LSR-IMAG research report

Specifying and reasoning in Coq with high-order impure programs using specifications a la Dijkstra and refinement.

Very preliminary DRAFT

Sylvain.Boulme@imag.fr
(LSR-IMAG)

June 19, 2006

Introduction

In this report, I propose a general method to specify and prove in Coq programs which marry non-purely-functional features (e.g. state and exception handling) with functional features (e.g. high-order functions, inductive data-types handling). If simple imperative programs can naturally be proved in usual Hoare logics [Hoa69], generic programs or programs handling complex data-structures are difficult to prove in such logics. Indeed, Hoare logics generally do not allow to express generic imperative programs such procedures parametrized by procedures with side-effects. And, usually, they do not allow to handle complex data-structures either: they often discourage the use of aliases and consequently the pointer manipulations that are used in standard imperative languages to express such data-structures. However, recently, some perspectives have been opened on this last point [OYR04, Rey02].

On the other hand, purely functional programming is well supported by theorem provers. It allows to handle functions as first class values. And theorem provers like Coq provide inductive data-types, allowing to express easily complex non-mutable data-structures like abstract syntax trees. Algorithms based on structural recursion can be proved in a very natural way.

I show here that refinement calculus is simple and elegant way to embed reasoning about non-purely-functional programs into a functional logic. Given a specific programming language, the corresponding refinement theory can be seen as an extension of the corresponding Hoare logic, and also as an extension of equational reasoning in the corresponding monad. Monads [JW93] provide a simple and common interface to embed *non-functional computations* into functional languages. Refinement theories extends monads with the following ideas:

- They express specifications a la Dijkstra: specification language is an extension of the programming language with non-computational constructs. In other words, refinement theories are monads where "*computations*" are specifications. Intuitively, such specifications look like "abstract" and "non-deterministic" programs, which describe a "set" of "effective" programs: the set of their implementations. They are a generalization of Dijkstra guarded commands [Dij75].

- Refinement is an (extensional and partial) order relation between specifications. Refinement order expresses that a specification simulates an other one. Hence, C refines A means that *every implementation of C is an implementation of A*. In particular, if C is itself an effective program, then it means that C is an implementation of A.

Hence, refinement theories are particular ordered monads where "*computations*" express non-purely-functional and non-fully-effective algorithms.

Related works and main contributions

Specification language in refinement calculus is inspired by Dijkstra guarded command language [Dij75]. The semantics of this language using weakest-preconditions has been invented by Dijkstra, who has also invented weakest-preconditions [Dij75, Dij76]. The concept of *stepwise refinement* (or top-down program development) have been also invented by Dijkstra [Dij69], and has been promoted by Wirth [Wir71].

Back has formalized this method of program development in a *refinement calculus* [Bac78]. In particular, he has introduced the refinement relation and its fundamental properties. Then, he extended the theory to various programming paradigms with many others authors [Mor90, Mor87, BvW98]. In particular, Abrial in [Abr96] extends the approach by providing a notion of module, called *machine*, and thus allowing to use this approach in real developments of software.

Gordon [Gor88] proposed to use high order logics to formalize particular programming logics such Hoare logic. This idea has been used in many works: among them, [Har98] formalizes weakest-precondition in HOL, and [BFM99] formalizes B in Coq and PVS. Actually, [vW94] inspired my first formalizations of refinement calculus in Coq. This work has also inspired the book [BvW98].

Filliâtre [Fil99b, Fil03a] has built a tool [Fil03b] to generate proof obligations of a Hoare logic in Coq. The programming language of this Hoare logic is a subset of ML. Hence, his works shows how to extend Hoare logic with high-order functions. The correctness of his system is based on a interpretation of non-functional features using monads [JW93]. Actually, in order to avoid the drawbacks of having a global state (as in state-handling monads), he introduces a generalization of monads using a type-system with effects [TJ92, Wad99]. His type system allows also [Fil99a] to reject programs with aliases. However, as I explain in chapter 2, if Hoare logic is not incompatible with high-order functions, it does not allow to express satisfying specifications of high-order functions with side-effects. From this point of view, refinement is strictly more powerful than Hoare logics.

The main contributions of my work with respect to all these works seem to be the following:

- Adapt refinement calculus to functional programming. These adaptations use monads theory. This allows to extend refinement calculus with high-order functions and structural recursion.
- Show that refinement calculus could be built upon an intuitionistic logic.
- Interpret weakest-preconditions as a CPS (continuation passing semantics), and emulate wp-calculi in calculus of construction. [1]

In my approach, the notion of "predicate transformers" invented by Dijkstra (a denotational semantics for wp-calculus) is derived from a general semantics: it corresponds to the instantiation of a CPS for state-transformers.

Informally, if $K(A)$ is the type of computations returning a result of type A. The state-transformers corresponding to this notion of computations are of type $S \to K(S)$ where S is the type of the global state. State-transformers are indeed functions from an initial state to a final state.

Computations based on continuations are such $K(A) = (A \to R) \to R$ where R is the type in which values are *observed*. A function of type $A \to R$ is called a continuation. It represents the *context* in which the returned value of type `A` is observed. Hence in this approach, a computation is a function which is parametrized by the context observing its result.

Thus, state-transformers based on continuations are of type $S \to ((S \to R) \to R)$. This type is isomorphic to $(S \to R) \to (S \to R)$. Functions of this type represent here the transformation of a context observing the final state into a context observing the initial state.

In the case of wp-calculus, we observe the state in logical propositions. Thus, R corresponds to *Bool*[2]. We thus recover the type of predicate transformers (see [BvW98]):

$$(S \to Bool) \to (S \to Bool)$$

[1] Of course, seeing weakest-preconditions as a CPS has been already used in other works, see [AZ99].

[2] In this informal explanation *Bool* represents the type of logical formulas.

Such a function represents the transformation of a logical condition on the final state (a postcondition) into a condition on the initial state (a precondition).

Hence, weakest-preconditions on state transformers are a special case of a more general notion of weakest-preconditions. This general notion of weakest-preconditions is simply a CPS semantics where values are observed into logical propositions.

This simple remark gives a general way to build a refinement theory suited for each programming paradigm (exception-handling, concurrency, ...): it suffices to instantiate the corresponding monad transformer on this CPS semantics.

Contents of this report

This report is organized as follow. Chapter 1 introduces the philosophy of my work: why I use Coq in order to express a high-order refinement calculus. It is also a brief presentation of this language dedicated to readers not familiar with it. Chapter 2 is an introduction to my work. It motivates the use of refinement with respect to Hoare logics in order to specify and prove high-order functions with side-effects. It also presents the main ideas of the report.

Chapter 3 studies recursion operators. If chapter 2 shows that structural recursion is get for free (it is a consequence of the monadic structure of refinement theories), other kinds of recursion need specific operators. First, chapter 3 introduces smallest and biggest fixpoints *a la* Tarski. Then, it also introduces a well-founded recursion operator, and presents the proof in Coq that a particular imperative refinement theory allows to ensure the total correctness for the corresponding imperative programming language. It thus illustrates on this example how refinement theories with well-founded recursion can be related to total correctness of programs.

Chapter 4 presents the prototype of a prover for high-order refinement calculus in Coq. This prover is mainly a Coq library with some tacticals. It also proposes a type system inspired from [Fil99a] in order to build a future concrete syntax to denotations written in Coq.

At last chapter 5 present some future works.

Acknowledgment

This work is part of Geccoo project[3].

[3] `http://geccoo.lri.fr/`

Contents

Chapter 1

Coq: a “universal program prover”

This chapter explains a fundamental aspect of my approach: the use of Coq proof assistant as a *“universal prover”* for proving programs. I mean here that Coq is the formal meta-language I use to express the semantics and the properties of Dijkstra specification languages, but also the kernel of a future tool, where the user will finally express her proof of refinement. Hence, the word *“ universal”* is an abusive reference to Turing *“universal machines”*, even if we know that, in theory, universal provers can not exist ! This chapter also introduces the main Coq aspects used in the paper. The reader is assumed to be familiar with functional programming and ML in particular.

This work is part of a long-term project: to provide an environment for developing software proved by construction. Defining and implementing such an environment is a hard and long task. Especially, providing a safe and practical environment, with theorems and programs libraries, with tools for proof development such decision procedures, etc. A traditional way in computer technology to break down this kind of complexity, is to split systems into levels, using interpretation or compilation to express a given level into a lower one.

In this work, I consider Coq as a target platform where express specifications, programs and proofs. Hence, my approach is to build a refinement calculus into Coq, such that the deductive rules of the calculus are themselves proved in Coq. This approach looks both very safe and quite flexible: extending the calculus with a new deductive rule is proving a new theorem. Coq seems appropriate to this purpose because:

- Coq is robust. The coherence of the whole Coq system (including user-defined tactics) itself relies only on the correctness of a small kernel, the type-checker. This kernel has itself an implementation proved in Coq. [Bar99].
- Coq benefits from an important development effort: it provides some powerful decisions procedures, syntactic extensions, simple tactic definitions (via a dedicated language), many libraries, etc.
- Coq is a (purely functional) programming language where formulas and proofs are first class values. Hence, weakest-precondition calculi can be programmed in Coq, in a very simple and natural way.
- Coq is a high-order logic: it seems necessary in order to express a high-order refinement calculus.

Below, section 1.1 introduces a necessary feature for proving programs: the marriage of deductions and computations. Section 1.2 introduces (briefly) the core logic of Coq: the Calculus of Inductive Constructions (Cic). Indeed, Coq is based on the Calculus of Constructions [CH88], enriched with inductive [PM93] and co-inductive [Gim94] type definitions. Introduction of section 1.2

is limited to the notions necessary to understand my refinement calculus and it is rather informal. In particular, I omit here extraction, co-induction, local bindings, etc. See [BC04, Coq04] to have a more detailed presentation. At last, section 1.3 presents some practical aspects of Coq.

1.1 Using partial evaluation in proofs of programs

A great strength of Coq lies in its ability to marry deductions and computations in a very powerful way. In particular, Coq is able to evaluate partially functional programs during proofs: Coq is actually a functional programming interpreter. That often simplifies proofs, which would otherwise require to explicit the computations as deduction steps. For instance, Peano's naturals being defined inductively from the constructors `O` (zero) and `S` (successor), the addition on naturals is defined as the following recursive function:

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
    | O => m
    | (S p) => (S (plus p m))
  end.
```

It is easy to prove associativity of `plus`:

```
  x : nat
  y : nat
  z : nat
  ============================
   (plus (plus x y) z)=(plus x (plus y z))
```

Applying induction on `x`, we are led to prove two subgoals:

```
subgoal 1 is:
  y : nat
  z : nat
  ============================
   (plus (plus (O) y) z)=(plus (O) (plus y z))

subgoal 2 is:
  x : nat
  y : nat
  z : nat
  IHx : (plus (plus x y) z)=(plus x (plus y z))
  ============================
   (plus (plus (S x) y) z)=(plus (S x) (plus y z))
```

Commanding Coq to simplify the two goals using the computational definition of `plus`, we have to prove:

