacing the events of `D` by their semantics. The resulting semantics no longer emits events in `D`. MiniBTL uses `mrec` in $[\![.]\!]_{\text{prog}}$ to combine the semantics of functions.

4.2.3 Refinement and axioms

Once the semantics of programs are defined, one wants to reason on them. In verified compilation, the goal is to prove a refinement between the source and target program. Like the definitions of semantics, proofs of refinements can often be decomposed into refinements between the semantics of the components of the programs. However, such decompositions may require a notion of refinement more expressive than the one used on closed programs. Indeed, compilers preserve the interaction between programs and their environment, but the interactions between the components may be different in the source and target program. When stating a refinement between a program component and its transformation, we thus need to specify how their interactions with their respective contexts are related. We use a standard heterogeneous relation on types, `Rel` (figure 4.5), to relate return values. We also need to relate the events emitted by the semantics, and the answers received from their contexts. We introduce the following notion for this purpose:

Definition 4.4 Given two families of events `D` and `E`, a *relation on events* between `D` and `E` is a predicate which relates pairs of events (`d`, `e`) to some (zero, one or several) relations ("post-conditions", `P`) on their answers. Formally:

`EventFam.Rel.t D E := ∀[A B] (d : D A) (e : E B) (P : Rel A B). Prop`

Such relations on events capture one-to-one, execution order preserving, correspondences between the events emitted by the source and target semantics. Intuitively, for a refinement to hold, the target semantics must emit an event `e` exactly when the source semantics emits an event `d`. Moreover, those events must be related by the chosen relation `Re`. That is, there must exist a post-condition `P` such that `Re d e P`. In exchange, we assume in the continuation of the proof of refinement that the answers returned by the contexts are related by `P`. Those relations on events have categorical structure (figure 4.5). Relations between the same event families are ordered: like relations on types, relations on events are a particular case of a 2-category where the categories on the sets of morphisms are pre-orders. Given two relations $\texttt{Re}_0$ and $\texttt{Re}_1$ in `EventFam.Rel.t E₀ E₁`, we define $\texttt{Re}_0 \leq \texttt{Re}_1$ so that it holds if $\texttt{Re}_0$ is more restrictive for the compared semantics (we say that $\texttt{Re}_0$ is *stronger* than $\texttt{Re}_1$). Formally, whenever $\texttt{Re}_0$ relates two events, $\texttt{Re}_1$ must also relate them. Since the answers are provided by the context, the order is reversed for the post-conditions. That is, the post-condition $\texttt{P}_1$ given by $\texttt{Re}_1$ must be stronger than the post-condition $\texttt{P}_0$ of $\texttt{Re}_0$ (i.e. $\texttt{P}_0$ is satisfied for any answer allowed by $\texttt{P}_1$).

We define the signature of refinements, using our relations on events and standard relations on return values:

Definition 4.5 A *refinement relation* on a domain `S` is a heterogeneous relation between elements of `S`. Such a relation `ref` has type:

```
ref [E₀ E₁ : EventFam.t] (Re : EventFam.Rel.t E₀ E₁)
  [A₀ A₁ : Type] (R : Rel A₀ A₁)
  (u₀ : S E₀ A₀) (u₁ : S E₁ A₁) : Prop
```

The interface of `SMonad` is parameterized (using a typeclass) by a refinement relation on the domain `S` considered. This refinement is called `sem_ref`. A refinement `sem_ref Re R u₀ u₁` is read as "$\texttt{u}_1$ refines $\texttt{u}_0$". Its meaning should be a kind of implication: "if $\texttt{u}_0$ is correct then $\texttt{u}_1$ also is". In a compilation setting, $\texttt{u}_0$ is a source semantics and $\texttt{u}_1$ a target semantics. We write $\texttt{u}_0 \gtrsim_{\texttt{Re},\texttt{R}} \texttt{u}_1$ for `sem_ref Re R u₀ u₁`. Since it is a refinement, `sem_ref` can be asymmetric. To factorize proofs, we define a stronger heterogeneous relation as the conjunction of the refinements in both directions (named `sem_eq`):

$$\texttt{u}_0 \approx_{\texttt{Re},\texttt{R}} \texttt{u}_1 := \left(\texttt{u}_0 \gtrsim_{\texttt{Re},\texttt{R}} \texttt{u}_1\right) \wedge \left(\texttt{u}_1 \gtrsim_{\dagger\texttt{Re},\dagger\texttt{R}} \texttt{u}_0\right)$$

We leave the relations implicit in $\gtrsim_{_,_}$ and $\approx_{_,_}$ when they are the identities of their categories. In particular, $(\gtrsim) := \left(\gtrsim_{\texttt{id},\texttt{id}}\right)$ and $(\approx) := \left(\approx_{\texttt{id},\texttt{id}}\right)$ are homogeneous relations. The latter is named `sem_eeq`.

We then axiomatize the properties of `sem_ref` and its interaction with the operators of section 4.2.2.

Those axioms capture several kinds of properties:

Properties of `sem_ref` We assume that the refinement relation is:

Reflexive that is, any semantics refines itself, with unchanged interactions with its context.

Monotone a refinement may be weakened by changing the relations on interactions to relations which are easier to satisfy for the semantics.

Transitive this property provides a way of decomposing the proof of a refinement into two refinements involving an intermediate semantics. The relations on the interactions for the full refinement are obtained by composing the relations of the two refinements. One of the applications of transitivity is the composition of the refinements proven on individual passes of compilation pipelines. But it also crucially enables rewriting semantics up to `sem_eeq` during proofs of individual passes. That is, we derive a reasoning principle:

$$\frac{\texttt{u}_0 \approx \texttt{u}_0' \qquad \texttt{u}_1 \approx \texttt{u}_1' \qquad \texttt{u}_0' \gtrsim_{\texttt{Re},\texttt{R}} \texttt{u}_1'}{\texttt{u}_0 \gtrsim_{\texttt{Re},\texttt{R}} \texttt{u}_1}\ \texttt{sem_ref_morph_u}$$

We derive similar properties for `sem_eq`. Those properties also imply that `sem_ref id id` is a pre-order and that `sem_eeq` is an equivalence.

Morphisms We assume that the operators are morphisms for `sem_ref`. That is, when applied to related results, they produce semantics related by `sem_ref`. Those properties enable the decomposition of refinements between similar semantics into refinements between their components. We derive similar properties for `sem_eq`, for instance:

$$\frac{\forall x_0\ x_1.\ \texttt{RA}\ x_0\ x_1 \Rightarrow \texttt{f}_0\ x_0 \approx_{\texttt{Re},(\texttt{RA}+\texttt{RB})} \texttt{f}_1\ x_1 \qquad \texttt{RA}\ \texttt{ini}_0\ \texttt{ini}_1}{\texttt{iter}\ \texttt{f}_0\ \texttt{ini}_0 \approx_{\texttt{Re},\texttt{RB}} \texttt{iter}\ \texttt{f}_1\ \texttt{ini}_1}\ \texttt{eq_iter}$$

We also derive that `sem_eeq` is a congruence.

Monad laws The monad laws capture equationally the expected semantics of `ret` and `bind`.

Interpretation laws The operators `trigger` and `interp` satisfy equations similar to the monad law of `ret` and `bind`, but at the level of events instead of the return values.

Distribution of interpretation Interpretation of events distributes over the operators `bind` and `iter` since we expect those operators to forward the events emitted by their components. We also derive that `interp` leaves `ret` unchanged.

Iteration laws The expected semantics of `iter` is captured equationally. `iter_unfold` unrolls the first iteration of an `iter`, according to our informal description of its behavior. Alone, it only allows unfolding iterations finitely many times. In section 4.4.1, we will show that its combination with `iter_codiagonal` enables proofs of more general loop transformations when the iteration can diverge.

Recursion laws We expect `mrec` to satisfy equations similar to `iter`, but on events instead of return values. Like for `iter`, `mrec_unfold` allows unfolding a call tree up to a finite depth. In section 4.4.2, we will show that its combination with `mrec_codiagonal` enables reasoning on unbounded recursion and can justify inlining transformations. The axiom `iter_as_mrec` asserts that `iter` is equivalent to a `mrec` performing tail-calls.

4.2.4 Additional effects

The core interface of `SMonad` only supposes that the domain `S` can represent semantics which diverge and emit events. Some domains support additional effects while satisfying the core interface. Semantics with those additional effects are built using additional operators of the domains. In some cases, it is even possible to reason on those effects with the equational and relational approaches. Operators and properties related to an effect can be defined for specific domains, but we also axiomatize some of them.

TODO (rocq)

- principe d'introduction avec ces axiomes

Properties of `sem_ref`

$$\frac{}{u \gtrsim_{id,id} u}\ \texttt{sem_ref_refl} \qquad \frac{Re_0 \leq Re_1 \quad R_0 \leq R_1 \quad u_0 \gtrsim_{Re_0,R_0} u_1}{u_0 \gtrsim_{Re_1,R_1} u_1}\ \texttt{sem_ref_mono}$$

$$\frac{u_0 \gtrsim_{Re_0,R_0} u_1 \quad u_1 \gtrsim_{Re_1,R_1} u_2}{u_0 \gtrsim_{(Re_0;Re_1),(R_0;R_1)} u_2}\ \texttt{sem_ref_trans}$$

Morphisms

$$\frac{R\ x_0\ x_1}{\texttt{ret}\ x_0 \gtrsim_{Re,R} \texttt{ret}\ x_1}\ \texttt{ref_ret} \qquad \frac{Re\ e_0\ e_1\ P}{\texttt{trigger}\ e_0 \gtrsim_{Re,P} \texttt{trigger}\ e_1}\ \texttt{ref_trigger}$$

$$\frac{u_0 \gtrsim_{Re,RA} u_1 \quad \forall x_0\ x_1.\ RA\ x_0\ x_1 \Rightarrow k_0\ x_0 \gtrsim_{Re,RB} k_1\ x_1}{\texttt{bind}\ u_0\ k_0 \gtrsim_{Re,RB} \texttt{bind}\ u_1\ k_1}\ \texttt{ref_bind}$$

$$\frac{\forall d_0\ d_1\ P.\ Rd\ d_0\ d_1\ P \Rightarrow h_0\ d_0 \gtrsim_{Re,P} h_1\ d_1 \quad u_0 \gtrsim_{Rd,R} u_1}{\texttt{interp}\ h_0\ u_0 \gtrsim_{Re,R} \texttt{interp}\ h_1\ u_1}\ \texttt{ref_interp}$$

$$\frac{\forall x_0\ x_1.\ RA\ x_0\ x_1 \Rightarrow f_0\ x_0 \gtrsim_{Re,(RA+RB)} f_1\ x_1 \quad RA\ ini_0\ ini_1}{\texttt{iter}\ f_0\ ini_0 \gtrsim_{Re,RB} \texttt{iter}\ f_1\ ini_1}\ \texttt{ref_iter}$$

$$\frac{\forall d_0\ d_1\ P.\ Rd\ d_0\ d_1\ P \Rightarrow f_0\ d_0 \gtrsim_{(Rd+Re),P} f_1\ d_1 \quad Rd\ ini_0\ ini_1\ Pini}{\texttt{mrec}\ f_0\ ini_0 \gtrsim_{Re,Pini} \texttt{mrec}\ f_1\ ini_1}\ \texttt{ref_mrec}$$

Monad laws

$$\frac{}{\texttt{bind}\ (\texttt{ret}\ x)\ k \approx k\ x}\ \texttt{mon_ret_l}$$

$$\frac{}{\texttt{bind}\ u\ \texttt{ret} \approx u}\ \texttt{mon_ret_r} \qquad \frac{}{\texttt{bind}\ (\texttt{bind}\ u\ f)\ g \approx \texttt{bind}\ u\ (\lambda x.\texttt{bind}\ (f\ x)\ g)}\ \texttt{mon_assoc}$$

Interpretation laws

$$\frac{}{\texttt{interp}\ h\ (\texttt{trigger}\ d) \approx h\ d}\ \texttt{interp_trigger} \qquad \frac{}{\texttt{interp}\ \texttt{trigger}\ u \approx u}\ \texttt{interp_trigger_h}$$

$$\frac{}{\texttt{interp}\ h_1\ (\texttt{interp}\ h_0\ u) \approx \texttt{interp}\ (\lambda e.\texttt{interp}\ h_1\ (h_0\ e))\ u}\ \texttt{interp_assoc}$$

Distribution of interpretation

$$\frac{}{\texttt{interp h (bind u k)} \approx \texttt{bind (interp h u) } (\lambda \texttt{x}.\,\texttt{interp h (k x)})}\ \texttt{interp_bind}$$

$$\frac{}{\texttt{interp h (iter f ini)} \approx \texttt{iter } (\lambda \texttt{x}.\,\texttt{interp h (f x)})\ \texttt{ini}}\ \texttt{interp_iter}$$

Iteration laws

$$\frac{}{\texttt{iter f x} \approx \texttt{r} \leftarrow \texttt{f x};\begin{cases}\texttt{iter f x}' & \text{if } \texttt{r} = \texttt{inl x} \\ \texttt{ret y} & \text{if } \texttt{r} = \texttt{inr y}\end{cases}}\ \texttt{iter_unfold}$$

$$\frac{\texttt{f} : \texttt{A} \to \texttt{S E } (\texttt{A} + (\texttt{A} + \texttt{B}))}{\texttt{iter } (\texttt{iter f})\ \texttt{x} \approx \texttt{iter } (\lambda \texttt{x}_1.\,\texttt{r} \leftarrow \texttt{f x}_1; \texttt{ret } (\texttt{mergeL r}))\ \texttt{x}}\ \texttt{iter_codiagonal}$$

where: $\texttt{mergeL } (\texttt{r} : \texttt{A} + (\texttt{A} + \texttt{B})) : \texttt{A} + \texttt{B} := \begin{cases}\texttt{inl x}_2 & \text{if } \texttt{r} = \texttt{inl x}_2 \\ \texttt{y} & \text{if } \texttt{r} = \texttt{inr y}\end{cases}$

Recursion laws

$$\frac{}{\texttt{mrec f ini} \approx \texttt{interp } \left(\lambda \texttt{c}.\begin{cases}\texttt{mrec f d} & \text{if } \texttt{c} = \texttt{inl d} \\ \texttt{trigger e} & \text{if } \texttt{c} = \texttt{inr e}\end{cases}\right) (\texttt{f ini})}\ \texttt{mrec_unfold}$$

$$\frac{\texttt{f} : \texttt{handler D } (\texttt{D} + (\texttt{D} + \texttt{E}))}{\texttt{mrec } (\texttt{mrec f})\ \texttt{ini} \approx \texttt{mrec } (\lambda \texttt{d}.\,\texttt{map_ev mergeL } (\texttt{f d}))\ \texttt{ini}}\ \texttt{mrec_codiagonal}$$

$$\frac{}{\texttt{iter f ini} \approx \texttt{mrec } \left(\begin{array}{l}\lambda(\texttt{d x}). \\ \quad \texttt{r} \leftarrow \texttt{map_ev Ev.inr } (\texttt{f x}); \\ \quad \begin{cases}\texttt{trigger } (\texttt{inl } (\texttt{d x}')) & \text{if } \texttt{r} = \texttt{inl x}' \\ \texttt{ret y} & \text{if } \texttt{r} = \texttt{inr y}\end{cases}\end{array}\right) (\texttt{d ini})}\ \texttt{iter_as_mrec}$$

where $\texttt{f} : \texttt{A} \to \texttt{S E } (\texttt{A} + \texttt{B})$ and `mrec` uses a family of events `D` generated by $\texttt{d } (\texttt{x} : \texttt{A}) : \texttt{D B}$

Figure 4.6: Core axioms of the `SMonad` library. [SMonad.Ref]

Axioms

$$\frac{}{\texttt{ret tt} \gtrsim_{\bot,\top} \texttt{any}_0\ \texttt{A}}\ \texttt{any0_trg_ref} \qquad \frac{\texttt{x} : \texttt{A}}{\texttt{any}_0\ \texttt{A} \gtrsim_{\bot,(\lambda \texttt{r}_.\,\texttt{r}=\texttt{x})} \texttt{ret tt}}\ \texttt{any0_src_ref}$$

Derived lemmas

$$\frac{\forall \texttt{x} : \texttt{A}.\ \texttt{u} \gtrsim_{\texttt{Re,R}} \texttt{v x}}{\texttt{u} \gtrsim_{\texttt{Re,R}} \texttt{bind } (\texttt{any A})\ \texttt{v}}\ \texttt{any_trg_ref_intro} \qquad \frac{\exists \texttt{x} : \texttt{A}.\ \texttt{u x} \gtrsim_{\texttt{Re,R}} \texttt{v}}{\texttt{bind } (\texttt{any A})\ \texttt{u} \gtrsim_{\texttt{Re,R}} \texttt{v}}\ \texttt{any_src_ref_intro}$$

Figure 4.7: Axioms and derived reasoning principle for non-deterministic choices

Axiom

$$\frac{}{\texttt{ub}_0 \gtrsim_{\bot,\bot} \texttt{ret tt}}\ \texttt{ub0_ref}$$

Derived lemmas

$$\frac{}{\texttt{bind ub u} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{v}}\ \texttt{ub_ref_intro}$$

$$\frac{\texttt{P}}{\texttt{ret tt} \gtrsim_{\top} \texttt{assert P}}\ \texttt{assert_trg_ref} \qquad \frac{}{\texttt{assert P} \gtrsim_{\top} \texttt{ret tt}}\ \texttt{assert_src_ref}$$

Figure 4.8: Axioms and derived reasoning principle for UBs

4.2.4.1 Non-determinism

We represent internal, demonic non-determinism (section 4.1.1) in our semantics using an operator $\texttt{any}_0$. Given a type `A`, the semantics $\texttt{any}_0\ \texttt{A} : \texttt{S}\ \varnothing\ \texttt{A}$ returns a value of type `A` chosen non-deterministically. It does not have any other side effects. In particular, such choice is not directly observable by the context of the semantics. Using `lift_ev_emp`, we derive an operator $\texttt{any}\ [\texttt{E}]\ \texttt{A} : \texttt{E}\ \texttt{A}$ which is easier to use when defining semantics. It is possible to reason on `any` during relational proofs using the axioms of figure 4.7. Those axioms relate $\texttt{any}_0$ to a pure computation `ret tt`. We also derive some reasoning principles which are easier to apply during proofs (and are equivalent to the axioms). A refinement can be understood as an implication between the correctness of the source program and the correctness of the target program. Because of the asymmetry of this implication, the demonic semantics of `any` affects the refinement differently depending on whether it is in the source or in the target. In the axioms, this is captured by the different post-conditions. In the derived principles, it is captured with different quantifiers.

- When `any` is in the target, the refinement must hold for all possible choices. We thus use a post-condition satisfied by all values in `any0_trg_ref` and quantify universally on the chosen value in `any_trg_ref_intro`.
- When `any` is in the source, it is enough to find a single value to justify the refinement. When applying `any0_src_ref`, one thus chooses a value `x` and the post-condition ensures that the value returned in the source is `x`. In `any_src_ref_intro` the existential quantifier lets us choose `x` during refinement proofs.

It is possible to give a proposition $\texttt{P} : \texttt{Prop}$ of Rocq as argument to `any`. Executions reaching $\texttt{any}\ \texttt{P}$ when `P` is not satisfied are halted invisibly without being taken into account in refinements. From the program perspective, $\texttt{any}\ \texttt{P}$ thus provides an assumption that `P` is satisfied after its execution. This behaves like a guard [36], often represented by an `assume` instruction in verification languages (e.g. Boogie [66]). In this thesis we use it to define the semantics of `Frag.joinable` in Frag IR (section 3.2.6). In particular, $\texttt{any}\ \texttt{False}$ represents a program which is always correct, that is, the dual of a UB. A proof of refinement can be terminated directly if it appears at the beginning of the target semantics. Such "stuck" but correct states do not appear in the semantics of a real machine. However, this effect is useful to reason on programs.

4.2.4.2 Undefined Behaviors

An Undefined Behavior (UB) is a program termination that should not happen during execution of correct programs. During proofs of refinements, UBs act as hypotheses when they appear on the source side, and as proof obligations when they appear on the target side. In compilation, UBs in the source thus act as guarantees given to the compiler. In semantics, we represent such points with an operator $\texttt{ub}_0 : \texttt{S}\ \varnothing\ \varnothing$. Its signature guarantees that it does not interact with its context: semantics that reach a UB are stuck. In particular, their continuation is irrelevant and can be replaced freely during refinements. For commodity, we derive using `bind` and `lift_ev_emp` an operator `ub` with an arbitrary event family and return type in its signature. Since UBs are unreachable in source programs, the compiler is even allowed to remove them. We capture this property using an axiom `ub0_ref`. This axiom makes `ub` the dual of $\texttt{any}\ \texttt{False}$. A proof of refinement can be terminated directly if `ub` appears at the beginning of the source semantics (`ub_ref_intro`). However, the axioms of `SMonad` to not allow exploiting "time-travelling" UBs, that is, if an event appear before them (section 4.6.6).

We derive from `ub` an `assert` operation which has UB if some specified proposition $\texttt{P} : \texttt{Prop}$ is not satisfied:

$$\texttt{assert}\ [\texttt{E}]\ (\texttt{P} : \texttt{Prop}) : \texttt{S}\ \texttt{E}\ \texttt{P} := \texttt{t} \leftarrow \texttt{any}\ (\texttt{P} + \neg\texttt{P}); \begin{cases} \texttt{ret pf} & \text{if } \texttt{t} = \texttt{inl pf} \\ \texttt{ub} & \text{if } \texttt{t} = \texttt{inr _} \end{cases}$$

This implementation relies on a non-deterministic choice to test if `P` is satisfied. The semantics of `assert P` is the dual of `any P`. When proving the correctness of a program, the former yields a proof obligation whereas the latter provides an assumption. Conversely, an `assert` provides an assumption to the compiler when it appears in the source program.

4.3 Using the theory to prove transformations

Section 4.3.1 describes an example of the relational reasoning approach enabled by the axioms of `SMonad`. We mainly use relations on events to reason on inter-procedural transformations, and section 4.3.2 gives examples of relations in this context. We derive unary predicates from `sem_ref`, to recover invariants of semantics during relational proofs (section 4.3.3).

4.3.1 Relational reasoning

As an example, this section considers an intra-procedural transformation between two given infinite loops. In the syntax of C, the transformation is:

```
void aux(int o) { output(o); }
int main() {
  for (int x = 0; true;) {
    x++;
    aux(x);
  }
}
```

→

```
void aux(int o) { output(o); }
int main() {
  for (int c = 1, x = 1; true;) {
    aux(x);
    x += c;
  }
}
```

We actually consider an equivalent transformation between two MiniBTL programs, where the loops are encoded using CFGs:

```
aux(o) { entry: output(o); return }
main() {
  entry:
    x = 0;
    goto loop
  loop:
    c = 1;
    x = x + c;
    aux(x);
    goto loop
}
```

→

```
aux(o) { entry: output(o); return }
main() {
  entry:
    x = 1;
    c = 1;
    goto loop
  loop:
    aux(x);
    x = x + c;
    goto loop
}
```

In the following, we will refer to those programs as $\texttt{pr}_0$ (source) and $\texttt{pr}_1$ (target). To justify the correctness of the transformation, our goal is to prove a refinement $[\![\texttt{pr}_0]\!]_{\text{prog}}\ [] \gtrsim [\![\texttt{pr}_1]\!]_{\text{prog}}\ []$ between their semantics. We start by proving a refinement between the transformed functions.

Example 4.3 The semantics of the target `main` function refines the semantics of the source `main` function:

$$[\![\texttt{pr}_0\texttt{@main}]\!]_{\text{func}}\ [] \gtrsim [\![\texttt{pr}_1\texttt{@main}]\!]_{\text{func}}\ []$$

The semantics $[\![.]\!]_{\text{func}}$ of isolated function is defined in figure 4.4. It does not depend on the program, but emits internal call events `CallE`.

Proof After unfolding the definition of $[\![.]\!]_{\text{func}}$, we obtain a refinement between two `iter`s with states `label` × `VEnv.t`. We use `ref_iter` to prove this refinement in a lock-step fashion. We instantiate the invariant of `ref_iter` to:

$$\texttt{RA}\ (\texttt{l}_0, \texttt{ve}_0)\ (\texttt{l}_1, \texttt{ve}_1) := (\texttt{l}_0 = \texttt{l}_1 \wedge (\texttt{l}_0 = \texttt{loop} \Rightarrow (\texttt{ve}_1\ \texttt{c} = 1 \wedge \texttt{ve}_1\ \texttt{x} = \texttt{ve}_0\ \texttt{x} + 1)))$$

The first conjunct reflects the fact that the corresponding blocks of the CFGs have the same labels. The second conjunct describes the values of the variables at the beginning of the block `loop`.
The axiom `ref_iter` has two premises, corresponding to the preservation and the initialization of the invariant. This second premise is satisfied: `RA (entry, []) (entry, [])` For the first premise, we reason by case analysis on the common label. The two blocks `entry` have pure semantics. By rewriting `mon_ret_l` several times, we reduce them to single operators `ret`s. We conclude using `ref_ret`.
The two blocks `loop` perform function calls by emitting events. After simplifying their semantics using `ref_ret` and the invariant `RA`, we obtain a goal:

$$\begin{array}{l}\texttt{trigger (mk_call aux [ve}_0\ \texttt{x} + 1\texttt{]);;}\\ \texttt{ret (Continue}\\ \quad\texttt{(loop, [ve}_0, \texttt{x} \mapsto \texttt{ve}_0\ \texttt{x} + 1\texttt{])}\texttt{)}\end{array} \quad \gtrsim_{\texttt{RA} + \texttt{id}} \quad \begin{array}{l}\texttt{trigger (mk_call aux [ve}_0\ \texttt{x} + 1\texttt{]);;}\\ \texttt{ret (Continue}\\ \quad\texttt{(loop, [ve}_1, \texttt{x} \mapsto \texttt{ve}_0\ \texttt{x} + 1\texttt{])}\texttt{)}\end{array}$$

The environment $\texttt{ve}_0$ (resp. $\texttt{ve}_1$) comes from the state of the `iter`. It contains the values of the source (resp. target) variables at the beginning of the block. The post-condition `RA + id` comes from `ref_iter`. It requires the two semantics to have the same kind of result (`Continue` or `Ret`). Moreover, they must reestablish the invariant `RA` when the iteration continues. The part `id` comes from the implicit post-condition of the refinements of the functions, but is not used here since the two iterations are infinite. Since the two semantics start with corresponding `trigger`s, we isolate them using `ref_bind`. We then relate them using `ref_trigger`. Here, the relation on the events is the identity, so the two call events must be equal, and we obtain the same result (but which is ignored) in both semantics. The second premise of `ref_bind` is a refinement between two `ret`s. We conclude using `ref_ret` and checking that the invariant `RA` is reestablished. □

We then derive the refinements of the programs from this local refinement.

Example 4.4 The semantics of the target program refines the semantics of the source:

$$[\![\texttt{pr}_0]\!]_{\text{prog}}\ [] \gtrsim [\![\texttt{pr}_1]\!]_{\text{prog}}\ []$$

Proof The semantics $[\![.]\!]_{\text{prog}}$ is defined using `mrec`, by composing the semantics of the functions in isolation. We thus apply `ref_mrec`. The transformation does not involve any inter-procedural invariant, so we instantiate the invariant `Rd` with the identity relation `id` on events. Similarly to `ref_iter`, we have to prove that functions preserve this invariant and that it holds for the initial call. This second premise holds for the invariant `id` since the two programs start with the same call to `main` and are implicitly compared with the identity relation as post-condition. For the first premise, we have to prove a refinement between any functions related by the invariant on call events. With the identity invariant, this premise simplifies to:

$$\forall \texttt{f args}.\ [\![\texttt{pr}_0\texttt{@f}]\!]_{\text{func}}\ \texttt{args} \gtrsim [\![\texttt{pr}_1\texttt{@f}]\!]_{\text{func}}\ \texttt{args}$$

That is, we must show that if we execute the same function call `f(args)`, then the target implementation $\texttt{pr}_1\texttt{@f}$ refines the source $\texttt{pr}_0\texttt{@f}$, performs the same function sub-calls and returns the same value. We prove this refinement by case analysis on `f`. For `aux`, we apply `sem_ref_refl`, since the function is not transformed. For `main`, we conclude using example 4.3. □

4.3.2 Using relation on events for inter-procedural transformations

In example 4.4, we applied `ref_mrec` with an identity relation on the internal call events. For some transformations, we instead need to specify an invariant between the source and target functions. In MiniBTL, such invariant is formally a relation `EventFam.Rel.t CallE CallE`. That is, it describes pairs of source and target calls (containing the identifier of the callee and its arguments), and for each related pair, a post-condition relating the source and target results of the call. Formally, relations on events can associate several post-conditions to a pair of call events, but we do not do it in practice.

We now give examples of practical uses of call invariants, by describing corresponding relation on events, defined as inductive predicates.

Example 4.5 Relations can describe transformations of the arguments of results, and even relate functions with different signatures. For instance,

$$\frac{}{\texttt{Re_tr}\ (\texttt{mk_call foo}\ [\texttt{n};\texttt{m}])\ (\texttt{mk_call foo}\ [\texttt{n}+\texttt{m}])\ (\lambda \texttt{a}_0\ \texttt{a}_1.\ \texttt{a}_1 = \texttt{n} + \texttt{a}_0)}$$

describes the transformation of a function `foo` taking two arguments $\texttt{n}$ and $\texttt{m}$ in the source, but a single argument in the target. It states that the argument of a target call is the sum of the arguments of the corresponding source call, and that the results $\texttt{a}_1$ of the target are equal to the sum of the source argument $\texttt{n}$ and result $\texttt{a}_0$.

Example 4.6 Even when the calls are not transformed, refinements may need some pre-conditions at the beginning of functions, and post-conditions on the results of calls. We turn those unary invariants into relations by enforcing the equality of the calls and results.
For instance, the relation on event `Re_pred` generated by:

$$\frac{\texttt{n} \leq 10}{\texttt{Re_pred}\ (\texttt{mk_call foo}\ [\texttt{n}])\ (\texttt{mk_call foo}\ [\texttt{n}])\ (\lambda \texttt{m}_0\ \texttt{m}_1.\ \texttt{m}_0 = \texttt{m}_1 \wedge \texttt{m}_0 \leq \texttt{n})}$$

propagates the invariants that the argument $\texttt{n}$ of `foo` is less than 10, and that its result $\texttt{m}$ is less than the argument $\texttt{n}$.
In section 4.3.3, we will also use this kind of invariants to define unary predicates on semantics.

Example 4.7 Functions with different names can be related, and the same source function may be related to several target functions. This could be used to generate specialized versions of functions. For instance, the relation `Re_spe` with two constructors:

$$\frac{}{\texttt{Re_spe}\ (\texttt{mk_call foo}\ [\texttt{n}])\ (\texttt{mk_call foo0}\ [\texttt{n}])\ \texttt{id}}$$

$$\frac{\texttt{n} \geq 10}{\texttt{Re_spe}\ (\texttt{mk_call foo}\ [\texttt{n}])\ (\texttt{mk_call foo1}\ [\texttt{n}])\ \texttt{id}}$$

relates the source function `foo` to two target functions. The first one, `foo_0`, can always be called and could be an unchanged copy of `foo`. The second one, `foo_1`, is a specialized version, which can only be called if the argument $\texttt{n}$ is greater than 10, and which could be optimized using this assumption.

Limitations of relation on events However, not all inter-procedural transformations can be proved by applying `ref_mrec` with an invariant on call events. A necessary condition is that the source and target call trees have the same structure. For instance, it does not work if some functions are inlined. However, `SMonad` contains other axioms on `mrec`, which enable proving more transformations. In particular, section 4.4.2 will show that inlining transformations are derivable using the axiomatization. Relations on events also do not handle some transformations where effects are moved across call boundaries. For instance, the following refinement between equivalent C programs cannot be proved by applying `ref_mrec`:

```
void aux() { ub; }                ≳   void aux() { }
int main() { aux(); }                 int main() { ub; aux(); }
```

Indeed, the refinement condition on `main` in isolation simplifies to `trigger` ... $\gtrsim$ `ub`, and the source `trigger` cannot be related to a corresponding target one. This particular transformation is derivable using `mrec_unfold`, since the recursion is finite.

4.3.3 Derived unary predicates

TODO

- post-condition
- format de `ve_1'`
- pas d'indice `P` dans le raffinement

Some transformations rely on invariants of the source program, for instance the typing or properties found by analyses. Those properties can be captured using a unary predicate, that is, a predicate on a single semantics. We derive such predicates from `sem_ref`, by relating the semantics to itself, with relations on events and on results asserting the equalities of the two copies of the values. For instance, using the relation on events `Re_pred` of the example 4.6, we can state invariants on the values of variables before and after an instruction `a` using a proposition:

$$\texttt{P} := \left(\forall \texttt{ve}_0.\ \texttt{ve}_0\ \ \texttt{y} \leq 10 \Rightarrow [\![\texttt{a}]\!]_{\text{instr}}\ \ \texttt{ve}_0 \gtrsim_{\texttt{Re_pred},(\lambda \texttt{o}\, \texttt{o}'.\, \texttt{o}=\texttt{o}' \wedge \exists \texttt{ve}_1.\, \texttt{o}=\texttt{OContinue}\ \texttt{ve}_1 \wedge \texttt{ve}_1\ \ \texttt{x} \geq \texttt{ve}_0\ \ \texttt{y})}\ [\![\texttt{a}]\!]_{\text{instr}}\ \ \texttt{ve}_0\right)$$

Ignoring the call events and exceptional outcomes, this proposition could be written as a Hoare triple, "$\{\texttt{y} \leq 10\}\ \texttt{a}\ \{\texttt{x} \geq \texttt{old(y)}\}$". However, the unary predicate derived from $\gtrsim$ only expresses properties that hold *if* the semantics reach the points where they are stated. In particular, diverging semantics admit all post-conditions (that is, we only express partial correctness). In a less standard way, this is also the case for UBs, since $\texttt{ub} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{ub}$ holds for any relations `Re` and `R`. But this is the notion we need in compilation, since analyses do not prove the absence of UBs, and instead assume it.
During proofs of refinements, we use invariants known on the source to strengthen the post-conditions of refinements of subcomponents. For instance, assume that `a` is transformed to $\texttt{a}'$ and that we already proved the source invariant `P` and a refinement $[\![\texttt{a}]\!]_{\text{instr}}\ \ \texttt{ve}_0 \gtrsim_{\texttt{R}} [\![\texttt{a}']\!]_{\text{instr}}\ \ \texttt{ve}'_0$. Then, under the assumption that $\texttt{ve}_0\ \ \texttt{y} \leq 10$, we can strengthen `R` into $(\lambda \texttt{o}\, \texttt{o}'.\, \exists \texttt{ve}_1.\, \texttt{o} = \texttt{OContinue}\ \texttt{ve}_1 \wedge \texttt{ve}_1\ \ \texttt{x} \geq \texttt{ve}_0\ \ \texttt{y} \wedge \texttt{R o o}')$, by composing the two refinements (using `sem_ref_trans`), then simplifying the relations (with `sem_ref_mono`).

Yoon et al. [103, §4.2] define a related notion of `image`, but as a kind of strongest post-condition. Our predicates are instead based on weakest-preconditions. In our development, we use those unary predicates to state and use the correctness of typing (section 6.5.2).

4.4 Derived properties

Section 4.3 explained the most common usage of `SMonad`, with a relational reasoning approach. However, the axiomatization also implies properties that cannot be derived only by relational reasoning. This is the case of loop transformations (section 4.4.1) and function inlining (section 4.4.2). On the contrary, section 4.4.3 shows that some refinements do not hold in non-trivial models.
We did not explore refinements involving both `iter` and `mrec` (e.g., tail call elimination), but they could maybe be derived similarly using `iter_as_mrec`.

4.4.1 Proving loop transformations using the codiagonal property

The proof approaches presented in section 4.3 do not work for some equivalences. It is notably the case for loop transformations. Consider the following equivalence between two infinite iterations:

Example 4.8 For any family of events `E`, type `A`, iteration body $\texttt{f} : \texttt{A} \to \texttt{S E A}$ and initial state $\texttt{xini} : \texttt{A}$:

$$\texttt{loop}_1 := \texttt{iter}\ \left(\begin{array}{l}\lambda \texttt{x}_0.\\ \quad \texttt{x}_1 \leftarrow \texttt{f}\ \texttt{x}_0;\\ \quad \texttt{ret}\ (\texttt{Continue}\ \texttt{x}_1)\end{array}\right)\ \texttt{xini} \approx \texttt{iter}\ \left(\begin{array}{l}\lambda \texttt{x}_0.\\ \quad \texttt{x}_1 \leftarrow \texttt{f}\ \texttt{x}_0;\\ \quad \texttt{x}_2 \leftarrow \texttt{f}\ \texttt{x}_1;\\ \quad \texttt{ret}\ (\texttt{Continue}\ \texttt{x}_2)\end{array}\right)\ \texttt{xini} =: \texttt{loop}_2$$

This is the equivalence needed to justify loop unrolling of `while(true) { f }` into `while(true) { f; f }` for arbitrary instructions `f` in a language with `while` loops. In languages with CFGs of blocks, such as MiniBTL, similar properties are needed to perform block fusion transformations [105, §5.3].

This property is not provable using the approaches presented in section 4.3. Indeed, although the two semantics are defined using `iter`, their iterations are not synchronous. That is, $\texttt{f}\ \texttt{x}_0$ is not equivalent to $\texttt{x}_1 \leftarrow \texttt{f}\ \texttt{x}_0; \texttt{f}\ \texttt{x}_1$. Intuitively, the left-hand side performs two iterations when the right-hand side performs only one, but `eq_iter` requires a lock-step correspondence between the iterations.

We will present proof approaches to prove such loop transformations. Our method relies crucially on the codiagonal property of `iter` (`iter_codiagonal`). In order to facilitate its use, we derive a generalized version in section 4.4.1.1. We present the method for proving loop transformations in section 4.4.1.2. We define a framework to simplify proofs based on this method (section 4.4.1.3).

4.4.1.1 Generalizing the codiagonal property of `iter`

The axiom `iter_codiagonal` can be used to merge two nested `iter`s into a single one. As stated in figure 4.6, it requires the two `iter`s to operate on the same state. We can derive an equivalence for the case where the two `iter`s have different types of states:

Lemma 4.1 For any iteration body $\texttt{f} : \texttt{A} \to \texttt{B} \to \texttt{S}\ \texttt{E}\ (\texttt{B} + (\texttt{A} + \texttt{C}))$ and initial states $\texttt{ini}_0 : \texttt{A}$ and $\texttt{ini}_1 : \texttt{A} \to \texttt{B}$:

$$\texttt{iter}\ (\lambda \texttt{x}.\ \texttt{iter}\ (\texttt{f}\ \texttt{x})\ (\texttt{ini}_1\ \texttt{x}))\ \texttt{ini}_0 \approx \texttt{iter}\ \left(\lambda(\texttt{x},\texttt{y}).\ \begin{array}{l}\texttt{r} \leftarrow \texttt{f}\ \texttt{x}\ \texttt{y};\\ \texttt{ret}\ (\texttt{hcodiag}\ \texttt{x}\ \texttt{r})\end{array}\right)\ (\texttt{ini}_0, \texttt{ini}_1\ \texttt{ini}_0)$$

where:

$$\texttt{hcodiag}\ (\texttt{x} : \texttt{A})\ (\texttt{r} : \texttt{B} + (\texttt{A} + \texttt{C})) : \texttt{A} \times \texttt{B} + \texttt{C} := \begin{cases} \texttt{Continue}\ (\texttt{x}, \texttt{y}) & \text{if } \texttt{r} = \texttt{inl}\ \texttt{y} \\ \texttt{Continue}\ (\texttt{x}', \texttt{ini}_1\ \texttt{x}') & \text{if } \texttt{r} = \texttt{inr}\ (\texttt{inl}\ \texttt{x}') \\ \texttt{Break}\ \texttt{z} & \text{if } \texttt{r} = \texttt{inr}\ (\texttt{inr}\ \texttt{z}) \end{cases}$$

Idea of the proof We change the states of the two `iter`s to $\texttt{A} \times \texttt{B}$ using `eq_iter`. We then merge the two `iter`s into a single one using `iter_codiagonal`. We simplify the body of the resulting `iter` thanks to the compatibility of `iter` with $\approx$.

In fact, we can even allow the type of state of the inner `iter` to depend on the state of the outer `iter`. In that case, the merged `iter` operates over dependent pairs $\{\texttt{x} : \texttt{A}\ \&\ \texttt{B}\ \texttt{x}\}$.

4.4.1.2 Rewriting nested combinators into `iter`

Our approach to proving loop transformations will be to:

1. Find a notion of "step" common to both semantics.
2. Use `iter_codiagonal` and its generalizations to rewrite each semantics into an `iter` which performs exactly one step at each iteration.
3. Use `eq_iter` to decompose the proof of equivalence into equivalences of semantics that perform a single step.

Proof of example 4.8

1. We choose to consider as a step the semantics containing a single execution of `f`.
2. The left-hand side is already written as an `iter` which executes `f` once per iteration, so we only need to transform the right-hand side. We first use `iter_unfold` to rewrite the `bind` into a finite `iter` which performs two iterations.

   $$\texttt{loop}_2 \approx \texttt{iter}\ (\lambda \texttt{x}_0.\ \texttt{iter}\ (\texttt{f}_1\ \texttt{x}_0)\ \varnothing)\ \texttt{xini}$$

   where the state of `iter` has type `option A` and:

   $$\texttt{f}_1\ \texttt{x}_0\ \texttt{y} := \begin{cases} \texttt{x}_1 \leftarrow \texttt{f}\ \texttt{x}_0; \texttt{ret}\ (\texttt{Continue}\ \lfloor \texttt{x}_1 \rfloor) & \text{if } \texttt{y} = \varnothing \\ \texttt{x}_2 \leftarrow \texttt{f}\ \texttt{x}_1; \texttt{ret}\ (\texttt{Break}\ (\texttt{Continue}\ \texttt{x}_2)) & \text{if } \texttt{y} = \lfloor \texttt{x}_1 \rfloor \end{cases}$$

We then apply the generalization of iter_codiagonal (lemma 4.1) to merge the nested iters into a single one:

$$\texttt{loop}_2 \approx \texttt{iter}\ \left(\lambda(\texttt{x}_0, \texttt{y}).\ \begin{array}{l}\texttt{r} \leftarrow \texttt{f}_1\ \texttt{x}_0\ \texttt{y};\\ \texttt{ret}\ (\texttt{hcodiag}\ \texttt{x}_0\ \texttt{r})\end{array}\right)\ (\texttt{xini}, \varnothing) =: \texttt{loop}_2'$$

3. At this point, we are left with a goal:

$$\texttt{loop}_1 \approx \texttt{loop}_2'$$

We represented graphically those iters as CFGs in figure 4.9. Both iters perform a single execution of f at each iteration. However, they do not have the same state: $\texttt{loop}_1$ iterates over a state of type A whereas $\texttt{loop}_2'$ iterates over a state of type $\texttt{A} \times \texttt{option A}$. The different kinds of states of $\texttt{loop}_2'$ correspond to different execution points in $\texttt{loop}_2$:

- A state $(\texttt{x}_0, \varnothing)$ corresponds to the beginning of an iteration of $\texttt{loop}_2$ from a state $\texttt{x}_0$.
- A state $(\texttt{x}_0, \lfloor \texttt{x}_1 \rfloor)$ corresponds to the point before the second execution of f during an iteration of $\texttt{loop}_2$ from a state $\texttt{x}_0$ where the first f returned $\texttt{x}_1$.

We thus relate the states of $\texttt{loop}_1$ and $\texttt{loop}_2'$ using an invariant Inv inductively defined by:

$$\frac{}{\texttt{Inv}\ \texttt{x}_0\ (\texttt{x}_0, \varnothing)} \qquad \frac{}{\texttt{Inv}\ \texttt{x}_1\ (\texttt{x}_0, \lfloor \texttt{x}_1 \rfloor)}$$

We apply eq_iter with this invariant and prove the equivalence between the iteration bodies by relational reasoning.

□

4.4.1.3 EqIter: a framework to prove equivalences with CFGs

Although the method presented in section 4.4.1.2 works on small examples, it is harder to apply it on larger semantics. Indeed, the iters obtained by applying the codiagonal property (e.g. $\texttt{loop}_2'$) operate over complex states corresponding to the original structure of the semantics. As detailed during the third step of the proof of example 4.8, one needs this meaning of the states to find the adequate invariant for eq_iter. We implemented a framework EqIter to facilitate the proof of loop transformations using iter_codiagonal. The main difference with the direct approach of section 4.4.1.2 is that we choose up-front a representation of the equivalent semantics with an iter performing the chosen notion of step at each iteration. One then proves the equivalence by transitivity, using this iter as intermediate semantics. In example 4.8, one would directly use the left-hand side of the equivalence ($\texttt{loop}_1$). In the general case, viewing iter as a CFG (figure 4.9) often helps define the intermediate iter. The goal of our framework is thus to prove an equivalence $\texttt{u}_0 \approx \texttt{iter f}\ \texttt{x}_0$ between a semantics $\texttt{u}_0$ defined compositionally and an iter. We propose a method which follows the structure of the semantics $\texttt{u}_0$. A difficulty is that sub-parts u of $\texttt{u}_0$ are not related to whole executions of iter f, but only to prefixes of such executions. The framework EqIter is thus centered around a predicate EqIter.p which describes prefixes of an iteration $\texttt{iter f}\ \texttt{x}_0$. This predicate[11] has signature:

TODO (rocq)

- version avec une fonction comme invariant

$$\texttt{EqIter.p}\ [\texttt{E A B}]\ (\texttt{f} : \texttt{A} \to \texttt{S E}\ (\texttt{A} + \texttt{B}))\ [\texttt{C}]\ (\texttt{inv} : \texttt{Rel C}\ (\texttt{A} + \texttt{B}))\ (\texttt{u} : \texttt{S E C})\ (\texttt{x} : \texttt{A}) : \texttt{Type}$$

Intuitively EqIter.p f inv u x is satisfied if the behaviors of u correspond to non-empty prefixes of the iteration of f starting from x. A strict prefix must end before an execution of f; the value r returned by u must be related by $\texttt{inv r}\ (\texttt{Continue}\ \texttt{x}_1)$ to the state $\texttt{x}_1$ of $\texttt{iter f x}$ when it is interrupted. For complete executions, the value r returned by u must be related by $\texttt{inv r}\ (\texttt{Break y})$ to the result y of iter f x. We define formally EqIter.p in definition 4.7 below. Users of the framework do not have to manipulate this definition. Indeed, we proved lemmas which enable proving EqIter.p following

[11] In Rocq we defined the predicate EqIter.p in Type instead of Prop to avoid using the axiom of choice.

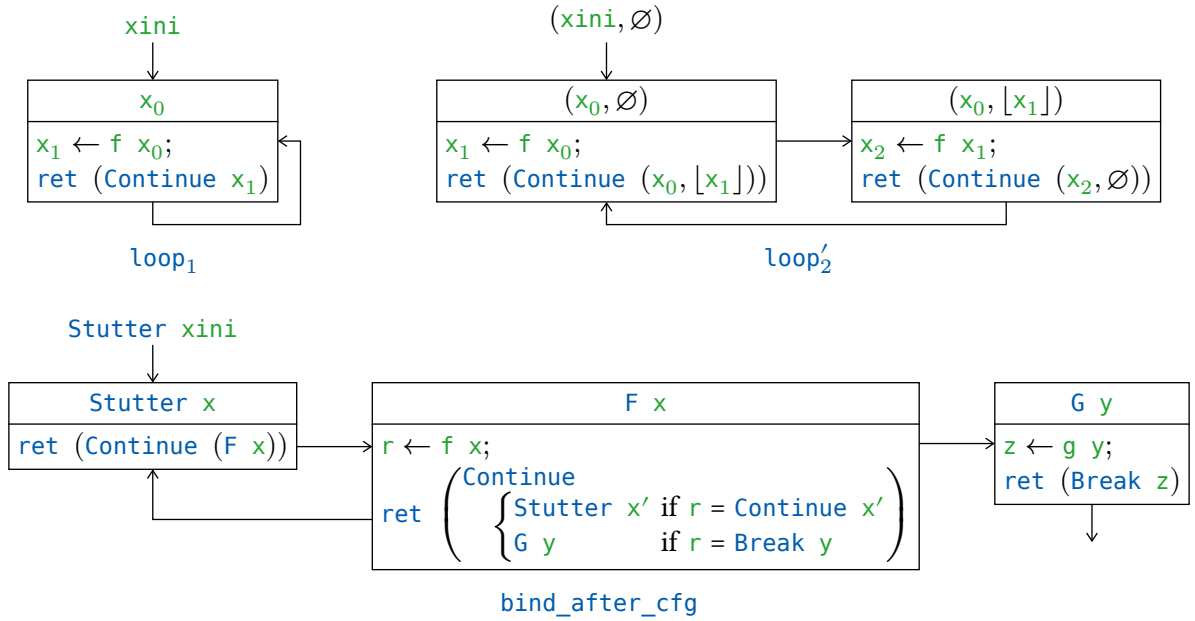


Figure 4.9: Representations of `iter`s as CFGs. The labels of the blocks are patterns, they define a partition of the states of the `iter` (this corresponds to a pattern matching in Rocq). The body of a block defines the semantics of the body on the states matched by its label. Those semantics depend on the variables appearing in the pattern.

$$\frac{\texttt{EqIter.p f } (\lambda \texttt{y r. r} = \texttt{Break y}) \texttt{ u}_0 \texttt{ x}_0}{\texttt{u}_0 \approx \texttt{iter f x}_0}\,\texttt{EqIter.init}$$

$$\frac{\texttt{u}_0 \approx \texttt{u}_1 \quad \texttt{EqIter.p f inv u}_1 \texttt{ x}}{\texttt{EqIter.p f inv u}_0 \texttt{ x}}\,\texttt{EqIter.morph_u} \qquad \frac{\texttt{u} \approx_{\texttt{inv}} \texttt{f x}}{\texttt{EqIter.p f inv u x}}\,\texttt{EqIter.one}$$

$$\frac{\begin{array}{c}\texttt{EqIter.p f } (\lambda \texttt{z r. r} = \texttt{Continue } (\texttt{inv z})) \texttt{ u x} \\ \forall \texttt{z. EqIter.p f inv}_0 \texttt{ (k z) (inv z)}\end{array}}{\texttt{EqIter.p f inv}_0 \texttt{ (bind u k) x}}\,\texttt{EqIter.bind}$$

$$\frac{\texttt{inv z}_0 = \texttt{x} \quad \forall \texttt{z. EqIter.p f inv}_1 \texttt{ (g z) (inv z)}}{\texttt{EqIter.p f inv}_0 \texttt{ (iter g z}_0\texttt{) x}}\,\texttt{EqIter.iter}$$

where: $\texttt{inv}_1 \texttt{ r}_0 \texttt{ r}_1 := \begin{cases} \texttt{r}_1 = \texttt{Continue (inv z)} & \text{if } \texttt{r}_0 = \texttt{Continue z} \\ \texttt{inv}_0 \texttt{ r}'_0 \texttt{ r}_1 & \text{if } \texttt{r}_0 = \texttt{Break r}'_0 \end{cases}$

Figure 4.10: Lemmas on `EqIter.p`

the structure of the semantics (figure 4.10). Those lemmas are derived from the axioms of `SMonad`, and in particular rely on `iter_codiagonal`. The lemma `EqIter.init` is used to start the proof of an equivalence using `EqIter.p`. The invariant used in its premise prevents the semantics $\texttt{u}_0$ from terminating earlier than `iter f` $\texttt{x}_0$. The lemmas `EqIter.bind` and `EqIter.iter` are then used to decompose the proof of `EqIter.p` following the structure of the semantics. Those lemmas require a function `inv` describing the invariant between the bound variables of the semantics (the value returned by `u` in `EqIter.bind` and the state of `iter g` in `EqIter.iter`) and the state of `iter f`. The semantics can be rewritten at any point during this decomposition, using `EqIter.morph_u`. Finally, `EqIter.one` is used to prove `EqIter.p` once the semantics have been decomposed into single steps.

We now describe on an example how the framework is used to prove equivalences between two semantics involving `iter`.

Example 4.9 For any semantics `f : A → S E (A + B)` and `g : B → S E C` and initial state `xini : A`:

$$\texttt{lhs} := \texttt{bind}\ (\texttt{iter}\ \texttt{f}\ \texttt{xini})\ \texttt{g}$$
$$\approx \texttt{iter}\ \left(\lambda \texttt{x}.\ \texttt{r} \leftarrow \texttt{f}\ \texttt{x};\begin{cases}\texttt{ret}\ (\texttt{Continue}\ \texttt{x}') & \text{if } \texttt{r} = \texttt{Continue}\ \texttt{x}' \\ \texttt{z} \leftarrow \texttt{g}\ \texttt{y};\texttt{ret}\ (\texttt{Break}\ \texttt{z}) & \text{if } \texttt{r} = \texttt{Break}\ \texttt{y}\end{cases}\right)\ \texttt{xini} =: \texttt{rhs}$$

That is, the continuation `g` after a loop can equivalently be executed during the last iteration. With a C-like syntax, this equivalence can be thought of as:

```
do {                          ↔      while(true){
  c = f;                               c = f;
} while(c);                            if (!c) { g; break; }
g;                                     }
```

with some statements `f` and `g` that do not use `break` or `continue`.

Proof We choose as notion of step single executions of `f` or `g`. We define an intermediate `iter`:

$$\texttt{bind_after_cfg} := \texttt{iter}\ \texttt{bind_after_f}\ (\texttt{Stutter}\ \texttt{xini})$$

For the sake of readability, we picture the definition of `bind_after_f` as a CFG in figure 4.9. This `iter` has three kinds of states: `F`, `G` and `Stutter`. Its body `bind_after_f` is defined by pattern matching over these three cases: in figure 4.9, each case is pictured as node of the CFG; edges between nodes represent the control-flow performed by `iter` between consecutive iterations. The first two kinds of states correspond to executions of `f` and `g`. We had to introduce a "stuttering" state `Stutter` in the definition of `bind_after_f`, to account for the isolated `ret` in `rhs` when `f` returns a value `Continue`. By transitivity, it is enough to prove that `lhs` ≈ `bind_after_cfg` and `rhs` ≈ `bind_after_cfg`.

Equivalence of `lhs` We apply `EqIter.init` to turn the goal into:

$$\texttt{EqIter.p}\ \texttt{bind_after_f}\ (\lambda \texttt{z}\ \texttt{r}.\ \texttt{r} = \texttt{Break}\ \texttt{z})\ \texttt{lhs}\ (\texttt{Stutter}\ \texttt{xini})$$

We then apply `EqIter.bind` with `inv := G` to decompose `lhs`. The goal for the right-hand side is:

$$\texttt{EqIter.p}\ \texttt{bind_after_f}\ (\lambda \texttt{z}\ \texttt{r}.\ \texttt{r} = \texttt{Break}\ \texttt{z})\ (\texttt{g}\ \texttt{y})\ (\texttt{G}\ \texttt{y})$$

Since the semantics `g y` performs exactly our chosen notion of step, we solve this goal using `EqIter.one`. For the left-hand side, we apply `EqIter.iter` with invariant `inv := Stutter` to decompose the `iter` of `lhs`. We are left with a goal:

$$\texttt{EqIter.p}\ \texttt{bind_after_f}\ \ldots\ (\texttt{f}\ \texttt{x})\ (\texttt{Stutter}\ \texttt{x})$$

Before executing `f`, we need to perform the stuttering step we introduced in `bind_after_cfg`. We add an artificial step by rewriting `f x` into `ret tt;; f x` (thanks to `EqIter.morph_u`). We then apply `EqIter.bind` to split this `bind` into two steps: `ret tt` and `f x`. Using `EqIter.one`, we relate those steps to the semantics of `Stutter` and `F`.

Equivalence of `rhs` We apply `EqIter.init`. By performing an artificial step, we change the state to `F xini`. We then apply `EqIter.iter` (with `inv := F`) and `EqIter.bind` to decompose `rhs`. We solve the goal for the left-hand side `f x` of `bind` using `EqIter.one`. In the goal for the right-hand side of `bind`, we have access to the value `r` returned by `f x`. We perform a case analysis on `r` and conclude in both cases with `EqIter.one`.

□

Example 4.9 is in fact the property "parameter" of the library surrounding Interaction Trees. Xia et al. [102, 4.1] gave a coinductive proof specific to Interaction Trees. Our proof is valid for any model of SMonad. Similarly, we derive using EqIter their "composition" property.

As illustrated in example 4.9, the definition of EqIter.p is not used when applying the framework. We now describe its formal definition, which is used in the proofs of lemmas of figure 4.10. We fix the first parameter $f : A \to S\ E\ (A + B)$ of EqIter.p (i.e. the body of the iter). We then define a notion of prefix of iter f:

Definition 4.6 We define a record type Prefix.t representing candidate prefixes returning values of type C:

$$\begin{array}{l}\texttt{Prefix.t E A C} := \{ \\ \quad \texttt{A}' : \texttt{Type} \\ \quad \texttt{f_A} : \texttt{A}' \to \texttt{A} \\ \quad \texttt{def} : \texttt{A}' \to \texttt{S E } (\texttt{A}' + \texttt{C}) \\ \quad \texttt{ini} : \texttt{A}' \\ \}\end{array}$$

Such candidate is described by an iteration of semantics def. We define the semantics of candidate prefixes as:

$$\texttt{Prefix.sem [E A C] (pf : Prefix.t E A C) : S E C} := \texttt{iter pf.def pf.ini}$$

The states $\texttt{A}'$ of a candidate prefix are mapped into states A of iter f by the function f_A. Given a candidate pf, a post-condition $\texttt{P} : \texttt{Rel C } (\texttt{A} + \texttt{B})$ for its results and an initial state $\texttt{x} : \texttt{A}$ for iter f, we say that pf is a *prefix* if f_A defines a lock-step equivalence:

$$\begin{array}{l}\texttt{is_prefix [A B C] (P : Rel C } (\texttt{A} + \texttt{B})) \texttt{ (f : A} \to \texttt{S E } (\texttt{A} + \texttt{B})) \texttt{ (x : A) (pf : Prefix.t A C)} := \\ \quad \texttt{pf.f_A pf.ini} = \texttt{x} \wedge \\ \quad \forall \texttt{a}' : \texttt{pf.A}'.\ \texttt{pf.def a}' \approx_{\texttt{R}} \texttt{f (pf.f_A a}')\end{array}$$

where:

$$\texttt{R } r_0\ r_1 := \begin{cases} r_1 = \texttt{Continue (pf.f_A a}') & \text{if } r_0 = \texttt{Continue a}' \\ \texttt{P c } r_1 & \text{if } r_0 = \texttt{Break c} \end{cases}$$

We then define EqIter.p so that it holds if the semantics given as argument is equivalent to a prefix of iter f:

Definition 4.7

$$\begin{array}{l}\texttt{EqIter.p [E A B] (f : A} \to \texttt{S E } (\texttt{A} + \texttt{B})) \texttt{ [C] (inv : Rel C } (\texttt{A} + \texttt{B})) \texttt{ (u : S E C) (x : A) : Type} := \{ \\ \quad \texttt{pf : Prefix.t S E A C} \\ \quad \texttt{PF : is_prefix inv f x pf} \\ \quad \texttt{EQV : u} \approx \texttt{Prefix.sem pf} \\ \}\end{array}$$

4.4.2 Function inlining

Like for iter, we assume that the combinator mrec satisfies a codiagonal property (mrec_codiagonal). We use this property to prove transformations which inline function calls. The codiagonal property justifies the changes in the control flow of programs. On some languages, inlining transformations would also require additional reasoning on data related to functions (e.g. the fusion of stack-frames). To facilitate proofs based on mrec_codiagonal, we implemented a framework called EqMRec. This framework is similar to EqIter, but is used to prove equivalences with mrec instead of equivalences with iter. In particular, it is derived from the axioms of SMonad.

The goal of the framework EqMRec is to prove equivalences with shape $u_0 \approx \texttt{mrec f } d_0$. Like in EqIter, we follow the structure of the semantics u_0. The framework EqMRec is thus centered around a predicate

`EqMRec.p` which relates components of u_0 to subparts of the call tree of `mrec f`. This predicate has signature:

```
EqMRec.p [D E] (f : handler S D (D + E))
  [F] (inv : ∀[B]. F B → EventFam.sum D E B)
  [A] (u : S F A) (v : S (D + E) A) :
  Type
```

During proofs, the first semantics `u` is a component of u_0 and the second semantics `v` is a component of the implementation of a function in `mrec f` (that is, a component of `f d` for some call `d`). The invariant `inv` relates the events emitted by `u` to either internal calls `D` of `mrec f` or events `E` emitted by `mrec f`. Intuitively, the predicate `EqMRec.p` is satisfied if the semantics `u` is a prefix of the call tree obtained by recursively interpreting the events of `v` using `f`. Like for `EqIter.p`, we derive lemmas around `EqMRec.p` so that users of the framework do not need to use its formal definition. The definition of `EqMRec.p` can be found in the Rocq development [SMonad.Generic.EqMRec:97]. Figure 4.11 reproduces the lemmas on `EqMRec.p`. The lemma `EqMRec.init` starts the proof of an equivalence using `EqMRec`. In the premise, the invariant `inr` expresses that all events emitted by u_0 are external events of `mrec f`. Proofs then follow the structure of the semantics u_0. Lemmas on `EqMRec.p` are used to decompose the combinators used to build u_0. Applying `EqMRec.mrec` extends the invariant on events with a chosen invariant `inv` on the function call of the decomposed `mrec`. The lemmas `EqMRec.bind` and `EqMRec.iter` are applied when the two semantics are defined with corresponding combinators `bind` or `iter`. If the decomposition reduces the second semantics to a call, the body of the callee can be inlined using `EqMRec.trigger`. The lemma `EqMRec.morph` allows rewriting equivalences in the two semantics at any point. We stop the decomposition with `EqMRec.one` once the first semantics is reduced to a subpart of a single node of the call tree of `mrec f`.

Example 4.10 We prove an equivalence between two programs computing the number of steps of the Syracuse sequence. For any $n_0 : \mathbb{N}$:

$$\texttt{lhs} := \texttt{mrec}\ \left(\lambda(\texttt{f n}).\ \begin{cases} \texttt{ret}\ 0 & \text{if } \texttt{n} = 1 \\ \texttt{c} \leftarrow \texttt{trigger}\ \left(\texttt{inl}\ \left(\texttt{f}\ \frac{\texttt{n}}{2}\right)\right); \texttt{ret}\ (\texttt{c}+1) & \text{if n is even} \\ \texttt{c} \leftarrow \texttt{trigger}\ (\texttt{inl}\ (\texttt{f}\ (3\texttt{n}+1))); \texttt{ret}\ (\texttt{c}+1) & \text{otherwise} \end{cases}\right)\ (\texttt{f}\ n_0)$$

$$\approx$$

$$\texttt{rhs} := \texttt{mrec}\ \left(\lambda(\texttt{f n}).\ \begin{cases} \texttt{ret}\ 0 & \text{if } \texttt{n} = 1 \\ \texttt{c} \leftarrow \texttt{trigger}\ \left(\texttt{inl}\ \left(\texttt{f}\ \frac{\texttt{n}}{2}\right)\right); \texttt{ret}\ (\texttt{c}+1) & \text{if n is even} \\ \texttt{c} \leftarrow \texttt{trigger}\ \left(\texttt{inl}\ \left(\texttt{f}\ \frac{3\texttt{n}+1}{2}\right)\right); \texttt{ret}\ (\texttt{c}+2) & \text{otherwise} \end{cases}\right)\ (\texttt{f}\ n_0)$$

where the internal calls of both `mrec`s are represented with a family of events `F` generated by `f (n : ℕ) : F ℕ`.

Proof The system `rhs` is obtained by inlining the recursive call when `n` is odd. We use `EqMRec` with `lhs` as target `mrec`. That is, we use the symmetry of $\approx$, then apply `EqMRec.init` to obtain a goal:

```
EqMRec.p lhs_body inr (mrec rhs_body (f n₀)) (lhs_body (f n₀))
```

where `lhs_body` (resp. `rhs_body`) is the body of the `mrec` defining `lhs` (resp. `rhs`). We apply `EqMRec.mrec` with the identity as invariant. We have to prove that:

```
∀n. EqMRec.p lhs_body id (rhs_body (f n)) (lhs_body (f n))
```

Following the common structure of the bodies, we perform a case analysis on `n`. When `n` is 1 or even, we conclude using `EqMRec.one`. In the last case, we first use `EqMRec.morph` to rewrite the goal into:

$$\frac{\texttt{EqMRec.p f inr } u_0\ (\texttt{f } d_0)}{u_0 \approx \texttt{mrec f } d_0}\ \texttt{EqMRec.init}$$

$$\frac{u_0 \approx u_1 \quad v_0 \approx v_1 \quad \texttt{EqMRec.p f inv } u_1\ v_1}{\texttt{EqMRec.p f inv } u_0\ v_0}\ \texttt{EqMRec.morph}$$

$$\frac{\texttt{u} \approx_{(\texttt{of_fun inv}),\texttt{id}} \texttt{v}}{\texttt{EqMRec.p f inv u v}}\ \texttt{EqMRec.one} \qquad \frac{\texttt{EqMRec.p f inv u } (\texttt{f d})}{\texttt{EqMRec.p f inv u } (\texttt{trigger } (\texttt{inl d}))}\ \texttt{EqMRec.trigger}$$

$$\frac{\texttt{f } (\texttt{inv ini}) \approx \texttt{v} \quad \forall \texttt{d}.\ \texttt{EqMRec.p f } \texttt{inv}_1\ (\texttt{g d})\ (\texttt{f } (\texttt{inv d}))}{\texttt{EqMRec.p f } \texttt{inv}_0\ (\texttt{mrec g ini})\ \texttt{v}}\ \texttt{EqMRec.mrec}$$

where: $\texttt{inv}_1\ \texttt{e} := \begin{cases} \texttt{inl } (\texttt{inv } e_0) & \text{if } \texttt{e} = \texttt{inl } e_0 \\ \texttt{inv}_0\ e_1 & \text{if } \texttt{e} = \texttt{inr } e_1 \end{cases}$

$$\frac{\texttt{EqMRec.p f inv } u_0\ u_1 \quad \forall \texttt{x}.\ \texttt{EqMRec.p f inv } (k_0\ \texttt{x})\ (k_1\ \texttt{x})}{\texttt{EqMRec.p f inv } (\texttt{bind } u_0\ k_0)\ (\texttt{bind } u_1\ k_1)}\ \texttt{EqMRec.bind}$$

$$\frac{\forall \texttt{x}.\ \texttt{EqMRec.p f inv } (g_0\ \texttt{x})\ (g_1\ \texttt{x})}{\texttt{EqMRec.p f inv } (\texttt{iter } g_0\ \texttt{ini})\ (\texttt{iter } g_1\ \texttt{ini})}\ \texttt{EqMRec.iter}$$

Figure 4.11: Lemmas on EqMRec.p

$$\texttt{EqMRec.p lhs_body id}$$
$$\left(\texttt{c}_1 \leftarrow \left(\texttt{c}_0 \leftarrow \texttt{trigger } \left(\texttt{inl } \left(\texttt{f } \tfrac{3\texttt{n}+1}{2}\right)\right); \texttt{ret } (\texttt{c}_0 + 1)\right); \texttt{ret } (\texttt{c}_1 + 1)\right)$$
$$(\texttt{c} \leftarrow \texttt{trigger } (\texttt{inl } (\texttt{f } (3\texttt{n}+1))); \texttt{ret } (\texttt{c}+1))$$

We then apply EqMRec.bind, and solve the obligation for the right-hand sides (the rets) using EqMRec.one. For the left-hand sides, we apply EqMRec.trigger to inline the call in lhs. The semantics trigger (inl (f (3n + 1))) is thus replaced by lhs_body (f (3n + 1)). We simplify this expression using the fact that n is odd. Finally, we conclude using EqMRec.one. □

4.4.3 Inconsistent properties

We have shown that several classes of transformations can be proven with the axiomatization of the SMonad library. However, not all desirable refinements are derivable from the axiomatization. Even worse, some desirable refinements are incompatible with the axiomatization. An example of such transformation incompatible with the axiomatization is the commutation of non-deterministic choices below events:

Definition 4.8 In a given domain S, *non-determinism commutes with events* if, for any event family E, event e in E and non-empty type A:

```
x ← any A;                y ← trigger e;
y ← trigger e;   ≳        x ← any A;
ret (x, y)                ret (x, y)
```

The reverse refinement is derivable using the axioms of any, for an arbitrary type A. For the direction of definition 4.8, if A is empty, the source is equivalent to any False (i.e. a semantics always correct, as discussed in section 4.2.4.1) whereas the target still emits an event. However, with the restriction

not_ub_insert:	`ret tt` $\not\gtrsim_\top$ `ub`			
not_any_emp_remove:	`any` $\varnothing \not\gtrsim_\top$ `ret tt`			
not_trigger_insert:	`ret tt` $\not\gtrsim_\top$ `trigger e`			
not_trigger_remove:	`trigger e` $\not\gtrsim_\top$ `ret tt`			
not_trigger_comm:	`trigger` e_0;; `trigger` e_1	$\not\gtrsim_\top$	`trigger` e_1;; `trigger` e_0	where $e_0 \neq e_1$
not_any_trigger_comm:	$b_0 \leftarrow$ `any bool`; $b_1 \leftarrow$ `trigger e`; `ret` (b_0, b_1)	$\not\gtrsim$	$b_1 \leftarrow$ `trigger e`; $b_0 \leftarrow$ `any bool`; `ret` (b_0, b_1)	where `e : E bool`

Figure 4.12: Some refinements which are inconsistent with the axiomatization of `SMonad`

that `A` is non-empty, it is reasonable to ask for this commutation property, and it could be useful to justify some optimizations (we will give an example in Frag IR, with figure 8.10). Indeed, from the point of view of the context, both semantics emit the event `e` then return its answer and a value of type `A` chosen independently. Formally the two semantics are equivalent for the notion of inclusion of behaviors used as top-level specification in verified compilation (we discuss this notion in section 5.4). However, the refinement of definition 4.8 is not derivable from the axiomatization of `SMonad`. That is, there exist domains which satisfy the axiomatization but where non-determinism does not commute with events. We will describe such domain in chapter 5. But in fact, we now show that any domain satisfying both the axiomatization and the refinement of definition 4.8 is degenerate (i.e. not consistent according to definition 4.9 below). We thus manipulate `SMonad` refinement relations which are correct but not complete with respect to the inclusion of behaviors. Proving transformations which perform commutations corresponding to definition 4.8 would require weakening the axiomatization of `SMonad` or using other proof approaches (e.g. directly reasoning on the sets of behaviors). In this thesis, we stick to the axiomatization of `SMonad` described in section 4.2.

The conclusions of all axioms of `SMonad` are refinements (figure 4.6). The axiomatization is thus satisfied by any domain with a degenerate refinement, that is such that `u` $\gtrsim$ `v` holds for any semantics `u` and `v`. Such domains are not useful for verifying transformations, so we exclude them in the remainder of this section.

Definition 4.9 A domain `S` is *consistent* if one of the following equivalent conditions hold:

- A proof of refinement between two computations returning immediately implies that the post-condition hold between their results:

$$\forall R\ x\ y.\ \texttt{ret}\ x \gtrsim_R \texttt{ret}\ y \Rightarrow R\ x\ y$$

- Two no-ops cannot be related with the empty post-condition:

$$\texttt{ret tt} \not\gtrsim_\bot \texttt{ret tt}$$

- `sem_ref` is not always true. That is, there exists some semantics `u`, `v` and relations `Re` and `R` such that and `u` $\not\gtrsim_{Re,R}$ `v`

We have proved that some refinements do not hold in any consistent domain which satisfies the axioms of `SMonad`. Figure 4.12 lists some of them. The property `not_ub_insert` reflects the fact that UBs cannot be inserted in the target during a refinement. This is expected, since correct programs should never execute UBs. Dually, `any` $\varnothing$ cannot be removed (`not_any_emp_remove`). The properties `not_trigger_insert`, `not_trigger_remove` and `not_trigger_comm` reflect a one-to-one, order preserving, correspondence between source and target events. Such correspondence is what we had in mind when defining relations on events. However, those properties rule out domains where arbitrary computations could be inserted, removed or commuted. There are domains without

events (such as the option and delay monads [24]) in which arbitrary computations commute. The lemma `not_trigger_comm` shows that those domains cannot be generalized to support events with the axiomatization of `SMonad` while preserving arbitrary commutations. Finally, the lemma `not_any_trigger_comm` shows that the commutation of non-determinism with events is incompatible with the axiomatization of `SMonad`.

We now describe the proofs of `not_trigger_remove` and `not_any_trigger_comm`. Those proofs proceed by contradiction and rely crucially on `sem_ref_trans`.