```
subgoal 1 is:
  y : nat
  z : nat
  ============================
   (plus y z)=(plus y z)

subgoal 2 is:
  x : nat
  y : nat
```

```
z : nat
IHx : (plus (plus x y) z)=(plus x (plus y z))
============================
 (S (plus (plus x y) z))=(S (plus x (plus y z)))
```

The first is trivially provable using reflexivity of equality. And the second is also trivially provable, by rewriting the hypothesis `IHx` into the goal, and then applying reflexivity.

This example looks quite simple, and the reader may wonder what is interesting here. Actually, marrying computation and deduction in a coherent and expressive logic is not obvious (several inconsistent systems have been proposed). For instance, the proof of the logical coherence of COQ relies on some strong properties of the calculus:

- evaluation is compatible with typing.
- the result of a computation does not depend on the order of evaluation in subexpressions.
- there is no infinite computations.

Partial (and symbolic) evaluation of programs during proofs seems the minimal requirement on an interactive program prover. Simulating non-functional programs into functional ones, we will use COQ to simplify non-functional programs. Of course, such a simulation is not very efficient. But in practice, during proofs, partial evaluation needs not to be very efficient. Usually, needed simplifications correspond to a few steps of computations. This comes from the fact, that "good" proofs mix computation steps with deductive steps where complex data and complex computations are specified abstractly.

1.2 Calculus of Constructions with inductive types (CIC)

Calculus of Constructions is a typed lambda-calculus, with *types* as first-class values (i.e. types are terms). Hence, *terms* may be:

- *identifiers*, defined constants or declared variables in the context.
- the *application* of a functional term `M` to a term `N`, denoted by "`M N`".
- the *abstraction* of a variable `x` of type `A` in a term `b`, denoted by "**fun** `(x:A) => b`".
- the *product* of a type family `B` indexed by a variable `x` of type `A` (i.e. the type of functions "**fun** `(x:A) => b`" where `b` is of type `B`, assuming `x` of type `A`). It is generally denoted by "**forall** `(x:A), B`" or simply "`A->B`" if `x` does not occur free in `B`.

In the spirit of Curry-Howard isomorphism, types may represent programming data-types or logical propositions. This is expressed in CIC using two special types of types, **Set** as type of data-types, and **Prop** as type of propositions. Hence, **forall** `(x:A), B` may also be interpreted as a universal quantification (as suggested by the keyword **forall**), and `->` as the logical implication.

1.2.1 Curry-Howard isomorphism

More generally, we can give the following intuitive interpretation of proposition types:

Proposition	**Type of**
`A -> B`	a function expecting a proof of `A` and returning a proof of `B`
forall `(x:T), (P x)`	a function expecting a term `x` and returning a proof of `(P x)`
`True`	`I`, the only proof of `True`.
`False`	nothing. There is no proof of `False`.
`A /\ B`	a pair containing a proof of `A` and a proof of `B`
`A \/ B`	an element of a union-type, containing either of proof of `A` or a proof of `B`
exists `x:T, (P x)`	a pair containing a term `x` and a proof of `(P x)`

Here, constructions `True`, `False`, `/\`, `\/` and **exists** are defined as inductive types (they are not primitive constructions). The negation of a formula, noted `~A`, is actually the type `A->False`. Hence a proof of `~A` is a function expecting a proof of `A` and returning a proof of `False`.[1]

Actually, CIC is an intuitionistic logic. It means that the proof of a disjunction contains an algorithm computing which side of the disjunction holds. Hence, a formula **forall** `x:A, (P x)\/~(P x)` means that the predicate `P: A ->` **Prop** is decidable. In the same way, the proof of an existential property provides a witness of the property: proving a formula **forall** `x:A,` **exists** `y:B, (P x y)` is like building a function `f: A -> B` and proving **forall** `x:A, (P x (f x))`. In particular, assuming that `nat` is the type of natural numbers, the following property is not provable:
forall `f: (nat -> nat),` **exists** `x:nat,` **forall** `y:nat, (f x) <= (f y).`
Indeed, the proof of this property would give an algorithm to find the minimal element of any function `f` of type `nat -> nat`.

If CIC type can be interpreted as logical propositions, they can also be interpreted as data-types. In the two next subsections, I detail the interpretation of product types as data-types, because such a construction is not very frequent in programming languages : a type of the form **forall** `(x:T), (P x)` can be interpreted as a *polymorphic type* if `T` is the type of a type such **Set** (section 1.2.2), or as a *dependent type* if `T` is a data-type such `nat` (section 1.2.3).

1.2.2 Polymorphic types and hierarchy of universes

Let me consider `(list A)` the type of lists of `A` (where `A` is a type). The term `list` is a type constructor of type `Set->Set`, where the parameter represents the type of the elements of the list. Hence `(list nat)` and `(list bool)` are respectively the type of lists of natural numbers and the type of lists of booleans. The type of the concatenation of two lists is the following polymorphic type:

```
app: forall A : Set, list A -> list A -> list A
```

In CIC, as types are terms, they have also a type. Types are ordered in a hierarchy of universes. This mechanism avoids Girard's paradox coming from the rule **Type:Type**. [Coq86]

The type of **Set** and **Prop** is Type_0 and then for each natural i, the type of Type_i is Type_{i+1}. A subtyping relation allows to coerce each Type_i into a Type_j where $i \leq j$.

Actually the user never writes index i. Implicit universe constraints are generated by the type-checker. When a constraint is violated, a type error is raised.

1.2.3 Dependent types and termination

Let me consider `(listN n)` the type of lists of relative integers with a fixed length `n` (where `n` is a natural number). The term `listN` is a type constructor of type `nat->Set`, where the parameter

[1] Such a proof means that a proof of `A` can not be constructed, since, without hypothesis, a proof of `False` can not be constructed.

represents the length of the lists. Hence `(listN 0)` is the type of the empty list, and `(listN 1)` is the type of lists with a single element. The type of the concatenation of two lists is the following dependent type:

```
appN: forall n m : nat, (listN n) -> (listN m) -> (listN (n + m))
```

In CIC, functions are total, but dependent types allow to express partiality, or more precisely constraints on parameters. For instance, the function to access the first element of a list is only defined on a non-empty list. Thus, we can give it the following type:

```
headN: forall n:nat, listN (1+n) -> Z.
```

Alternatively, in order to express this constraint, we can add an additional parameter: the proof that the length is not zero:

```
headN': forall n:nat, n<>0 -> (listN n) -> Z.
```

This last form is generally the most expressive: constraints can be expressed by any relation on parameters. However, in the first form, the proof that the parameter satisfies the constraint is discharged by the type-checker, whereas in the second form, the user must provide the proof. Hence, if `l1` is of type `listN 1`, the term `(headN 0 l1)` can be type-checked, whereas in the second form, the user must write `(headN' 1 p l1)` where `p` is a proof that `1<>0`.

Let us remark that the verification of the term `(headN 0 l1)` needs the ability to evaluate the expression `(listN (1+0))` in `(listN 1)` during the type-checking. During type-checking, two expressions are considered equals if they lead to the same irreducible term (modulo renaming of bound variables). As types may be arbitrary expressions, CIC requires that evaluation of arbitrary expressions terminate in order to guarantee termination of type-checking. This internal equality of the type-checking, which is thus decidable, is called inter-convertibility. It must not be confused with Leibniz's equality, which may appear in user's propositions, such in example page 8.

More fundamentally, the termination of evaluation in CIC guarantees the consistency of CIC: indeed, there is no proof of `False`, because, syntactically, there is no irreducible proof of `False`. On the contrary, if we assume a generic recursion operator as in ML, we can build a proof of `False`. A such recursion operator `Y` has type **forall** `(A,B:`**Type**`), ((A -> B) -> (A -> B)) -> A -> B`. Intuitively, given two types `A` and `B`, `F` of type `(A -> B) -> (A -> B)` and `x` of type `A`, then `(Y A B F x)` is computed into `(F (Y A B F) x)`. Let us call `Absurd` the term
`Y True False (`**fun** `f:(True->False) => f) I`
We see that `Absurd` is an infinite loop, which computes a value of type `False`. In the Curry-Howard isomorphism, there is thus a connexion between some non-ending computations and proofs of `False`.

1.2.4 Tiny examples in COQ

In this section, I illustrate some of the previous concepts: proofs as terms and computations in types.

1.2.4.1 Proofs as terms

In the standard library, `le` the usual order relation on naturals is defined by an inductive definition. In the following examples, we only need to know that `le` of type `nat -> nat ->` **Prop** satisfies the two properties `le_n` (reflexivity) and `le_S`.

```
Inductive le: nat -> nat -> Prop :=
| le_n: forall (n:nat), (le n n)
| le_S: forall (n m:nat), (le n m) -> (le n (S m)).
```

The proof that for all natural n, n is lower than $n + 2$ is given by a function `le_SSn` expecting a natural `n` and building a proof of `(le n (S (S n)))`, using `le_n` and `le_S`. The type of this function corresponds to announced property:

```
Definition le_SSn: forall (n:nat), (le n (S (S n)))
  := fun n:nat => (le_S n (S n) (le_S n n (le_n n))).
```

Of course, building such a proof at hand is very tedious, and almost impossible for large proofs. Coq provide a tactical language with predefined tactics to build such a proof. I use here resolution *a la* Prolog. First, I construct a basis of lemmas containing `le_n` and `le_S`.

```
Hint Resolve le_n le_S: my_lemma_basis.
```

Then, I reprove this lemma, by using the tactic `auto` (resolution) on this basis.

```
Lemma le_SSn': forall (n:nat), (le n (S (S n))).
  auto with my_lemma_basis.
Qed.
```

Ultimately, each term produced by these tactics is checked (and stored) by the type-checker. This takes place above in the **Qed** keyword. It may happen that tactics fail to construct a proof, or even build an incorrect proof (this may come from a bug of the tactics involved, or a bad use of them). This never breaks the consistency of the system, because of the separation between the proof construction and the proof checking.

Let me now try to prove $5 \leq 10$. I use the same tactic. Below, the numerals 5 and 10 are Coq macros to represent the corresponding Cic terms written from `O` and `S`. See section 1.3.

```
Lemma le_5_10_ex1: (le 5 10).
  auto 20 with my_lemma_basis.
Qed.
```

The proof built by this tactics is the following:

```
le_5_10_ex1 =
le_S 5 9 (le_S 5 8 (le_S 5 7 (le_S 5 6 (le_S 5 5 (le_n 5)))))
     : le 5 10
```

Thus, resolution succeeds in building this proof. But this is not the most efficient way when the difference between the two number is big. In particular, the produced proof may become quite large.

1.2.4.2 Replacing deduction steps by computations steps

For this example, there is a simple technique which both allows to provide an automatic way to build such a proof, and to reduce drastically the size of the proof. I present this technique in this mini-tutorial as an illustration of the ability to compute propositions (or types). Actually, in my refinement calculus, I will use the same principle to simplify proofs using a weakest-precondition calculus. Here as `le` is a decidable predicate (on closed terms), we can reduce all proof of `le` to the proof of `True`. And the proof of `True` is `I` (thus, it is very short !). First, I give an alternate definition to `le`. This definition `le'` constructs a proposition by induction on the left number. This proposition is either the proposition `True` or either the proposition `False` (which has no proof).

```
Fixpoint le' (n m:nat) {struct n}: Prop :=
  match n, m with
  | 0, _ => True
  | (S n0),(S m0) => (le' n0 m0)
  | (S _),0 => False
  end.
```

Hence, `I` is a proof of `(le' 5 10)`. Now, I show that `le'` is included in `le`. This lemma proceeds by induction on `n` (I do not give the proof here). The converse lemma is also provable, but we will not use here.

```
Lemma le'_le: forall (n m:nat), (le' n m)->(le n m).
```

Now, a proof of `(le 5 10)` can be given by `(le'_le 5 10 I)`. More generally given `n` and `m` two *closed* naturals, either `(le'_le n m I)` type-checks and its a proof of `(le n m)`, or either it does not type-check, and we know (at a meta-level) than `n` is strictly greater than `m`.