Lemma 4.2 For any consistent domain `S` satisfying the axioms of `SMonad`, event family `E` and event `e` in `E`:

$$\texttt{trigger e} \not\gtrsim_{\texttt{id},\top} \texttt{ret tt}$$

Proof We assume that such refinement holds. We consider an event family $\texttt{E}_0$ with a single event $\texttt{e}_0 : \texttt{E}_0\ \texttt{unit}$ and the relation on events $\texttt{Re} : \texttt{EventFam.Rel.t}\ \texttt{E}_0\ \texttt{E}$ defined by:

$$\frac{}{\texttt{Re}\ \texttt{e}_0\ \texttt{e}\ \bot}$$

The following refinements hold:

$$\texttt{trigger e}_0 \gtrsim_{\texttt{Re},\bot} \texttt{trigger e} \gtrsim_{\texttt{id},\top} \texttt{ret tt} \gtrsim_{\bot,\top} \texttt{ret tt}$$

From left to right, those refinements are justified by `ref_trigger`, the assumption and `ref_ret`. By transitivity (`sem_ref_trans`), we derive:

$$\texttt{trigger e}_0 \gtrsim_{(\texttt{Re};\texttt{id};\bot),(\bot;\top;\top)} \texttt{ret tt}$$

By monotonicity, we obtain:

$$\frac{\texttt{Re};\texttt{id};\bot \leq \bot \qquad \bot;\top;\top \leq \bot \qquad \texttt{trigger e}_0 \gtrsim_{(\texttt{Re};\texttt{id};\bot),(\bot;\top;\top)} \texttt{ret tt}}{\texttt{trigger e}_0 \gtrsim_{\bot,\bot} \texttt{ret tt}}\ \texttt{sem_ref_mono}$$

The first premise is an inequality between relations on events. It is the main ingredient of the proof of the lemma. Indeed, the left-hand side contains `Re`, which relates events with a strong post-condition ($\bot$). Such relation on events is a strong requirement on the contexts: they cannot give any answers to the semantics. As a consequence, we were able to prove the refinement $\texttt{trigger e}_0 \gtrsim_{\texttt{Re},\bot} \texttt{trigger e}$ with an arbitrary post-condition as soon as events were emitted. On the contrary, the relation on events $\bot$ on the right-hand side does not relate any events. It is thus a weak requirement on the contexts: semantics are not allowed to emit events so the contexts can handle them arbitrarily. We can now derive a contradiction by placing the semantics in a context using `interp`. Formally, we instantiate `ref_interp` as:

$$\frac{\forall \texttt{d}_0\ \texttt{d}_1\ \texttt{P}.\ \bot\ \texttt{d}_0\ \texttt{d}_1\ \texttt{P} \Rightarrow \texttt{ret tt} \gtrsim_{\texttt{P}} \texttt{ret tt} \qquad \texttt{trigger e}_0 \gtrsim_{\bot,\bot} \texttt{ret tt}}{\texttt{interp}\ (\lambda_.\ \texttt{ret tt})\ (\texttt{trigger e}_0) \gtrsim_{\bot} \texttt{interp}\ (\lambda_.\ \texttt{ret tt})\ (\texttt{ret tt})}\ \texttt{ref_interp}$$

The first premise is satisfied since $\bot$ does not relate any pair of events. Using the equational theory of `interp`, we simplify both sides of the resulting refinement and obtain:

$$\texttt{ret tt} \gtrsim_{\bot} \texttt{ret tt}$$

which contradicts the assumption that `S` is consistent. □

Lemma 4.3 For any consistent domain `S` satisfying the axioms of `SMonad`, event family `E` and event `e` in `E` with `bool` as answer type:

$$\texttt{lhs} := \begin{pmatrix} \texttt{b}_0 \leftarrow \texttt{any bool}; \\ \texttt{b}_1 \leftarrow \texttt{trigger e}; \\ \texttt{ret}\ (\texttt{b}_0, \texttt{b}_1) \end{pmatrix} \not\gtrsim_{\texttt{id},\texttt{id}} \begin{pmatrix} \texttt{b}_1 \leftarrow \texttt{trigger e}; \\ \texttt{b}_0 \leftarrow \texttt{any bool}; \\ \texttt{ret}\ (\texttt{b}_0, \texttt{b}_1) \end{pmatrix} =: \texttt{rhs}$$

Idea of the proof

- On the left-hand side, we replace $\mathtt{e}$ by an event which takes as argument the Boolean $\mathtt{b}_0$ chosen by $\mathtt{any}$. Thanks to this additional dependency, we interpret the event so that $\mathtt{b}_1 = \mathtt{b}_0$.
- On the right-hand side, $\mathtt{any}$ may depend on the answer $\mathtt{b}_1$ to the event. We refine $\mathtt{any}$ to obtain a post-condition that $\mathtt{b}_0 \neq \mathtt{b}_1$. This refinement uses a relation on events with a weak post-condition on answers to accommodate the fact that the event handler for the left-hand side will depend on $\mathtt{b}_0$, whereas the handler for the right-hand side does not.

After interpretation, we derive a contradiction from the mismatch between the semantics of the left-hand side and the post-condition.

Proof We consider the event families $\mathtt{E}_0$ (the same as in lemma 4.2) and $\mathtt{E}_1$ containing the events:

$$\mathtt{e}_0 : \mathtt{E}_0\ \mathtt{unit} \qquad \mathtt{e}_1\ (\mathtt{b} : \mathtt{bool}) : \mathtt{E}_1\ \mathtt{bool}$$

We define the relations $\mathtt{ReT}$ between two arbitrary families of events and $\mathtt{Re} : \mathtt{EventFam.Rel.t}\ \mathtt{E}_1\ \mathtt{E}$ by:

$$\frac{}{\mathtt{ReT}\ \mathtt{d}_0\ \mathtt{d}_1\ \top} \qquad \frac{}{\mathtt{Re}\ (\mathtt{e}_1\ \mathtt{b})\ \mathtt{e}\ \mathtt{id}}$$

The relation $\mathtt{ReT}$ thus relates any pair of events but does not enforce any post-condition on their answers. Similarly, $\mathtt{Re}$ relates $\mathtt{e}$ to $\mathtt{e}_1$ with any argument $\mathtt{b}$, but its post-condition enforces the equality of their Boolean answers. We define a relation $\mathtt{R}_{\neq} : \mathtt{Rel}\ (\mathtt{bool} \times \mathtt{bool})\ \mathtt{unit}$ by:

$$\mathtt{R}_{\neq}\ (\mathtt{b}_0, \mathtt{b}_1)\ \mathtt{tt} := (\mathtt{b}_0 \neq \mathtt{b}_1)$$

We have the following chain of refinements:

$$\mathtt{lhs}' := \begin{pmatrix} \mathtt{b}_0 \leftarrow \mathtt{any}\ \mathtt{bool}; \\ \mathtt{b}_1 \leftarrow \mathtt{trigger}\ (\mathtt{e}_1\ \mathtt{b}_0); \\ \mathtt{ret}\ (\mathtt{b}_0, \mathtt{b}_1) \end{pmatrix} \gtrsim_{\mathtt{Re},\mathtt{id}} \mathtt{lhs} \gtrsim_{\mathtt{id},\mathtt{id}} \mathtt{rhs} \gtrsim_{\mathtt{ReT},\mathtt{R}_{\neq}} \mathtt{trigger}\ \mathtt{e}_0 =: \mathtt{rhs}'$$

The proof of the refinement of the left-hand side uses $\mathtt{ref_trigger}$ to replace $\mathtt{e}$ with $\mathtt{e}_1\ \mathtt{b}_0$. In the right-hand side, we replace $\mathtt{e}$ with $\mathtt{e}_0$ using $\mathtt{ref_trigger}$. We then apply $\mathtt{any0_src_ref}$ and choose $\mathtt{b}_1 := \neg \mathtt{b}_0$ to satisfy the post-condition $\mathtt{b}_0 \neq \mathtt{b}_1$.

By transitivity and monotonicity, we derive:

$$\mathtt{lhs}' \gtrsim_{\mathtt{ReT},\mathtt{R}_{\neq}} \mathtt{rhs}'$$

We interpret the events using $\mathtt{ref_interp}$:

$$\frac{\forall \mathtt{b}.\ \mathtt{ret}\ \mathtt{b} \gtrsim_{\top} \mathtt{ret}\ \mathtt{tt}}{\dfrac{\forall \mathtt{b}\ \mathtt{P}.\ \mathtt{ReT}\ (\mathtt{e}_1\ \mathtt{b})\ \mathtt{e}_0 \Rightarrow \mathtt{ret}\ \mathtt{b} \gtrsim_{\mathtt{P}} \mathtt{ret}\ \mathtt{tt} \qquad \mathtt{lhs}' \gtrsim_{\mathtt{ReT},\mathtt{R}_{\neq}} \mathtt{rhs}'}{\mathtt{interp}\ (\lambda(\mathtt{e}_1\ \mathtt{b}).\ \mathtt{ret}\ \mathtt{b})\ \mathtt{lhs}' \gtrsim_{\mathtt{R}_{\neq}} \mathtt{interp}\ (\lambda_.\ \mathtt{ret}\ \mathtt{tt})\ \mathtt{rhs}'}\ \mathtt{ref_interp}}$$

The first premise is a refinement between the events handlers. This refinement holds because the post-condition $\top$ on answers given by $\mathtt{ReT}$ does not require the handlers to return the same value. Using the equational theory of $\mathtt{interp}$, we simplify the resulting refinement into:

$$\mathtt{b}_0 \leftarrow \mathtt{any}\ \mathtt{bool}; \mathtt{ret}\ (\mathtt{b}_0, \mathtt{b}_0) \gtrsim_{\mathtt{R}_{\neq}} \mathtt{ret}\ \mathtt{tt}$$

Using $\mathtt{any0_trg_ref}$, we also prove that:

$$\mathtt{ret}\ \mathtt{tt} \gtrsim_{\lambda_(\mathtt{b}_0,\mathtt{b}_1).(\mathtt{b}_0=\mathtt{b}_1)} \mathtt{b}_0 \leftarrow \mathtt{any}\ \mathtt{bool}; \mathtt{ret}\ (\mathtt{b}_0, \mathtt{b}_0)$$

By transitivity and monotonicity, we obtain:

$$\mathtt{ret}\ \mathtt{tt} \gtrsim_{\bot} \mathtt{ret}\ \mathtt{tt}$$

which contradicts the assumption that $\mathtt{S}$ is consistent. □

4.5 Transformers

Defining the semantics of a language requires a domain which supports all its effects. Monad transformers [4; 70; 103] enable modular definitions of monads supporting multiple effects. We use a similar notion of *transformers* to derive models of `SMonad`. A transformer encodes an effect within an existing

domain. That is, assume that we already have a domain S, but which only supports a subset of the effects needed; applying a transformer T to S defines a new domain T S which:
- Still satisfies the axioms of SMonad. In particular, it implements all core operators, including iter and mrec.
- Still has the effects of S we are interested in.
- Additionally supports other effects.
- Optionally, provides a lifting operation, which maps inhabitants of S into inhabitants of T S.

We then use the domain T S to define semantics, or apply other transformers to add more effects. In this section, we prefix the names of the operators of SMonad on T S (e.g. T.ret), to distinguish them from the operaotrs on S (left unprefixed).

Example 4.11 We define a domain BTL_S supporting the effects used by MiniBTL, thanks to transformers that will be detailed in the next sections, and to a LTS-based domain described in chapter 5.

```
BTL_S : sem_s := State.S Mem.t (ExternalEvents.S IO LTS.t)
```

That is:
- We start from a LTS-based domain. This domain supports the effects required by SMonad (divergence and events). It also natively features non-determinism and UBs.
- We apply a transformer ExternalEvents to add input/output external events.
- We then apply a transformer State to add a global memory state.

Both of those transformers preserve the effects that were already available in the domain they are applied to. We thus obtain a domain which supports all effects required on the parameter S of figure 4.4.

Our use of transformers is motivated by two reasons:
- It factorizes the definition of domains. Moreover, it simplifies the definition of the base domain (in our case LTS.t), which only needs to support a minimal set of effects.
- It facilitates the interaction between semantics with different effects. Consider a domain S and a transformer T. A semantics u defined in S can be compared to a semantics v defined in T S either by:
  - Unfolding the definition of T to see v as an element of S, and then using the refinement relation on S.
  - Lifting u into an element of T S, and then using the refinement relation on T S.

  The first approach can more generally compare two domains T_0 S and T_1 S defined by applying different transformers T_0 and T_1 to a common base domain S. However, it requires unfolding the encoding used by T. On the other hand, the second approach requires the lifting operation to be defined for all elements of S we are comparing. We only used the first approach, for the translations described in the sections 9.3 and 10.3.

We did not attempt to formalize a general interface for transformers. The following sections describe the transformers we used.

4.5.1 Adding external events

The transformer ExternalEvents is parameterized by a family EXT of events. It adds them as external events, which can be emitted by any semantics of the domain ExternalEvents.S EXT S, independently of their signatures. The encoding relies on the events of the underlying domain:

```
ExternalEvents.S EXT S E A := S (E + EXT) A
```

The operator external emits an external event using the trigger of the underlying domain:

```
external [E A] (e : EXT A) : ExternalEvents.S EXT E A := trigger (inr e)
```

Since external events are hidden from the signature of semantics, this operator is defined for an arbitrary family E.

Most operators are directly defined using their instances on `S`. In `interp` we use `trigger` to directly re-emit the external events.

$$\texttt{ExternalEvents.interp}\ [\texttt{D}\ \texttt{E}]\ (\texttt{h} : \texttt{handler}\ \texttt{D}\ \texttt{E})\ [\texttt{A}]\ (\texttt{u} : \texttt{ExternalEvents.S}\ \texttt{EXT}\ \texttt{D}\ \texttt{A}) :=$$
$$\texttt{interp}\ \left(\lambda(\texttt{e} : (\texttt{D} + \texttt{E})\ _).\ \begin{cases}\texttt{h}\ \texttt{d} & \text{if } \texttt{e} = \texttt{inl}\ \texttt{d} \\ \texttt{trigger}\ (\texttt{inr}\ \texttt{ext}) & \text{if } \texttt{e} = \texttt{inr}\ \texttt{ext}\end{cases}\right)\ \texttt{u}$$

The refinement relation on `ExternalEvents.S` is also derived from its instance on `S`, using the identity relation on external events:

$$\texttt{ExternalEvents.sem_ref}\ \texttt{Re}\ \texttt{R}\ \texttt{u}_0\ \texttt{u}_1 := \texttt{sem_ref}\ (\texttt{Re} + \texttt{id})\ \texttt{R}\ \texttt{u}_0\ \texttt{u}_1$$

Hence, external events are preserved by a refinement in `ExternalEvents.S`.

4.5.2 Adding a state

The transformer `State` is parameterized by a type `ST`, modeling a global, readable and writable state. The encoding is standard: a semantics is a function which takes as argument the initial state and returns the state at the end of its execution.

$$\texttt{State.S}\ \texttt{ST}\ \texttt{E}\ \texttt{A} := \texttt{ST} \to \texttt{S}\ (\texttt{State.ev}\ \texttt{ST}\ \texttt{E})\ (\texttt{A} \times \texttt{ST})$$

The state is also added to the events, thanks to the family of events `State.ev ST E` generated by a constructor `State.mk_ev [B] (e : E B) (s : ST) : State.ev ST E (B × ST)`. That is, an event emitted by the semantics contains the value `s` of the state at the point where the event is emitted. The second component of the answer of the context (of type `B × ST`) is the value of the state at the end of the event. The context can thus modify the state when the semantics emits an event.

We implement all core operators of `SMonad` for `State.S ST S` from their instances on `S`. The state is propagated according to the order of execution. We implement `iter` by storing the value of the global state in the state of the iteration:

$$\texttt{State.iter}\ [\texttt{E}\ \texttt{A}\ \texttt{B}]\ (\texttt{f} : \texttt{A} \to \texttt{State.S}\ \texttt{ST}\ \texttt{S}\ (\texttt{A} + \texttt{B}))\ (\texttt{x}_0 : \texttt{A}) : \texttt{State.S}\ \texttt{ST}\ \texttt{S}\ \texttt{B} :=$$
$$\lambda \texttt{s}_0.\ \texttt{iter}\ \left(\lambda(\texttt{x}, \texttt{s}).\ (\texttt{r}, \texttt{s}') \leftarrow \texttt{f}\ \texttt{x}\ \texttt{s};\ \texttt{ret}\ \begin{cases}\texttt{Continue}\ (\texttt{x}', \texttt{s}') & \text{if } \texttt{r} = \texttt{Continue}\ \texttt{x}' \\ \texttt{Break}\ (\texttt{y}, \texttt{s}') & \text{if } \texttt{r} = \texttt{Break}\ \texttt{y}\end{cases}\right)\ (\texttt{x}_0, \texttt{s}_0)$$

Similarly, we apply `State.ev` to the family of internal events of `mrec` to propagate the value of the state across function calls. We also implement the operators providing non-determinism and UBs, if those effects are available in `S`. There is a lifting operation from `S` to `State.S`, but only for semantics which do not emit events:

$$\texttt{State.lift}\ [\texttt{A}]\ (\texttt{u} : \texttt{S}\ \varnothing\ \texttt{A}) : \texttt{State.S}\ \texttt{ST}\ \texttt{S}\ \varnothing\ \texttt{A} :=$$
$$\lambda \texttt{s}.\ \texttt{bind}\ (\texttt{lift_ev_emp}\ \texttt{u})\ (\lambda \texttt{x}.\ \texttt{ret}\ (\texttt{x}, \texttt{s}))$$

This lifting is defined by returning the initial state unchanged. We cannot generalize it to semantics which emit events. Indeed, the context would be able to modify the state during those events, but we have no way of propagating the value of the state between events.

From the refinement on `S`, we derive a refinement relation on `State.S ST S` with the signature expected by `SMonad`. This refinement enforces the equality of the source and target states at corresponding points of the interfaces between the semantics and their contexts:

$$\texttt{State.sem_ref}\ \texttt{Re}\ \texttt{R}\ \texttt{u}_0\ \texttt{u}_1 :=$$
$$\forall \texttt{s}_0.\ \texttt{sem_ref}\ (\texttt{lift_ev_rel}\ \texttt{Re})\ (\texttt{lift_res_rel}\ \texttt{R})\ (\texttt{u}_0\ \texttt{s}_0)\ (\texttt{u}_1\ \texttt{s}_0)$$

where `lift_ev_rel` and `lift_res_rel` are generated by:

$$\frac{\texttt{Re}\ \texttt{e}_0\ \texttt{e}_1\ \texttt{R}}{\texttt{lift_ev_rel Re}\ (\texttt{State.mk_ev}\ \texttt{e}_0\ \texttt{s})\ (\texttt{State.mk_ev}\ \texttt{e}_1\ \texttt{s})\ (\texttt{lift_res_rel R})}$$

$$\frac{\texttt{R}\ \texttt{x}_0\ \texttt{x}_1}{\texttt{lift_res_rel R}\ (\texttt{x}_0, \texttt{s})\ (\texttt{x}_1, \texttt{s})}$$

In particular, we apply the semantics to the same arbitrary initial state $\texttt{s}_0$; and require their final states to be equal.

4.6 Related works

Section 4.6.1 presents the techniques used to define the semantics of programs in different existing verified compilers and IRs. We then focus on several aspects related to semantics.

4.6.1 Semantics in verified compilers

4.6.1.1 Small-step semantics in CompCert

The semantics of all languages of CompCert are defined using small-step operational semantics [69, §3.5]. In low-level languages (e.g., RTL [69, §6.1]), functions are defined by CFGs of simple instructions. Except for calls, the semantics executes each instruction in a single step. To model the control flow, the states of the semantics contain a call stack recording the next instruction to execute in each active function. Higher-level languages (e.g. Cminor [69, §4.2]) feature structured statements, such as sequence and loop combinators. With small-step semantics, the control flow of those languages requires states corresponding to execution points within the statements. In CompCert, this is achieved thanks to continuations [69, §4.2.2]. Those structured languages also have alternative big-step semantics [69, §4.2.3]. Coinductive judgments are used to capture diverging executions.

The transformations are proved sound using simulations. For each transformation, a monolithic invariant is defined between the states of the source and target programs. The simulation game is then played directly. That is, the proofs *coinductively* construct executions[12]. In proofs using `SMonad`, invariants are instead defined locally, as invariants of `iter` and `mrec`, or as induction property for syntactic elements defined inductively (e.g. statements). Moreover at top-level, our proofs are *inductive* over the structure of the programs.

In order to prove that the target program only has behaviors allowed by the source program, one should prove a *backward* simulation [69, §2.1]. That is, that the source program simulates the target program. In the framework of CompCert, it is often easier to prove a *forward* simulation. That is, that the target program simulates the source program. When the target language is deterministic, forward simulation implies backward simulation. Since most of its languages are deterministic, CompCert uses forward simulations.

4.6.1.2 Big-step semantics with clocks in CakeML

CakeML [54; 60] uses big-step operational semantics. All its languages are deterministic. CakeML exploits this property to simplify its proofs of simulation in two ways. First, like CompCert, CakeML proves forward simulation. Secondly, the semantics are instrumented with "clocks" to avoid reasoning on infinite executions [60, §7]. Those clocks represent some fuel consumed by semantic constructs which can introduce divergence (the function calls in CakeML). For any initial amount of fuel, diverging executions run out of fuel after a finite time, and thus generate behaviors in the inductive big-step judgment. Thanks to those two simplifications, CakeML proves the soundness of its transformation by induction on the big-step evaluation of the source programs.

[12] In CompCert, the notion of simulation is defined as the existence of an invariant, so the proofs do not use the primitive notion of coinduction of Rocq.

4.6.1.3 Interaction Trees in VIR

The second iteration of Vellvm [105] defines a denotational semantics for VIR (a large subset of LLVM IR) using Interaction Trees (ITrees) [102]. ITrees are a coinductive datatype representing all executions of a fragment of a program as a tree. The internal nodes of the tree either represent internal τ-steps or are labelled with events. Those events are used not only to represent interactions with the context (i.e. internal function calls and external events), but also to represent effects. Indeed, the only effects supported natively by ITrees are divergence and events. However, they can be used to define domains which support additional effects (e.g. non-determinism and UB [105]). Our LTSs (without UB and restricted to deterministic internal steps) can be seen as an encoding of ITrees using transition systems instead of coinductive types.

Xia et al. [102] advocate defining semantics in several steps. With their approach, programs are first denoted into ITrees where effectful primitives are represented by events. To associate semantics to those effects, one defines an *interpretation* function. Such function maps ITrees into domains which support the effects.

ITrees are equipped with a notion of bisimulation, called "equivalence up to tau" (`eutt`). This notion is defined coinductively, but the Rocq library surrounding ITrees provides an equational theory. This theory avoids direct coinductive proofs of transformations. It is the main inspiration of `SMonad`. The relation `eutt` is heterogeneous with respect to the return types, but homogeneous with respect to the events. We generalized `eutt` with a relation `EventFam.Rel.t`. Instantiating `sem_ref` to the resulting heterogeneous equivalence yields a model of `SMonad` based on ITrees.

4.6.2 Axiomatization of denotational domains

Large parts of the interface of `SMonad` are inherited from ITrees [102]. The ITrees library was itself influenced by previous theories:

- The operators `ret` and `bind` define monads [76].
- Iteration theories [13] provide axiomatizations of the `iter` operator. Those axiomatizations are known to be complete with respect to some models based on trees [13, chapter 6].

Those theories also apply to `trigger`, `interp` and `mrec`, in a more abstract setting based on event handlers instead of continuations.

Yoon et al. [103] propose an axiomatization of a subset of the theory satisfied by ITrees and of domains derived from them. They focus on the use of events to represent effects. Hence, they do not include `mrec` in their axiomatization.

Monae [90] is a Rocq library which provides infrastructure for equational reasoning on programs defined in monads. It includes axiomatization of several effects and multiple models. Kawakami et al. [53] recently added an interface and model supporting divergence.

4.6.3 Abstraction of effects

Interaction Trees [102] represent effectful computation with events (section 4.6.1.3). This representation enables abstracting over the effects, by abstracting over the families of events. In practice, the events are interpreted incrementally [103; 105]. This yields a sequence of semantics in increasingly expressive semantic domains. Thanks to a compatibility between this interpretation and refinements, transformations that preserve effectful computations may be proven without interpreting all events, that is, in simpler domains.

For instance, in a language with internal non-determinism, the semantics could be first defined in a domain $\mathtt{S}_0$ which does not support non-determinism (e.g. Interaction Trees). When the program performs non-deterministic choices, this first semantics instead emits events in some family `Pick`. Assuming that we have a domain $\mathtt{S}_1$ which supports non-determinism, those events are interpreted

thanks to a handler `handle_Pick [E A] (u : S₀ (Pick + E) A) : S₁ E A` (we assume that the domains have the signature expected by SMonad). Semantic preservation properties are then refinements in S_1 with shape $\texttt{handle_Pick}\ u_0 \gtrsim \texttt{handle_Pick}\ u_1$, where u_0 and u_1 are semantics in S_0. If the interpretation `handle_Pick` is compatible with the refinements on S_0 and S_1, then this refinement is implied by the refinement $u_0 \gtrsim u_1$ in S_1. This compatibility property is defined in a general way by Yoon et al. [103, LiftEqmR] an example is lemma 5.6 of Zakowski et al. [105].

This approach enables the use of proof techniques specific to the domain in which the refinement is proved (S_0 in our example), for instance coinductive reasoning. However, the refinement on S_0 is generally not complete with respect to the refinement on S_1. That is, uninterpreted computations (`Pick` events) must be preserved exactly during refinements on S_0.

In the particular case of non-determinism we used as an example, the effect of the approach is to turn internal non-determinism into external non-determinism. This idea was used in CompCertTSO [92, 4.4] (with the "labellized" semantics) and by Barrière et al. [8, 4.4.3] (with the "loud" semantics) to use forward simulations on non-deterministic languages.

Our approach is instead to directly prove refinements in an abstract domain supporting all effects. The drawback is that we can only reason with an axiomatic theory (the relational reasoning approach of SMonad), but this theory can be extended with additional refinements proved separately on the abstracted computations. The erasure pass of section 7.2 will give an example of transformation enabled by those additional refinements. Our approach will enable removing effectful operations, such as standard memory reads (accessing a global state). To this end, we will describe the semantics property required on removable operators as a refinement in the abstract domain (definition 7.3).

4.6.4 Relations on events

Silver et al. [93, §4.1] proposed a heterogeneous generalization, `refines`, now called "refinement up to tau" (`rutt`). They use pre- and post-conditions to relate events. There is an injection from their relations on events into our `EventFam.Rel.t` [SMonad.Event:370]. Michelland et al. [80] define a notion of `EventLattice`[13], specialized for abstract interpretation. Those two developments relate semantics with different kinds of events. We instead use our relations on events for inter-procedural invariants between function calls. They are thus closer to the "simulation conventions" of Koenig et al. [59], introduced to reason on the compilation of open components.

4.6.5 Transformers

Moggi [76] describes monad transformers (calling them "monad constructors") in a categorical setting. Monad transformers are also useful to define modular interpreters [70]. They are formalized in Monae [4]. Yoon et al. [103] implements an infrastructure in Rocq to reason about layered interpreters. The domain of those interpreters is obtained by successive applications of monad transformers.

4.6.6 Undefined Behaviors

During refinements, UBs guarantee that some points are not reached in the source. Some formalizations of UBs allow exploiting those assumptions even if there are external events before the UB. Those "time-travelling" UBs enable more optimizations, are allowed by the specification of some languages and can be observed in production compilers [28; 29]. However, they can also make programs harder to debug [86]. Without time-travel, compilers must preserve the observable behaviors of programs up to the point where the source has a UB.

UBs are not time-travelling in the LTS-based model of SMonad (chapter 5). The original specification of CompCert (`transf_c_program_is_refinement` [69, definition 3]) allowed time-travelling UBs. In

[13] https://gitlab.inria.fr/sebmiche/itree-ai/blob/icfp24-artifact/theories/Core/Events.v#L17

order to obtain guarantees about a particular execution of the target code, one formally had to prove that the source program does not have UBs for any inputs from the environment. However, the simulations of CompCert actually guarantee the absence of time-travel. The specification of CompCert has been strengthened to reflect it (`transf_c_program_preservation` [86]). In VIR, the interpretation of UBs described by Zakowski et al. [105, §4.3] did not allow time-travel. However, the specification was changed in more recent versions to allow time-travel [14, §4.4] (implemented by the commit `vellvm:65270497` and modified by the commit `vellvm:e20334a5`).

Whether UBs are time-travelling or not, their effect on the refinement of behaviors can be modelled either by:

- Changing the refinement relation, to allow the behaviors of the target to be different from the source if there are UBs. This approach is used by CompCert (both in `transf_c_program_is_refinement` and `transf_c_program_preservation`). Trace-relating refinements [1] are a possible way of defining such refinements.
- Changing the sets of behaviors, so that programs have all behaviors once a UB is exploitable. This approach is used by VIR ([105, §4.3] and commit `vellvm:e20334a5`) and in section 5.4.

4.6.7 Non-determinism

Most languages of CompCert [69] and all languages of CakeML are deterministic. Both compilers leverage this property to simplify their proofs, notably by using forward simulations.

A possible way of adding non-determinism to an existing domain is to use sets of semantics [103, example 4.4]. In Rocq such sets are represented as predicates `S → Prop`, where `S` is a type representing deterministic semantics. Zakowski et al. [105, §5.2] define the monad operators on those sets from the operators of the underlying domain. However, their `bind` does not allow the non-deterministic choices in the right-hand side to depend on internal (i.e. not returned) values of the left-hand side. A consequence is that their `bind` is not associative.

An alternative is to use domains with built-in non-determinism, such as Choice Trees [26], DTrees [31, §6.2] or our LTSs (chapter 5). In those three domains, non-deterministic choices are represented locally, at the level of "transitions", rather than as a top-level set. Choice Trees define a notion of "delayed choice" to represent internal choices which are not observable by the context, not even as τ-steps. DTrees support dual non-determinism, that is two kinds of non-determinism (angelic and demonic) which are quantified differently during refinements.

5

A LTS-based model

The theories of `SMonad` presented in chapter 4 are parameterized by a domain `S` satisfying the axioms. Those axioms were inspired by properties of Interaction Trees. They are thus a model of the core of the axiomatization [SMonad.Model.ITreeS]. This chapter describes another model, based on Labelled Transition Systems (LTSs). We will use it as base model for our representations Frag IR and Flat IR. It natively supports the effects of non-determinism and undefined behaviors. We provide additional theories specific to LTSs.

Section 5.1 defines our representation of LTSs. Section 5.2 implements the operators of `SMonad` using combinators of LTSs. Section 5.3 instantiates the refinement relation by a notion of weak simulation. Section 5.4 associates behaviors to LTSs. Section 5.5 describes frameworks to relate our definitions to operational semantics. Finally, section 5.6 discusses related works on simulation-based refinements.

5.1 Definition of the domain

LTSs are a classical approach to formally model systems which evolve and interact with their context. With this approach, the main elements used to describe a system are:

- An arbitrary set of states describing the internal state of the system
- A transition relation between states, which depends on the interactions with the context.

In order to implement the interface of `SMonad`, we consider LTSs which interact with their context using events and return values. To instantiate the parameter `S` of `SMonad`, we define a family of types `LTS.t E A` indexed by the family `E` of events emitted and by the type `A` of values returned.

In our formalization (figure 5.1), a LTS is thus a record defining:

- A set `istate` of internal states. This set is extended using `State.F` with states in which the system interacts with its context. The names of the different kinds of states are taken from corresponding constructors of Interaction Trees. Their meaning is similar and explained after the definition 5.1.
- A predicate `istep` describing the evolution of the system when it is in an internal state.
- An initial state from which the system starts.

We gathered the fields `istate` and `istep` in a separate record `graph_t` which represents the graph of a LTS, without initial state. We implicitly coerce LTSs into their underlying graph.

The full transition relation depends on the kind of state:

```
State.F (E : EventFam.t) (iS : Type) (A : Type) : Type :=
  | Ret (x : A)
  | Tau (s : iS)
  | Vis [B] (e : E B) (k : B → iS)
  | Err

graph_t (E : EventFam.t) (A : Type) : Type := {
  istate : Type
  state := State.F E istate A
  istep : istate → state → Prop
}

LTS.t (E : EventFam.t) (A : Type) : Type := {
  graph : graph_t E A
  init : graph.state
}
```

Figure 5.1: Formal definition of our type of LTSs

Definition 5.1 The transitions of a system `u : LTS.t E A` are defined between elements of `u.state` by:

$$\frac{\texttt{u.istep s } s'}{\texttt{Tau s} \xrightarrow[\texttt{u}]{\tau} s'} \qquad \frac{\texttt{e : E B} \qquad \texttt{a : B}}{\texttt{Vis e k} \xrightarrow[\texttt{u}]{\texttt{e, a}} \texttt{Tau (k a)}}$$

That is:

- `Ret x` and `Err` are terminal states. They signal respectively the end of the execution with a result `x` and a UB.
- From an internal state `Tau s`, the system performs invisible transitions (labelled by τ) described by the predicate `istep`. Those τ-transitions are used to model two effects:
  - Systems may diverge silently (that is without interacting with their context) by performing an infinite sequence of τ transitions.
  - A state may have zero or multiple successors since `istep` is a predicate. This models internal non-determinism. We emphasize that in our formalism, a `Tau` state without successors is not a UB (which are represented with states `Err`). From such state, systems have in fact the semantics of `any False`, which is the dual of UB, as explained in section 4.2.4.1.
- A state `Vis e k` signals the emission of an event `e`. If the context provides an answer `a`, the system continues its execution in the state `k a`. The existence of multiple successors is a case of external non-determinism, since `a` is chosen by the context.

When representing LTSs graphically (e.g. in figure 5.2), we annotate the transitions with their interactions with the context (or τ). Those annotations are redundant with the value of the source state.

Our formalization of LTSs thus imposes some structure on the states and transitions:

- We separate the part of the events emitted by the LTS from the answer provided by the context.
- The different kinds of states are mutually exclusive. For instance, a system cannot perform both τ transitions and emit events from the same state; or perform transitions from a `Ret` state. Moreover, a `Vis` state emits a single kind of events.
- Internal non-determinism may only occur during τ transitions. Indeed, the successor of a `Vis` state is a deterministic function of the answer received from the context.

Our LTSs natively models the following effects:

Divergence Infinite chains of transitions correspond to diverging executions. A LTS may have such infinite chains if there are loops in the graph of its transitions or if it has an infinite set of states.

Events A LTS may interact with its context during its execution, thanks to the `Vis` states.

UB Executions can get stuck in the `Err` state.

Non-determinism The definition of `istep` as a predicate enables internal non-determinism.

5.2 Operators

We implemented the operators of `SMonad` on `LTS.t`. The operators `ret`, `trigger`, $\texttt{any}_0$ and $\texttt{ub}_0$ are LTSs with the expected behaviors. We represented them graphically in figure 5.2.

The operators `bind`, `iter`, `map_ev` and `mrec` are functions which combine several LTSs. Figure 5.3 reproduces the formal definition of the operators.

`bind` takes as argument a LTS `u` which returns values of type `A` and a family `k` of LTS indexed by `A`. The type of its internal states is the disjoint union of the internal states of its arguments. The constructors `Bind.Su` and `Bind.Sk (x : A)` are the corresponding injections. Modulo those injections, the transitions of `bind` are mostly the same as the transitions of `u` and `k`. The only exception is that whenever `u` would go to a state `Ret x` (as initial state or after an internal step), `bind` instead goes to the initial state of `k x`. Figure 5.4 depicts this transformation. Formally, we apply an auxiliary function `State.map` on the target state of internal transitions. This function applies an injection `f_iS` on the internal states and replaces `Ret` states according to `f_ret`.

`iter` takes as argument a family `f` of LTSs indexed by a type `A`. The return type of those LTSs is a disjoint union $\texttt{A} + \texttt{B}$. Similarly to `bind`, `iter` includes all its sub-LTSs in its states and transitions, through an injection `Iter.Sf`. To connect them, `iter` must go to the initial state of `f x` whenever its body goes to `Ret (Continue x)`. However, this initial state may itself have shape `Ret (Continue _)`, and thus needs to be replaced. This redirection process can happen infinitely many times in a silently diverging `iter`, for instance in figure 5.5. We capture this divergence by adding a τ-step between iterations. For this purpose, we add an internal state `Iter.Sloop x` for each `x : A`. From this state, `iter` performs a τ-transition to the initial state of `f x`. Figure 5.6 gives an example of a LTS defined using `iter`.

`mrec`

> **TODO** (rocq)
>
> actuellement Stack.t est plus général: la forme du stack peut dépendre de la valeur de retour

takes as argument a family of LTSs indexed by events representing function calls [SMonad.Model.LTS.Op:278]. The state of `mrec` is a call stack. Each frame of the stack stores the current state of the LTS corresponding to the call. Except for the function currently executed (i.e. the deepest call), this state depends on the return value of the call which interrupted the execution of the function. The transitions of `mrec` depend on the transition of the LTS of the current function. When this LTS returns, we pop its frame from the stack. If the stack becomes empty, `mrec` goes to a `Ret` state. When the current function goes to a state `Vis (inl d) k` (that is, performs an internal call), `mrec` pushes a frame for `d` on the stack. Like `iter`, `mrec` performs a τ step before going to the initial state of `d`. Thanks to this step, `mrec` performs an infinite sequence of τ steps if an infinite sequence of functions directly start with a function call.

We give a direct definition to `map_ev` and use it to define `interp` as a particular case of `mrec` [SMonad.Generic.DeriveOp:50].

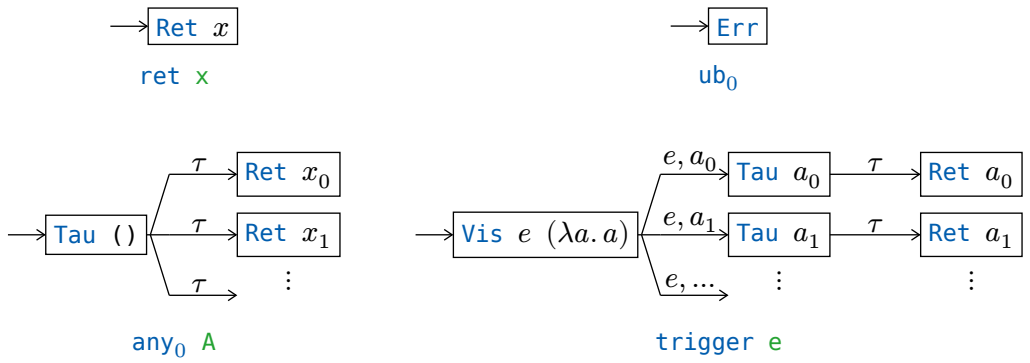


Figure 5.2: Graphical representation of the implementations of operators on the domain `LTS.t`

```
State.map [E iS₀ iS₁ A₀ A₁] (f_iS : iS₀ → iS₁) (f_ret : A₀ → State.F iS₁ A₁)
  (s : State.F E iS₀ A₀) : State.F E iS₁ A₁ :=
  { f_ret x               if s = Ret x
  { Tau (f_iS s₀)         if s = Tau s₀
  { Vis e (λa. f_iS (k a)) if s = Vis e k
  { Err                   if s = Err
```

```
(bind u k).istate := Bind.Su (s : u.state) | Bind.Sk (x : A) (s : (k x).state)
Bind.map_k (x : A) : (k x).state → (bind u k).state := State.map (Bind.Sk x) Ret
Bind.map_u : u.state → (bind u k).state := State.map Bind.Su (λx. Bind.map_k x (k x).init)
(bind u k).init := Bind.map_u u.init
```

$$\frac{\texttt{u.istep s t}}{\texttt{(bind u k).istep (Bind.Su s) (Bind.map_u t)}}$$

$$\frac{\texttt{(k x).istep s t}}{\texttt{(bind u k).istep (Bind.Sk x s) (Bind.map_k x t)}}$$

```
(iter f ini).istate := Iter.Sf (x : A) (s : (f x).state) | Iter.Sloop (x : A)
Iter.map_f (x : A) : (f x).state → (iter f ini).state :=
  State.map (Iter.Sf x) (λr. { Tau (Iter.Sloop x')  if r = Continue x'
                             { Ret y                if r = Break y )
(iter f ini).init := Iter.map_f ini (f ini).init
```

$$\frac{\texttt{(f x).istep s t}}{\texttt{(iter f ini).istep (Iter.Sf x s) (Iter.map_f x t)}}$$

$$\frac{}{\texttt{(iter f ini).istep (Iter.Sloop x) (Iter.map_f x (f x).init)}}$$

Figure 5.3: Definition of combinators of LTSs

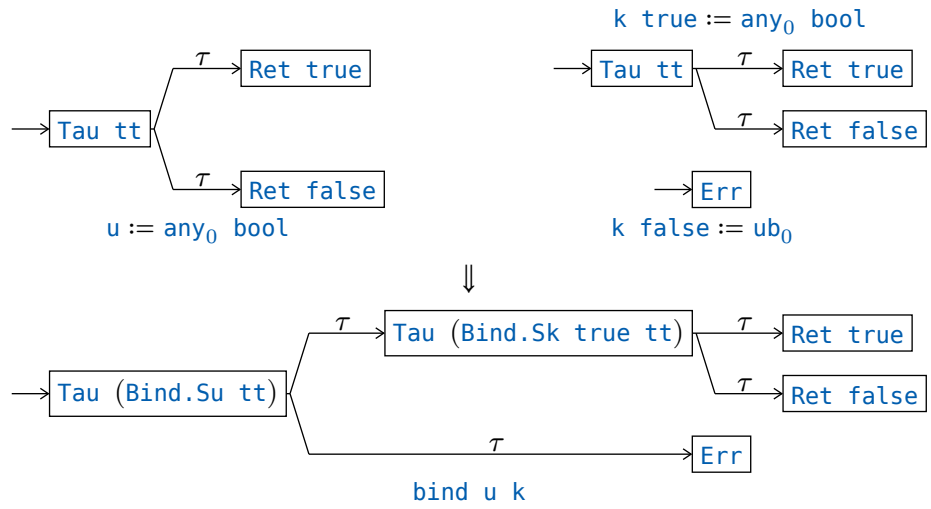


Figure 5.4: Example of a LTS defined using `bind`

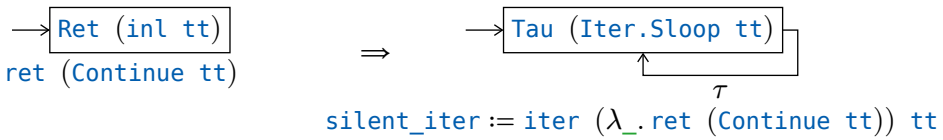


Figure 5.5: Silent divergence introduced by `iter`

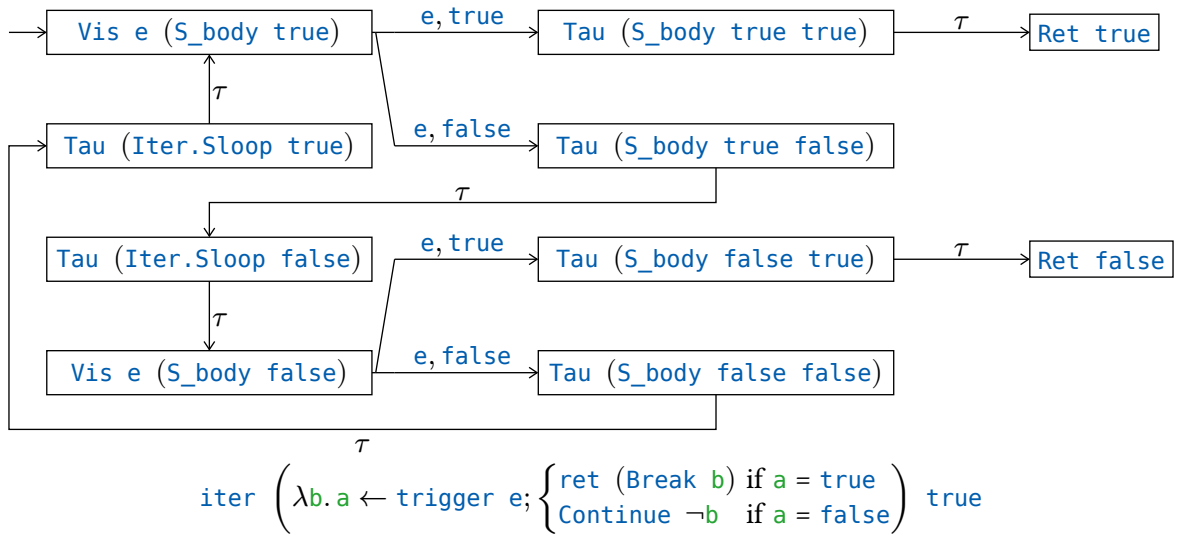


Figure 5.6: Example of a LTS defined using `iter`. The system emits some event `e` until it receives `false` as answer. It returns `true` if it emitted an odd number of events. An internal state `S_body b a` is an abbreviation for `Iter.Sf b (Bind.Su a)`. It is reached when `trigger` receives an answer `a` during an iteration of `iter` with state `b`.

5.3 Simulations

We define a heterogeneous refinement relation between LTSs (`sem_ref`) using a notion of *simulation*. Simulations are proof techniques to prove some relation (e.g. inclusion) between the sets of executions of two LTSs. Those approaches reduce this problem to local comparisons between finite sequences of transitions starting at similar states related by an *invariant*. In the following definition, we fix two LTSs u_0 : `LTS.t` E_0 A_0 and u_1 : `LTS.t` E_1 A_1 and some relations `Re` : `EventFam.Rel.t` E_0 E_1 and `R` : `Rel` A_0 A_1 on their events and return values. In the refinement, u_0 will be the source and u_1 the target.

Definition 5.2 A candidate invariant `M` : `Rel` u_0.`istate` u_1.`istate` is a relation between the internal states of the LTSs. We extend such relation to all states by defining $\overline{\texttt{M}}$: `Rel` (u_0.`state`) (u_1.`state`) by:

$$\frac{\texttt{R}\ x_0\ x_1}{\overline{\texttt{M}}\ (\texttt{Ret}\ x_0)\ (\texttt{Ret}\ x_1)} \qquad \frac{\texttt{M}\ s_0\ s_1}{\overline{\texttt{M}}\ (\texttt{Tau}\ s_0)\ (\texttt{Tau}\ s_1)} \qquad \frac{}{\overline{\texttt{M}}\ \texttt{Err}\ s}$$

$$\frac{\texttt{Re}\ e_0\ e_1\ \texttt{P} \qquad \forall a_0\, a_1.\, \texttt{P}\ a_0\ a_1 \Rightarrow \texttt{M}\ (k_0\ a_0)\ (k_1\ a_1)}{\overline{\texttt{M}}\ (\texttt{Vis}\ e_0\ k_0)\ (\texttt{Vis}\ e_1\ k_1)}$$

We compare the transitions starting from two states s_0 : u_0.`state` and s_1 : u_1.`state` using a predicate `simu_step` s_0 s_1 b_0 b_1. This predicate is satisfied if $\overline{\texttt{M}}$ is established after some τ-steps of u_0 and u_1. If the Boolean b_i is set, u_i must perform a strictly positive number of τ-steps. Formally, we define this predicate inductively:

$$\frac{\overline{\texttt{M}}\ s_0\ s_1}{\texttt{simu_step}\ s_0\ s_1\ \texttt{false}\ \texttt{false}}\ \texttt{RefNow}$$

$$\frac{\exists s_0'.\, u_0.\texttt{istep}\ s_0\ s_0' \wedge \texttt{simu_step}\ s_0'\ s_1\ \texttt{false}\ b_1}{\texttt{simu_step}\ (\texttt{Tau}\ s_0)\ s_1\ b_0\ b_1}\ \texttt{TauSrc}$$

$$\frac{\forall s_1'.\, u_1.\texttt{istep}\ s_1\ s_1' \Rightarrow \texttt{simu_step}\ s_0\ s_1'\ b_0\ \texttt{false}}{\texttt{simu_step}\ s_0\ (\texttt{Tau}\ s_1)\ b_0\ b_1}\ \texttt{TauTrg}$$

The relation `M` is a *simulation invariant* if it is reestablished strictly later after any states it relates, that is if:

$$\forall s_0\, s_1.\, \texttt{M}\ s_0\ s_1 \Rightarrow \texttt{simu_step}\ (\texttt{Tau}\ s_0)\ (\texttt{Tau}\ s_1)\ \texttt{true}\ \texttt{true}$$

The LTS u_0 *simulates* u_1 if there exists a simulation invariant which relates their initial states.

We instantiate `sem_ref` on `LTS.t` as the *backward* simulation: $u_0 \gtrsim_{\texttt{Re},\texttt{R}} u_1$ holds if and only if u_0 simulates u_1.

- Such simulation describes an algorithm which takes as input an execution of u_1 and builds a corresponding execution of u_0. The algorithm maintains a pair of states related by `M`. At each step, it reads a finite number of transitions performed by u_1 and produces corresponding transitions of u_0.
- This definition contains two sources of asymmetry between u_0 and u_1:
  - $\overline{\texttt{M}}$ relates a source `Err` to any target state. The simulation relation is thus satisfied if the source reaches a UB.
  - The predicate `simu_step` quantifies universally on the target τ-steps (`TauTrg`) but existentially on the source τ-steps (`TauSrc`). This is the correct way of handling non-deterministic τ-steps, since one wants to generate a source execution from a target execution.

- `simu_step` defines a *weak* simulation. That is, the source and target LTSs do not need to perform the same number of τ-transitions. This enables adding or removing finite sequences of τ-transitions during simulations. The Boolean indexes of `simu_step` ensure that at least a step is performed in the source and in the target. This restriction rules out situations of "infinite stuttering" [69, §3.7], where the simulation never progresses in one of the LTSs. In particular, simulations cannot add or remove infinite sequences of τ-steps (except if the source has a UB).

We prove all axioms of `SMonad` for this refinement.

5.4 Behaviors of LTSs

A behavior represents a possible execution of a LTS. Those behaviors are used to specify programs and compilation. We define the sets of behaviors of our LTSs in section 5.4.1. Section 5.4.2 compares our definitions with those used in CompCert.

5.4.1 Set of behaviors of a LTS

Our type `Beh.t E A` of behaviors is indexed by the family `E` of events emitted and the type `A` of values returned. It is defined coinductively:

```
coinductive Beh.t E A := Beh.Ret (x : A)
                       | Beh.Vis [B] (e : E B) (a : B) (tl : Beh.t)
                       | Beh.Stp [B] (e : E B)
                       | Beh.Div
                       | Beh.Err
```

A behavior starts with a stream of interactions `Beh.Vis` with the environment through events of `E`. Each element of this stream contains the event `e` emitted by the LTS and the answer `a` received from the context. This stream is infinite if the LTS diverges reactively. Otherwise, it is terminated by one of the other constructors:

`Beh.Ret x` if the LTS terminates its execution by returning a value `x`.

`Beh.Stp e` if the LTS emits an event `e` but does not receive an answer from its context (its execution is "stopped").

`Beh.Div` if the LTS diverges without emitting any events.

`Beh.Err` if the LTS reaches an error state, thus signaling a UB.

Because of internal and external non-determinism, a LTS can have several behaviors. Figure 5.7 define the set of behaviors `lts_behaves` of a LTS, as the set `stbhv` ("state behaves") of its initial state. This second set is defined as an inductive-coinductive predicate. That is, in order to include infinite behaviors (silent divergence `Beh.Div` and infinite sequences of events `Beh.Vis`), proofs of `stbhv` can be infinite sequences of its constructors. For instance:

Example 5.1 The empty infinite loop `silent_iter := iter (λ_. ret (Continue tt)) tt` (depicted in figure 5.5) can diverge silently:

```
lts_behaves silent_iter Beh.Div
```

Proof We construct coinductively an infinite sequence of `bhv_Tau`s, using to the self loop `silent_iter.istep (Iter.Sloop tt) (Tau (Iter.Sloop tt))` on the unique state of `silent_iter`.□

However, like for simulation, one needs to restrict the use of coinduction to only capture the desired behaviors. This is the reason why we use an inductive-coinductive predicate instead of a coinductive predicate. In figure 5.7, the two recursive occurrences of `stbhv` are annotated with a boolean `b` (with the notation $\boxed{\ldots}^{b}$). The occurrences where this boolean is true are coinductive, the others are inductive. Proofs of inductive-coinductive predicates must not contain infinite sequences of inductive constructors. For `stbhv`, this means that an infinite sequence of `bhv_Tau` (not containing any `bhv_Vis`) can only be used once the behavior is `Beh.Div`. Without this restriction, `silent_iter` would admit all behaviors. With our definition, we can instead prove that:

```
stbhv [E A] (u : graph_t E A) (s : u.state) (beh : Beh.t E A) : Prop

------------------------------ bhv_Ret
stbhv u (Ret x) (Beh.Ret x)

------------------ bhv_Err
stbhv u Err beh

stbhv u (Tau (k a)) tl [true]
------------------------------ bhv_Vis
stbhv u (Vis e k) (Beh.Vis e a tl)

------------------------------ bhv_Stp
stbhv u (Vis e k) (Beh.Stp e a)

u.istep s0 s1    stbhv u s1 beh [beh=Beh.Div]
------------------------------ bhv_Tau
stbhv u (Tau s0) s1

lts_behaves [E A] (u : LTS.t E A) : Beh.t E A → Prop := stbhv u.graph u.init
```

Figure 5.7: Sets of behaviors of LTSs

Example 5.2 `silent_iter` does not admit any other behavior than `Beh.Div`:

$$\forall \texttt{beh}.\ \texttt{lts_behaves silent_iter beh} \Rightarrow \texttt{beh} = \texttt{Beh.Div}$$

Combined with example 5.1, this proves that `lts_behaves silent_iter` $= \{$`Beh.Div`$\}$.

Proof By induction on `stbhv`, using the fact that the only outgoing transition of `Iter.Sloop tt` is its self loop. □

From an `Err` state, a LTS has all behaviors, including `Beh.Err`. This captures the semantics of UBs directly in the definition of the set of behaviors.

The "stop" terminator, `Beh.Stp`, is useful to distinguish LTSs emitting events with empty answer type from LTSs which get stuck in a state without outgoing transitions. In our formalism, such states are the opposite of UBs, and do not produce behaviors. Consider the three following LTSs, where the $\texttt{e}_{\texttt{B}}$ are events with return type `B`:

$$\texttt{lts_behaves}\ (\texttt{any False}) = \varnothing$$
$$\texttt{lts_behaves}\ (\texttt{trigger}\ \texttt{e}_{\texttt{unit}};;\texttt{any False}) = \{\texttt{Beh.Stp}\ \texttt{e}_{\texttt{unit}}\}$$
$$\texttt{lts_behaves}\ (\texttt{trigger}\ \texttt{e}_{\varnothing}) = \{\texttt{Beh.Stp}\ \texttt{e}_{\varnothing}\}$$

The two first LTSs above are stuck at empty internal non-deterministic choices (`any False`), but the second one emits an event before. The third LTS is instead stuck at an external event with empty answer type ($\texttt{e}_{\varnothing}$). Without `Beh.Stp`, the three LTSs would all have an empty set of behaviors.

Set-inclusion provides a homogeneous refinement relation between sets of behaviors. We thus define a refinement relation between two LTSs `src` and `trg` with the same events and return type as:

$$\texttt{behavior_ref src trg} := \texttt{lts_behaves src} \supseteq \texttt{lts_behaves trg}$$

This definition allows the target behavior to refine the non-deterministic choices made by the source. It also allows the target to have any behavior when the source has a UB. The correctness of homogeneous backward simulation with respect to refinements of behaviors is a well-known property:

Lemma 5.1 For any LTSs `src` and `trg` with the same signature:

$$\texttt{src} \gtrsim_{\texttt{id},\texttt{id}} \texttt{trg} \Rightarrow \texttt{behavior_ref src trg}$$

5.4.2 Comparison with the formalization used by CompCert

The main correctness theorems of CompCert relate sets of behaviors of the source and target programs [69, §3.5]. The formalization of behaviors we have presented in the previous sections differs from the one used in CompCert on several points:

- Our type `Beh.t` represents termination and silent divergence as terminators, after an unbounded number of `Beh.Vis`. CompCert instead distinguishes the different kinds of behaviors up-front. It uses

```
trace E := Enil | Econs [B] (e : E B) (a : B) (tl : trace)
coinductive traceinf E := Econsinf [B] (e : E B) (a : B) (tl : traceinf)
behavior E A := Terminates (t : trace) (r : A)
              | Stop_event (t : trace) [B] (e : E B)
              | Diverges (t : trace)
              | Reacts (T : traceinf)
              | Goes_wrong (t : trace)
```

Figure 5.8: Alternative type of behaviors, by case, in the style of CompCert. The type `behavior` of all behaviors is a variant with a constructor for each kind of behaviors. The auxiliary types `trace` and `traceinf` describe respectively the finite and infinite sequences of events.

a separate type to represent each kind of behavior. The type of all behaviors is obtained as their disjoint union. In the constructive logic of Rocq, it is possible to decide how a behavior terminates or diverges if it is given with the definition of CompCert, but we conjecture that it is not possible with our definition.

- The behaviors of CompCert do not have the equivalent of `Beh.Stp`. In the LTS of CompCert, states without outgoing transitions represent UB and all events have a non-empty return type. Stopping behaviors at events is thus not useful since partial executions can always be completed. However, events with empty answer types could be useful for languages with empty types, like `!` in Rust.
- Thanks to our single type of behavior, we are able to define the behaviors starting from a state as a single inductive-coinductive predicate. On the contrary, CompCert gives a separate predicate for each kind of behavior.
- In CompCert, an error state produces only an error behavior (the equivalent of our `Beh.Err`). Set inclusion thus does not allow replacing source UBs by arbitrary behaviors in the target. CompCert instead captures this refinement as a comparison relation `behavior_improves` between behaviors. In its correctness theorem `transf_c_program_preservation` sets of behaviors (`program_behaves`) are then compared using a trace relating refinement [1]:

  $$\forall \texttt{b} \in \texttt{program_behaves trg}.\ \exists \texttt{b}' \in \texttt{program_behaves src}.\ \texttt{behavior_improves b' b}$$

  This property implies the inclusion of the sets of behaviors if the source program does not have any UB (`transf_c_program_is_refinement`). This corollary was the original specification of CompCert [69, definition 3].

We also defined the behaviors of our LTSs in the style of CompCert (figure 5.8). However, contrary to CompCert, we added behaviors `Stop_event`, emitted as soon as the LTS emits an event. We proved an equivalence between our two definitions.

Lemma 5.2 A LTS `u` admits a coinductive behavior `beh_coind` : `Beh.t E A`, if and only if it admits a corresponding behavior `beh_case` : `behavior E A` in the type defined by case:

$$\texttt{lts_behaves u beh_coind u beh_coind} \leftrightarrow$$
$$(\exists \texttt{beh_case}.\ \texttt{beh_conv beh_case beh_coind} \wedge \texttt{lts_behaves_cases u beh_case})$$

where `lts_behaves_cases u` is the set of behaviors of `u` with the definition by case (with a definition similar to the one of CompCert), and `beh_conv` is a conversion relation between `Beh.t` and `behavior`.

We use this alternative definition of behaviors when interfacing with the theory of CompCert in section 5.5.4. This equivalence also gives more confidence in our inductive-coinductive definition of sets of behaviors.

5.5 Connection with operational semantics

With `SMonad`, we advocate the use of denotational semantics to reason on programs. However, one sometimes needs to connect those semantics to operational semantics, either in small-step or big-step

$$\frac{\forall x.P_0\ x \leftrightarrow P_1\ x \qquad u_0 \approx u_1 \qquad \texttt{FE.p}\ P_0\ u_0}{\texttt{FE.p}\ P_1\ u_1}\ \texttt{FE.morph} \qquad \frac{}{\texttt{FE.p}\ (\lambda x'.\, x' = x)\ (\texttt{ret}\ x)}\ \texttt{FE.ret}$$

$$\frac{\texttt{FE.p}\ P\ u \qquad \forall x.\, P\ x \Rightarrow \texttt{FE.p}\ (Q\ x)\ (k\ x)}{\texttt{FE.p}\ (\lambda y.\, \exists x.\, P\ x \wedge Q\ x\ y)\ (\texttt{bind}\ u\ k)}\ \texttt{FE.bind} \qquad \frac{}{\texttt{FE.p}\ (\lambda _.\, \texttt{False})\ \texttt{ub}}\ \texttt{FE.ub}$$

Figure 5.9: Lemmas enabling proofs of `FE.p`

style. This is needed to interact with existing developments which use operational semantics. Another motivation for using small-step semantics is that their definition models the way programs are actually executed on computers. They are thus more adapted to be used in trusted specifications for low-level languages.

We implemented two frameworks enabling proofs of equivalence between our denotational semantics and operational semantics. The first one, `FromEval`, handles deterministic, event-free big-step semantics (section 5.5.1). The second one, `FromOperational`, is specific to the model `LTS.t` but handles more expressive small-step semantics (section 5.5.2). We use the framework `FromOperational` in section 5.5.3 to prove the soundness of executable interpreters. As an example of a connection with an existing semantics, we gave a denotational semantics to the language Cminor of CompCert. We use our frameworks to prove its equivalence with the original semantics in section 5.5.4. The Rocq development [SMonadExamples.MiniBTL.SemEquivalence] also contains proofs of equivalence using `FromOperational` between the operational and denotational semantics of MiniBTL (figures 4.2 and 4.4). We will also use `FromEval` in section 10.4.1 to give a denotational semantics to another existing language, Obc.

5.5.1 `FromEval`: equivalences with big-step semantics

Our framework `FromEval` (abbreviated `FE`) enables proofs of equivalence with big-step semantics whose sole effects are undefined behaviors. Such semantics is defined by a predicate `P : A → Prop` satisfied by at most one return value of type `A`. A predicate `P` which is never satisfied models a UB. Our framework is centered around a relation between those predicates and semantics defined in a domain `S` of `SMonad` supporting UBs. The domain `S` is a parameter of the framework, and `LTS.t` a possible instance.

This central relation is `FE.p [E A] (P : A → Prop) (u : S E A) : Prop`, where `P` is a predicate defining a big-step semantics and `u` a monadic semantics. This relation distinguishes two cases: successful deterministic evaluations and UBs.

Definition 5.3 The relation `FE.p P u` is satisfied if either:
- There exists a result `x : A` such that `P x'` holds exactly for $x' = x$ and $u \approx$ `ret x`.
- `P x'` does not hold for any x' and $u \approx$ `ub`.

It was convenient to define `FE.p` for a semantics `u` with arbitrary events `E` in its signature. However, `FE.p P u` implies that `u` does not emit any events. It also implies that the predicate `P` is deterministic, that is, satisfied by at most one value.

We prove lemmas which enable compositional proofs of `FE.p`, following the structure of the monadic semantics and without unfolding the definition of `FE.p`. Figure 5.9 reproduces the main lemmas. Using them, we prove goals `FE.p P u` with the following strategy:

1. Using the lemmas corresponding to the operators used to build the monadic semantics `u` (e.g., `FE.ret`, `FE.bind` and `FE.ub`), we prove `FE.p P' u` between `u` and a predicate `P'` inferred from its structure.
2. We apply `FE.morph` and conclude by proving an equivalence between the propositions `P` and `P'`.

5.5.2 `FromOperational`: equivalences with small-step semantics

Our framework `FromOperational` (abbreviated `FO`) enables the proof of equivalences between small-step operational semantics and denotational semantics defined in `LTS.t`. Formally, such equivalence is a goal:

$$\texttt{opr} \approx_{\texttt{Re},\texttt{R}} \texttt{den}$$

where `opr` and `den` are respectively the operational and denotational semantics. Both of them are LTSs, but they are defined differently:

- `opr` should be an operational small-step semantics. That is, it is directly defined by a set of states and a transition relation.
- `den` should be defined compositionally, using the operators of `SMonad`.

Remember that we define the equivalence between semantics as refinement in both directions. On LTSs, those refinements are simulation. Instead of directly specifying the invariant of such simulation, the framework leverages the compositional definition of the denotational semantics.

It is centered around a predicate `FO.p` which relates a subpart of the denotational semantics to a subpart of the graph of the operational semantics. This predicate has the following signature:

```
FO.p [E_ext A_ext] (G : graph_t E_ext A_ext)
  [E] (pE : ∀(g : bool) [B] (e : E_ext B) (s₁ : G.state)
             (pB : B → G.state → Prop). Prop)
  [A] (pA : bool → A → G.state → Prop)
  (u : LTS.t E A) (s₀ : G.state) :
  Prop
```

This predicate is parameterized by the graph `G` of the operational semantics, which is fixed during the whole proof of equivalence. It relates a fragment `u` of the denotational semantics to a state $\texttt{s}_0$ of the operational semantics. The state $\texttt{s}_0$ should correspond to the beginning of the execution of `u`. The arguments `pA` and `pE` describe how the interactions of `u` with its context are implemented in the operational semantics. Those two predicates relate interactions of `u` (return value and events) to the states where the operational semantics can be when those interactions are performed. To avoid infinite stuttering, we track using Booleans ("guards") whether the operational semantics has performed some steps during segments of the execution. In more details:

- `pA g x` $\texttt{s}_1$ holds if the operational semantics can be in the state $\texttt{s}_1$ when `u` returns `x`. If the guard `g` is set, the operational semantics must have performed at least a step since the beginning of the execution of `u` (i.e. since $\texttt{s}_0$) when this state is reached.
- `pE g e` $\texttt{s}_1$ `pB` holds if:
  - The operational semantics can be in the state $\texttt{s}_1$ when `u` emits `e`. If the guard `g` is set, there must have been a step since the beginning of the execution of `u`.
  - When `u` receives an answer `b` from event `e`, the operational semantics is in a state $\texttt{s}_2$ satisfying the post-condition `pB b` $\texttt{s}_2$.

Figure 5.10 describes the properties satisfied by `FO.p`. This definition uses helpers `guard`, `unguard` and `unguard_if` to manage the guards. We overload the names to use them both for propositions and for the predicates at return points and events. The helper `guard` prevents uses of the wrapped invariant if its Boolean argument is not set, on propositions it is defined as `guard (g : bool) (P : Prop) : Prop := (g = true ∧ P)`. On the contrary, `unguard` ignores its Boolean argument and calls its wrapped predicate with `true` as argument, on propositions `unguard (P : bool → Prop) (_g : bool) : Prop := P true`. Finally, `unguard_if` $\texttt{g}_0$ applies `unguard` if $\texttt{g}_0$ is `true`, and is the identity otherwise. The rule `FO.mono` involves comparisons "$\leq$" between the predicates for the interactions of the semantics. We do not define them here, but in first approximation, they can be thought of as pointwise implications.

TODO (rocq)
- rename: monadic -> denotational / compositional
- tau_opr: comment PSet.eq
- pE_ext, pR_ext

$$\frac{\texttt{FO.p opr } (\lambda_.\,\texttt{FO.pE_ext Re}) \ (\lambda_.\,\texttt{FO.pR_ext R}) \ \texttt{den } (\texttt{opr.init})}{\texttt{opr} \approx_{\texttt{Re,R}} \texttt{den}}\ \texttt{FO.intro_sem_eq}$$

where: $$\frac{\texttt{Re } e_0 \ e_1 \ \texttt{P}}{\texttt{FO.pE_ext Re } e_1 \ (\texttt{Vis } e_0 \ k_0) \ (\lambda x_1\, s_0.\, \exists x_0.\, s_0 = k_0 \ x_0 \wedge \texttt{P } x_0 \ x_1)} \qquad \frac{\texttt{R } x_0 \ x_1}{\texttt{FO.pR_ext R } x_1 \ (\texttt{Ret } x_0)}$$

$$\frac{\texttt{pE}_0 \le \texttt{pE}_1 \quad \texttt{pA}_0 \le \texttt{pA}_1 \quad \texttt{FO.p G pE}_0 \ \texttt{pA}_0 \ \texttt{u s}}{\texttt{FO.p G pE}_1 \ \texttt{pA}_1 \ \texttt{u s}}\ \texttt{FO.mono}$$

$$\frac{u_0 \approx u_1 \quad \texttt{FO.p G pE pA } u_0 \ \texttt{s}}{\texttt{FO.p G pE pA } u_1 \ \texttt{s}}\ \texttt{FO.morph}$$

$$\frac{\begin{array}{c}\exists s_1.\, \texttt{G.istep } s_0 \ s_1 \\ \forall s_1.\, \texttt{G.istep } s_0 \ s_1 \Rightarrow \texttt{FO.p G (unguard pE) (unguard pA) u } s_1\end{array}}{\texttt{FO.p G pE pA u (Tau } s_0)}\ \texttt{FO.tau_opr}$$

$$\frac{\texttt{pA false x s}}{\texttt{FO.p G pE pA (ret x) s}}\ \texttt{FO.ret} \qquad \frac{\texttt{pE false e } s_0 \ \texttt{(pA false)}}{\texttt{FO.p G pE pA (trigger e) } s_0}\ \texttt{FO.trigger}$$

$$\frac{\begin{array}{c}\texttt{FO.p G pE pA u } s_0 \\ \forall \texttt{l x } s_1.\, \texttt{pA l x } s_1 \Rightarrow \texttt{FO.p G (unguard_if l pE) (unguard_if l pB) (k x) } s_1\end{array}}{\texttt{FO.p G pE pB (bind u k) } s_0}\ \texttt{FO.bind}$$

$$\frac{\begin{array}{c}\texttt{pA false ini } s_0 \\ \forall \texttt{gl } x_0\, s_1.\, \texttt{pA gl } x_0 \ s_1 \Rightarrow \texttt{FO.p G (unguard_if gl pE) (FO.iter_pAB gl) (f } x_0) \ s_0\end{array}}{\texttt{FO.p G pE pB (iter f ini) } s_0}\ \texttt{FO.iter}$$

where: $$\texttt{FO.iter_pAB gl g r } s_2 := \begin{cases} \texttt{guard g (pA true } x_1 \ s_2) & \text{if } \texttt{r} = \texttt{inl } x_1 \\ \texttt{pB (gl} \vee \texttt{g) y } s_2 & \text{if } \texttt{r} = \texttt{inr y} \end{cases}$$

$$\frac{\begin{array}{c}\texttt{pD false Aini ini } s_0 \ \texttt{(pAini true)} \\ \forall \texttt{gr } d_0\, s_1\, \texttt{pA}.\, \texttt{pD gr } d_0 \ s_1 \ \texttt{pA} \Rightarrow \\ \texttt{FO.p G (FO.mrec_pDE gr) } (\lambda \texttt{g x } s_2.\, \texttt{guard g (pA x } s_2)) \ (\texttt{f } d_0) \ s_1\end{array}}{\texttt{FO.p G pE pAini (mrec f ini) } s_0}\ \texttt{FO.mrec}$$

where: $$\texttt{FO.mrec_pDE gr g de } s_2 \ \texttt{pB} := \begin{cases} \texttt{guard g (pD true } d_1 \ s_2 \ \texttt{pB}) & \text{if } \texttt{de} = \texttt{inl } d_1 \\ \texttt{pE (gr} \vee \texttt{g) e } s_2 \ \texttt{pB} & \text{if } \texttt{de} = \texttt{inr e} \end{cases}$$

$$\frac{\forall \texttt{x} : \texttt{A}.\, \exists s_1.\, \texttt{G.istep } s_0 \ s_1 \wedge \texttt{pA true x } s_1 \qquad \forall s_1.\, \texttt{G.istep } s_0 \ s_1 \Rightarrow \exists \texttt{x} : \texttt{A}.\, \texttt{pA true x } s_1}{\texttt{FO.p G pE pA (any A) (Tau } s_0)}\ \texttt{FO.any}$$

Figure 5.10: Interface of the framework FromOperational

$$\frac{}{\texttt{FO.p pE pA ub Err}}\ \texttt{FO.ub}$$

A proof using `FromOperational` starts with an application of the lemma `FO.intro_sem_eq`. This lemma transforms the goal $\approx$ into a `FO.p` for the whole denotational semantics `den`, starting at the initial state `opr.init` of the operational semantics. The whole semantics `den` only interacts with the external environment. Those interactions are described by `FO.pE_ext` and `FO.pR_ext`, which relate them to the states `Vis` and `Ret` of `opr`. Proofs using `FromOperational` then follow the structural definition of the denotational semantics. For each operator of `SMonad`, a lemma enables the decomposition of `FO.p`. For instance:

- `FO.ret` proves a `FO.p` between a denotational semantics `ret x` and a state `s` related to `x` by `pA`.
- To apply `FO.bind`, one must specify the relation `pA` between the result of the left-hand side `u` and the corresponding states of the operational semantics. The first premise then requires `FO.p` for the left-hand side from the initial state s_0 of the whole `bind`, but with `pA` as a predicate on the results. The second premise requires `FO.p` for the right-hand side `k x` from s_1, for any intermediate result `x` and state s_1 related by `pA`.

Thanks to the structure of the denotational semantics, there are no coinductive rules. Diverging executions are handled with local invariants `pA` (resp. `pD`) in `FO.iter` (resp. `FO.mrec`). In particular, the τ-steps of the denotational semantics do not appear in the interface of `FromOperational`. At any point, `FO.morph` can be used to perform rewritings in the fragment of the denotational semantics. The first premise of this rule only requires $\approx$, which is an equivalence modulo finite sequences of τ-steps. However, we do not assume any structure on the operational semantics. In particular, its τ-steps are handled explicitly, with the lemma `FO.tau_opr`. Those τ-steps are performed independently of the structure of the denotational semantics. The guard Booleans prevent situations of infinite stuttering, where an `iter` or `mrec` diverges silently while the operational semantics never perform transitions. For instance in `FO.iter`, the invariant `pA` is guarded in `FO.iter_pAB`. It can thus only be reestablished if the operational semantics performs some step during the execution of the body `f`.

5.5.3 Application: a library to define interpreters

The semantics of a program describes its possible behaviors. However in general, a semantics is not enough to execute a program. In particular, our LTSs describe the τ-transitions using a predicate. Giving a description of the semantics of a program as an inhabitant of `LTS.t` thus does not allow executing the program, either using the reduction mechanism of Rocq or its extraction procedure. However, the compositional approach used to define semantics is also applicable to the definition of interpreters. We implemented in Rocq a library which provides operators similar to those of `SMonad` for an executable representation. In order to perform effects (divergence, external events) the library defines interpreters as Executable Transition Systems (ETSs). Those interpreters are executed by extracting their definition to OCaml and combining them with a small loop which calls their step function repeatedly.

The difference between ETSs and LTSs is that internal steps are defined by a function rather than by a predicate. We still allow non-deterministic choices in the ETSs, represented by dedicated "τ-events". The family `TE` of those events depends on the kind of program interpreted, and is a parameter of the interpreter library. The execution environment (i.e. the OCaml loop which runs the interpreter) is responsible for interpreting the τ-events, for instance by choosing a default value.
The operators of LTSs could be adapted to this description of steps. However, the generated interpreters would be rather inefficient. Indeed, their states would be the same as their LTS equivalents, that is, nested applications of type constructors corresponding to the compositional definition of the interpreter. Each step of the execution would descend in the value of the state before actually performing operations corresponding to the current operator. In the worst case, this descent multiplies the time complexity of executing a program by the size of the program. For instance, on the following family of programs, the interpreter would have time complexity $\Theta(n \times m)$ instead of $\Theta(n + m)$:

```
{ … { for (i = 0; i < n; ++i) { skip; } }; skip … }; skip
                                          m skips
```

To avoid this inefficiency, our library defines interpreters which operate over a state which focuses on the subpart currently executed.

We explain the key ideas of our library of interpreters using an idealized version, reproduced in figure 5.11. For reasons we will detail, this idealized version is rejected by Rocq. To encode this implementation, we had to defunctionalize it [88]. This principled transformation yields an implementation which behaves in the same way as the idealized version, both during execution and proofs. However, it requires fixing the set of primitive operators which builds interpreters.

The states of our interpreters are trees, defined by `Interp.tree` in figure 5.11. This datatype is similar to Interaction Trees [102]. It represents the possible future execution of the system. Each subtree represents a state of the system. The root node of a subtree describes the kind of state represented. Its children are the states which can be reached by performing a transition. The transitions are defined by the function `Interp.istep`, which simply returns the root of the tree of the source state.

The instances of the operators of `SMonad` for interpreters thus build some `Interp.tree`s. We use continuation passing style to ensure that this construction remains efficient when those operators are nested. All our definitions are parameterized by the external events `E_ext` and return type `A_ext` of the full interpreter. We define a family of types `Interp.t` of components of interpreters. This family is indexed by the events `E` and return type `A` of the components. A component is a function which produces a tree describing executions starting at the beginning of the component. This function takes as argument continuations `k_E` and `k_A` describing the execution trees after an event is emitted or a value is returned. The initial state `Interp.init` is the execution tree of the whole program. It is built from an `Interp.t E_ext A_ext` using the continuations `Interp.kE_ext` and `Interp.kR_ext`. Those continuations add single tree nodes performing the expected interactions with the external environment. The implementations of the operators `Interp.ret`, `Interp.bind` and `Interp.trigger` purely perform combinations of their arguments. They do not depend on the definition of `Interp.tree`. For instance, `Interp.bind` passes its right-hand side as continuation to its left-hand side. The operator `Interp.any` takes as argument a τ-event `e` and emits it. It thus represents a non-deterministic choice "`any A`" of a value with the answer type `A` of `e`. The definitions of `Interp.iter` and `Interp.mrec` are more involved. Indeed, those operators can introduce divergence. Silent transitions to `Tau` states are used to ensure the termination of individual steps. We may thus imagine defining a fixpoint operator `Interp.tfix` on functions producing trees `Interp.tree`. Using such operator, we define:

`Interp.iter` by applying `Interp.tfix` to continuations taking as argument the state `x : A` of the iteration. The functional defining the fixpoint applies the body `f` of `Interp.iter` to the current iteration state `x`. In the case where the result `r` of `f` continues the iteration, the continuation given to `f` is defined using the continuation `kA` provided by `Interp.tfix`.

`Interp.mrec` by applying `Interp.tfix` to (uncurried) functions taking as argument an internal call `d : D A` and a continuation `kA : Interp.kR_t A`. The functional defining the fixpoint applies the interpreter `f d` corresponding to the call `d` to `kA`. In the continuation for events, recursive calls are implemented thanks to the function `kD` provided by `Interp.tfix`.

TODO (rocq)

Rocq:
- Type d'évènements au lieu de `AE`
- `Op -> ETS` ?
- `interp -> exec`
- `s -> state`

Parameters

`TE : EventFam.t`	non-deterministic choices
`E_ext : EventFam.t`	external events
`A_ext : Type`	final return type

States: execution trees

```
Interp.tree := Interp.Step (sub : State.F (TE + E_ext) Interp.tree A_ext)

Interp.istep : Interp.tree → State.F (TE + E_ext) Interp.tree A_ext :=
  λ(Interp.Step sub). sub

Interp.tfix [A : Type] (f : (A → Interp.tree) → (A → Interp.tree)) : A → Interp.tree :=
  λx. Interp.Step (Tau (f (Interp.tfix f) x))
```

Components of interpreters

```
Interp.kR_t (A : Type) := A → Interp.tree

Interp.kE_t (E : EventFam.t) := ∀[A] (e : E A) (kA : Interp.kR_t A). Interp.tree

Interp.t (E : EventFam.t) (A : Type) :=
  ∀(kE : Interp.kE_t E) (kA : Interp.kR_t A). Interp.tree
```

Initial state: execution tree of the whole program

```
Interp.kE_ext : Interp.kE_t E_ext := λe k. Interp.Step (Vis (inr e) k)

Interp.kR_ext : Interp.kR_t A_ext := λx. Interp.Step (Ret x)

Interp.init (u : Interp.t E_ext A_ext) : Interp.tree := u Interp.kE_ext Interp.kR_ext
```

Operators (subset)

```
Interp.ret [E A] (x : A) : Interp.t E A :=
  λ_ kA. kA x

Interp.bind [E A B] (u : Interp.t E A) (k : A → Interp.t E B) : Interp.t E B :=
  λkE kB. u kE (λx. k x kE kB)

Interp.trigger [E A] (e : E A) : Interp.t E A :=
  λkE kA. kE e kA

Interp.any [E A] (e : TE A) : Interp.t E A :=
  λ_ kA. Interp.Step (Vis (inl e) kA)

Interp.iter [E A B] (f : A → Interp.t E (A + B)) (xini : A) : Interp.t E B :=
  λkE kB. Interp.tfix (λkA x. f x kE (λr : A + B. { kA x' if r = inl x'
                                                  { kB y  if r = inr y )) xini

Interp.mrec [D E] (f : ∀A (d : D A). Interp.t (D + E) A)
  [Aini] (dini : D Aini) : Interp.t E Aini :=
  λkE kAini. Interp.tfix
    (λkD (d, kA). f d (λde kB. { kD1(d ; kB) if de = inl d
                               { kE e kB     if de = inr e ) kA)
    (dini, kAini)
```

Figure 5.11: Idealized version of the implementation of interpreters [SMonad.Model.LTS.Interp:362]

```
Interp.tree := Interp.Step (sub : State.F (TE + E_ext) Interp.tree A_ext)
             | Interp.Iter [E] (kE : Interp.kE_t E)
                 [A B] (f : A → Interp.t E (A + B)) (x : A) (kB : Interp.kR_t B)
             | Interp.MRec [E D] (kE : Interp.kE_t E)
                 (f : ∀[A] (d : D A). Interp.t (D + E) A) [A] (d : D A) (kA : Interp.kR_t A)

Interp.istep : Interp.tree → State.F (TE + E_ext) Interp.tree A_ext
Interp.istep (Interp.Step sub) := sub
Interp.istep (Interp.Iter kE f x kB) :=
  Tau (f x kE (λr. { Interp.Iter kE f x' kB  if r = inl x'
                   { kB y                    if r = inr y ))
Interp.istep (Interp.MRec kE f d kA) :=
  Tau (f d (λde kB. { Interp.MRec kE f d' kB  if de = inl d'
                    { kE e kB                 if de = inr e ) kA)
```

Figure 5.12: Encoding of `Interp.tfix` in Rocq [SMonad.Model.LTS.Interp:462]

However, the definition of `Interp.tfix` given in figure 5.11 contains a recursive occurrence of `Interp.tfix` and is thus rightfully rejected by Rocq. Indeed, we defined `Interp.tfix` to generate infinitely deep trees. One could hope to fix this issue by defining `Interp.tree` as a coinductive type and `Interp.tfix` as a cofixpoint. However, the definition of `Interp.tfix` would be rejected by the guardedness check of Rocq because the recursive call to `Interp.tfix` appears as argument of the abstract function `f`.

Instead, we keep a definition of `Interp.tree` as an inductive type. To encode the definitions of figure 5.11, we follow the ideas of the *defunctionalization* transformation [88]. Classically, this transformation is used to remove the definition of functions with closures. In our case, we wish to replace the definitions of elements of `Interp.tree` with `Interp.tfix`. We thus collect our two uses of `Interp.tfix` (for `Interp.iter` and `Interp.mrec`) and add them as cases in `Interp.tree`. Figure 5.12 reproduces the definition of `Interp.tree` actually used in Rocq. The new constructors take as arguments the current state of the original `Interp.tfix` (for instance `x` in `Interp.Iter`) and the free variables of the functional taken as argument (for instance `kE`, `f` and `kB` in `Interp.Iter`). We handle those additional cases in `Interp.istep`, following the intended semantics of `Interp.tfix`. With this transformation alone, the new definition of `Interp.tree` does not satisfy the positivity condition of Rocq. Indeed, `Interp.tree` appears in the type of the argument `kA` in `Interp.kE_t`. We solve this issue by also defunctionalizing `Interp.kE_t` and `Interp.t` (not represented in figure 5.12). The drawback of this encoding is that it can only be performed once we know all the way elements of `Interp.kE_t` and `Interp.t` are built, as well as all the uses of `Interp.tfix`. However, the encoded version behaves exactly like the idealized version of figure 5.11. In the following we thus elide details related to the encoding and pretend that the idealized version is used.

We thus have defined a library of operators which build executable interpreters. We now prove that interpreters built using those operators have the same behaviors as the semantics built using the equivalent operators of LTSs. To express this property, we define a function `Interp.to_lts : Interp.t → LTS.t` which converts ETSs into LTSs. This conversion only changes the internal steps: when the ETS steps to a state `Vis (inl e) k` emitting a τ-event `e`, the corresponding LTS instead performs non-deterministic τ-steps to the states `Tau (k x)`, for all possible answers `x` to `e`. We then express simultaneously the correctness and completeness of the interpreters as an equivalence between LTSs.

Lemma 5.3 Let `int : Interp.t E_ext A_ext` be an interpreter built using the operators of our library. Let `lts : LTS.t E_ext A_ext` be the LTS obtained by replacing those operators by the corresponding operators of `SMonad` on `LTS.t`. Then `Interp.to_lts int` $\approx$ `lts`.

Proof We use the framework `FromOperational`. Indeed, the left-hand side of the equivalence is a kind of small-step semantics, directly defined by a state and step function, whereas the right-hand side is a compositional definition in `LTS.t`. Let `G` be the LTS graph corresponding to our interpreters. This graph does not depend on the program interpreted since `Interp.tree` and `Interp.istep` do not.

The core of the proof is an induction on the common structure of `int` and `lts`. The induction property is:

$$\forall \texttt{kE kA}.\ \texttt{FO.p G}\ (\lambda_.\ \texttt{pE kE})\ (\lambda_.\ \texttt{pA kA})\ \texttt{lts}\ (\texttt{Tau}\ (\texttt{int kE kA}))$$

where `pA` and `pE` are defined by:

$$\frac{}{\texttt{pA kA x}\ (\texttt{Tau}\ (\texttt{kA x}))} \qquad \frac{}{\texttt{pE kE e}\ (\texttt{Tau}\ (\texttt{kE e kA}))\ (\texttt{pA kA})}$$

We have a proof obligation for each operator. We solve them using the lemmas of figure 5.10 on `FO.p`. For instance:

`Interp.ret x` The goal reduces to:

`FO.p G` $(\lambda_.$ `pE kE`$)$ $(\lambda_.$ `pA kA`$)$ `(ret x)` `(Tau (kA x))`

we apply `FO.ret` and conclude using the definition of `pA`.

`Interp.bind int_u int_k` The goal reduces to:

`FO.p G` $(\lambda_.$ `pE kE`$)$ $(\lambda_.$ `pA kB`$)$
`(bind lts_u lts_k)` `(Tau (int_u kE` $(\lambda$`x. int_k x kE kB)))`

We apply `FO.bind` where we relate a result `x` of `lts_u` to the state `Tau (int_k x kE kB)`. We prove the two premises of `FO.bind` using the induction hypotheses between `int_u` and `lts_u` and between `int_k` and `lts_k`.

`Interp.iter int_f xini` The goal reduces to:

`FO.p G` $(\lambda_.$ `pE kE`$)$ $(\lambda_.$ `pA kB`$)$
`(iter lts_f xini)` `(Tau (Interp.iter int_f xini kE kB))`

We apply `FO.iter`. As invariant, we relate a state `x` of `iter` with the state `Tau (Interp.iter int_f x kE kB)` of the interpreter. At each iteration, we start by executing, with `FO.tau_opr`, the τ-step of the interpreter at the beginning of `Interp.tfix`. This unguards the invariant added by `FO.iter`. We conclude using `FO.mono` and the induction hypothesis between `int_f` and `lts_f`.

□

5.5.4 Example: a denotational semantics for Cminor

Cminor [69, §4] is a low-level imperative language used as source by the back-end of CompCert. We used Cminor as an example to demonstrate the possibility of connecting our definitions with an existing development based on small-step semantics. The functions of Cminor are defined using an inductive type of statements. In this presentation, we focus on the simplified subset presented in figure 5.13. Our Rocq development supports the whole language, including the memory and unstructured control flows with `goto`s. This type of statement enables structured representation of the control flow. In particular, statements may contain loops defined with `Sloop`. Contrary to general C-like `while` loops, a `Sloop` does not have an associated Boolean guard. Loops thus only terminate if their body has an exceptional result. For the subset described in this manuscript, such exceptional result is either:

- A function return, using `Sreturn`.
- A jump to the end of the $(\texttt{n}+1)^{\text{th}}$ enclosing `Sblock`, using `Sexit n`.

```
stmt ::= Sskip                                                    no-op
       | Sassign (lhs : var) (rhs : expr)                         lhs = rhs
       | Scall (lhs : var) (f : expr) (args : list expr)          lhs = f(args)
       | Sbuiltin (lhs : var) (ef : external_function) (args : list expr)  lhs = ef(args)
       | Sseq (lhs rhs : stmt)                                    lhs; rhs
       | Sifthenelse (c : expr) (thn els : stmt)                  if(c) thn else els
       | Sloop (body : stmt)                                      while(true) body
       | Sblock (body : stmt)                                     {body}
       | Sexit (n : ℕ)                                            exits n blocks
       | Sreturn (res : expr)                                     return res
```

Figure 5.13: Syntax of a simplified subset of the statements of Cminor [compcert.backend.Cminor:140]

```
mk_call (f : function) (args : list val) : CallE val

outcome ::= Out_normal (e : env)          continue in sequence
          | Out_exit (n : ℕ) (e : env)    terminate n + 1 enclosing blocks
          | Out_return (res : val)        terminate the enclosing function

⟦a : expr⟧_expr (e : env) : LTS.t ∅ val

⟦l : list expr⟧_exprs (e : env) : LTS.t ∅ (list val)

⟦s : stmt⟧_stmt (p : program) (e : env) : LTS.t (CallE + Ext) outcome

⟦Sskip⟧_stmt p e := ret (Out_normal e)

⟦Sassign lhs rhs⟧_stmt p e := v ← lift_ev_emp (⟦rhs⟧_expr e); ret (Out_normal [e, lhs ↦ v])

⟦Scall lhs f args⟧_stmt p e := (p_f ← lift_ev_emp (⟦f⟧_expr e);
                                vargs ← lift_ev_emp (⟦args⟧_exprs e);
                                fc ← find_funct p p_f;
                                v ← trigger (inl (mk_call fc vargs));
                                ret (Out_normal [e, lhs ↦ v]))

⟦Sbuiltin lhs ef args⟧_stmt p e := (vargs ← lift_ev_emp (⟦args⟧_exprs e);
                                    v ← map_ev Ev.inr (⟦ef⟧_extcall vargs);
                                    ret (Out_normal [e, lhs ↦ v]))

⟦Sseq lhs rhs⟧_stmt p e := o ← ⟦lhs⟧_stmt p e; { ⟦rhs⟧_stmt p e'  if o = Out_normal e'
                                               { ret o             otherwise

⟦Sifthenelse c thn els⟧_stmt p e := v ← ⟦c⟧_expr e; { ⟦els⟧_stmt p e  if v = 0
                                                    { ⟦thn⟧_stmt p e  otherwise

⟦Sloop body⟧_stmt p e :=
  iter (λe0. o ← ⟦body⟧_stmt p e0; { ret (Continue e1)  if o = Out_normal e1
                                   { ret (Break o)      otherwise) e

⟦Sblock body⟧_stmt p e := o ← ⟦body⟧_stmt p e; ret { Out_normal e'      if o = Out_exit 0 e'
                                                   { Out_exit n e'      if o = Out_exit (n + 1) e'
                                                   { o                  otherwise

⟦Sexit n⟧_stmt p e := ret (Out_exit n e)

⟦Sreturn res⟧_stmt p e := v ← lift_ev_emp (⟦res⟧_expr e); ret (Out_return v), )
```

Figure 5.14: Our denotational semantics of statements of Cminor [SMonadExamplesOpt.Cminor:72]

Denotational semantics We give a denotational semantics to Cminor. We use a domain based on `LTS.t`. Figure 5.14 describes our semantics $[\![.]\!]_{\text{stmt}}$ of the statements. It depends on auxiliary definitions not reproduced, in particular a denotational semantics $[\![.]\!]_{\text{expr}}$ of expressions. The semantics of statements take as argument the current local environment `e : env` of the enclosing function, which maps variables (`var`) to values (`val`). Statements emit two kinds of events: internal calls `CallE` and external events `Ext`. In the former, call events directly contain the syntactic definition `f : function` of the callee. We could have instead used an identifier (i.e. a function pointer) like we did for MiniBTL, but this presentation is closer to the original semantics of Cminor. A drawback of passing function definitions in events is that $[\![$`Scall _ _ _`$]\!]_{\text{stmt}}$ looks up the declaration of the callee from its function pointer `p_f` (using `find_funct`). It thus takes as argument the program (`p : program`) in which the instruction is evaluated[14]. In order to handle exceptional results, the semantics of statements returns an `outcome`. An `outcome` represents either a normal result (`Out_normal`), a jump to the exit of an enclosing block (`Out_exit`) or a function return (`Out_return`). In the first two cases, it contains the new local environment of the function. This notion of "outcome" was already used in CompCert to define a big-step semantics for a subset of Cminor [69, §4.2.3].
From the semantics of instructions, we define the semantics of functions. We handle using `mrec` the mutual recursion between functions and obtain the semantics of programs. Its signature is:

$$[\![\texttt{p : program}]\!]_{\text{prog}} : \texttt{LTS.t Ext int}$$

It only emits external events and its result is an integer return code. Some of the definitions used for our semantics of Cminor, such as the events `CallE`, could be used for other languages of CompCert.

Interfacing with the small-step semantics of CompCert We wish to prove an equivalence between our denotational semantics and the original semantics of Cminor. This semantics is defined in a small-step fashion, using a formalism defined by CompCert and common to all its languages [compcert.common.Smallstep]. This formalization of transition systems differs from our `LTS.t` on several points:

- The initial and final states of small-step semantics are described by predicates. Our LTSs have a single initial state and special `Ret` states.
- UBs are stuck states in small-step semantics (that is, non-final states without outgoing transitions), whereas our LTSs have a special `Err` state.
- Events are represented in CompCert with a type `compcert.event` [compcert.common.Events:64]. Inhabitants of `compcert.event` contain both the arguments and the answer of the event. Moreover, any state can emit an event. In our LTSs the `Vis` states separate the part of the event emitted by the system (inhabiting the family `Ext` in our semantics of Cminor) from the answer it receives from the environment.

We defined a function `CompCertLTS.lts` which wraps small-step semantics of CompCert into our interface for LTSs:

$$\texttt{CompCertLTS.lts} : \texttt{Smallstep.semantics} \to \texttt{LTS.t Ext int}$$

The internal states (`CompCertLTS.lts L`)`.istate` include the states of the small-step semantics `L`, as well as auxiliary states used to encode the transitions of `L`. For instance, we add an unique initial state `Sinit`, with non-deterministic τ-transitions to the initial states of `L`. We also add transitions to `Ret` (from final states) and `Err` (from stuck states). The function `CompCertLTS.lts` preserves the behaviors of the system. Formally, we defined an injection `beh_to_lts` from the formalization of behaviors of CompCert into our formalization on `LTS.t` in the style of CompCert (section 5.4.2). This injection

[14] In Rocq, the semantics actually takes as argument the "global environment" of the program, and the semantics of expressions also depends on it.

```
cont  ::= Kstop                                                    end of the program
        | Kseq (s : stmt) (k : cont)                               s followed by k
        | Kblock (k : cont)                                        block exit followed by k
        | Kcall (lhs : var) (caller : function) (e : env) (k : cont)  end of function

state ::= State (f : function) (s : stmt) (k : cont) (e : env)     before s;k
        | CallState (f : function) (args : list val) (k : cont)    beginning of a call
        | ReturnState (v : val) (k : cont)                         end of a call
```

$$\frac{}{\texttt{step (State f Sskip (Kseq s k) e) [] (State f s k e)}}\ \texttt{step_skip_seq}$$

$$\frac{\texttt{eval_expr e rhs v}}{\texttt{step (State f (Sassign lhs rhs) k e) [] (State f Sskip k [e, lhs} \mapsto \texttt{v])}}\ \texttt{step_assign}$$

$$\frac{}{\texttt{step (State f (Sseq } s_0\ s_1\texttt{) k e) [] (State f } s_0\ \texttt{(Kseq } s_1\ \texttt{k) e)}}\ \texttt{step_seq}$$

Figure 5.15: Simplified extract of the standard small-step semantics of Cminor

accounts for the absence of the case `Stop_event` in the definition of CompCert. We then prove an equivalence between sets of behaviors:

Lemma 5.4 Let `L` be a small-step semantics in the style of CompCert. If individual steps of `L` emit at most one event[15], then for any behavior `beh` in the formalization of CompCert:

`program_behaves L beh` $\leftrightarrow$ `lts_behaves_cases (CompCertLTS.lts L) (beh_to_lts beh)`

Equivalence of the semantics For expressions, the original semantics of Cminor is defined in big-step fashion by a predicate `eval_expr : env → expr → val → Prop`. We prove the equivalence with our monadic semantics $[\![.]\!]_{\text{expr}}$ using our framework `FromEval`. However, most of the semantics of the language is given by a single small-step operational semantics. This includes the semantics of statements. Figure 5.15 presents a simplified subset, the full semantics is described by Leroy [69, §4.2] and is formalized in CompCert [compcert.backend.Cminor:181]. This monolithic semantics uses continuations `cont` to encode both the call stack and the position of the execution within the functions currently executed. The different kinds of `state`s of Cminor thus describe both the next element to execute and its continuation: Like in most languages of CompCert [69, §3.6], states `CallState` and `ReturnState` are used to handle context switches between callee and caller during function calls. A function call starts by a transition to a `CallState` and ends in a `ReturnState`. The transitions of the operational semantics are defined using a monolithic inductive predicate `step` (`s₀ : state`) (`t : list compcert.event`) (`s₁ : state`) : `Prop` Figure 5.15 only reproduces three of the rules relevant to our subset. As illustrated by `step_assign`, instructions which terminate normally reduce to `Sskip`. Several rules for states with shape `State _ Sskip _ _` then examine the continuation to continue the execution (we only reproduced the case of `Kseq` with `step_skip_seq`). The original semantics of Cminor, `Cminor.semantics`, is defined by `state`, `step` and predicates describing the initial and final states. After wrapping `Cminor.semantics` into a `LTS.t`, we compare it to our denotational semantics using the equivalence relation on our LTSs.

Lemma 5.5 For any Cminor program `p`:

`CompCertLTS.lts (Cminor.semantics p)` $\approx [\![\texttt{p}]\!]_{\text{prog}}$

Idea of the proof We use our framework `FromOperational`. The core of the proof is the equivalence between our denotational semantics of instructions and the intra-procedural part of the operational semantics. The induction property for a statement `s` is the following:

[15] a standard well-formedness property of the semantics of CompCert, called `single_events`

$$\forall e\ k.\ \texttt{FO.p}\ \texttt{G}\ (\lambda_.\ \texttt{evt_inv})\ (\lambda_.\ \texttt{ret_inv}\ k)\ (\llbracket s \rrbracket_{\text{stmt}}\ p\ e)\ (\texttt{Tau}\ (\texttt{State}\ f\ s\ k\ e))$$

where G is the graph of the operational semantics, f the function containing the statement and evt_inv and ret_inv are defined by:

$$\frac{\texttt{is_call_cont}\ k}{\texttt{evt_inv}\ (\texttt{inl}\ (\texttt{mk_call}\ \texttt{callee}\ \texttt{args}))\ (\texttt{Tau}\ (\texttt{CallState}\ \texttt{callee}\ \texttt{args}\ k))\ (\texttt{ret_inv_return}\ k)}$$

$$\frac{}{\texttt{evt_inv}\ (\texttt{inr}\ e)\ (\texttt{Vis}\ e\ sk)\ (\lambda x\ s_1.\ s_1 = \texttt{sk}\ x)}$$

$$\frac{}{\texttt{ret_inv}\ k\ (\texttt{Out_normal}\ e)\ (\texttt{Tau}\ (\texttt{State}\ f\ \texttt{Sskip}\ k\ e))}$$

$$\frac{}{\texttt{ret_inv}\ k\ (\texttt{Out_exit}\ e)\ (\texttt{Tau}\ (\texttt{State}\ f\ (\texttt{Sexit}\ n)\ k\ e))}$$

$$\frac{\texttt{ret_inv_return}\ (\texttt{call_cont}\ k)\ v\ s_1}{\texttt{ret_inv}\ k\ (\texttt{Out_return}\ v)\ s_1} \qquad \frac{}{\texttt{ret_inv_return}\ k\ v\ (\texttt{Tau}\ (\texttt{ReturnState}\ v\ k))}$$

and:

- `is_call_cont k` asserts that the continuation `k` starts with `Kstop` or `Kcall`.
- `call_cont` removes the head of the continuation `k` until it finds a `Kstop` or a `Kcall`.

Those invariants reflect the structure of the operational semantics of Cminor. For instance:

- For `Out_normal`, `ret_inv` reflects that statements reduce to `Sskip` when they terminate normally.
- For internal calls, `evt_inv` reflects that they begin in a `CallState` and end in a `ReturnState` with the same continuation.

Those invariants are similar to those used by CompCert to prove the soundness of the alternative big-step semantics of Cminor. In particular, our `ret_inv` is similar to `outcome_state_match`.
Each case of the induction is solved using the lemmas of figure 5.10 on `FO.p`.

Combining the equivalence between the two semantics, the preservation of behaviors by `CompCertLTS.lts` and the soundness of the simulation on our LTS yields the following corollary:

Lemma 5.6 For any Cminor programs `src` and `trg`, if $\llbracket \texttt{src} \rrbracket_{\text{prog}} \gtrsim \llbracket \texttt{trg} \rrbracket_{\text{prog}}$ then:

$$\forall \texttt{beh}.\ \texttt{program_behaves}\ (\texttt{Cminor.semantics}\ \texttt{trg})\ \texttt{beh} \Rightarrow \exists \texttt{beh}'.\ \texttt{program_behaves}\ (\texttt{Cminor.semantics}\ \texttt{src})\ \texttt{beh}' \wedge \texttt{behavior_improves}\ \texttt{beh}'\ \texttt{beh}$$

That is, a refinement between our denotational semantics implies the preservation of the behaviors in the standard semantics of Cminor. One could thus use `SMonad` to prove the correctness of a transformation, at least for the specification on the behaviors. However, lemma 5.6 does not ensure directly the existence of a simulation between the standard semantics. Such simulation (backward or forward) could maybe be derived using the completeness of simulations with respect to the inclusion of behaviors on deterministic languages [73, theorem 3.11].

5.6 Related works: simulations

Divergence-preserving weak simulations allow the source and target to differ by finite sequences of τ-steps. However, their definitions must rule out "infinite stuttering", that is, situations where the simulation never progresses in one of the two programs. Otherwise, one could introduce or remove infinite sequences of τ steps (i.e. silent divergence). There are several approaches to enforce this restriction:

- The current `forward_simulation` of CompCert (which is close to the "star" simulation described by Leroy [69, definition 12]) is an *explicit normed* simulation (as called by Chappe [27]). That is, each simulation step performs exactly one step in the simulated program $\texttt{P}_1$ (i.e. in the source program

in forward simulations), but can perform zero, one or several steps in the other program P_2. To ensure that some steps are always eventually performed in P_2, the simulation relation is indexed by a measure. This measure strictly decreases when no step is performed in P_2 and is reset to an arbitrary value otherwise. Provided that the order on the measures is well-founded, this rules out infinite sequences of simulation steps without progress in P_2. This approach requires choosing a well-founded order and maintaining a measure in the simulation invariant. However, most simulations of CompCert always perform at most one step in P_2 during each simulation step. Those "plus" simulations ("lock-step" if P_2 always performs exactly one step) are particular cases of the "star" simulation but enjoy specifications which do not mention the measure.

- In *implicit normed* simulations, each simulation step performs a strict number of steps in both programs. In Rocq, the simulation steps are defined as inductive predicates. This approach avoids maintaining a measure in the simulation invariant. It is used to define `eutt` on Interaction Trees [102, 2.2]. FreeSim [31] combines explicit and implicit norms. One motivation is to avoid synchronous τ-steps. The `sem_ref` simulation defined on our LTSs (section 5.3) uses the particular case of Boolean norms.
- Chappe [27] studied several notions of simulations. He defines a notion of weak simulation which is strictly more liberal than normed simulations. Its definition involves a separate inductive-coinductive predicate to ensure the preservation of divergence. The reasoning principles of implicit normed simulations are recovered as "up-to" lemmas. Those lemmas also sometimes avoid reasoning on the separate predicate.

In all cases, simulations are coinductive proofs, where the steps act as guards. As a consequence, it is not possible to reason modulo weak equivalence (i.e., up to finite sequences of τ-steps) during those proofs [94; 104].

6

A parameterized representation in functional SSA

Our compilation pipeline, described in section 3.3, involves two new IRs: Frag IR and Flat IR. They have different memory models. Frag IR stores memory fragments in the variables of the program, while Flat IR features a standard global memory. As a consequence, we will define their semantics in different domains. However, the two IRs share the same syntactic structure. In order to factorize our development, we defined a *parameterized IR*. Syntactically, this parameterized IR defines the structure of programs but takes as parameters the sets of the types and operators. Frag IR and Flat IR are both obtained as instances of the parameterized IR, with different sets of operators. Large parts of their meta-theory are also defined generically on the parameterized IR. Thanks to the axiomatic interface of the `SMonad` library, the choice of the semantic domain is left to the instances.

The parameterized IR uses a functional flavor of static single-assignment form (SSA). This structure enables the use of dependent types, provides reasoning principles on programs and enables compositional definitions of denotational semantics.

Section 6.1 recalls the core properties and benefits of SSA and its implementation with the functional approach. We then present informally our parameterized IR in section 6.2. In particular, we introduce the grammar of programs that we use in most of this document. In Rocq however, the formalization of our IRs involves two different syntaxes, described in section 6.3. A first syntax is used to reason on programs while the second enjoys a type-safe and efficient extraction to OCaml. We define a semantics (section 6.4) and a typing judgment (section 6.5) on the parameterized IR.

6.1 Static Single Assignment

Static single-assignment (SSA) [12; 34; 85] is a property enforced by some IRs. There are multiple approaches to enforce it. Section 6.1.1 describes the core properties of SSA and our motivations to enforce it. Section 6.1.2 recalls the standard approach to implement SSA. Our IRs instead use a functional flavor of SSA. We describe this approach in Section 6.1.3 and compare it to the standard approach.

6.1.1 Common properties of SSA

Programs store temporary data in named variables. In imperative languages, the value stored in a variable is changed by assignments. The core of SSA is a *unicity* property: each variable must be assigned exactly once in the program text. As a counter-example, the following C fragment does not satisfy SSA because the variable `x` is assigned twice.

```
x = 0;
y = x + 1;
x = 1;
```

Because of those two assignments, the value of `x` depends on the program point where it is accessed. In particular, the value of `x` used to compute `y` (`0`) is not the same as the value of `x` at the end of the sequence (`1`). Hence, at the end of the sequence, `y` is no longer equal to `x + 1`. Such changes of values may invalidate dependent types: for instance, if, like in Frag IR, the assignment to `y` also returned a variable with type `y = x + 1`, then this type would no longer be inhabited after the second assignment to `x`.

SSA ensures *referential transparency*: values obtained by accessing a variable do not depend on the program point where the access is performed. This property enables the use of dependent types. It has other benefits:

- SSA enables efficient resolution of some data-flow problems with sparse analyses [25; 85]: one only needs to use a single data-flow variable per program variable instead of a data-flow variable per program variable and program point.
- IRs satisfying SSA provide an *equation-lemma* [12, §6]: when reasoning on programs, variable assignments can be seen as equations.

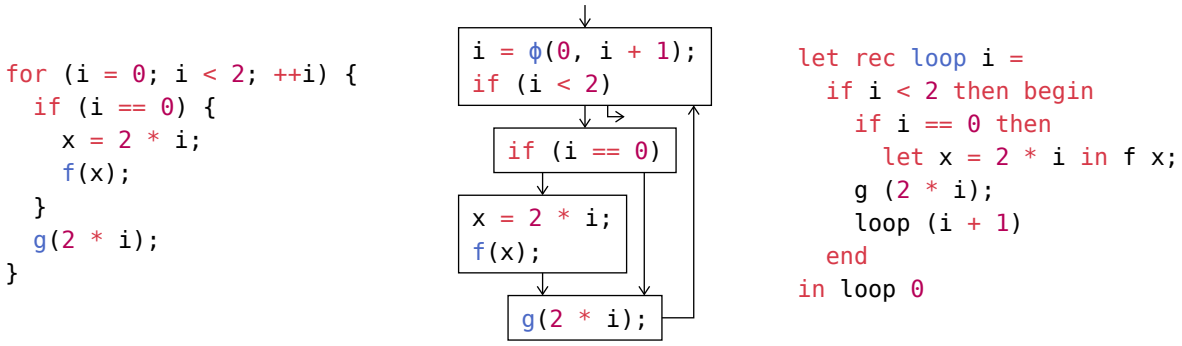
6.1.2 Standard SSA

Programs are not restricted to linear sequences of instructions. Their control-flow involves conditional branchings and loops. Even when SSA is enforced, the value of variables depends on the control flow path followed during the execution. The standard approach to SSA [34] uses some specific ϕ-functions to account for this dependency. With this approach, programs are represented using Control Flow Graphs (CFG) (e.g. figure 6.1b). The ϕ-functions are placed at the beginning of blocks. Their value depends on the CFG edge taken to reach their block. For instance in figure 6.1b, `i` takes the value `0` when the program enters the loop but the value `i+1` when the program loops.

Because of the control-flow, the values of variables can change multiple times during executions. The equations enforced by assignments thus do not remain valid indefinitely. Consider for instance the loop of figure 6.1. Although `x` has a single assignment, it is not equal to `2*i` when calling `g` during the second iteration. In particular, it would be incorrect to replace the argument of `g` with `x`. In fact, reasoning on SSA representations also involves a relation of *dominance* between the program points. A program point p_0 is said to dominate another point p_1 if any syntactic path from the entry to p_1 in the CFG contains p_0. The equation-lemma for a variable x only holds at program points dominated by the definition of x. Standard SSA representations enforce a *strictness* property from which dominance relations can be recovered: any use of a variable must be dominated by its definition.

6.1.3 Functional SSA

Functional SSA is a syntactic structure used by some IRs to enforce SSA [85, chapter 6]. This approach leverages a parallel between the unicity property of SSA and the immutability of variables in functional programming languages. It was popularized by Appel [7] and recently used in MLIR [61]. In functional SSA, variables are defined (e.g. with `let x = 2 * i` in figure 6.1c) instead of being assigned. Control flow is implemented using mutually recursive continuations (e.g. `let rec loop` in figure 6.1c). However, contrary to high-level functional languages where the `let` construct may define both functions and



a: C program b: Standard SSA c: Functional SSA

Figure 6.1: Three representations of a program fragment

TODO (rocq)

- ordre des constructeurs (`FCall`)

Type expression `typ`	::=	`var` | `i32` | `bool` | `true` | `false` | `typ = typ` | `typ ≤ typ` ...
Operator `op`	::=	`add{i64}` | `input{i32}` | `output{bool}` | `Frag.read{i16}` | `Frag.write{i64}` | `lemma{…}` ...
Annotation `annot`	::=	`ghost` | `dead`
List of binders `binders`	::=	$(\texttt{annot}^?\ \texttt{var : typ})^*$
Instruction `instr`	::=	`return (var*)` — values of variables
		$\texttt{annot}^?$ `op<typ*>(var*)` — operation
		`fcall fct(var*)` — function call
		`let binders = instr; instr` — sequence
		`if var var then instr else instr` — conditional
		`kcall knt(var*)` — continuation call
		$\texttt{krec\{kont}^+\texttt{\}(var*)}$ — declaration of continuations
Continuation `kont`	::=	`knt(binders): instr`
Function `func`	::=	`fn fct(binders) : (binders) {instr}`
Program `prog`	::=	$(\texttt{func} \mid \texttt{glob})^*$

Figure 6.2: Grammar of our functional SSA IR. The type expressions (`typ`), operators, annotations are taken from the instantiation for Frag IR. The other categories are defined by our generic IR.

simpler values, continuations are not necessarily first-class values in functional SSA IRs. The formal arguments of the continuations play the same role as the ϕ-functions of standard SSA. Their value depends on the control-flow followed to reach the continuation, and in particular on the point where the continuation was called. The body of each function is defined by an inductive instruction rather than by a CFG. Each variable has lexical scope, corresponding to a subtree of the instruction. The explicit notion of scope under-approximates the implicit notion of dominance in the equivalent CFG. That is, if a variable x is in the scope of an instruction u, then the definition of x dominates u. Hence, the strictness property of SSA is implied by the absence of free variables and the equation lemma for a variable x is valid in the whole scope of x.

6.2 Our functional SSA IR

We describe informally our generic IR. The syntax of our IR is presented in figure 6.2. The generic part of the IR defines the type of instructions (`instr`, including `binders` and `kont`), of functions (`func`) and programs (`prog`). The definition of the other syntactic categories is left to the instances: the symbols

in type expressions (`typ`), the set of operators (`op`), annotations on variables and operators (`annot`) and global declarations (`glob`).

In most of this document, we use a presentation with names for all symbols. We use some informal types `var`, `knt` and `fct` for the names of variables, local continuations and global functions. The representation used in our development is explained in section 6.3, and actually involves De Bruijn indices for variables and continuations.

Types are defined using type expressions `typ`. They can contain variables, which enable dependent types. Subparts of type expressions either evaluate to a value (e.g. `true`) or to a set of values (e.g. `bool` or `b = true`). We only use complete type expressions which evaluate to sets of values (e.g. we avoid declaring a variable with type `true`), but we did not enforce this property in the syntax or in the type system. The instructions of our generic representation may return several values. Their syntax is defined inductively in figure 6.2. The `return` instruction evaluates to the values of the variables given as arguments. It is monadic, that is, it does not always terminate the enclosing function: if it appears in the left-hand side of a sequence, the execution continues with the right-hand side. The evaluation of operators of the IR instance returns their results but can also perform side effects. Operators take as arguments types (syntactically described by type expressions) and values (syntactically described by variables). In the grammar we use to write programs, some operators also take parameters within `{…}`. Those parameters are instance-specific; in figure 6.2 we consider that they are part of the operator `op`. For instance, in Frag IR, we use parameters for operators that are defined for several scalar types, like in `add{i64}`. However, contrary to the type arguments, parameters cannot depend on the variables in the context of the instruction. For this reason, the `assert` operator of Frag IR has a type argument rather than a parameter. Like in functional languages, a `let` construct executes sequentially two instructions with the results of the left-hand side bound in the right-hand side. Functions are called using `fcall`. They return the evaluation of their bodies, which are instructions. We allow functions with multiple results, which is useful to return ghost variables (propositions and memory fragments in Frag IR). Programs are defined by a list of mutually recursive functions, with a distinguished `main`. Programs also include instance-specific global declarations. For instance in Frag IR, we declare the names of the abstract predicates (section 3.1.3).

In the implementation, we use a syntax close to the one described in figure 6.2 for parsing and printing programs. We also use this syntax in most of this document, but take some liberties for the sake of conciseness. For instance, we often omit the types of the variables, omit unused variables (e.g. post-conditions and branch hypotheses) and directly write the value of constants instead of producing them with operators.

Continuation-based control flow Functional SSA [7] implements arbitrary control flow graphs (including irreducible ones like figure 6.3a) using locally defined functions that are called in terminal position. Our representation thus features `krec` blocks, which define local, mutually recursive continuations. Figure 6.3 illustrates their use to encode loops. Each block defines one or several continuations, which take some arguments and are implemented by instructions. The execution of a block starts with its first continuation, with the arguments given to `krec`. We note `first ks` the identifier of the first continuation of a block `ks`. In figure 6.1c we were instead binding the continuations around an instruction using `let  in`. In our syntax this can be achieved with an initial continuation which is never called recursively, for instance `entry` in figure 6.3a. Continuations are not values; `kcall` must be used to jump to them from within the block. Such a terminal call takes arguments corresponding to the signature of the continuation and does not return to the call point. The execution of a `krec` block ends when one of its continuations returns or calls a continuation defined in an outer block. Local continuations act as confluence points of the control flow graph. The arguments given to `kcall` (and to `krec`) play the same role as ϕ-functions in standard SSA representations [34]: the value bound to an

```
krec {
  entry():
    if b
    then kcall k0()
    else kcall k1()
  k0():
    fcall f0();
    kcall k1()
  k1():
    fcall f1();
    kcall k0()
}()
```

a: An infinite loop with two entries

```
let c0 = 0; let c1 = 1; let c2 = 2;
krec {
  loop(i):
    let b0 = u_lt(i, c2);
    if b0 _ then {
      let b1 = u_eq(i, c0);
      if b1 _ then {
        let x = mul(c2, i); fcall f(x)
      } else return ();
      let y  = mul(c2, i); fcall g(y);
      let i1 = add(i, c1); kcall loop(i1)
    } else return ()
}(c0)
```

b: The program of figure 6.1

Figure 6.3: Loops encoded in our functional SSA IR using `krec` [compiler/test/src/example/two_entries.flat] [compiler/test/src/example/loop_not_dom.flat]

argument depends on the point where the call is performed. It is crucial to be able to define continuations locally with `krec` blocks and not only at the top-level of functions. This enables continuations to access variables defined in outer scopes, such as `c2` in figure 6.3b. Otherwise, continuations would take as arguments all live variables, which would be the equivalent of adding a ϕ-function for every variable at every confluence point. Without syntactic restrictions, there can also be confluence points at the end of `if` and `krec`. If desirable, those confluence points could be turned into `kcall`.

6.3 Two syntactic representations in Rocq

We defined in Rocq two syntaxes of our parameterized IR. A Dependently-Typed Syntax (DTS) enforces the well-formedness of programs to simplify definitions and reasoning (section 6.3.1). We relax it into an Algebraic Syntax (AS) based on algebraic datatypes and which is extracted to OCaml (section 6.3.2). This AS enables interfacing OCaml and Rocq code without jeopardizing the properties proved on the latter. It also uses more efficient structures than the DTS. The two syntaxes are formally related (section 6.3.3). Definitions on the DTS are thus used to specify implementations on the AS. We discuss the benefits and drawbacks of using two syntaxes in section 6.3.4.

6.3.1 Well-formed, Dependently-Typed Syntax (DTS)

We use the dependent types of Rocq to define a representation of the syntax of well-formed programs. More precisely, this DTS rules out programs with arity mismatches or unbound references. This approach avoids considering those cases (using default values or partial functions) in definitions and reasoning. We reproduced our definitions in figure 6.4a.

The syntax is parameterized by a set of type symbols `Typ.sym_s` and a family of operators `Op.sym_s`. The latter is indexed by the arity of the operators, described by a record `Op.Arity.s` containing the number of type arguments (`argT`), of value arguments (`argV`) and of results (`res`). We implemented this parameterization using some typeclasses `Typ.S` and `Op.S`. The whole development is thus parameterized by instances of those typeclasses. We implicitly fix some instances during definitions and proofs on the parameterized IR. In Rocq, we use the `Section` mechanism.

The types representing the different syntactic categories are indexed by arities and the references in scope. In particular our type family `Instr.s` of instructions is indexed by the number `r` : $\mathbb{N}$ of values returned and by an environment `E` : `Env.s` which specifies the sets of symbols in scope. Our language uses three kinds of references: variables, local continuations and top-level functions. The scopes corresponding to the different kinds of references are stored separately, using the fields of `Env.s`. A scope is represented by the type (or type family) of the declared symbols. Following the approach of well-scoped De Bruijn indices [2], environments are extended with new variables (`Env.app_var`) or

local continuations (`Env.app_knt`) using the disjoint union "+" of types. In particular `V + Fin.t n` adds `n` variables to a scope `V`, using the finite type `Fin.t n` with `n` elements.

TODO (rocq)

- Spécialiser `Typ.t` pour `Var.t`

```
Class Typ.S : Type := Typ.sym_s : Type.
Class Op.S  : Type := Op.sym_s  : Op.Arity.s -> Type.


Definition Var.s := Type.
Definition Knt.s := Knt.Arity.s → Type.
Definition Fct.s := Fct.Arity.s → Type.
Record Env.s := {var : Var.s; knt : Knt.s; fct : Fct.s}.
Definition Env.app_var (E : Env.s) (V : Var.s): Env.s :=
  {| var := var E + V; knt := knt E; fct := fct E |}.

Inductive Typ.s (V : Var.s) : Type :=
  | Typ.Var (x:V)
  | Typ.Op (o:Typ.sym_s) (xs:list (Typ.s V)).
Definition Binders.s (n : ℕ) (V : Var.s) : Type :=
  Vector.t (Typ.s (V + Fin.t n)) n.
Inductive Instr.s (E : Env.s) : ∀r : ℕ, Type :=
  | Op [a] (op : Op.sym_s a)
      (argT : Vector.t (Typ.s E.var) a.argT)
      (argV : Vector.t        E.var  a.argV)
    : Instr.s E a.res
  | Bind [m r] (tys : Binders.s m E.var)
      (lhs : Instr.s              E            m)
      (rhs : Instr.s (Env.app_var E (Fin.t m)) r)
    : Instr.s E r
  | KRec [r] (K : Knt.s)
      (ks: ∀[a] (k : K a), Kont.s (Env.app_knt E K) a r)
      [a0] (kini : K a0) (arg : Vector.t E.var a0)
    : Instr.s E r …
Record Kont.s (E : Env.s) (a r : ℕ) : Type := {
  arg  : Binders.s a E.var;
  body : Instr.s (Env.app_var E (Fin.t a)) r;
}.
Record Func.s (F : Fct.s) (a : Fct.Arity.s) : Type := {
  arg  : Vector.t (Fin.t a.arg             ) a.arg;
  res  : Vector.t (Fin.t a.arg + Fin.t a.res) a.res;
  body : Instr.s (Env.func_body F a.arg) a.res;
}.
Record Prog.s (a : Fct.Arity.s) := {
  F    : Fct.s;
  func : ∀[a] (f : F a), Func.s F a;
  main : F a;
}.
```

a: Dependently-Typed Syntax (DTS)

```
Class Typ.T := Typ.sym_t : Type.
Class Op.T  := Op.sym_t  : Type.

Record RRef.t := { lvl : N.t; num : ℕ }.
Definition Var.t := RRef.t.
Definition Knt.t := RRef.t.
Definition Fct.t := positive.

Inductive Typ.t : Type :=
  | Typ.Var (x:Var.t)
  | Typ.Op (o:Typ.sym_t) (xs:list Typ.t).
Definition Binders.t : Type :=
  list (An.var × Typ.t).
Inductive Instr.t :=
  | Op (op : Op.sym_t)
       (argT : list Typ.t)
       (argV : list Var.t)

  | Bind (tys : Binders.t)
         (lhs : Instr.t)
         (rhs : Instr.t)

  | KRec
       (ks  : list Kont.t)
       (arg : list Var.t)
    …
Record Kont.t := {
  arg  : Binders.t;
  body : Instr.t;
}.
Record Func.t := {
  arg  : Binders.t;
  res  : Binders.t;
  body : Instr.t;
}.
Record Prog.t := {

  func : PMap.t Func.t;(*Fct.t ⇀ Func.t*)
  main : Fct.t;
}.
```

b: Algebraic Syntax (AS)

Figure 6.4: Mechanizations of the syntax of the parameterized IR in Rocq. We denote objects of the DTS with the letter `s` (e.g. `Prog.s`) and those of the AS with the letter `t` (e.g. `Prog.t`).

All variables are declared with their type. Type expressions with free variables in `V` are represented with `Typ.s V`. Syntactically, a type expression is either a variable or the application of a type symbol (`Typ.sym_s`) to a list of type expressions. For instance, given a variable `x : V` an expression `x = 0` could be represented in `Typ.s V` as:

```
Typ.Op Eq [Typ.Var x; Typ.Op [Const 0]]
```

using some instance specific type operators `Eq` and `Const 0` in `Typ.sym_s`.
A declaration `Binders.s n V` defines the types of `n` new variables in a scope `V`. The scope `V + Fin.t n` allows those types to depend on the new variables, binders `(i:i32, H: 0 ≤ i)` are thus valid. Instructions are defined inductively by `Instr.s` in figure 6.4a, with the same constructors as in figure 6.2. We use `Vector.t A n` (lists of `A`s of length `n`, noted A^n) instead of `list A` to enforce the respect of arities. Local continuations are declared in `KRec` by specifying their family `K` and their definitions with a total dependent function `ks`. The constructor also specifies the initial continuation `kini`, which corresponds to the continuation "`first`" in representations of `krec` with a list. The environment of a local continuation contains the continuation itself, its siblings and its arguments.

A function `Func.s F a` is defined by the types of its arguments and results and by its body. Its scope `Env.func_body` contains the `a.arg` arguments of the function, the functions `F` declared in the program but no local continuations. Finally, a program `Prog.s a` is defined by a family `F` of functions with a distinguished `main` function which has the arity `a` of the program. The total function `func` defines an implementation for each function.

6.3.2 Extracted, Algebraic Syntax (AS)

The extraction mechanism of Rocq is able to generate OCaml types directly from the definitions of the DTS. However, the generated code is not entirely satisfactory for usage within a verified compilation pipeline:

- The type system of OCaml is not expressive enough to translate faithfully the dependent types of Rocq. The extraction procedure thus removes some indexes of dependent types (for instance the size `n` of A^n) and erases all scopes to OCaml's `Obj.t`. As a consequence, the type system of OCaml does not guarantee that syntax built in OCaml satisfies the well-formedness assumptions enforced in Rocq. This is an issue for combining passes extracted from Rocq with code written directly in OCaml (such as parsers or oracles). The use of ill-formed syntax could generate errors, non-termination or even silent miscompilation within parts proven correct in Rocq [40]. However, communication between OCaml and Rocq with algebraic datatypes is safe [40, §2.4].
- The implementation of references with well-scoped De Bruijn indices is rather inefficient. Because we use sum types to extend scopes, a reference to a variable x is implemented with nested constructors `inl` around the reference to x in the scope where it was declared. The number of `inl` constructors is the number of scopes opened since x was declared and can be linear in the size of the enclosing function for programs containing long sequences of instructions.

Instead of extracting the DTS, we defined another syntax in Rocq (figure 6.4b). This AS is better extracted to OCaml. It follows the same structure as the DTS and is parameterized by typeclasses with the implementations of type and operator symbols. We avoid dependent types and use standard `list`s instead of `Vector.t`. Functions (`ks` of `KRec`, `func` of `Prog.s`) are replaced with `list`s or binary trees `PMap.t`. The types of references are independent of the scopes. Functions have global identifiers in `Fct.t`. Variables and continuations are represented with relative references `RRef.t`, which are similar to standard De Bruijn indices [35] but carry an additional field `num` which differentiates variables declared simultaneously. We use binary naturals `N.t` to represent concisely the number `lvl` of scopes opened since the declaration of the referenced element. Finally, we include instance-specific semantically irrelevant annotations, for instance `An.var` on variable declarations. Those annotations

```
IImpl.t (T S : Type) : Type := {f : S → T; g : T → option S; EQV : ∀t s. t = s.f ↔ t.g = ⌊s⌋}
IImpl.fin (n : ℕ) : IImpl.t ℕ (Fin.t n) := {f := Fin.to_nat; ...}

Env.r (E : Env.s) : Type := {
  var : IImpl.t Var.t E.var
  knt : IImpl.t Knt.t {a : Knt.Arity.s & E.knt a}
  fct : IImpl.t Fct.t {a : Fct.Arity.s & E.fct a}
}
Env.app_var_r [E V] (rE : Env.r E) (rV : IImpl.t ℕ V) : Env.r (Env.app_var E V) := ...

Binders.impl [V n] (rV : IImpl.t Var.t V) (t_bs : Binders.t) (s_bs : Binders.s n V) : Prop := ...
```

$$\frac{\texttt{Binders.impl rE.var t_tys s_tys} \quad \texttt{Instr.impl rE t_lhs s_lhs} \quad \texttt{Instr.impl (Env.app_var_r rE (IImpl.fin m)) t_rhs s_rhs}}{\texttt{Instr.impl rE (Bind t_tys t_lhs t_rhs) (Bind s_tys s_lhs s_rhs)}}\ \texttt{ImplBind}$$

Figure 6.5: Implementation relation between the DTS and the AS

are used to guide the compilation passes (e.g. by marking ghost variables) or to transmit data through the compilation pipeline (e.g. names of variables).

6.3.3 Formal implementation relation between the syntaxes

We formalize implementation relations between corresponding DTS and AS types. Those implementation relations are parameterized by implementation relations for the instance-specific type and operator symbols. They are also parameterized by injections `IImpl.t T S` (defined in figure 6.5) from the types (`S`) used to represent references in the DTS to their implementation (`T`) in the AS. For instance, the implementation relation for instructions has signature:

```
Instr.impl [E r] (rE : Env.r E) (t_u : Instr.t) (s_u : Instr.s E r) : Prop
```

where the record `rE : Env.r E` (defined in figure 6.5) specifies the injection of variables, continuations and functions. This implementation relation `Instr.impl` is defined inductively, following the common structure of the instructions `t_u` and `s_u`. The injections in `rE` are extended when new scopes are opened. For instance, in the case of a let-binding of `m` variables (`ImplBind` in figure 6.5), the right-hand side `s_rhs` has a scope of variables $\texttt{E.var} + \texttt{Fin.t m}$. Its injection into `RRef.t` is defined as follows. For variables in the outer scope `E.var`, we compose the injection `rE.var` with a shift by one of `lvl`. For the new variables in `Fin.t m`, we set `lvl` to 0 and use the standard injection of `Fin.t m` into $\mathbb{N}$ for `num`. We derive a relation `Prog.impl` between DTS and AS programs.

Those implementation relations enable specifying passes on the AS using properties defined on the DTS. For each syntactic element in the AS, specifications quantify over corresponding elements in the DTS (universally for inputs, existentially for outputs). Those DTS elements guarantee that the syntactic elements are well-formed. The effect of the passes is then specified on the DTS. For instance, the specification of a total pass $\texttt{transf_prog} : \texttt{Prog.t} \to \texttt{Prog.t}$ which preserves the semantics of programs (defined on the DTS by $[\![\texttt{s_pr} : \texttt{Prog.s a}]\!]_{\text{prog}}\ (\texttt{vargmain} : \texttt{val}^{\texttt{a.arg}}) : \texttt{S} \varnothing\ (\texttt{val}^{\texttt{a.res}})$ in section 6.4) would be:

$$\begin{aligned}
&\forall(\texttt{t_pr_src} : \texttt{Prog.t})\ \texttt{a}\ (\texttt{s_pr_src} : \texttt{Prog.s a}).\\
&\texttt{Prog.impl t_pr_src s_pr_src} \Rightarrow\\
&\exists\texttt{s_pr_trg} : \texttt{Prog.s a}.\\
&\texttt{Prog.impl}\ (\texttt{transf_prog t_pr_src})\ \texttt{s_pr_trg} \wedge\\
&\forall\texttt{vargmain} : \texttt{val}^{\texttt{a.arg}}.\ [\![\texttt{s_pr_src}]\!]_{\text{prog}}\ \texttt{vargmain} \gtrsim [\![\texttt{s_pr_trg}]\!]_{\text{prog}}\ \texttt{vargmain}
\end{aligned}$$

This specification reads as follows. We consider a source AS program `t_pr_src`. We assume that this program implements a source DTS program `s_pr_src` with arity `a`. This assumption implies

that the source program is well-formed. The specification ensures that the target AS program `transf_prog t_pr_src` implements some target DTS program `s_pr_trg` which has the same arity `a` as the source. This implies that the target program is well-formed. Finally, semantic preservation is stated as a refinement between the semantics of the DTS programs `s_pr_src` and `s_pr_trg`.

6.3.4 Comparison with the use of a single syntax

Instead of formalizing two syntaxes, an alternative approach could have been to only use the AS. This would require defining the meta-theory of our IRs (semantics and typing judgments) directly on the AS. In the semantics, UBs could be used for ill-formed programs to recover well-formedness assumptions during proofs of semantic preservation. Passes can also check defensively the well-formedness of programs instead of assuming that they are well-formed.

Our DTS acts as an intermediate layer between the AS and the semantics. It abstracts away some parts of the implementation but keeps a static structure on which we define the typing judgment. By ruling out ill-formed programs, it enables the use of total functions in the meta-theory. Moreover, it separates proofs on the structures used for efficiency in the implementation (e.g. binary naturals in `RRef.t`) from the proofs that the syntactic transformations preserve semantics and typing. We also use the DTS to propagate well-formedness properties that cannot be checked automatically. Indeed, in section 8.1.3.4, we will restrict the set of DTS operators of Frag IR to enforce the invariant that programs only contain sound lemmas. Finally, when designing some passes, we first specified their syntactic effects as a relation on the DTS, before implementing them on the AS. During the first step, the dependent types of Rocq were often useful to avoid mistakes, in particular when maintaining De Bruijn indices.

However, a drawback of our approach is that it requires defining two syntaxes. Moreover, we can only reason on well-formed programs. This means that we need to prove the well-formedness of the programs in our IRs (source programs written directly in our IRs, or programs generated by passes) before reasoning on their semantics.
We implemented a type-checker (section 7.4), which guarantees among other things that its input program is well-formed, under some side-conditions (for Frag IR, the soundness of the lemmas). This well-formedness is stated as the existence of a corresponding DTS. However, running this type-checker on generated program does not allow us to avoid well-formedness proofs. Indeed, it does not give any guarantee on the existentially quantified DTS, except that it is implemented by the AS. We thus cannot prove properties on its semantics, except maybe by reasoning by inversion on its relation with the AS. During our proofs of passes, we instead more directly build a DTS implemented by the AS. We thus have access to its definition and can reason on its semantics. In Rocq this construction is automated using Ltac tactics.

We did not attempt to use an inherently well-typed syntax. This would have probably added a lot of complexity, since our IR has a dependent type system. Moreover, we would have had to prove systematically that passes preserve typing.

6.4 Semantics

We give a denotational semantics to the parameterized IR. This semantics is also parameterized, notably by the semantics of the instance-specific operators. We leverage the structure of the functional SSA instructions to obtain a compositional definition. The definition of the semantics thus directly reflects the nesting of lexical scopes, which itself under-approximates the dominance relation between program locations. Our denotational approach also enables parameterizing the definition of the semantics by an instance-specific domain, and thus by the effects of operators. This parameterization relies on the axiomatization of the `SMonad` library.

Parameters Like the syntax, the semantics is parameterized using typeclasses. We take as parameter the type `val` of values and the semantics $[\![.]\!]_{\text{op}}$ of program operators (its signature is given in figure 6.6). In order to define the semantics of conditionals, we also take as parameter a predicate `to_bool` : `val` → `bool` and a value `vunit` : `val` for the branch hypotheses. Most importantly, the semantics is parameterized by a monadic domain `S` in which programs are denoted. We assume that `S` is equipped with operators and a refinement satisfying the axioms of the `SMonad` library, but we do not put other restrictions. In particular, the domain may support any combination of side effects compatible with the axiomatization of `SMonad`, for instance undefined behaviors or non-determinism.

Signature The signatures of the denotations of the different syntactic elements are given in figure 6.6. The denotation $[\![.]\!]_{\text{instr}}$ of an instruction is a function which takes as argument a value environment `ρ` mapping variables in scope to values and which returns an element of the monadic domain `S`. The result of the execution of an instruction is of type `Res.t K r`. Such a result is either a normal evaluation `Res.Ret vs` to `r` values `vs` or a call `Res.Knt k arg` to a continuation `k` in the scope `K`. We handle general recursion between functions by emitting events in the family `Fct.Call.ev F` to represent internal calls. An inhabitant `Fct.Call.mk f arg` of this family contains the identifier `f` of the callee (in the scope `F`) and the values `arg` passed as arguments. The answer to such an event is the list of values returned by the callee.

Definition The semantics of instructions is defined by structural induction on their syntax in figure 6.6. In this presentation with named variables, we use the notations $[\texttt{xs} \mapsto \texttt{vs}]$ and $[\texttt{ρ}, \texttt{xs} \mapsto \texttt{vs}]$ to define and extend value environments. We emphasize that an extension $[\texttt{ρ}, \texttt{xs} \mapsto \texttt{vs}]$ cannot change the values of variables already defined by `ρ`. The value of a variable thus remains unchanged in its whole scope.
The semantics of each kind of instruction is defined using pure operations of Rocq (such as mapping a value environment `ρ` over a container `u`, with our notation `ρ(u)`) and operators of the domain `S`. The semantics of operators is defined by the parameter $[\![.]\!]_{\text{op}}$ as an element of the domain. Operators thus have access to all effects, except continuation and function calls. Function calls emit corresponding events using the `trigger` operator. The semantics of the `let` construction relies on the `bind` of the monad. If the left-hand side evaluates to some values `vs`, the execution continues with the right-hand side in a value environment extended with the values `vs` for the binders. If the left-hand side calls a continuation, the right-hand side is not executed. Since continuations are recursive, `krec` blocks can diverge. Their semantics rely on the iteration operator `iter` of `S`. We use as iteration state a pair containing the identifier of the next continuation to execute and values for its arguments. This state is initialized with the first continuation `first ks` and the arguments given to `krec`. At each loop iteration, we retrieve the body of the current continuation and execute it in an environment extended with the values of its arguments. If this execution ends with a call to a continuation defined in the block, we continue the loop by returning `Continue` with the new state. Otherwise, we terminate the block by returning `Break` with the same result.
The semantics $[\![.]\!]_{\text{prog}}$ of a program is defined from the semantics of its functions using the recursion operator `mrec` of `S`. Its most general definition, $[\![\texttt{pr}, \texttt{f}_0]\!]_{\text{prog}}$, starts from a call to an arbitrary function $\texttt{f}_0$. By default, we consider the semantics $[\![\texttt{pr}]\!]_{\text{prog}}$, which starts from the main function. In both cases, `mrec` recursively replaces internal calls by their semantics. We obtain semantics which no longer emit any events.
We defined the semantics on the DTS. The use of dependent types and of well-scoped terms simplifies its definition. In particular the environment `ρ` is a total function which is easily mapped on vectors. Moreover, the signature of $[\![.]\!]_{\text{instr}}$ ensures that semantics only perform calls to declared functions and to continuations in scope. Their implementations are thus retrieved using total functions `pr.func` and

TODO (rocq)

- `Kont.sem`

$$[\![\texttt{o} : \texttt{Op.sym_s a}]\!]_{\text{op}}\ (\texttt{vT} : (\texttt{Typ.s val})^{\texttt{a.argT}})\ (\texttt{vV} : \texttt{val}^{\texttt{a.argV}}) : \texttt{S}\ \varnothing\ \texttt{val}^{\texttt{a.res}}$$

$$[\![\texttt{u} : \texttt{Instr.s E r}]\!]_{\text{instr}}\ (\rho : \texttt{E.var} \to \texttt{val}) : \texttt{S}\ (\texttt{Fct.Call.ev E.fct})\ (\texttt{Res.t E.knt})$$

$$[\![\texttt{u} : \texttt{Kont.s E a r}]\!]_{\text{kont}}\ (\rho : \texttt{E.var} \to \texttt{val})\ (\texttt{varg} : \texttt{val}^{\texttt{a}}) : \texttt{S}\ (\texttt{Fct.Call.ev E.fct})\ (\texttt{Res.t E.knt})$$

$$[\![\texttt{fc} : \texttt{Func.s F a}]\!]_{\text{func}}\ (\texttt{varg} : \texttt{val}^{\texttt{a.arg}}) : \texttt{S}\ (\texttt{Fct.Call.ev F})\ \texttt{val}^{\texttt{a.res}}$$

$$[\![\texttt{pr} : \texttt{Prog.s a}, \texttt{f}_0 : \texttt{pr.F a}_0]\!]_{\text{prog}}\ (\texttt{varg}_0 : \texttt{val}^{\texttt{a}_0\texttt{.arg}}) : \texttt{S}\ \varnothing\ \texttt{val}^{\texttt{a}_0\texttt{.res}}$$

$$[\![\texttt{pr} : \texttt{Prog.s a}]\!]_{\text{prog}}\ (\texttt{vargmain} : \texttt{val}^{\texttt{a.arg}}) : \texttt{S}\ \varnothing\ \texttt{val}^{\texttt{a.res}}$$

$$[\![\texttt{return (arg)}]\!]_{\text{instr}}\ \rho := \texttt{ret}\ (\texttt{Res.Ret}\ \rho(\texttt{arg}))$$

$$[\![\texttt{op<argT>(argV)}]\!]_{\text{instr}}\ \rho := \texttt{r} \leftarrow [\![\texttt{op}]\!]_{\text{op}}\ \rho(\texttt{argT})\ \rho(\texttt{argV}); \texttt{ret}\ (\texttt{Res.Ret r})$$

$$[\![\texttt{fcall f(arg)}]\!]_{\text{instr}}\ \rho := \texttt{r} \leftarrow \texttt{trigger}\ (\texttt{Fct.Call.mk f}\ \rho(\texttt{arg})); \texttt{ret}\ (\texttt{Res.Ret r})$$

$$[\![\texttt{let xs = lhs; rhs}]\!]_{\text{instr}}\ \rho := \texttt{r} \leftarrow [\![\texttt{lhs}]\!]_{\text{instr}}\ \rho; \begin{cases} [\![\texttt{rhs}]\!]_{\text{instr}}\ [\rho, \texttt{xs} \mapsto \texttt{vs}] & \text{if } \texttt{r} = \texttt{Res.Ret vs} \\ \texttt{ret}\ (\texttt{Res.Knt c}) & \text{if } \texttt{r} = \texttt{Res.Knt c} \end{cases}$$

$$[\![\texttt{if c H then thn else els}]\!]_{\text{instr}}\ \rho := [\![\texttt{thn if to_bool}\ (\rho\ \texttt{c}), \texttt{els otherwise}]\!]_{\text{instr}}\ [\rho, \texttt{H} \mapsto \texttt{vunit}]$$

$$[\![\texttt{kcall k(arg)}]\!]_{\text{instr}}\ \rho := \texttt{ret}\ (\texttt{Res.Knt k}\ \rho(\texttt{arg}))$$

$$[\![\texttt{krec\{ks\}(arg)}]\!]_{\text{instr}}\ \rho := \texttt{iter}\ \begin{pmatrix} \lambda(\texttt{k}, \texttt{varg}). \\ \quad \texttt{r} \leftarrow [\![\texttt{ks k}]\!]_{\text{kont}}\ \rho\ \texttt{varg}; \\ \quad \texttt{ret}\ (\texttt{select_knt r}) \end{pmatrix}\ (\texttt{first ks}, \rho(\texttt{arg}))$$

where: $\texttt{select_knt r} := \begin{cases} \texttt{Continue}\ (\texttt{k}, \texttt{varg}) & \text{if } \texttt{r} = \texttt{Res.Knt k varg with k} \in \texttt{ks} \\ \texttt{Break r} & \text{otherwise} \end{cases}$

$$[\![\texttt{k(arg): body}]\!]_{\text{kont}}\ \rho\ \texttt{varg} := [\![\texttt{body}]\!]_{\text{instr}}\ [\rho, \texttt{arg} \mapsto \texttt{varg}]$$

$$[\![\texttt{fn f(arg) : (res) \{body\}}]\!]_{\text{func}}\ \texttt{varg} := (\texttt{Res.Ret vres}) \leftarrow [\![\texttt{body}]\!]_{\text{instr}}\ [\texttt{arg} \mapsto \texttt{varg}]; \texttt{ret vres}$$

$$[\![\texttt{pr}, \texttt{f}_0]\!]_{\text{prog}}\ \texttt{varg}_0 := \texttt{mrec}\ \begin{pmatrix} \lambda(\texttt{Fct.Call.mk f varg}). \\ [\![\texttt{pr.func f}]\!]_{\text{func}}\ \texttt{varg} \end{pmatrix}\ (\texttt{Fct.Call.mk}\ \texttt{f}_0\ \texttt{varg}_0)$$

$$[\![\texttt{pr}]\!]_{\text{prog}}\ \texttt{vargmain} := [\![\texttt{pr}, \texttt{pr.main}]\!]_{\text{prog}}\ \texttt{vargmain}$$

Figure 6.6: Semantics of instructions and programs in our generic functional SSA IR

`ks` in $[\![.]\!]_{\text{prog}}$ and the `krec` case. We thus do not have to account in the semantics for errors that are statically ruled out by the well-formedness of the programs.

6.5 Typing

Section 6.5.1 defines a typing judgment on the parameterized IR. We prove that this judgment is sound with respect to the semantics, under the assumption that its instance-specific parameters are sound. Section 6.5.2 describes this soundness property, which is a type preservation property. This property ensures that variables always contain well-typed values, and is used during proofs of transformations.

6.5.1 Typing judgment

We define a typing judgment on programs in the parameterized IR. This judgment depends on instance-specific parameters:

- The signature of operators, defined by a function `Op.sig [a] (o : Op.sym_s a) : Op.Sig.s a`. Each operator is given a signature with shape `<xsT>(xsV : targ) : (xsR : tres)`, where `xsT`, `xsV` and `xsR` are lists of formal variables for the type arguments, value arguments and results. The types `targ` depend on `xsT` and `xsV`. The types `tres` additionally depend on `xsR`.

- A sub-typing relation `Typ.sub [A] (t₀ t₁ : Typ.s A) : Prop` between open types. We use the notation $t_0 \subseteq t_1$ for `Typ.sub t₀ t₁`. We assume that this relation is a pre-order and compatible with variable renaming. We generalize the relation to lists of types in a pointwise fashion. The equality is a possible instance for `Typ.sub`. In Frag IR, we define a type `any` containing all values in order to give a signature to `Frag.joinable`. The relation $t \subseteq \texttt{any}$ is satisfied for any type `t`. In all other cases, sub-typing is the equality.
- The typing judgment of conditionals uses a type `bool` for the guard and types `c = true` and `c = false` (parameterized by a type expression `c`) for the branch hypotheses.

The typing judgment on programs is defined on the DTS. It is a predicate `Prog.wt [a] (pr : Prog.s a) : Prop`. Its definition (figure 6.7) involves typing judgments on functions and instructions:

- The typing judgment on functions has signature `Func.wt [F] (Γ : Fct.Ty.s F) [a] (fc : Func.s F a) : Prop`. It depends on a typing environment `Γ` containing the signatures of the functions defined in the program. The signature of a function has shape `(xsA : targ) : (xsR : tres)`. The types `targ` depend on the formal arguments `xsA`. The types `tres` additionally depend on the formal results `xsR`.
- The typing judgment on instructions has signature `Instr.wt [E] (Γ : Env.Ty.s E) [r] (u : Instr.s E r) (ts : Binders.s r E.var) : Prop`. We use the notation `Γ ⊢ u : (xs : ts)`, where `xs` are formal names for the values returned. The typing environment `Γ` is a record containing the types and signatures of the variables (`var`), continuations (`knt`) and functions (`fct`) in scope. The judgment is also parameterized by the types `ts` of the values returned by the instruction. The types of the variables, of continuation arguments and the types `ts` depend on the variables in scope. The types `ts` additionally depend on `xs`.

The typing judgment is defined inductively on instructions in figure 6.7. Most rules involve the comparison of open types, using `Typ.sub`. The free variables of those types are the variables in the context of the instruction, and in some cases (operator and function calls) the values `xs` returned by the instruction. For instance, for a context `Γ` containing the variables `x : i64`, `y : i64` and `H : x ≤ y`:

$$\texttt{Γ} \vdash \texttt{return (x, H) : (z : i64, H' : z ≤ y)} \quad \Leftrightarrow \quad \texttt{(i64, x ≤ y)} \subseteq \texttt{(i64, z ≤ y)[x/z, H/H']}$$

where `(i64, z ≤ y)[x/z, H/H']` denotes the substitution of `z` by `x` and of `H'` by `H` in the types `(i64, z ≤ y)`. This substitution evaluates to `(i64, x ≤ y)`, so the instruction is well-typed.

The rules for operator, continuation and function calls compare the signature of the callee with the arguments and return types of the instruction. For continuation calls, the return types of the instruction are ignored since those calls do not return to the caller. In `let` bindings, the left-hand side is typed with the types of the bound variables as return types. The right-hand side has the same return types as the whole sequence, but its environment additionally contains the types of the bound variables. In conditionals, the branch hypothesis `H` has different types in the two branches to reflect the evaluation of the Boolean guard. The arguments of `krec` blocks must match the types of the arguments of the initial continuation. Each declared continuation `kn` is typed (using an auxiliary predicate `Kont.wt`) in an environment extended with the signatures of the declared continuations, and with the types of the arguments of `kn`. A function is well-typed if its body is. A program is well-typed if all its functions are.