```
Definition le_5_10: (le 5 10) := (le'_le 5 10 I).
```

Of course, if we *compute* this proof by unfolding the proof of `le'_le` and expand recursive computations of this proof, we will recover a proof like `le_5_10_ex1`. But, in practice, this computation may never be necessary. COQ provides a mechanism to control the unfolding of definitions and to prevent undesired unfolding.

In practice, we want to use `le'` for simplification in interactive proofs. For instance, having a goal

```
============================
 (le 5 10)
```

We use the tactic `apply le'_le`. This tactics finds an instantiation of `le'_le` such its returned type matches the goal, and discharges the goal, making subgoals from hypotheses of `le'_le`. Here, we have the subgoal

```
============================
 le' 5 10
```

By simplification (tactic `simpl`), this reduces to

```
============================
 True
```

which is trivially provable. Conversely, assuming that we have to prove the goal `(le 10 5)`. Using the same sequence of tactics (`apply le'_le; simpl`), we would find the goal `False`.

This example illustrates, in a very simplified form, a general paradigm to design decision or simplification procedures for COQ prover. In this paradigm, called *reflection*, the kernel of the procedure (like `le'`) is programmed in COQ, and the correctness of the procedure (like `le'_le`) is also proved in COQ. Then invoking the procedure in interactive proofs is applying the correctness theorem, and then simplifying the goal. On more complex examples (see [Bou97, VGPA00]), the inference mechanism of `apply` is not powerful enough to find proper instantiations of the correctness theorem. Thus, a dedicated procedure written in ML (the implementation language of COQ) is needed. The purpose of this procedure is to reformulate the goal (with some hints from the user) as a problem expressed in the theory of the simplification procedure. The main advantage of the reflection approach is that the resulting proofs are small and thus, this may save a lot of time in type-checking. The drawback is that the simplification procedure must be proved correct in COQ which may be hard and limit some tricky optimizations.

1.2.4.3 Computations of types

In this last example, I give a more complex example of type computation. I define `(nary_arrow n A B)` as a type isomorphic to $A^n \rightarrow B$. It is defined by recursion on `n`:

```
Fixpoint nary_arrow (n:nat) (A B:Type) { struct n } : Type :=
  match n with
  | O => B
  | (S m) => A -> (nary_arrow m A B)
  end.
```

Hence, `(nary_arrow 3)` is evaluated into

```
fun A B : Type => A -> A -> A -> B
```

Of course, this type operator can now be used for typing some terms. Hence I define the term `(nary_anticons n nat)` as a function taking a list of natural `l`, `n` naturals and returning a list in which the `n` naturals have been pushed successively to `l`.

```
Fixpoint nary_anticons (n:nat) (A:Set) (l:(list A)) { struct n } :
 (nary_arrow n A (list A))
:=
  match n return (nary_arrow n A (list A)) with
  | O => l
  | (S n) => fun x:A => (nary_anticons n A (x::l))
  end.
```

Hence, `(nary_anticons 3 nat (3::2::nil) 4 5 6)` is evaluated into

```
6::5::4::3::2::nil
```

As expected `(nary_anticons 3 nat (3::2::nil) 4 5 6 7)` raises a type error.

1.2.5 Inductive types

As suggested by the previous examples, induction is a fundamental feature of CIC. The general idea of an inductive type T is that values of type T are generated from a finite set of *constructors* (functions with T as co-domain). Hence, the type T is defined by its constructors: it is the smallest type finitely generated from the constructors. This semantics of smallest fixpoint is given by the *induction scheme* (also called *elimination scheme* in reference of natural deduction rules) associated to each inductive definition.

Through eyes of Curry-Howard isomorphism, induction scheme has a double face: it is a reasoning scheme in deductions and it is a structural recursion operator in computations. Syntactic restrictions on the type of constructors ensure that the strong normalization property of CIC (termination of evaluation for any reduction strategy) is preserved: I detail them later.

Let me now give an example of the duality of induction (deduction and computation) on the well-known natural numbers. Peano's natural numbers are defined by the following inductive definition `nat`, introducing two constructors `O` and `S`:

```
Inductive nat: Set :=   O: nat  |   S: nat -> nat.
```

The inductive scheme generated for this inductive definition is the usual induction principle on naturals:

```
nat_rect: forall (P: nat -> Type),
  P O -> (forall n:nat, P n -> P (S n)) -> forall n:nat, P n
```

The computational behavior of this scheme is the following. Given `P: nat -> Type`, `f0: (P O)` and `fS: (forall n : nat, P n -> P (S n))`, we have the two following rewriting rules:

$$\begin{array}{rcl} \texttt{(nat\_rect P f0 fS O)} & \leadsto & \texttt{f0} \\ \texttt{(nat\_rect P f0 fS (S n))} & \leadsto & \texttt{(fS n (nat\_rect P f0 fS n))} \end{array}$$

Hence, the first parameter of `fS` corresponds to the access of the predecessor, where as its second parameter corresponds to a recursive call. Using this induction scheme, the predecessor of a natural can be computed by (the predecessor of `O` being here `O`):

```
Definition pred := (nat_rect (fun k:nat => nat) O (fun (k r:nat) => k)).
```

We verify that `(pred (S n))` reduces to `n` (in one step).

Similarly, using this induction scheme, the `plus` definition of page 8 can be written:

```
Definition plus (n m:nat) :=
  (nat_rect (fun k:nat => nat) m (fun (k r:nat) => (S r)) n).
```

Hence, `(plus O m)` reduces into `m` and `(plus (S n) m)` is inter-convertible with `(S (plus n m))`. The fact that `(plus m O)` equals `m` (for Leibniz's equality) requires a proof by induction in the style of example page 8. This proof by induction also uses `nat_rect`.

Using high-order computations, it is even possible to compute Ackerman's function:

```
Definition ack :=
 (nat_rect (fun k:nat => nat -> nat)
           (fun m:nat => S m)
           (fun (k:nat) (r:nat -> nat) =>
              (nat_rect (fun k:nat => nat) (r (S O)) (fun (k' r':nat) => r r')))).
```

Hence, we can check that `(ack O m)` reduces to `(S m)`, and `(ack (S n) O)` is inter-convertible with `(ack n (S O))`, and `(ack (S n) (S m))` is inter-convertible with `(ack n (ack (S n) m))`.

Actually, in all definitions given above, instead of `nat_rect`, we could instead use a weaker version, called *non-dependent elimination scheme*. This scheme is defined by:

```
Definition nat_ndrec: forall P: Type, P -> (nat -> P -> P) -> nat -> P
 := fun (P: Type) => (nat_rect (fun (k:nat) => P)).
```

Actually, the full version is used only in a few computation examples (like example `nary_anticons` page 14) or in proofs.

The constant `nat_rect` and its associated rewriting rules are not primitive constructions of the system. They are instance of two general constructions: a fixpoint operator (construction **Fixpoint** in previous examples) and a case-analysis operator (construction **match** in previous examples). But, these operators (in particular the fixpoint operator) are syntactically restricted in order to allow only definitions that could be written using induction schemes instead. These two constructions makes the system more user-friendly. Furthermore, the fixpoint operator allows to associate a name to each fixpoint constructions. Hence, with the definition of `plus` using the fixpoint operator, `(plus (S n) m)` reduces to `(S (plus n m))` instead of being only inter-convertible with it.

Actually, like variant-types of ML, inductive types of CIC allow to express booleans, pairs, unions, lists, trees, and so on. Furthermore, logical connectives such conjunction, disjunction and existential can be defined using inductive definitions. Propositions `True` and `False` are also defined using inductive propositions. Hence, `False` is defined as the inductive proposition with no constructor.

```
Inductive False: Prop := .
```

The non-dependent elimination scheme associated to `False` of type `forall P : Type, False -> P` expresses that if `False` is inhabited then any type is inhabited. Existential `ex` is defined by the following inductive proposition with `ex_intro` as single constructor.

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> ex P
```

The parameters `A` and `P` expresses that the whole inductive definition is parametrized by `A` and `P`. Given `A` and `P`, it defines the smallest proposition `P0` such `(forall x : A, P x -> P0)`. In other word, given `A` and `P`, the non-dependent elimination scheme is

```
forall (P0 : Prop), (forall x : A, P x -> P0) -> ex P -> P0
```

Hence, the full type of the non-dependent elimination scheme is

```
forall (A : Type) (P : A -> Prop) (P0 : Prop),
      (forall x : A, P x -> P0) -> ex P -> P0
```

However, not all variant-types of ML are allowed into CIC. For instance, the following definition (expressible in ML) is not authorized in CIC:

```
Inductive Absurd: Prop := foo: (Absurd -> False) -> Absurd.
```

If case-analysis were authorized on `Absurd`, we could define:

```
Definition Delta (x:Absurd) : False :=
 match x with
 | (foo g) => (g (foo g))
 end.
```

And then, we could define

```
Definition Omega: False := Delta (foo Delta).
```

Hence, CIC would be not consistent. Let us remark that in the computation of `Omega`, the evaluation of `(Delta (foo Delta))` reduces to itself. It is an infinite computation.

To forbid inductive definitions like `Absurd`, there are syntactic restrictions on the type of constructors: parameters of constructors can not be functions having a parameter of the type being defined; and, the type being defined can not be the effective parameter of a function.

To sum up this section, the semantics of inductive types as smallest fixpoint is ensured by the existence of the associated elimination scheme and by the strong normalization property of the calculus (see [Wer94] for a proof of this property). This property ensures that all closed terms of an inductive type can always be reduced to a term only built with constructors.

1.2.6 Equalities in CIC

Leibniz's equality (corresponding to the infix operator "`=`") is also defined by an inductive definition with a single constructor `refl_equal`:

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  refl_equal : (eq A x x)
```

Given `A` and `x` of type `A`, `(eq A x)` is the smallest predicate `P` of type `A -> Prop` such `P x`. In other word, given `A` and `x`, the non-dependent induction scheme is

```
forall (P : A -> Prop), P x -> forall y : A, (eq A x y) -> P y
```

Hence, the full type of the non-dependent induction scheme is

```
eq_ind:forall (A : Type) (x : A) (P : A -> Prop),
         P x -> forall y : A, (eq A x y) -> P y
```

The elimination scheme expresses that if two elements are equals, they satisfy the same predicates: in other words, no predicate can distinguish them. It is a principle of equality defined by Leibniz. From the typing point of view, this scheme can be considered as a way to explicitly cast a proof of type `(P x)` into a proof of type `(P y)`, when a proof of `(eq A x y)` is provided. To illustrate this point, let us build a proof that for all `n`, `O<>(S n)`. We have to show that for all `n`, we can build a proof of `False` from a proof of `(eq nat O (S n))`. We simply apply `eq_ind`, instantiating `P` on a predicate that returns proposition `True` if the parameter is `O`, and `False` in other cases. Provided a proof of `(eq nat O (S n))`, we can thus convert a proof of `True` into a proof of `False`.

```
Definition O_not_eq_S (n:nat): (eq nat O (S n)) -> False
 := (eq_ind nat O (fun n => match n with O => True | (S k) => False end) I (S n)).
```

In this proof, let us remark that the predicate `P` computes a proposition: it thus uses a second-order case-analysis over naturals.

Let us remark that if Leibniz's equality on closed terms reduces to inter-convertibility, it is not inter-convertibility in general. Indeed, proof of equality may for instance use induction, as in example page 8. For example, I construct below the proof that forall `n`, `n=(plus n O)`, where as `n` and `(plus n O)` are not inter-convertible terms.