6.5.2 Type preservation

We prove that the typing judgment is sound with respect to the semantics. This soundness is a kind of preservation property: if a well-typed element receives well-typed inputs, then its outputs are well-typed. This result is parameterized by an instance-specific interpretation of closed types as sets of values, `Typ.sem (ty : Typ.s val) (v : val) : Prop`. We write `v ∈ ty` and extend the notation to lists of types and values in a pointwise fashion. This interpretation must be compatible with the parameters of the typing judgment and of the semantics. That is:

TODO (rocq)

- `Kont.wt`

$$\frac{\Gamma.\mathtt{var}(\mathtt{arg}) \subseteq \mathtt{ts}[\mathtt{arg}/\mathtt{xs}]}{\Gamma \vdash \mathtt{return\ (arg)} : (\mathtt{xs} : \mathtt{ts})}$$

$$\frac{\mathtt{Op.sig\ op} = (\mathtt{<xsT>(xsV} : \mathtt{targ}) : (\mathtt{xsR} : \mathtt{tres})) \quad \Gamma.\mathtt{var}(\mathtt{argV}) \subseteq \mathtt{targ}[\mathtt{argT}/\mathtt{xsT}, \mathtt{argV}/\mathtt{xsV}] \quad \mathtt{tres}[\mathtt{argT}/\mathtt{xsT}, \mathtt{argV}/\mathtt{xsV}, \mathtt{xs}/\mathtt{xsR}] \subseteq \mathtt{ts}}{\Gamma \vdash \mathtt{op<argT>(argV)} : (\mathtt{xs} : \mathtt{ts})}$$

$$\frac{\Gamma.\mathtt{fct\ f} = ((\mathtt{xsA} : \mathtt{targ}) : (\mathtt{xsR} : \mathtt{tres})) \quad \Gamma.\mathtt{var}(\mathtt{arg}) \subseteq \mathtt{targ}[\mathtt{arg}/\mathtt{xsA}] \quad \mathtt{tres}[\mathtt{arg}/\mathtt{xsA}, \mathtt{xs}/\mathtt{xsR}] \subseteq \mathtt{ts}}{\Gamma \vdash \mathtt{fcall\ f(arg)} : (\mathtt{xs} : \mathtt{ts})}$$

$$\frac{\Gamma \vdash \mathtt{u} : (\mathtt{xs}_0 : \mathtt{ts}_0) \quad \Gamma, (\mathtt{xs}_0 : \mathtt{ts}_0) \vdash \mathtt{k} : (\mathtt{xs}_1 : \mathtt{ts}_1)}{\Gamma \vdash (\mathtt{let\ xs}_0 : \mathtt{ts}_0 = \mathtt{u;\ k}) : (\mathtt{xs}_1 : \mathtt{ts}_1)}$$

$$\frac{\Gamma.\mathtt{var\ c} \subseteq \mathtt{bool} \quad \Gamma, (\mathtt{H} : \mathtt{c} = \mathtt{true}) \vdash \mathtt{thn} : (\mathtt{xs} : \mathtt{ts}) \quad \Gamma, (\mathtt{H} : \mathtt{c} = \mathtt{false}) \vdash \mathtt{els} : (\mathtt{xs} : \mathtt{ts})}{\Gamma \vdash (\mathtt{if\ c\ H\ then\ thn\ else\ els}) : (\mathtt{xs} : \mathtt{ts})}$$

$$\frac{\Gamma.\mathtt{knt\ k} = (\mathtt{xsA} : \mathtt{targ}) \quad \Gamma.\mathtt{var}(\mathtt{arg}) \subseteq \mathtt{targ}[\mathtt{arg}/\mathtt{xsA}]}{\Gamma \vdash \mathtt{kcall\ k(arg)} : (\mathtt{xs} : \mathtt{ts})}$$

$$\frac{\forall \mathtt{kn} \in \mathtt{ks.Kont.wt}\ (\Gamma, \mathtt{ks})\ \mathtt{kn}\ (\mathtt{xs} : \mathtt{ts}) \quad (\mathtt{first\ ks}).\mathtt{arg} = (\mathtt{xsA}_0 : \mathtt{targ}_0) \quad \Gamma.\mathtt{var}(\mathtt{arg}_0) \subseteq \mathtt{targ}_0[\mathtt{arg}_0/\mathtt{xsA}_0]}{\Gamma \vdash \mathtt{krec\{ks\}(arg}_0) : (\mathtt{xs} : \mathtt{ts})}$$

$$\frac{\Gamma, (\mathtt{xarg} : \mathtt{targ}) \vdash \mathtt{body} : (\mathtt{xs} : \mathtt{ts})}{\mathtt{Kont.wt}\ \Gamma\ (\mathtt{k(xarg} : \mathtt{targ}) \mathtt{:\ body})\ (\mathtt{xs} : \mathtt{ts})}$$

$$\frac{\Gamma, (\mathtt{xarg} : \mathtt{targ}) \vdash \mathtt{body} : (\mathtt{xres} : \mathtt{tres})}{\mathtt{Func.wt}\ \Gamma\ (\mathtt{fn\ f(xarg} : \mathtt{targ}) : (\mathtt{xres} : \mathtt{tres})\ \{\mathtt{body}\})}$$

$$\frac{\forall \mathtt{fc} \in \mathtt{pr.func.Func.wt\ pr.func\ fc}}{\mathtt{Prog.wt\ pr}}$$

Figure 6.7: Typing judgment for programs in the parameterized IR

- `Typ.sub` must be sound with respect to the semantics of types, that is:

$$\forall \mathtt{ty}_0\ \mathtt{ty}_1.\ \mathtt{Typ.sub\ ty}_0\ \mathtt{ty}_1 \Rightarrow \forall \mathtt{v}.\ \mathtt{Typ.sem\ ty}_0\ \mathtt{v} \Rightarrow \mathtt{Typ.sem\ ty}_1\ \mathtt{v}$$

- The semantics of operators return well-typed results when applied to well-typed arguments.
- The types of the guard hypotheses are inhabited by `vunit` if the guard has the prescribed value. Formally, if $\mathtt{b} \in \mathtt{bool}$ then $\mathtt{vunit} \in (\mathtt{b} = \mathtt{to_bool\ b})$.

TODO (rocq)

- soundness du typage pour les fonctions et programmes
- version dépliée de `Instr.wt_sound`

Using those assumptions, we derive a type preservation property for instructions. The main invariant is the fact that the variables are mapped to well-typed values:

Definition 6.1 In a scope `V` of variables, a *value environment* `ρ : V → val` is *well-typed* with respect to a typing environment `Γ : V → Typ.s V` if:

$$\forall x.\ \rho\ x \in \rho(\Gamma\ x)$$

We capture type preservation as a refinement of the semantics of an instruction by itself, as presented in section 4.3.3. We use the relations on the interactions with the context to express the typing invariants on those interactions.

Lemma 6.1 If `Γ ⊢ u : (xs : ts)` then for any well-typed value environment `ρ : Γ.var → val`:

$$[\![u]\!]_{\text{instr}}\ \rho \gtrsim_{(\text{Re_ty}\ \Gamma),(\text{R_ty}\ \Gamma\ ts)} [\![u]\!]_{\text{instr}}\ \rho$$

where the post-condition `R_ty Γ ts` on results is defined by:

$$\frac{vs \in [\rho, xs \mapsto vs](ts)}{\text{R_ty}\ \Gamma\ ts\ (\text{Res.Ret}\ vs)\ (\text{Res.Ret}\ vs)}$$

$$\frac{\Gamma.\text{knt}\ k = (xsA : targ) \qquad arg \in [\rho, xsA \mapsto arg](targ)}{\text{R_ty}\ \Gamma\ ts\ (\text{Res.Knt}\ k\ arg)\ (\text{Res.Knt}\ k\ arg)}$$

and the invariant on function calls `Re_ty Γ` (a relation on events) by:

$$\frac{\Gamma.\text{fct}\ f = ((xsA : targ) : (xsR : tres)) \qquad arg \in [xsA \mapsto arg](targ)}{\begin{array}{l}\text{Re_ty}\ \Gamma\ (\text{Fct.Call.mk}\ f\ arg)\ (\text{Fct.Call.mk}\ f\ arg) \\ \quad (\lambda res_0\ res_1.\ res_0 = res_1 \wedge res_0 \in [xsA \mapsto arg, xsR \mapsto res_0](tres))\end{array}}$$

Idea of the proof by induction on the typing judgment.

Since we relate the semantics $[\![u]\!]_{\text{instr}}\ \rho$ to itself, the relations `Re_ty` and `R_ty` ensure that the interactions with the context are unchanged. Moreover, they assert that those interactions are well-typed, that is:

- When the instruction terminates normally, the case `Res.Ret` of `R_ty` ensures that the returned values inhabit the return types `ts` of the instruction.
- When the instruction calls a continuation, the case `Res.Knt` of `R_ty` ensures that the values passed as arguments have the types expected by the callee.
- When the instruction calls a function, `Re_ty` ensures that the arguments have the types expected by the callee. In return, its post-condition ensures that the semantics receives well-typed results.

Some transformations (e.g., the erasure of section 7.2.3) use the invariant that the source value environment is well-typed. To maintain this invariant in combinators, we need the fact that the outputs of components are well-typed. For instance in `let xs : ts = u; k`, we need the fact that the results of `u` have types `ts` to prove that `k` is evaluated inside a well-typed value environment. However, we do not need to reprove that those outputs are well-typed. Instead, we use the transitivity of `sem_ref`, to combine lemma 6.1 with weaker refinements of `u`, as explained in section 4.3.3.

6.6 Related works

6.6.1 Formalizations of SSA

Conventional forms of SSA and transformations exploiting it have been formalized in Vellvm [107; 108] and CompCertSSA [12]. Both of those developments reason on the dominance relation in CFGs. In our approach based on functional SSA [7], syntactic scopes provide an under-approximation of dominance and structure the semantics of programs. Ghalayini and Krishnaswami [43] use a similar approach to give a denotational semantics to SSA, in a more abstract categorical setting. Schneider [91] defines

a notion of *coherence* to reason on programs in SSA. This notion implies the equivalence between a functional and an imperative semantics. In contrast, we only define a functional semantics for our IR.

6.6.2 Representation of variables

CompCert [69], Vélus [21] and VIR [105] mainly use named representations of variables. The initial phase of CakeML [54] converts local variables to De Bruijn indices [35]. Well-scoped De Bruijn indices [2] rule out unbound references and often guide definitions and proofs. However, they still require some administrative lemmas. In some cases, Autsubst [95] can provide those lemmas automatically. PHOAS [30] avoids those lemmas by relying on the host proof assistant's binders. However, it is unclear whether this approach could be used for the continuations of our IRs since they are not values.

6.6.3 Modular compilers

MLIR [61] is extensible thanks to dialects and enables the encoding of Domain Specific Languages. Abstracting over effects and arbitrary computations would be necessary for the formalization of such highly generic compiler infrastructure. Bhat et al. [18] proposed a semantics for MLIR in Lean. Their representation is parameterized by a dialect which defines the sets of types and operators and their semantics. Dialects even define the available control flow structures, thanks to operators taking code blocks (called "regions") as arguments. However, their theory does not support side effects, in particular non-termination.

CompCert [69] now supports several target assemblies. The intermediate representations and passes remain generic thanks to the use of target-specific modules with a common signature. Vélus [21, §4.1] uses the module functors provided by Rocq for a similar purpose. The generic parts of those developments abstract over the semantics of operators, but the effects of those semantics are a strict subset of the effects of the languages. In contrast, we abstract over operators that use nearly all effects and even over the effects themselves. Ebresafe et al. [38] refactored CompCert to improve its extensibility thanks to a Rocq plugin (Rocqet) providing features such as extensible inductive families.

7

Passes on the parameterized representation

In chapter 6, we defined a parameterized IR to factorize the definition of several IRs. This modular approach also enables defining passes on the parameterized IR. The implementation of each of those passes has a large generic part, and in some cases some instance-specific parameters. The generic parts deal with aspects related to the structure of the parameterized IR, such as the inductive instructions and the handling of variables. The parameters handle instance-specific aspects, such as the effect of the passes on the operators of the IRs. The parameterized passes are thus available on multiple IRs, and their generic parts are shared between IRs. Indeed, instantiating a pass on an IR only requires implementing the parameters of the pass.

Thanks to the meta-theory around the parameterized IR, the proofs of correctness of several instances of a parameterized pass are also largely factorized. Indeed, we prove correctness properties on the parameterized pass under some assumptions on the instance-specific parameters. Deriving the correctness of instances of the passes only requires proving those assumptions. In particular, since the parameterized IR has a semantics, we are able to prove that all our transformations preserve the semantics of programs.

This chapter describes the parameterized passes we implemented: a simplification pass (section 7.1), a dead code elimination pass (section 7.2), a block-sinking pass (section 7.3) and a type-checker (section 7.4). We mention the parameters of each pass but focus on their generic parts.

7.1 A simplification pass

We implemented a simplification pass relying on the monad laws to linearize sequences of `let` binders and to reduce `return`s when they appear on the left of a `let`. Figure 7.1 describes an example of simplification. This pass does not need anything specific to the instance, except the monad laws on the domain `S`, which we always assume.

Definition Figure 7.2 describes the transformation of instructions. The transformation of a source instruction `u` is given by `BindSimpl.transf_instr M u k`. The argument `M` maps the free variables of `u` to target variables containing the same values. The argument `k` is an optional target continuation that must be performed after a normal termination of the instruction. When this continuation is present, it is defined by a pair of:

```
let z0, z1 = {                          let y = fcall f(x0, x1);
  let y = fcall f(x0, x1);       →      fcall g(y, x1);
  return (y, x1) };
fcall g(z0, z1)
```

Figure 7.1: Example of the simplification pass [compiler/test/src/pass/simpl/t1_fcall]

```
BindSimpl.transf_instr M (return (xs)) ∅ := return (M(xs))

BindSimpl.transf_instr M (return (xs)) ⌊_, kf⌋ := kf M(xs)

BindSimpl.transf_instr M (let xs : ts = lhs; rhs) k :=
  BindSimpl.transf_instr M lhs ⌊(xs : ts), λys. BindSimpl.transf_instr [M, xs ↦ ys] rhs k⌋

BindSimpl.transf_instr M (kcall l(xs)) _ := kcall l(M(xs))

BindSimpl.transf_instr M (if c H then thn else els) ∅ :=
  if (M c) H
  then BindSimpl.transf_instr [M, H ↦ H] thn ∅
  else BindSimpl.transf_instr [M, H ↦ H] els ∅

BindSimpl.transf_instr M (if c H then thn else els) ⌊(xs : ts), kf⌋ :=
  let xs : M(ts) = BindSimpl.transf_instr M (if c H then thn else els) ∅; kf xs
```

Figure 7.2: Definition of the simplification pass on instructions

- Binders for the results of `u`, that is their types and annotations (e.g. names).
- A function generating the target instruction for the continuation. This function takes as argument the list of target variables which store the results of `u`.

Target continuations are introduced when transforming `let`s. In this case, the target instruction is generated by transforming the left-hand side with a continuation which emits the right-hand side. The target continuation `k` of the whole `let` is given to the transformation of the right-hand side. This ensures that sequences of `let`s are right-nested in the target. We suppress `return`s under continuations (that is, when they are in the left-hand side of a `let`). We instead directly pass the arguments of `return` to the function which generates the target continuation. Since calls to functional SSA continuations never return, we ignore the target continuation for `kcall`. In all other types of instructions, if a target continuation `k` is present, we emit a `let` to execute it after the instruction `u`. Even if we do not perform simplification on the head of `u`, we call the transformation function recursively on its sub-instructions. For instance, figure 7.2 reproduces the transformation of conditionals. In this case, we simplify recursively the two branches. When there is a continuation `⌊k⌋`, `let`-binding the results instead of passing `k` to the branches avoids duplicating `k`.

Implementation The transformation is directly defined as a function on the AS (section 6.3). The implementation follows the structure of figure 7.2. However, in the implementation, our syntax uses De Bruijn indices [35] to represent variables. That is, the level of variables is represented relatively to the point where they are referred to, by counting the number of scopes opened since the declaration of the variable. With this representation, the reference to a given variable depends on the program point of the reference. Our transformation function involves a map from source to target variables. With a representation of variables with standard De Bruijn indices, we would have to update all bindings each time a scope is opened in the source or target. This would lead to a time complexity quadratic in the size of the transformed instructions. Instead, our implementation of the map uses reversed De Bruijn indices [33] (also called "levels" [35, §4]). That is, we use *absolute* levels by counting the number of scopes opened between the beginning of the function and the point where the variable is declared. This representation is independent of the point where the variable is referred to. In the implementation of `BindSimpl.transf_instr`, we track the current source and target absolute levels to convert between standard and reversed De Bruijn indices.

Correctness We proved that this transformation preserves the semantics of programs, that is that the semantics of the target refines the semantics of the source. In the development, we proved simultaneously the existence of a DTS for the target program, since it is necessary to reason on its semantics. We did not prove that the transformation preserves the typing, but we expect that it is the case. Since the transformation is intra-procedural, the source and target perform the same function calls. Using `ref_mrec` with the identity as invariant, we thus reduce the proof of the refinement of programs to refinements between instructions. The core of the proof is then to give a specification to `BindSimpl.transf_instr M u k`. To define this specification, we first formalize the effect of the parameters `M` and `k` of the transformation.
The map `M` between source and target variables defines a relation between the value environments:

Definition 7.1

TODO (rocq)

définit comme une relation en Rocq

Let `M` be a variable map from a source scope V_0 to a target scope V_1. Let $\rho_0 : V_0 \to \texttt{val}$ (resp. $\rho_1 : V_1 \to \texttt{val}$) be value environments for the variables V_0 (resp. V_1). We say that `M` matches ρ_0 and ρ_1 if and only if, for any source variable $x : V_0$, $\rho_1\ (M\ x) = \rho_0\ x$.

We define a relation associating semantics `sem_k` to target continuations `k`:

Definition 7.2 Let `M` be a variable map from a source scope V_0 to a target scope V_1. Let `k` be an optional target continuation for an instruction returning `n` variables, in a target environment with variables V_1, continuations `K` and functions `F`. Let `sem_k` : $(V_0 \to \texttt{val}) \to \texttt{val}^n \to S\ (\texttt{Fct.Call.ev}\ F)\ (\texttt{Res.t}\ K)$ be a candidate semantics for this continuation. We say that `k` refines the semantics `sem_k` if either:

- `k` is $\varnothing$ and `sem_k` is $\lambda_\ \texttt{vs}.\ \texttt{ret}\ (\texttt{Res.Ret}\ \texttt{vs})$
- **TODO** (rocq)

  en Rocq, on quantifie sur les valeurs renvoyées

  `k` is $\lfloor \texttt{bs}, \texttt{kf} \rfloor$ and for any extension V_1' of the scope V_1, list `xs` of `n` variables of V_1', value environments $\rho_0 : V_0 \to \texttt{val}$ and $\rho_1 : V_1 \to \texttt{val}$; if `M` matches ρ_0 and ρ_1 then:

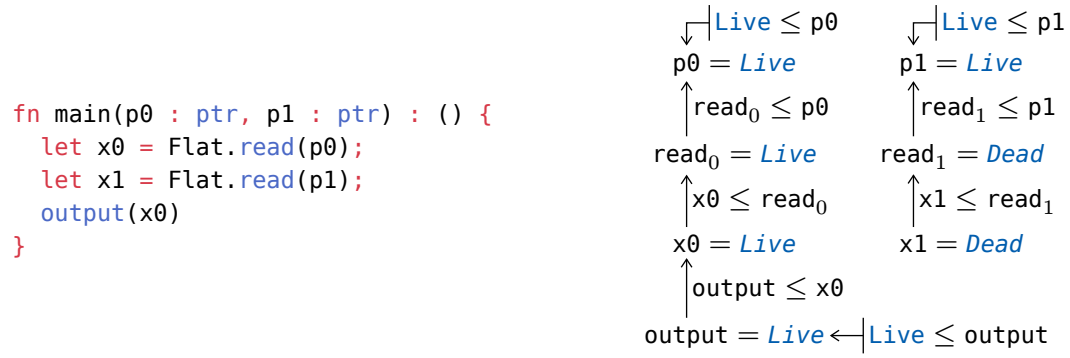
$$\texttt{sem_k}\ \rho_0\ \rho_1(\texttt{xs}) \gtrsim [\![\texttt{kf}\ \texttt{xs}]\!]_{\text{instr}}\ \rho_1$$

Note that, although `k` describes a continuation in the target, we relate it directly to a source semantics `sem_k` (that is, which depends on the source value environment). The reason is that this semantics `sem_k` must be defined when using the specification of `BindSimpl.transf_instr`. At those points in the proofs, we have access to the part of the source instruction refined by the target continuation, and we define `sem_k` from this syntactic description.
We give a specification to `BindSimpl.transf_instr`, thanks to those formalizations of its parameters:

Lemma 7.1 Let u_1 be the target instruction generated from a source instruction u_0 by `BindSimpl.transf_instr M` u_0 `k`. If `sem_k` is a semantics refined by the target continuation `k`, and if ρ_0 and ρ_1 are value environments related by `M`, then:

$$\texttt{r} \leftarrow [\![u_0]\!]_{\text{instr}}\ \rho_0; \begin{cases} \texttt{sem_k}\ \rho_0\ \texttt{vs} & \text{if } \texttt{r} = \texttt{Res.Ret vs} \\ \texttt{ret}\ (\texttt{Res.Knt c}) & \text{if } \texttt{r} = \texttt{Res.Knt c} \end{cases} \gtrsim [\![u_1]\!]_{\text{instr}}\ \rho_1$$

Idea of the proof By induction on the syntax of the source instruction u_0, using the monad laws on the semantic domain `S`.



a: Source

b: Data-flow problem and *solution*

```
fn main(p0 : ptr, p1 : ptr) : () {
  let      x0 =      Flat.read(p0);
  let dead x1 = dead Flat.read(p1);
  output(x0)
}
```

c: Annotated

```
fn main(p0 : ptr, p1 : ptr) : () {
  let x0 = Flat.read(p0);
  let () = return ();
  output(x0)
}
```

d: Target

Figure 7.3: Example of the DCE on Flat IR [compiler/test/src/pass/flat_dce/t0_read]

In particular, for u_0 the body of a function, M the identity on the arguments of the function and $k{=}\varnothing$, we obtain after some rewritings a refinement $[\![u_0]\!]_{\text{instr}}\ \rho_0 \gtrsim [\![u_1]\!]_{\text{instr}}\ \rho_1$. We use this refinement to justify the premise of `ref_mrec` and to conclude the proof.

7.2 Inter-procedural Dead Code Elimination

We define an erasure pass on the parameterized IR. This pass removes some variables and operations which are not used to compute the outputs of the program. Figure 7.3 reproduces its effect on a Flat IR program, that is, with a global memory. In this example, figure 7.3d is obtained from figure 7.3a by removing the variable `x1` and an operation `Flat.read` (a read of the global memory). The pass is inter-procedural: it is able to remove arguments and results of functions but not the function calls themselves. The pass is parameterized by Boolean predicates which signal the variables and operations to remove. The erasure pass uses those predicates defensively and fails if they are not coherent (e.g. if an erased variable is used in an operation which is kept). Those predicates avoid the removal of effectful operations (section 7.2.1). They also allow using the pass either to erase ghost code (using `ghost` annotations on variables and operations) or to remove dead code. For the second use, we implemented a sparse liveness analysis on the parameterized IR. We thus obtain a Dead Code Elimination (DCE) transformation by:

1. Running the liveness analysis (section 7.2.2) and annotating its results on the program (figure 7.3c).
2. Removing the annotated dead code using the erasure pass (section 7.2.3).

7.2.1 Removable operations and variables

Some operations have side effects. Those side effects may influence the outputs of the program even if the results returned by the operation are not used. For instance the `output` operation does not return any value but emits an external event. Since compilation must preserve external events, this operation cannot be removed. Memory writes `Flat.write` also cannot be removed by our pass, since they can influence the continuation of the program through the global memory state. On the contrary, memory reads `Flat.read` are removable despite their effects (undefined behavior and access to the memory).

We formally characterize semantic parts which can be removed during a transformation using a refinement:

Definition 7.3 A semantics `u : S E A` is *removable* if $u \gtrsim_{\top} \texttt{ret tt}$.

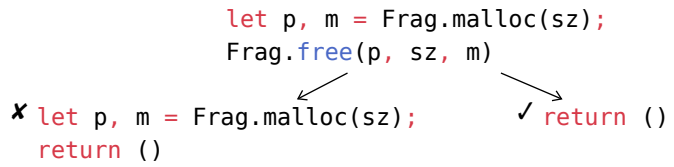


Figure 7.4: Two semantically correct erasures of a dead allocation [compiler/test/src/transform/free_DCE.frag]. However, the left one introduces a memory leak and uses memory fragments in an affine but non-linear way. Currently, our DCE leaves such a program unchanged.

That is, removable semantics are those that can be replaced by a semantics which has no effect (`ret tt`) if we ignore their result (post-condition $\top$). Examples of removable semantics are pure computations (`ret`), undefined behaviors (`ub`), non-deterministic choices (`any`) in non-empty types and finite sequences of them (`bind`). We derive a corresponding notion on program operators:

Definition 7.4 A program operator is *removable* if for any well-typed arguments, its semantics is removable.

Since this notion is specific to the set of operators, both the liveness analysis and the erasure pass take instance-specific parameters (figure 7.5) which ensure that only the semantically removable operations can be considered dead.

In Flat IR, the operator `Flat.read` is removable, despite its effects (access to the global memory and UB). Our generic erasure pass is thus able to remove this operation; although its proof is performed in the abstract domain `S`, that is, without reasoning on the global memory.

In Frag IR, the operator `Frag.free` does not return any value and its semantics has no side effects. Semantically, it would thus always be correct to remove it. However, as illustrated in figure 7.4, doing so without removing corresponding `Frag.mallocs` would introduce memory leaks and would not preserve the linearity condition on memory fragments (section 3.2.5). Removing unused memory fragments can also break the linearity condition in some edge-cases involving infinite loops and unreachable paths [compiler/test/src/pass/frag_dce/6_unused_frag]. We conservatively never consider `Frag.free` and memory fragments as dead.

7.2.2 Analysis

The goal of the liveness analysis is to compute a set of variables and operations which can be removed from a given program. This computation is performed by generating and solving a data-flow problem (figure 7.3b). A *data-flow problem* is defined by a set of data-flow variables and inequalities between them. A solution of a problem is a valuation of the variables satisfying the inequalities. In the liveness analysis, the variables take values in a semi-lattice `Liveness.t` with three elements: `Dead` $<$ `Ghost` $<$ `Live`. In solutions of the problem, `Dead` is associated to program components which can be removed. The element `Ghost` is associated to program components which are used only in ghost computations (e.g. memory fragments or for dependent types, chapter 3). Our goal is to remove as much code as possible, so we are looking for the smallest solution of the problem.
Annotating code as ghost ensures that it will ultimately be removed, at the end of our compilation pipeline. Some of our transformations operate differently on ghost code (in section 9.2 we will not transform ghost variables, and in section 9.3 we will not transform ghost reads). In the future, optimizations could also ignore parts annotated as ghosts, since they do not directly influence the efficiency of the final target program.

The problems we generate are sparse thanks to the SSA structure [25; 85]. Indeed, since program variables have a unique definition, we can easily link the use of variables to their definitions. Hence,

TODO (rocq)
- spécialisation de `Typ.t`
- pas de prise en compte des arguments de type
- ? `res` pour les `op_`

```
is_typ_linear (ty : Typ.t) : bool

get_op_properties (op : Op.sym_t) : { min_live : Liveness.t
                                    ; arg_ghost : list bool }

set_var_l (l : Liveness.t) (v : An.var) : An.var
set_op_l (l : Liveness.t) (o : Op.sym_t) : Op.sym_t
```

a: Liveness analysis

```
is_var_erased (an : An.var) : bool
is_op_erased (op : Op.sym_t) : bool
```

b: Erasure

Figure 7.5: Instance-specific parameters of the implementation of the DCE, described in section 7.2.2 and section 7.2.3

there is a single data-flow variable per program variable, whereas analyses on representations without SSA may use one variable per program variable and program point.

The liveness analysis is inter-procedural: we generate a single data-flow problem for the whole program. This problem contains a data-flow variable for each program variable (variables bound in `let`, branch hypotheses and formal arguments and results of continuations and functions). For instance figure 7.3b contains the variables `x0`, `x1`, `p0` and `p1`. We also add variables representing the liveness of operators (e.g. $\texttt{read}_0$ in figure 7.3b). Each use of a variable generates an inequality reflecting the dependency between the use and the definition (e.g. `output` $\leq$ `x0`). The instance-specific parameters (figure 7.5a) `is_typ_linear` and `get_op_properties` (with its field `min_live`) give a minimal liveness to variables and operators. We add corresponding lower-bounds in the data-flow problem (e.g. `Live` $\leq$ `output`). The record returned by the parameter `get_op_properties` also contains a field `arg_ghost` which signals the arguments used in a ghost fashion by operators. For instance, this is the case for the memory fragment arguments of memory operations in Frag IR since the fragments are not used by their translation in Flat IR. We add a `min` (using the total order of `Liveness.t`) to weaken the inequalities corresponding to those arguments. That is, we emit inequalities with shape `min Ghost x_use` $\leq$ `x_def`. Finally, we add lower bounds on the arguments and results of the main function (e.g. `Live` $\leq$ `p0`) to ensure that the signature of the whole program remains unchanged.

We generate the data-flow problem for the whole program by structural recursion on the syntax of the program. We then find the smallest solution of this problem using a worklist-based iteration. We reused the worklists of CompCert's implementation of Kildall's algorithm [55; 69, §7.1]. We annotated the solution next to the variables in figure 7.3b. In this example, the analysis detects that the value read at address `p1` can be removed. We store the solution directly in the AS of the program using annotations on the variables and operations (figure 7.3c). In our implementation, those annotations are parts of our instances of `An.var` and `Op.sym_t`. They are thus managed by the instances of the generic IR. The liveness analysis sets them using some parameters `set_var_l` and `set_op_l`. We did not prove the correctness of our data-flow solver, nor the coherence of the annotations added by the liveness analysis. Instead, the erasure pass uses the annotations defensively.

7.2.3 Erasure transformation

Once we know the set of variables and operators to remove, we run the erasure pass to transform the program. The transformation follows the structure of the instructions. We captured its effect on the

DTS using a predicate `Erasure.instr_match eE er r_ts0 s_u0 s_u1`. This predicate relates a source instruction s_u_0 to a target instruction s_u_1. It is indexed by:

- An erasure environment `eE` which links the symbols in the environment of s_u_1 to symbols of s_u_0. For variables, this link is given by an injection `eE.var` from the target variables to the source variables. We need an injection, and not simply the identity relation, because our well-scoped instructions have distinct sets of free variables. We use injections rather than general functions to preserve the typing in presence of dependent types. As a counter-example, the following ill-typed transformation would be accepted by `Erasure.instr_match` if we allowed a function mapping both `y0` and `y1` to `y`:

```
let (x: i64, H0: x = 0) = …;          let (x: i64, H0: x = 0) = …;
let (y: i64, H1: y = 0) =             let (y0: i64, y1: i64, H1: y0 = 0) =
  return (x, H0);              →        return (x, x, H0);
let (z: i64, H2: z = 0) =             let (z: i64, H2: z = 0) =
  return (y, H2);                       return (y1, H1);
```

  The relation between source and target continuations is given by a set `eE.knt` containing triples $(k_0, k_1, \texttt{eka})$ of a source continuation k_0, a target continuation k_1 and an injection `eka` from the formal arguments of the target to the corresponding arguments of the source. A set `eE.fct` links functions similarly, by also specifying the injection between the formal results.
- An injection `er` from the formal results of s_u_1 to the formal results of s_u_0.
- Binders r_ts_0 for the results of the source s_u_0.

Similar relations are defined for the other syntactic constructs: continuations, functions, programs (`Erasure.prog_match`)… In particular, figure 7.6 depends on the relation `Erasure.binders_match eV er` between source and target binders. This relation is parameterized by injections of the variables (`eV`) and of the formal bound variables (`er`). Figure 7.6 reproduces a subset of the lemmas on `Erasure.instr_match` that we use to inductively transform the instructions. In this presentation, we see lists as functions with formal variables as domain. For instance, the premise of `Erasure.Ret` is an equality between functions whose domain is the set of target formal results and codomain the set of source variables. The combinators of instructions (`let`, `if` and `krec`) are preserved. In particular, our erasure pass does not remove unreachable code. Erased variables are removed from the lists of binders (e.g. in `Erasure.Bind`). Using `Erasure.Op`, erased operations are replaced by `return ()`, which are removed by later passes (the simplification pass of section 7.1 or the block sinking of section 7.3). The lemma `Erasure.erase_results` is used when a first erasure u_1 of an instruction u_0 returns more results than needed. In this case, we `let`-bind the results then return only the subset expected by the context. This rule is the reason why we propagate the types r_ts_0 of the results in the source. Indeed, we use them to give correct types r_ts_1 to the `let`-bound variables.

This presentation of the transformation as a relation does not specify how the erased code is chosen. Instead, it captures all coherent choices of erased code. For instance, when applying `Erasure.Bind` to a given source instruction, one must choose the subset of the bound variables xs_0 which are erased, by providing some target binders xs_1 and their injection `em` into xs_0. On the contrary, our implementation of the pass on the AS is a function. This implementation is defined by recursion on the syntax of the source instruction and takes as parameters Boolean predicates (figure 7.5b) which specify which variables and operations are erased. In practice, those predicates reflect the annotations present in the source program. In the case of a DCE, the predicates use the `dead` annotations put by the liveness analysis. For a ghost code erasure, we use `ghost` annotations, which are propagated during the compilation pipeline. In all cases, the erasure pass uses its parameters defensively: we do not require any formal specification on the choice of erased variables. The pass fails if this choice is not coherent, that is, if the resulting program would depend on an element (variable or operation) which has been erased. However, in order to justify the premise of `Erasure.Op`, the pass assumes that the predicates

$$\frac{\texttt{xs}_0 \circ \texttt{er} = \texttt{eE.var} \circ \texttt{xs}_1}{\texttt{Erasure.instr_match eE er r_ts}_0\ (\texttt{return (xs}_0\texttt{)})\ (\texttt{return (xs}_1\texttt{)})}\ \texttt{Erasure.Ret}$$

$$\frac{\begin{array}{c}\texttt{em} \text{ is an injection from the } \texttt{xs}_1 \text{ to the } \texttt{xs}_0 \\ \texttt{Erasure.binders_match eE.var em ts}_0\ \texttt{ts}_1 \\ \texttt{Erasure.instr_match eE em (xs}_0\ \texttt{:}\ \texttt{ts}_0\texttt{)}\ \texttt{lhs}_0\ \texttt{lhs}_1 \\ \texttt{Erasure.instr_match [eE, em] er r_ts}_0\ \texttt{rhs}_0\ \texttt{rhs}_1\end{array}}{\begin{array}{l}\texttt{Erasure.instr_match eE er r_ts}_0 \\ \quad \texttt{(let xs}_0\ \texttt{:}\ \texttt{ts}_0\ \texttt{=}\ \texttt{lhs}_0\texttt{;}\ \texttt{rhs}_0\texttt{)}\ \texttt{(let xs}_1\ \texttt{:}\ \texttt{ts}_1\ \texttt{=}\ \texttt{lhs}_1\texttt{;}\ \texttt{rhs}_1\texttt{)}\end{array}}\ \texttt{Erasure.Bind}$$

$$\frac{\texttt{op} \text{ is removable}}{\texttt{Erasure.instr_match eE } \varnothing\ \texttt{r_ts}_0\ \texttt{(op<argT>(argV))}\ \texttt{(return ())}}\ \texttt{Erasure.Op}$$

$$\frac{\begin{array}{c}\texttt{er} \text{ is an injection from the } \texttt{xs}_1 \text{ to the } \texttt{xs}_0 \\ \texttt{er}' \text{ is a function from the formal target results to the } \texttt{xs}_1 \\ \texttt{Erasure.binders_match eE.var er r_ts}_0\ \texttt{r_ts}_1 \\ \texttt{Erasure.instr_match eE er (xs}_0\ \texttt{:}\ \texttt{r_ts}_0\texttt{)}\ \texttt{u}_0\ \texttt{u}_1\end{array}}{\begin{array}{l}\texttt{Erasure.instr_match eE (er} \circ \texttt{er}'\texttt{)}\ \texttt{(xs}_0\ \texttt{:}\ \texttt{r_ts}_0\texttt{)} \\ \quad \texttt{u}_0\ \texttt{(let xs}_1\ \texttt{:}\ \texttt{r_ts}_1\ \texttt{=}\ \texttt{u}_1\texttt{;}\ \texttt{return (er}'\texttt{)}\texttt{)}\end{array}}\ \texttt{Erasure.erase_results}$$

Figure 7.6: Subset of the transformation rules applied to erase instructions

`is_op_erased` only choose to remove operators which are semantically removable according to definition 7.4. We prove that when the implementation successfully generates a target program, its syntax is related to the source syntax by the relations we defined on the DTS. In particular, corresponding instructions are related by `Erasure.instr_match` and programs are related by `Erasure.prog_match`.

We proved that the erasure pass preserves the typing and semantics of the program. Those proofs are carried on the DTS. The proven properties are transferred to the AS thanks to the syntactical relation `Erasure.prog_match` between the source and the target program generated by the implementation. In our definition 7.4 of removable operators, we only consider the semantics of operators applied to well-typed arguments. The correctness of the erasure pass thus assumes that the source program and the inputs are well-typed. The core of the proof is the correctness of the relation `Erasure.instr_match`. Like for the simplification pass of section 7.1, we need to formalize the semantic meaning of the arguments `eE` and `er` of the relation:

Definition 7.5 Given an erasure environment eE between two environments E_0 and E_1 and an injection er of r_1 target formal results into r_0 source formal results, we define the following relations between semantic elements:

- Erasure.venv_match eE ρ_0 ρ_1 holds between two value environments $\rho_0 : E_0.\mathtt{var} \to \mathtt{val}$ and $\rho_1 : E_1.\mathtt{var} \to \mathtt{val}$ if $\rho_1 = \rho_0 \circ \mathtt{eE.var}$.
- Erasure.res_match eE er res_0 res_1 is inductively defined between instruction results res_0 : Res.t E_0.knt r_0 and res_1 : Res.t E_1.knt r_1 by

$$\frac{}{\mathtt{Erasure.res_match\ eE\ er\ (Res.Ret\ vres)\ (Res.Ret\ (vres \circ er))}}$$

$$\frac{(k_0, k_1, \mathtt{eka}) \in \mathtt{eE.knt}}{\mathtt{Erasure.res_match\ eE\ er\ (Res.Knt}\ k_0\ \mathtt{varg)\ (Res.Knt}\ k_1\ \mathtt{(varg} \circ \mathtt{eka))}}$$

- Erasure.fcall_match eE : EventFam.Rel.t (Fct.Call.ev E_0.fct) (Fct.Call.ev E_1.fct) is a relation between internal function calls defined by:

$$\frac{(f_0, f_1, \mathtt{efa}, \mathtt{efr}) \in \mathtt{eE.fct}}{\begin{array}{l}\mathtt{Erasure.fcall_match\ eE\ (Fct.Call.mk}\ f_0\ \mathtt{varg)\ (Fct.Call.mk}\ f_1\ \mathtt{(varg} \circ \mathtt{efa))} \\ \quad (\lambda \mathtt{vres}_0\ \mathtt{vres}_1.\ \mathtt{vres}_1 = \mathtt{vres}_0 \circ \mathtt{efr})\end{array}}$$

Since we prove the preservation of the typing, we also define a relation Erasure.tyenv_match between source and target typing contexts [Compiler.Theory.Pass.Erasure:309]. The correctness of Erasure.instr_match is then stated as:

Lemma 7.2 Let u_0 and u_1 be two instructions related by Erasure.instr_match eE er r_ts_0 u_0 u_1. Let Γ_0 be a source typing context in which u_0 has return types r_ts_0 (i.e. Instr.wt Γ_0 u_0 r_ts_0). Let Γ_1 and r_ts_1 be target typing context and result types matching Γ_0 and r_ts_0 (formally, Erasure.tyenv_match eE Γ_0 Γ_1 and Erasure.binders_match eE.var er r_ts_0 r_ts_1). Then:

- The target instruction u_1 is well-typed, i.e. Instr.wt Γ_1 u_1 r_ts_1.
- The semantics of u_1 refines the semantics of u_0 in matching and well-typed value environments. Formally, for any well-typed source value environments ρ_0 and corresponding target environment ρ_1 (i.e. such that Erasure.venv_match eE ρ_0 ρ_1):

$$[\![u_0]\!]_{\mathrm{instr}}\ \rho_0 \gtrsim_{(\mathtt{Re_ty}\ \Gamma_0;\mathtt{Erasure.fcall_match\ eE}),(\mathtt{Erasure.res_match\ eE\ er})} [\![u_1]\!]_{\mathrm{instr}}\ \rho_1$$

In Rocq, lemma 7.2 is actually the definition of Erasure.instr_match, and the rules presented in figure 7.6 are lemmas on this definition. This specification of the erasure directly relates the semantics of the source and target instructions. It is thus simpler than the specification of the simplification pass (lemma 7.1), in which we had to bind a continuation after the source semantics. This is due to the fact that in the erasure, the target program has the same structure as the source. However, contrary to the simplification pass, the specification of the erasure involves relations on the results and function call events to account for the removal of some variables in results and arguments. We do not use directly Erasure.fcall_match as relation on function calls since we need the additional assumption that functions return well-typed results. We provide this invariant by composing on the source side with the relation Re_ty used in the statement of the type preservation (lemma 6.1). We do not need to specify in the post-condition that instructions return well-typed results and instead recover this property thanks to the soundness of the typing judgment (lemma 6.1).

7.3 Block-sinking

Block sinking [85, chap 6] is an intra-procedural transformation on functional SSA IRs. It moves definitions of local continuations closer to their call points. It can improve the effect of subsequent optimizations. We implemented a block-sinking pass on our parameterized functional SSA IR. Like the simplification pass, this transformation does not depend on any instance-specific parameter, and

```
0:krec {
  entry():
    let b,x0,y0 = {
      1:let c = 1;
      2:fcall f0(c) };
    3:if b H
    then 4:kcall k0(x0)
    else 5:kcall k1(x0,y0)
  k0(x):
     6:let z0 = fcall f1(x);
     7:kcall k2(x,z0)
  k1(x,y):
     8:let z1 = fcall f2(x,y);
     9:kcall k2(x,z1)
  k2(x,z):
    10:let y1 = fcall f3(z);
    11:kcall k1(x,y1)
}()
```

a: Source

```
0:let c = 1;
2:let b,x0,y0 = fcall f0(c);
krec {
  entry():
    3:if b H
    then {
      4:let z0 = fcall f1(x0);
      7:kcall k2(x0,z0)
    } else
      5:kcall k1(x0,y0)
  k1(x,y):
     8:let z1 = fcall f2(x,y);
     9:kcall k2(x,z1)
  k2(x,z):
    10:let y1 = fcall f3(z);
    11:kcall k1(x,y1)
}()
```

b: Target

Figure 7.7: Example of the block sinking pass [compiler/test/src/pass/block_sinking/t7_example]. We annotated locations with numeric labels to link them to intermediate structures used by the implementation.

can be applied in any instance. We use figure 7.7 as a running example. Section 7.3.1 describes the transformation and motivates its use. Like for the erasure, section 7.3.2 captures the effect of the transformation using a syntactic predicate and shows that this predicate implies the preservation of the semantics. Section 7.3.3 then presents our implementation, which is proven to only perform transformations allowed by the predicate.

7.3.1 Motivation and effects

In functional SSA, the nesting of lexical scopes is used as under-approximation of dominance between program points (section 6.1.3). There are cases where this approximation is strict. For instance, in figure 7.7a, the definition of `x0` dominates the whole iteration of `k1` and `k2`, but is not in their scopes. In particular, left-nested `let`s reduce the scopes of variables and lead to under-approximation of dominance. Thus in figure 7.7a, `c` dominates the conditional on `b` but is not in its scope. Those approximations can prevent some optimizations. In figure 7.7, the parameter `x` is loop invariant and equal to the value that was computed in `x0`. However, it cannot be directly removed in figure 7.7a because `x0` is not in the scope of the continuations.

Block sinking [85, chap 6] is a transformation which makes nesting of lexical scopes equivalent to dominance. The transformation leverages the fact that the dominance relation in a given CFG can be represented by a tree, called *dominator tree*. The transformation changes the representation of the control flow so that its structure coincides with the dominator tree. It moves definitions of local continuations closer to their call points without duplicating them. In our block-sinking transformation, "continuations" are either mutually recursive continuations declared in `krec` blocks or right-hand sides of `let`s. We thus consider both `kcall` and normal instruction returns (in particular `return`) as calls to continuations. For instance, in figure 7.7a, the continuations `k0` and `k1` are defined in a `krec` at the top-level of the instruction. In the generated target program (figure 7.7b), their `krec` block is instead at the end of a sequence of two `let`s. A consequence of this transformation is that the `let`-bound variables `c`, `b`, `x0` and `y0` become available in the scope of the `krec`. A hypothetical parameter dropping pass [85, chap 6] could remove `x` from the list of arguments of the continuations. As a byproduct, the pass also flattens sequences: the left-nested `let`s of figure 7.7a become right-nested in figure 7.7b. Another interesting consequence of block-sinking is that it removes unreachable continuations and moves

continuations which are called from exactly one place (like `k0` in figure 7.7a) to their unique call site. As an extra simplification, we inline those continuations. Since `return`s are treated as particular `kcall`s, they are substituted when they appear on the left-hand sides of `let`s. Our block-sinking transformation thus generalizes the simplification pass presented in section 7.1. However, its implementation and proof are significantly more complex.
In our IR, we formulate the equivalence between dominance and lexical nesting with the following implication:

Definition 7.6 In an instruction `u`, *dominance is represented by the syntactic structure* if the following implication holds. For any sub-instructions u_0 and u_1 in `u`, if:
- u_0 dominates u_1
- u_0 is an operator or function call

then u_0 is the left-hand side of a `let`-binding "`let xs =` u_0`; rhs`" whose right-hand side (`rhs`) contains u_1.

We did not prove formally that this property is satisfied in the output of our block-sinking transformation. In fact, it sometimes does not hold in the whole body of the functions. Those exceptions are related to the handling of function termination (section 7.3.3.4), and only affect small parts.

7.3.2 Syntactic specification

We define a syntactic relation `KntMorph.instr_match` between DTS instructions which over-approximates the effect of the block-sinking transformation. In the following, the syntactical relation `KntMorph.instr_match` is written:

$$\mathtt{ctx}_0, \mathtt{ui}_0, \mathtt{kri}_0 \underset{\mathtt{invK}}{\equiv} \mathtt{ctx}_1, \mathtt{ui}_1, \mathtt{kri}_1$$

$\mathtt{ui}_0$ (source) and $\mathtt{ui}_1$ (target) are the instructions compared. They have access to the same variables and functions, but to different scopes $\mathtt{K}_0$ and $\mathtt{K}_1$ of continuations. To unify the handling of normal returns with the continuation calls, we pair instructions with continuation names $\mathtt{kri}_0$ and $\mathtt{kri}_1$ which represent normal returns. That is, the pair $(\mathtt{ui}_0, \mathtt{kri}_0)$ is interpreted like an instruction `let xs = ui`$_0$`; kcall kri`$_0$`(xs)`. Note that this instruction can end with a call to a continuation different from $\mathtt{kri}_0$ if the call is performed by $\mathtt{ui}_0$. We take into account the *contexts* of the instructions: $\mathtt{ctx}_0$ (resp. $\mathtt{ctx}_1$) is a partial function which maps some continuation names of $\mathtt{K}_0$ (resp. $\mathtt{K}_1$) to their definitions. Such definition is given by a tuple $(\mathtt{xs}, \mathtt{u}, \mathtt{kr})$ of a list `xs` of formal arguments, an instruction `u` and a continuation name `kr` for its normal return. The scope of `u` and `kr` is the scope of $\mathtt{ui}_0$ (for $\mathtt{ctx}_0$, $\mathtt{ui}_1$ for $\mathtt{ctx}_1$) extended with the formal arguments `xs` of the continuation. We also parameterize the relation by an *invariant* `invK` which relates source and target continuations from which the systems have the same behaviors. This set `invK` contains pairs of continuation names with the same arities.

Example 7.1 To showcase the use of `KntMorph.instr_match`, we manually proved that the source and target of our running example (figure 7.7) are syntactically related [Compiler.Theory.Pass.KntMorph_Test:153]. As an example of `KntMorph.instr_match`, here is an extract of the sub-goal we had to prove to justify the equivalence between the `then` branches:

$$\mathtt{ctx}_0, \mathtt{kcall\ k0}^{\mathtt{S}}\mathtt{(x0)}, \mathtt{ke}^{\mathtt{S}} \underset{\{(\mathtt{k2}^{\mathtt{S}}, \mathtt{k2}^{\mathtt{T}}), \ldots\}}{\equiv} [\ldots], \begin{array}{l}\mathtt{let\ z0 = fcall\ f1(x0);}\\ \mathtt{kcall\ k2}^{\mathtt{T}}\mathtt{(x0, z0)}\end{array}, \mathtt{ke}^{\mathtt{T}}$$

where $\mathtt{ctx}_0\ \mathtt{k0}^{\mathtt{S}} := ((\mathtt{x}), (\mathtt{let\ z0 = fcall\ f1(x);\ kcall\ k2}^{\mathtt{S}}\mathtt{(x, z0)}), \mathtt{ke}^{\mathtt{S}})$
The free variables and functions (here `x0` and `f1`) are common between the source and target. On the contrary, we formally distinguish source and target continuations. The continuations named `k2` in the source and target thus have two distinct names $\mathtt{k2}^{\mathtt{S}}$ and $\mathtt{k2}^{\mathtt{T}}$. The names $\mathtt{ke}^{\mathtt{S}}$ and $\mathtt{ke}^{\mathtt{T}}$ represent the exits of the functions; here they are only used as placeholders return continuations since the instructions always perform some `kcall`. At this point in the proof, the two names of `k2` have been related and added to the invariant. However, the invariant alone is not enough to relate the source and target instructions. Indeed, the former calls the continuation $\mathtt{k0}^{\mathtt{S}}$ while the latter calls the function `f1` then the continuation $\mathtt{k2}^{\mathtt{T}}$. The proof of this goal thus relies on the definition of $\mathtt{k0}^{\mathtt{S}}$, which is given in the source context. As we will see, the syntactic relation `KntMorph.instr_match` allows inlining continuation calls.

The intended meaning of `KntMorph.instr_match` is a correspondence between prefixes of the executions of the source and target. Intuitively, we interpret a triple $(\mathtt{ctx}, \mathtt{ui}, \mathtt{kri})$ as a system:

```
krec{
  entry(): let xs = ui; kcall kri(xs)
  k0(arg0): let xs = u0; kcall kr0(xs)
  k1(arg1): let xs = u1; kcall kr1(xs)
  ...
}()
```

where $(\mathtt{arg}_i, \mathtt{u}_i, \mathtt{kr}_i)$ is the definition associated to $\mathtt{k}_i$ by `ctx`. This `krec` defines a kind of transition system. A state of such system is either the initial state (`entry`) or a call to a continuation (that is, a continuation name and values for its arguments). Each transition executes an instruction which

ends with a call to a continuation. The next state is defined by this terminal call. Those transitions depend on a constant value environment for the free variables of the system. The initial transition executes `let xs = ui; kcall kri(xs)`. From a state corresponding to a call to a continuation `k`, the system can perform a transition if `ctx` contains a definition for `k`. In this case, the transition executes the instruction associated to `k`. We emphasize that the instructions associated to the transitions can be arbitrarily complex. In particular, they can contain some `krec` and diverge. Intuitively the relation `KntMorph.instr_match` implies that for any complete execution of the target system, there exists:
- a prefix of the target iteration, which can be infinite
- a partial or complete execution of the source

such that the two executions have the same effects. Moreover, if the executions are finite, they end with two continuation calls k_0 and k_1 with the same arguments and such that $(k_0, k_1) \in$ `invK`.

The syntactic relation `KntMorph.instr_match` is defined inductively. Figure 7.8 presents a subset of this definition. Since in our IR continuations can either be declared with `let` or `krec`, we define two auxiliary predicates which recognize continuation calls and declarations. The predicate `instr_is_kcall (u, kr) k arg` is satisfied by instructions `let xs = u; kcall k(xs)` which are equivalent to a continuation call `kcall k(arg)`. The rule `KCallRet` recognizes a `return` as a call to the normal return continuation `kr`. The rule `KCallKCall` recognizes the case where `u` is a `kcall` and the call to `kr` is unreachable. The predicate `instr_is_kont (u, kr) K dK (u', kr')` is satisfied by instructions `let xs = u; kcall kr(xs)` which are equivalent to a declaration of a set `K` of continuations around an instruction `let xs' = u'; kcall kr'(xs')`. The function `dK` describes the definitions of the continuations in `K`. The rule `KontBind` interprets a `let` as the definition of a single continuation `krhs` around the left-hand side. The blocks of continuations `krec` of our IR start with an initial call rather than by executing an instruction. The rule `KontKRec` thus interprets generic `krec` blocks as declarations of some continuations around a `kcall` to the initial continuation k_0 (we use `kr` as a placeholder return continuation after this call).

TODO (rocq)

Est-ce qu'on peut dériver `KntMorph.KontKRecIni` ?

In the case where the initial continuation is not recursive, `KontKRecIni` instead interprets `krec` as the declaration of continuations around the initial `entry` continuation.
The definition of `KntMorph.instr_match` uses the predicates `instr_is_kcall` and `instr_is_kont` in the source and target. Most rules follow the structure of the source and target instructions. The rule `MatchKCall` relates calls to two continuations in `invK` with the same arguments. When the source instruction declares continuations, the rule `MatchKont0` moves those declarations to the context. There is a similar rule for the target. Instructions which always terminate normally (operators and function calls) are matched by `MatchBasic` when they appear both in the source and target. Corresponding conditionals are related by `MatchBranch` if their branches are related. There are also rules which are not guided by the structures of the instructions. Continuation calls can be inlined in the source (`MatchInline0`) and target instead of being matched with a corresponding call. Once corresponding source and target continuations have been moved to the context, the rule `MatchAugment` is used to add them to the invariant `invK`. The first premise requires that any pair added to the invariant (i.e. in `R`) contains continuations with definitions in their respective contexts and such that those definitions are related by `KntMorph.instr_match`. This comparison is performed with the augmented invariant, which enables proving the relation between infinite loops of continuations.

The syntactic relation `KntMorph.instr_match` captures strictly more transformations than those actually performed by block-sinking. For instance, it enables duplicating continuations, and does not

guarantee the equivalence between dominance and lexical scopes in the target. In fact, it is symmetric with respect to a swap of the source and target.

instr_is_kcall

$$\frac{}{\texttt{instr_is_kcall}\ (\texttt{return (arg)}, \texttt{kr})\ \texttt{kr}\ \texttt{arg}}\ \textsc{KCallRet}$$

$$\frac{}{\texttt{instr_is_kcall}\ (\texttt{kcall k(arg)}, \texttt{kr})\ \texttt{k}\ \texttt{arg}}\ \textsc{KCallKCall}$$

instr_is_kont

$$\frac{\texttt{krhs} \text{ is a fresh continuation name}}{\texttt{instr_is_kont}\ (\texttt{let xs = lhs; rhs}, \texttt{kr})\ \{\texttt{krhs}\}\ [\texttt{krhs} \mapsto (\texttt{xs}, \texttt{rhs}, \texttt{kr})]\ (\texttt{lhs}, \texttt{krhs})}\ \textsc{KontBind}$$

$$\frac{}{\begin{array}{l}\texttt{instr_is_kont}\ (\texttt{krec\{k}_0\texttt{(xs}_0\texttt{): u}_0\texttt{; ... k}_n\texttt{(xs}_n\texttt{): u}_n\texttt{\}(args), kr})\\ \quad \{\texttt{k}_i \mid i \in [0; n]\}\ [\texttt{k}_i \mapsto (\texttt{xs}_i, \texttt{u}_i, \texttt{kr})]\ (\texttt{kcall k}_0\texttt{(args), kr})\end{array}}\ \textsc{KontKRec}$$

$$\frac{\texttt{entry} \text{ is not called by } \texttt{u} \text{ nor by any } \texttt{u}_i}{\begin{array}{l}\texttt{instr_is_kont}\ (\texttt{krec\{entry(): u; k}_0\texttt{(xs}_0\texttt{): u}_0\texttt{; ... k}_n\texttt{(xs}_n\texttt{): u}_n\texttt{\}(), kr})\\ \quad \{\texttt{k}_i \mid i \in [0; n]\}\ [\texttt{k}_i \mapsto (\texttt{xs}_i, \texttt{u}_i, \texttt{kr})]\ (\texttt{u}, \texttt{kr})\end{array}}\ \textsc{KontKRecIni}$$

KntMorph.instr_match

$$\frac{\begin{array}{c}\texttt{instr_is_kcall}\ (\texttt{ui}_0, \texttt{kri}_0)\ \texttt{k}_0\ \texttt{arg} \quad \texttt{instr_is_kcall}\ (\texttt{ui}_1, \texttt{kri}_1)\ \texttt{k}_1\ \texttt{arg}\\ (\texttt{k}_0, \texttt{k}_1) \in \texttt{invK}\end{array}}{\texttt{ctx}_0, \texttt{ui}_0, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{ui}_1, \texttt{kri}_1}\ \textsc{MatchKCall}$$

$$\frac{\begin{array}{c}\texttt{instr_is_kcall}\ (\texttt{ui}_0, \texttt{kri}_0)\ \texttt{k}\ \texttt{arg} \quad \texttt{ctx}_0\ \texttt{k} = \lfloor \texttt{xarg}, \texttt{u}, \texttt{kr} \rfloor\\ \texttt{ctx}_0, \texttt{u[arg/xarg]}, \texttt{kr} \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{ui}_1, \texttt{kri}_1\end{array}}{\texttt{ctx}_0, \texttt{ui}_0, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{ui}_1, \texttt{kri}_1}\ \textsc{MatchInline0}$$

$$\frac{\begin{array}{c}\texttt{instr_is_kont}\ (\texttt{ui}_0, \texttt{kri}_0)\ \texttt{K}\ \texttt{dK}\ (\texttt{ui}_0', \texttt{kri}_0')\\ [\texttt{ctx}_0, \texttt{dK}], \texttt{ui}_0', \texttt{kri}_0' \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{ui}_1, \texttt{kri}_1\end{array}}{\texttt{ctx}_0, \texttt{ui}_0, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{ui}_1, \texttt{kri}_1}\ \textsc{MatchKont0}$$

$$\frac{\texttt{ui} \text{ does not contain continuation calls} \quad (\texttt{kri}_0, \texttt{kri}_1) \in \texttt{invK}}{\texttt{ctx}_0, \texttt{ui}, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{ui}, \texttt{kri}_1}\ \textsc{MatchBasic}$$

$$\frac{\texttt{ctx}_0, \texttt{thn}_0, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{thn}_1, \texttt{kri}_1 \quad \texttt{ctx}_0, \texttt{els}_0, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{els}_1, \texttt{kri}_1}{\texttt{ctx}_0, \texttt{if cnd H then thn}_0 \texttt{ else els}_0, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{if cnd H then thn}_1 \texttt{ else els}_1, \texttt{kri}_1}\ \textsc{MatchBranch}$$

$$\frac{\begin{array}{l}\forall (\texttt{k}_0, \texttt{k}_1) \in \texttt{R}.\ \exists \texttt{xs}\ \texttt{u}_0\ \texttt{kr}_0\ \texttt{u}_1\ \texttt{kr}_1.\\ \quad \texttt{ctx}_0\ \texttt{k}_0 = \lfloor \texttt{xs}, \texttt{u}_0, \texttt{kr}_0 \rfloor \wedge \texttt{ctx}_1\ \texttt{k}_1 = \lfloor \texttt{xs}, \texttt{u}_1, \texttt{kr}_1 \rfloor \wedge \texttt{ctx}_0, \texttt{u}_0, \texttt{kr}_0 \underset{\texttt{invK} \cup \texttt{R}}{\equiv} \texttt{ctx}_1, \texttt{u}_1, \texttt{kr}_1\\ \texttt{ctx}_0, \texttt{ui}_0, \texttt{kri}_0 \underset{\texttt{invK} \cup \texttt{R}}{\equiv} \texttt{ctx}_1, \texttt{ui}_1, \texttt{kri}_1\end{array}}{\texttt{ctx}_0, \texttt{ui}_0, \texttt{kri}_0 \underset{\texttt{invK}}{\equiv} \texttt{ctx}_1, \texttt{ui}_1, \texttt{kri}_1}\ \textsc{MatchAugment}$$

Figure 7.8: Extract of the inductive definition of `KntMorph.instr_match`. This syntactic relation over-approximates the effect of block-sinking.

We derive a syntactic relation between functions:

Definition 7.7 The relation `KntMorph.func_match` between function declarations is defined by:

$$\frac{\varnothing, \texttt{body}_0, \texttt{ke}^S \underset{\{(\texttt{ke}^S,\texttt{ke}^T)\}}{\equiv} \varnothing, \texttt{body}_1, \texttt{ke}^T}{\texttt{KntMorph.func_match}\ (\texttt{fn f(arg) : (res) \{body}_0\texttt{\}})\ (\texttt{fn f(arg) : (res) \{body}_1\texttt{\}})}$$

That is, a source function $\texttt{fn}_0$ is related to a target function $\texttt{fn}_1$ by $\texttt{KntMorph.func_match}\ \texttt{fn}_0\ \texttt{fn}_1$ if they have the same signature and if their bodies are related by `KntMorph.instr_match`. In this premise, we introduce two continuation names $\texttt{ke}^S$ and $\texttt{ke}^T$ to represent normal returns, which terminate the functions. Those continuations do not have definitions in the contexts (which are empty), but they are related by the initial invariant $\{(\texttt{ke}^S, \texttt{ke}^T)\}$. We prove that this syntactic specification implies the semantic equivalence of the functions.

Lemma 7.3 If two functions $\texttt{fn}_0$ and $\texttt{fn}_1$ are related by `KntMorph.func_match`, then their semantics are equivalent, that is:

$$\forall \texttt{varg}.\ [\![\texttt{fn}_0]\!]_{\text{func}}\ \texttt{varg} \approx [\![\texttt{fn}_1]\!]_{\text{func}}\ \texttt{varg}$$

Idea of the proof We use the codiagonal property of `iter`, and the approach described in section 4.4.1. We do not use the framework `EqIter` (section 4.4.1.3). Indeed, although both instructions have similar CFGs, we do not know in advance where "stuttering states" must be introduced. Instead, our proof builds this CFG by induction on `KntMorph.instr_match` and simultaneously proves equivalences between the semantics and prefixes of executions of the graph. We reuse the formalization `Prefix.t` introduced for `EqIter`. In fact, this approach is not specific to the syntax of our parameterized representation. We thus implemented it as a framework `EqIter2` in `SMonad`. It enables proving equivalences between two semantics involving `iter`, and does not require providing an intermediate graph up-front.

Our syntactic specification does not ensure the preservation of the typing. In particular, it allows arbitrary (possibly ill-typed) code in unreachable target continuations. However, it over-approximates the effect of block-sinking and we expect that our implementation preserves typing.

7.3.3 Implementation

In section 7.3.2, we gave a syntactic relation which over-approximates block-sinking transformations on the DTS. We implemented block-sinking as a function on the AS. Since it is an intra-procedural pass, this function takes as argument the body of a source function and outputs a target instruction. As depicted in figure 7.9, our implementation performs four steps:

1. We generate a continuation graph which encodes the control flow of the source function (section 7.3.3.1).
2. We compute the dominator tree of this graph using the algorithm of Lengauer and Tarjan [72] (section 7.3.3.2).
3. We annotate this tree with a choice of syntactical combinators that will be used in the target. The result, called target tree, has the same structure as the dominator tree (section 7.3.3.3).
4. We generate the target instruction by following the target tree (section 7.3.3.4).

Section 7.3.3.5 describes our strategy for proving the correctness of the implementation. Our implementation uses the results of the second and third steps defensively, so this proof only involves the first and last step.

7.3.3.1 Continuation graph of the source inductive instructions

The first step of our implementation collects the control-flow of the inductive source instruction and represents it as a graph (e.g. figure 7.10a for our running example). We call those graphs *continuation graphs*. We introduced them as an adaption of the notion of CFG for our functional SSA representation. A continuation graph contains a set of nodes corresponding to locations in the source instruction. Each

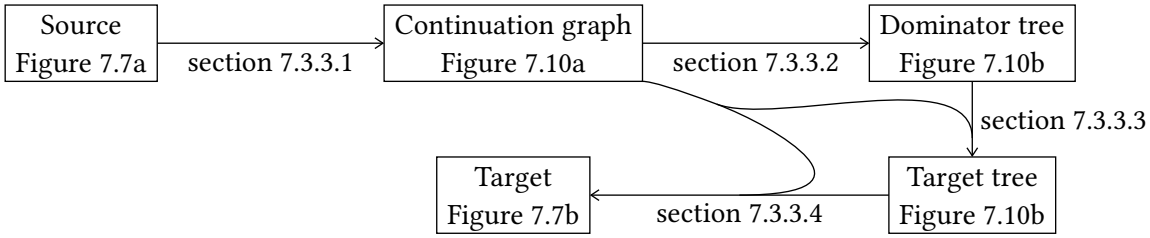
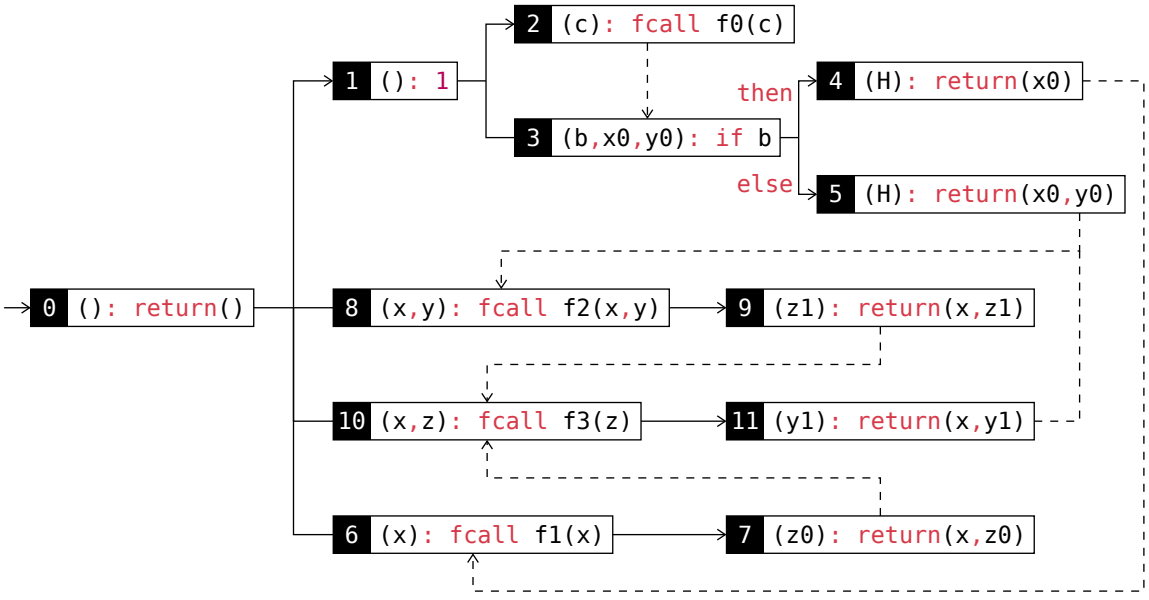
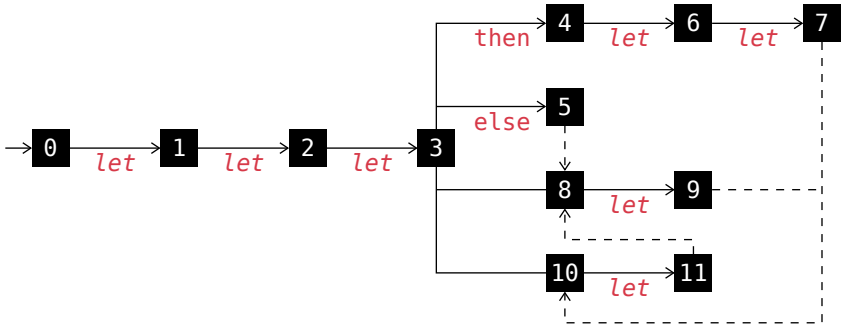


Figure 7.9: Implementation of the block-sinking pass as a sequence of four steps



a: Continuation graph of the source



b: Dominator tree and *choice of* `let`*s in the target*

Figure 7.10: Intermediate structures used by the implementation of the block-sinking on the example of figure 7.7

node is identified by a numeric label. In figure 7.7a, we reported the labels of the nodes corresponding to each location in the instruction. Those annotations do not exist in our implementation. In fact, we do not need to remember the link between the source instruction and its graph. Indeed, we store directly in the continuation graph all the information needed to generate the target instruction. Each node of the graph is a continuation, defined by a list of formal arguments and a simple instruction. The possible instructions are the monadic `return`, calls to functions (`fcall`) and operators (e.g. the constant `1` in figure 7.10a) and `if` conditionals. The continuation graph includes the Control-Flow Graph (CFG) of the instruction it represents. In figure 7.10a, the CFG is represented with directed edges. Most nodes have a single outgoing edge, pointing toward their successor in the execution. The

intended semantics is to bound the values returned by the instruction to the formal arguments of the successor. For instance, the successor of node 2 is the node 3. Indeed, in the source instruction (figure 7.7a), the results of the function call to `f0` are bound to the variables `(b, x0, y0)` and the execution continues with the conditional. The `if` instruction has two successors, corresponding to the two branches. There can also be exit edges, which terminate the function (none in figure 7.10a, since it never terminates).

Variables have the same scope in the continuation graph and in the source instruction. To track those scopes, the continuation graph includes not only the CFG but also a tree corresponding to the nesting structure of the lexical scopes. In figure 7.10a, this tree is defined by the solid lines and drawn from left to right. Some edges belong both to the CFG and to the tree (e.g. 0 to 1) but some edges only belong to the tree (e.g. 0 to 6) or to the CFG (e.g. 4 to 6). An instruction has access to all variables bound in its ancestors in the tree. For instance, the `return` instruction with label 7 has access to the variable `x` bound at label 6, but not to the `x0` bound at label 3. Using scoped variables allows us to keep their representation with De Bruijn indices. On the contrary, continuation graphs switch to a named representation (the numeric labels) for the continuations.

The continuation graph is generated by recursion on the source inductive instruction. In the graph, `return` instructions are used both for `return` and `kcall`. For instance, the `kcall k0` at location 4 in figure 7.7a leads to a `return` with the same argument in figure 7.10a. The successor of this `return` is the first instruction (labelled 6) of the callee `k0`. The combinators `let` and `krec` do not have corresponding instructions in the continuation graph. Indeed, their effect is mostly to declare continuations. They are encoded using CFG and tree edges. The initial call of `krec` generates a `return` (e.g. label 0 in figure 7.10a).

7.3.3.2 Computing the dominator tree

The goal of the block-sinking pass is that the nesting of lexical scopes in the target coincides with the dominance relation between program points. To this end, we compute the dominator tree of the source CFG.

Definition 7.8 The *dominator tree* of a CFG is a rooted tree containing its reachable nodes, and such that a node j is dominated by i in the CFG if and only if i is an ancestor of j in the dominator tree.

Note that the concepts of reachability and dominance only depend on the static graph of the CFG, not on the semantics of the nodes. Any CFG has a unique dominator tree. The dominator tree of our running example is drawn in figure 7.10b, with solid edges and from left to right.

We compute the dominator tree of the CFG contained in the continuation graph of the source. That is, in figure 7.10a, we only look at the directed edges. The content of the nodes is also ignored. We use the algorithm of Lengauer and Tarjan [72]. We implemented the algorithm in Rocq, but only proved its termination. That is, we did not prove any formal specification on the result of our implementation. We instead only use the dominator tree defensively in the next steps of the pass. This algorithm depends on an auxiliary structure involving forests with path compression [98]. This structure relies on imperative updates, which are not possible in Rocq. We replaced those imperative updates by the use of a functional lookup table, following the implementation of the union-find in CompCert. Since our implementation uses the dominator tree defensively, we could have instead used an oracle implemented in OCaml. Despite its relative inefficiency, an advantage of our implementation is that it can be executed in Rocq on small examples, and could be verified in future works.

We are not the first to use dominator trees in a verified compiler. We implemented the standard algorithm of Lengauer and Tarjan [72] in Rocq, but did not prove that it produces the dominator tree. Zhao et al. [108] instead implemented and proved in Rocq a simpler algorithm, but with a higher

time complexity. CompCertSSA [12, §5] uses an implementation of the algorithm of Lengauer and Tarjan in OCaml to generate the SSA form of programs. This generation pass is validated defensively. Léchenet [63] implemented and proved correct a validator of dominator trees.

7.3.3.3 Choice of the structure of the target

The goal of the third step is to choose the syntactical combinator which will be used to declare continuations in the target. We recall that several syntactic constructions of our IR can declare the "continuations" used in continuation graphs. Indeed, continuations are either introduced as the right-hand sides of `let`s, or within `krec` blocks. We could declare all continuations with `krec` in the target, but the resulting programs would contain deeply nested `krec` blocks. The results of operations and function calls would also not be directly `let`-bound in the instructions they dominate, that is definition 7.6 would not hold. On the contrary, using `let`s defines sequences of instructions. In our compilation pipeline, the main interest of using `let`s is to produce more readable programs. In the future, this could also trigger more optimizations by hypothetical passes which operate over sequences of instructions (e.g. peephole optimizations). However, there are restrictions on the continuations introduced using `let`: they cannot be recursive and only one such continuation is accessible (with normal returns) from a given program point.