```
Fixpoint plus_n_O (n:nat) : (eq nat n (plus n O)) :=
  match n return (eq nat n (plus n O)) with
  | O => (refl_equal nat O)
  | (S m) =>
      (eq_ind nat m (fun k => (eq nat (S m) (S k)))
                    (refl_equal nat (S m))
                    (plus m O)
                    (plus_n_O m))
  end.
```

Coming back to the discussion on dependent types, let us remark that the inter-convertibility induced some limitations on implicit type conversions. For instance, let us assume a variable `n` of type `nat`. The result type of `(appN n O)` is `(listN (plus n O))` which **is not** inter-convertible with `(listN n)`, because `(plus n O)` and `n` are not. But, the result type of `(appN O n)` which is `(listN (plus O n))` is inter-convertible with `(listN n)`, because `(plus O n)` reduces to `n`. Using the elimination scheme of equality, we can however convert `(appN n O)` into the type `(listN n)`. But this needs an explicit instantiation of the elimination scheme. Because of this delicate notion of equality in types, the use of dependent types is sometimes a bit tricky.

1.2.7 Well-founded recursion

CIC is not Turing-complete, because functions definable in CIC (without adding axioms) terminate. We have seen how to code Ackerman's function page 15 using the structural recursion of CIC. What are the recursive functions that we can define in CIC? Actually, we can code every recursive function for which we can prove (in CIC) that recursive calls strictly decrease with respect to a *well-founded* relation. I now define what is a well-founded relation in CIC.

Given a relation R, I say y is an ancestor of x for R, if $(R\ y\ x)$. Classically, a well-founded relation R is a relation such there is no infinite sequence of ancestors. In other words, R is well-founded if there is no sequence $(x_i)_{i\in\mathbb{N}}$ such for all i, x_{i+1} is an ancestor of x_i.

This is a definition by negation: it forbids some behavior. Usually, this kind of definition is not very interesting in intuitionistic logic, because double negation does not imply affirmation. In particular, in intuitionistic logic, we can not derive from a such negative definition an induction principle. Here, we would like to derive from the definition of well-foundation the usual well-founded induction principle.

To achieve this, COQ library introduces the accessibility predicate of a relation R. This accessibility predicate is defined inductively: an element x is accessible if all its ancestors are accessible. Hence, we can build a proof that an element is accessible by induction. Minimal elements (with no ancestor) are accessible. Then, elements such all ancestors are minimal are accessible. And so on. Formally, accessibility is given by the following inductive definition with a single constructor `AccT_intro`:

```
Inductive AccT (A:Type) (R: A -> A -> Prop) : A -> Prop :=
   AccT_intro : forall x : A, (forall y : A, R y x -> AccT A R y) -> AccT A R x.
```

Given a type `A` and a relation `R` over that type, `(AccT A R)` is the smallest predicate `P` such for all `x`, if `P` holds for all ancestors of `x`, then `P x`. We can thus derive the following induction scheme [2] for `AccT`:

```
forall (A : Type) (R : A -> A -> Prop) (P : A -> Prop),
  (forall x : A, (forall y : A, R y x -> P y) -> P x)
   -> forall y : A, AccT A R y -> P y
```

A relation is well-founded if all elements are accessible.

```
Definition well_foundedT (A:Type) (R: A -> A -> Prop) :=
 forall a : A, AccT A R a.
```

Then, by simply reformulating the previous induction principle on `AccT`, we can recover the usual induction principle:

```
Definition well_foundedT_ind:
 forall (A:Type) (R: A -> A -> Prop),
  (well_foundedT A R)
   ->  forall (P : A -> Prop),
         (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
            forall a : A, P a.
```

Given two types `A` and `B`, and a fixpoint generator `F: (A -> B) -> (A -> B)`, we say that `F` defines a *provably recursive* fixpoint if:

[2]It is weaker than the usual non-dependent elimination scheme, because I omit the case-analysis parameter in the induction hypothesis.

- there is a relation `R: A -> A -> Prop` with a proof `Rwf: (well_founded A R)`.
- there is a function `Fg: forall x:A, (forall y: A, (R y x) -> B) -> B` such for all `f:A->B` and all `x:A`, `(F f x)=(Fg x (fun y H => f y))`. In other word, recursive calls induced by `F` can be proved decreasing for `R`. Furthermore, `Fg` must also be monotonic in the following sense:

```
forall (x:A) (f g: (forall (y:A), (R y x)->B)),
  (forall (y:A) (H:(R y x)), (f y H)=(g y H))
    -> (Fg x f)=(Fg x g).
```

Under these assumptions, we can prove in Coq that `F` admits a unique fixpoint `fixF: A -> B`.

To construct this fixpoint, we first introduce some auxiliary functions. If an element is accessible, then, by definition, all its ancestors are also. In Coq, it is expressed by the following case-analysis:

```
Definition AccT_inv (A : Type) (R : A -> A -> Prop) (x : A) (H: AccT A R x):
   forall y : A, R y x -> AccT A R y
 := match H in (AccT _ _ x) return (forall y : A, R y x -> AccT A R y) with
    | (AccT_intro x IH) => IH
    end.
```

For any `H: (AccT A R x)`, `(AccT_inv A R x H)` returns a family of *structural predecessors* of `H`. These predecessors are parametrized by `y` of type `A` and a proof of type `(R y x)`. This is used in the definition of `fixFrec` below.

```
Fixpoint fixFrec (x:A) (H:AccT A R x) {struct H}: B
 := (Fg x (fun y Ryx => (fixFrec y (AccT_inv A R x H y Ryx)))).
```

`fixFrec` is defined by structural induction over `H`. Assuming `y:A` and `Ryx:(R y x)`, the recursive call `(fixFrec y (AccT_inv A R x H y Ryx))` satisfies the syntactic criteria of structural induction, since `(AccT_inv A R x H y Ryx)` is a structural predecessor of `H`.

At last, we can define `fixF: A -> B` and prove that it is the unique fixpoint of `F`.

```
Definition fixF (x:A) := (fixFrec x (Rwf x)).

Theorem fixF_is_fix:
  forall (x:A), (fixF x)=(F fixF x).

Theorem fixF_is_unique:
  forall (fix0: A -> B),
    (forall (x:A), (F fix0 x) = (fix0 x))
     -> forall (x:A), (fixF x)=(fix0 x).
```

We see here a very general way to encode recursive functions for which we can prove the termination in Coq: we exhibit an auxiliary parameter on which the proof of termination can be coded as structural recursion. Of course, in practice, it may not be very simple. Exhibiting `Fg` may be hard, especially in presence of nested recursive calls. Actually, in this case, the method above may need to be generalized using for instance a result type `B: A->Type` depending on the parameter. This allows to express a property on the result which is necessary to prove the decreasing of recursive calls.

Actually, even `Rwf` can be hard to prove. For instance, there is a function which is not definable in Coq without an axiom: the function deciding the equality between two well-typed terms of

Coq. Indeed, the well-foundation of Coq reduction over well-typed terms is not provable in Coq, because it is exactly the strong normalization property. If it were provable, then it would allow to prove Coq consistency in Coq, which contradicts a standard Gödel theorem. See [Bar99].

At end of this section, I do not know if the class of functions definable in Cic (without adding axioms) appears more clearly. Practicing Coq for a few years, I am convinced that most of the recursive functions that I will meet, are definable in Coq.

1.3 Coq in practice

If Cic is the kernel of the specification language of Coq, it is not the specification language of Coq. Indeed, Coq extends Cic syntax with a lot of syntactic sugar. But internally, Coq specifications are translated into Cic terms. I describe here the main syntactic sugar used in the report. Furthermore, I also present briefly Coq proof engine, since in chapter 4, I show how to write simple tactics to simplify refinement proofs using weakest-preconditions calculi.

1.3.1 The specification language

Actually the Coq specification language (called Gallina) is constituted by the language of commands (for instance, the **Definition** command that introduce a new definition) and the language of logical objects (terms). Here are some feature of this language that make easier the development of Coq specifications.

1.3.1.1 Implicit arguments

Polymorphisms and dependent types introduce some redundancies in terms. More precisely, when applying a function, some arguments of the function may be deduced by a type inference algorithm like ML's one. For instance, let us consider the inductive definition of polymorphic bounded lists. Given `A:Type` and `n:nat`, I define `(listT A n)` the type of the lists of size `n` containing elements of type `A`.

```
Inductive listT (A: Type): nat -> Type :=
  | nilT: (listT A O)
  | consT: forall (n:nat), A -> (listT A n) -> (listT A (S n)).
```

Given a type `A`, `(listT A)` is the smallest parametrized type `nat->Type` generated from `nilT` and `consT`. As this definition is parametrized by `A:Type`, `A` is the first parameter of `nilT` and `consT`. Hence, in Cic, a list of three booleans (of type `listT bool (S (S (S O)))`) is given by:

```
consT bool (S (S O)) true (consT bool (S O) false (consT bool O true (nilT bool)))
```

However, in these applications the type and the size of the lists could be inferred from the context. To avoid to write explicitly these instantiations, a first solution is to tell the type-checker to infer them by replacing the expected term with a _ pattern:

```
consT _ _ true (consT _ _ false (consT _ _ true (nilT _)))
```

A second solution is to make the corresponding arguments implicit, with the following command:

```
Implicit Arguments nilT [A].
Implicit Arguments consT [A n].
```

Then, Coq type-checker do not expect explicit instantiations for `A` or `n`, and automatically tries to infer them.

```
consT true (consT false (consT true nilT))
```

However, as type inference is not decidable in Coq (because of high-order unification), the type-checker may fail to infer terms using implicit arguments. In this case, Coq prints a type-error message, and the user is responsible to interpret this as a real type-error or a weakness of the inference algorithm. For instance, in the context of the following definition:

```
Definition f (g: bool -> (listT bool O) -> (listT bool (S O)))
 := g true nilT.
```

Then `(f consT)` fails to type-check. In this case, the user may help the type-checker by giving the explicit instantiation for implicit arguments, as `(f (consT (n:=O)))`.

The drawback of implicit arguments is that it makes the correction of a type-error more delicate: in presence of many implicit arguments, the user has to guess those which really miss. But they make the syntax very more light: the size of terms written by the user may be reduced logarithmically. Furthermore, they also improve the internal representation of terms, because the replacement of patterns lead terms to share (in the sense of "pointer sharing") many sub-terms. Hence, unexpectedly, implicit arguments make the type-checking faster.

In the remaining chapters of this report, I consider the following implicit arguments (as in the standard library of Coq):

```
Implicit Arguments eq [A].
Implicit Arguments AccT [A].
Implicit Arguments well_foundedT [A].
Implicit Arguments AccT_inv [A].
```

1.3.1.2 Predefined macros (records and numerals)

As we have already seen, instead of writing `(S (S (S O)))`, user can write `3` which is syntactically translated into this term.

Another macro is records. A record is simply coded as a type with a single constructor. Field access to the record are simply coded as projections (or destructors). For instance, the type constructor for product type can be defined as the following "record":

```
Record prodT(A B:Type) : Type :=
 pairT
 {
   fstT: A ;
   sndT: B
 }.
```

The use of this macro corresponds exactly to:

```
Inductive prodT(A B:Type): Type :=
  pairT: forall (fstT:A) (sndT: B), (prodT A B).

Definition fstT (A B:Type) (p: prodT A B): A
 := match p with
    | (pairT fstT sndT) => fstT
    end.