The third step traverses the dominator tree and produces a *target tree*. This target tree has the same structure as the dominator tree, but each node is annotated with the kind of syntactical construct used to declare its children (`let`, `krec` or none if no continuations are declared around this node). In figure 7.10b, we directly reported the *`let`* annotations on the dominator tree. We also separate the continuations corresponding to branches of conditionals (which are necessarily direct children of their `if` in the dominator tree) from the other continuations. For instance, figure 7.10b indicates that:

- we will use a `let` to define the continuation 3 around the location 2
- the continuations 4 and 5 are the branches of the conditional 3, and we will use `krec` to define the continuations 8 and 10 around it

Our algorithm tries to maximize the length of the sequences of `let`s using a greedy heuristic. We traverse the dominator tree from the leaves to the root (i.e. right to left in figure 7.10b). For each node i:

- If i is recorded as an `if` in the continuation graph, we look for its branches among its children and separate them.
- We annotate i with `let` if all the following conditions hold:
  - i has a single child j (excluding the branches)
  - j is not recursive, that is, all calls to j are performed by i or its branches
  - we have not yet chosen to use a `let` around a subtree containing a call to j

For definition 7.6 to hold, any operation or function call i with children must be annotated with `let`. Our heuristic guarantees this property. Indeed, since operations and function calls terminate by returning normally, the unique successor j of i is necessarily `let`-bound in the source. It is thus not recursive. Moreover, lemma 7.4 below ensures that i is either a leaf or j is its only child in the dominator tree. This implies that our heuristic annotates i with `let`.

Lemma 7.4 Let i be a node with a single successor j in the CFG. Then, i has at most one child in the dominator tree, and this child is necessarily j.

Proof (not mechanised in Rocq) Assume for contradiction that i has a child $k \neq j$. By the definition of the dominator tree, i strictly dominates k (i.e. i dominates k and $k \neq i$). Since j is the only successor of i, j also dominates k. Hence, j is an ancestor of k in the dominator tree, and thus a strict ancestor of i (since $j \notin \{i, k\}$). Thus i does not dominate j and there is a path p_0 from the root r of the CFG to j which does not contain i. Let p be a path from r to k. Since i dominates k and j is the only successor of i, we can decompose p as $r \longrightarrow^* i \longrightarrow j \underset{p_1}{\longrightarrow^*} k$, where p_1 does not contain i. Composing p_0 with

p_1 yields a path from the root to k which does not contain i. This contradicts the fact that i dominates k. □

7.3.3.4 Generation of the target instruction

Finally, we generate the target instruction from the continuation graph and the target tree. We annotated figure 7.7b, with labels of the continuations corresponding to each program location. We follow the structure of the target tree (which has the same structure as the dominator tree) from its root to its leaves. We use the continuation graph to retrieve the instructions associated to the labels. Because we represent variables with De Bruijn indices, their representation changes between the source and target. Our implementation thus maintains a map from source to target variables. The existence of such a map relies on the fact that the transformation only moves instructions downward, which have access to more variables in the target than in the source.

We use the patterns recognized by `instr_is_kont` to declare continuations around instructions. The choice of pattern depends on the annotations left by the third step in the target tree. By default, we emit the patterns recognized by `KontBind` (for `let` annotations) and by `KontKRecIni` (for `krec` annotations). We perform some simplification when binding continuations around a continuation graph instruction `return`:

- we inline `let`-bound continuations
- we use the pattern of `KontKRec` to declare continuations with `krec` without generating an `entry` continuation

We chose to perform those simplifications directly during the block-sinking pass rather than as a separate pass (for instance as an extension of the simplification pass of section 7.1), because it does not require changing the invariant of the pass.

In some examples [compiler/test/src/pass/block_sinking/t3_exit], an instruction that was terminating the function by returning normally is moved within the left-hand side of a `let`. From this point, the instruction can no longer terminate directly the function. Indeed, a normal return continues with the right-hand side instead of terminating the function. When necessary, we thus declare using `krec` a continuation `exit` at the top-level of the function:

```
fn f(args) : (results) {
  krec {
    entry() : … (* function body *)
    exit(results) : return (results)
  }
}
```

This continuation terminates the function using a normal `return`. It can be called from any point of the body using `kcall`. Since the `exit` continuation is declared at the top-level of the function, it does not have access to the variables defined by instructions that dominate it. We instead only expect definition 7.6 to be satisfied in the instruction defining `entry`.

The generation of the target instruction uses the target tree defensively. In particular, it will fail in the following cases:

- If the successor of a node is a continuation which has not yet been declared in the target. To avoid such situation, the targets of CFG edges (directed edges in figure 7.10b) must be direct children of an ancestor of their source (included). This condition is ensured by the definition 7.8 of the dominator tree.
- If a continuation i is declared higher in the target than in the source. That is, if a node j is an ancestor of i in the tree of the source, but not in the dominator tree. If that happened, the continuation i

would have access to fewer variables in the target. This situation does not occur because the source tree is an under-approximation of the dominance.
- If the successor of a node i is `let`-bound, but a normal return from i continues with another `let`. The heuristic of the third step avoids those situations.

7.3.3.5 Correctness of the implementation

We prove that the generated target function is related to the source by the syntactical specification we defined in section 7.3.2. By lemma 7.3, this implies that the source and target are semantically equivalent. Although the second and third steps of the pass are implemented in Rocq, we did not prove any correctness property on the target tree they generate. We thus prove that if:
- the first step generates a continuation graph `G` from a source function $\texttt{fn}_0$
- the last step successfully generates a target function $\texttt{fn}_1$ from `G` and an arbitrary target tree

Then $\texttt{fn}_0$ and $\texttt{fn}_1$ are related by `KntMorph.func_match`. For this proof we used the following approach:
- we define a syntactic relation between the source $\texttt{fn}_0$ and the continuation graph `G`

> **TODO** (rocq)
>
> [Compiler.Impl.Pass.BlockSinking.KntGraph:585] combiner les définitions

- we prove that the first step generates a graph `G` satisfying this syntactic specification
- we prove that given a graph `G` syntactically related to some function $\texttt{fn}_0$, the last step generates a target $\texttt{fn}_1$ syntactically related to $\texttt{fn}_0$

This approach was motivated by the fact that the syntactic relations we defined in section 7.3.2 directly relate the source and target syntaxes. Since those relations do not mention the intermediate continuation graph, we chose to only consider the graph as an alternative syntactic representation of the source. A drawback of this approach is that the proof of the last step involves three objects: the source program, the continuation graph and the target program. Another burden we faced in our proofs is related to the representation of variables and continuations using well-scoped De Bruijn indices in our syntactic relation. In the relation `KntMorph.instr_match`, a single type `V` represents the free variables of the systems. Hence, both the instructions under focus and the definitions of the continuations in the contexts are represented using some `Instr.s E _` with `E.var` $=$ `V`. Similarly, the family $\texttt{K}_0$ (resp. $\texttt{K}_1$) of continuations is used by the whole source (resp. target) system. Issues arise when we introduce new symbols for a subset of the systems. For instance, the rule `MatchKont0` moves a set `K` of continuations from the source instruction $\texttt{ui}_0$ under focus to the context. In the premise, those continuations are only called by subparts of $\texttt{ui}_0$, that is `dK`, $\texttt{ui}_0'$ and $\texttt{kri}_0'$; not by $\texttt{ctx}_0$. However, the fact that we use a single set of continuations implies that $\texttt{ctx}_0$ has access to those newly introduced continuations in the premise. The type representing the continuations free in $\texttt{ctx}_0$ thus changes between the conclusion and the premise. In Rocq, we thus have to reindex the references occurring in $\texttt{ctx}_0$. Those reindexings are not performed in the implementation, thanks to the use of reversed De Bruijn indices and to the map from source to target variables. However, we still have to reason on the reindexings when proving invariants between the theoretical representations of the systems and the instructions and continuation graphs used in the implementation.
The implementation of the last step does not take as argument the source program, only its continuation graph (contrary to the proof of correctness of this step). This suggests that we could have instead defined a semantics of continuation graphs. Using names instead of De Bruijn indices for variables in the continuation graph could also simplify the implementation and proofs. Following Schneider [91], we could then maybe use a semantics based on assignments instead of maintaining the structure of the source to handle the scopes of the variables.
Another approach could be to implement a validator which checks the syntactic relations of

section 7.3.2 directly between the source and target programs, independently of the intermediate continuation graph.

7.4 Type-checker

The well-typedness of programs (section 6.5) is an invariant of our compilation pipelines. Since we are interested in intermediate representations rather than in source languages, we could assume that the input programs of our pipelines are well-typed. To preserve this well-typedness, all our passes must produce well-typed target programs when applied to well-typed source programs. However, we did not prove formally this property for all passes. For instance we only proved the preservation of well-formedness and of semantics for the simplification (section 7.1) and block-sinking (section 7.3) passes. On the other hand, the semantic correctness of some passes (e.g. the erasure of section 7.2.3) formally relies on the assumption that their input programs are well-typed. Instead of justifying once and for all those assumptions by formalizing proofs of well-typedness preservation, we verify defensively that programs are well-typed using a type-checker. A type-checker is a pass which takes as argument a program and fails if it is ill-typed. We implemented a formally verified type-checker on the parameterized IR. This type-checker also checks that its input program is well-formed (e.g. that there are no unbound variables). Those well-formedness checks are only useful to detect errors in programs manually written in our IRs.

The type-checker is implemented on the AS. It is parameterized by the following instance-specific functions:

- A test `check_typ_sym (ts : Typ.sym_t) : res unit`, which checks that a given AS type symbol `ts` is well-formed. That is, it should succeed only if there exists a corresponding DTS type symbol. For instance, in Frag IR, this function checks that the abstract predicates have been declared.
- A comparison `check_typ_sub (ty0 ty1 : Typ.t) : res unit` which checks the sub-typing relation between two types.
- The signatures of the operators, given by `get_op_ty (op : Op.sym_t) : res Op.Sig.t`. The type `Op.Sig.t` contains the type of the arguments and results of the operator. It is the representation of `Op.Sig.s` in the implementation (that is, using AS instead of DTS). The types in operator signatures contain variables representing the type and value arguments and the results of the operator.

All those parameters use the result monad `res` to raise errors if the checks fail or if they receive ill-formed arguments. Those parameters must be sound with respect to the parameters used in the typing judgment.

The generic part of the type-checker traverses recursively the syntax of programs while maintaining the typing environment. It uses its parameter `check_typ_sub` to compare individual types, after having performed the substitutions described in figure 6.7.

We give the following specification to the type-checker:

Lemma 7.5 Let `t_pr` be an AS program with a DTS representation `s_pr`, that is, `Prog.impl t_pr s_pr`. If the type-checker accepts `t_pr`, then `s_pr` is well-typed, that is, `Prog.wt s_pr`.

This specification assumes the existence of a DTS representation of the program, and thus that it is well-formed. In our compilation pipelines, we always prove formally that the programs are well-formed, and are thus able to use this specification. We also gave a specification which reflects those well-formedness checks [Compiler.Impl.Pass.TypCheck:1019], but we do not use it in our pipelines, for the reason explained in section 6.3.4.

Part III
Propagation of information

8

Definition of Frag IR

Chapter 3 informally described Frag IR, an intermediate representation we designed to propagate semantic information in a compilation pipeline. This chapter is a more formal presentation of the language. In particular, we use the `SMonad` library (chapter 4) to define its semantics. Frag IR is defined in Rocq as an instance of our parameterized SSA IR (chapter 6). Large parts of the implementation and meta-theory of Frag IR are thus inherited from it. Another benefit is that the parameterized passes described in chapter 7 are available on Frag IR.

Section 8.1 describes the formal definitions used to instantiate the parameterized IR to obtain Frag IR. Section 8.2 discusses the effects of the language, and how they affect proofs using `SMonad`. Section 8.3 describes implementations required to instantiate the parameterized passes on Frag IR. Finally, section 8.4 shows how we use small Frag IR programs to test the language and the passes. We describe our larger examples, which involve arrays and linked lists.

8.1 Instantiation of the parameterized IR

This section describes the formal definition of Frag IR, as an instance of the parameterized IR. Our goal is twofold: we give a precise meaning to the concepts introduced in chapter 3 and we illustrate the use of the parameterized IR. We follow the structure of the Rocq development, by first defining the types and values (section 8.1.1), then the domain of the semantics (section 8.1.2) and finally the operators and their semantics (section 8.1.3).

8.1.1 Types and values

Figure 8.1 and figure 8.2 reproduce the formal definitions used to instantiate the sets of values and type symbols for Frag IR. We describe the different cases in the following subsections. The set `val` is only used in the meta-theory, it is not extracted. In the parameterized IR, types `Typ.t` (in the AS) and `Typ.s` (in the DTS) are expressions containing variables and applications of type symbols to sub-expressions. For instance, the type "`b = true`" is represented in the AS by the expression:

```
Typ.Op (Tprop Peqp) [Typ.Var b;Typ.Op (Tscalar_op (Const (Bool true))) []]
```

The type symbols and their semantics are defined by the instance. For Frag IR, we define inductively the type symbols in the AS (figure 8.2), but directly identify them with their semantics in the DTS. That

```
val :=
  | Vsc (s : Scalar.t)                                          scalar value
  | Vopt (v : option val)                                       optional value
  | Vstrc (vs : list val)                                       memory structure node
  | Vcell (ty : Scalar.Ty.t) (p : Ptr.t) (v : Byte.t^sizeof ty) memory cell
  | Vmem (h : heaplet)                                          unstructured memory fragment
```

Figure 8.1: Values in Frag IR

```
Typ.sym_t :=
  – Types
  | Tscalar (ty : Scalar.Ty.t)      scalar type
  | Tprop (pP : tprop_sym)          proposition types
  | Topt                            option type opt
  | Tstrc (s : Structure.t)         structured fragment types
  | Tmem (mP : tmem_sym)            unstructured fragment types
  | Tany                            type containing all values
  – Values
  | Tscalar_op (op : Scalar.Op.t)   result of a scalar operation
  | Tsome | Tnone                   optional values
  | Tderef                          content of a memory cell
  | Tmkstrc                         memory structure node
```

Figure 8.2: Type symbols in the AS of Frag IR

is, in the DTS, the type symbols are functions taking as arguments the evaluation of their arguments and returning the result of their application:

$$\texttt{Typ.sym_s} := \texttt{list tsem} \to \texttt{tsem}$$

where a `tsem` is the evaluation of a type expression. Since types are dependent, type expressions may evaluate to a value or to a set of values (i.e. a type). We also add a case `TErr` to represent errors during the evaluation of operators within type expressions.

$$\texttt{tsem} := \texttt{TVal (v : val)} \mid \texttt{TSet (s : val} \to \texttt{Prop)} \mid \texttt{TErr}$$

The semantics `Typ.sem` of a type is obtained by evaluating recursively the semantics of its sub-expressions. If the result is a `TVal v`, we interpret it as a proposition type inhabited if and only if `v` is the Boolean `true`. Like introduced in section 6.5.2, we write $\texttt{v} \in \texttt{ty}$ if a value `v` inhabits a type `ty`.

For Frag IR, the implementation relation between the AS and the DTS is a partial function (as indicated by $\rightharpoonup$) on type symbols. It maps type symbols to their semantics:

$$\texttt{ty_impl_opt} : \texttt{Typ.Env.s} \to \texttt{Typ.sym_t} \rightharpoonup \texttt{Typ.sym_s}$$

This function depends on a global environment `te` : `Typ.Env.s` which contains:

- The interpretation of abstract predicates `prop` and `mem`, `te.int` : `pred_id` $\rightharpoonup$ (`list tsem` $\to$ `heaplet` $\to$ `Prop`). This function is chosen by the producer of the program, its domain must be exactly the set of predicate identifiers `pred_id` declared at the top-level. This interpretation function is used both for propositions and memory fragments. In the first case, a placeholder value is given for the `heaplet` argument, and should be ignored by the interpretation. The `tsem` arguments are the evaluations of the type expressions the predicate is applied to.
- An environment `te.strc` containing declarations of structures. In the AS, named structures can be declared at the top-level.

The implementation function `ty_impl_opt` replaces references to abstract predicates by their interpretation and inline the declarations of structures. As a result, type symbols in the DTS are functions `Typ.sym_s` which do not depend on any environment.

```
Int.t        := Int8 (i : I8.t) | Int16 (i : I16.t) | Int32 (i : I32.t) | Int64 (i : I64.t)
Float.t      := Float32 (f : F32.t) | Float64 (f : F64.t)
Scalar.t     := Vunit | Vbool (b : bool) | Vint (n : Int.t)
              | Vfloat (f : Float.t) | Vptr (p : Ptr.t)
Int.Size.t   := S8 | S16 | S32 | S64
Float.Size.t := S32 | S64
Scalar.Ty.t  := Unit | Bool | Int (sz : Int.Size.t) | Float (sz : Float.Size.t) | Ptr
```

Figure 8.3: Definition of scalar values and types in Rocq

8.1.1.1 Scalars

Programs perform computations on scalar values. In Rocq (figure 8.3), `Scalar.t` is the set of all scalar values. This set is partitioned into several scalar types, and `Scalar.Ty.t` is the set of those types. In most of this document, we use the textual names of the scalar types (e.g., `bool`) instead of their representation in the AST (e.g., `Bool`). Our scalar types are:

- The `unit` type, with a single inhabitant `Vunit`.
- The Booleans: `bool`.
- Bounded machine integers with various bit lengths: `i8`, `i16`, `i32` and `i64`. Contrary to higher-level languages like C, and like in the back-end of CompCert [69, §4.1], signedness indications are carried by the operators rather than by the types. In Rocq, the types of machine integers are `I8.t`, `I16.t`, `I32.t` and `I64.t`. The type `Int.t` is their disjoint union. We reuse the formalization of bounded integers of CompCert to define those types. We use the alias `Byte.t` for the type of bytes, that is, 8-bit machine integers `I8.t`.
- Single and double length floating-point numbers: `f32` and `f64`. In Rocq, they are represented using the types `F32.t` and `F64.t`. We reused the formalization of the IEEE-754 standard of the Flocq library [20], and additional definitions of CompCert.
- Pointers `ptr`, which are 64-bit machine integers. In Rocq, their set `Ptr.t` is an alias for the set `I64.t` of 64-bit integers.

We define a function `Scalar.ty_val : Scalar.Ty.t → Type`, which associates to each scalar type `t` a Rocq type representing the set of inhabitants of `t`. For instance `Scalar.ty_val Bool` is the type `bool` of Rocq. The set `Scalar.t` of all scalar values is isomorphic to the disjoint union of the `Scalar.ty_val` over all scalar types `t : Scalar.Ty.t`. In Rocq, this disjoint union is a sigma type `{t : Scalar.Ty.t & Scalar.ty_val t}`. Its inhabitants are pairs `(t, v)`[16], where `t : Scalar.Ty.t` is a scalar type and `v : Scalar.ty_val t` a scalar value of this type. The bijection with `Scalar.t` is accomplished by two functions `Scalar.to_t (t : Scalar.Ty.t) (v : Scalar.ty_val t) : Scalar.t` and `Scalar.of_t (v : Scalar.t) : {t : Scalar.Ty.t & Scalar.ty_val t}`. In some definitions, we use `Scalar.ty_val` and the dependent type system of Rocq to avoid some pattern matchings and ensure well-typedness by construction.

Each scalar type `t` has a byte-size, given by `sizeof t`. Scalar values with type `t` can be encoded as a list of bytes of length `sizeof t`. This encoding is described by the functions:

```
encode (t : Scalar.Ty.t) (v : Scalar.ty_val t) : Byte.t^(sizeof t)
decode (t : Scalar.Ty.t) (bs : Byte.t^(sizeof t)) : Scalar.ty_val t
```

For machine integers (including pointers) and floats, we reuse the encoding of the memory model of CompCert [68, §4.3]. Within a given scalar type `t`, `encode t` is injective (but not on `Scalar.t` as a whole, for instance an `i64` can have the same encoding as a `f64`), and `decode t` is its left inverse.

[16] In Rocq, those pairs are actually written `existT Scalar.Ty.t t v`.

However, the encoding is not surjective for `unit` and `bool` since those types have a byte-size of 1 but strictly less than 2^8 inhabitants.

`Scalar.Op.t` is our set of scalar operators. It contains partial and total operations. Each operator `o` : `Scalar.Op.t` is monomorphic, and has a signature consisting of a list of scalar types for its arguments and a scalar type for its result. For instance, the operator `not` has a single Boolean argument and returns a Boolean result. We write `not(bool) : bool`. In our textual syntax, we use the same name for several operators and use parameters within `{…}` to distinguish them. For instance, `add{i32}(i32, i32) : i32` and `add{f64}(f64, f64) : f64` are two distinct elements of `Scalar.Op.t`, corresponding to the addition on 32-bit integers and the addition on double-length floating-point numbers. The type `Scalar.Op.t` only contains meaningful operators. For instance, there is no operator corresponding to "`add{bool}`" and it is rejected when parsing programs. We also use parameters to specify the value of constants. For instance, `const{i64, 0}() : i64` returns the representation of 0 as a 64-bit integer. If an operation has a different semantics when interpreted as signed or unsigned, `Scalar.Op.t` contains two different operators. Textually, we prefix those variants with `u_` (unsigned) or `s_` (signed). For instance, there are two division operators `u_div{i64}(i64, i64) : i64` and `s_div{i64}(i64, i64) : i64`. Here are some examples of classes of operators:

- For any scalar type `t`, `eq{t}(t, t) : bool` tests the equality of its two arguments.
- Some operators return a constant value: `true() : bool` and `false() : bool` for the Booleans; `const{t, n}() : t` for an integer or float type `t` and value `n`; and `null() : ptr` for the null pointer (i.e. the representation `Ptr.null` of 0 as a machine integer).
- `cast{src, trg}(src) : trg` converts values between two scalar types `src` and `trg`. Allowed casts include conversion from any type to `bool`, signed and unsigned extensions of integers and conversion between integers and floats. There are also identity casts between `ptr` and `i64`.

Using our set of operations, we are able to encode the operations of CompCert C on Booleans, integers and floats. We provide a module `CopToScalar` which implements this encoding. This encoding is similar to the one performed by CompCert to generate C#minor programs [compcert.cfrontend.Cshmgen]. In general, the encoding of a CompCert C operation requires several operations on our scalars. In particular, we need to insert explicit casts to account for the type-directed conversions performed implicitly in CompCert C.

The scalar values are injected into the values of Frag IR by `Vsc` : `Scalar.t` → `val`. The scalar types are included in the types of Frag IR. Syntactically, a scalar type `t` is represented by the expression `Typ.Op (Tscalar t) []`. The semantics of the type symbol `Tscalar t` (i.e. the corresponding `Typ.sym_s`) does not expect any arguments and returns `TSet {Vsc s | s inhabits t}`. Scalar operators can appear in dependent types, for instance in `r = add{i64} n m`. Syntactically, we use expressions `Typ.Op (Tscalar_op o) args` where `o` is a scalar operation and `args` type expressions evaluating to scalar values (i.e. to `tsem` with shape `TVal (Vsc _)`). When it is applied to the correct number of scalar values, the semantics of `Tscalar_op o` is to return `TVal (Vsc r)` if `o` successfully evaluates to a result `r`. Evaluation errors (for instance a division by zero) are signaled by returning a `TErr`. For readability, we sometimes use infix notations in this presentation, like `n + m` instead of `add{i64} n m`.

8.1.1.2 Propositions

Propositions are dependent types reflecting relations between variables. A proposition is inhabited if and only if the property it represents is satisfied. If it is the case, its unique inhabitant is `Vsc Vunit`. That is, a proposition either evaluates to `TSet` ∅ or `TSet {Vsc Vunit}`. Syntactically, a proposition is the application `Typ.Op (Tprop pP) args` of a proposition symbol `pP` : `tprop_sym` to some arguments defined by type expressions. Those arguments may evaluate to types or values. The set `tprop_sym` of proposition symbols contains:

- Abstract predicates `prop{P}` parameterized by their identifier `P : pred_id`. The semantics of those type symbols is chosen by the producer of the program.
- Pre-defined propositions, for instance:
  - `eqp` denotes the equality between two arbitrary values. In this presentation, we write e_0 `=` e_1 instead of `eqp` e_0 e_1.
  - `not_error` is applied to a scalar operation, and is satisfied if the evaluation of this operation does not produce an error. For instance, if e_0 and e_1 are type expressions evaluating to 64-bit integers, then `not_error (u_div{i64}` e_0 e_1`)` is inhabited if and only if e_1 does not evaluate to zero. Formally, since scalar operators evaluate to `TErr` on failure, `not_error` is satisfied if and only if its argument is a `TVal`.

8.1.1.3 Memory fragments

Memory fragments are values which represent a subset of the memory. The function `footprint` maps memory fragments into heaplets describing their subset of the memory (data and permissions). It is defined by cases in figure 8.4. If `es` are expressions evaluating to values, the proposition `joinable es…` is satisfied if and only if they have disjoint footprints. Frag IR features two kinds of memory fragments:

- Unstructured fragments `Vmem h` are directly defined by a valid heaplet `h : heaplet`. They inhabit types whose head is `Tmem mP`. Similarly to proposition types, our set `tmem_sym` of memory predicate symbols includes abstract predicates (written `mem{P}`) and predefined predicates. For instance, `malloc e_p e_sz` is a type of unstructured fragments representing the result of a memory allocation. If the type expressions `e_p` and `e_sz` evaluate respectively to a pointer `p` and a 64-bit integer `sz`, then `malloc e_p e_sz` is inhabited by fragments `Vmem h` such that:
  - either `p` is null and `h` is the empty heaplet
  - or `p` is non-null, `valid_range p sz` and `h` contains exactly a deallocation permission for the range $[\texttt{p}; \texttt{p} + \texttt{sz})$ and values at the addresses in this range.
- Structured fragments are trees whose internal nodes are `Vstrc` and leaves `Vcell` or `Vmem` (for padding bytes). Those fragments inhabit types `Typ.Op (Tstrc s) [e_p]`, where `s : Structure.t` describes the memory layout of the structure and `e_p` should evaluate to its base address. The type `Structure.t` is defined inductively in figure 8.5.
  - An element `Id i` refers to a top-level declaration of a structure with identifier `i : Structure.id_t`. Using the global environment, we replace recursively those identifiers with their definition when interpreting the types.
  - An element `Cell ty` (written textually as `cell{ty}`) denotes a storage location for a scalar of type `ty`. If `e_p` evaluates to a non-null pointer `p` and `p + sizeof ty` does not wrap around, then the type `cell{ty} p` contains values `Vcell ty p bs`. The bytes `bs` are values for the addresses of the range $[\texttt{p}; \texttt{p} + \texttt{sizeof ty})$. The value stored by the cell is the decoding of the `bs`, as a scalar of type `ty`.
    Within `Vcell`, we store a byte encoding because the encoding of scalar values is not surjective. If we had instead only stored a scalar value and defined `footprint` using its encoding, then some memory fragments with the appropriate domain could not be interpreted as cells. On the contrary, our `Vcell` enables for instance reinterpreting any fragment containing only a byte at address `p` (even uninitialized) as a `cell{bool} p`.
    In types, the content of a cell `c` can be obtained as a scalar value using the expression `deref c`. We abbreviate this operation as `*c`.
  - A structure node `Mk sz` $[(\texttt{ofs}_i, \texttt{f}_i); \ldots]$ is defined by a size `sz` and a list of fields. Each field $(\texttt{ofs}_i, \texttt{f}_i)$ is a pair, composed of an offset and a structure layout. Those layouts are written `{sz :` $\texttt{ofs}_i$ $\texttt{f}_i$`;…}` in the textual representation of programs. The type `{sz :` $\texttt{ofs}_i$ $\texttt{f}_i$`;…} e_p` is inhabited by values `Vstrc` $[\texttt{v_pad}; \texttt{v}_i; \ldots]$ with $\texttt{v}_i \in \texttt{f}_i$ $(\texttt{e_p} + \texttt{ofs}_i)$ and $\texttt{v_pad} \in$ `mpadding{{sz :` $\texttt{ofs}_i$ $\texttt{f}_i$`;…}} e_p`. The type `mpadding{{sz :` $\texttt{ofs}_i$ $\texttt{f}_i$`;…}} e_p` is inhabited by unstructured memory fragments

```
footprint : val → heaplet
footprint (Vsc _) := emp
footprint (Vopt ⌊v⌋) := footprint v
footprint (Vopt ∅) := emp
footprint (Vstrc vs) := ⊎_{v ∈ vs} footprint v
footprint (Vcell _ p bs) := hrange p bs
footprint (Vmem h) := h
```

Figure 8.4: Definition of the `footprint` function

```
Structure.t ::= Id (i : Structure.id_t)                    structure declared at top-level
              | Cell (ty : Scalar.Ty.t)                    cell storing a scalar
              | Mk (sz : ℤ) (fs : list (ℤ × Structure.t))  structure node
```

Figure 8.5: AST of descriptions of structure layouts. We use relative numbers $\mathbb{Z}$ for sizes and offsets, but they are non-negative in practice.

TODO (rocq)

remplacer par `N.t`

containing padding bytes for the structure node taken as parameter. Formally, `Vmem h` ∈ `mpadding{{sz : ofs_i f_i;…}} e_p` if `e_p` evaluates to a non-null pointer `p`, `p + sz` does not wrap around and for any `v_i` ∈ `f_i (p + ofs_i)` there exist `sz` bytes `bs` such that `h` ⊎ (⊎$_i$ `footprint v_i`) = `hrange p bs`. As a consequence of the presence of this padding fragment, all fields must have disjoint footprints. Although layouts with overlapping fields (e.g. `{8 : 0 cell{i64}; 0 cell{i64}}`) are syntactically accepted, they yield types without inhabitants.
Within types, a structure node can be built from its subtrees (including the padding fragment) using the operator `mkstrc`.

8.1.1.4 Optional values

In our functional SSA representation, any variable in scope must contain a well-typed value. However, it is possible for a variable to be in scope but not used, depending on a dynamic condition. In some situations, it is useful to allow such a variable not to contain a value, that is to be uninitialized. For instance, in the C program of figure 8.6a, the variable `a` is only initialized (that is, its value defined) if `b0` is true, and used only if `b1` is true. This function has a well-defined semantics if `b1` implies `b0`. There are several possible approaches to formally define the semantics of programs where the initialization of variables is not guaranteed statically:

- Use partial maps to associate values to the variables, and raise a UB when accessing a variable which is not in the domain of the current map.
- Associate some particular value to uninitialized variables. An interesting choice is to use the poison value section 2.1.3 if it is present in the semantics. This is the approach used in CompCert for the representations after Clight [compcert.cfrontend.Clight:291].

In Frag IR, we use a total map from variables to values, and use option types to allow *some* variables to be uninitialized. Given any type `T`, the type `opt T` contains values `Vopt v`, where the content of `v : option val` has type `T` when it is present. A value `Vopt` ∅ represents an undefined value. For instance, we encode our example by giving type `opt i64` to `a` in figure 8.6b. This encoding and our operations on option types will be described in section 8.1.3.2.

In the type system, `opt` behave like the `option` type constructor of Rocq or OCaml. Values `Vopt` can be constructed within type expressions using `some e` and `none` (defined using `Tsome` and `Tnone` in the AST). We also provide propositions `is_some e`, which are satisfied if the argument `e` evaluates to a value with shape `Vopt ⌊_⌋` (i.e. a defined value). Types do not have any monotonicity property

TODO (rocq)

- renommer aux en foo

```
int foo(bool b0, bool b1) {
  int a;
  if (b0) {

    a = 1;

  }



  if (b1) {

    return a;

  } else {
    return 2;

  }
}
```

a: C

```
fn foo(b0 : bool, b1 : bool) : (_ : i64) {
  let a : opt i64 =
    if b0 _ then {
      let v : i64,      _ = const{1}();
      let a : opt i64, _ = Opt.some<i64>(v);
      return (a)
    } else {
      let a : opt i64, _ = Opt.none<i64>();
      return (a)
    };
  if b1 _ then {
    let ghost H : is_some a = assert<is_some a>();
    let v : i64, _ = Opt.pre_get<i64>(a, H);
    return (v)
  } else {
    let v : i64, _ = const{2}();
    return (v)
  }
}
```

b: Frag IR

Figure 8.6: A function where a variable a is only initialized in one branch of a conditional [compiler/test/src/example/opt.frag]

```
external_event : EventFam.t :=
  | Output (v : Scalar.t) : external_event unit
  | Input (ty : Scalar.Ty.t) : external_event (Scalar.ty_val ty)
FragIR.S := ExternalEvents.S external_event LTS.t
to_LTS [E A] (u : FragIR.S E A) : LTS.t (E + external_event) A := u
```

Figure 8.7: Domain of the semantics of Frag IR

with respect to optional values. That is, some types such as `v = none` are inhabited if a variable `v` is undefined (i.e. has value `Vopt ∅`) but not if it is defined. As a consequence, we do not have a reasoning principle to replace uninitialized value `Vopt ∅` by initialized ones. This is a drawback compared to other representations of uninitialized values. For instance, the poison value can always be replaced in CompCert, using `lessdef` simulations.

8.1.1.5 Type containing all values

In order to give a signature to the operator `Frag.joinable`, which can be applied to values of any types, we defined a type `any` containing all values. In the AST this type is represented as `Typ.Op Tany []`. We define the sub-typing relation (on the DTS) so that any variable can be given when an argument with type `any` is expected:

$$\frac{}{ty \subseteq ty} \qquad \frac{}{ty \subseteq \texttt{Typ.Op}\ (\texttt{ty_impl_opt}\ \texttt{Tany})\ []}$$

This sub-typing relation satisfies the property we required in section 6.5: it is a pre-order, compatible with variable renaming and consistent with the semantics `Typ.sem` of types.

8.1.2 Domain

TODO (rocq)

consistency

```
op_sym1 :=
  | Osc (op : Scalar.Op.t)          total or partial scalar operation
  | Osc_pre (op : Scalar.Op.t)      scalar operation with a precondition
  | Otrig (t : trig_sym)            external events
  | Oopt_some                       Opt.some
  | Oopt_none                       Opt.none
  | Oopt_get_prt                    Opt.get
  | Oopt_get_pre                    Opt.pre_get
  | Oread (ty : Scalar.Ty.t)        Frag.read{ty}
  | Owrite (ty : Scalar.Ty.t)       Frag.write{ty}
  | Omalloc                         Frag.malloc
  | Ofree                           Frag.free
  – Ghost operators
  | Olem (l : lem_use)              lemma{l}
  | Oassert                         assert
  | Ostrc_make (d : Structure.t)    Strc.make{d}
  | Ostrc_elim (d : Structure.t)    Strc.elim{d}
  | Ojoinable (a : ℕ)               Frag.joinable
Op.sym_t := {op_live : Liveness.t; op_sym : op_sym1}

Op.sym_s :=
  | Olem [a] (l : Lem.s a) (L : Lem.correct l) : Op.sym_s a
  | ...
```

Figure 8.8: AS of operations of Frag IR, and extract of the DTS

In order to define the semantics of Frag IR, we build a monadic domain `FragIR.S` supporting the effects of Frag IR and implementing the interface of `SMonad`. This domain is obtained in figure 8.7 by applying the transformer `ExternalEvents` (section 4.5.1) to the model `LTS.t` (chapter 5). We use a toy family of external events, `external_event`. It enables programs to:

- Emit a scalar value `v`, with `Output v`. The environment does not provide any data when answering those events, hence the answer type `unit`.
- Read a scalar value of a chosen type `ty` from the environment, with `Input ty`. The answer type `Scalar.ty_val ty` ensures that the environment replies with a well-typed value.

The domain `FragIR.S` supports the effects of divergence, non-determinism, undefined behaviors and internal and external events. That is, it provides all core operators of `SMonad` (`ret`, `bind`, `mrec`, ...) plus `any`, `ub` and an operator `external [E A] (e : external_event A) : FragIR.S E A`. It does not have any global state. In practice, the states of the underlying LTSs only store control points of programs (including a call stack) and values of local variables. We note `to_LTS` the identity conversion from `FragIR.S` to `LTS.t`. This conversion adds explicitly the external events in the signature of the semantics. We use it in section 10.4.2 to compare semantics defined in `FragIR.S` with semantics defined directly in `LTS.t`. Since the semantics $[\![.]\!]_{\text{prog}}$ of programs does not emit internal events, we simplify the signature of its unfolding in `LTS.t` by defining:

$$[\![\texttt{pr} : \texttt{Prog.s a}]\!]^{\text{Frag}}_{\text{prog}}\ (\texttt{vargmain} : \texttt{val}^{\texttt{a.arg}}) : \texttt{LTS.t external_event val}^{\texttt{a.res}} :=$$
$$\texttt{map_ev FragIR.get_ev}\ (\texttt{to_LTS}\ ([\![\texttt{pr}]\!]_{\text{prog}}\ \texttt{vargmain}))$$

where `FragIR.get_ev : EventFam.Fun.t (∅+external_event) external_event` is the expected bijective function on events.

8.1.3 Operators

Figure 8.8 reproduces the definition of the set of program operators. In Frag IR programs, some operations are considered as "ghost". That is, they are erased by our compilation pipeline, so they do not directly lead to the emission of instructions in the target. Their purpose is instead to propagate

information. We consider as "ghost" some operations which influence the semantics of programs at the level of Frag IR, as long as we are sure that they will be erased later in the compilation pipeline. We thus divert from the standard definition of ghost code. Even if we are not able to remove in Frag IR some of the operations we consider as ghost, we take into account their future removal:

- Their presence does not directly increase the cost of executing the target program.
- Their semantics is not necessarily effective. In particular, like explained in section 3.2.7, the semantics of `lemma` operations depend on the interpretation of the abstract predicates, which is not provided within the program.
- The arguments of those operations do not need to be computed in the target program. We thus also consider them as ghosts.

Some operators (e.g., `lemma` or `Frag.joinable`) are always ghost. The other operators are by default not ghost. Some occurrences may be considered as ghost if their results are only used for dependent types or for ghost operations. In this case, we annotate the operations with a `ghost` attribute, and we remove them during the erasure at the end of the compilation pipeline. This is thus only possible if we are able to remove the translation of the operator in Flat IR. For instance, scalar operations can be marked as `ghost` but `Frag.write` cannot (since its translation with `Flat.write` is not semantically removable according to definition 7.3).

In the AS, the set of operators is defined using an algebraic datatype `op_sym1`. The type `Op.sym_t` of operator symbols adds a semantically irrelevant liveness annotation `op_live`. This annotation is set by the liveness analysis (section 7.2.2) and by the output of some transformations (e.g. section 9.2). By default `op_live` is `Live` and left implicit in the textual syntax. An annotation `Ghost` (`ghost` in program text) signals ghost operations whose operator is not always ghost. Those operations are removed by the erasure at the end of the compilation pipeline. An annotation `Dead` (`dead` in program text) is used transitorily in the DCE. Those operations are removed immediately by the erasure pass which follows the liveness analysis.

In the DTS, operator symbols `Op.sym_s` are indexed by their arity (number of type and value arguments and of results). We also enforce some well-formedness invariants using dependent types. In particular, the DTS only contains correct lemmas, thanks to an argument `Lem.correct` in the constructor `Olem`. In this document, we use the textual representation of the operators, commented on the right of figure 8.8. The following subsections describe the different kinds of operators.

We define an implementation relation between the AS and DTS of operator symbols. It has signature:

$$\texttt{Op.sym_impl}\ (\texttt{globs} : \texttt{Globs.s})\ [\texttt{a}]\ (\texttt{t_op} : \texttt{Op.sym_t})\ (\texttt{s_op} : \texttt{Op.sym_s}\ \texttt{a}) : \texttt{Prop}$$

It relates corresponding constructors of the DTS and AS. Ill-formed AS operators (e.g., incorrect lemmas) do not implement any DTS operators. The implementation relation depends on the global declarations `globs` : `Globs.s` present in the program. Those declarations define an environment `Typ.Env.s` used in the implementation function of the type symbols. It additionally contains top-level declarations of lemmas.

As explained in section 6.5, we define the signature of the operators in the DTS, as a function `Op.sig` [a] (o : `Op.sym_s` a) : `Op.Sig.s` a. We also define the semantics of DTS operators:

$$[\![\texttt{o} : \texttt{Op.sym_s}\ \texttt{a}]\!]_{\text{op}}\ (\texttt{vT} : (\texttt{Typ.s val})^{\texttt{a.argT}})\ (\texttt{vV} : \texttt{val}^{\texttt{a.argV}}) : \texttt{FragIR.S}\ \varnothing\ \texttt{val}^{\texttt{a.res}}$$

The semantics of an operator `o` is thus an element of the domain `FragIR.S` which depends on the types (`vT`) and value (`vV`) arguments the operator is applied to. We build those semantics using the core operators of `SMonad` and those specific to the effects of `FragIR.S`. Note that the Rocq type of the semantics of operators does not take into account the signature given by `Op.sig`, and instead use arbitrary `val`s as arguments and results. We use a pattern matching to extract from those arguments

the data expected by the operators. We use a placeholder semantics (a UB) for the cases incompatible with the types (in Frag IR, not Rocq) of the arguments. In the following, we only describe the semantics in the well-typed cases. We prove a posteriori that our semantics preserves the types of the values. That is, when applied to well-typed arguments, operators return well-typed results (or never return).

8.1.3.1 Scalar operators

Scalar operators are included in Frag IR using two constructors: `Osc` and `Osc_pre`. The second variant is reserved for partial scalar operations, and takes an additional pre-condition `not_error`. Thanks to this pre-condition, the dependent type system of Frag IR enables considering `Osc_pre` operations as total. On the contrary, partial operations used with `Osc` have UB for some values of their arguments. Textually, we add a prefix `pre_` to the operator name when using the variant with a pre-condition. Both variants return two results: the scalar `r` computed by the scalar operation, and a post-condition `r = …` which reflects in the type system the value of `r` as a function of the arguments.

Example 8.1 The partial scalar operator `u_div{i64}(i64, i64) : i64` yields two operators in Frag IR.

- A partial operator:

```
u_div{i64}(n : i64, m : i64) : (r : i64, ghost POST : r = u_div{i64} n m)
⟦u_div{i64}⟧op [] [Vsc (Int (Int64 n)); Vsc (Int (Int64 m))] :=
  { ret [Vsc (Int (Int64 (I64.divu n m))); Vsc Vunit]   if m ≠ 0
  { ub                                                  if m = 0
```

- A total operator with a pre-condition:

```
pre_u_div{i64}(n : i64, m : i64, ghost PRE : not_error (u_div{i64} n m))
  : (r : i64, ghost POST : r = u_div{i64} n m)
⟦u_div{i64}⟧op [] [Vsc (Int (Int64 n)); Vsc (Int (Int64 m)); Vsc Vunit] :=
  ret [Vsc (Int (Int64 (I64.divu n m))); Vsc Vunit]
```

In both cases, `I64.divu` is the unsigned division on 64-bits integers. Note that `{i64}` is a parameter of the operators, not a type argument.

External events are emitted by `output{t}(o : t) : ()` and `input{t}() : (i : t)`, where `t` is a scalar type. In the AST, those operators are both represented by `Otrig`. The argument `trig_sym` specifies the kind of event emitted and the scalar type `t` considered. The semantics of this operation is defined using the semantic operator `external`:

```
⟦output{t}⟧op [] [Vsc v] := external (Output v);; ret []

⟦input{t}⟧op [] [] := v ← external (Input t); ret [Scalar.to_t t v]
```

8.1.3.2 Operations on optional values

Optional values are constructed using the operators `Opt.some` and `Opt.none`:

```
Opt.some<T>(v : T) : (r : opt T, ghost POST : r = some v)
⟦Opt.some⟧op [T] [v] := ret [Vopt ⌊v⌋; Vsc Vunit]

Opt.none<T>() : (r : opt T, ghost POST : r = none)
⟦Opt.none⟧op [T] [] := ret [Vopt ∅; Vsc Vunit]
```

Those operators have pure semantics. They take a type argument `T` specifying the type of the content of the option. Contrary to scalar operators, which may take a parameter restricted to some scalar types, options are defined for arbitrary types. However, in practice we currently only use them on scalar types. Like the scalar operators, operators on options return a post-condition `POST`.

The only way of using an optional value in a non-ghost context is by unboxing the content `v` of a defined value `Vopt ⌊v⌋`. Frag IR provides an operator for this purpose. Like partial scalar operations, it has two variants:

```
Opt.get<T>(o : opt T) : (r : T, ghost POST : o = some r)
```

$$[\![\texttt{Opt.get}]\!]_{\text{op}}\ [\texttt{T}]\ [\texttt{Vopt o}] := \begin{cases} \texttt{ret}\ [\texttt{v}; \texttt{Vsc Vunit}] & \text{if } \texttt{o} = \lfloor \texttt{v} \rfloor \\ \texttt{ub} & \text{if } \texttt{o} = \varnothing \end{cases}$$

```
Opt.pre_get<T>(o : opt T, PRE : is_some o) : (r : T, ghost POST : o = some r)
```

$$[\![\texttt{Opt.pre_get}]\!]_{\text{op}}\ [\texttt{T}]\ [\texttt{Vopt } \lfloor \texttt{v} \rfloor; \texttt{Vsc Vunit}] := \texttt{ret}\ [\texttt{v}; \texttt{Vsc Vunit}],$$

The first variant has UB if its argument is `Vopt` $\varnothing$, while the second is a pure operation, thanks to the pre-condition which ensures that the argument is defined. In figure 8.6b, we used a combination of `assert` and `Opt.pre_get` to unbox the value of `a` in the `then` branch of the conditional on `b1`. Semantically, this has the same effect as using `Opt.get`, but it syntactically separates obtaining information from the computations. As future work, we could automatically replace all `Opt.get` by this encoding.

Our set of operators is thus restricted so that, in non-ghost contexts, optional values behave like potentially undefined values. That is, it is not possible to observe whether an optional value is defined, but only to unbox a value which is already known to be defined. This contrasts with the unrestricted access to optional values in the types, where they behave like standard option types (section 8.1.1.4). As a consequence of the restriction of the operators, it is possible to replace any non-ghost operation on an optional value `o` by an operation on a more defined value $\texttt{o}'$ (i.e., if `o` is undefined or $\texttt{o}' = \texttt{o}$). We leverage this property in section 9.2 to remove the optional values. This principle is similar to reasoning with `lessdef` relations on poison values (section 2.1.3). However, our optional values are more restrictive than semantics of uninitialized variables based on poison values. Indeed, our set of operators does not provide a way of performing operations on values that may be undefined. This is possible with poison values, by having operations return the poison value if one of their arguments is poisoned. For instance, if we interpret uninitialized variables as holding a poison value, we could insert a pure operation such as `a++` between the two conditionals of figure 8.6a. The function would then have a well-defined semantics under the same condition $\texttt{b1} \Rightarrow \texttt{b0}$. However, we would no longer be able to encode it in Frag IR using optional values. Indeed, contrary to poison values, our `opt` types guarantee that operations are always performed on defined values.

FEEDBACK DM260916

Nous avons plusieurs notions reliées: fonctions partielles, fonctions totales avec argument supplémentaire, valeurs poisons, valeurs optionnelles. Ça manque un peu de synthèse pour justifier l'usage de plusieurs cas distincts: quand utilises-tu telle notion plutôt qu'une autre?

8.1.3.3 Memory operations

Frag IR provides operators on memory fragments with the same purpose as standard operators on a global memory: read, write and dynamic allocation (figure 8.9). We informally presented those operators in section 3.2.4. In our compilation pipeline, they lead to the emission of corresponding operators on the global memory during the translation to Flat IR. As a consequence, those operators on memory fragments are not considered as ghosts. In particular, their pointer arguments must be computed effectively by the program. On the contrary, the size argument `sz` of `Frag.free` is only used for the dependent type of the fragment argument `m`. It is thus considered as ghost and not necessarily computed in the target program.

```
Frag.read{t}(p: ptr, ghost c: cell{t} p) : (r: t, ghost POST: *c = r)
⟦Frag.read{t₀}⟧_op [] [Vsc (Vptr p₀); Vcell t₁ p₁ bs] :=
  ret [Vsc (Scalar.to_t t₁ (decode bs)); Vsc Vunit]

Frag.write{t}(p: ptr, ghost c: cell{t} p, v: t)
  : (ghost c': cell{t} p, ghost POST: *c' = v)
⟦Frag.write{t₀}⟧_op [] [Vsc (Vptr p₀); Vcell t₁ p₁ _; Vsc v] :=
  ret [Vcell t₂ p₁ (encode v'); Vsc Vunit]
  where Scalar.of_t v = (t₂, v')

Frag.malloc(sz: i64) : (p: ptr, ghost m: malloc p sz)
⟦Frag.malloc⟧_op [] [Vsc (Vint (Int64 sz))] :=
  (p, h) ← any {(p, h) | op_malloc_choice (I64.to_nat sz) p h};
  ret [Vsc (Vptr p); Vmem h]
```

where `op_malloc_choice (sz : ℕ) (p : Ptr.t) (h : heaplet) : Prop` is generated by:

$$\frac{}{\texttt{op_malloc_choice sz Ptr.null emp}}$$

$$\frac{\texttt{valid_range p sz}}{\texttt{op_malloc_choice sz p (hrange p bs} \uplus (\varnothing, [\texttt{p} \mapsto \texttt{sz}]))}$$

```
Frag.free(p: ptr, ghost sz: i64, ghost m: malloc p sz) : ()
⟦Frag.free⟧_op [] [Vsc (Vptr p); Vsc (Vint (Int64 sz)); Vmem h] := ret []
```

Figure 8.9: Signature and semantics of operators on memory fragments

The semantics of the operators is reproduced in figure 8.9. `Frag.read` and `Frag.write` have pure semantics, which encode or decode the bytes stored in `Vcell`. We do not have to check whether pointers are non-null or if the accessed parts of the memory are allocated. Indeed, a `cell` always represents a valid memory location. More anecdotally, we also simplify the semantics by not checking whether the various scalar types (t_0, t_1, t_2) and pointers (p_0, p_1) are equal, since this is already ensured by the types of the arguments.

The semantics of `Frag.malloc` is a non-deterministic choice of a pointer and heaplet satisfying the predicate `op_malloc_choice`. There are several sources of indeterminacy:

1. The allocation can always fail by returning a null-pointer and an empty memory fragment. Note that this failure is not a UB.
2. When the allocation succeeds, the result `p` is an arbitrary pointer compatible with the requested size. This non-determinism is linked to our use of concrete pointers. It abstracts both the implementation of `malloc` and transformations performed by the compiler. In memory models based on logical pointers, one can instead choose deterministically the next block available.
3. The initial values `bs` contained in the allocated memory fragment are chosen non-deterministically. If we had a notion of poison value, we could have instead filled the heaplet with it to prevent programs from depending on uninitialized values.

As explained in section 3.2.6, in case of success the choice of the returned pointer is not influenced by other existing allocations. In Frag IR, a deallocation `Frag.free` is semantically a no-op. It ignores the values of its arguments and does not have any side effect. However, it influences the linearity criterion used during the translation to Flat IR (section 3.2.5), where it generates a standard memory deallocation. More precisely, it must be applied to an active memory fragment, and consumes it.

8.1.3.4 Ghost operators

Ghost operators do not directly lead to the emission of instructions in the target program. However, they may influence the semantics at the level of Frag IR.

Lemma The operator `lemma` is used to derive propositions (section 3.1.3) or to merge and split memory fragments (section 3.2.7). It is parameterized by its signature, written:

```
<xsT>(xsV : targ) : (xsR : tres) consume (csm)
```

where `xsT`, `xsV` and `xsR` are lists of formal names for the type arguments, the value arguments and results; `targ` and `tres` are lists of types for the value arguments (depending on `xsT` and `xsV`) and for the results (depending on `xsT`, `xsV` and `xsR`); and `csm` is a list of consumed arguments. We omit the `consume` clause if `csm` is empty. In the textual representation of programs, the signatures of lemmas are either defined at the point where they are used or at the top-level. The second option enables giving a name to lemmas and reusing the same lemma at multiple program points. In the AS, this is reflected by the parameter `lem_use` of `Olem`, which is either a name or an inline signature. The implementation relation with the DTS replaces names by their definition. The type `Lem.s a` represents the signature of lemmas in the DTS. It is indexed by its arity `a : Op.Arity.s`. In the following, we consider a lemma signature `l : Lem.s a` representing `<xsT>(xsV : targ) : (xsR : tres) consume (csm)`. We describe the results of lemmas with a predicate:

Definition 8.1 Given values `vT` and `vV` for its arguments, the possible results `vR` of a lemma with signature `l` are those satisfying the post-condition:

$$\begin{array}{l}\texttt{Lem.post l } (\texttt{vT} : (\texttt{Typ.s val})^{\texttt{a.argT}})\ (\texttt{vV} : \texttt{val}^{\texttt{a.argV}})\ (\texttt{vR} : \texttt{val}^{\texttt{a.res}}) : \texttt{Prop} := \\ \quad \texttt{vR} \in \texttt{tres}[\texttt{vT/xsT}, \texttt{vV/xsV}, \texttt{vR/xsR}] \wedge \qquad (\texttt{TyRes}) \\ \quad \uplus_{\texttt{x} \in \texttt{csm}}\ \texttt{footprint}\ (\texttt{vV x}) \neq \bot \wedge \qquad (\texttt{ValidFootprint}) \\ \quad \uplus_{\texttt{x} \in \texttt{csm}}\ \texttt{footprint}\ (\texttt{vV x}) = \uplus_{\texttt{v} \in \texttt{vR}}\ \texttt{footprint v} \qquad (\texttt{EqFootprint})\end{array}$$

The semantics of `lemma` is thus a non-deterministic choice. Formally:

$$[\![\texttt{lemma\{l\}}]\!]_{\text{op}}\ \texttt{vT vV} := \texttt{any}\ \{\texttt{vR} \mid \texttt{Lem.post l vT vV vR}\}$$

The post-condition `Lem.post` of a lemma is defined by the conjunction of three clauses. The clause `TyRes` ensures that the results are well-typed. In particular, it implies that proposition types in `tres` are inhabited, and thus satisfied. The two other clauses describe a correspondence between the footprints of the consumed arguments and the footprints of the results. The clause `EqFootprint` ensures an equality between their disjoint unions. However, this equality alone would be satisfied if both sides were the invalid heaplet $\bot$. In particular, this would happen as soon as two consumed arguments and two results were not disjoint, regardless of the footprint of the other arguments. The purpose of `ValidFootprint` is to rule out those cases. Thanks to this additional condition, we can deduce from `EqFootprint` equalities between data stored in memory fragment:

Example 8.2 Assume that `l` contains some `cell{t}` with the same address within its consumed arguments and results. That is, that there is a formal consumed argument `xA` $\in$ `csm` and a formal result `xR` $\in$ `xsR` which both have type `cell{t} p`. Then, *regardless* of the other arguments and results, `Lem.post l vT vV vR` implies that `vV xA` $=$ `vR xR`.

Such properties could be used to propagate values stored in memory fragments through lemmas during transformations. However, there is currently no pass in our pipeline which uses `ValidFootprint`.
Lemmas are subject to a soundness condition. Propositions returned by lemmas should be derivable from the propositions taken as arguments. Moreover, when they are applied to memory fragments, the transformations described by lemmas should be feasible. In both cases, those conditions amount

to requiring non-deterministic choice associated to lemmas to be performed within non-empty sets. Formally, a lemma is thus sound if for any well-typed arguments, there exist results satisfying its post-condition:

Definition 8.2 The lemma `l` is *sound* if:

```
Lem.correct l :=
  ∀(vT : (Typ.s val)^a.argT) (vV : val^a.argV). vV ∈ targ[vT/xsT, vV/xsV] ⇒
  ∃vR : val^a.res. Lem.post l vT vV vR
```

Programs should only contain sound lemmas. We formally propagate this invariant thanks to the `Lem.correct` argument in the DTS of `Olem`.

Operations on structure nodes The operators `Strc.make{d}` and `Strc.elim{d}` were informally described in section 3.2.7. They reorganize structured memory fragments by creating or destructing structure nodes. They are parameterized by the layout `d` of the structured fragment they operate on. The root of this layout must be an internal node `Mk`, not a `Cell`. In the AS, this description is either given inline within the operator[17] or by the name of a structure declared at the top-level. Those operators have pure semantics:

```
Strc.make{{sz : ofs_i field_i;…}}
    (ghost p : ptr, ghost pad : mpadding{{sz : ofs_i field_i;…}} p,
     ghost f_i : field_i (p + ofs_i),… )
  : (ghost s : {sz : ofs_i field_i;…} p,
     ghost POST : s = (mkstrc pad f_i…))
⟦Strc.make{{sz : ofs_i field_i;…}}⟧_op [] [Vsc (Vptr p); Vmem pad; fs…] :=
  ret [Vstrc [Vmem pad; fs…]; Vsc Vunit]

Strc.elim{{sz : ofs_i field_i;…}}
    (ghost p : ptr, ghost s : {sz : ofs_i field_i;…} p)
  : (ghost pad : mpadding{{sz : ofs_i field_i;…}} p,
     ghost f_i : field_i (p + ofs_i),…
     ghost POST : s = (mkstrc pad f_i…))
⟦Strc.elim{{sz : ofs_i field_i;…}}⟧_op [] [Vsc (Vptr p); Vstrc [Vmem pad; fs…]] :=
  ret [Vmem pad; fs…; Vsc Vunit]
```

One could use `lemma` instead of those specialized operators. Their interest is that they have simpler semantics. In particular, contrary to general transformations with lemmas, `Strc.make` and `Strc.elim` have deterministic semantics.

Assertions Lemmas only derive propositions implied by information that was already present in the type system. On the contrary, the operator `assert` adds new information with a semantic justification encoded with UB. This operator takes a proposition[18] `P` as type argument, which is the return type. This proposition may depend on the variables in the contexts where `assert` is used. The semantics of `assert` (the program operator) uses the `assert` operation of the domain `FragIR.S` (section 4.2.4.2) to check that `P` is satisfied, that is, inhabited by `Vsc Vunit`:

```
assert<P>() : (POST : P)
⟦assert⟧_op [P] [] := assert (Vsc Vunit ∈ P);; ret [Vsc Vunit]
```

[17] this leads to double brackets in the textual representation, to delimit the operator arguments and the description of the node, e.g. `Strc.elim{{8 : 0 cell{i64}}}`

[18] Syntactically, any type can be given as argument to the program operator `assert`, but it is not useful to give types that are not propositions.

If P is satisfied, assert<P>() is semantically a no-op. Otherwise, it is a UB.

Disjointness assumptions The operator Frag.joinable requires the compiler to guarantee that its arguments have disjoint footprints. It takes any number of arguments, with arbitrary types (but is used on memory fragments in practice). Because the interface of the parameterized IR requires operators to have a single signature, in the AST Ojoinable is annotated with the number a of arguments. This number is automatically added when parsing the textual representation of programs. Moreover, we use the sub-typing mechanism and the type any to allow arbitrary arguments.
For the producer of the program, Frag.joinable provides a semantics assumption. Formally, this semantics is a non-deterministic choice within a Rocq proposition (i.e., in a Prop, section 4.2.4.1):

```
Frag.joinable(ghost args : any…) : (ghost POST: joinable args…)
```
$[\![\texttt{Frag.joinable}]\!]_{\text{op}}\ []\ \texttt{args} := \texttt{any}\ \left(\biguplus_{\texttt{a} \in \texttt{args}} \texttt{footprint a} \neq \bot\right);; \texttt{ret}\ [\texttt{Vsc Vunit}]$

If the arguments are disjoint, Frag.joinable is a no-op, but returns a proposition joinable which reflects this disjointness in the type system. Otherwise, the execution is halted without emitting further behaviors.

8.2 Side effects of the operators

The semantics of operators have access to all effects supported by FragIR.S (except calls to continuations and internal functions). However, most operators have pure semantics. In particular, memory operations do not depend on any global state. In section 3.2.8, we already mentioned some transformations enabled by this purity. However, some operators still have side effects: emission of external events, UB and non-determinism (including stuck choices within empty sets). This section recalls the side effects performed by the operators, and how they affect semantic refinement within our framework SMonad. We do not discuss here the other invariants that passes must preserve: the typing and the linear usage of memory fragments. As illustrated in section 7.2.1, some semantically correct transformations are invalid because they break those invariants.

External events The operators input and output emit external events. Compilation must preserve those events. That is, target executions must emit the same events as corresponding source executions. In our domain FragIR.S, we use the transformer ExternalEvents.S to hide the external events from the signature of semantics. When reasoning on FragIR.S (that is, without unfolding its definition to LTS.t), external events cannot be interpreted nor changed by non-identity relations on events. In fact, there is no non-trivial property on the operator external. When it appears in a refinement, the reflexivity rule sem_ref_refl must be used to relate it to a corresponding external event on the other side.

Undefined Behaviors On well-typed arguments, only assert and the variants of partial operations without pre-condition (Osc and Oopt_get_prt) may have UB. As illustrated in section 2.1.2, operators with UB cannot in general be inserted, or moved up in the control flow. In SMonad, this property is reflected by the fact that ub cannot be inserted in the target semantics (not_ub_insert in figure 4.12). That is, during a refinement, the target semantics cannot execute ub unless the source also does. This is true in any consistent model of SMonad and in particular in FragIR.S. It is expected since UBs represent errors or semantically justified information. Freely inserting them in the target would be inherently unsound.
Our simulation-based notion of refinement on LTS.t does not allow time-travelling UBs, that is, to exploit UBs that occur after events that have not yet been emitted (section 4.6.6). As a consequence, operators with UB cannot be commuted above external events. They also cannot be commuted above trigger. That is, above function calls when reasoning on the semantics of functions without interpreting recursive calls.

TODO (rocq)

Je crois que `trigger; ub >/= ub` est vrai pour tout modèle consistent de `SMonad`.

Assumptions Semantically, the operator `Frag.joinable` is the dual of UB (section 4.2.4.1). After its execution, programs may assume that a disjointness property holds. In `SMonad`, this semantics is encoded by a non-deterministic choice `any` in a `Prop`. We are not interested in the element chosen (which is a proof of the disjointness), but only in the fact that such an element exists (which is not the case if the arguments of `Frag.joinable` are not disjoint). Dually to UBs, `any` cannot be removed when the choice is performed within a type that may be empty (`not_any_emp_remove` in figure 4.12). It also cannot be commuted below `trigger` (in particular external events). This is expected and desirable for `any` $\varnothing$: otherwise the target programs could emit some events while the source program is stuck (without generating further behaviors).

(Non-empty) non-deterministic choices Frag IR features two non-deterministic operators: `Frag.malloc` and `lemma`. Their semantics are defined by an `any` on a non-empty type (thanks to the failure case for `Frag.malloc` and the soundness condition for `lemma`). Semantically, those operators can thus be inserted or removed. They also commute with all other operators, except external events. Indeed, `any` cannot be commuted below `trigger` in `SMonad` (`not_any_trigger_comm` in figure 4.12). As a consequence, the non-deterministic operators of Frag IR cannot be commuted below arbitrary code.

This prevents using `sem_ref` to prove some optimizations, such as the reuse of a memory allocation depicted in figure 8.10. In this example, the source program figure 8.10a contains two allocations with syntactically intersecting live ranges (that is, `m0` is freed after `m1` is allocated). However, the two allocated fragments are actually used in disjoint intervals, modeled by the calls to `f0` and to `f1`. It is thus desirable to optimize this program as in figure 8.10b, by reusing the same allocation. This saves some memory space and avoids a call to `malloc`. Note that this transformation would be invalid if we use a standard stateful `malloc`. Indeed, in the source `p0` and `p1` are always distinct or null (since they have intersecting allocation ranges), whereas in the target `f1` is given twice the same pointer as arguments. However, in Frag IR, the semantics of `Frag.malloc` does not depend on other existing allocations. The two `Frag.malloc` of figure 8.10a are thus allowed to return the same non-null pointer. Unfortunately, we faced the following difficulty when trying to prove this transformation with the relational approach of `SMonad`. During the refinement, we need to choose in the source the data of `m10` so that it matches the data seen by `f1` in the target, that is, what is left in `m1`. However, to obtain those data, we need to execute `f0` (simultaneously in the source and target), and this can only be done after the non-deterministic choice of `m10`. The transformation is provable with `SMonad` if we consider a source program where `m10` is allocated after the call to `f0`. Being able to commute `any` with `trigger` (hence internal function calls) could thus enable proving the transformation of figure 8.10 with `SMonad`. On this example, another way of enabling the proof of the transformation could be to introduce a poison value and to use it to initialize deterministically the content of allocated fragments.

8.3 Instantiation of the generic passes

Frag IR is defined as an instance of our generic parameterized SSA IR (chapter 6). This enables using the generic passes described in chapter 7 on Frag IR programs. The simplification pass (section 7.1) and block sinking (section 7.3) are available for free, without implementing anything specific to Frag IR. Instantiating the type-checker (section 7.4) and the DCE (section 7.2) requires implementing some parameters specific to Frag IR and proving their correctness. This section describes those parameters.

```
let p0 : ptr, m00 : malloc p0 sz =
  Frag.malloc(sz);
let p1 : ptr, m10 : malloc p1 sz =
  Frag.malloc(sz);
let m01 : malloc p0 sz =
  fcall f0(sz, p0, p1, m00);
let m11 : malloc p1 sz =
  fcall f1(sz, p0, p1, m10);
Frag.free(p0, sz, m01);
Frag.free(p1, sz, m11)
```

a: Source

≱

```
let p : ptr, m0 : malloc p sz =
  Frag.malloc(sz);

let m1 : malloc p sz =
  fcall f0(sz, p, p, m0);
let m2 : malloc p sz =
  fcall f1(sz, p, p, m1);

Frag.free(p, sz, m2);
```

b: Target

Figure 8.10: A transformation which cannot be proved using `SMonad` because of the absence of commutation of non-deterministic choices below events [compiler/test/src/transform/malloc.frag:107]

TODO (rocq)

preuve directe sur les comportements

8.3.1 Type-checker

We instantiate the parameters of the type-checker [Compiler.Impl.Inst.TypCheck:651] so that it checks:

- The well-formedness of programs, except the soundness of its lemmas. That is, if an AS program `t_pr` is accepted by the type-checker, and if all its lemmas are correct, then there exists a DTS program `s_pr` implemented by `t_pr`. As explained in section 7.4, we prove formally this property [Compiler.Impl.Inst.TypCheck:653], but we do not use it in our compilation pipeline.
- That programs are well-typed. That is, we instantiate lemma 7.5 for Frag IR: if an AS program `t_pr` implements a DTS program `s_pr` and if `t_pr` is accepted by the type-checker, then `s_pr` is well-typed.

In Frag IR the implementation relation for the type symbols and operators depends on global declarations (abstract predicates, structures and lemmas). Our instances of the parameters of the type-checker thus access the global declarations. We implement the following parameters:

`check_typ_sym` checks that a given type symbol (`Typ.sym_t`) is well-formed. We check that:

- abstract predicates `prop<P>` and `mem<P>` refer to a declared predicate identifier `P`
- structures `s : Structure.t` do not contain unbound references. That is, that any `Id i` refers to a structure `i` declared at the top level. For `mpadding{s}`, we additionally check that `s` unfolds to a structure node, not a `cell`.

Note that we only check type symbols, our type-checker does not reject meaningless types such as `add{i64} bool i32` (our semantics of types contains placeholders for such cases).

`check_typ_sub` checks that the sub-typing relation holds between two types. Except when the right-hand side is the type `any`, this amounts to checking that the DTS of the types are equal. We thus check that the AS of the types have identical tree structures and compare the type symbols. In most cases, we test the equality of the AS type symbols. However, when structures are involved, we test their equality modulo the unfolding of the top-level declarations. For instance, if `s` is declared by `struct s {8: 0 cell{i64}}` and `p` is a pointer, then the types `s p` and `{8: 0 cell{i64}} p` have distinct representations in the AS but implement the same DTS type. By unfolding the definition of `s`, our type-checker enables using the two types interchangeably. This unfolding is done lazily: we do not unfold `s` when checking an equality $s = s$.

In the DTS of Frag IR, type symbols `Typ.sym_s` are defined semantically, rather than by a syntactic enumeration like in the AS (`Typ.sym_t`). As a consequence, the notion of equality used in the definition of the typing judgment has a semantic aspect: type symbols with equivalent semantics are considered equal. In particular, two abstract predicates with the same interpretation are considered equivalent in the typing judgment. On the contrary, `check_typ_sub` compares the

syntactic identifiers of the predicates. It is thus not complete. This is not an issue in practice because we expect programs to be well-typed with respect to the syntactical comparison implemented by the type-checker.

`get_op_ty` returns the signature of operators. We access the global declarations to obtain the definitions of structures and lemmas. This function also checks that the operator symbol is well-formed, that is:

- In `Osc_pre o`, the scalar operator `o : Scalar.Op.t` must be partial. For instance, we reject `pre_add{i64}`. Indeed, its pre-condition is always satisfied and `add{i64}` should be used instead.
- The parameter `l : lem_use` of a `lemma{l}` must be either:
  - the identifier of a lemma declared at the top level
  - a closed signature with well-formed type symbols
- The parameter `d : Structure.t` of `Strc.make{d}` and `Strc.elim{d}` must be well-formed and unfold to a structure node.

8.3.2 DCE

The DCE is a sequence of a liveness analysis [Compiler.Impl.Inst.Live:95] followed by an erasure pass [Compiler.Impl.Inst.Erasure:94]. The former annotates its results on the variables and operators with a qualifier "`dead`". In the AS of Frag IR, those qualifiers are part of the annotations `An.var` on variables and of the operator symbols (field `op_live` of `Op.sym_t`). We implement the parameters of the liveness analysis and of the erasure (figure 7.5) to:

- write the result of the liveness analysis in the AS of the program and read them during the erasure
- restrict the removal of variables and operators to preserve side-effectful operations and the linear usage of fragments

In particular, during the DCE:

- The semantics of external events and of `Frag.joinable` are not removable (with respect to definition 7.3). We force the DCE to consider those operations as live by setting the field `min_live` to `Live` in the result of `get_op_properties` for those operators.
- We prevent the variables containing memory fragments from being considered as `dead` by having `is_typ_linear` return `true` for types of memory fragments.
- Although they are semantically removable, we force the liveness analysis to consider the operations `Frag.write`, `Frag.malloc` and `Frag.free` as `Live`. Indeed, they should not be removed in Frag IR since it could break the linearity condition (section 7.2.1). Moreover, they cannot be considered as ghost operations since their translation to Flat IR involves semantics with non-removable side effects. We thus set the field `min_live` to `Live` for those operators. On the contrary, `Frag.read` can be removed in Frag IR without breaking the linearity condition and its translation to Flat IR can be removed by the ghost code erasure. It can thus be marked as `dead` or `ghost`, and has a trivial lower bound `min_live` = `Dead`.
- Similarly, removing `lemma`s consuming memory fragments, `Strc.make` and `Strc.elim` can break the linearity condition. Since those operators can be ghost (and even always are), we only give them a lower bound `min_live` = `Ghost`.

During the erasure, `is_var_erased` and `is_op_erased` read the liveness annotations in `An.var` and `Op.sym_t`. We need to guarantee formally that we only erase operations with removable side effects. Our implementation of `is_op_erased` thus defensively checks if the operator is semantically removable, and raises an error if an operator which cannot be removed is marked as `dead`.

8.4 Examples

We wrote several toy programs directly in Frag IR. Those small examples have several purposes:

- They demonstrate how programs and data-structures are encoded in Frag IR.

- We test on them the automatic passes we implemented (described in section 8.3 and in chapter 9).
- We manually transform them and prove refinements, to support our claims about the transformations enabled by Frag IR (in particular those of section 3.2.8).

In total, we wrote 79 programs directly in Frag IR, for a total of nearly 1.8k lines. All Frag IR programs presented in this document are adapted from those examples. Section 8.4.1 describes our process for writing and reasoning on Frag IR programs. We then give more details on two examples which make heavy use of memory fragments: to encode arrays (section 8.4.2) and singly linked lists (section 8.4.3).

8.4.1 Writing programs in Frag IR

Figure 8.11 summarizes the operations we perform with our examples. We extract the DTS of Frag IR and our passes to OCaml and link them with a printer and a parser. We use a human-readable syntax close to the one used in this document (figure 6.2). The syntax is quite verbose, since Frag IR is designed as an intermediate representation rather than as a source language. In particular, there is no type inference, so each variable must be annotated with its type. Most programs are tests for our passes: we parse a source program, run a pass or a composition of passes and compare the result with an expected target program.

When writing programs directly in Frag IR, one must ensure that they are well-formed (that is, that they implement a DTS program). Our type-checker (section 8.3.1) automatically checks syntactical well-formedness conditions (such as the absence of unbound variables) but does not check the soundness of lemmas.

As explained in section 3.2.7, Frag IR programs cannot be directly executed. We test the behaviors of our programs by translating them into Flat IR and by using an interpreter on Flat IR (described in section 9.5). Note that, since Frag IR has a non-deterministic semantics, those tests do not guarantee that the behavior observed is the only one allowed by the program.

In order to prove the full well-formedness of our programs and to reason on them, we implemented in OCaml a printer which exports parsed programs into Rocq file containing their AS. In Rocq, we then choose interpretations for the abstract predicates, and prove the soundness of lemmas with respect to those interpretations. We can then build automatically the DTS of the program using Ltac tactics of Rocq. This enables reasoning on the semantics of the program. We only performed this Rocq part for a subset (25) of our test programs. Depending on the purpose of the program we:

- prove refinements between the semantics of two Frag IR functions, to justify the soundness of a transformation performed by hand
- prove that the semantics of the program refines a specification written directly with the operators of `SMonad`

However, this approach is limited to toy programs by the verbosity of Frag IR and by the slowness of our tactic-based proofs. For instance, the example on singly linked list described in section 8.4.3 is three hundred lines long in Frag IR, whereas an equivalent C program takes less than forty lines. The proof of its correctness represents a thousand lines of Rocq and is checked in around thirty seconds.

8.4.2 Example: arrays

An array is a data structure which stores `len` elements of a given type `t`. Each element is accessed through its index, an integer within $[0; \texttt{len})$. The length `len` of an array remains constant during its use, but may depend on the program execution. In memory (figure 8.12), an array is implemented as a continuous range of $\texttt{len} \times \texttt{sizeof t}$ bytes. The base pointer `p` of this range provides access to the data stored in the array. Indeed, elements are accessed through pointer arithmetic: the element with index `i` is stored at address $\texttt{p} + \texttt{i} \times \texttt{sizeof t}$.

Figure 8.13 reproduces a toy C program involving arrays. This program starts by reading an integer `n`, and allocates two arrays of `n` integers (if `n` is not too large), with base pointers `p0` and `p1`. Using auxiliary functions, it then fills the first array with input values, copies them to the second array and

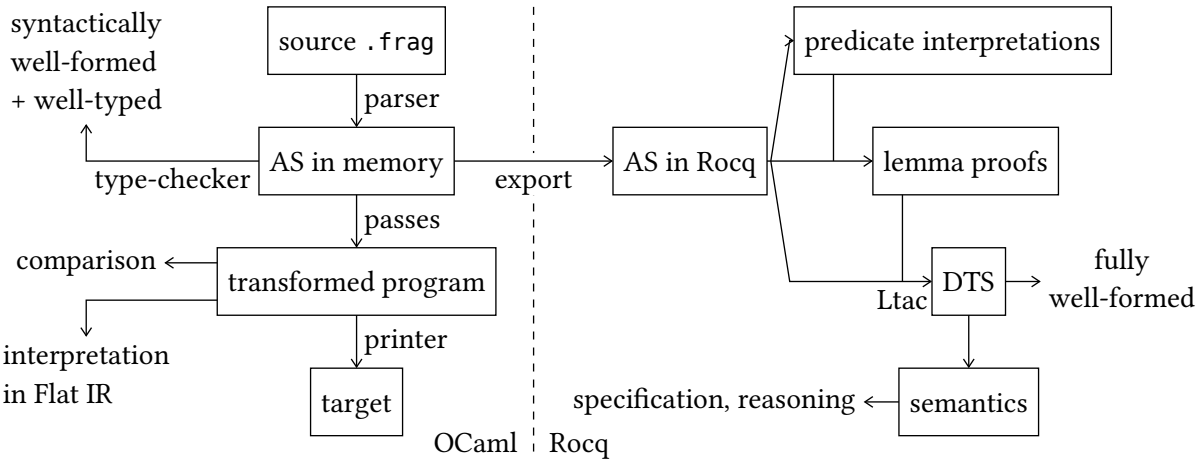


Figure 8.11: Processing of our examples

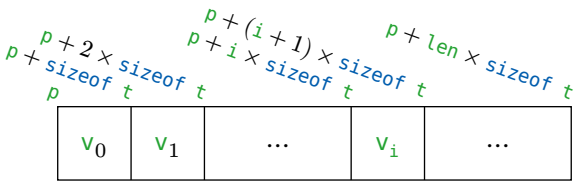


Figure 8.12: Memory implementation of an array. The array has base pointer `p`, length `len` and stores elements $(\texttt{v}_\texttt{i})_{0 \leq \texttt{i} < \texttt{len}}$ of type `t`. Each square represents a range of `sizeof t` bytes, starting at the address annotated above the top-left corner, and which encode the specified value.

outputs its content. The observable behavior of this program (excluding failures) is thus to read an integer `n`, then to read `n` integers and finally to output those `n` integers. In C, the syntax `p[i]` is a shorthand for pointer arithmetic to access the $\texttt{i}^{\text{nth}}$ element of an array with base pointer `p`.
We encoded this program in Frag IR [compiler/test/src/example/array.frag]. This encoding is significantly longer (around two hundred lines) than the equivalent C program of figure 8.13. In the following, we only describe a representative subset.

Our Frag IR program starts with the declaration of abstract predicates:

```
predicate i64_array (* (p : ptr, len : i64) : mem *)
predicate i64_array_without (* (p : ptr, len : i64, p1 : ptr) : mem *)
predicate valid_len (* (len : i64) : prop *)
```

We use the abstract memory predicate `i64_array` to represent arrays storing some `i64`s. We will interpret these predicates so that a memory fragment with type `mem{i64_array} p len` is the memory implementation of an array with base address `p` and length `len`. We also declare a memory predicate `i64_array_without`, interpreted so that `mem{i64_array_without} p len p1` represents an array with base address `p` and length `len`, but without the bytes corresponding to the element at address `p1`. The abstract proposition `valid_len` is used to propagate upper bounds on the length of arrays. Indeed, the test `n` ≤ 100 at the beginning of the program is necessary to justify that some operations (e.g., `n` $\times$ `sizeof i64`) do not overflow. We will later define formally the interpretation of the predicates in Rocq. However, it is not described in the program syntax. Even the signatures of the predicates are only commented informally.

We then define a small library of lemmas and functions which provide basic operations on arrays. In particular, we enable accesses to the elements of an array (both for reading and writing) with the following definitions:

```c
void input_array(int *p, unsigned len) {
  for (unsigned i = 0; i < len; ++i) p[i] = input();
}

void output_array(int *p, unsigned len) {
  for (unsigned i = 0; i < len; ++i) output(p[i]);
}

void copy_array(int *p_src, int *p_trg, unsigned len) {
  for (unsigned i = 0; i < len; ++i) p_trg[i] = p_src[i];
}

int main(void) {
  int res = 0;
  unsigned n = input();
  if (n <= 100) {
    int *p0 = malloc(n * sizeof(int));
    if (p0 != NULL) {
      int *p1 = malloc(n * sizeof(int));
      if (p1 != NULL) {
        input_array(p0, n);
        copy_array(p0, p1, n);
        output_array(p1, n);
        free(p1);
      } else res = 1;
      free(p0);
    } else res = 1;
  } else res = 1;
  return res;
}
```

Figure 8.13: C program equivalent to our example on arrays [compiler/test/src/example/array.c]

```
fn array_extract_begin(
     p : ptr, ghost len : i64, i : i64,
     ghost a0 : mem{i64_array} p len,
     ghost LT : i < len)
  : (p1 : ptr,
     ghost c : cell{i64} p1, ghost a1 : mem{i64_array_without} p len p1)
{
  let SIZE_INT : i64, ghost EQ_SIZE_INT = const{8}();
  let ofs      : i64, ghost EQ_ofs      = mul{i64}(i, SIZE_INT);
  let p1       : ptr, ghost EQ_p1       = add{ptr}(p, ofs);
  let ghost c  : cell{i64} p1, ghost a1 : mem{i64_array_without} p len p1 = lemma {
      (p : ptr, len : i64, i : i64, SIZE_INT : i64, ofs : i64, p1 : ptr,
       a0 : mem{i64_array} p len,
       LT          : i < len,
       EQ_SIZE_INT : SIZE_INT = const{8},
       EQ_ofs      : ofs      = mul{i64} i SIZE_INT,
       EQ_p1       : p1       = add{ptr} p ofs)
    : (c : cell{i64} p1, a1 : mem{i64_array_without} p len p1)
      consume (a0)
    }(p, len, i, SIZE_INT, ofs, p1, a0, LT, EQ_SIZE_INT, EQ_ofs, EQ_p1);
  return (p1, c, a1)
}

lemma array_extract_end
```

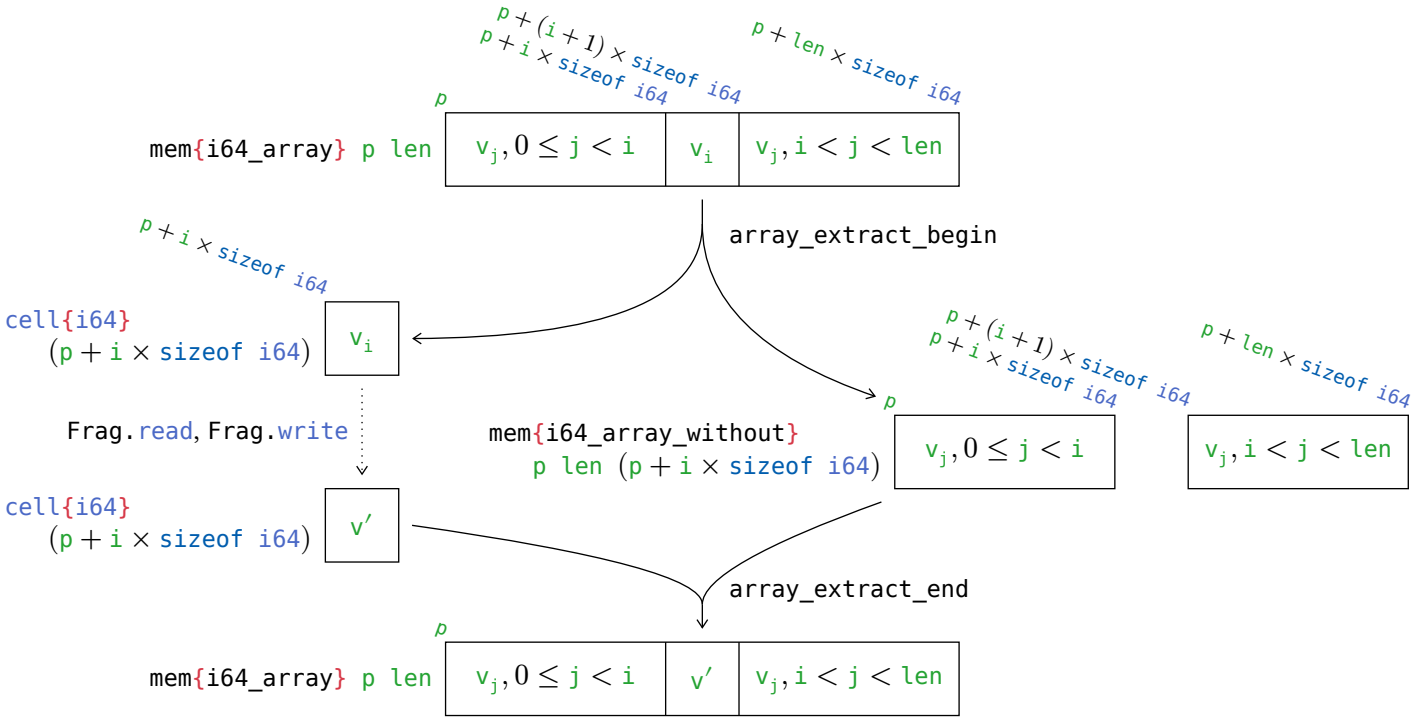


Figure 8.14: Pattern used to access array elements in Frag IR

```
    (p : ptr, len : i64, p1 : ptr,
     c : cell{i64} p1, a0 : mem{i64_array_without} p len p1)
  : (a1 : mem{i64_array} p len)
    consume (c, a0)
```

Using those operations, an element with index `i` is accessed by "extracting" the corresponding `cell`. This is achieved with the following pattern, also depicted in figure 8.14. We first call `array_extract_begin` to compute the address `p1` of the element. As a side effect, this function consumes the memory fragment `a0` and returns a cell `c` corresponding to the element and a fragment `a1` with type `mem{i64_array_without} p len p1` containing the remaining of the array. The cell `c` can then be used to read or write the value with index `i`. The access is terminated by calling `array_extract_end`, which consumes a cell at address `p1` (either the one returned by `array_extract_begin`, or a cell derived from it using `Frag.write`) and the remainder of the array. The lemma merges the two fragments to rebuild a full `mem{i64_array}`. Note that we do not have access to the full array between `array_extract_begin` and `array_extract_end` (accessing the original array would break the linearity condition). This pattern can thus only be used to access a single element of an array at a given time. This is expected, since having `cell`s for two elements at the same time is only possible if they are non-aliasing, that is, if they have distinct indexes. This is similar to restrictions of the Rust language on mutable references to elements of `slices`[19], illustrated in figure 8.15.

The operation `array_extract_begin` is defined as a function since it computes the address `p1`. It takes as ghost argument the length `len` of the array and has a pre-condition `i < len` on the index accessed. It starts by computing `p1` using pointer arithmetic. A lemma then splits the memory fragment storing the array to extract the cell at index `i`. This lemma takes as pre-condition the bound on `i` and the equalities reflecting the computation of `p1`. The operation `array_extract_end` is defined as a top-level lemma since it is a purely ghost operation. In our compilation pipeline, it thus does not directly generate any instruction in the target Frag IR program.

[19] a safe Rust type similar to our arrays, https://doc.rust-lang.org/1.92.0/book/ch04-03-slices.html

```
fn foo(a : &mut[i64], i0 : usize, i1 : usize) {
  let x0 = &mut a[i0];
  let x1 = &mut a[i1];
  // error[E0499]: cannot borrow `a[_]` as mutable more than once at a time
  println!("{x0} {x1}");
}
```

Figure 8.15: Mutable borrows to elements of a `slice` in Rust (1.92.0). The borrow checker rejects this program at compile time, because it would break the aliasing discipline of Rust when `i0` and `i1` are equal.

Using our basic operations on arrays, we implement the auxiliary functions of figure 8.13. In Frag IR, they have the following signatures:

```
fn input_array(p : ptr, len : i64, ghost a : mem{i64_array} p len)
  : (ghost a : mem{i64_array} p len)
fn output_array(p : ptr, len : i64, ghost a : mem{i64_array} p len)
  : (ghost a : mem{i64_array} p len)
fn copy_array(
     p_src : ptr, p_trg : ptr, len : i64,
     ghost src : mem{i64_array} p_src len, ghost trg : mem{i64_array} p_trg len)
  : (ghost src : mem{i64_array} p_src len, ghost trg : mem{i64_array} p_trg len)
```

Compared to the C functions, their arguments and results contain not only the pointers used to access the arrays but also memory fragments storing their content. Since all fragments taken as argument are consumed, the functions return new fragments even for the arrays that are not modified (`a` in `output_array` and `src` in `copy_array`). Like in C, the implementation of those functions have similar loop structures. As an example, we encode `input_array` as:

```
fn input_array(p : ptr, len : i64, ghost a : mem{i64_array} p len)
            : (ghost a : mem{i64_array} p len) {
  let i, _ = const{0}(); let ONE : i64, _ = const{1}();
  krec {
    loop(i : i64, ghost a : mem{i64_array} p len):
      let c : bool, ghost EQ_c = u_lt{i64}(i, len);
      if c Hc
      then {
        let ghost LT : i < len = lemma{i_lt_test}(i, len, c, EQ_c, Hc);
        let p1 : ptr,
            ghost c : cell{i64} p1, ghost a : mem{i64_array_without} p len p1
          = fcall array_extract_begin(p, len, i, a, LT);
        let v : i64 = input{i64}();
        let ghost c : cell{i64} p1 = Frag.write{i64}(p1, c, v);
        let ghost a : mem{i64_array} p len
          = lemma{array_extract_end}(p, len, p1, c, a);
        let i : i64, _ = add{i64}(i, ONE);
        kcall loop(i, a)
      } else return (a)
  }(i, a)
}
```

We use `krec` to implement the loop. For each index, we read an input value `v` and write it in the array using the pattern based on `array_extract_begin` and `array_extract_end`. The bound `LT` on the index `i` (requested as pre-condition by `array_extract_begin`) is derived using a `lemma` from the branch hypothesis `Hc` of the guard of the loop.

The `main` function has the same structure in Frag IR as in C. Calls to `malloc` are replaced by the primitive operation `Frag.malloc`. When an allocation fails, we use a lemma to consume the empty memory fragment returned by `Frag.malloc` in order to satisfy the linearity criterion.

Using our compilation pipeline, we obtain a Flat IR program, which uses an implicit global memory instead of memory fragments. It is thus similar to the C version. In particular, we obtain Flat IR functions with the same signatures as in C. The Flat IR program remains more verbose (around a hundred lines) than the C version, because expressions are decomposed as sequences of operations, `krec` is used instead of `for` loops and the auxiliary functions on array (e.g., `array_extract_begin`) are not inlined. For comparison, the representation of the program takes roughly 100 lines in LLVM IR and 60 lines in CompCert RTL.

Using our printer implemented in OCaml, we export the AS of the source Frag IR program in Rocq. We define formally the interpretations of the abstract predicates, following the informal description we gave. Concretely, this choice is done by defining the global function `te.int` used in the implementation relation of types. For instance, we choose the semantics of `valid_len` so that for any integer `len : I64.t`:

$$\texttt{Vsc Vunit} \in \texttt{prop\{valid_len\}}\ \texttt{len} \quad \leftrightarrow \quad \texttt{len} \times \texttt{sizeof i64} < 2^{64}$$

For the abstract memory fragments, we first define predicates indexed by the data stored in the fragments, represented as Rocq lists of integers. That is, we define the following Rocq predicates:

```
i64_array (p : Ptr.t) (len : I64.t) (data : list I64.t) (h : heaplet) : Prop :=
  ∃bs : Byte.t^(len×sizeof i64).
  h = hrange p bs ∧ is_n_i64_encoding len data bs

i64_array_without
  (p : Ptr.t) (len : I64.t) (p₁ : Ptr.t) (data₀ data₁ : list I64.t) (h : heaplet) : Prop :=
  ∃(i : ℕ) (bs₀ : Byte.t^(i×sizeof i64)) (bs₁ : Byte.t^((len − i −1)×sizeof i64)).
  h = hrange p bs₀ ⊎ hrange (p + (i + 1) × sizeof i64) bs₁ ∧
  i < len ∧ p₁ = p + i × sizeof i64 ∧ valid_range p (len × sizeof i64) ∧
  is_n_i64_encoding i data₀ bs₀ ∧ is_n_i64_encoding (len − i − 1) data₁ bs₁
```

where the auxiliary predicate `is_n_i64_encoding n data bs` expresses that the bytes `bs` encode a list `data` of `n` integers. It is formally defined by:

$$\texttt{is_n_i64_encoding}\ (\texttt{n} : \mathbb{N})\ (\texttt{data} : \texttt{list I64.t})\ (\texttt{bs} : \texttt{Byte.t}^{\texttt{n}\times\texttt{sizeof i64}}) : \texttt{Prop} :=$$
$$\texttt{data} = \left(\texttt{decode}\ \left(\left(\texttt{bs}_{\texttt{i}\times\texttt{sizeof i64}+\texttt{j}}\right)_{0\le \texttt{j} < \texttt{sizeof i64}}\right)\right)_{0\le \texttt{i} < n}$$

The Rocq predicate `i64_array` is thus parameterized by the list `data` of the integers stored in the array and expresses that the heaplet `h` is a range containing their encoding. The predicate `i64_array_without` is parameterized by two lists: the integers $\texttt{data}_0$ before the element extracted (which has index `i`) and the integers $\texttt{data}_1$ after it. It asserts that its argument $\texttt{p}_1$ is the address of the extracted element `i` and that `h` is the disjoint union of the two ranges storing the remaining elements. We also assert the validity of the full range, since it is not implied by the validity of `h`.
The abstract memory predicate only describes the shape of the memory fragments, not their data. We define their interpretation by quantifying existentially over the stored data:

```
Vmem h ∈ mem{i64_array} p len ↔
  ∃data : list I64.t. i64_array p len data h ∧ h≠⊥
Vmem h ∈ mem{i64_array_without} p len p₁ ↔
  ∃data₀ data₁ : list I64.t. i64_array_without p len p₁ data₀ data₁ h
```

```
Spec.input_array (n : ℕ) : FragIR.S ∅ (list I64.t)
Spec.input_array 0 := ret []
Spec.input_array (n + 1) :=
  x ← external (Input (Int S64)); xs ← Spec.input_array n; ret (x :: xs)

Spec.output_array (l : list I64.t) : FragIR.S ∅ unit
Spec.output_array [] := ret tt
Spec.output_array [x; xs] := external (Output (Vint (Int64 x)));; Spec.output_array xs

Spec.main : FragIR.S ∅ unit :=
  len ← external (Input (Int S64));
  { ret 1    if len > 100
  { ok ← any bool; { l ← Spec.input_array len; Spec.output_array l;; ret 0 if ok
                   { ret 1                                              otherwise
```

Figure 8.16: Specification of our toy program on arrays

We then prove that all lemmas used in the program are correct with respect to those interpretations. For instance, after simplification, the soundness condition of the lemma in `array_extract_begin` becomes:

```
∀p len i (data_a0 : list I64.t) (h_a0 : heaplet).
  i64_array p len data_a0 h_a0 ∧ h_a0≠⊥ ⇒
  i < len ⇒
  ∃(bs_c : Byte.t^sizeof i64) (data_a1_0 data_a1_1 : list I64.t) (h_a1 : heaplet).
  i64_array_without p len (p + i × sizeof i64) data_a1_0 data_a1_1 h_a1 ∧
  h_a0 = hrange (p + i × sizeof i64) bs_c ⊎ h_a1
```

Note that in this soundness condition, the values `bs_c`, $\texttt{data_a1}_0$ and $\texttt{data_a1}_1$ stored in the output fragments are existentially quantified. When reasoning on programs, we will need to know those values. For Frag IR lemmas which produce memory fragments, we thus prove stronger Rocq lemmas, which imply their soundness condition but describe the data stored in the output fragments. For instance, our Rocq lemma corresponding to the Frag IR lemma in `array_extract_begin` is:

Lemma 8.1 The soundness condition of the lemma in `array_extract_begin` is satisfied, with output fragments (cell and remainder of the array) containing subparts of the data stored in the input array:

```
∀p len i (data_a0 : list I64.t) (h_a0 : heaplet).
  i64_array p len data_a0 h_a0 ∧ h_a0≠⊥ ⇒
  i < len ⇒
  ∃(bs_c : Byte.t^sizeof i64) (data_a1_0 data_a1_1 : list I64.t) (h_a1 : heaplet).
  i64_array_without p len (p + i × sizeof i64) data_a1_0 data_a1_1 h_a1 ∧
  data_a0 = data_a1_0 ++ decode bs_c :: data_a1_1 ∧ List.length data_a1_0 = i ∧
  h_a0 = hrange (p + i × sizeof i64) bs_c ⊎ h_a1
```

Idea of the proof After unfolding the definitions of `i64_array` and `i64_array_without`, we decompose `data_a0` as a concatenation of a prefix $\texttt{data_a1}_0$ and suffix $\texttt{data_a1}_1$ around the $\texttt{i}^{\text{nth}}$ element. We then use properties of `is_n_i64_encoding` and `hrange` related to concatenation.

After proving the soundness of all lemmas, we build the DTS `s_pr` of the program. This is done automatically, thanks to Ltac tactics we implemented. This has thus access to the semantics of the program and of its functions. We prove that this semantics refines a specification defined in `FragIR.S` (figure 8.16). This specification uses pure `list`s and operations of Rocq (in particular pattern matching and structural induction), the operator of `SMonad` (e.g., `bind`, `any`) and the operation `external` of `FragIR.S`. This specification defines several semantics, corresponding to the semantics of the program with the same names. Note that `copy_array` does not appear in this specification, since it is an internal operation of the program without any observable effects. In the specification `Spec.main`, we take into

account the possible failures of the program (if the input `len` is too large or if `Frag.malloc` fails). In those cases, the program is allowed to abort by returning 1. In particular, the program which performs an input and always immediately returns 1 satisfies this specification. However, once the program starts reading the `len` integers, it is no longer allowed to abort.
The correctness of our Frag IR implementation is thus expressed by the following refinement:

Lemma 8.2

$$\texttt{Spec.main} \gtrsim_{(\lambda \texttt{rcode res.res=[rcode]})} [\![\texttt{s_pr}]\!]_{\text{prog}} \ []$$

Idea of the proof We use the relational reasoning approach of `SMonad`. We give a specification to each program function, as a refinement with a semantics defined directly in `FragIR.S`.
For instance, the specification of `input_array` is [CompilerTest.progs.example.array.sem:110]:

$$\forall \texttt{p len data_ini h_a0}.\ \texttt{i64_array p len data_ini h_a0} \Rightarrow \texttt{Spec.input_array len} \gtrsim_{\texttt{Post}} [\![\texttt{s_pr}, \texttt{input_array}]\!]_{\text{prog}}\ [\texttt{Vsc (Vptr p)}; \texttt{Vsc (Vint (Int64 len))}; \texttt{Vmem h_a0}]$$

where the post-condition `Post` on the result of the function is defined by:

$$\frac{\texttt{i64_array p len data h_a1}}{\texttt{Post data [Vmem h_a1]}}$$

This specification ensures that `input_array` performs `len` input events and returns a memory fragment `h_a1` containing the array of the results.
Since `copy_array` implements a deterministic operation without external effects, its specification is a refinement of `ret tt`.
The body of each auxiliary function is a loop encoded using `krec`. Their semantics is thus an `iter`. However, there is no corresponding `iter` in the specifications, which are either `ret tt` or defined by structural induction. Indeed, those loops are bounded. We thus do not use `ref_iter` and instead reason by induction, using `iter_unfold` at each step.
In Frag IR, fragments returned by a lemma are chosen non-deterministically among those satisfying `Lem.post`. This post-condition ensures that the fragments have the shape expected but does not give direct information about their values. We prove that the fragments chosen non-deterministically store the values we expect. To this end, we use our Rocq lemmas guaranteeing the existence of fragments containing the expected values (e.g., lemma 8.1). However, the results quantified existentially in those lemmas are a priori distinct from those chosen by the semantics of the Frag IR lemma. We show that this choice is in fact deterministic, and that we thus have an equality between the two sets of results. We leverage the equality with the input footprint ensured by `Lem.post` and our Rocq lemmas. When a single fragment is returned, we directly obtain by transitivity the equality desired. However, when multiple fragments are returned, we obtain an equality between disjoint unions and have to derive equalities between their components. In our program, there is only one such lemma, the one used in `array_extract_begin` and which returns a `cell` and a `mem{i64_array_without}`. After executing the semantics of this lemma and using lemma 8.1 [CompilerTest.progs.example.array.sem:88], we obtain an equality $\texttt{hrange p1 bs_c} \uplus \texttt{h_a1} = \texttt{hrange p1 bs_c}' \uplus \texttt{h_a1}'$, where `bs_c` and `h_a1` (resp. $\texttt{bs_c}'$ and $\texttt{h_a1}'$) are the bytes of `c` and the heaplet of `a1` chosen by the semantics of the Frag IR lemma (resp. existentially quantified in lemma 8.1). We derive $\texttt{bs_c} = \texttt{bs_c}'$ and $\texttt{h_a1} = \texttt{h_a1}'$ using properties of `hrange`. In the general case, as described in section 3.2.7, the result types of lemmas should be restrictive enough to ensure the correctness of the program despite the non-deterministic semantics of the lemmas. For correct programs, we thus expect that the shapes of the output fragments ensure the unicity of the decomposition of the heaplets.

FEEDBACK SB260816

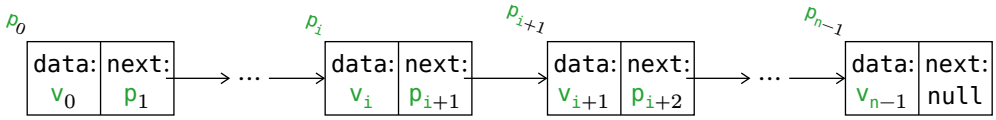


Figure 8.17: Memory implementation of a singly linked list. The list has head p_0, and stores n elements $(v_i)_{0 \leq i < n}$.

> Lien avec les travaux basés sur des raffinements ? Dijkstra et Wirth à la fin des années 60, formalisée dans les années 80 par Ralph Back, Caroll Morgan, JR Abrial (et la "méthode B" puis "event B"), etc.

Contrary to tests by interpretation, proving refinements (such as lemma 8.2) covers all behaviors of the program. That is, such refinement quantifies universally on all kinds of non-determinisms: the external non-determinism related to the input provided by the environment (the answer to the external events) and the internal non-determinism of Frag IR (for `Frag.malloc` and `lemma`).

8.4.3 Example: singly linked lists

A linked list is a data structure which stores a variable number of elements of a given type `t`. Contrary to arrays, linked lists do not store their elements at consecutive addresses in the memory. Instead, as depicted in figure 8.17, each element is stored in an individual list cell, and there is no fixed relation between the addresses of the list cells. In a singly linked list, each cell is a structure containing two fields: its `data` (of type `t`), and a pointer to the `next` cell in the list. The end of the list is signaled by storing the null pointer in the field `next` of the last cell. The head of a list is a pointer to its first cell, or the null pointer if the list is empty. A list can be traversed from its head by following the fields `next` of the cells. Contrary to arrays, linked lists support the insertion and removal of elements, by changing the head and the fields `next`.
Figure 8.18 reproduces a toy C program involving singly linked lists. List cells are represented by the structure type `list_cell`. The type `linked_list` is used to store entries of linked lists, it is an alias for pointers to list cells. The program reads an arbitrary number of non-zero inputs and stores them in a singly linked list. Each input is added at the beginning of the list, in a cell allocated using `malloc`. When a `0` is read, the program traverses the list to output the stored values. Finally, it performs a second traversal to free the memory allocated for the cells. The observable behavior of this program is thus to output in reverse order (because we add inputs at the beginning of the list) an input list of integers terminated by a `0` (which is not part of the list).

We encode this program in Frag IR, in a way similar to the example on arrays. In particular, the memory implementation of singly linked lists and list cells are represented by memory fragments:

```
predicate linked_list (* (head : ptr, next : ptr) : mem *)
predicate list_cell (* (addr : ptr, next : ptr) : mem *)
```

A fragment with type `mem{linked_list} p p_next` stores a segment of a linked list whose head is `p`. A segment is a subpart of a list, which may stop at a cell whose field `next` is not the null pointer. We index the segments by the value `p_next` of the pointer stored in their last cell. If the segment is empty, `p_next` $=$ `p`. A linked list is a segment for which `p_next` is null. A `mem{list_cell} p p_next` is a list cell at address `p` and whose field `next` is `p_next`. Since we allocate the cells individually, such fragment also contains the permission to deallocate the memory associated to the cell (that is, the range $[$`p`$;$ `p` $+$ $16)$ since our list cells have a size of 16 bytes).

We then define basic operations on lists and cells, for instance:

```
lemma intro_nil(p : ptr) : (ll : mem{linked_list} p p)

fn intro_cons
```

```c
struct list_cell { int data; struct list_cell* next; };
typedef struct list_cell list_cell;
typedef list_cell *linked_list;

linked_list input_list() {
  linked_list p = NULL;
  int i;
  while(i = input()) {
    list_cell *p0 = malloc(sizeof(list_cell));
    if (p0 == NULL) break;
    p0->data = i;
    p0->next = p;
    p = p0;
  }
  return p;
}

void output_list(linked_list p) {
  for (; p != NULL; p = p->next) output(p->data);
}

void free_list(linked_list p) {
  while(p) {
    list_cell *p0 = p;
    p = p->next;
    free(p0);
  }
}

int main(void) {
  linked_list p = input_list();
  output_list(p);
  free_list(p);
}
```

Figure 8.18: C program equivalent to our example on singly linked lists [compiler/test/src/example/singly_linked_list.c]

```
      (ghost p0 : ptr, ghost p1 : ptr, ghost p2 : ptr,
       ghost hd : mem{list_cell} p0 p1, ghost tl : mem{linked_list} p1 p2)
    : (ghost ll : mem{linked_list} p0 p2) {
  let ghost JOINABLE : joinable hd tl = Frag.joinable(hd, tl);
  lemma {
      (p0 : ptr, p1 : ptr, p2 : ptr,
       hd : mem{list_cell} p0 p1, tl : mem{linked_list} p1 p2,
       JOINABLE : joinable hd tl)
    : (ll : mem{linked_list} p0 p2)
      consume (hd, tl)
    }(p0, p1, p2, hd, tl, JOINABLE)
}

fn elim_cons
      (p0 : ptr, ghost p2 : ptr,
       ghost ll : mem{linked_list} p0 p2,
       ghost NEMP : ¬prop{is_empty} ll)
    : (p1 : ptr, ghost hd : mem{list_cell} p0 p1, ghost tl : mem{linked_list} p1 p2)
  { … }
```

The lemma `intro_nil` enables creating a memory fragment containing an empty list. The returned fragment has an empty footprint, so the lemma returns it without consuming anything. The function `intro_cons` consumes a list cell `hd` and a list segment `tl` and merges them into the list segment where `hd` has been added at the beginning of `tl`. This function does not update the field `next` of `hd` and instead assumes (through the type of `hd`) that it already points to the head `p1` of `tl`. This is thus a ghost operation, which only reorganizes the memory fragments. It is defined as a function because we need to use `Frag.joinable`. Indeed, as explained in figure 3.11, memory fragments (here `hd` and `tl`) can only be merged if they are disjoint. Since we allocate list cells individually with `Frag.malloc`, we need `Frag.joinable` to request their disjointness. The function `elim_cons` performs the opposite of `intro_cons`: it splits a list `ll` into its first element `hd` and its tail `tl`. It takes a pre-condition `NEMP` which ensures that the list is not empty. It is defined as a function because it uses `Frag.read` to retrieve the head `p1` of the tail from the field `next` of `hd`.

Using those basic operations, we encode the auxiliary and main functions of figure 8.18. Their signatures additionally describe their use of memory fragments. For instance, `output_list` takes as argument and returns fragments containing the list:

```
fn output_list(p : ptr, ghost ll0 : mem{linked_list} p null)
            : (ghost ll1 : mem{linked_list} p null)
```

Our pipeline successfully compiles this program to Flat IR, which enables its execution by interpretation. We also export the AS of the source Frag IR program in Rocq. Like for arrays, we first define predicates on heaplets indexed by data they contain. We use them to interpret the abstract predicates of the program by existentially quantifying on data. For instance, we define:

```
list_cell (addr next : Ptr.t) (data : I64.t) (h : heaplet) : Prop :=
  ∃(bs_data : Byte.t^sizeof ptr) (bs_next : Byte.t^sizeof i64).
  h = hrange addr (bs_data ## bs_next) ⊎ (∅, [addr ↦ 16]) ∧
  decode bs_data = data ∧ decode bs_next = next

linked_list (head next : Ptr.t) (data : list I64.t) (h : heaplet) : Prop

linked_list head next [] h := (head = next ∧ h = emp)

linked_list head next data_hd :: data_tl h :=
  ∃(p1 : Ptr.t) (hd tl : heaplet).
  list_cell head p1 data_hd hd ∧ linked_list p1 next data_tl tl ∧ h = hd ⊎ tl
```

where `linked_list` is defined by recursion on `data`. We then prove the soundness of the lemmas contained in the program and build the DTS `s_pr`. We show that the semantics of the program refines a specification defined in `FragIR.S` (figure 8.19). This specification is defined similarly to the specification of the program on array (figure 8.16). In particular, it also uses pure `list`s and `Spec.output_list` is the same. A noteworthy difference is that we use `iter` in `Spec.input_list` since there is no bound on its loop[20].

Lemma 8.3

$$\texttt{Spec.main} \gtrsim_{(\lambda_\,\texttt{res.res}=[])} [\![\texttt{s_pr}]\!]_{\text{prog}}\ []$$

Idea of the proof Like for the example on arrays, by relational reasoning. We give a specification to each Frag IR function, using the definition of figure 8.19 and our data-indexed predicates on heaplets. For instance, the specification of `output_list` describes that it outputs the values contained in the list received as argument, and returns a list with the same head `p` and containing the same `data`:

$$\forall \texttt{p data h_ll0}.\ \texttt{linked_list p Ptr.null data h_ll0} \Rightarrow \texttt{Spec.output_list data} \gtrsim_{\texttt{Post}} [\![\texttt{s_pr}, \texttt{output_list}]\!]_{\text{prog}}\ [\texttt{Vsc (Vptr p)}; \texttt{Vmem h_ll0}]$$

[20] in this program, an alternative could be to find a bound using the finiteness of our set of concrete pointers

```
Spec.input_list : FragIR.S ∅ (list I64.t) :=
  iter (λl.
    x ← external (Input (Int S64));
    { ret (Break l)                                  if x = 0
    { ok ← any bool; { ret (Continue (x :: l))       if ok
                     { ret (Break l)                 otherwise) []

Spec.output_list (l : list I64.t) : FragIR.S ∅ unit := Spec.output_array l

Spec.main : FragIR.S ∅ unit := l ← Spec.input_list; Spec.output_list l
```

Figure 8.19: Specification of our toy program on linked lists

where the post-condition `Post` is defined by:

$$\frac{\texttt{linked_list p Ptr.null data h_ll1}}{\texttt{Post _ [Vmem h_ll1]}}$$

Most loops are proven correct by induction, except for `input_list`, where we use `ref_iter`.

9

Compilation to a lower-level language

Our intermediate representation Frag IR differs from standard low-level languages in several aspects. It features propositions and memory fragments and operations on them. Moreover, it does not have a global memory. Those specificities were introduced to propagate information and facilitate program transformations. Our goal, however, is ultimately to execute programs on a real machine. Without restrictions, Frag IR programs are not executable because of abstract memory fragments (as explained in section 3.2.7). Fortunately, memory fragments can be implemented with a global memory if they are used linearly.

We implemented a compilation pipeline `frag_to_flat` (figure 9.1) from Frag IR to another intermediate representation, called Flat IR (section 9.1). This representation still uses functional SSA but features a global memory state and is closer to existing low-level languages. Flat IR programs generated by our pipeline only use standard operations. Our pipeline includes some optional passes on Frag IR inherited from the parameterized representation (block-sinking, DCE). We use the type-checker to sequence passes which do not guarantee formally the preservation of typing with passes which assume it. An optional pass `OptDefault` specific to Frag IR can be used to avoid uses of optional values in the final Flat IR program (section 9.2). The core of our pipeline is the translation from Frag IR to Flat IR (section 9.3). This translation defensively checks the linear usage of memory fragments, and inserts standard operations where operations specific to Frag IR are used. The complete removal of propositions and memory fragments is deferred to an erasure pass (section 9.4). The compilation of Flat IR to an existing low-level representation is left for future work, but we implemented a verified interpreter enabling the execution of the programs generated by our pipeline (section 9.5). The correctness of our pipeline is derived by composing the correctness of its passes (section 9.6).

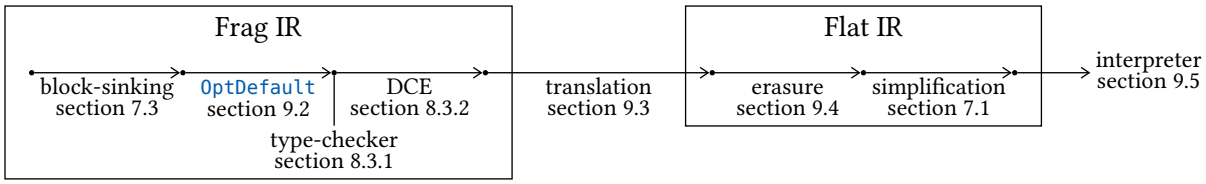


Figure 9.1: Compilation pipeline `frag_to_flat` from Frag IR to Flat IR [Compiler.Impl.Inst.Compil:138]

9.1 Flat IR

Flat IR is the target language of our compilation pipeline. It is designed to be similar to existing low-level representations, such as Cminor or VIR. The main difference with Frag IR is that Flat IR features a global memory and operators to access it. Like Frag IR, we define Flat IR as an instance of our parameterized functional SSA IR (chapter 6). We use the same types and values as for Frag IR, but extend the set of operators. Without restrictions, programs in Flat IR can thus contain propositions, memory fragments and ghost operations on them. We use this possibility during the translation of Frag IR, but we later remove those ghost parts to obtain programs in a subset of Flat IR which only involves operations on scalars.

Section 9.1.1 describes our formal representation of the global memory. In section 9.1.2, we use this memory type to define the semantics domain of Flat IR. Section 9.1.3 describes the operators of Flat IR and their semantics.

9.1.1 Flat memory

Formally, a state of the global memory of Flat IR is an element of the following type:

Definition 9.1 A *flat memory* is a pair $(\texttt{data}, \texttt{perm}) : \texttt{Mem.t}$ where:

$$\texttt{Mem.t} \coloneqq (\texttt{Ptr.t} \to (\texttt{Byte.t} \times \texttt{Mem.perm})) \times (\texttt{Ptr.t} \rightharpoonup \mathbb{N})$$
$$\texttt{Mem.perm} ::= \texttt{Empty} \mid \texttt{Writable}$$

The first component (`data`) describes the value and access permission of each byte. The second component (`perm`) is a partial function describing deallocation permissions, like in `heaplet` (definition 3.3).

Contrary to `heaplet`, the `data` component of `Mem.t` is a total function. However, programs can only access bytes with permission `Writable`. The formalization of the memory is inspired by the type `mem` of CompCert [68]. However, we currently only support a single non-empty permission (`Writable`), whereas CompCert has a more fine-grained permission system. Several aspects of Flat IR further simplify our memory model:

- `Mem.t` models an unstructured memory, which maps data to a single range of machine integers (hence the name "flat"). This is a consequence of our use of concrete pointers. With logical pointers (like in CompCert [62]), the memory is instead split between disjoint blocks.
- All values that can be stored in memory have a byte encoding. This enables associating a `Byte.t` to every memory address. This is not possible for memory models supporting values which cannot be encoded as a sequence of bytes (e.g., logical pointers or poison values, see section 2.2).

9.1.2 Domain

We build a monadic domain `FlatIR.S` supporting the effects of Flat IR. We use the transformer `State` (described in section 4.5.2) to add a global state with type `Mem.t` to the domain `FragIR.S` of Frag IR[21].

$$\texttt{FlatIR.S} \coloneqq \texttt{State.S Mem.t FragIR.S}$$

Deriving `FlatIR.S` from `FragIR.S` has several benefits:

- `FlatIR.S` inherits the effects of `FragIR.S`. It thus supports the effects of divergence, non-determinism, undefined behaviors and internal and external events. It additionally has a global memory state.
- We obtain a lifting operation `State.lift` which converts semantics defined in `FragIR.S` into semantics in `FlatIR.S`. As mentioned in section 4.5.2, this operation is only available for semantics without events in their signature. However, it works for semantics with external events since they

[21] In Rocq, we actually define `FlatIR.S` as a subset of this domain. Indeed, we reason modulo an equivalence relation on memories `Mem.t` (instead of using the axiom of functional extensionality) and only consider semantics respecting this equivalence.

are hidden from the signature in `FlatIR.S`. We leverage this operation to define the semantics of operators inherited from Frag IR (section 9.1.3).
- By unfolding the definition of `FlatIR.S`, semantics in `FlatIR.S` can be interpreted as semantics in `FragIR.S`. We signal those reinterpretations using an identity conversion `to_Frag [E A] (u : FlatIR.S E A) : Mem.t → FragIR.S (State.ev Mem.t E) (A × Mem.t)`. This conversion enables using the notion of refinement of `FragIR.S` to compare semantics in `FragIR.S` with semantics in `FlatIR.S`. We leverage this idea for the proof of the translation from Frag IR to Flat IR (section 9.3).

Like with $[\![.]\!]^{\text{Frag}}_{\text{prog}}$, we simplify the unfolding of the semantics of Flat IR programs to `LTS.t`, using the fact that they do not emit internal events. By choosing non-deterministically the initial memory and ignoring the final one, we obtain the same signature as $[\![.]\!]^{\text{Frag}}_{\text{prog}}$. This enables handling uniformly Frag IR and Flat IR programs in compilation pipelines.

```
[[pr : Prog.s a]]^Flat_prog (vargmain : val^a.arg) : LTS.t external_event val^a.res :=
  m_0 ← any Mem.t;
  (r, _) ← map_ev FlatIR.get_ev (to_LTS (to_Frag ([[pr]]_prog vargmain)));
  ret r
```

where the empty family of internal events is removed from the signature of the semantics using a bijective function on events `FlatIR.get_ev : EventFam.Fun.t (State.ev Mem.t ∅ + external_event) external_event`.

9.1.3 Operators

Flat IR extends the operators of Frag IR with operators that access the global memory. Those memory operators include standard `Flat.read`, `Flat.write`, `Flat.malloc` and `Flat.free`. We also add a ghost operator `Flat.getalloc` which is used transitorily during the translation from Frag IR to Flat IR (section 9.3). In the AS, this set of operators is defined by an inductive type `op_sym1` (figure 9.2). Like in Frag IR, operator symbols `Op.sym_t` also contain a liveness annotation.

- Operators of Frag IR (in particular scalar operators) are injected into the set of operators of Flat IR by the constructor `Ofr`. They have the same signature as in Frag IR and their semantics is defined from their semantics in `FragIR.S` using `State.lift`.
- `Flat.read` and `Flat.write` are partial operations, contrary to `Frag.read` and `Frag.write`. They have UB if the accessed memory range is not valid (in particular if the base pointer is null) or if any address in this range has access permission `Empty`. In order to access the global memory, we unfold the definition of `FlatIR.S` and define their semantics using operators of `FragIR.S`. We thus take as argument and return the states of the global memory before and after the operations. For `Flat.read`, we retrieve the memory bytes in the accessed range and decode them as a scalar of the specified type `t`. The memory is unchanged.

```
Flat.read{t}(p: ptr) : (r: t)
to_Frag ([[Flat.read{t}]]_op [] [Vsc (Vptr p)]) (data, perm) :=
  { ret ([Vsc (Scalar.to_t t (decode ((data (p + i))_1)_{0≤ i < sizeof t}))], (data, perm))
      if valid_range p (sizeof t) and ∀0 ≤ i < sizeof t. (data (p + i))_2 = Writable
  { ub otherwise
```

```
op_sym1 :=
  | Ofr (op : op_sym1)          operators inherited from Frag IR
  | Oread (ty : Scalar.Ty.t)    Flat.read{ty}
  | Owrite (ty : Scalar.Ty.t)   Flat.write{ty}
  | Omalloc                     Flat.malloc
  | Ofree                       Flat.free
  | Ogetalloc                   Flat.getalloc
Op.sym_t := {op_live : Liveness.t; op_sym : op_sym1}
```

Figure 9.2: AS of operations of Flat IR

Syntactically, `Flat.write` does not return any value. However, it updates the bytes of the accessed memory range to store the encoding of the scalar given as argument.

$$\begin{array}{l}\texttt{Flat.write\{t\}(p: ptr, v: t) : ()}\\ \texttt{to_Frag}\ \left([\![\texttt{Flat.write\{t\}}]\!]_{\text{op}}\ []\ [\texttt{Vsc (Vptr p)}; \texttt{Vsc v}]\right)\ (\texttt{data}, \texttt{perm}) :=\\ \quad \begin{cases} \texttt{ret}\ \left([], \left(\left[\texttt{data}, \left(\texttt{p} + \texttt{i} \mapsto \left((\texttt{encode v'})_{\texttt{i}}, \texttt{Writable}\right), 0 \le \texttt{i} < \texttt{sizeof t}\right)\right], \texttt{perm}\right)\right) \\ \quad \text{if } \texttt{valid_range p (sizeof t)}, \forall 0 \le \texttt{i} < \texttt{sizeof t}.\, (\texttt{data}\ (\texttt{p} + \texttt{i}))_2 = \texttt{Writable} \\ \quad \text{and } \texttt{Scalar.of_t v} = (\texttt{t}, \texttt{v'}) \\ \texttt{ub} \text{ otherwise} \end{cases}\end{array}$$

- `Flat.malloc` is non-deterministic: it either fails and returns the null pointer or returns the base pointer of a valid range not currently allocated. Those two cases are described by a predicate `is_malloc`:

$$\frac{}{\texttt{is_malloc Ptr.null sz m m}}$$

$$\frac{\begin{array}{c}\texttt{valid_range p sz} \quad \texttt{perm}_0\ \texttt{p} = \varnothing \\ \forall \texttt{q} \in [\texttt{p}; \texttt{p} + \texttt{sz}).\, (\texttt{data}_0\ \texttt{q})_2 = \texttt{Empty} \wedge (\texttt{data}_1\ \texttt{q})_2 = \texttt{Writable} \\ \forall \texttt{q} \notin [\texttt{p}; \texttt{p} + \texttt{sz}).\, (\texttt{data}_0\ \texttt{q})_2 = (\texttt{data}_1\ \texttt{q})_2\end{array}}{\texttt{is_malloc p sz}\ (\texttt{data}_0, \texttt{perm}_0)\ (\texttt{data}_1, [\texttt{perm}_0, \texttt{p} \mapsto \texttt{sz}])}$$

Contrary to `Frag.malloc`, the semantics of `Flat.malloc` depends on other active allocations. Indeed, in case of success, the chosen range must be disjoint from other allocated ranges. A failed allocation does not change the memory. During a successful allocation, the values of the bytes of the allocated range may be changed non-deterministically. Their access permission is changed from `Empty` to `Writable` and a deallocation permission is associated to the returned pointer.

$$\begin{array}{l}\texttt{Flat.malloc(sz: i64) : (p: ptr)}\\ \texttt{to_Frag}\ \left([\![\texttt{Flat.malloc}]\!]_{\text{op}}\ []\ [\texttt{Vsc (Vint (Int64 sz))}]\right)\ \texttt{m}_0 :=\\ \quad (\texttt{p}, \texttt{m}_1) \leftarrow \texttt{any}\ \{(\texttt{p}, \texttt{m}_1) \mid \texttt{is_malloc p sz m}_0\ \texttt{m}_1\}; \texttt{ret}\ ([\texttt{Vsc (Vptr p)}], \texttt{m}_1)\end{array}$$

`Flat.free` is a partial operation. If its argument is not null, it checks that it is associated to a deallocation permission. This permission provides the size of the allocated range. We then drop the allocation permission and the access permissions of the addresses of the range.

$$
\begin{array}{l}
\texttt{Flat.free(p: ptr) : ()} \\
\texttt{to_Frag}\ \left(\llbracket \texttt{Flat.free} \rrbracket_{\mathrm{op}}\ []\ [\texttt{Vsc}\ (\texttt{Vptr}\ \texttt{p})]\right)\ (\texttt{data}, \texttt{perm}) := \\
\quad \begin{cases}
\texttt{ret}\ ([], (\texttt{data}, \texttt{perm}))\ \text{if}\ \texttt{p} = \texttt{Ptr.null} \\
\texttt{ret}\ \left([], \left(\left[\texttt{data}, \left(\texttt{q} \mapsto \left((\texttt{data}\ \texttt{q})_1, \texttt{Empty}\right), \texttt{q} \in [\texttt{p}; \texttt{p} + \texttt{sz})\right)\right], [\texttt{perm}, \texttt{p} \mapsto \varnothing]\right)\right) \\
\quad \text{if}\ \texttt{perm}\ \texttt{p} = \lfloor \texttt{sz} \rfloor\ \text{and}\ \forall \texttt{q} \in [\texttt{p}; \texttt{p} + \texttt{sz}).\ (\texttt{data}\ \texttt{q})_2 = \texttt{Writable} \\
\texttt{ub}\ \text{otherwise}
\end{cases}
\end{array}
$$

- The operator `Flat.getalloc` is meant to be used after `Flat.malloc`. It takes as arguments the pointer `p` returned by `Flat.malloc` and the size `sz` of the allocated range. It returns a memory fragment with type `malloc p sz` corresponding to the freshly allocated range. If the allocation has failed (that is, if `p` is null), `Flat.getalloc` returns the empty heaplet. Otherwise, it retrieves the initial values of the allocated range from the global memory.

$$
\begin{array}{l}
\texttt{Flat.getalloc(p: ptr, sz: i64) : (ghost m: malloc p sz)} \\
\texttt{to_Frag}\ \left(\llbracket \texttt{Flat.getalloc} \rrbracket_{\mathrm{op}}\ []\ [\texttt{Vsc}\ (\texttt{Vptr}\ \texttt{p}); \texttt{Vsc}\ (\texttt{Vint}\ (\texttt{Int64}\ \texttt{sz}))]\right)\ (\texttt{data}, \texttt{perm}) := \\
\quad \begin{cases}
\texttt{ret}\ ([\texttt{Vmem}\ \texttt{emp}], (\texttt{data}, \texttt{perm}))\ \text{if}\ \texttt{p} = \texttt{Ptr.null} \\
\texttt{ret}\ \left(\left[\texttt{Vmem}\ \left(\texttt{hrange}\ \texttt{p}\ ((\texttt{data}\ (\texttt{p} + \texttt{i}))_1)_{0 \le \texttt{i} < \texttt{sz}}\right) \uplus (\varnothing, [\texttt{p} \mapsto \texttt{sz}])\right], (\texttt{data}, \texttt{perm})\right) \\
\quad \text{if}\ \texttt{valid_range}\ \texttt{p}\ \texttt{sz} \\
\texttt{ub}\ \text{otherwise}
\end{cases}
\end{array}
$$

9.2 A pass to remove optional values

In section 8.1.1.4, we introduced optional values in Frag IR to allow uninitialized variables while maintaining typing and functional SSA. Since Flat IR uses the same functional SSA structure, every variable must also be assigned a value. Like in Frag IR, optional values can be used to represent uninitialized variables. Alternatively, we implemented a pass `OptDefault` on Frag IR, whose goal is to ensure that no optional values are ultimately used in the target Flat IR program. This pass assigns some default values to variables whenever `Opt.none` is used. The pass is optional: if it is not activated, our pipeline still produces valid Flat IR programs, but containing optional values. Removing optional values could be useful to translate Flat IR programs to a representation with initialization constraints. Currently, the main interest of this pass is to demonstrate the possibility of performing program transformations in the presence of dependent types.

Although the `opt` transformer can be applied to any type, the pass `OptDefault` assumes that it is only used on scalar types. In particular, it does not support nested `opt`, like `opt (opt i64)`. To simplify the specification of the pass, we also check that the signature of the main function does not contain optional values. Our current use of optional values satisfies those assumptions. If the pass is applied to a program where they do not hold, it will likely raise an error. The results we proved also do not exclude that the pass could silently generate an ill-typed program, but we are not aware of such examples. However, we prove that when the pass succeeds, it produces a well-formed program which semantically refines the source.

The crucial property that enables the transformation is that non-ghost operations involving optional values preserve a kind of `lessdef` invariant (section 8.1.3.2). In our formalism, given two optional values $\texttt{o}_0$ and $\texttt{o}_1$ with the same type `opt ty`, we note `opt_le` $\texttt{o}_0$ $\texttt{o}_1$ if $\texttt{o}_0$ is less defined than $\texttt{o}_1$. Formally, this relation is generated by:

$$
\frac{}{\texttt{opt_le}\ (\texttt{Vopt}\ \varnothing)\ (\texttt{Vopt}\ \texttt{o})} \qquad \frac{}{\texttt{opt_le}\ (\texttt{Vopt}\ \lfloor \texttt{v} \rfloor)\ (\texttt{Vopt}\ \lfloor \texttt{v} \rfloor)}
$$

Informally, for all non-ghost operators we can always "replace" some of the optional values taken as arguments by more defined values. This replacement yields a target semantics which refines the source and which returns more defined values.
However, optional values can also appear in ghost contexts, in particular in dependent types. As explained in section 8.1.1.4, it is not possible to replace optional values by more defined values within types. In particular, `Opt.none` returns a post-condition `o = none`, and it would be invalid to change it to `some d = none` with a default value `d`. As a consequence, our pass does not replace directly the optional values. They are instead kept as ghost variables. In the target, we add variables (here with suffix `_v`) which store a more-defined value and propositions (here with prefix `LE_`) reflecting the invariant between the source and new variables. That is, we perform the following transformation on the binders:

```
(a : opt T)  →  (ghost a : opt T, a_v : T, ghost LE_a : opt_le a (some a_v))
```

where `opt_le` is here a proposition type in Frag IR, whose semantics is defined by the predicate `opt_le` in Rocq. Since we always initialize the new variables in the target (here `a_v`), we give them a type without `opt` and use the operator `some` in `opt_le`. For source binders containing multiple variables, we add new target variables for each non-ghost source variable with a type `opt`, annotate those variables as `ghost` and leave the other variables unchanged. We add the new variables at the end of the lists of binders. This simplifies the implementation of the transformation because source variables keep the same index and representation with `RRef.t`. For instance, binders for the results of `Opt.none<T>` are transformed as follows:

```
(a : opt T, ghost EQ_a : a = none)  →
(ghost a : opt T, ghost EQ_a : a = none, a_v : T, ghost LE_a : opt_le a (some a_v))
```

The original optional variables and the invariants `opt_le` are removed later in the pipeline, either by the DCE on Frag IR (section 8.3.2) if the optional variables are not used within ghost code, or at the latest, by the erasure on Flat IR (section 9.4).

Syntactically, aside from the binders, the only program parts affected by the transformation are the non-ghost operations on optional values: `Opt.some`, `Opt.none` and `Opt.pre_get`. Currently, we assume that `Opt.get` (without pre-condition) is not used in the source program.

TODO (rocq)

Support dans OptDefault

In the future, we may either support it in `OptDefault` or add a pass that encodes it using `assert` and `Opt.pre_get`, like in figure 8.6. All other operations are unaffected, and use the same variables as in the source. Figure 9.3 describes the transformation of the affected operations. The leading `let`s recall the types of the returned variables in the source and target, but are not part of the instructions. The pass introduces additional variables whenever an optional value appears in the source. Since `Opt.pre_get` takes as argument a variable `a : opt T`, its transformation in the target has access to additional variables `a_v : T` and `LE_a : opt_le a (some a_v)`. Reciprocally, `Opt.some` and `Opt.none` return a variable `a : opt T`, so their transformation returns additional variables.
In the transformation of `Opt.some`, the argument `v` given in the source is bound to the result `a_v`. The target still contains a ghost operation `Opt.some` to build the now-ghost result `a : opt T`. A lemma derives the invariant `LE_a : opt_le a (some v)` from the post-condition `EQ_a : a = some v` of `Opt.some`.
The transformation of `Opt.none` is similar, except that we use a default value `d` to initialize `a_v`. This value is built by an operation, which depends on the scalar type `T`. For instance, we use `const{0}()` to build an `i64` and `null()` for `ptr`.

Source → Target

```
let (
  a: opt T,
  ghost EQ_a: a = some v
) = Opt.some<T>(v)
```

→

```
let (
  ghost a: opt T, ghost EQ_a: a = some v,
  a_v: T, ghost LE_a: opt_le a (some a_v)
) = {
  let (ghost a: opt T, ghost EQ_a: a = some v) =
    ghost Opt.some<T>(v);
  let ghost LE_a: opt_le a (some v) = lemma {
    <T'>(v': T', a': opt T', EQ: a' = some v')
    : (LE_a': opt_le a' (some v'))
    }<T>(v, a, EQ);
  return (a, EQ_a, v, LE_a)
}
```

```
let (
  a: opt T,
  ghost EQ_a: a = none
) = Opt.none<T>()
```

→

```
let (
  ghost a: opt T, ghost EQ_a: a = none,
  a_v: T, ghost LE_a: opt_le a (some a_v)
) = {
  let (ghost a: opt T, ghost EQ_a: a = none) =
    ghost Opt.none<T>();
  let (d: T, …) = (* default value *)
  let ghost LE_a: opt_le a (some d) = lemma {
    <T'>(d': T', a': opt T', EQ: a' = none)
    : (LE_a': opt_le a' (some d'))
    }<T>(d, a, EQ);
  return (a, EQ_a, d, LE_a)
}
```

```
let (
  v: T,
  ghost EQ_a: a = some v
) = Opt.pre_get<T>(a, Hsome)
```

→

```
let (v: T, ghost EQ_a: a = some v) = {
  let (ghost EQ_a: a = some a_v) = lemma {
    <T'>(a': opt T', a_v': T',
         LE_a' : opt_le a' (some a_v'),
         Hsome': is_some a')
    : (EQ_a': a' = some a_v')
    }<T>(a, a_v, LE_a, Hsome);
  return (a_v, EQ_a)
}
```

Figure 9.3: Transformations of the operators on optional values during the pass `OptDefault`

For `Opt.pre_get`, we return the always initialized value `a_v`. The combination of the invariant `LE_a : opt_le a (some a_v)` and of the argument `Hsome : is_some a` ensures that `a` contains the value `a_v`.

We prove that when the pass succeeds, it generates a target program whose semantics refines the source semantics. Since it is a prerequisite for stating this result, we also prove that it preserves the well-formedness of the program. In particular, we prove the soundness of all lemmas introduced. We do not assume that the source is well-typed, but do not prove that the pass preserves the typing. Interestingly, the only semantic invariant needed for the proof of refinement in Rocq is that the source variables have the same values in the target. Indeed, relations between new initialized variables (`a_v`) and original source variables (`a`) are already encoded in the target program using propositions (`LE_a`). The only point where those relations are needed for justifying our invariant on the source variables is for the value `v` returned by `Opt.pre_get`. We prove that it has the same value in the source and target thanks to the post-condition `a = some a_v` of the lemma in the target.

The Rocq proof takes more than 800 lines. This number is arguably large given the simplicity of the invariant and the restricted number of affected operators. This is explained by several factors:

- The transformation must be performed for each function, and under the combinators of instructions (`let`, `if` and `krec`).
- The addition of new variables must be taken into account in instructions that are not directly affected. For instance, continuations and functions take additional arguments, which are added in `kcall` and `fcall`.
- We have to prove that the target program is well-formed. In particular, although we do not need any semantic invariant on the values of the new variables, we still have to justify that they are bound in the target.

9.3 Translation from Frag IR to Flat IR

Frag IR encodes information using operations with pre-conditions and memory fragments. Those features are not standard for low-level representations, so our compilation pipeline removes them. This is achieved thanks to the translation pass from Frag IR to Flat IR. This pass checks defensively that the source program uses memory fragments linearly (section 9.3.1). When this check succeeds, it transforms the operations appearing in the program (section 9.3.2). Operations with pre-conditions are replaced by their partial variants and operations on memory fragments are implemented with operations on the global memory. To simplify the translation, the removal of propositional variables and memory fragments is deferred to the erasure pass performed after. The Flat IR program generated by the translation thus performs all computationally relevant operations using standard operators, but still contains ghost code involving propositions and memory fragments.

9.3.1 Linearity condition

As explained in section 3.2.5, the implementation of memory fragments using a global memory requires that the program use them linearly [100]. This linearity condition is checked defensively by the translation pass. Currently, our Rocq development does not contain a standalone formalization of the linearity condition, but we describe here semi-formally the criterion that we expect the translation pass to check. When our linearity check succeeds, we associate to each program point a set of active variables satisfying the following property:

Definition 9.2 During an execution, a variable storing a memory fragment is *consumed* if:

- It is given as argument to `Frag.write`, `Frag.free` or `lemma` (as an argument within the `consume` list).
- Its content is assigned to another variable. This happens if the variable is given as argument to `return`, `kcall`, `fcall`, or `krec`.

A variable is *active* if it is in scope but not yet consumed.

Given a program point in a scope `V` of variables, a set of variables `A` $\subseteq$ `V` is a set of *active variables* if for any execution reaching this point, and any variable `x` : `V` storing a memory fragment, `x` is active if and only if `x` $\in$ `A`.

In figure 9.6(a), we annotated the sets of active variables associated to each program point. In figure 9.4, we describe the linearity condition as a predicate, following the inductive structure of instructions. This predicate has signature `Γ, aK` $\vdash$ {`aPre`} `u` {`aPost`} where:

- `u` is an instruction in some environment `E` : `Env.s` (of variables `E.var`, continuations `E.knt` and functions `E.fct`).
- `Γ` is a typing environment mapping the variables of `E.var` to their type.
- `aK` maps each continuation in scope to the set of active variables at the beginning of the continuation. Hence, given a continuation name `k` : `E.knt a`, the set `aK k` is a subset of the variables in scope at the declaration point of `k`. This set does not include the formal arguments of `k`, since they are always initially active.

$$\frac{\texttt{aPre} \equiv_\Gamma \texttt{arg} + \texttt{aPost}}{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{return (arg)}\ \{\texttt{aPost}\}} \qquad \frac{\texttt{aPre} \equiv_\Gamma \texttt{arg} + \texttt{aPost}}{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{fcall f(arg)}\ \{\texttt{aPost}\}}$$

$$\frac{\texttt{aPre} \equiv_\Gamma \texttt{op_csm op argV} + \texttt{op_ctx op argV} + \texttt{aFrame} \qquad \texttt{aPost} \equiv_\Gamma \texttt{op_ctx op argV} + \texttt{aFrame}}{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{op<argT>(argV)}\ \{\texttt{aPost}\}}$$

$$\frac{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{lhs}\ \{\texttt{aItm}\} \qquad (\Gamma, (\texttt{xs : ts})), \texttt{aK} \vdash \{\texttt{aItm} + \texttt{xs}\}\ \texttt{rhs}\ \{\texttt{aPost}\}}{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{let xs : ts = lhs; rhs}\ \{\texttt{aPost}\}}$$

$$\frac{(\Gamma, (\texttt{H : c = true})), \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{thn}\ \{\texttt{aPost}\} \qquad (\Gamma, (\texttt{H : c = false})), \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{els}\ \{\texttt{aPost}\}}{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{if c H then thn else els}\ \{\texttt{aPost}\}}$$

$$\frac{\texttt{aPre} \equiv_\Gamma \texttt{arg} + \texttt{aK k}}{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{kcall k(arg)}\ \{\texttt{aPost}\}}$$

$$\frac{\texttt{aPre} \equiv_\Gamma \texttt{arg} + \texttt{ak k}_0 \qquad \forall \texttt{i}.\ (\Gamma, (\texttt{xs}_\texttt{i} : \texttt{ts}_\texttt{i})), [\texttt{aK}, \texttt{ak}] \vdash \{\texttt{ak k}_\texttt{i} + \texttt{xs}\}\ \texttt{u}_\texttt{i}\ \{\texttt{aPost}\}}{\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{krec}\{\texttt{k}_\texttt{i}\texttt{(xs}_\texttt{i}\texttt{ : ts}_\texttt{i}\texttt{): u}_\texttt{i}\texttt{...}\}\texttt{(arg)}\ \{\texttt{aPost}\}}$$

Figure 9.4: Linearity criterion checked by the translation pass

- $\texttt{aPre} \subseteq \texttt{E.var}$ is the set of active variables at the beginning of u.
- $\texttt{aPost} \subseteq \texttt{E.var}$ is the set of active variables after a normal return of u. This set does not include the formal variables bound to the results of u, which are always initially active.

A proof of this predicate associates a set of active variables to each program point within the instruction. Those sets contain each variable at most once, but in figure 9.4, the union + is interpreted as a multi-set operation. It sums the number of occurrences of each variable. This prevents fragments from being consumed twice by the same instruction. For instance, the rule for fcall does not allow passing twice as argument (within arg) the same variable containing a memory fragment. We compare the multi-sets of variables using an equivalence relation $\equiv_\Gamma$, which depends on the types Γ of the variables. This relation checks that each multi-set contains the same number of occurrences of each variable, but only for the variables that can contain memory fragments according to Γ. For instance, if $\Gamma\ \texttt{p} = \texttt{ptr}$ and $\Gamma\ \texttt{c} = (\texttt{cell\{i64\}\ p})$, then $\{\texttt{c}\} \equiv_\Gamma \{\texttt{c}, \texttt{p}\} \equiv_\Gamma \{\texttt{c}, \texttt{p}, \texttt{p}\}$ but $\varnothing$, $\{\texttt{c}\}$ and $\{\texttt{c}, \texttt{c}\}$ are mutually not equivalent. Thanks to this equivalence relation, the linearity criterion does not constrain the use of variables that do not store memory fragments, such as scalars and propositions.

The rules for return, kcall and fcall reflect that the memory fragments given as arguments are consumed, and remove them from the set of active variables. In fact, those operations do not modify the memory, and those consumptions only reflect that their arguments are bound to new variables. For instance, return consumes its arguments because its results are initially active, and can appear later in the criterion if they are let-bound. The linearity criterion is intra-procedural: all arguments are consumed by fcall. Because of the control flow junction at the end of a conditional, both branches must have the same set aPost of active variables after a normal return. Our linearity criterion is not affine, that is, memory fragments cannot be discarded without being consumed. At a junction point, it is thus not possible to remove elements from the incoming active sets to reduce them to their intersection. In the rule for let, aItm is the set of active variables after a normal return of the left-hand side, not including the bound variables xs. Another consequence of the fact that our criterion is not affine is that memory fragments must be consumed before the program reaches the end of their syntactic scope. In figure 9.4, this is enforced by the implicit upper bounds on the sets of active variables. For instance, the set aPost after a let cannot contain the bound variables xs, so they must be consumed by the right-hand side. In the rule for krec, the function ak maps each declared continuation

to the sets of active variables at its beginning, not including its formal arguments. This rule involves both the consumption of the arguments of `krec` (given to the first continuation), junction points (at the beginning of the continuations and after a normal return) and variables with a local scope (the arguments of the continuations). The effects of operations depend on the operator executed. The rule of figure 9.4 involves two multi-sets of variables `op_csm op argV` and `op_ctx op argV`. For the main operators, those sets are given in figure 9.5, and we will motivate their definitions in each case when explaining the translation of the operations in section 9.3.2. Those sets are always subsets of the arguments `argV` of the operation. The arguments `op_csm` are those that are consumed by the operation (for instance, the cell given to `Frag.write`). They must be active before the operation (that is, in `aPre`) and are removed in `aPost`. The arguments `op_ctx` must be active, but are not removed in `aPost`. An example of such an argument is the cell given to `Frag.read`.

The linearity condition for a whole program is defined intra-procedurally:

Definition 9.3 A function `fn f(xs`$_0$ `: ts`$_0$`) : (xs`$_1$ `: ts`$_1$`) {u}` uses memory fragments linearly if $\texttt{xs}_0, \texttt{ts}_0, \varnothing \vdash \{\texttt{xs}_0\}\ \texttt{u}\ \{\varnothing\}$.
A program uses memory fragments linearly if all its functions do.

The sets of active variables are not annotated in Frag IR. The translation pass thus has to infer the sets `aItm` and `ak` in `let` and `krec`. This inference is simplified by the fact that we do not allow affine usages of memory fragments. Indeed, there is at most one possible active set for a given program point (modulo $\equiv_\Gamma$) and this set is found by following an arbitrary syntactic control path from the entry of the function to the point considered. The translation pass thus traverses instructions in a depth-first fashion with respect to their inductive structure. We only process a sub-instruction once the set of active variables at its beginning has been inferred. While traversing an instruction, we find active sets for the exit points of the instruction (normal return and continuation calls). If an active set was already known for such a point, we check its equivalence with the new one. Otherwise, we register the inferred active set and will later use it to process the instruction starting after the exit point (that is, the right-hand side of `let` or a continuation declared in `krec`). Currently, the pass raises an error if we cannot find an active set for some program points. This happens if an instruction is syntactically unreachable. Running the block-sinking pass (section 7.3) ensures that there is no such unreachable code. The translation accepts non-terminating functions with unreachable exit.

9.3.2 Transformation of the program

The translation recursively traverses the instructions and transforms locally the operations, by replacing them with Flat IR instructions which return exactly the same variables. Figure 9.5 gives the translation of a representative set of operations, and figure 9.6(b) an example of output of the pass. The program generated by the pass has thus the same structure as the source and the same scopes of variables, continuations and functions (except within the translation of operations, where the target contains additional local variables).

We prove both the preservation of the typing and of the semantics. For the latter, we have to compare the source semantics in `FragIR.S` with the target semantics in `FlatIR.S`. To do so, we unfold the definition of `FlatIR.S` and, using the conversion operation `to_Frag`, view the target semantics as a semantics in `FragIR.S`. This gives us access to the global memory of the Flat IR program. Although our transformation is performed locally, its proof involves a non-trivial invariant between the active memory fragments in the source and the global memory state of the target. We first define a relation between `heaplet` and `Mem.t`:

Source Frag IR operation	op_ctx	op_csm	Target Flat IR instruction
Scalar operation `Osc op(args…)` or `Osc_pre op(args…)`	∅	∅	Scalar operation without pre-condition `Osc op(args…)`
`Opt.some<ty>(x)`	∅	{x}	`Opt.some<ty>(x)`

`Frag.read{ty}(p, c)`	{c}	∅	`let v = Flat.read(p);`
			`let ghost H = assert<*c = v>();`
			`return (v, H)`

`ghost Frag.read{ty}(p, c)`	∅	∅	`ghost Frag.read{ty}(p, c)`
`Frag.write{ty}(p, c, v)`	∅	{c}	`Flat.write{ty}(p, v);` `Frag.write{ty}(p, c, v)`

`Frag.malloc(sz)`	∅	∅	`let p = Flat.malloc(sz);`
			`let ghost c = Flat.getalloc(p, sz);`
			`return (p, c)`

`Frag.free(p, sz, m)`	∅	{m}	`Frag.free(p, sz, m);` `Flat.free(p)`

`lemma {`
`  <xsT>(xsV : targ) : (xsR : tres)`
`  consume (csm)`
`}<argT>(argV)`
	∅	csm[argV/xsV]	`lemma {`
			`  <xsT>(xsV : targ) : (xsR : tres)`
			`  consume (csm)`
			`}<argT>(argV)`

`Frag.joinable(args…)`	args	∅	`assert<joinable args…>()`

Figure 9.5: Translation of operations and multi-sets op_csm and op_ctx

Definition 9.4 The relation `mem_match h (mdata, mperm)` holds between a heaplet `h : heaplet` and a global memory `(mdata, mperm) : Mem.t` if `h` has shape `(hdata, hperm)` (and is thus not ⊥) and:

- the addresses in the domain of `hdata` are exactly those with permission `Writable` in `mdata`
- `hdata` and `mdata` contain the same bytes at those addresses
- `mperm = mdata`

We then specify the translation of instructions using a refinement in `FragIR.S`. Our invariants are:

- the well-typing of values
- the equality of the values of corresponding source and target variables
- the disjointness of the footprints of active variables, and the fact that they are implemented in the current global memory (formally, a relation `mem_match`)
- objects that were allocated in the memory at the beginning of the instruction but not covered by the footprints of the active variables are unchanged (formally, we use a "frame" heaplet)

In more detail, our specification of the translation of the instruction is:

Lemma 9.1 Assume that a source instruction Frag IR u_0 is successfully translated into a Flat IR instruction u_1. Assume that u_0 is well-formed in an environment `E : Env.s`. Then u_1 is also well-formed in the same environment. Let `Γ : Env.Ty.s E` be the common typing environment of u_0 and u_1. Assume that u_0 is well-typed in `Γ`. Then u_1 also is, and for the same return type. Moreover, let `aK`, `aPre` and `aPost` be the sets of active variables inferred by the translation at the boundaries of u_0. Since the translation succeeded, u_0 uses memory fragments linearly, that is: `Γ.var, aK ⊢ {aPre} u₀ {aPost}`. Let:

- ρ : E.var → val be a well-typed valuation of the variables, common to u_0 and u_1
- mPre : Mem.t be the global memory at the beginning of u_1
- frame : heaplet an arbitrary heaplet, representing parts of mPre not covered by the active memory fragments in the scope of u_0

If:

$$\texttt{mem_match}\ (\texttt{footprint}\ \rho(\texttt{aPre}) \uplus \texttt{frame})\ \texttt{mPre}$$

then:

$$[\![u_0]\!]_{\text{instr}}\ \rho \gtrsim_{(\texttt{Re_ty}\ \Gamma;\texttt{Tr.fcall_match}),(\texttt{Tr.res_match}\ \texttt{aK}\ \texttt{aPost}\ \rho\ \texttt{frame})}\ \texttt{to_Frag}\ \left([\![u_1]\!]_{\text{instr}}\ \rho\right)\ \texttt{mPre}$$

where Tr.res_match, Tr.fcall_match and Tr.fcall_post are generated by:

$$\frac{\texttt{mem_match}\ (\texttt{footprint}\ \rho(\texttt{aPost}) \uplus \texttt{footprint}\ \texttt{vs} \uplus \texttt{frame})\ \texttt{mPost}}{\texttt{Tr.res_match}\ \texttt{aK}\ \texttt{aPost}\ \rho\ \texttt{frame}\ (\texttt{Res.Ret}\ \texttt{vs})\ (\texttt{Res.Ret}\ \texttt{vs}, \texttt{mPost})}$$

$$\frac{\texttt{mem_match}\ (\texttt{footprint}\ \rho(\texttt{aK}\ \texttt{k}) \uplus \texttt{footprint}\ \texttt{vs} \uplus \texttt{frame})\ \texttt{mPost}}{\texttt{Tr.res_match}\ \texttt{aK}\ \texttt{aPost}\ \rho\ \texttt{frame}\ (\texttt{Res.Knt}\ \texttt{k}\ \texttt{vs})\ (\texttt{Res.Knt}\ \texttt{k}\ \texttt{vs}, \texttt{mPost})}$$

$$\frac{\texttt{mem_match}\ (\texttt{footprint}\ \texttt{arg} \uplus \texttt{frame}')\ m_0}{\texttt{Tr.fcall_match}\ (\texttt{Fct.Call.mk}\ \texttt{f}\ \texttt{arg})\ (\texttt{State.mk_ev}\ (\texttt{Fct.Call.mk}\ \texttt{f}\ \texttt{arg})\ m_0)\ (\texttt{Tr.fcall_post}\ \texttt{frame}')}$$

$$\frac{\texttt{mem_match}\ (\texttt{footprint}\ \texttt{res} \uplus \texttt{frame}')\ m_1}{\texttt{Tr.fcall_post}\ \texttt{frame}'\ \texttt{res}\ (\texttt{res}, m_1)}$$

Idea of the proof By induction on the syntax of u_0 and relational reasoning in FragIR.S. We maintain the invariant that the values are well-typed, notably thanks to Re_ty Γ in the relation on events. The most interesting cases are those of the operations and of the function calls. The former depends on the operator evaluated and is detailed below. For the latter, the relation on events Tr.fcall_match ensures that the footprints of the arguments are replaced by the footprints of the results. However, we also have to preserve the footprints of the active variables that are not given as argument to the callee. Those variables are the active set aPost in $\Gamma, \texttt{aK} \vdash \{\texttt{aPre}\}\ \texttt{fcall f(arg)}\ \{\texttt{aPost}\}$. Using the decomposition of aPre given by the linearity check (figure 9.4), we derive from the invariant before the call that mem_match (footprint ρ(arg) ⊎ footprint ρ(aPost) ⊎ frame) mPre. After the call, the post-condition Tr.res_match reduces to mem_match (footprint ρ(aPost) ⊎ footprint res ⊎ frame) mPre. We thus prove that the call preserves our invariant by adding the variables aPost in the frame of the callee, that is, by choosing frame′ := footprint ρ(aPost) ⊎ frame in Tr.fcall_match.