Definition sndT (A B:Type) (p: prodT A B): B
 := match p with
    | (pairT fstT sndT) => sndT
    end.
```

In the **Record** macro, the name of the constructor (here `pairT`) is optional. In its absence, a generic name is chosen instead: `Build_prodT`.
For this definition of `prodT`, it is a good idea to use implicit arguments:

```
Implicit Arguments pairT [A B].
Implicit Arguments fstT [A B].
Implicit Arguments sndT [A B].
```

Hence, given a pair `p:(prodT nat bool)` the first element of `p` is `(fstT p)` of type `nat`.

1.3.1.3 User notations

Coq provides a powerful mechanism to extend its concrete syntax, by adding new parsing and pretty-printing rules, expressed directly in the specification language. This allows to give a "friend-notation" to user constructions. The resulting grammar must be LL1. This mechanism is performed by Camlp4 [dR03].

For instance, Coq standard library introduces the standard infix symbol for equality. This notation is given by the following rule:

```
Notation "x = y" := (eq x y) (at level 70, no associativity).
```

Such syntactic rules allow to give the usual notations for arithmetic operators. See [Coq04].

1.3.1.4 Sections and Modules

1.3.2 The proof engine

Chapter 2

Proving impure programs in type theory

This chapter is intended to introduce intuitively the main ideas of my work. This chapter also advocates the use of Dijkstra specification and refinement. Indeed I present here two "standard" ways to embed imperative programs in type theory. The first one consists in plugging Hoare logic into type theory. The second one consists in giving a denotational interpretation of imperative programs into type theory, and then directly reason on this interpretation. At last, I show that refinement theory, which in some way combine this two approaches, allow to overcome the limitations of both these approaches.

2.1 The Hoare logic approach

In Hoare logic, specifications are annotations of the source of the program. These annotations consist in formulas expressing properties of the states of the program. For instance, in the following ADA-like program, comments are interpreted as assertions. The first one is named precondition and the last one is named postcondition. That means that during every call of `Example1`, if the precondition is satisfied *before* the call, then the postcondition is satisfied *after* the call. In the postcondition `x@` denotes the value of `x` before the call.

```
x: Integer ;

procedure Example1(y: in Integer) is
--@   y > 0  and  2*y <= x < 3*y
begin
  x := x - y ;
  x := x - y ;
end Example1 ;
--@   x = x@ % y
```

The main characteristic of this approach is to split the verification of a program into two steps: the generation of PO (proof obligations) that ensures (if there are proved) the correctness of the program, and the proof of these obligations themselves. This approach offers several interesting advantages: the generation of proof obligations can be done outside Coq and thus can be quite optimized, and it can be largely reused for other tools than Coq such other interactive theorems provers, or specialized decision programs. The tool Why is a beautiful illustration of these ideas [Fil03b].

$$\begin{tikzcd}[column sep=small] & \text{Annotated program} \arrow[dl, "\text{external PO generation}"'] \arrow[dr, "\text{forgetting annotations}"] & \\ \text{proofs} & & \text{compilation} \end{tikzcd}$$

On the preceding example, a possible proof obligation can be:

$$\forall x : \mathbb{Z}. \forall y : \mathbb{Z}. y > 0 \land 2 \times y \leq x < 3 \times y \Rightarrow x - y - y = x \bmod y$$

2.1.1 Calculus of weakest-precondition

PO generation is based on the notion of weakest-precondition calculus (wp-calculus). We have seen in the proof of the `plus` example of section 1.1 page 8, that partial evaluation of programs into the goals is a crucial feature in order to simplify the proof. The wp-calculus consists in the same idea: it performs a partial evaluation of the program (that is, it simulates successive state transformations) into the postcondition of the program. Hence, the wp-calculus maps a predicate R on the final state into a predicate on the initial state, which is the necessary and sufficient condition on the initial state to ensure R on the final state. The wp-calculus can be considered as a particular *Continuation Passing Style* semantics of imperative programs, where continuations are predicate on states. Such CPS semantics are standard denotational semantics (e.g. functional interpreters) for non-purely-functional languages.

Let us now briefly detail how PO are generated. To represent the semantics of the preceding function `Example1`, we suppose to work on a program with a global state of type S (in the preceding example, it is the type $\mathbb{Z}$ of the single variable x) and which values are of type V (which is $\mathbb{Z}$ in the example). The type Prop is the type of mathematical proposition. I define here the wp-calculus directly using the denotational semantics of imperative programs.[1] That means, I consider affectations as state transformers: functions a of type $S \to S$. Expressions are considered as functions e of type $S \to V$. Preconditions (and more generally predicates on state) are functions P of type $S \to \text{Prop}$. Postconditions are functions which map each initial state to a predicate on the state: they are functions Q of type $S \to (S \to \text{Prop})$. Procedures (called by values) are considered as functions mapping a tuple of values $\overrightarrow{v}$ into a Hoare-triple : (P, i, Q) where i denotes an instruction block, and is a value of the type Inst defined by the grammar below :

$$i ::= \{a\} \mid i_1; i_2 \mid p[\overrightarrow{e}]$$

Hence, a procedure is denoted by a function p of type $\overrightarrow{V} \longrightarrow ((S \to \text{Prop}) \times \text{Inst} \times (S \to (S \to \text{Prop})))$. If $\overrightarrow{v}$ is a tuple of type $\overrightarrow{V}$, we denote $P_{(p\ \overrightarrow{v})}$ (resp. $i_{(p\ \overrightarrow{v})}$ and $Q_{(p\ \overrightarrow{v})}$) the first (resp. second and third) projection of $(p\ \overrightarrow{v})$. An instruction i may be :

- an affectation $\{a\}$ with a of type $S \to S$.
- a sequence $i_1; i_2$ with i_1 and i_2 of type Inst.
- a procedure call $p[\overrightarrow{e}]$ consisting of a pair built with a procedure p and a tuple of effective parameters $\overrightarrow{e}$.

For instance, the procedure p associated to `Example1` is of type $\mathbb{Z} \longrightarrow ((\mathbb{Z} \to \text{Prop}) \times \text{Inst} \times (\mathbb{Z} \to (\mathbb{Z} \to \text{Prop})))$, and if $y : Z$, then

- $P_{(p\ y)} \stackrel{def}{=} \lambda x. y > 0 \land 2 \times y \leq x < 3 \times y$

[1]This trains the reader for the Coq embedding of wp-calculus

- $i_{(p\ y)} \stackrel{def}{=} \{\lambda x.x - y\}; \{\lambda x.x - y\}$
- $Q_{(p\ y)} \stackrel{def}{=} \lambda x.\lambda x'.x' = x \bmod y$

The wp-calculus is expressed as a function wp of type Inst $\rightarrow (S \rightarrow \text{Prop}) \rightarrow (S \rightarrow \text{Prop})$. For each defined procedure p, the generated OP is

$$\forall s : S.\forall \overrightarrow{v} : \overrightarrow{V}.(P_{(p\ \overrightarrow{v})}\ s) \Rightarrow (\text{wp}\ i_{(p\ \overrightarrow{v})}\ \lambda s'.(Q_{(p\ \overrightarrow{v})}\ s\ s')\ s)$$

The function wp is itself defined by induction on the structure of instruction blocks :

$$\begin{array}{rcl}
(\text{wp}\ \{a\}\ R\ s) & \stackrel{def}{=} & (R\ (a\ s)) \\
(\text{wp}\ i_1; i_2\ R\ s) & \stackrel{def}{=} & (\text{wp}\ i_1\ \lambda s'.(\text{wp}\ i_2\ R\ s')\ s) \\
(\text{wp}\ p[\overrightarrow{e}]\ R\ s) & \stackrel{def}{=} & (P_{(p\ \overrightarrow{(e\ s)})}\ s) \wedge (\forall s' : S.(Q_{(p\ \overrightarrow{(e\ s)})}\ s\ s') \Rightarrow (R\ s'))
\end{array}$$

In this definition, let us remark that affectation (resp. sequence) corresponds to its usual state transformers CPS semantics : application of function (resp. composition of functions). It is easy to check for `Example1` that we get the announced PO, applying the definitions above.

In procedure calls, preconditions must be satisfied, and the procedure is considered *as a black box* : the state s' after the call is only specified by the postcondition of the procedure. The instruction block of the procedure is not consider in the wp-calculus at the call. It is only considered in the PO associated to the definition of the procedure.

2.1.2 Limitations of Hoare logics

I present here some limitations of Hoare logics. These limitations appear to me when I tried to extend Hoare logic to support high-order functions with side-effects.

2.1.2.1 A *static* specification language

Specifications express static properties on the states at some given control point of programs, such "$x \leq 0$". Usually, some tricks allow to express some dynamics aspects.

Actually, we have already seen the first of these tricks: the use of special variable qualifier (e.g. $x@$), which allows to express a condition on the dynamic change of the state between the current control point and a preceding control point. For instance, "$x \leq x@$". JML, a hoare-logic for Java, introduces other qualifiers, to express for instance that an object has been newly allocated.

Another common trick resolves an important problem due to the use of procedures as black-box in procedure calls: procedures must specify that unchanged variables are unchanged. This problem is solved by adding a special clause to Hoare-triple, which specifies the lists of modified variables. However, the problem often still persists for arrays: non-modified indexes must be specified, and the specification comes sometimes more verbose than the program itself:

```
type TabInt is array(Integer range 0..N-1) of Integer ;
T: TabInt ;

procedure IncrT(i: Integer) is
--@ 0<= i < N
begin
  T(i):=T(i)+1 ;
end IncrT;
--@ T(i)=T@(i)+1
--@ and forall j:Integer, 0<=j<N and j /= i implies   T(j)=T@(j)
```

Often, white-box modularity is not supported in Hoare logics. For instance, let us suppose we want define a procedure `IncrT2` which calls two times `Incr`. We can not exactly specify `IncrT2` as "`IncrT2(i)` *performs successively two times* `IncrT(i)`". We are actually obliged to compute (manually) the specification of `IncrT2` :

```
procedure IncrT2(i: Integer) is
--@ 0<= i < N
begin
  IncrT(i) ;
  IncrT(i) ;
end IncrT;
--@ T(i)=T@(i)+2
--@ and forall j:Integer, 0<=j<N and j /= i implies   T(j)=T@(j)
```

The problem is, that if later we change the specification (and the implementation) of `IncrT`, we are obliged to change the specification of `IncrT2`, even if morally its specification is unchanged: it calls successively two times `IncrT`. In this case, we see that the implementation (which remains unchanged) is more expressive, more compact, and more easier to understand than the specification. However, to overcome this problem, we can still imagine a trick: give no specification to `IncrT2` and indicate to the wp-calculus via a special comment that calls to `IncrT2` must unfold its code, instead of abstract it using the specification. Hence `IncrT2` is considered as kind of macro, and it is called *a white-box*, because its specification is its code.

2.1.2.2 Difficulty to specify high-order procedures with side-effects

Let us now consider the following OCAML imperative iterator on lists:

```
let rec iter f = function
    [] -> ()
  | a::l -> f a; iter f l
```

How specify `iter`? Here, we do not want to give a precise specification to the parameter `f` : it is an action taken as a parameter of the code. The intuitive specification of `iter` is to perform all over the list the action `f` in parameter. Hence, we recover the problem of `IncrT2`. A naive solution to express this, is here to consider `iter` as a white-box.

However, in practice, it seems useful to mix the two points of view: this leads to *grey-box* specifications, using the terminology introduced in [BW99]. For instance, such high-order iterators are very useful when handling a complex data-structure, which representation should remain abstract: they provide a generic traversal of the data-structure. To specify such iterators, while keeping the representation abstract, a solution consists in introducing an abstract view of the data-structure as a list, and to express that the iteration on the concrete data-structure corresponds to the iteration on the abstract list. The correspondence between the concrete data-structure and the list may be quite abstract (e.g. non-deterministic). Such a specification is said to be *grey-box*, because it mixes hiding of the data-structure (black-box) with effective computations (white-box). Let us remark, that such high-order iterators can be implemented in any serious imperative programming language: using genericity of ADA, using function pointers in C, or using objects in Java. I present other examples of grey-box specifications chapter 5 (in particular section 5.2.3).

To express such grey-boxes, an idea is to extend the specification language: it must at least contain the implementation language. Hence, our specifications may have an imperative behavior. Further more, as we are interested in having high-order functions in the language, imperative functions must also be values of the logic. Here, we are faced to an other problem: is it sensible to extend the PO generation of Hoare logic with imperative functions as objects of the logic? Answering this question require to analysis a bit more the interaction between the user and the system during the proof process.

2.1.2.3 A poor control of the proof process

Actually, a user of Hoare logic does not need to really deeply understand the process of PO generation: she only has to understand the meaning of the PO (this helps her to prove them), but she does not need to know how they are generated. Hence, a nice point of Hoare logic, is that the user does not need to exactly know how programs are logically interpreted.

However, in practice, it is sometimes hard to understand the meaning of PO: the structure of the program has disappeared, a single variable may be split into several logical variables representing different observation points of the variable. Further more, size of formulas can explode during the wp-calculus.[2] Actually, the user has usually no real control on the PO generation process: she has no interactive control on the global structure of her proof. Usually, the only form of control is to modify the specification or the annotated program, by inserting for instance some intermediate assertions. The global structure of the proof is guided by the structure of the annotated code.

Dealing with high-order functions, this approach seems not sensible. In particular, in the grey-box approach, the user needs to control the unfolding of definitions. This allows her to avoid the explosion of the goal or to repeat some proofs already done. Indeed, some proofs can be simplified by applying some high-order lemma already proved: the structure of the code is not sufficient to guess this. That means that proofs of high-order programs with side-effects must combine standard deduction steps with wp-computation steps. It is not possible to separate the two worlds any more.

2.2 A naive denotational approach: equational reasoning in monads

Drawbacks of Hoare Logics come fundamentally from the fact that programs are not object of the logic. Another approach to reason about non-purely functional programs into Coq is to represent them as functions in Coq, and to reason directly about their denotation. This approach, often called *shallow embedding*, consists in using the denotational semantics to specify and prove programs. In practice, we can imagine to provide a concrete syntax for specifications and programs, and a translator into Coq to build interactive proofs. But, I do not consider this external mechanism here, and I suppose the user write directly her specifications and programs in Coq (possibly using Coq macros to have a nicer syntax). Hence, to write her specifications (and programs), the user imports a Coq library, the denotational semantics of the language, that provides at least:

- functions encoding the different constructions of the programming language.
- an observational equivalence between programs (that should actually be a congruence on programs[3]).
- some common equivalence lemmas, often used in proofs.

User specifications are thus Coq lemmas, expressing an equivalence property under several assumptions (see for instance [BH01a, BH01b]). In this section, we will illustrate this approach on a language for expressions with side-effects and a global state.

[2] A big difficulty in concrete implementations of PO generation is to adapt the wp-calculus in order to limit this explosion.

[3] That means, that constructs of the language should preserve equivalence

2.2.1 Introduction to monads

Following Haskell effort, embeddings of non-purely-functional features into purely-functional language are now well-developed. Monad theory provides a relatively generic way to handle these embeddings [Mog91, JW93]. There are monads to describe side-effects such state or exception handling, environment interactions and finite non-determinism. Monads provide a simple interface to extend all functional constructions with such non-functional features: when a monad interface is provided for these non-functional features, then we automatically get the possibility to mix these non-functional features with all the functional ones: high-order functions, inductive data-types, structural recursion, pattern-matching... All these kind of side-effects can be get without losing the good compositionality property of functional programming. Let us now introduce the intuition behind monads.

2.2.1.1 Intuitive presentation of monads

The fundamental property of pure functional languages, called Church-Rosser property, is that the result of a computation does not depend on the strategy to compute subexpressions. In usual functional languages, the strategy may have an influence on the termination of the computation, but not on the result if there is one. In Coq, all strategy terminates (but in counter-part, well-founded recursions must be justify by a proof of termination). The Church-Rosser property simplify a lot the reasoning about programs: two expressions are equivalent if they produce the same result.

Let us illustrate this point on a example in Coq. We define a `combine` function, taking a binary operator `op`, and an element `x`, and computing `(op x x)`.

```
Definition combine (A B:Type) (op:A->A->B) (x:A) := (op x x).
Implicit Arguments combine [A B].
```

If we compute `(combine mult (plus (1) (1)))`, we obtain `(4)` as expected. How can we get this result. To start the computation, we can have at least two strategies: first, we compute `(plus (1) (1))` and then substitute arguments in `combine`, or second, we start by substituting arguments. The first way leads to compute `(combine mult (2))` whereas the second way leads to compute `(mult (plus (1) (1)) (plus (1) (1)))`. However, the final result is the same.

Now let us suppose that Coq allows to write imperative programs. Assuming `x` an imperative variable on naturals, and assuming that `++x` means incrementing `x` of 1 and then returning the value of `x`. What is the result of `(combine mult ++x)`, when `x` is initially `0` ?

In the first way, we may have the following computations:

```
combine mult ++x        # x=0
→ combine mult 1        # x=1
→ mult 1 1              # x=1
→ 1                     # x=1
```

In the second way, we may have:

```
combine mult ++x        # x=0
→ mult ++x ++x          # x=0
→ mult 1 ++x            # x=1
→ mult 1 2              # x=2
→ 2                     # x=2
```

Here, the Church-Rosser property is lost. Indeed, the observational equivalence between impure computations can not be reduced to the equivalence of their results (for instance, two imperative computations are observationally equivalent if and only if they produce the same value, and the same effect on the memory). In particular, the type language must distinguish *computations* from *values* (i.e results of computations).

The monadic interface allows simply this distinction. A monad is defined by a parametrized type `K` (where `(K A)` represents the type of a computation returning a value of type `A`), a function `val` lifting values to computations and a function `bind` to compose computations, by passing the result of a computation to an other computation parametrized by a value of a compatible type.

```
K : Type -> Type.
```

```
val : forall (A:Type), A -> (K A).
Implicit Arguments val [A].

bind : forall (A B:Type), (K A) -> (A -> (K B)) -> (K B).
Implicit Arguments bind [A B].
```

The `bind` operator corresponds to a "let-in" construct in ML: it generalizes sequence of imperative languages. The `val` operator is side-effect free: it only allows to consider a value as a degenerated form of computation. Of course, the monads must come with its own equivalence relation on computation, that `bind` must preserve. In the following, we consider that observational equality on values is Leibniz equality:

```
equiv : forall (A:Type), (K A) -> (K A) -> Prop.
Implicit Arguments equiv [A].
```

A monad must thus satisfies the following properties:

```
equiv_refl : forall (A:Type) (x:(K A)), (equiv x x).
equiv_sym : forall (A:Type) (x y:(K A)), (equiv x y)->(equiv y x).
equiv_trans : forall (A:Type) (x y z:(K A)), (equiv x y)->(equiv y z)->(equiv x z).

bind_compat :
  forall (A B:Type) (k1 k2:(K A)) (f1 f2: A ->(K B)),
    (equiv k1 k2)
     -> (forall (x:A), (equiv (f1 x) (f2 x)))
       -> (equiv (bind k1 f1) (bind k2 f2)).

bind_val_l : forall (A B:Type) (a:A) (f:A->(K B)), (equiv (bind (val a) f) (f a)).

bind_val_r : forall (A:Type) (k:(K A)), (equiv (bind k (fun a => (val a))) k).

bind_assoc :
  forall (A B C:Type) (k:(K A)) (f: A->(K B)) (g: B -> (K C)),
    (equiv (bind (bind k f) g) (bind k (fun a => (bind (f a) g)))).
```

Hence, when embedding a monad into a functional language, evaluation of computations is split into two levels: the monad level (at this level, computation are built using the monad-specific operators) and the functional level, where computations of the monad level appear as values. Church-Rosser is valid at the functional level, and at the monad level, the order of evaluation is explicit. For instance, lifting "pure" functions into monads needs to explicit if the functions will take computations or values as parameter. Hence, their are several ways to lift `mult` into a monad:

```
Definition lifted_mult1 (x y: nat) := (val (mult x y)).
```

Or

```
Definition lifted_mult2 (x y: (K nat)) :=
   (bind x (fun v1 => bind y (fun v2 => val (mult v1 v2)))).
```

When "applying" `combine` to each version of these lifted multiplication, we recover (with a clear distinction in the syntax) the two preceding behaviors:

```
bind ++x (fun v => (combine lifted_mult1 v))
```

Or

```
combine lifted_mult2 ++x
```

Hence, the monad interface provides a way to make "independent" functional computations (substitutions of parameters, pattern-matching, structural recursion) from non-functional computations (state modification, input/outputs, etc), preserving the Church-Rosser property of functional computations.

2.2.1.2 Mixing functional and non-functional programming

Given a monad, we can automatically extend it with functional programming features. For instance, for a call-by-value language, high-order expressions returning functions with side-effects are of type `(K A->(K B))`. Hence, we can provide the following programming construct: `lambda` (λ-abstraction of a variable representing a value) and `apply` (call-by-value application[4]).

```
Definition lambda (A B:Type) (f:(A -> (K B))): K (A->(K B)) := (val f).
Implicit Arguments lambda [A B].

Definition apply (A B:Type) (f: K (A->(K B))) (a:K A): (K B)
 :=(bind a (fun x => bind f (fun g => g x))).
Implicit Arguments apply [A B].
```

We can easily prove the β-reduction equivalence:

```
Lemma beta_equivalence: forall (A B:Type) (f: A -> (K B)) (a:(K A)),
  (equiv (apply (lambda f) a) (bind a f)).
```

Then, we can provide a construction to lift pure functions into the monad level:

```
Definition ext (A B:Type) (f: A -> B): K (A->(K B)) :=
  (lambda (fun x => val (f x))).
Implicit Arguments ext [A B].
```

Monads are also fully compatible with a nice programming feature of Coq: inductive types with associated pattern-matching and structural recursion. For instance, given the inductive scheme on naturals,

```
nat_rect: forall P : nat -> Type,
             P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

we can lift it on monads by:

```
Definition nat_recm (A:Type) (k: K nat) (init:K A) (rec: nat->(K A)->(K A)): (K A)
  := (bind k (nat_rect (fun _ => (K A)) init rec)).
Implicit Arguments nat_recm [A].
```

The good news is that this inductive scheme on natural is compatible with equivalence:

```
Lemma nat_recm_compat:
  forall (A:Type) (n1 n2:(K nat))   (i1 i2:(K A)) (r1 r2: nat->(K A)->(K A)),
   (equiv n1 n2)
    -> (equiv i1 i2)
      -> (forall (n:nat) (hr1 hr2:(K A)),
             (equiv hr1 hr2)->(equiv (r1 n hr1) (r2 n hr2)))
        -> (equiv (nat_recm n1 i1 r1) (nat_recm n2 i2 r2)).
```

[4] We follow the OCaml convention: arguments are evaluated from right to left.

The proof of this lemma is trivial: by induction on `m`, it is easy to prove that

```
forall (A:Type) (i1 i2:(K A)) (r1 r2: nat->(K A)->(K A)),
  (equiv i1 i2)
    ->(forall (n:nat) (hr1 hr2:(K A)),
          (equiv hr1 hr2)->(equiv (r1 n hr1) (r2 n hr2)))
      -> forall (m:nat), (equiv (nat_rect (fun _ => (K A)) i1 r1 m)
                                (nat_rect (fun _ => (K A)) i2 r2 m)).
```

Then, applying `bind_compat`, we can conclude.

Such a definition can be adapted to all inductive types, such lists, binary trees, etc. Instead of using inductive schemes, we can also use directly pattern-matching and structural induction. For instance, to define a `If` in monads, we instantiate the pattern-matching on booleans.

```
Definition If (A:Type) (ck:(K bool)) (tk:(K A)) (ek:(K A)) : (K A)
 := bind ck (fun b => match b with true => tk | false => ek end).
Implicit Arguments If [A].
```

The proof that `If` is compatible with equivalence is the same as above.