To simplify the specification of the translation of the whole program, the pass optionally checks that there are no memory fragments within the arguments of the main function. When this is the case, we can drop the pre-condition mem_match on the initial memory and prove:

Lemma 9.2 Assume that the translation pass successfully generates a Flat IR program $\texttt{pr}_1$ from a well-formed and well-typed Frag IR program $\texttt{pr}_0$. Then $\texttt{pr}_1$ is well-formed and well-typed and its main function has the same signature as the main function of $\texttt{pr}_0$. Assume that the pass has checked that the main function does not have any memory fragment as argument. Then, for any well-typed common arguments varg and arbitrary initial global memory m_0:

$$[\![\texttt{pr}_0]\!]_{\text{prog}}\ \texttt{varg} \gtrsim_{\bot, \lambda \texttt{res}_0\ (\texttt{res}_1, m_1).\ \texttt{res}_0 = \texttt{res}_1}\ \texttt{to_Frag}\ \left([\![\texttt{pr}_1]\!]_{\text{prog}}\ \texttt{varg}\right)\ m_0$$

Idea of the proof We combine the refinements of corresponding function bodies (lemma 9.1) using ref_mrec in FragIR.S. To discharge the pre-condition mem_match of the initial call to the main function,

we build a heaplet `frame` containing all objects allocated in the initial memory. That is, satisfying `mem_match frame` m_0.

We now explain the translation of each operator (figure 9.5):

scalar operators In the target, we always emit the variant without precondition to avoid using proposition variables. Those operations do not affect the linearity condition since scalar values have empty footprints. External events `Otrig` are handled similarly.

optional values The pass supports operations on optional values (e.g., `Opt.some`) and leaves them unchanged. In practice, we only use `opt` for scalar values, but our syntax allows its use for arbitrary types, including memory fragments. In this case, the operations act as assignments in the linearity criterion and thus consume their arguments.

`Frag.read` We replace the `Frag.read` by an access `Flat.read` to the global memory. We add an assertion to recover the post-condition ensuring the equality between the value read and the content of the cell (line 4 in figure 9.6(b)). During the refinement, we prove this equality using the fact that the cell is active and the invariant `mem_match`. The dependent type `cell{ty} p` ensures that the cell is stored at the address `p` accessed by `Flat.read` in the target (contrary to `Frag.read`, whose semantics given in figure 8.9 ignores its pointer argument). The cell remains active after the read, so we use `op_ctx` instead of `op_csm`. Operations `Frag.read` can be annotated as `ghost` when they are used to obtain the content of the cell as a ghost variable. In this case, we keep the operation unchanged and do not require the cell to be active.

`Frag.write` We insert a `Flat.write` next to `Frag.write` (line 7 in figure 9.6(b)), but keep the latter to generate the new cell. Since `Flat.write` updates the global memory in a destructive fashion, the old cell is consumed.

`Frag.malloc` Memory allocations are performed in the target using `Flat.malloc`. We use `Flat.getalloc` to recover the corresponding memory fragment. During the refinement, we first execute the target non-deterministic choice of `Flat.malloc`, then simulate this choice in the source by choosing a corresponding result (pointer and memory fragment) for `Frag.malloc` in the source.

`Frag.free` We add `Flat.free` next to `Frag.free`. Semantically `Frag.free` is a no-op, so we could also directly remove it. Since the footprint of the fragment given as argument is freed in the global memory, it is consumed by the operation.

`lemma` Lemma operations are kept unchanged. When they transform memory fragments, they consume the arguments corresponding to their `consume` clause. We justify the preservation of the invariant `Heaplet.mem_match` thanks to the equality `EqFootprint` of the footprints in the post-condition of lemmas (definition 8.1). Operations `Strc.make` and `Strc.elim` on structures are handled similarly, and consume all their memory fragment arguments.

`Frag.joinable` In the source, `Frag.joinable` is an assumption given to the program. This assumption is given by the compiler, and proven during the translation. Syntactically, the operation is replaced by an `assert` (line 10 in figure 9.6(b)), which has UB if the fragments are not disjoint. During the refinement, we thus have to prove this disjointness. We use the invariant `mem_match` and require all fragments given as arguments to `Frag.joinable` to be active (we interpret `op_ctx` as a multi-set, so the linearity condition ensures that a fragment can only appear once within the arguments of a given `Frag.joinable`). Indeed, `mem_match` implies the validity of its memory fragment argument and thus the disjointness of all active variables. Since the global memory is unchanged, the arguments are not consumed.

9.4 Erasure and simplification

After the translation pass (section 9.3), the Flat IR program still contains propositions, memory fragments and operations on them (an example is given in figure 9.6(b)). However, the part of the program

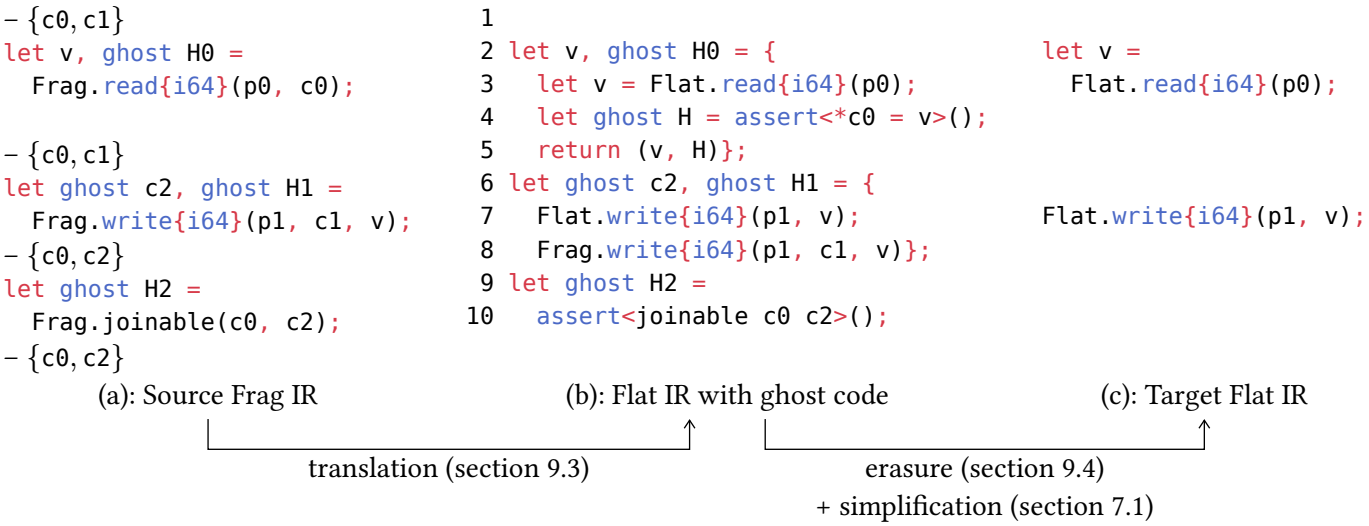


Figure 9.6: Compilation of a program from Frag IR to Flat IR, and sets of active variables inferred in the source [compiler/test/src/pipeline/tr]

that computes the observable actions (external events and scalar results) no longer depends on them. We thus remove those variables and operations using an instance of the parameterized erasure pass (section 7.2.3). After this pass, the program no longer contains any memory fragments. In order to keep the same syntax of operations and conditionals, the program contains binders for post-conditions of operations and branch hypotheses, but does not use those variables.

To instantiate the pass, we implement the functions `is_var_erased` and `is_op_erased`. We use the annotations `ghost` on the variables and operations. We also always require the removal of operations that are always ghosts at the level of Flat IR, regardless of their annotations. This includes the operations that were already always ghost in Frag IR (e.g., `lemma` or `assert`) but also `Frag.write` and `Flat.getalloc` since they are only used to obtain memory fragments after the translation. The parameterized erasure pass checks defensively that the choice of erased code is consistent. To prove the correctness of our instance, we only have to prove that our implementation of `is_op_erased` never requires the removal of operations that are not semantically removable (according to definition 7.3). Like for the DCE on Flat IR (section 7.2), our implementation raises an error if an operator is annotated as `ghost` but does not have a removable semantics.

For simplicity, both the translation and the erasure passes preserve the structure of the instructions. They do not attempt to simplify sequences of instructions (hence the left-nested `let`s in figure 9.6(b), delimited by `{…}`) and ghost operations are replaced by `return ()`. To improve the readability of the generated program and to reduce its size, we run the generic simplification described in section 7.1. Figure 9.6(c) reproduces an example of a final Flat IR program.

9.5 Verified interpreter

We implement an interpreter for a subset of Flat IR, which enables executing the programs output by our pipeline (either interactively, or against a predefined sequence of inputs and outputs). The interpreter only supports the subset of the operators of Flat IR corresponding to standard low-level operations: scalar operations (`Osc`), external events (`Otrig`), operations on optional values (`Opt.none`, `Opt.some` and `Opt.get`, since we use them to represent uninitialized variables), and operations on the global memory (`Flat.read`, `Flat.write`, `Flat.malloc`, `Flat.free`). In particular, we exclude operations on memory fragments, since `lemma`s cannot be executed in the general case (as explained in section 3.2.7). The interpreter starts by checking that the program given as argument syntactically only contains supported operators. This avoids raising an error later not corresponding to a UB of the

semantics of the program. Although we do not prove it formally, the programs generated by successful runs of our pipeline should fall in the supported subset.

Given a program in the supported subset, our interpreter builds an executable transition system using the library described in section 5.5.3. The system has a structure similar to the semantics of Flat IR (and thus to the semantics of our parameterized representation), with the following noteworthy differences:

- The set of values of the interpreter is restricted to the scalars and optional values.
- The interpreter uses the operators of `SMonad` on `Interp.t` instead of those on `FragIR.S` (derived from those on `LTS.t`).
- For efficiency, the Rocq functions used for the valuation of the variables (`ρ : E.var → val`) and the memory (`Mem.t`) are implemented with binary trees.
- We provide a toy implementation for `Flat.malloc`. This implementation maintains the next address not yet allocated as an additional field within the memory state of the interpreter. We allocate memory by increasing address and never reuse freed addresses. If we reach the upper bound of our address space (2^{64}), we abort the execution and signal an out-of-memory error using a τ-event `OutOfMemory` with empty answer type. An alternative would be to return the null pointer.

FEEDBACK SB260906

Ce serait plus dans l'esprit de la spec ?

Oui, mais on parle d'un cas qui n'arrivera de toute façon pas en pratique (à moins de demander une très grosse plage mémoire). Idéalement, l'interpréteur pourrait être paramétré pour demander à l'utilisateur à chaque malloc ce qu'on veut renvoyer au programme. Je verrai si j'ai le temps d'améliorer ça.

We specify the correctness of our interpreter as a refinement in `LTS.t`, using the conversion `Interp.to_lts` and the semantics $[\![.]\!]^{\text{Flat}}_{\text{prog}}$ of Flat IR programs in `LTS.t`. This refinement implies that the observable behaviors of the interpreter are a subset of those allowed by the semantics of Flat IR. In general, this subset is strict since we use a particular implementation of `Flat.malloc`.

Lemma 9.3 If our interpreter successfully generates an executable transition system `int` for a program `pr` and arguments `vargs` (i.e. a list of values in the subset of the interpreter), then:

$$[\![\texttt{pr}]\!]^{\text{Flat}}_{\text{prog}}\ \texttt{vargs} \gtrsim \texttt{Interp.to_lts int}$$

Idea of the proof We prove the refinement by induction on the syntax of the program, relational reasoning in `LTS.t` and using compatibilities between `Interp.to_lts` and the operators on `Interp.t` given by lemma 5.3. In the case where the interpreter aborts with an error `OutOfMemory`, its semantics given by `Interp.to_lts` is `any False`, since `OutOfMemory` is a τ-event with empty answer type. At this point the semantics of the interpreter thus refines any semantics.

9.6 Correctness of the pipeline

We compose the correctness properties of the passes to prove the correctness of the pipeline from Frag IR to Flat IR (figure 9.1). Its statement ensures both the preservation of the well-formedness (which includes the correctness of the source lemma instructions) and of the semantics. In this section, we make explicit the use of two syntaxes in our formalization (AS and DTS, as described in section 6.3). We prove that the semantics of the target refines the semantics of the source in the domain `LTS.t`, that is, that the source simulates the target.

Lemma 9.4 Let `t_src` be the AS of a source Frag IR program. Assume that `t_src` implements a DTS program `s_src` (and is thus well-formed), for a chosen interpretation of its abstract predicates. If our pipeline `frag_to_flat` successfully generates the AS `t_trg` of a target Flat IR program, then there exists a DTS program `s_trg` implemented by `t_trg` (so `t_trg` is well-formed). Moreover, the semantics of the target refines the semantics of the source, for any well-typed arguments `vargs` of the main function:

$$[\![\texttt{s_src}]\!]^{\text{Frag}}_{\text{prog}}\ \texttt{vargs} \quad \gtrsim \quad [\![\texttt{s_trg}]\!]^{\text{Flat}}_{\text{prog}}\ \texttt{vargs}$$

Proof By composing the specifications of the successive passes, at the level of programs. That is:
- the block-sinking (lemma 7.3 and section 7.3.3.5)
- `OptDefault` (section 9.2)
- the type-checker instantiated on Frag IR (lemma 7.5 and section 8.3.1)
- the DCE instantiated on Frag IR (section 8.3.2)
- the translation from Frag IR to Flat IR (lemma 9.2)
- the erasure on Flat IR (section 9.4)
- the simplification pass instantiated on Flat IR (section 7.1)

During the whole pipeline, we maintain as invariant the existence of a DTS implemented by the current AS. We also expect all programs involved to be well-typed. However, we did not prove that all passes preserve the typing (nor assume that the source is well-typed), so this invariant is only propagated formally between the use of the type-checker (before the DCE on Frag IR) and the Flat IR program generated by the erasure.
Each transformation ensures a semantic refinement between the semantics of the DTS of its source and target. Refinement in `FragIR.S` and `FlatIR.S` implies refinements in `LTS.t`. We compose those refinements by transitivity (`sem_ref_trans`). □

At a high level, this correctness ensures the following properties. Given a source program and an interpretation of its abstract predicates, if the source program:
- is syntactically well-formed (e.g., does not contain free variables)
- only contains correct lemmas
- does not have UB

Then all behaviors of the target are observable on the source. Hence, the target:
- does not have errors (e.g., invalid memory accesses)
- is functionally correct if the source is
- does not diverge, unless the source does

By composing the correctness of the pipeline and of the interpreter on Flat IR, we obtain a verified interpreter for Frag IR.

Lemma 9.5 Let `t_pr` be the AS of a Frag IR program implementing a DTS `s_pr`. If the composition of our pipeline `frag_to_flat` and interpreter successfully generates an executable transition system `int` from `t_pr` and some well-typed arguments `vargs` in the subset of the interpreter, then:

$$[\![\texttt{s_pr}]\!]^{\text{Frag}}_{\text{prog}}\ \texttt{vargs} \quad \gtrsim \quad \texttt{Interp.to_lts int}$$

Proof Composition of lemma 9.4 and lemma 9.3 using `sem_ref_trans`. □

10

Using Vélus as a front-end

In chapter 9, we defined a compilation pipeline taking Frag IR programs as inputs. However, Frag IR is an intermediate representation, not a source language. Although we experiment by writing Frag IR programs by hand (section 8.4), the representation is instead meant to be used to translate higher-level languages. In this chapter, we experiment with the use of Frag IR as a back-end for an existing language. We use the verified compiler Vélus as a front-end, and implement a verified translation pass from its representation Obc to our Frag IR. This translation tests the ability of Frag IR to encode a higher-level representation, and also the connection between our `SMonad`-based semantics and existing semantics. By combining this translation with our pipeline and the existing infrastructure of Vélus, we obtain a verified pipeline from the source language Lustre of Vélus to Flat IR. Its specification has assumptions similar to the standard compilation of Vélus to assembly, but its conclusion differs because our languages involve internal non-determinism.

Section 10.1 gives background on Vélus, the representation Obc and the standard pipeline to assembly. Section 10.2 then explains our motivations for choosing Obc as a source language. Section 10.3 describes the structure of the translation to Frag IR. Section 10.4 details its top-level specification, and explains our strategy for proving it.

10.1 Vélus and its Clight back-end

Vélus [21] is a compiler of a synchronous language. It is formally verified in Rocq. Its standard version uses CompCert as back-end to compile the programs down to assembly. This section describes some elements of Vélus that are relevant for our alternative back-end to Frag IR. Brun [23] gives a detailed description of the full pipeline of Vélus and Pesin [83] describes more recent features. Section 10.1.1 presents the source language "Lustre" and section 10.1.2 "Obc", the last intermediate representation of Vélus, that we also use as source for our translation to Frag IR. Section 10.1.3 mentions some aspects of the standard translation of Obc to Cminor to provide points of comparison with our alternative translation.

```
node acc(x : int) returns (y : int)
let
  y = (0 fby y) + x;
tel

node main(i : int)
  returns (c : bool; a : int)
  var b  : bool;
  var a0 : int;
  var a1 : int when b;
let
  b  = (i ≥ 0);
  c  = (b ≠ (true fby b));
  a0 = acc(i);
  a1 = acc(i when b);
  a  = merge b a1 (a0 when not b);
tel
```

a: Textual representation of the program

i	3	−1	8	5	7	−2	...
b	⊤	⊥	⊤	⊤	⊤	⊥	...
true fby b	⊤	⊤	⊥	⊤	⊤	⊤	...
c	⊥	⊤	⊤	⊥	⊥	⊤	...
a0	3	2	10	15	22	20	...
i when b	3		8	5	7		...
a1	3		11	16	23		...
a0 when not b		2				20	...
a	3	2	11	16	23	20	...

b: Chronogram of an execution

Figure 10.1: An artificial Lustre program [velus/tests/to_frag/src/two_acc.lus]

10.1.1 Lustre: a synchronous source language

Figure 10.1a reproduces a Lustre program, that we will use as running example. Such a program contains a set of node declarations. Each node depends on a list of input variables (e.g., `x` for the node `acc`) and defines one or several outputs (e.g., `y`). In synchronous languages like Lustre, variables and expressions represent streams of values. Figure 10.1b represents admissible streams for some expressions of the `main` node as a *chronogram*. In this table, each line defines (a prefix of) a stream. Each column contains the values of the streams at a given instant. The first line describes an input stream for `i` and the others the corresponding values computed by the program, including its outputs `c` and `a`.

Typing Each variable and expression has a type, and Vélus type-checks programs before compiling them. The set `Velus.type` of types is the union of a subset of the scalar types of CompCert (e.g., 32-bit integers `int`) and of enumeration types (`bool` being a particular, pre-defined enumeration). In Rocq, the corresponding set `Velus.value := Vscalar s | Venum n` of values is shared by all intermediate representations of Vélus.

Equations Contrary to imperative programs which define a sequence of operations producing their output, synchronous programs consist of equations relating input, output and local variables. Those equations contain standard operators, including the scalar operators of Clight. For instance, the equation `b = (i ≥ 0)` constrains values of the local Boolean variable `b` as a function of `i`. In an admissible execution of the program, this equation must be satisfied at each instant. For instance, in figure 10.1b, we can check that at the third instant `b` is true since `i` has the non-negative value 8. Once defined, a node may be used within equations in later nodes. For instance, we use `acc` twice in `main`. Node declarations are ordered, so there cannot be recursive equations.

Delays Lustre features delay operators which enable equations to depend on values at previous instants. In figure 10.1, we only use the `fby` ("followed by") operator. Given two expressions e_0 and e_1, the value of the expression e_0 `fby` e_1 is equal at the first instant to the value of e_0. At a later instant, it is equal to the value of e_1 at the previous instant. In figure 10.1b, the stream `true fby b` is thus obtained by shifting `b` by one instant to the right, and by adding `true` as first value. The `main` node uses this expression to signal in `c` changes of the value of `b`. We also use `fby` in the `acc` node, to compute the sum `y` of the arguments `x` seen since the beginning of the execution.

Absent values During execution of a program, the streams are synchronized on the same notion of discrete time, corresponding to the columns in figure 10.1b. However, streams are allowed to have no defined values for some instants. In figure 10.1b, those *absent values* are represented by empty cells. Absent values can be introduced by subsampling a stream using an operator `when`. For instance, `i when b` contains the same values as `i` when `b` is true. Because of this subsampling, the occurrence of `acc` in the equation for `a1` is only evaluated when `i` is non-negative. On the contrary, `a0` is always evaluated, and subsampled later within `a`. As a consequence, its instance of `acc` sees all values of `i`, including the non-negative ones. Yet another possibility, not used in figure 10.1, is to evaluate a node while some of its arguments are absent. The `merge` operator combines the values of two streams which are never defined simultaneously. Vélus features a clock system (e.g., the annotation `int when b` on the declaration of `a1`) and a corresponding check to track statically when variables contain absent values. In particular, the inputs and outputs of the `main` node do not contain absent values.

Semantics Well-clocked Lustre programs have deterministic semantics. That is, given some streams for the input variables, there is at most one set of stream for the output variables and all intermediate expressions. In Rocq, the semantics of a node `nd` in a global environment `G` (containing previous node declarations) is defined as a relation `sem_node G nd ins outs` between streams `ins` and `outs` of values for the inputs and outputs. The semantics of a program is the semantics of its `main` node.

10.1.2 The Obc intermediate representation

Obc (Object Code Language [23, §4.1]) is the last intermediate representation before Clight in the standard pipeline of Vélus. It is an imperative language but inherits the structure of the source synchronous programs. In particular, it has a tree-shaped global memory reflecting the node hierarchy.

Syntax Figure 10.2a reproduces a subset of the Obc program generated by Vélus from the example of figure 10.1a. An Obc program contains an ordered list of classes. Each Obc class implements[22] a Lustre node. Like object-oriented languages, a class defines a memory layout and methods which can be called on its instances. Syntactically, a class is defined by:

- Memory variables (`state`), which declare data stored in instances of the class and persist between method calls. Vélus uses those variables to implement the delay operators. In our `main`, the state variable `mb` implements the source `fby` operator.
- Instances (`instance`) of classes defined earlier. Each sub-instance implements an occurrence of the corresponding node in the equations. There are thus two instances of the class `acc` in `main`.
- Methods, which are called on instances of the node. The body of a method is defined by inductive expressions and statements, and has access to the memory variables (with the syntax `state(…)`) and sub-instances (through calls, like `acc(ma0).step(…)`). Statements are composed sequentially (`lhs; rhs`) and by case analysis on enumerations (`switch`). In the Obc programs generated by Vélus, each class has two methods:
  - `reset` has neither arguments nor results and initializes the memory of the instance it is called on.
  - `step` which has the same arguments and results as the source node. This method computes the outputs of the node at a given instant, and also updates the memory state of the node.

Typing Variable declarations are annotated with their types (whose set `Velus.type` is the same as in Lustre). A typing judgment is defined on Obc, and Vélus formally guarantees that it only generates well-typed programs.

Semantics The semantics of an Obc statement `a` within a program `prog` (a list of top-level declarations) is defined by an inductive big-step predicate `stmt_eval prog` $\mathtt{me}_0$ $\mathtt{ve}_0$ `a` $(\mathtt{me}_1, \mathtt{ve}_1)$. The arguments $\mathtt{me}_0$ and $\mathtt{me}_1$ are states `menv` of the memory (memory variables and sub-instances) and

[22] Our example is chosen so that there is a direct correspondence between the source Lustre program and its compilation to Obc. On other programs, Vélus may perform transformations which make the relation more complex.

```
class acc {
  state my : int;

  step(x : int) returns (y : int) { … }
  reset() { … }
}

class main {
  instance ma1 : acc;
  instance ma0 : acc;
  state mb : bool;

  step(i : int) returns (c : bool; a : int)
    var b : bool; a0 : int; a1 : int
  {
    a0 := acc(ma0).step([i]);
    b  := i ≥ 0;
    c  := b ≠ state(mb);
    state(mb) := b;
    switch b {
      | True  => a1 := acc(ma1).step([i]);
                 a  := a1
      | False => a := a0
    }
  }

  reset() { … }
}
```

a: Obc program

```
main
├─ ma1 : acc
│  └─ my : int
├─ ma0 : acc
│  └─ my : int
└─ mb : bool
```

b: Memory tree for the class `main`

Figure 10.2: Obc program generated from figure 10.1a [velus/tests/to_frag/src/two_acc_expected.obc]

ve_0 and ve_1 are environments `venv` storing the values of the temporary variables (inputs, outputs and local `vars` of the method). The predicate relates the states (me_0, ve_0) at the beginning of the instruction to the states (me_1, ve_1) at its end. The effects of Obc statements are accesses (reads and writes) to the memory and temporary variable states and UB (for instance, evaluation errors of partial operators). They can call methods of sub-instances, but `stmt_eval` directly calls inductively the semantics of the body of the callees, thanks to the argument `prog`. The semantics of a call to a method `cl.mt` is indeed defined mutually with the semantics of statements. It has signature `stmt_call_eval prog` me_0 `cl mt argv` $(me_1, resv)$, where me_0 (resp. me_1) is the memory state of the instance of `cl` before (resp. after) the call and `argv` (resp. `resv`) the list of arguments (resp. results). The fact that the semantics of Obc statements is completely captured by a big-step predicate is explained by their restricted effects. Indeed, they cannot diverge nor emit external events.

The semantics of an expression `e` is defined similarly by a big-step predicate `exp_eval me ve e v`. This predicate depends on the state of the memory (`me`) and of the temporary variables (`ve`) in which the expression is evaluated. It defines the value `v : option Velus.value` the expression evaluates to, where a result $\varnothing$ signals an undefined value. Contrary to statements, expressions cannot modify the states nor call methods of sub-instances.

The semantics `sem_prog pr ins outs` of an Obc program `pr` is defined similarly to `sem_node`, as a relation between streams `ins` and `outs` of input and output values of the main node. This semantics starts with an initial memory given by the `reset` method of `main`, then checks that the relation defined by the `step` method holds at each instant. In Obc, all semantics (expressions, statements and program) are deterministic.

Tree-shaped memories In the semantics, a state `menv` of the global memory is a tree whose shape matches the layout defined by the current class. Figure 10.2b depicts the shape of global memories used during the evaluation of instructions within the `main` class. A node of a memory tree is either the root of an instance of a class or a leaf storing the value of a memory variable. The children of an instance of a class match its `instance` and `state` declarations. The evaluation of a statement only accesses directly the children of the root of the memory. A method call to a sub-instance is evaluated on the corresponding subtree. As a consequence, the effect of a method call is confined to the subtree of the sub-instance. Moreover, there is no aliasing within the same Obc method. Despite the use of a global memory, the structure of Obc enables recovering precise dependency information between the accesses. For instance, in figure 10.2, the method call `acc(ma0).step` does not have access to the memory variable `mb`. Hence, the later accesses to `state(mb)` do not depend on it.

Undefined values The semantics of Obc allows temporary variables to have undefined values [22]. This feature is used to translate the absent values of Lustre. Similarly to absent values, undefined values can appear in the arguments and results of methods but cannot be given as an argument to an operator or as a guard to a `switch` statement. They also cannot be assigned to temporary or memory variables. The difference between undefined values and absent values is that Obc does not track statically when variables are defined. Instead, its semantics has UB if undefined values are used as arguments of operations or as the right-hand side of an assignment. In particular, the expressions of Obc contain validity assertions `[x]`, which have UB if the variable `x` has an undefined value and evaluate to its value otherwise.

10.1.3 The existing Clight back-end

In the standard pipeline of Vélus, a pass [23, chapter 5] translates Obc programs to the representation Clight of CompCert, which is then used to compile the program down to assembly. Figure 10.3 reproduces the declarations contained in the translation of our running example. The translation follows the syntax of the program. The memory layout of each class is declared as a `struct`. Each method is translated by a function taking a pointer to the memory of the instance as an additional argument `obc2c$self`. Obc types, expressions and statements are translated to their Clight equivalent. Vélus also emits a function `main` implementing an infinite loop which computes the outputs of the program. The memory used by this function (that is, the instance of the main node) is allocated as a global variable `glob$self$main`. The loop interacts with the external environment by performing volatile loads and stores.
The remainder of this section focuses on some aspects of this translation relevant for the comparison with our alternative back-end.

Memory Obc has a structured, tree-shaped, memory while Clight uses the memory model of CompCert. In the semantics of the latter, `struct`s are represented by a range of bytes within a memory block. This difference between the two memory models has two consequences:

- The proof of the translation of Obc uses a separation logic to justify the encoding of trees into flat sequences of bytes.
- Some properties guaranteed by the semantics of Obc are harder to establish at the level of Clight. For instance, although `fun$step$main` only calls `fun$step$acc` with a pointer to the field `a0` or `a1`, the signature of `fun$step$acc` does not constrain the parts of the memory accessed. In particular, the function could access the field `mb` through the global variable `glob$self$main`. To check that it is not the case, one needs to look at the implementation of `fun$step$acc`, hence to use an inter-procedural analysis.

Undefined values The undefined values of Obc are translated by the poison value `Vundef` of CompCert. However, Obc allows undefined values in the arguments and results of methods, while Clight (following the C standard) does not allow `Vundef` within function arguments [22, §3]. Vélus

```
struct acc  { int my; };
struct main { unsigned char mb; struct acc a1; struct acc a0; };
struct fun$step$main { unsigned char c; int a; };

int fun$step$acc(struct acc *obc2c$self, int x);
void fun$reset$acc(struct acc *obc2c$self);
void fun$step$main(struct main *obc2c$self, struct fun$step$main *obc2c$out, int i);
void fun$reset$main(struct main *obc2c$self);

struct main glob$self$main;
int main(void);
```

Figure 10.3: Standard Clight translation of figure 10.2a. The type `unsigned char` translates the enumeration `bool`. [velus/tests/to_frag/src/two_acc_expected.light.c]

handles this difference by transforming the Obc program before the translation [23, §5.1]. This transformation `ObcAddDefault` initializes some variables to ensure that undefined values are never passed as argument. The translation pass to Clight alone then only has to be sound for a syntactic subset of Obc, where validity assertions have been added on variables given directly as arguments to method calls.

Multiple results Clight does not support functions with multiple results. When translating Obc methods with multiple results (arising from Lustre nodes with multiple outputs), Vélus introduces an auxiliary structure. This structure is allocated in memory by the caller and the callee writes its results inside it. In figure 10.3, this is the case for the translation of the method `step` of the class `main`, which involves a structure `fun$step$main`.

Correctness The passes of Vélus and CompCert are proved sound with respect to the semantics of the source, target and intermediate languages. By composing those results, Vélus provides an end-to-end guarantee `behavior_asm`, relating the semantics of the generated assembly program to the Lustre source [23, theorem 6.1.1]. This theorem is given in a forward style. That is, it ensures that, if the source admits a given trace of inputs and outputs (according to `sem_node`), then the target has a corresponding behavior (as a sequence of volatile read and writes, which are observable in the behaviors of CompCert). Ultimately, the desired property is instead an upper bound on the set of behaviors of the target. Such a property is implied by the theorem in forward style thanks to the determinism of the target assembly language.

10.2 Motivations

Our first motivation for implementing the translation from Obc to Frag IR was to obtain a front-end for our compilation pipeline. More precisely, this translation provides a source of Frag IR programs to test our pipeline. Indeed, combined with the beginning of the standard pipeline of Vélus (up to Obc), this translation enables generating Frag IR programs from Lustre programs. Contrary to Frag IR, Lustre is designed as a source language where we easily write new programs by hand. Moreover, we also use existing programs as sources.

Our second motivation was to test the ability of Frag IR to encode information provided by the structure of higher-level languages. The tree-shaped memory model of Obc is rather simple and enforces separation properties between the memory of instances. One of the goals of our translation pass is thus to encode this model using memory fragments, while keeping information about the dependencies between memory accesses.

Our third motivation was to test the suitability of Frag IR as a back-end for a formally verified pipeline. As explained in section 10.1.3, the standard translation to Clight involves several complexities, which affect both the implementation of the pass and its proof. On some aspects, the translation to Frag IR and its proof are thus simpler than the translation to Clight:

- The memory fragments of Frag IR provide a structured semantics of the memory, where operations on a fragment do not affect the others. The memory model of Obc is closer to this model than to the flat memory of Clight. The implementation of the encoding to Frag IR needs to manage memory fragments, but its proof does not have to use a separation logic. Ultimately, our pipeline still produces Flat IR programs with a flat global memory thanks to the translation pass from Frag IR to Flat IR (section 9.3). One of the benefits of this later switch to a flat memory is that our translation pass to Flat IR is not specific to Obc. Moreover, its defensive linearity check avoids a formal proof of the linear usage of memory fragments during the translation of Obc.
- Frag IR represents variables that can be uninitialized using optional values with types `opt`. Contrary to the `Vundef` value of CompCert, there are no restrictions on the use of optional values in the arguments of functions in Frag IR. We thus do not need to initialize the arguments of method calls, and remove the pass `ObcAddDefault`. Currently, we can keep the option values within our pipeline to Flat IR. To encode our programs to an existing low-level representation, there would be two possibilities:
  - If the representation has an equivalent of undefined or poison values and does not prohibit their use within arguments of functions, then we do not have to add initializations. For instance, this would be the case for LLVM IR [71], where poison values are allowed in function calls except when a `noundef` attribute is specified.
  - Otherwise, one would need to initialize the arguments of functions, either using the `ObcAddDefault` pass of Vélus or our pass `OptDefault`. The latter is more generic since it is defined in Frag IR, but has the drawback of initializing all `opt` variables. In order to rely on `ObcAddDefault`, one would need to propagate the information that arguments are initialized. Currently, our translation ignores the validity assertions of Obc, but we could emit corresponding `assert`s in Frag IR. Clight is an example of target language which would require adding initializations. It is now also the case for the IRs of CompCert up to RTL, since a recent change modified their semantics to propagate typing information available in Clight (commit `CompCert:0eabf20d`).
- Like Obc, Frag IR allows functions returning multiple variables. We thus do not have to introduce auxiliary structures to compile such functions. In a full pipeline to a low-level language such as assembly, this encoding would still have to be done somewhere to respect standard calling conventions. However, we argue that this encoding is not specific to Obc and could be done on a more generic representation like Frag IR. We leave its implementation as a compilation pass for future work.

10.3 Translation of Obc to Frag IR

We expect the translation from Obc to Frag IR to:

- Succeed on almost any Obc program produced by Vélus. The only exceptions are external calls to pure functions. In the standard back-end, those functions are implemented in C. As future work, we could support calls to pure functions written in Frag IR (or even Flat IR), but this would require at least a modification of the interface and specification of the translation from Obc.
- Produce well-formed programs, in particular containing only sound lemmas.
- Produce well-typed programs.
- Produce programs which use memory fragments linearly.
- Produce programs whose semantics are sound with respect to the Obc sources. The soundness property and its proof are detailed in section 10.4.
- Produce programs whose translation to Flat IR by the pipeline of chapter 9 is similar to the standard translation to Clight. In particular, functions update the states of the nodes in-place rather than by copy.

```
struct acc  {4: 0 cell{i32}}
struct main {9: 0 cell{bool}; 1 acc; 5 acc}

fn acc_step(p : ptr, ghost self_0 : acc p, x : opt i32)
  : (ghost self_1 : acc p, y : opt i32)
fn acc_reset(p : ptr, ghost self_0 : acc p)
  : (ghost self_1 : acc p)
fn main_step(p : ptr, ghost self_0 : main p, i : opt i32)
  : (ghost self_1 : main p, c : opt bool, a : opt i32)
fn main_reset(p : ptr, ghost self_0 : main p)
  : (ghost self_1 : main p)

fn main() : ()
```

Figure 10.4: Top-level Frag IR declarations in the translation of figure 10.2a [velus/tests/to_frag/src/two_acc_expected.frag]

Among these properties, only the well-formedness and the semantics soundness of the generated program are proved formally (in section 10.4).

TODO (rocq)

Test sans les optims (DCE)

We experimentally test the other properties by checking the absence of compilation errors, and examining the generated Flat IR programs from a few Lustre programs. Our test suite of Lustre programs includes the examples provided by Vélus and programs designed to test specific aspects of our translation.

This section describes the syntactic translation of the different concepts of Obc: types (section 10.3.1), generalized variables (section 10.3.2), expressions (section 10.3.3), statements (section 10.3.4), methods (section 10.3.5) and the main loop (section 10.3.6). Figures 10.4 and 10.7 reproduce extracts of the translation of our running example. In particular, figure 10.7a is the implementation of the method `main.step`. We recalled the source statements on the right of their translation. At some lines, we also annotated some subsets of the invariant `M` of the translation, which is a map from source generalized variables to target variables (section 10.3.2).

10.3.1 Types and signatures

Our translation processes the classes in the order of their declarations. The translation of different classes and methods are mostly independent, except for the allocation of names for structures and functions in the target.

Types and values We translate the values of Vélus using the scalars of Frag IR. The translation of an enumeration type is the smallest scalar type within `unit`, `bool` and our integer types containing enough values to inject its constructors. In particular, the enumeration `bool` of Vélus is translated by the `bool` scalars of Frag IR. We note $(\!|\texttt{ty}|\!)_{\text{type}}$ the scalar type translating a Vélus type `ty` : `Velus.type`. When translating a temporary variable, we wrap its scalar type inside the `opt` type constructor to allow undefined values.

Memories For each Obc class `cl`, we declare a structure type with identified $(\!|\texttt{cl}|\!)_{\text{struct}}$ defining the layout of its memory. Figure 10.4 reproduces the declarations corresponding to our running example. In this presentation, $(\!|.|\!)_{\text{struct}}$ is the identity from class to structure names. For this program, the memory of the class `main` is thus encoded by a 9-byte-long structure with three fields at offsets 0, 1 and 5 corresponding to the memory variable and sub-instances of the class. In the general case, the fields of the translation of the class are:

Source declaration	Generalized variable	`GVar.src_ty`	`GVar.trg_ty`
`x : ty`	`var x`	`ty`	`opt` $(\!|\texttt{ty}|\!)_{\text{type}}$
`state mv : ty`	`state mv`	`ty`	`cell{`$(\!|\texttt{ty}|\!)_{\text{type}}$`}` `(GVar.ptr mv)`
`instance mi : cl`	`instance mi`	`cl`	$(\!|\texttt{cl}|\!)_{\text{struct}}$ `(GVar.ptr mi)`

Figure 10.5: Attributes of generalized variables

- Its memory variables. A declaration `state mx : ty` is translated by a field `cell{`$(\!|\texttt{ty}|\!)_{\text{type}}$`}`.
- Its sub-instances. A declaration `instance ma : cl` is translated by a field $(\!|\texttt{cl}|\!)_{\text{struct}}$.

Contrary to Clight, Frag IR does not infer the sizes and the offsets of structures. They are computed during the translation of Obc, using an auxiliary module `StrcBuilder` independent of Obc. Since there are currently no alignment constraints in Frag IR, we do not insert padding between the fields.

Method signatures A method `mt` in a class `cl` is translated by a function with identifier $(\!|\texttt{cl}, \texttt{mt}|\!)_{\text{method}}$. The signature of this function only depends on the signature of the source. Indeed, the translation of a method with signature `mt(`$\texttt{arg}_i$ `: `$\texttt{ty_arg}_i$`...) returns (`$\texttt{res}_i$ `: `$\texttt{ty_res}_i$`...)` has signature:

```
fn (|cl, mt|)_method(p : ptr, ghost self_0 : (|cl|)_struct p, arg_i : opt (|ty_arg_i|)_type...)
  : (ghost self_1 : (|cl|)_struct p, res_i : opt (|ty_res_i|)_type...)
```

This Frag IR signature thus contains:

- The arguments and results of the Obc method. They are optional values since they can be undefined.
- Like in Clight, a pointer argument `p` specifying the address where the memory of the instance is stored.
- Structured memory fragments `self_0` (resp. `self_1`) containing the state of the memory of the class before (resp. after) the method call.

10.3.2 Generalized variables

We factorize the encoding of the different parts of the state of an Obc method, by introducing a new notion, local to the method:

Definition 10.1 A *generalized variable* is either a temporary variable `x` (that is, an argument, a local variable or a result of the method), a memory variable `mv` or a sub-instance `mi`. Their set `GVar.t` is defined as a disjoint union:

```
GVar.t ::= var x | state mv | instance mi
```

The generated Frag IR program contains variables storing the translation of the values of generalized variables. The types of those target variables vary between the different kinds of generalized variable and are given by `GVar.trg_ty` in figure 10.5. Given `gv : GVar.t`, we write `GVar.src_ty gv` for its source type. This type is a standard Vélus type for temporary and memory variables but a class identifier for sub-instances. The translation of a memory variable or sub-instance `mx` has a dependent Frag IR type indexed by the pointer where it is stored in memory. This pointer is stored in a Frag IR variable `GVar.ptr mx` which does not change within the translation of the body of the method. In figure 10.7a, `GVar.ptr mx` is named "`p_mx`".

During the translation of the body of a method, we maintain a map `M : GVar.t ⇀ var` from the generalized variables declared in Obc to corresponding Frag IR variables. Like for the simplification pass (section 7.1), our implementation uses reversed De Bruijn indices [33] as a stable representation of target variables.

10.3.3 Expressions

Contrary to Clight, Frag IR does not support expressions within program computations. An Obc expression `e` is instead translated to a Frag IR instruction $(\!|\texttt{e}|\!)^{\texttt{o}}_{\text{expr}}$ `M` returning the evaluation of the expression. The parameter `M` maps generalized variables to their translation. The parameter `o` is a Boolean specifying whether the result of the expression must be computed as an optional value. That is, if `o` is set (for instance when translating the arguments of methods), the Frag IR instruction has result type `opt` $(\!|\texttt{ty}|\!)_{\text{type}}$, where `ty` is the Vélus type of `e`. Otherwise (for instance when translating the arguments of an operator) it has type $(\!|\texttt{ty}|\!)_{\text{type}}$. In order to respect the parameter `o`, we use `Opt.some` and the combination of `assert<is_some _>()` and `Opt.pre_get` to convert values between $(\!|\texttt{ty}|\!)_{\text{type}}$ and `opt` $(\!|\texttt{ty}|\!)_{\text{type}}$. An alternative could be to always produce an optional value, but $(\!|.|\!)^{\texttt{false}}_{\text{expr}}$ avoids some round trips between $(\!|\texttt{ty}|\!)_{\text{type}}$ and `opt` $(\!|\texttt{ty}|\!)_{\text{type}}$.

The translation is performed by recursion on the syntax of the Obc expression. The map `M` remains constant since expressions do not modify the state of the Obc program. This map is used to obtain the value of temporary and memory variables in the target:

$(\!|\texttt{x}|\!)^{\texttt{true}}_{\text{expr}}$ `M` := `(return (M (var x)))`

$(\!|\texttt{state(mv)}|\!)^{\texttt{false}}_{\text{expr}}$ `M` :=

```
  let r, _ = Frag.read{(|GVar.src_ty mv|)_type}(GVar.ptr mv, M (state mv));
  return (r)
```

For unary and binary operations, the target program starts by computing the results of the sub-expressions. We then use an encoding of the operators of Vélus as Frag IR instructions. In particular, we use our module `CopToScalar` (already mentioned in section 8.1.1.1) to translate the scalar operators of Clight used by Obc. This encoding is not always a single Frag IR operation. As an example, consider the translation of `b ≠ state(mb)` at lines 29-42 of figure 10.7a. After computing the two arguments (within `arg_0` and `arg_1`), the operator `≠` is translated by a sequence of an equality test and a Boolean negation.

10.3.4 Statements

Like expressions, statements are translated by recursion on the Obc syntax. However, there are significant differences between the handling of generalized variables in Obc and the functional SSA discipline of Frag IR:

- The syntactic scope of Obc variables is the whole method.
- Memory variables and sub-instances may be over-written, whereas the variables are immutable in Frag IR. The syntax of Obc also allows modifying defined temporary variables, but Vélus guarantees that it is not the case in the programs it generates.
- The value of a variable may depend on the control flow selected by `switch` statements. For instance, after the `switch` of figure 10.2, the output `a` is equal to `a0` if the `False` branch was taken, but to the result of `acc(ma1).step` otherwise. Moreover, the state of the sub-instance `ma1` is only updated in the second case.
  In Frag IR, new bindings must be inserted (with `let` or a local continuation) for variables which depend on the edge leading to a control flow junction in order to use them after the junction. In standard SSA, those variables are also signaled using ϕ-functions. There are no equivalent syntactic marks in Obc.

Because of those differences, the translation of Obc involves a kind of SSA construction. This construction is simplified by the absence of loops. This enables translating statements in a single pass, following the direction of the control flow.

```
{ x := …;                          let x_0 = …; {
  y := … };                   →    let y_0 = …; {
z := x;                            let z_0 = return (x_0); □ }}
```

a: Sequence

```
switch b {                                  – M_pre (state y) = y_0
  | False => state(y) := …                  let (x_2, y_3) = {
  | True  => _                                if b_0 then {
  | _     => x := …;                            let x_0 = …;
             state(y) := …                      let y_1 = …;
}                                               – M_true (var x) = x_0; M_true (state y) = y_1
                                                return (x_0, y_1)
                                              } else {
                                                let y_2 = …;
                                                – M_false (state y) = y_2
                                                let (x_1, _) = Opt.none<…>();
                                                return (x_1, y_2)
                                              }
                                            };
                                            – M_post (var x) = x_2; M_post (state y) = y_3
                                            □
```

b: Conditional

Figure 10.6: Translation of statement combinators

Translation of variables The central element of our SSA construction is the map `M` from the generalized Obc variables to Frag IR variables. This map varies within statements. When the value of an Obc variable `gv` changes, a new variable $\texttt{gv}_i$ is bound in Frag IR and `M` is updated with the binding $\texttt{M gv} = \texttt{gv}_i$. Following SSA conventions, in our examples, the different Frag IR variables associated to an Obc variable (its "versions" in SSA parlance [85, §3.1.3]) have its name with a numerical suffix. For instance, in figure 10.7a, at line 43 before the assignment `state(mb) := b`, the memory variable `mb` is mapped to `mb_1`. The translation of the assignment generates a new variable `mb_2` using `Frag.write`. Line 48, after the assignment, `mb` is mapped to `mb_2`, which contains its updated value.

Sequences Since the scope of Obc variables is not restricted, we must ensure that their corresponding Frag IR variables remain accessible. In particular, left-nested sequences must be compiled to right-nested sequences. In figure 10.6a, the variable `x_0` would otherwise not be visible during the translation of the assignment to `z`. In order to generate right-nested sequences, we defined a monad, called `InstrMonad`, and whose effect is to generate sequences of instructions. A significant part of the implementation, including the translation of statements, is defined within this monad. In this presentation, we unfold the definition of `InstrMonad`. With this unfolding, an Obc instruction is translated to a Frag IR instruction with a hole □ for a continuation[23], placed at a point which has access to the translation of the variables. The implementation of `InstrMonad` represents those "instructions with a hole" using functions `Instr.t` → `Instr.t` taking as argument the continuation.

Conditionals We support `switch` statements on arbitrary enumerations. Figure 10.6b and figure 10.7a depict the case of Boolean guards, where the target contains a single `if` conditional. In the general case, we emit a linear sequence of `if` conditionals with equality tests, nested in their `then` branches. As future work, a more efficient translation could be obtained by adding switch statements to Frag IR or by implementing a dichotomy with a balanced tree of `if` conditionals.
In all cases, the Frag IR translation contains new binders for the generalized variables modified in one or several branches. For instance, in figure 10.6b, the temporary `x` is modified only in the case `True` and

[23] In this context, a "continuation" is an arbitrary instruction, not necessarily a continuation declared with `krec`.

the memory variable `y` in both cases, but with different assignments. Those variables are mapped by M_{post} to new `let`-bound variables `x_2` and `y_3` in the continuation $\square$ of the translation. In the general case, we insert a new binder for a generalized variable `gv` if and only if there exists a branch such that the map M_b at the end of this branch differs on `gv` with the map M_{pre} before the conditional. To efficiently compute this set of modified variables, our implementation maintains the set of generalized variables whose binding in `M` has changed since the beginning of the innermost conditional. We could reduce the size of the translation by not emitting new binders for dead temporary variables (that is, not used in the continuation). For instance, in figure 10.7a, we insert at line 53 a new binder `a1_2` for `a1`, although it is not used after the `switch`. Removing those variables would be a kind of pruned SSA construction [85, 2.4]. Our DCE on Frag IR (section 8.3.2) achieves it a posteriori.
Each branch returns the translation of the variables for which we insert new binders, using their own maps M_b. In figure 10.6b, the `then` branch thus returns its translations `x_0` and `y_1` of `x` and `y`. As illustrated by the result for `x` in the `else` branch, we use `Opt.none` to return a value for temporary variables not in the map M_b of the branch. Those variables are necessarily uninitialized in this path.

Omitting some implementation details (such as translation failures or the use of reversed De Bruijn indices) and unfolding `InstrMonad`, the translation of statements is given by a function with signature:

$$(\!|.|\!)_{\text{stmt}} \ \left(M_{pre} : \texttt{GVar.t} \rightharpoonup \texttt{var}\right) : (\texttt{GVar.t} \rightharpoonup \texttt{var}) \times (\texttt{Instr.t} \to \texttt{Instr.t})$$

where the two maps from generalized variables to Frag IR variables describe their relations before and after the statement and the function `Instr.t` $\to$ `Instr.t` represents a Frag IR instruction with a hole $\square$ for the continuation.

```
fn main_step(p: ptr, ghost self_0: main p, i: opt i32)
         : (ghost self_1: main p, c: opt bool, a: opt i32) {
▶ Prologue: elimination of self_0
  let (ghost pad  : mpadding{main} p,
       ghost mb_0 : cell{bool} (add{ptr} p const{i64, 0}),
       ghost ma1_0: acc (add{ptr} p const{i64, 1}),
       ghost ma0_0: acc (add{ptr} p const{i64, 5}), _) =
    Strc.elim{main}(p, self_0);
  let (ofs_mb: i64, ghost EQ_ofs_mb: ofs_mb = const{i64, 0}) = const{i64, 0}();
  let (p_mb: ptr, ghost EQ0_p_mb: p_mb = add{ptr} p ofs_mb) = add{ptr}(p, ofs_mb);
  let (ghost EQ_p_mb: p_mb = add{ptr} p const{i64, 0}) = lemma{
      (p: ptr, ofs: i64, EQ_ofs: ofs = const{i64, 0},
       p_f: ptr, EQ0_p_f: p_f = add{ptr} p ofs)
    : (EQ_p_f: p_f = add{ptr} p const{i64, 0})
    }(p, ofs_mb, EQ_ofs_mb, p_mb, EQ0_p_mb);                          | Computation of p_mb
  let (ghost mb_1: cell{bool} p_mb, _) = lemma{
      (p: ptr, p_f: ptr, EQ_p_f: p_f = add{ptr} p const{i64, 0},
       f_0: cell{bool} (add{ptr} p const{i64, 0}))
    : (f_1: cell{bool} p_f, EQ: f_1 = f_0)
      consume (f_0)
    }(p, p_mb, EQ_p_mb, mb_0);                                        | Cast of mb_0 into mb_1
  ··· | Computation of p_ma1 and p_ma0 and casts into ma1_1 and ma0_1
▶ Translation of the method body
  let (arg_ma0_0: opt i32) = return (i);
  let (ghost ma0_2: acc p_ma0, a0_0: opt i32) =                       | a0 := acc(ma0).step([i])
    fcall acc_step(p_ma0, ma0_1, arg_ma0_0);
  let b_0: opt bool = ...;                                            | b := i ≥ 0
  – M (var b) = b_0; M (state mb) = mb_1
  let c_0: opt bool = {                                               |
    let v_c: bool = {                                                 |
      let arg_0: bool = {                                             |
        let (ghost H_some: is_some b_0) = assert<is_some b_0>();      |
        let (v_b: bool, _) = Opt.pre_get<bool>(b_0, H_some); return (v_b)};
      let arg_1: bool = {                                             |
        let (v_mb: bool, _) = Frag.read{bool}(p_mb, mb_1); return (v_mb)};
      let v_op_0: bool = {                                            | c := b ≠ state(mb)
        let (v_op_0: bool, _) = eq{bool}(arg_0, arg_1); return (v_op_0)};
      let v_op_1: bool = {                                            |
        let (v_op_1: bool, _) = not(v_op_0); return (v_op_1)};        |
      return (v_op_1)};                                               |
    let (c: opt bool, _) = Opt.some<bool>(v_c); return (c)            |
  };                                                                  |
  – M (state mb) = mb_1                                               |
  let new_mb: bool = {                                                |
    let ghost H_some: is_some b_0 = assert<is_some b_0>();            | state(mb) := b
    let (v_b: bool, _) = Opt.pre_get<bool>(b_0, H_some); return (v_b)};
  let (ghost mb_2: cell{bool} p_mb, _) = Frag.write{bool}(p_mb, mb_1, new_mb);
  – M (state mb) = mb_2
▶ Translation of the switch
  let cond: bool = {
    let ghost H_some: is_some b_0 = assert<is_some b_0>();
    let (v_b: bool, _) = Opt.pre_get<bool>(b_0, H_some); return (v_b)};
  let (ghost ma1_3: acc p_ma1, a_2: opt i32, a1_2: opt i32) =
    if cond _ then {
      let (arg_ma1_0: opt i32) = return (i);
      let (ghost ma1_2: acc p_ma1, a1_0: opt i32) =                   | a1 := acc(ma1).step(i)
        fcall acc_step(p_ma1, ma1_1, arg_ma1_0);
      let (a_0: opt i32) = return (a1_0);                             | a := a1
      return (ma1_2, a_0, a1_0)
```

```
60 ||   } else {
61 ||     let (a_1: opt i32) = return (a0_0);            |a := a0
62 ||     let (a1_1: opt i32, _) = Opt.none<i32>();
63 ||     return (ma1_1, a_1, a1_1)
64 ||   };
65 ▶ Epilogue: building of self_1
66 |  let (ghost mb_3: cell{bool} (add{ptr} p const{i64, 0}), _) = lemma{
67 |      (p: ptr, p_f: ptr, EQ_p_f: p_f = add{ptr} p const{i64, 0},
68 |       f_0: cell{bool} p_f)
69 |    : (f_1: cell{bool} (add{ptr} p const{i64, 0}), EQ: f_1 = f_0)
70 |      consume (f_0)
71 |    }(p, p_mb, EQ_p_mb, mb_2);
72 |  ··· | Casts of ma1_3 and ma0_2 into ma1_4 and ma0_3
73 |  let (ghost self_1: main p, _) = Strc.make{main}(p, pad, mb_3, ma1_4, ma0_3);
74 |  return (self_1, c_0, a_2)
75 }
```

Cast of mb_2 into mb_3

a: Translation of the method main.step

```
fn main() : () {
  let (sz : i64, ghost EQ_sz : sz = const{i64, 9}) = const{i64, 9}();
  let (nl : ptr, ghost EQ_nl : nl = null) = null();
  let (p : ptr, ghost ma_0 : malloc p sz) = Frag.malloc(sz);
  let (err : bool, ghost EQ_err : err = eq{ptr} p nl) = eq{ptr}(p, nl);
  if err H_err then {
     lemma{
         (nl : ptr, err : bool, p : ptr, sz : i64, ma_0 : malloc p sz,
          EQ_nl : nl = null, EQ_err : err = eq{ptr} p nl, EQ_sz : err = true)
       : () consume (ma_0)
       }(nl, err, p, sz, ma_0, EQ_nl, EQ_err, H_err);
     return ()
  } else {
    let (ghost m_0 : main p, ghost ma_1 : mallocated p sz) = lemma{
         (nl : ptr, err : bool, p : ptr, sz : i64, ma_0 : malloc p sz,
          EQ_nl : nl = null, EQ_err : err = eq{ptr} p nl,
          H_err : err = false, EQ_sz : sz = const{i64, 9})
       : (r_0 : main p, r_1 : mallocated p sz)
         consume (ma_0)
       }(nl, err, p, sz, ma_0, EQ_nl, EQ_err, H_err, EQ_sz);
     let ghost m_1 : main p = fcall main_reset(p, m_0);
     krec {
       loop(ghost m_2 : main p):
         let v_i : i32 = input{i32}();
         let (i : opt i32, _) = Opt.some<i32>(v_i);
         let (ghost m_3 : main p, c : opt bool, a : opt i32) =
           fcall main_step(p, m_2, i);
         let ghost H_some_c : is_some c = assert<is_some c>();
         let (v_c : bool, _) = Opt.pre_get<bool>(c, H_some_c);
         output{bool}(v_c);
         let (ghost H_some_a : is_some a) = assert<is_some a>();
         let (v_a : i32, _) = Opt.pre_get<i32>(a, H_some_a);
         output{i32}(v_a);
         kcall loop(m_3)
     }(m_1)
   }
}
```

b: Main loop

Figure 10.7: Extract of the translation of figure 10.2a to Frag IR [velus/tests/to_frag/src/two_acc_expected.frag]

10.3.5 Methods

The function translating a method performs some operations before (prologue) and after (epilogue) the instruction translating its body. Those operations interface the arguments and results of the function with the invariant maintained during the translation of the statements (defined by the translation `M` of the generalized variables). Most importantly, the memory of the instance is given (resp. returned) as a single structured memory fragment `self_0` (resp. `self_1`).

The prologue of the translation starts by splitting `self_0` into its fields, using `Strc.elim`. This yields a Frag IR variable for each memory variable and sub-instance. However, the types of those variable involve pointer arithmetic. For instance, in figure 10.7a, we obtain a cell `mb_0 : cell{bool} (add{ptr} p const{i64, 0})` containing the value of the memory variable `mb`. The translation of statements instead requires this translation to be stored in a variable with type `cell{bool} (GVar.ptr (state mb))` where `GVar.ptr (state mb)` is a pointer variable[24]. For each memory variable or sub-instance `mx`, we first compute a pointer `GVar.ptr mx` to the corresponding field. This computation is the sequence of a constant loading and an addition. Using a `lemma`, we derive a proposition reflecting the value of `GVar.ptr mx`. We then use a second lemma to cast the fragment returned by `Str.elim` into a fragment indexed `GVar.ptr mx`. Figure 10.7a only reproduces the operations for `state mb`, which produce a pointer `GVar.ptr (state mb) = p_mb`, a proposition `EQ_p_mb: p_mb = add{ptr} p const{i64, 0}` and a fragment `mb_1: cell{bool} p_mb`. The initial map M_{pre} of the translation of the method body contains the arguments translating the inputs of the node and the cast fields of `self_0`.

The epilogue performs the inverse of the prologue. We build `self_1` by grouping the translation of the memory variables and sub-instances given by the final map M_{post} of the method body. Before invoking `Str.make`, we have to cast back the fragments to types involving addresses relative to the base pointer `p` of the structure. We achieve it with `lemma`s, and reuse the equalities generated by the prologue (e.g., `EQ_p_mb`). We terminate the function by returning `self_1` and the translation of the outputs given by M_{post}.

Initially, we wanted to perform those pointer arithmetic operations and casts of memory fragments directly during the translation of instructions, around the memory accesses and calls to sub-instances. Compared to the translation we actually use, this would reduce the live ranges of the pointers to the fields. However, with this encoding, the translation of a read of a memory variables would consume its cell, because a lemma would cast it before `Frag.read`. This would make the translation of expressions more complex, because it would need to change the `cell` associated to the memory variable in order to use fragments linearly. Moreover, this would introduce dependencies between the reads of the same memory variable.
To summarize, performing the pointer computations where we initially wanted would have been harder, and would have introduced more dependencies between the operations. The root cause is an interaction between the linearity condition and the dependent types of the fragments. For future work, we consider modifying Frag IR to avoid dependent types for memory fragments. A cell would thus have a type `cell{…}` not indexed by its address, in order to avoid casting memory fragments. Relations between memory fragments and pointers could still be enforced using separate proposition variables (e.g., with a type "`&c = p`" to express that a cell `c` is stored at address `p`), taken as pre-condition by `Frag.read` and `Frag.write`.

10.3.6 Main loop

The translation generates a main function which executes the source program indefinitely, using input and outputs external events. This function only depends on the signature of the `step` method of the

[24] Here, `mb` is stored at offset `0`, but `cell{bool} (add{ptr} p const{i64, 0})` is not equivalent to `cell{bool} p` in the type system of Frag IR.

main node (that is, the signature of the main node in the Lustre source) and of the size of the structure storing its memory. Figure 10.7b reproduces its definition for our example.

It starts by allocating a structured fragment `m_0` for the memory. This allocation is implemented using `Frag.malloc` followed by a test of success. If `Frag.malloc` fails, the main function terminates immediately (a lemma consumes the empty fragment `malloc` in order to satisfy the linearity criterion). Otherwise, a lemma extracts `m_0` from `malloc` (that is, split it from the deallocation permission `ma_1`). The function `main_reset` (translating the method `main.reset`) then initializes the memory.

The program then enters an infinite loop implemented with `krec` and with the memory of the node as loop variable. At each step, the program reads the inputs using a sequence of `input` events and wraps them in `opt` types. The function `main_step` (translating the method `main.step`) is called with the current state of the memory and the inputs to execute a step of the system. It returns the new state of the memory and the outputs for the current instant, wrapped in `opt`. Those outputs are unwrapped by `Opt.pre_get` (the outputs of the main node must always be present) then emitted by `output`.

10.4 Correctness of the translation

We prove the correctness of the translation from Obc to Frag IR. In order to use the relational reasoning approach of `SMonad`, section 10.4.1 defines an alternative semantics for Obc, within the domain `LTS.t`. Section 10.4.2 proves a refinement between the generated Frag IR program and this alternative semantics. Section 10.4.3 gives a specification for the behaviors of the main loop and shows that it is preserved by our compilation pipeline up to Flat IR.

10.4.1 Alternative monadic semantics for Obc

To facilitate the proof of the translation from Obc to Frag IR, we defined an alternative semantics for Obc. The main differences with the standard semantics are that:

- The alternative semantics is denotational and its domain is the model `LTS.t` of `SMonad`. This simplifies refinements between Obc and Frag IR programs, since the semantics of Frag IR is defined in a domain `FragIR.S` derived from `LTS.t`.
- We reformulate the type of program states using the notion of generalized variables introduced in section 10.3.2. The connection between standard Obc states and our generalized variables is thus handled during the switch to the alternative semantics, rather than during the proof of the translation.
- We define a semantics for the main loop which allows programs to terminate immediately, in a non-deterministic fashion. This is a change with respect to the standard semantics of Obc. This additional behavior is motivated by the implementation of the main loop in Frag IR, which terminates immediately if the allocation of the memory of the main node fails. It was easier to make this change during the switch to a `SMonad`-based semantics. Indeed, later, the notion of refinement of `SMonad` does not allow introducing new behaviors.

Figure 10.8 describes the correspondence between the original and alternative semantics of the components of Obc. We prove equivalences between the standard big-step and alternative semantics of the expressions, statements and methods. For this purpose, we use our framework `FromEval`, described in section 5.5.1. For the translation of Obc to Frag IR, we actually only need a unidirectional correctness property between standard big-step semantics `P` and alternative semantics `u` of Obc: $\forall$`x`. `P x` $\Rightarrow$ `u` $\approx$ `ret x`. This weaker property is implied by the relation `FE.p P u` of `FromEval`, and proving equivalences is not significantly harder thanks to the framework surrounding it. For the semantics of the main loop, we only prove a unidirectional correctness property, which is detailed in section 10.4.3.2. In the following, we describe the signatures of our alternative semantics, and their connection with the standard semantics.

Component	Original	Alternative	Relation
Expressions	`exp_eval`	$[\![.]\!]_{\text{expr}}$	equivalence `FE.p`, lemma 10.1
Statements	`stmt_eval`	$[\![.]\!]_{\text{stmt}}$	equivalence `FE.p` with a call handler, lemma 10.2
Methods	`stmt_call_eval`	$[\![.]\!]_{\text{call}}$	equivalence `FE.p`, lemma 10.3
Main loop	`sem_prog`	$[\![.]\!]^{\not\downarrow}_{\text{prog}}$	unidirectional on the behaviors, lemma 10.8

Figure 10.8: Correspondence between the original and alternative semantics of Obc

State In Obc, methods are evaluated within a state composed of two environments `menv` and `venv`. The former stores the memory variables and the sub-instances as a tree, while the latter stores the temporary variables. We group those two components in a single state type `Obc.State.t` := `menv` × `venv`. This type has the interface of a lookup map, whose keys are generalized variables and whose type of values depends on the kind of key accessed:

$$\texttt{GVar.val_t gv} := \begin{cases} \texttt{option Velus.value} & \text{if } \texttt{gv} = \texttt{var x} \text{ and where } \varnothing \text{ means undefined} \\ \texttt{option Velus.value} & \text{if } \texttt{gv} = \texttt{state mv} \text{ and where } \varnothing \text{ means uninitialized} \\ \texttt{menv} & \text{if } \texttt{gv} = \texttt{instance mi} \end{cases}$$

```
Obc.State.get (st : Obc.State.t) (gv : GVar.t) : GVar.val_t gv
Obc.State.set (gv : GVar.t) (v : GVar.val_t gv) (st : Obc.State.t) : Obc.State.t
```

Temporary and memory variables are mapped to optional values. For temporaries, a $\varnothing$ is interpreted as an undefined value, which can be read by the Obc program, as long as no operations are performed on it. This enables the representation of absent values, and their presence in method arguments. For memory variables, a $\varnothing$ represents uninitialized memory, which triggers a UB if the variable is read. In practice, the method `main.reset` initializes the memory. Sub-instances are mapped to the representation of their state as a memory tree `menv`.

Expressions The alternative semantics of expression has signature:

$$[\![\texttt{e}]\!]_{\text{expr}}\ (\texttt{st} : \texttt{Obc.State.t}) : \texttt{LTS.t}\ \varnothing\ (\texttt{option Velus.value})$$

That is, given a state `st`, our denotational semantics of an expression `e` is a LTS which does not emit events and which returns the value it evaluates to. Like in the standard semantics the `option Velus.value` allows expressions to return undefined results.
This semantics is defined by structural recursion on `e`, using the operators of `SMonad` on `LTS.t` to combine the semantics of sub-expressions. For instance, the semantics of a binary operation is:

> **TODO** (rocq)
>
> preuve que notre sémantique est équivalente

$$[\![\texttt{e}_0\ \texttt{op}\ \texttt{e}_1]\!]_{\text{expr}}\ \texttt{st} :=$$
$$\texttt{o}_0 \leftarrow [\![\texttt{e}_0]\!]_{\text{expr}}\ \texttt{st}; \texttt{o}_1 \leftarrow [\![\texttt{e}_1]\!]_{\text{expr}}\ \texttt{st}; \begin{cases} \texttt{ret}\ \lfloor \texttt{v} \rfloor & \text{if } \texttt{o}_0 = \lfloor \texttt{v}_0 \rfloor,\ \texttt{o}_1 = \lfloor \texttt{v}_1 \rfloor \text{ and } [\![\texttt{op}]\!]_{\text{bop}}\ \texttt{v}_0\ \texttt{v}_1 = \lfloor \texttt{v} \rfloor \\ \texttt{ub} & \text{otherwise} \end{cases}$$

where $[\![\texttt{op}]\!]_{\text{bop}}\ (\texttt{v}_0\ \texttt{v}_1 : \texttt{Velus.value}) : \texttt{option Velus.value}$ denotes the semantics of a binary operator `op`, and returns $\varnothing$ for UBs. A drawback of our monadic semantics with respect to the standard big-step semantics is that we have to define an evaluation order. We arbitrarily evaluate $\texttt{e}_0$ before $\texttt{e}_1$. However, since $[\![.]\!]_{\text{expr}}$ has restricted effects, we could commute those evaluations during proofs.
The equivalence with the standard big-step semantics of expressions is a direct application of our framework `FromEval`:

Lemma 10.1 For any Obc expression `e` and method state `(me, ve)`:

$$\texttt{FE.p}\ (\texttt{exp_eval}\ \texttt{me}\ \texttt{ve}\ \texttt{e})\ \left(\llbracket \texttt{e} \rrbracket_{\text{expr}}\ (\texttt{me}, \texttt{ve})\right)$$

Idea of the proof By structural induction on `e`, using the lemmas on `FE.p`.

Statements Contrary to expressions, statements emit events corresponding to internal method calls[25]. We represent those events using a family `Intern` generated by:

```
mk_Intern cl mt (me : menv) (argv : list (option Velus.value)) :
  Intern (menv × list (option Velus.value))
```

where `cl` and `mt` are the class and method of the callee, `me` the state of the instance of `cl` before the call and `argv` the list of values given as arguments. The answer to an internal call is the state of the instance of `cl` after the call and the list of values returned.
The alternative semantics of statements has thus signature:

$$\llbracket \texttt{a} \rrbracket_{\text{stmt}}\ (\texttt{st}_0 : \texttt{Obc.State.t}) : \texttt{LTS.t Intern Obc.State.t}$$

The semantics of a statement `a` takes as argument the state $\texttt{st}_0$ of the method before its execution and returns the state after. It is defined by structural recursion on `a`. For instance the semantics of sequences is defined using the `bind` combinator on `LTS.t`:

$$\llbracket \texttt{a}_0\texttt{;}\ \texttt{a}_1 \rrbracket_{\text{stmt}}\ \texttt{st}_0 := (\texttt{st}_1 \leftarrow \llbracket \texttt{a}_0 \rrbracket_{\text{stmt}}\ \texttt{st}_0; \llbracket \texttt{a}_1 \rrbracket_{\text{stmt}}\ \texttt{st}_1)$$

For method calls, we emit corresponding events using `trigger`.
We prove the equivalence with the big-step semantics using `FromEval`. However, we cannot relate directly $\llbracket . \rrbracket_{\text{stmt}}$ to `stmt_eval`. Indeed, the former emits events corresponding to internal calls while the latter depends on previous top-level declarations `prog` and uses them to directly evaluate the calls. Our equivalence is thus parameterized by a handler, which is assumed to replace internal call events by monadic semantics equivalent to the big-step semantics of the callees.