Using the predefined type `unit` of single value `tt`, we say a monadic expression is an instruction if its type is `(K unit)`. The type `unit` is very similar to the type `void` of C. The skip instruction is thus `(val tt)`. Sequencing instructions is just applying `bind`. Using induction on naturals, we can thus define a `forN` construction:

```
Definition skip: (K unit) := (val tt).

Definition seq (A:Type) (i1:(K unit)) (i2:(K A)) : (K A)
  := bind i1 (fun _ => i2).
Implicit Arguments seq [A].

Fixpoint forN (n:nat) (i:(nat -> (K unit))) {struct n}: (K unit)
 := match n with
    | O => skip
    | (S n) => (seq (forN n i) (i (S n)))
    end.
```

Hence, informally, a program of the form:

```
for i=1 to n do
  p(i)
done
```

could be translated into `(bind n (fun x => (forN x (fun i => p i))))`. Let us remark that `seq` and `forN` are of course compatible with equivalence.

2.2.1.3 Main applications of monads

The main success story of monads is their application in Haskell to provide an interface for I/O programming in this functional language. As Haskell is a lazy functional language, it is not easy for a user to to predict the order of evaluation. Hence, the monadic interface of the I/O library, allows the user to keep the control on evaluation order for I/O effects inside her lazy programs. [JW93]

An other application of monads is the structuring of programming language semantics (or interpreters/compilers) [LH96]. Indeed, studying the semantics of a programming language (or implementing an interpreter or a compiler), it sounds interesting to define separately "atomic features" of the language, and then to study how combinations of these "atomic features" can work. This is a basic idea of software engineering: "keeping things as modular as possible". By providing an abstract interface to handle "non functional features" of programming language, monads are a way to achieve such a dream. I discuss section 2.4.1 how to reuse this idea in our context.

2.2.2 State handling monad

We have seen in the previous section that if we build a monad for a given non-functional programming language in Coq, then we can automatically extend it with high-order functions and inductive types. We now study how to build a monad and how to use it for specifying, programming and proving. We examine these points on the state handling monad: a monad that describe expressions modifying a global state.

2.2.2.1 User interface of the state handling monad

The state handling monad is parametrized by a variable `St: Type`, the type of the global state. The state handling monad is a monad with two additional constructions to set or get the current value of the state.

```
set : St -> (K unit).
get : (K St).
```

These operations satisfy the following properties:

```
get_set : (equiv (bind get set) skip).
set_get : forall (st:St), (equiv (seq (set st) get) (seq (set st) (val st))).
set_set : forall (st1 st2:St), (equiv (seq (set st1) (set st2)) (set st2)).
```

I said before that user specifications are written as equivalence properties. Actually, both in specifications and in proofs, it is much easier to reason on an equality between computation for a given initial state. Indeed, let us defines

```
Definition eqInEnv (A:Type) (st:St) (k1 k2:(K A))
  := (equiv (seq (set st) k1) (seq (set st) k2)).
Implicit Arguments eqInEnv [A].
```

Using the properties of state handling monad given above, it is easy to prove:

```
forall (A:Type) (k1 k2:(K A)), (equiv k1 k2) <-> forall (s:St), (eqInEnv s k1 k2).
```

A main reason to use `eqInEnv` is it has good decomposition properties. For instance,

```
forall (A:Type) (k1 k2:(K unit)) (k3 k:(K A)) (st:St),
  (eqInEnv st k1 k2)
   ->(eqInEnv st (seq k2 k3) k)
    ->(eqInEnv st (seq k1 k3) k).
```

or

```
forall (A B:Type) (k1:(K A)) (k2: A->(K B)) (k3:(K B)) (st1 st2:St) (a:A),
  (eqInEnv st1 k1 (seq (set st2) (val a)))
    -> (eqInEnv st2 (k2 a) (seq (set st1) k3))
       -> (eqInEnv st1 (bind k1 k2) k3)
```

Hence, eqInEnv allows to express precisely how computation is running. It has also some simple properties such:

```
forall (s:St), (eqInEnv s get (val s))
```

or

```
forall (A:Type) (k1 k2:(K A)) (s:St),
  (eqInEnv s k1 k2) -> (eqInEnv s (seq (set s) k1) k2).
```

2.2.2.2 Building the state handling monad

The state handling monad is built using the definitions below. A computation is a function taking a value of the state and returning a pair formed by a new value of the state and a result. Observational equality on computations is extensionality.

```
Definition K (A:Type) := St -> (prodT St A).

Definition equiv (A:Type) (x y: St -> (prodT St A)) := forall st:St, (x st)=(y st).

Definition val (A:Type) (a:A) (st:St) := (pairT st a).

Definition bind (A B:Type) (k:(K A)) (f:(A -> (K B))) (st:St)
 := match (k st) with (pairT st v) => (f v st) end.

Definition set (st dum:St) := (pairT st tt).

Definition get (st:St) := (pairT st st).
```

It is straightforward to check that these definitions satisfy the required properties.

2.2.2.3 Using the state handling monad

We have seen in the previous monad what could be a Coq library for state handling monad. On simple examples, if there is a sufficient base of lemmas in the library (in particular concerning eqInEnv), the user do not need to know how the monad is implemented. Syntax using monad constructions is very close to concrete syntax of imperative programs: the state remains implicit. Furthermore, Coq evaluation helps the user in simplifying proofs. For instance, let us suppose that we have defined the following function (described in a informal concrete syntax) :

```
var x: nat ;
function decrN(n: nat): unit is
  for i=1 to n do
    match x with
    | O => skip
    | (S p) => x:=p
    end
  done
```

This function take an argument n and decrements the variable x. As we have no exceptions here, if n is bigger than x, then we just perform any action: here x will be set to O). The function decrN can be coded in state handling monads with `St=nat` in the following way:

```
Definition decrN: (K nat->(K unit)) :=
 (lambda (fun (n:nat) => forN n (fun i =>
                                 bind get (fun m =>
                                      match m with
                                      | O    => skip
                                      | (S p) => (set p)
                                      end)))).
```

Specification of this function can be given by:

```
Lemma decrN_spec: forall (n m:nat),
   (le n m) -> (eqInEnv m (apply decrN (val n)) (set (minus m n))).
```

This lemma is easy to prove by induction on `n`, and using properties of `minus`.

2.2.3 Mini-example mixing functional and non-functional features

To illustrate how specify and use high-order functions or structural induction with side-effects, we suppose to work on a monad with state handling and error raising. With respect to the state-handling monad, we suppose to have also the `error` operator which can abort the execution:

```
error : forall (A:Type), (K A).
Implicit Arguments error [A].
```

This operator satisfy the following property:

```
bind_error_l :
 forall (A B:Type) (f: A->(K B)) (equiv (bind error f) error).
```

We define in Coq syntax below a function `lookup`, parametrized by a `test` function, that lookups in a list the first element satisfying the test, and that aborts if no such element exists :

```
Fixpoint lookup (A:Set) (test:A->(K bool)) (l:(list A)) {struct l}: (K A) :=
  match l with
  | nil => error
  | (cons a m) => If (test a) (val a) (lookup test m)
  end.
Implicit Arguments lookup [A].
```

A priori, this function does not seem to modify the state. But actually, `test` can modify the state, and hence `lookup` also.

Equational reasoning allows to specify some behavior of `lookup` depending on the behavior of `test`. For instance, we say that `test` is side-effect free, if there is `p: A -> St -> bool` such that

```
forall (a:A), (equiv (test a) (bind get (fun st => (val (p a st))))).
```

Indeed, in this case, `test` do not abort and do not modify the state **observationally**.[5] Hence, it is possible to prove the two following properties of `lookup`. The first indicates that if `test` returns

[5] It means that internally, `test` can modify the state, but at the end of its execution, the state must be reset to its initial value.

always `false` for elements in the list, then `lookup` aborts.

```
Lemma lookup_when_pur_reject:
  forall (A : Set) (test : A -> K bool) (p : A -> St -> bool),
   (forall a : A, equiv (test a) (bind get (fun st => val (p a st))))
  -> forall (l : list A) (sti : St),
      (forall a : A, In a l -> (p a sti)=false)
     -> eqInEnv sti (lookup test l) error
```

The second specification indicates that when there is an element in the list satisfying `p`, then the first element satisfying `p` is returned.

```
Lemma lookup_when_pur_accept:
 forall (A:Set) (test:A->K bool) (p: A -> St -> bool),
   (forall (a:A), equiv (test a) (bind get (fun st => val (p a st))))
  -> forall (l m:(list A)) (a:A) (sti:St),
      (forall (b:A), (In b l) -> (p b sti)=false)
      -> (p a sti)=true
       -> eqInEnv sti (lookup test (app l (cons a m))) (val a).
```

It also possible to specify a particular behavior of `lookup` when the result of `test` does not depend on the state (that `test` can modify). Hence, we consider the case, where there are `p: A->bool` and `k: (K unit)` such **forall** `(a:A), (equiv (test a) (seq k (val (p a))))`. A typical example of such a test, is a function that counts the number of time it is called.[6] We can prove two other properties of `lookup`:

```
Lemma lookup_non_pur_reject:
 forall (A:Set) (test:A->(K bool)) (p: A -> bool) (k:(K unit)),
  (forall (a:A), equiv (test a) (seq k (val (p a))))
  -> forall (l:(list A)),
     (forall (a:A), (In a l) -> (p a)=false)
      -> equiv (lookup test l) (seq (forN (length l) (fun _ => k)) error).

Lemma lookup_non_pur_accept:
 forall (A:Set) (test:A->(K bool)) (p: A -> bool) (k:(K unit)),
  (forall (a:A), equiv (test a) (seq k (val (p a))))
  -> forall (l l':(list A)) (a:A),
      (forall (b:A), (In b l) -> (p b)=false)
      -> (p a)=true
       -> (equiv (lookup test (app l (cons a l')))
                (seq (forN (S (length l)) (fun _ => k)) (val a))).
```

These four properties are proved by structural induction on `l`.

This example illustrates in particular how high-order functions can be mixed with side-effects. We see here that there is not a single “specification” attached to such a code, but there are “patterns” that can be reused in different contexts.

[6] Applying `lookup` to a test counting the number of time it is called allows for instance to know the position of the element returned by `lookup` in the list.

2.2.4 Monads versus Hoare logic

We have seen above, that this approach allows to overcame some weaknesses of Hoare logic:

- it allows to marry functional and non-functional features in specifications and in programs.
- it allows to express dynamic specifications. For instance, it is possible to express that under given assumptions, an expression is observationally equivalent to an other. Specifications can also be parametrized by a non-functional computation.
- User can fully control the structure of the proof, and the prover helps the user using symbolic evaluation of the program to simplify goals.

However, this approach is not a serious alternative to Hoare logic, because of several important limitations.

2.2.4.1 Reasoning with observational equality is not sufficient in practice

Given an imperative expression `expr` on a global state of type `St` returning a result of type `A`, a Hoare specification of `e` can be seen as the couple of two predicates:

- a precondition `P: St -> Prop`, a predicate on the initial state.
- a postcondition `Q: St -> St -> A -> Prop`, a predicate on the initial state, the final state and the result.