Lemma 10.2 Let `prog` be a list of top-level declarations and `h : handler Intern ∅` a handler for internal calls. Assume that for any internal call $\texttt{c} := \texttt{mk_Intern cl mt me}_0\ \texttt{argv}$, the semantics `h c : LTS.t ∅ (menv × list (option Velus.value))` is related by `FE.p` to the standard semantics of the call in `prog` (and in particular is `ub` if `cl.mt` is not declared in `prog`):

$$\texttt{FE.p}\ (\texttt{stmt_call_eval prog}\ \texttt{me}_0\ \texttt{cl mt argv})\ (\texttt{h c})$$

Then, for any Obc statement `a` and initial method state $(\texttt{me}_0, \texttt{ve}_0)$:

$$\texttt{FE.p}\ (\texttt{stmt_eval prog}\ \texttt{me}_0\ \texttt{ve}_0\ \texttt{a})\ (\texttt{interp h}\ (\llbracket \texttt{a} \rrbracket_{\text{stmt}}\ (\texttt{me}_0, \texttt{ve}_0)))$$

Idea of the proof By structural induction on `a`, using the lemmas on `FE.p`. The hypothesis on `h` is used in the case of method calls. We use lemma 10.1 when expressions are evaluated.

Method Given a list `prog` of top-level declarations, we define the semantics of an arbitrary method call `cl.mt`:

```
⟦prog, cl.mt⟧call (me₀ : menv) (argv : list (option Velus.value)) :
  LTS.t ∅ (menv × list (option Velus.value))
```

This semantics is defined by composing the semantics of the bodies of the method using `mrec`. However, like the standard semantics of Obc, we rule out recursion by assigning UB to calls to methods not defined strictly earlier in the list `prog`. We chose to use `mrec` instead of a fixpoint of Rocq because it yields a definition closer to the semantics of Frag IR and because it avoids proving the termination of chains of calls.

[25] Obc statements can also call pure external functions. Those calls are supported by our alternative semantics through events, but not by the translation to Frag IR. We omit them in this presentation.

$$
\begin{array}{l}
[\![\texttt{prog}]\!]_{\text{prog}} : \texttt{LTS.t Obc.IO} \varnothing := \\
\quad (\texttt{me}_0, _) \leftarrow [\![\texttt{prog}, \texttt{main.reset}]\!]_{\text{call}}\ \texttt{mempty}\ [\,]; \\
\quad \texttt{iter}\ \left(\begin{array}{l}
\lambda \texttt{me}_1 . \\
\quad \texttt{i}_0 \leftarrow \texttt{trigger}\ (\texttt{Obc.Input}\ (\texttt{ty_arg prog})_0); \\
\quad \texttt{i}_1 \leftarrow \ldots; \\
\quad (\texttt{me}_2, \texttt{rs}) \leftarrow [\![\texttt{prog}, \texttt{main.step}]\!]_{\text{call}}\ \texttt{me}_1\ [\lfloor \texttt{i}_0 \rfloor; \lfloor \texttt{i}_1 \rfloor; \ldots]; \\
\quad \texttt{o} \leftarrow \left(\begin{cases} \texttt{ret}\ [\texttt{o}_0; \texttt{o}_1; \ldots] & \text{if } \texttt{rs} = [\lfloor \texttt{o}_0 \rfloor; \lfloor \texttt{o}_1 \rfloor; \ldots] \\ \texttt{ub} & \text{otherwise} \end{cases}\right); \\
\quad \texttt{trigger}\ (\texttt{Obc.Output}\ (\texttt{ty_res prog})_0\ \texttt{o}_0);; \\
\quad \ldots;; \\
\quad \texttt{ret}\ (\texttt{Continue}\ \texttt{me}_2)
\end{array}\right)\ \texttt{me}_0 \\
[\![\texttt{prog}]\!]^{\lightning}_{\text{prog}} : \texttt{LTS.t Obc.IO unit} := \left(\texttt{ok} \leftarrow \texttt{any bool}; \begin{cases} \varnothing \leftarrow [\![\texttt{prog}]\!]_{\text{prog}}; _ & \text{if } \texttt{ok} = \texttt{true} \\ \texttt{ret tt} & \text{if } \texttt{ok} = \texttt{false} \end{cases}\right)
\end{array}
$$

Figure 10.9: Semantics of the main loop in our alternative semantics of Obc

Lemma 10.3 For any list `prog` of top-level declarations, method `cl.mt`, memory state $\texttt{me}_0$ before the call and arguments `argv`:

$$\texttt{FE.p}\ (\texttt{stmt_call_eval prog me}_0\ \texttt{cl mt argv})\ \left([\![\texttt{prog}, \texttt{cl.mt}]\!]_{\text{call}}\ \texttt{me}_0\ \texttt{argv}\right)$$

Idea of the proof By strong induction on the order of declaration of the classes in `prog`. At each step, we use `mrec_unfold` to rewrite $[\![.]\!]_{\text{call}}$ into a semantics with shape $\texttt{interp h}\ ([\![\texttt{a}]\!]_{\text{stmt}}\ \texttt{st})$, where the statement `a` is the body of the callee and the handler `h` is defined using $[\![.]\!]_{\text{call}}$. We conclude with lemma 10.2. Its hypothesis on `h` is justified using our induction hypothesis, since calls to classes that are not defined strictly earlier have UB in the standard and alternative semantics.

Main loop Our semantics of the main loop is defined in figure 10.9. A first variant $[\![.]\!]_{\text{prog}}$ defines the non-terminating loop (which is reflected by its empty return type, hence a later binder $\varnothing \leftarrow _; _$ for its result), and $[\![.]\!]^{\lightning}_{\text{prog}}$ allows the semantics to return immediately (depending on a non-deterministic choice of a Boolean `ok`), to model memory allocation failures. Both semantics emit external input/output events, whose family `Obc.IO` is generated by:

```
Obc.Input (ty : Velus.type) : Obc.IO Velus.value
Obc.Output (ty : Velus.type) (v : Velus.value) : Obc.IO unit
```

We include the type of the value expected/emitted in the events, because the translation of values to Frag IR depends on those types. Moreover, we do not wrap values in `option`s, since the main node should not have absent values in its arguments and results.
The semantics $[\![.]\!]_{\text{prog}}$ of the main loop starts by initializing the memory of the main node using the method `main.reset`. The memory `mempty` before this call is completely uninitialized. The infinite loop is then defined using an `iter`, whose state is the memory of the main node. At each iteration, the semantics reads the inputs by emitting sequentially an event `Obc.Input` for each argument of the main node (in figure 10.9, `ty_arg` denotes the list of the types of the arguments of the method `main.step`). It then calls the method `main.step` with the current state $\texttt{me}_1$ of the memory and the inputs as arguments. This call returns a new memory state $\texttt{me}_2$ and some results `rs`. We raise a UB if any result is undefined. Otherwise, we sequentially emit `Obc.Output` events corresponding to those results, joined with their types (given by the list `ty_res` of the result types of `main.step`). We then continue the loop from the new state $\texttt{me}_2$.

10.4.2 Specifications of the translation functions

Thanks to our alternative `SMonad`-based semantics for Obc, we prove the correctness of the translation to Frag IR by relational reasoning. More precisely, we prove for each syntactic category that:
1. Its translation is well-formed, that is, implements a DTS.

```
tr_val (ty : Velus.type) (vsrc : Velus.value) (vtrg : val) : Prop
tr_val_opt (ty : Velus.type) (vsrc : option Velus.value) (vtrg : val), Prop
tr_val_cell (ty : Velus.type) (vsrc : option Velus.value) (vtrg : val), Prop
```

TODO (rocq)

- example Vlong

$$\frac{}{\texttt{tr_val int64 (Vscalar (Vlong i)) (Vsc (Vint (Int64 i)))}}$$

$$\frac{}{\texttt{tr_val_opt ty} \varnothing \texttt{ (Vopt} \varnothing\texttt{)}} \qquad \frac{\texttt{tr_val ty vsrc vtrg}}{\texttt{tr_val_opt ty } \lfloor\texttt{vsrc}\rfloor \texttt{ (Vopt } \lfloor\texttt{vtrg}\rfloor\texttt{)}}$$

$$\frac{}{\texttt{tr_val_cell ty} \varnothing \texttt{ (Vcell } (\!|\texttt{ty}|\!)_{\text{type}} \texttt{ p bs)}} \qquad \frac{\texttt{tr_val ty vsrc vtrg} \quad \texttt{bs} = \texttt{encode vtrg}}{\texttt{tr_val_cell ty } \lfloor\texttt{vsrc}\rfloor \texttt{ (Vcell } (\!|\texttt{ty}|\!)_{\text{type}} \texttt{ p bs)}}$$

Figure 10.10: Translation relations for the values (extract for `tr_val`)

2. Its semantics refines the semantics of the source. This refinement is expressed in the domain `LTS.t`.

Our translation of values depends on their type, so we always assume that the Obc source is well-typed. This section describes the main invariants of our proof, and the statement of the correctness for the translation of some elements.

Values Figure 10.10 describes several translation relations between source and target values. We first define a relation `tr_val ty vsrc vtrg`, which handles unwrapped source values `vsrc : Velus.value`. It is parameterized by the source type `ty`, and handles both the scalars inherited from CompCert and the enumerations. The target value `vtrg` is always a scalar `Vsc`. The definition of the relation depends on `ty`, as an example, figure 10.10 reproduces the case of 64-bit integers (where `Vlong` is the corresponding value constructor in CompCert).

We derive translation relations for the values of temporary and memory variables. The former are translated by `tr_val_opt` to optional values `Vopt`, defined if and only if the source is. The later are translated by `tr_val_cell` to cell fragments `Vcell`. Here, a source $\varnothing$ represents an uninitialized memory, which is related to arbitrary bytes `bs`, since there is no notion of uninitialized memory in Frag IR. The address `p` of the cell is quantified existentially in `tr_val_cell`. More generally, the invariants of our proofs do not track the values of pointers. Indeed, the dependent type system of Frag IR already ensures that memory accesses are performed with correct pointer arguments, so we do not have to reprove it.

Memory Memory trees of Obc are translated by structured memory fragments of Frag IR with the same shape. The corresponding translation relation for the memory of a class `cl` has signature `tr_mem cl (me : menv) (vtrg : val) : Prop`, where `me` is the source memory tree, and `vtrg` its translation as a `Vstrc`. Formally, this relation depends on the structure layout chosen by the translation pass, but we leave this dependency implicit here. It is defined inductively, following the common structure of the source and target values:

Definition 10.2 Given an Obc memory tree `me : menv` and some Frag IR structure fields `fs : list val` and padding `pad`, the relation `tr_mem cl me (Vstrc (pad :: fs))` holds if `fs` has the expected length and:

- For any memory variable `state mv : ty` declared in `cl`, the corresponding field `f : val` in `fs` satisfies `tr_val_cell ty (me mv) f`, where `me mv : option Velus.value` is the value of `mv` in `me`.
- For any sub-instance `instance mi : cl'` declared in `cl`, the corresponding field `f : val` in `fs` satisfies `tr_mem cl' (me mi) f`, where `me mi : menv` is the subtree of `me` corresponding to `mi`.

Generalized variables A map `M : GVar.t ⇀ var` from generalized Obc variables to Frag IR variables defines two invariants, described by definition 10.3 and definition 10.4 below. The first is static, and ensures that the target contains the variables involved in `M`:

Definition 10.3 A scope `V` of Frag IR variables *contains the image* of `M` if:

- For any binding `M gv = trg_x`, the variable `trg_x` is bound in `V`.
- For any memory variable or sub-instance `mx` bound in `M`, the variable `GVar.ptr mx` storing the address of its translation is bound in `V`.

This static invariant relates the state `M` of the translation function to the structure of the generated Frag IR program. It is used to prove that this program is well-formed. Assuming that this first invariant holds, we define a second, semantic, invariant between the static state `M` of the translation function and the dynamic states of the Obc and Frag IR programs.

Definition 10.4 Given a scope `V` containing the image of `M`, a state `st : Obc.State.t` and values `ρ : V → val` for the variables of `V`, the relation `tr_state M V st ρ` holds if:

- For each binding `M gv = trg_x`, the target value `ρ trg_x : val` translates `Obc.State.get st gv : GVar.val_t gv`. The translation relation depends on the kind of `gv`, and is either `tr_val_opt` (for temporaries), `tr_val_cell` (for state variables) or `tr_mem` (for sub-instances).
- If a temporary `x` is defined (that is, `Obc.State.get st (var x) ≠ ∅`), then it is bound in `M`.
- For any memory variable or sub-instance `mx` bound in `M`, the value of `GVar.ptr mx` is an arbitrary pointer: `∃p : Ptr.t. ρ (GVar.ptr mx) = Vsc (Vptr p)`.

The second condition allows the translation to use `Opt.none` to obtain the translation of temporaries not bound in `M`. The third condition does not specify the value of the variables `GVar.ptr` since we do not track the values of pointers. However, we currently need the fact that they store pointer values to prove that pointer arithmetic operations do not have UB. We could maybe remove this part of the invariant by relying on the soundness of the type system of Frag IR (this would require running the type-checker directly on the generated program), or by modifying the semantics of total scalar operators so that they never have UB (even on ill-typed arguments).

Internal calls Internal calls are represented by events in our alternative Obc semantics and in the semantics of Frag IR. The invariant between those calls is defined as a relation on events `tr_int_call : EventFam.Rel.t Intern (Fct.Call.ev F)`, where `F` is the set of Frag IR functions.

Definition 10.5 Given an Obc method `cl.mt`, the corresponding Frag IR function identifier `f : fct`, source memory $\texttt{msrc}_0$: `menv` and arguments `arg_src : list (option Velus.value)` and target arguments `p : Ptr.t`, $\texttt{mtrg}_0$: `val` and `arg_trg : list val`, the event relation `tr_int_call` holds between `mk_Intern cl mt` $\texttt{msrc}_0$ `arg_src` and `Fct.Call.mk f (Vsc (Vptr p)) ::` $\texttt{mtrg}_0$ `:: arg_trg` if:

- `tr_mem cl` $\texttt{msrc}_0$ $\texttt{mtrg}_0$
- `tr_val_opt` holds between corresponding elements of `arg_src` and `arg_trg`

If those two pre-conditions hold, the calls are related with a post-condition ensuring that the source and target answers have respective shapes ($\texttt{msrc}_0$: `menv`, `res_src : list (option Velus.value)`) and ($\texttt{mtrg}_1$ :: `res_trg`) : `list val`, and that they satisfy:

- `tr_mem cl` $\texttt{msrc}_1$ $\texttt{mtrg}_1$
- `tr_val_opt` holds between corresponding elements of `res_src` and `res_trg`

Expressions Let `e` be an Obc expression with type `ty : Velus.type`. The specification of its translation $(\!|\texttt{e}|\!)^{\texttt{o}}_{\text{expr}}$ is a refinement between $[\![\texttt{e}]\!]_{\text{expr}}$ and the semantics of the target instruction. Its statement depends on the parameter `o`. If `o` is false, we add a UB in the source semantics if its result is undefined. This semantically justifies the use of `assert<is_some _>` to unwrap the result in the target.

Lemma 10.4 Let `M` be a map for the generalized variables and `V` a scope containing its image. Then $(\!|\texttt{e}|\!)^{\texttt{o}}_{\text{expr}}$ `M` is well-formed. Moreover, for any state `st : Obc.State.t` and valuation `ρ` of `V` satisfying `tr_state M V st ρ`:

- if `o` $=$ `true`, the target semantics directly refines the semantics of the expression:

$$[\![\texttt{e}]\!]_{\text{expr}}\ \texttt{st} \ \gtrsim_{\bot,\texttt{tr_val_opt ty}}\ \texttt{to_LTS}\ \left([\![(\!|\texttt{e}|\!)^{\texttt{true}}_{\text{expr}}\ \texttt{M}]\!]_{\text{instr}}\ \rho\right)$$

- if `o` $=$ `false`, the option result is unwrapped in the source:

$$\texttt{r} \leftarrow [\![\texttt{e}]\!]_{\text{expr}}\ \texttt{st}; \begin{cases} \texttt{ret v} & \text{if } \texttt{r} = \lfloor \texttt{v} \rfloor \\ \texttt{ub} & \text{if } \texttt{r} = \varnothing \end{cases} \ \gtrsim_{\bot,\texttt{tr_val ty}}\ \texttt{to_LTS}\ \left([\![(\!|\texttt{e}|\!)^{\texttt{false}}_{\text{expr}}\ \texttt{M}]\!]_{\text{instr}}\ \rho\right)$$

Idea of the proof By induction on the syntax of `e`.

Statements The translation of statements generates instructions with a hole □ for the continuation. Such instruction with a hole is represented by a function `Instr.t` → `Instr.t` in the implementation. However, we only use a subset of this type. Indeed, except for the presence of the hole, the generated instruction does not depend on the continuation. Moreover, the continuation is always executed, at the end of the instruction. The semantics of the completed instruction is thus equivalent to a sequence, whose left-hand side does not depend on the continuation and whose right-hand side is the semantics of the continuation. Our specification reflects those restrictions on the returned function.

TODO (rocq)

unfold the definitions

Lemma 10.5 Let a be a well-typed Obc statement, M_{pre} a map for the generalized variables before a and V_{pre} a scope containing the image of M_{pre}. Assume that the translation of a returns M_{post} and u : Instr.t → Instr.t, i.e., $(\!|a|\!)_{stmt}\ M_{pre} = (M_{post}, u)$. Then, there exists:

- A scope $V_{post} \supseteq V_{pre}$ for the hole of u
- A semantics sem_u $(\rho_{pre} : V_{pre} \to val) : FragIR.S\ (Fct.Call.ev\ F)\ (V_{post} \to val)$, taking as argument the values of the target variables before u and returning the values ρ_{post} before the continuation

Such that:

- V_{post} contains the image of M_{post}.
- The valuation $\rho_{post} : V_{post} \to val$ returned by sem_u extends the valuation $\rho_{pre} : V_{pre} \to val$ given as argument. That is, for any variable $x \in V_{pre}$, $\rho_{post}\ x = \rho_{pre}\ x$.
- For any continuation k : Instr.t well-formed in V_{post}, the completed instruction u k is well-formed in V_{pre}.
- For any such continuation k and values $\rho_{pre} : V_{pre} \to val$:

$$[\![u\ k]\!]_{instr}\ \rho_{pre} \approx bind\ (sem_u\ \rho_{pre})\ [\![k]\!]_{instr}$$

Moreover, for any state $st_{pre} : Obc.State.t$ and valuation $\rho_{pre} : V_{pre} \to val$ satisfying tr_state $M_{pre}\ V_{pre}\ st_{pre}\ \rho_{pre}$:

$$[\![a]\!]_{stmt}\ st_{pre} \gtrsim_{(tr_int_call;\, of_fun\ Ev.inl),(tr_state\ M_{post}\ V_{post})} to_LTS\ (sem_u\ \rho_{pre})$$

where the composition with of_fun Ev.inl : EventFam.Rel.t (Fct.Call.ev F) (Fct.Call.ev F + external_event) accounts for the events external_event, which are in the signature of to_LTS but not emitted by the translation of statements.

Idea of the proof In this specification, we unfolded some definitions provided by a general framework we formalized around InstrMonad. This framework also provides lemmas to reason on the instructions built using the monad. Our proof of lemma 10.5 uses those lemmas. It is performed by induction on the syntax of a. Lemma 10.4 is used for statements containing expressions.

Methods The relation tr_int_call on internal calls is also the semantic specification of the translation of the methods. We thus prove the following result on the semantics of the target program starting with a call to a translation of a method:

Lemma 10.6 If a well-typed Obc program prog_src is successfully translated to a Frag IR program prog_trg, and if an Obc method call and a Frag IR function call are related by:

$$tr_int_call\ (mk_Intern\ cl\ mt\ msrc_0\ arg_src)\ (Fct.Call.mk\ f\ vargs)\ Post$$

then:

$$[\![prog_src, cl.mt]\!]_{call}\ msrc_0\ arg_src \gtrsim_{\bot, Post} to_LTS\ ([\![prog_trg, f]\!]_{prog}\ vargs)$$

Idea of the proof We first prove a similar result for the semantics $[\![.]\!]_{func}$ of isolated target functions, using tr_int_call as relation on events. This first result is derived from the correctness of the translation of method bodies (lemma 10.5), by showing that the prologue and epilogue of target functions interface correctly the generalized variables with the arguments and results. We then combine the correctness of the functions using ref_mrec.

Main loop We defined our alternative semantics $[\![.]\!]^{\lightning}_{prog}$ for the main loop of Obc so that it is similar to the semantics of the generated `main` function. In particular, $[\![.]\!]^{\lightning}_{prog}$ allows the program to terminate immediately when `Frag.malloc` fails. Moreover, there is a one-to-one correspondence between the external events, described by a relation on events tr_io : EventFam.Rel.t Obc.IO external_event. This relation is defined using tr_val, thanks to the presence of the source types in Obc.IO:

$$\frac{}{\texttt{tr_io}\ (\texttt{Obc.Input ty})\ (\texttt{Input}\ (\texttt{ty})_{\text{type}})\ (\lambda \texttt{vsrc vtrg.}\ \texttt{tr_val ty vsrc}\ (\texttt{Vsc}\ (\texttt{Scalar.to_t}\ (\texttt{ty})_{\text{type}}\ \texttt{vtrg})))}$$

$$\frac{\texttt{tr_val ty vsrc}\ (\texttt{Vsc}\ (\texttt{Scalar.to_t}\ (\texttt{ty})_{\text{type}}\ \texttt{vtrg}))}{\texttt{tr_io}\ (\texttt{Obc.Output ty vsrc})\ (\texttt{Output vtrg})\ \top}$$

Thanks to this similarity between their semantics, we relate the Obc and Frag IR programs using a refinement in `LTS.t`:

Lemma 10.7 If a well-typed Obc program `prog_src` is successfully translated to a Frag IR program `prog_trg`, then `prog_trg` is well-formed and:

$$[\![\texttt{prog_src}]\!]^{\lightning}_{\text{prog}} \quad \gtrsim_{\texttt{tr_io},\top} \quad [\![\texttt{prog_trg}]\!]^{\text{Frag}}_{\text{prog}}\ []$$

Idea of the proof By relational reasoning. We simulate the non-deterministic semantics of `Frag.malloc` with `any bool` using the refinements rules of `any`. We relate the two loops using `ref_iter`. We relate the calls to `main.reset` and `main.step` to their translations using lemma 10.6.

10.4.3 Correctness of the main loop

In section 10.4.1, we defined an alternative semantics for Obc, including $[\![.]\!]^{\lightning}_{\text{prog}}$ for its main loop. In section 10.4.2, we showed with lemma 10.7 that the semantics of the Frag IR program generated by the translation refines $[\![.]\!]^{\lightning}_{\text{prog}}$. In this section, we relate $[\![.]\!]^{\lightning}_{\text{prog}}$ to the standard stream-based semantics of Obc. By composing this relation with the correctness of Vélus and of our pipeline to Flat IR, we obtain a correctness property relating the semantics of the target Flat IR program to the semantics of the source Lustre program. Because our languages are non-deterministic, this property is more complex than the one of the standard back-end to Clight.

Section 10.4.3.1 describes the property we prove on the behaviors of generated programs. Section 10.4.3.2 uses this specification for the main loop of Obc and section 10.4.3.3 composes the different correctness results.

10.4.3.1 Specification

From Lustre (`sem_node`) to the standard semantics of Obc (`sem_prog`), the semantics of a program is a set of pairs of streams (`ins`, `outs`). However, starting from our alternative semantics of Obc ($[\![.]\!]^{\lightning}_{\text{prog}}$), the semantics of a program is instead defined as a LTS. The predicate `lts_behaves` defines the set of behaviors of those semantics, where behaviors contain finite or infinite sequences of events (section 5.4). This is similar to the standard translation to Clight. The correctness theorem for the standard pipeline states that, if the Obc source admits some streams (`ins`, `outs`), then the Clight target admits a corresponding behavior. However, such property in forward style is not strong enough for our non-deterministic languages and LTSs (Frag IR, Flat IR and the alternative semantics of Obc). Indeed, because of internal non-determinism, proving that the set of behaviors of a program contains an expected behavior does not rule out the existence of undesirable behaviors (with wrong outputs or even UB) for the same inputs. Instead, we directly give an upper bound on sets of behaviors. This upper bound is the predicate `beh_react_or_ret B sg ins outs beh`, defined in figure 10.11 and described in the remainder of this section.

Structure of the executions The predicate `beh_react_or_ret` describes the set of behaviors `beh` (using our type `Beh.t` of coinductive behaviors) compatible with some streams `ins` and `outs` of inputs and outputs. Here, those streams are actually functions $\mathbb{N} \to$ `list Velus.value` specifying at each instant `i` the inputs `ins i` (resp. outputs `outs i`) of the main node, in the order of declaration of its arguments (resp. results). The argument `sg` is the signature of the main node. The predicate is parameterized by the events `E` and results `A` of the behaviors (implicitly) and by a record `B` specifying the behaviors compatible with a single input (`B.input`) or output (`B.output`). Those instances of `B` for Obc (`Obc.B`) and the external events of Frag IR and Flat IR (`FragIR.B`) are given in figure 10.11b. This parameterization enables sharing the definition of `beh_react_or_ret` between those languages. We detail below the semantics of the parameters, but intuitively `B.input (ty, v) K beh` (resp. `B.output (ty, v) K beh`) takes as argument a single input (resp. output) value `v`, its type `ty` and the set `K` : `Beh.t E A` $\to$ `Prop` of behaviors admissible after the event. It describes the behaviors `beh` compatible with the event corresponding to `v` and the continuation `K`. Using the parameter `B`, we first define the behaviors `beh_react B sg ins outs i beh` compatible with the streams `ins` and `outs` after the instant `i`. This predicate is defined coinductively. For each instant `i`, we examine the head of the behavior to check that it is compatible with the sequence of inputs and outputs at `i`. We use an auxiliary function `beh_seq` to recognize behaviors whose head is a sequence of events accepted by `B.input` or `B.output`. We derive `beh_react_or_ret` by also allowing immediate returns `Beh.Ret`, corresponding to allocation failures.

Events The instances `Obc.B` and `FragIR.B` of `B` are defined similarly. Both use the auxiliary predicate `beh_event e K beh`, which checks that a behavior `beh` starts with the emission of an event `e` and that its tail (if any) satisfies `K`. This predicate `K` takes as argument the answer received from the event. Thanks to `beh_event`, the behavior can either stop at the emission of `e` (`Beh.Stp`) or receive an answer (`Beh.Vis`) for which `K` is satisfied on the tail.
Consider the instance `Obc.B`, for the events `Obc.IO` emitted by our alternative semantics of Obc. Those events contain the same source `Velus.value`s as the streams. When the streams prescribe an output of a value `v`, `Obc.B.output (v, ty) K` thus requires the behavior to start with the emission of a corresponding event `Obc.Output ty v`. Moreover, the tail of the behavior (for a `Beh.Vis`) must continue to match the streams, as described by the continuation `K`. Similarly, when the streams prescribe an input `v`, `Obc.B.input (v, ty) K` requires the behavior to start with an event `Obc.Input ty`. However, the answer `a` : `Velus.value` is chosen by the environment, not the program. Hence, the programs necessarily admit behaviors where the inputs are not the chosen streams `ins`. We thus do not require

```
beh_pred : Type := (Beh.t E A → Prop) → (Beh.t E A → Prop)
B.input, B.output : Velus.type × Velus.value → beh_pred

beh_seq [X : Type] (beh_X : X → beh_pred) : list X → beh_pred
beh_seq beh_X [] K := K
beh_seq beh_X (x :: xs) K := beh_X x (beh_seq beh_X xs K)
```

$$\text{coinductive}\ \frac{\begin{array}{l}\texttt{beh_seq B.input (List.combine sg.ty_arg (ins i)) (}\\ \texttt{beh_seq B.output (List.combine sg.ty_res (outs i)) (}\\ \texttt{beh_react B sg ins outs (i + 1)))}\end{array}}{\texttt{beh_react B sg ins outs i beh}}$$

$$\frac{}{\texttt{beh_react_or_ret B sg ins outs (Beh.Ret res)}} \qquad \frac{\texttt{beh_react B sg ins outs 0 beh}}{\texttt{beh_react_or_ret B sg ins outs beh}}$$

a: Definitions parametrized by the signature of the behaviors

```
beh_event [C] (e : E C) (K : C → Beh.t E A → Prop) : Beh.t E A → Prop
```

$$\frac{}{\texttt{beh_event e K (Beh.Stp e)}} \qquad \frac{\texttt{K a tl}}{\texttt{beh_event e K (Beh.Vis e a tl)}}$$

```
Obc.B.input (ty, v) K := beh_event (Obc.Input ty) (λa beh. a = v ⇒ K beh)
Obc.B.output (ty, v) K := beh_event (Obc.Output ty v) (λ_. K)

FragIR.B.input (ty, v) K :=
  beh_event (Input (ty)type) (λa beh. tr_val ty v (Vsc (Scalar.to_t (ty)type a)) ⇒ K beh)
FragIR.B.output (ty, v) K beh :=
  ∃vtrg. tr_val ty v (Vsc vtrg) ∧ beh_event (Output vtrg) (λ_. K) beh
```

b: Instances for Obc and Frag IR (also used for Flat IR)

Figure 10.11: Set of behaviors compatible with some given streams `ins` and `outs` of inputs and outputs

the answer `a` provided by the environment to be equal to the input `v` chosen in `ins`. Moreover, we only constrain the tail of the behavior when the program receives the expected input, that is, when `a` = `v`.

Intuitively, the set `beh_react_or_ret Obc.B sg ins outs` contains the behaviors where the program emits the outputs `outs` *as long as* it receives the inputs `ins`. However, it no longer provides any guarantees if the environment gives an unexpected input (not matching `ins`).

Example 10.1 Consider infinite streams extending the interactions represented in figure 10.1, that is:

$$\texttt{ins} := [0 \mapsto [3], 1 \mapsto [-1], 2 \mapsto [8], 3 \mapsto [5], \ldots]$$
$$\texttt{outs} := [0 \mapsto [\bot; 3], 1 \mapsto [\top; 2], 2 \mapsto [\top; 11], 3 \mapsto [\bot; 16], \ldots]$$

The set `beh_react_or_ret Obc.B {ty_arg := [int]; ty_res := [bool; int]} ins outs` contains the following behaviors:

- The infinite, reactive behavior corresponding to (`ins`, `outs`):

```
Beh.Vis (Obc.Input int) 3 (Beh.Vis (Obc.Output bool ⊥) tt
  (Beh.Vis (Obc.Output int 3) tt (Beh.Vis (Obc.Input int) −1 ...)))
```

- Any prefix of this behavior, where the program is stopped after emitting an event:

```
Beh.Vis (Obc.Input int) 3 (Beh.Stp (Obc.Output bool ⊥))
```

- The behavior which returns immediately:

```
Beh.Ret tt
```

- Any behavior (including UB) after an `Obc.Input` not receiving the expected answer:

$$\texttt{Beh.Vis}\ (\texttt{Obc.Input}\ \texttt{int})\ 4\ \texttt{Beh.Err}$$

However, since we use `beh_react_or_ret` as an upper bound, what is formally interesting is the behaviors it does not contain. We will show that the set of behaviors of our target programs is included in `beh_react_or_ret`. Hence, as long as they receive inputs matching `ins`, they cannot:

- Have UB, diverge silently or emit an `Obc.Input` (resp. `Obc.Output`) when an output (resp. input) was expected:

$$\texttt{Beh.Vis}\ (\texttt{Obc.Input}\ \texttt{int})\ 3\ (\texttt{Beh.Vis}\ (\texttt{Obc.Output}\ \texttt{bool}\ \bot\ \texttt{tt}\ \texttt{Beh.Err}))$$

- Terminate after emitting some events:

$$\texttt{Beh.Vis}\ (\texttt{Obc.Input}\ \texttt{int})\ 3\ (\texttt{Beh.Ret}\ \texttt{tt})$$

- Emit wrong outputs:

$$\texttt{Beh.Vis}\ (\texttt{Obc.Input}\ \texttt{int})\ 3\ (\texttt{Beh.Vis}\ (\texttt{Obc.Output}\ \texttt{bool}\ \top\ \texttt{tt}\ \ldots))$$

10.4.3.2 Behaviors of the main loop of Obc

We show that any pair of input and output streams admitted by `sem_prog` provides a corresponding upper bound `beh_react_or_ret` on the behaviors of $[\![.]\!]^{\lightning}_{\text{prog}}$. This correctness property mixes forward and backward style: we assume that the source admits some streams (forward style), but we prove an upper bound on the set of target behaviors (backward style).

Lemma 10.8 For any Obc program `pr` with signature `sg` and streams `(ins, outs)` of inputs and outputs satisfying `sem_prog pr ins outs`:

$$\texttt{lts_behaves}\ [\![\texttt{pr}]\!]^{\lightning}_{\text{prog}} \subseteq \texttt{beh_react_or_ret Obc.B sg ins outs}$$

Idea of the proof We reason on the behaviors of $[\![\texttt{pr}]\!]^{\lightning}_{\text{prog}}$. Instead of reasoning directly on `lts_behaves` and on the states of $[\![.]\!]^{\lightning}_{\text{prog}}$, we proved some generic lemmas about the behaviors of LTSs built using some operators of `SMonad`. Those lemmas give expressions for the behaviors of all operators involved in $[\![.]\!]^{\lightning}_{\text{prog}}$ (`ret`, `ub`, `trigger`, `bind` and `iter`), but we do not yet have lemmas for `interp` and `mrec`. For instance, the lemma for `bind` (`lts_bind_eq` in the development) intuitively states that the behaviors of `bind u k` are obtained from the behaviors of `u` by replacing terminations `Beh.Ret x` by behaviors of `k x`.

We use the equivalence given by lemma 10.3 between the standard and alternative semantics of `main.reset` and `main.step`. We first handle the case of an immediate return. Then, the assumption `sem_prog` ensures that the initialization $[\![\texttt{pr}, \texttt{main.reset}]\!]_{\text{call}}$ is deterministic and silent. We obtain a goal `beh_react` that we prove by coinduction, using for each iteration the assumption given by `sem_prog` for the corresponding instant.

10.4.3.3 Correctness of the compilation pipeline

The instance `FragIR.B` handles the external events `external_event` of Frag IR in a way similar to `Obc.B`. It only differs by the use of the translation function $(\!|.|\!)_{\text{type}}$ on the types and of the relation `tr_val` on the values. We are thus able to prove a correctness property for the translation from Obc to Frag IR which relates the behaviors of the generated program to the standard semantics of Obc.

Lemma 10.9 If a well-typed Obc program `prog_src` with signature `sg` is successfully translated to a Frag IR program `prog_trg`, then `prog_trg` is well-formed. Moreover, for any streams `(ins, outs)` of inputs and outputs satisfying `sem_prog prog_src ins outs`:

$$\texttt{lts_behaves}\ \left([\![\texttt{prog_trg}]\!]^{\text{Frag}}_{\text{prog}}\ []\right) \subseteq \texttt{beh_react_or_ret FragIR.B sg ins outs}$$

```
fn main_step(p : ptr, i : i32) : (c : bool, a : i32) {
  let (ofs_mb  : i64,  _) = const{i64, 0}();
  let (p_mb    : ptr,  _) = add{ptr}(p, ofs_mb);
  let (ofs_ma1 : i64,  _) = const{i64, 1}();
  let (p_ma1   : ptr,  _) = add{ptr}(p, ofs_ma1);
  let (ofs_ma0 : i64,  _) = const{i64, 5}();
  let (p_ma0   : ptr,  _) = add{ptr}(p, ofs_ma0);
  let (a0_0    : i32    ) = fcall acc_step(p_ma0, i);
  let (zero    : i32,  _) = const{i32, 0}();
  let (b_0     : bool, _) = s_ge{i32}(i, zero);
  let (v_mb    : bool   ) = Flat.read{bool}(p_mb);
  let (v_op_0  : bool, _) = eq{bool}(b_0, v_mb);
  let (c_0     : bool, _) = not(v_op_0);
  Flat.write{bool}(p_mb, b_0);
  let (a_2 : i32) = if b_0 _ then {
    let (a1_0 : i32) = fcall acc_step(p_ma1, i);
    return (a1_0)
  } else return (a0_0);
  return (c_0, a_2)
}
```

Figure 10.12: Extract of the compilation of figure 10.1a to Flat IR [velus/tests/to_frag/src/two_acc_expected.flat]

Idea of the proof By lemma 10.8, we have an upper bound `beh_react_or_ret Obc.B` on the behaviors of $[\![\texttt{prog_src}]\!]^{\lightning}_{\text{prog}}$. By lemma 10.7, $[\![\texttt{prog_trg}]\!]^{\text{Frag}}_{\text{prog}}$ [] refines $[\![\texttt{prog_src}]\!]^{\lightning}_{\text{prog}}$. This refinement is a heterogeneous backward simulation, with a relation `tr_io` on the events. We thus cannot use lemma 5.1, since it only handles homogeneous simulations. Generalizing this lemma requires a heterogeneous refinement relation between sets of behaviors. We explored several definitions, and found one (`bhs_href_set` in the development) implied by the heterogeneous simulation (that is, satisfying a generalization `simu_bhs_href_set` of lemma 5.1). This definition is not completely satisfactory, because it is strictly weaker than the inclusion of behaviors in the homogeneous case. However, it is strong enough to show the preservation of `beh_react_or_ret`.

By composing the pipeline of Vélus from Lustre to Obc, our translation from Obc to Frag IR and our pipeline `frag_to_flat` from Frag IR to Flat IR, we obtain a compilation pipeline `lus_to_flat` from Lustre to Flat IR. Figure 10.12 reproduces the Flat IR function implementing the `step` function of the `main` node of our running example. Taking into account the details of the formalization of Vélus and of our development, the correctness of the whole pipeline is stated as:

TODO (rocq)

- renommage ?

Lemma 10.10 Let `lus` be a Lustre abstract syntax tree. Assume that `lus` is successfully elaborated into a Lustre program `G` containing a main node `main` and let `sg` be the signature of this node. If our pipeline `lus_to_flat` successfully produces the AS `flat` of a Flat IR program, then there exists a well-typed DTS program `s_p` implemented by `flat`. Moreover, if `ins` are well-typed and well-clocked input streams, and `outs` some output streams such that `sem_node G main ins outs` is satisfied, then:

$$\texttt{lts_behaves}\ \left([\![\texttt{s_p}]\!]^{\text{Flat}}_{\text{prog}}\ []\right) \subseteq \texttt{beh_react_or_ret FragIR.B sg ins outs}$$

Idea of the proof The correctness of Vélus implies that it generates an intermediate Obc program `obc` satisfying `sem_prog obc ins outs`. By lemma 10.9, its translation `frag` in Frag IR implements a DTS and its set of behaviors is included in `beh_react_or_ret FragIR.B sg ins outs`. By lemma 9.4, the Flat IR program `flat` generated by our pipeline also implements a DTS and its semantics refines the

semantics of `frag`. This refinement is a homogeneous simulation, so implies the inclusion of the sets of behaviors by lemma 5.1.
The pipeline `lus_to_flat` contains additional defensive checks to simplify the proof:

- We run the type-checker on the final program to ensure that it is well-typed.
- In lemma 10.9, the argument `sg` of `beh_react_or_ret` is the signature of the step function of the main class of Obc. Here, it is the signature of the main node in the Lustre source. Vélus does not prove formally that it preserves the signature of programs, so we check it defensively.

This lemma has the same assumptions on the source as the standard pipeline of Vélus to assembly (`behavior_asm` [23, theorem 6.1.1]). However, its conclusion on the behaviors of the target differs because of the internal non-determinism of Flat IR.
Bourke et al. [19, figure 5] recently proposed a new version of the correctness theorem of the standard translation to Clight. It involves a new "functional" semantics of Lustre, and implies the existence of reactive behaviors in the target under a static condition on the source, satisfied by a large class of program. Since our lemma 10.10 has the same hypothesis `sem_node` as the original theorem of Vélus, it could also be strengthened using those new results. However, the interaction with our target specification `beh_react_or_ret` has to be investigated.

Conclusion

We conclude by recalling the key aspects of our development and outline current limitations and directions for future extensions.

Encoding information in Frag IR

Our language Frag IR encodes semantic information using two mechanisms. The dependent proposition types provide relations between variables. The memory fragments reflect the structure of the memory, with a semantics considering by default that objects are not aliased. We retain the ability to encode programs featuring aliasing, by using distinct variables for pointers and fragments. While uses of memory fragments are subject to a linearity condition, pointers remain unconstrained. In both of our approaches, we make explicit in the syntax most semantic dependencies. There are exceptions, and some effectful instructions do not commute: some partial operators like `assert`, `joinable`, the external events, the infinite loops (defined using `krec`) and function calls.

Both encodings of information rely on almost standard program variables. This simplifies their propagation during program transformations. They still require specific language features, namely dependent types and a linearity condition. The implementation of passes sometimes needs to consider the propagation of information. For instance, we restrict the DCE to prevent the removal of memory fragments (section 8.3.2). However, dependent types and linearity do not influence the semantics of programs and can be checked defensively. The correctness of some passes is thus proven without reasoning on them.

By writing toy programs in Frag IR (section 8.4), we show that it can indeed encode program-specific memory structures. Our translation of Obc (chapter 10) demonstrates that it can also encode properties ensured by construction in higher-level languages.

There are several directions to improve Frag IR. During the encoding of Obc (section 10.3.5), we faced an undesirable interaction between dependent types and linearity. We envision avoiding dependent types for memory fragments, but this change remains to be implemented. Currently, Frag IR also misses a notion of read-only fragment. Such an extension could reduce the dependencies between reads, by allowing several active fragments providing access to the same region of the memory. In particular, it would be useful to encode a language like Capla [75]. However, it is not yet clear how read only fragments should be modeled in Frag IR. A source of inspiration could be the use of fractional permissions in separation logics, in particular within OptiTrust [15]. But since fractional permissions are just a particular case of monoid [49], they may introduce an undesirable coupling between the compiler and the verification techniques used at the source level. In contrast, the ownership shares used in the VST [5, chap 42] are erased to the simpler permissions of CompCert. Similarly, it would be desirable, even if fractional permissions were added in Frag IR and the compiler, that frameworks using more expressive monoids could be applied to verify source programs.

Semantics

Our parameterized SSA IR and its instances (notably Frag IR) have denotational semantics. Contrary to small-step operational semantics, denotational semantics are by nature compositional. By defining semantics for all syntactic components (e.g., instructions and functions) in isolation, they enable stating properties about them. Our proofs thus follow the syntax of programs, and avoid monolithic invariants. This is particularly effective on structured syntaxes, like functional SSA. Compared to big-step semantics, denotational semantics retain the ability to model effectful computations, including divergence.

We leverage the annotations of Frag IR to simplify its semantics. A first simplification comes from its dependent type system, which guarantees the absence of failures for most operations. In particular, it tracks the value of pointers. Combined with information on memory permissions provided by memory

fragments, this allows giving a UB-free semantics to memory operations (figure 8.9), while compiling to a target with partial memory operations (section 9.1.3). This avoids formal invariants on the values of pointers for the translation of Obc (section 10.4.2). Memory fragments enable other simplifications. The most prominent is the absence of global state (except the control state). This avoids implicit dependencies, in particular for memory allocations. Moreover, memory fragments are immutable and handled like standard values in the semantics, similarly to Capla [75] or translations of safe Rust to functional languages [37; 47]. Finally, since fragments already provide an isolation mechanism between memory regions, Frag IR features concrete pointers.

Using information for transformations

Currently, none of our automated optimizations leverage information encoded in Frag IR. However, we perform proofs on manual transformations on toy programs to check their semantic validity. This includes transformations leveraging non-aliasing information, such as commutations of memory accesses on distinct fragments, or propagations of the value of a fragment. Those proofs are dramatically simplified by the semantics of Frag IR. Indeed, fragments are values and `Frag.read` and `Frag.write` are pure, so justifying those transformations is as simple as for local variables. However, transformations on `lemma` operations remain to be explored.

Our library `SMonad` for denotational semantics

The library `SMonad` was developed to formalize our compilation pipeline, but is independent and could be reused for other developments. In fact, during his internship and now as part of his thesis, Benjamin Gaudin extended a proof of concept leveraging `SMonad` to formalize semantics of faulted programs, and to verify hardening compilers [42]. More generally, the domains and operations defined in `SMonad` can be used for multiple languages, thanks to the separation between semantic domains and program syntax in our denotational approach. This property also simplifies proofs involving distinct languages, but defined in a common domain. In our pipeline, the domain `FragIR.S` (and its unfolding to `LTS.t`) thus acts as a common ground between Frag IR and Flat IR. This is similar to the use of the same formalism `Smallstep.semantics` in all representations of CompCert. `SMonad` even includes frameworks simplifying connections with operational semantics (section 5.5).

`SMonad` is inspired by interaction trees [102] and the relational theory surrounding them, but generalizes it in several directions. Firstly, we use an axiomatization to obtain a development independent of any particular model. Secondly, we axiomatize additional effects (non-determinism and undefined behaviors) and provide a model `LTS.t` supporting them natively. Finally, we introduce relations on events to reason on functions in isolation. While most of our `SMonad`-based proofs use the same relational reasoning approach as interaction trees, we also show that the codiagonal properties of the axiomatization enable proving loop and function inlining transformations (sections 4.4.1 and 4.4.2).

The existence of models such as `LTS.t` guarantees the consistency of our axiomatization. However, we found out to our surprise that this axiomatization is inconsistent with additional desirable reasoning rules, in particular the commutation of non-determinism with events (section 4.4.3). Looking for axiomatizations compatible with those properties (and corresponding models) is an interesting direction for future work, which could allow proving more transformations by relational reasoning.

Modular implementation of a verified compiler

`SMonad` modularizes our development, by decoupling large parts of our semantic formalism from the syntax of programs. Despite their different effects and memory model, many definitions are also shared between Frag IR and Flat IR, thanks to our parameterized SSA IR. This includes both the implementation of their syntax, their meta-theory (semantics and typing judgment) and some passes. We even prove once and for all the main parts of the correctness of our parameterized passes. This factorization

is possible thanks to the use of denotational semantics and of the axiomatization of `SMonad`. The former enables quantifying over effectful semantics of operators, and the latter over the domain of the semantics.
We believe that this modular approach could be taken further. In particular, one could draw inspiration from MLIR [61] and take as parameter combinators taking instructions ("regions" in MLIR parlance) as arguments instead of only variables. Bhat et al. [18] already successfully used such parameterization in Lean, although they only have restricted support for effects. We do not anticipate any obstacle to generalizing the operators into combinators in the definition and semantics of our parameterized IRs. `SMonad` would provide the abstraction needed to support effects. The parameterized passes may require more attention. In particular, generalizing the block-sinking transformation is not obvious.
Another limitation of our current implementation is that the instances are defined monolithically. In particular, sets of operators are implemented by inductive definitions which list them exhaustively. We do share some definitions between Frag IR and Flat IR, but only because the former is a subset of the latter. For more general cases, it could be useful to be able to combine several sets of operators and values, as well as passes on them. In MLIR, this is achieved thanks to the concept of "dialect", which is implemented with object-oriented idioms in C++. Implementing this extensibility is harder with the inductive datatypes of Rocq, but automatic encodings (like the extensible datatypes provided by Rocqet [38]) could help.

Using defensive checks

To simplify proofs, our pipeline involves multiple defensive checks:

- The type-checker is run before passes that assume that their input is well-typed (section 7.4). In particular, typing assumptions are used by the translation to Flat IR to obtain relations between the pointers and fragments given as arguments to memory operations. Using a defensive check avoids proving that our passes preserve the typing judgment. We prove this preservation only for the erasure and for the translation to Flat IR, but could have used the type-checker instead.
- The translation to Flat IR checks that its input Frag IR program uses fragments linearly (section 9.3).
- The erasure pass checks defensively the consistency of the choice of erased variables and operators (section 7.2.3). This notably avoids proving the correctness of our liveness analysis.
- The last step of the block-sinking transformation uses the chosen target tree defensively (section 7.3.3).

Additionally, all our passes may fail if their input program is ill-formed (e.g., if it contains unbound references). We did not formalize in Rocq the conditions checked by our passes, except for typing and well-formedness. Even for those criteria, we do not prove that our defensive tests succeed if they are satisfied. In particular, our type-checker is not formally complete (section 8.3.1). We only prove the partial correctness of our compilation pipeline. We test on examples whether it successfully generates a target program. Currently, some edge-cases lead to compilation errors for some programs written directly in Frag IR. However, when starting from a Lustre program, we are not aware of any unexpected compilation failure of our pipeline.

Formally verified compilation pipeline

Thanks to the Vélus front-end, we obtained a verified compilation pipeline from Lustre to Flat IR. The correctness of this pipeline is proved in Rocq, by combining a fragment of the standard proof of Vélus with the correctness of our pipeline. Combining proofs of different passes, and interfacing with other frameworks is crucial to test our definitions, and the meaning of our correctness results.
As future work, new passes could be added in our pipeline. For instance:

- We defined a framework for proving inlining transformations in `SMonad` (section 4.4.2), but do not use it in our pipeline. We could implement an automatic inlining pass on the parameterized SSA IR.

- Transformations that move or factorize instructions (e.g. Common Sub-expression Elimination (CSE)) should benefit from the syntactical representation of dependencies in Frag IR. Their proof could also be simplified by the absence of state in the semantics, but it could also be interesting to try to prove them more abstractly, on the parameterized representation.
- Because of dependent types, transformations that change the operations used (e.g., constant propagation) could be harder to implement. The strategy used for `OptDefault` (section 9.2) may also apply to other passes. That is, passes could preserve all source variables (but annotate some as ghosts), so that dependent types can refer to them.

Currently, our pipeline ends at Flat IR, but executing programs natively requires compiling them to binary. The most reasonable approach would be to use an existing compiler as a back-end.

- Translating Flat IR to an IR of CompCert would provide end-to-end guarantees. Moreover, section 5.5.4 provides an interface between our LTSs and the small-step semantics of CompCert. However, we use concrete pointers, while all representations of CompCert use logical pointers. Compiling Flat IR in a formally correct way would thus require some encoding. For instance, one could compile our concrete pointers to integers, and use a single array to store the memory of the program.
- To avoid such encoding, one would need to target a language with a concrete memory model. This is the case of the second phase of VIR, with its finite memory model [14]. Vellvm currently does not include a verified compilation of VIR to binary, but it enables exporting the programs to the textual representation of LLVM IR. One can then use the standard, unverified, LLVM compiler to generate binaries. As part of his postdoctoral research, Nicolas Chappe implemented a translation from a subset of Flat IR to VIR. The semantics of VIR uses the operators of interaction trees, which are those axiomatized by `SMonad`. However, its current domain is based on sets of interaction trees [105], while we use LTSs. The correctness of the translation could be proved by reinterpreting the operators used by VIR in our domain `LTS.t`, before proving a formal correspondence between the two domains. Such two-step approach would be more robust to changes in VIR, such as the use of Choice Trees [26].

References

[1] C. Abate *et al.*, "An extended account of trace-relating compiler correctness and secure compilation," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 4, pp. 1–48, Dec. 2021, doi: 10.1145/3460860.
Referenced pages 22, 97 and 106

[2] R. Adams, "Formalized metatheory with terms represented by an indexed family of types," in *Types for Proofs and Programs*, vol. 3839, Springer, 2006, pp. 1–16. doi: 10.1007/11617990_1.
Referenced pages 124 and 135

[3] Agda Developers, "Agda." [Online]. Available: https://agda.readthedocs.io/
Referenced page 57

[4] R. Affeldt and D. Nowak, "Extending equational monadic reasoning with monad transformers," *LIPIcs, Volume 188, TYPES 2020*, vol. 188, p. 2:1–2:21, 2021, doi: 10.4230/LIPICS.TYPES.2020.2.
Referenced pages 91 and 96

[5] A. W. Appel *et al.*, *Program logics for certified compilers*, 1st ed. Cambridge University Press, 2014. doi: 10.1017/CBO9781107256552.
Referenced pages 39, 39 and 235

[6] A. W. Appel, "Foundational proof-carrying code," in *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 2001, pp. 247–256.
Referenced page 56

[7] A. W. Appel, "SSA is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, Apr. 1998, doi: 10.1145/278283.278285.
Referenced pages 14, 121, 123 and 134

[8] A. Barrière, S. Blazy, O. Flückiger, D. Pichardie, and J. Vitek, "Formally verified speculation and deoptimization in a JIT compiler," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–26, Jan. 2021, doi: 10.1145/3434327.
Referenced pages 38 and 96

[9] A. Barrière, S. Blazy, and D. Pichardie, "Formally verified native code generation in an effectful JIT: turning the CompCert backend into a formally verified JIT compiler," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 249–277, Jan. 2023, doi: 10.1145/3571202.
Referenced page 38

[10] F. Besson, S. Blazy, and P. Wilke, "CompCertS: a memory-aware verified C compiler using a pointer as integer semantics," *Journal of Automated Reasoning*, vol. 63, no. 2, pp. 369–392, Aug. 2019, doi: 10.1007/s10817-018-9496-y.
Referenced pages 33, 35 and 37

[11] M. Braun, S. Buchwald, and A. Zwinkau, "Firm—a graph-based intermediate representation," technical report 35, 2011. [Online]. Available: https://pp.ipd.kit.edu/publication.php?id=braun11wir
Referenced page 36

[12] G. Barthe, D. Demange, and D. Pichardie, "Formal verification of an SSA-based middle-end for CompCert," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 1, pp. 1–35, Mar. 2014, doi: 10.1145/2579080.
Referenced pages 37, 120, 121, 134 and 154

[13] S. L. Bloom and Z. Ésik, *Iteration theories*. Springer Berlin Heidelberg, 1993. doi: 10.1007/978-3-642-78034-9.
Referenced pages 95 and 95

[14] C. Beck, I. Yoon, H. Chen, Y. Zakowski, and S. Zdancewic, "A two-phase infinite/finite low-level memory model: reconciling integer–pointer casts, finite space, and undef at the LLVM IR level of abstraction," *Proceedings of the ACM on Programming Languages*, vol. 8, no. ICFP, pp. 789–817, Aug. 2024, doi: 10.1145/3674652.
Referenced pages 32, 34, 34, 38, 97 and 238

[15] G. Bertholon, A. Charguéraud, T. Kœhler, B. Bytyqi, and D. Rouhling, "Interactive source-to-source optimizations validated using static resource analysis," in *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, Copenhagen Denmark: ACM, Jun. 2024, pp. 26–34. doi: 10.1145/3652588.3663320.
Referenced pages 36 and 235

[16] A. Bérard, B. Bonneau, S. Boulmé, and D. Monniaux, "Formally verified compilation for aligned memory copies," Aug. 2026. [Online]. Available: https://hal.science/hal-05493413 to be published in IEEE TCAD as "Improving aligned memory copies of CompCert".
Referenced page 38

[17] Y. Bertot, B. Grégoire, and X. Leroy, "A structured approach to proving compiler optimizations based on dataflow analysis," in *Types for Proofs and Programs*, Springer Berlin Heidelberg, 2006, pp. 66–81.
Referenced page 25

[18] S. Bhat, A. Keizer, C. Hughes, A. Goens, and T. Grosser, "Verifying peephole rewriting in SSA compiler IRs," *LIPIcs, Volume 309, ITP 2024*, vol. 309, p. 9:1–9:20, 2024, doi: 10.4230/LIPICS.ITP.2024.9.
Referenced pages 135 and 237

[19] T. Bourke, P. Jeanmaire, and M. Pouzet, "Functional stream semantics for a synchronous block-diagram compiler," Singapore, Jun. 2025. [Online]. Available: https://inria.hal.science/hal-05107499v1
Referenced page 234

[20] S. Boldo and G. Melquiond, "Flocq: a unified library for proving floating-point algorithms in Coq," in *2011 IEEE 20th Symposium on Computer Arithmetic*, Tuebingen, Germany: IEEE, Jul. 2011, pp. 243–252. doi: 10.1109/ARITH.2011.40.
Referenced page 162

[21] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelona Spain: ACM, Jun. 2017, pp. 586–601. doi: 10.1145/3062341.3062358.
Referenced pages 38, 55, 135, 135 and 206

[22] T. Bourke and M. Pouzet, "Arguments cadencés dans un compilateur Lustre vérifié," Jan. 2019. [Online]. Available: https://inria.hal.science/hal-02005639
Referenced pages 210 and 210

[23] L. Brun, "Mechanized semantics and verified compilation for a dataflow synchronous language with reset," Doctoral dissertation, 2020. Accessed: Sep. 04, 2023. [Online]. Available: https://

theses.hal.science/tel-03068862
Referenced pages 206, 208, 210, 211, 211 and 234

[24] V. Capretta, "General recursion via coinductive types," *Logical Methods in Computer Science*, vol. Volume 1, Issue 2, p. 2265, Jul. 2005, doi: 10.2168/LMCS-1(2:1)2005.
Referenced page 90

[25] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '91*, Orlando, Florida, United States: ACM Press, 1991, pp. 55–66. doi: 10.1145/99583.99594.
Referenced pages 121 and 140

[26] N. Chappe, P. He, L. Henrio, Y. Zakowski, and S. Zdancewic, "Choice trees: representing nondeterministic, recursive, and impure programs in Coq," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 1770–1800, Jan. 2023, doi: 10.1145/3571254.
Referenced pages 97 and 238

[27] N. Chappe, "A family of sims with diverging interests," *Proceedings of the ACM on Programming Languages*, vol. 10, no. POPL, pp. 2083–2111, Jan. 2026, doi: 10.1145/3776714.
Referenced pages 118 and 119

[28] R. Chen, "Undefined behavior can result in time travel (among other things, but time travel is the funkiest)." Accessed: Mar. 18, 2026. [Online]. Available: https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633
Referenced page 96

[29] R. Chen, "On the limits of time travel in the face of undefined behavior in C." Accessed: Mar. 18, 2026. [Online]. Available: https://devblogs.microsoft.com/oldnewthing/20241104-00/?p=110466
Referenced page 96

[30] A. Chlipala, "Parametric higher-order abstract syntax for mechanized semantics," in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, Victoria BC Canada: ACM, Sep. 2008, pp. 143–156. doi: 10.1145/1411204.1411226.
Referenced page 135

[31] M. Cho, Y. Song, D. Lee, L. Gäher, and D. Dreyer, "Stuttering for free," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 1677–1704, Oct. 2023, doi: 10.1145/3622857.
Referenced pages 97 and 119

[32] F. Chow, S. Chan, S. M. Liu, R. Lo, and M. Streich, "Effective representation of aliases and indirect memory operations in SSA form," in *Compiler Construction*, vol. 1060, Springer Berlin Heidelberg, 1996, pp. 253–267. doi: 10.1007/3-540-61053-7_66.
Referenced page 36

[33] P. Crégut, "An abstract machine for lambda-terms normalization," in *Proceedings of the 1990 ACM conference on LISP and functional programming*, 1990, pp. 333–340.
Referenced pages 137 and 214

[34] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991, doi: 10.1145/115372.115320.
Referenced pages 120, 121 and 123

[35] N. G. De Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," in *Indagationes mathe-*

maticae (proceedings), Elsevier, 1972, pp. 381–392.
Referenced pages 127, 135, 137 and 137

[36] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975, doi: 10.1145/360933.360975.
Referenced page 77

[37] X. Denis, J.-H. Jourdan, and C. Marché, "Creusot: a foundry for the deductive verification of Rust programs," in *Formal Methods and Software Engineering*, vol. 13478, Cham: Springer International Publishing, 2022, pp. 90–105. doi: 10.1007/978-3-031-17244-1_6.
Referenced pages 23, 27, 39, 57 and 236

[38] O. Ebresafe, I. Zhao, E. Jin, A. Bright, C. Jian, and Y. Zhang, "Certified compilers à la carte," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 372–395, Jun. 2025, doi: 10.1145/3729261.
Referenced pages 135 and 237

[39] R. W. Floyd, "Nondeterministic algorithms," *Journal of the ACM*, vol. 14, no. 4, pp. 636–644, Oct. 1967, doi: 10.1145/321420.321422.
Referenced page 50

[40] Y. Forster, M. Sozeau, and N. Tabareau, "Verified extraction from Coq to OCaml," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 52–75, Jun. 2024, doi: 10.1145/3656379.
Referenced pages 127 and 127

[41] L. Gäher *et al.*, "Simuliris: a separation logic framework for verifying concurrent program optimizations," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–31, Jan. 2022, doi: 10.1145/3498689.
Referenced page 35

[42] B. Gaudin, B. Bonneau, S. Boulmé, and M.-L. Potet, "Formally verified compilers against hardware attacks," Sep. 2026. [Online]. Available: https://hal.science/hal-05748290 submitted at CPP27.
Referenced page 236

[43] J. E. Ghalayini and N. Krishnaswami, "The denotational semantics of SSA." Accessed: Nov. 14, 2025. [Online]. Available: http://arxiv.org/abs/2411.09347
Referenced page 134

[44] R. Ghiya, D. Lavery, and D. Sehr, "On the importance of points-to analysis and other memory disambiguation methods for C programs," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, Snowbird Utah USA: ACM, May 2001, pp. 47–58. doi: 10.1145/378795.378806.
Referenced page 35

[45] L. Gourdin, B. Bonneau, S. Boulmé, D. Monniaux, and A. Bérard, "Formally verifying optimizations with block simulations," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 59–88, Oct. 2023, doi: 10.1145/3622799.
Referenced pages 25, 25, 35, 37, 38 and 59

[46] H. Herbelin *et al.*, "Rocq 9.2.0." [Online]. Available: https://doi.org/10.5281/zenodo.19256047
Referenced pages 23, 27 and 57

[47] S. Ho and J. Protzenko, "Aeneas: Rust verification by functional translation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. ICFP, pp. 711–741, Aug. 2022, doi: 10.1145/3547647.
Referenced pages 23, 27, 39, 57 and 236

[48] W. Hwu *et al.*, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1625–1640, Dec. 1995, doi: 10.1109/5.476079.
Referenced page 36

[49] R. Jung *et al.*, "Iris: monoids and invariants as an orthogonal basis for concurrent reasoning," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Mumbai India: ACM, Jan. 2015, pp. 637–650. doi: 10.1145/2676726.2676980.
Referenced pages 39 and 235

[50] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: an aliasing model for Rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020, doi: 10.1145/3371109.
Referenced pages 8, 26 and 35

[51] R. Jung, "An intermediate language to formally justify memory access reordering," Bachelor's Thesis, 2013. [Online]. Available: https://research.ralfj.de/bachelor/bachelor.pdf
Referenced page 36

[52] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis, "A formal C memory model supporting integer-pointer casts," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland OR USA: ACM, Jun. 2015, pp. 326–335. doi: 10.1145/2737924.2738005.
Referenced pages 33, 34 and 35

[53] R. Kawakami, J. Garrigue, T. Saikawa, and R. Affeldt, "Monadic equational reasoning for while loop in Rocq," University of Strathclyde, Glasgow, Scotland, Jun. 2025. [Online]. Available: https://msp.cis.strath.ac.uk/types2025/
Referenced page 95

[54] Y. Kiam Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified CakeML compiler backend," *Journal of Functional Programming*, vol. 29, p. e2, 2019, doi: 10.1017/S0956796818000229.
Referenced pages 38, 38, 94 and 135

[55] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*, Boston, Massachusetts: ACM Press, 1973, pp. 194–206. doi: 10.1145/512927.512945.
Referenced page 141

[56] T. Klappe, "An operational semantics for the C99 restrict type qualifier," Doctoral dissertation, 2024.
Referenced page 35

[57] R. Krebbers, "Aliasing restrictions of C11 formalized in Coq," in *Certified Programs and Proofs*, vol. 8307, Springer International Publishing, 2013, pp. 50–65. doi: 10.1007/978-3-319-03545-1_4.
Referenced page 35

[58] R. J. Krebbers, "The C standard formalized in Coq," PhD Thesis, 2015. [Online]. Available: https://robbertkrebbers.nl/thesis.html
Referenced page 35

[59] J. Koenig and Z. Shao, "CompCertO: compiling certified open C components," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Virtual Canada: ACM, Jun. 2021, pp. 1095–1109. doi: 10.1145/3453483.3454097.
Referenced pages 37 and 96

[60] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego California USA: ACM, Jan. 2014, pp. 179–191. doi: 10.1145/2535838.2535841.
Referenced pages 38, 94 and 94

[61] C. Lattner *et al.*, "MLIR: scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14. doi: 10.1109/CGO51591.2021.9370308.
Referenced pages 121, 135 and 237

[62] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *Journal of Automated Reasoning*, vol. 41, no. 1, pp. 1–31, 2008, doi: 10.1007/s10817-008-9099-0.
Referenced pages 33, 34, 34 and 191

[63] J.-C. Léchenet, "Mechanized dominator tree certification," in *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Rennes France: ACM, Jan. 2026, pp. 143–156. doi: 10.1145/3779031.3779107.
Referenced page 154

[64] J. Lee *et al.*, "Taming undefined behavior in LLVM," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelona Spain: ACM, Jun. 2017, pp. 633–647. doi: 10.1145/3062341.3062343.
Referenced pages 8, 24, 26, 32, 32 and 32

[65] J. Lee, C.-K. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes, "Reconciling high-level optimizations and low-level code in LLVM," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, Oct. 2018, doi: 10.1145/3276495.
Referenced pages 33, 34 and 34

[66] K. R. M. Leino, "This is Boogie 2," Jun. 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
Referenced pages 43, 50 and 77

[67] K. R. M. Leino, "Dafny: an automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, vol. 6355, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. doi: 10.1007/978-3-642-17511-4_20.
Referenced page 39

[68] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart, "The CompCert memory model, version 2," Research Report RR-7987, Jun. 2012. [Online]. Available: https://hal.inria.fr/hal-00703441
Referenced pages 34, 34, 162 and 191

[69] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, Dec. 2009, doi: 10.1007/s10817-009-9155-4.
Referenced pages 14, 33, 34, 37, 41, 60, 62, 63, 63, 94, 94, 94, 94, 94, 94, 96, 97, 104, 105, 106, 114, 116, 117, 117, 118, 135, 135, 141 and 162

[70] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*, San Francisco, California, United States: ACM Press, 1995, pp. 333–343. doi: 10.1145/199448.199528.
Referenced pages 91 and 96

[71] LLVM Project, "LLVM 22.1.0." [Online]. Available: https://releases.llvm.org/22.1.0/docs/index.html
Referenced pages 35, 38 and 212

[72] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121–141, Jan. 1979, doi: 10.1145/357062.357071.
Referenced pages 151, 153 and 153

[73] N. Lynch and F. Vaandrager, "Forward and backward simulations," *Information and Computation*, vol. 121, no. 2, pp. 214–233, 1995.
Referenced page 118

[74] C. McBride, "Turing-completeness totally free," in *Mathematics of Program Construction*, vol. 9129, Cham: Springer International Publishing, 2015, pp. 257–275. doi: 10.1007/978-3-319-19797-5_13.
Referenced page 68

[75] G. Melquiond and J. Moreau, "A safe low-level language for computer algebra and its formally verified compiler," *Proceedings of the ACM on Programming Languages*, vol. 8, no. ICFP, pp. 121–146, Aug. 2024, doi: 10.1145/3674629.
Referenced pages 38, 57, 235 and 236

[76] E. Moggi, *An abstract view of programming languages*. University of Edinburgh. Laboratory for foundation of computer science, 1990.
Referenced pages 95 and 96

[77] Y. Matsushita, T. Tsukada, and N. Kobayashi, "RustHorn: CHC-based verification for Rust programs," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 4, pp. 1–54, Dec. 2021, doi: 10.1145/3462205.
Referenced page 39

[78] L. de Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37.
Referenced page 57

[79] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, "Verified peephole optimizations for CompCert," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 448–461, Aug. 2016, doi: 10.1145/2980983.2908109.
Referenced pages 33 and 37

[80] S. Michelland, Y. Zakowski, and L. Gonnord, "Abstract interpreters: a monadic approach to modular verification," *Proceedings of the ACM on Programming Languages*, vol. 8, no. ICFP, pp. 602–629, Aug. 2024, doi: 10.1145/3674646.
Referenced page 96

[81] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '97. New York, NY, USA: Association for Computing Machinery, 1997, pp. 106–119. doi: 10.1145/263699.263712.
Referenced page 56

[82] S. Novack, J. Hummel, and A. Nicolau, "A simple mechanism for improving the accuracy and efficiency of instruction-level disambiguation," in *Languages and Compilers for Parallel*

Computing, Springer Berlin Heidelberg, 1996, pp. 289–303.
Referenced page 35

[83] B. Pesin, "Verified compilation of a synchronous dataflow language with state machines," Theses, 2023. [Online]. Available: https://theses.hal.science/tel-05144899
Referenced page 206

[84] J. Protzenko *et al.*, "Verified low-level programming embedded in F*," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–29, Aug. 2017, doi: 10.1145/3110261.
Referenced page 41

[85] F. Rastello and F. Bouchez Tichadou, Eds., *SSA-based compiler design*. Cham: Springer International Publishing, 2022. doi: 10.1007/978-3-030-80515-9.
Referenced pages 120, 121, 121, 140, 144, 145, 145, 216 and 217

[86] J. Regehr, "A guide to undefined behavior in C and C++, part 3." Accessed: Mar. 18, 2026. [Online]. Available: https://blog.regehr.org/archives/232
Referenced pages 96 and 97

[87] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark: IEEE Comput. Soc, 2002, pp. 55–74. doi: 10.1109/LICS.2002.1029817.
Referenced pages 39 and 45

[88] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the ACM annual conference on - ACM '72*, Boston, Massachusetts, United States: ACM Press, 1972, pp. 717–740. doi: 10.1145/800194.805852.
Referenced pages 111 and 113

[89] V. Robert and X. Leroy, "A formally-verified alias analysis," in *Certified Programs and Proofs*, vol. 7679, Springer Berlin Heidelberg, 2012, pp. 11–26. doi: 10.1007/978-3-642-35308-6_5.
Referenced pages 7 and 25

[90] A. Saito and R. Affeldt, "Towards a practical library for monadic equational reasoning in Coq," in *Mathematics of Program Construction*, vol. 13544, Cham: Springer International Publishing, 2022, pp. 151–177. doi: 10.1007/978-3-031-16912-0_6.
Referenced page 95

[91] S. Schneider, "A verified compiler for a linear imperative / functional intermediate language," Doctoral dissertation, 2018. doi: 10.22028/D291-27566.
Referenced pages 38, 134 and 156

[92] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: a verified compiler for relaxed-memory concurrency," *Journal of the ACM*, vol. 60, no. 3, pp. 1–50, Jun. 2013, doi: 10.1145/2487241.2487248.
Referenced pages 37 and 96

[93] L. Silver, E. Westbrook, M. Yacavone, and R. Scott, "Interaction tree specifications: a framework for specifying recursive, effectful computations that supports auto-active verification," *LIPIcs, Volume 263, ECOOP 2023*, vol. 263, p. 30:1–30:26, 2023, doi: 10.4230/LIPICS.ECOOP.2023.30.
Referenced page 96

[94] D. Sangiorgi and R. Milner, "The problem of "weak bisimulation up to," in *CONCUR '92*, vol. 630, W. Cleaveland, Ed., Springer Berlin Heidelberg, 1992, pp. 32–46. doi: 10.1007/BFb0084781.
Referenced page 119

[95] K. Stark, S. Schäfer, and J. Kaiser, "Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions," in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Cascais Portugal: ACM, Jan. 2019, pp. 166–180. doi: 10.1145/3293880.3294101.
Referenced page 135

[96] B. Steensgaard, "Sparse functional stores for imperative programs," in *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, San Francisco California USA: ACM, Mar. 1995, pp. 62–70. doi: 10.1145/202529.202536.
Referenced pages 36, 44, 47 and 50

[97] N. Swamy *et al.*, "Dependent types and multi-monadic effects in F*," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg FL USA: ACM, Jan. 2016, pp. 256–270. doi: 10.1145/2837614.2837655.
Referenced page 57

[98] R. E. Tarjan, "Applications of path compression on balanced trees," *Journal of the ACM*, vol. 26, no. 4, pp. 690–715, Oct. 1979, doi: 10.1145/322154.322161.
Referenced page 153

[99] N. Villani, J. Hostert, D. Dreyer, and R. Jung, "Tree borrows," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1019–1042, Jun. 2025, doi: 10.1145/3735592.
Referenced pages 26 and 35

[100] P. Wadler, "Linear types can change the world!," in *Programming concepts and methods*, Citeseer, 1990, p. 5.
Referenced pages 39, 45, 49, 57 and 197

[101] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?," in *Proceedings of the Asia-Pacific Workshop on Systems*, Seoul Republic of Korea: ACM, Jul. 2012, pp. 1–7. doi: 10.1145/2349896.2349905.
Referenced page 26

[102] L.-yao Xia *et al.*, "Interaction trees: representing recursive and impure programs in Coq," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020, doi: 10.1145/3371119.
Referenced pages 13, 27, 38, 59, 70, 70, 86, 95, 95, 95, 95, 111, 119 and 236

[103] I. Yoon, Y. Zakowski, and S. Zdancewic, "Formal reasoning about layered monadic interpreters," *Proceedings of the ACM on Programming Languages*, vol. 6, no. ICFP, pp. 254–282, Aug. 2022, doi: 10.1145/3547630.
Referenced pages 81, 91, 95, 95, 96, 96 and 97

[104] Y. Zakowski, P. He, C.-K. Hur, and S. Zdancewic, "An equational theory for weak bisimulation via generalized parameterized coinduction," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, New Orleans LA USA: ACM, Jan. 2020, pp. 71–84. doi: 10.1145/3372885.3373813.
Referenced page 119

[105] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic, "Modular, compositional, and executable formal semantics for LLVM IR," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021, doi: 10.1145/3473572.
Referenced pages 32, 38, 70, 82, 95, 95, 95, 96, 97, 97, 97, 135 and 238

[106] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM intermediate representation for verified program transformations," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 427–440, Jan. 2012, doi: 10.1145/2103621.2103709.
Referenced page 38

[107] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formal verification of SSA-based optimizations for LLVM," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 175–186, Jun. 2013, doi: 10.1145/2499370.2462164.
Referenced pages 38 and 134

[108] J. Zhao and S. Zdancewic, "Mechanized verification of computing dominators for formalizing compilers," in *Certified Programs and Proofs*, vol. 7679, Springer Berlin Heidelberg, 2012, pp. 27–42. doi: 10.1007/978-3-642-35308-6_6.
Referenced pages 134 and 153