A such specification can be expressed in monads by the following formula:

```
forall (sti:St),
 (P si)
   -> (exists stf:St,
        (exists r:A, ((Q sti stf r) /\ (eqInEnv sti expr (seq (set stf) (val r)))))).
```

In the proof of such a formula, the user will have to provide a value for the final state and the result (proof of existence), whereas the wp-calculus compute such values automatically. Actually, knowing the implementation of monads, the user can rewrite its specification in

```
forall (si:St),
  (P si)
    ->(match (expr si) with (pairT sf r) => (Q si sf r) end)
```

Then, the preceding problem disappears. This shows that observational equality is not sufficient to reason in monads, and that monads must be extended with constructions to express Hoare Logic specifications.

In particular, in Hoare Logic, it is useful to express assertions at some control point of the code. For example in ML programs, it may factorize some proofs to be able to express assertions of the following form, where `x_1` and `x_2` are non-mutable variables (they are not in the state).

```
...
 let x_1 = ... in
 ...
 let x_2 = ... in
 ...
  (* here P depending on x_1 and x_2 holds *)
 ...
```

In the monadic approach, it is not easy for the user to express such assertions.

2.2.4.2 Representation of non-terminating programs

The constructions of the state-handling monad have a simple denotations in Coq. But it is not always the case. A good example is `while`-loop.

Well-founded recursion of Coq requires a priori a proof of termination: the user must supply this proof in the term instantiating well-founded recursion. Indeed, Coq rejects a priori potential non-terminating terms. This is too constraining here, because this obliges to prove the correction of the program (proving termination often requires extra invariants properties) while defining the program. It would be preferable to prove termination a posteriori: after the definition of the program. This means that we must represent non-terminating programs.

It is possible to define denotations to represent non-terminating programs (for instance, using co-induction in Coq), but they make the embedding more complicated to encode.

2.2.4.3 Dealing with a local state

In monads dealing with a local state is not easy: the user must give explicitly the transformation of the state space. Using a specification and programming language outside Coq, would allow to infer these informations (by an effect analysis for instance). [Wad99, Fil99a] In practice, it seems thus still necessary to provide to the user a specification and programming language external to Coq.

$$\begin{tikzcd}
& \text{Specifications and programs} \arrow[dl, "\text{syntactic translation}"'] \arrow[dr] & \\
\text{Coq text} & & \text{compiler}
\end{tikzcd}$$

2.3 Dijkstra Specification Languages

In this section, we show how to solve the limitations exposed in the preceding section, excepted the problem of the transformation of the state space (for which, I propose some partial solution chapter 4). Indeed, we show how to extend the preceding approach to embed Hoare logic specifications in monads, using refinement.

The main idea of this approach is the following: the specification language must be an extension of the programming language. Hence, a specification is a kind of abstract algorithm, that describes a collection of *authorized* behavior. The refinement is an order relation between specifications such that S_1 refines S_2 if and only if every behavior authorized by S_1 is authorized by S_2. A (deterministic) program is seen as a specification authorizing syntactically at most one behavior. Observational equality between programs is just a special case of refinement. And, the relation P *implements* S, where P is a program and S is a specification, is also a special case of refinement.

2.3.1 Algebraic refinement theory: an extension of monad theory

2.3.1.1 Definition of Dijkstra Specification Languages

Using monad terminology, specifications are computations. For a given monad, the corresponding specification language can be obtained by extending the monad with three constructions:

- `require :` **Prop** `-> (K unit)` expresses that the environment at execution must satisfy a property. Any behavior is authorized if the environment do not satisfy the property at this point of control in the specification. Hence, this construction allows to state a precondition on the environment at the execution of authorized behaviors.

- `ensure : Prop -> (K unit)` expresses that a property is guaranteed at execution. Only behavior satisfying this property at this point of control of the specification are authorized. Hence, this construction allows to state a postcondition on the authorized behaviors.
- `any : forall (A:Type), (K A)` specify any concrete value of the same type. This construction is often called unbound (demonic) non-determinism in refinement theory. It allows in particular to abstract a complex computation in a black box.

For instance, in a such specification monad, the following expression (where `n` is a natural) specifies that assuming `(P n)` holds, any value of the form `n+m`, such `(Q n m)` holds, is computed.

```
seq (require (P n))
    (bind (any nat) (fun m => (seq (ensure (Q n m)) (val (plus n m)))))
```

The semantics of such specifications can be expressed using refinement. Refinement is an order relation `refines` of type `forall (A:Type), (K A)->(K A)->Prop`, which associate equivalence is observational equivalence. Furthermore it is compatible with all constructions of the language (`bind` in particular). Hence, all constructions of the language are increasing for this order.

At last, `any`, `ensure` and `require` satisfy the following properties:

```
any_refined : forall (A : Type) (a : A), refines (val a) (any A).
any_refines : forall (A B : Type) (s1 : A -> K B) (s2 : K B),
    (forall a : A, refines (s1 a) s2)
       -> refines (bind (any A) s1) s2.

bind_require : forall (A : Type) (p1 p2 : Prop) (s1 s2 : K A),
    (p2 -> (p1 /\ refines s1 s2)) ->
      refines (seq (require p1) s1) (seq (require p2) s2).
require_True : equiv (require True) skip.

bind_ensure : forall (A : Type) (p1 p2 : Prop) (s1 s2 : K A),
    (p1 -> (p2 /\ refines s1 s2)) ->
      refines (seq (ensure p1) s1) (seq (ensure p2) s2).
ensure_True : equiv (ensure True) skip.
```

Let us remark that the term `(fun A => seq (require False) (any A))`, called `abort`, is an aborting operation quite similar to the preceding `error`.[7] This `abort` authorizes every behavior. Thus, it is the maximum of the refinement order.

The construction `(fun A => seq (ensure False) (any A))` corresponds to an unimplementable specification. It is called `magic` in refinement theory. It authorizes no behavior. Hence, it is the minimum of the refinement order.

Definition 2.3.1 (Dijkstra specification language) *A* Dijkstra specification language *(DSL in short) is a monad with an order* `refines`*, such* `bind` *is compatible with it, and with three operators* `any`*,* `require` *and* `ensure` *satisfying the six properties above.*

In the following, we use the name `DS` instead of `K` to indicate that we work on a DSL, and not on a "simple" monad.

[7] Actually, on a state handling monad, `abort` is required to satisfied an additional property expressing that when `abort` is raised, the state can be anything. On the contrary, an `error` operator usually leaves the state unchanged when it is raised.

2.3.1.2 Extending monads into DSL: example of the state handling monad

Actually, for any specific monad, it seems possible[8] to get a nice DSL by adding to the preceding ones only a few axioms, expressing that `any`, `require` and `ensure` do not produce specific effects. For instance, for the state-handling monad, we need to add:

```
set_any : forall (A:Type) (st:St),
  (equiv  (seq (set st) (any A))  (bind (any A) (fun a => seq (set st) (val a)))).

set_require : forall (p:Prop) (st:St),
  (equiv  (seq (set s) (require p))  (seq (require p) (set s))).

set_ensure : forall (p:Prop) (s:St),
  (equiv  (seq (set s) (ensure p))  (seq (ensure p) (set s))).
```

A nice DSL is a monadic theory (by construction) where Hoare logic specification can be expressed, and such the corresponding wp-calculus can be expressed. Let us illustrate this point for refinement in state-handling monad.

In state-handling monad, it is convenient to have a notion of refinement in a environment (as for equivalence).

```
Definition refInEnv (st:St) (A:Type) (s1 s2: DS A)
 :=  (refines (seq (set st) s1)
              (seq (set st) s2)).
Implicit Arguments refInEnv [A].
```

Using `refInEnv`, we can express the *precondition* of a specification. Given a specification `s`, the precondition of `s` is a predicate on the initial state `sti` expressing that if `s` starts in the state `sti`, then it can not abort. In other word, `pre s sti` is satisfied when `(seq (set sti) s)` returns some results.[9]

```
Definition pre (A : Type) (s : DS A) (sti : St) : Prop :=
  refInEnv sti s
    (bind (any St)
       (fun stf => seq (set stf) (any A))).
Implicit Arguments pre [A].
```

On simple operators, it is straightforward to check that `pre` satisfies the following properties.

```
Lemma pre_val : forall (st:St) (A:Type) (a:A), pre (val a) st.

Lemma pre_any : forall (st:St) (A:Type), pre (any A) st.

Lemma pre_require : forall (st:St) (p:Prop), p -> pre (require p) st.

Lemma pre_ensure : forall (st:St) (p:Prop), pre (ensure p) st.

Lemma pre_set : forall (sti stf: St), pre (set stf) sti.

Lemma pre_get : forall (st : St), pre get st.
```

[8]We have actually only show this result for the state-handling and the exception-handling monads

[9]Let us remark here, that we can still have here an interpretation of `pre` in terms of partial correctness. At this level, `pre` does not implies termination, because `s` can *magically* terminates. In particular, the precondition of `magic` is still true, even if `magic` can be interpreted as non-termination. The chapter 3 detail how deal with total-correctness.

To compute recursively the precondition for `bind` operator, we need to introduce a notion of weakest-precondition. Here, we introduce the demonic liberal weakest-precondition. In other word, given a specification `s` of result type `A`, given a postcondition `p` (a predicate on the result and the final state), `(demon A s p)` is a precondition on the initial state `sti` guaranteeing that if `(pre s sti)` then `p` holds.

```
Definition demon (A:Type) (s:DS A) (p: St->A->Prop) (sti:St) :=
  (refInEnv sti
            (bind s (fun a => bind get (fun s => seq (require (p s a)) (val a))))
            s).
Implicit Arguments demon [A].
```

or equivalently:

```
Definition demon_def2 (A:Type) (s:DS A) (p: St->A->Prop) (sti:St) :=
  (refInEnv sti
            s
            (bind s (fun a => bind get (fun s => seq (ensure (p s a)) (val a))))).
```

Thus, `pre` can be computed using the following properties:

```
Lemma pre_bind :
 forall (sti : St) (A B : Type) (s1 : DS A) (s2 : A -> DS B),
   pre s1 sti /\ demon s1 (fun st a => pre (s2 a) st) sti ->
      pre (bind s1 s2) sti.

Lemma val_demon : forall (st : St) (A : Type) (a : A) (p : St -> A -> Prop),
   p st a -> demon (val a) p st.

Lemma bind_demon:
 forall (sti:St) (A B:Type) (s1: DS A) (s2: A -> DS B) (p:St -> B -> Prop),
   demon s1 (fun st a => demon (s2 a) p st) sti ->
     demon (bind s1 s2) p sti.

Lemma any_demon: forall (st:St) (A:Type) (p: St -> A -> Prop),
   (forall a : A, p st a) -> demon (any A) p st.

Lemma require_demon: forall (st:St) (p:Prop) (q:St -> unit -> Prop),
   q st tt -> demon (require p) q st.

Lemma ensure_demon: forall (st:St) (p: Prop) (q: St -> unit -> Prop),
   (p -> q st tt) -> demon (ensure p) q st.

Lemma set_demon: forall (sti stf: St) (p: St -> unit -> Prop),
   p stf tt -> demon (set stf) p sti.

Lemma get_demon : forall (st: St) (p : St -> St -> Prop),
  p st st -> demon get p st.
```

Let us remark, that if we restrict the state-handling monad constructs: `val`, `bind`, `set` and `get`, we see that `demon` corresponds exactly to the standard weakest-precondition calculus.

The operator `demon` satisfies another important property very classical for weakest-precondition calculi:

```
Lemma demon_weaken : forall (sti: St) (A: Type) (s: DS A) (p q: St -> A -> Prop),
 (forall (stf : St) (a : A), p stf a -> q stf a)
   ->  demon s p sti  -> demon s q sti.
```

Now, we show how to embed Hoare Logic specification. Given a predicate `q: St -> A -> Prop`, we define a specification choosing a final state `stf` and a result `a` such `(q st a)` holds.

```
Definition absPost (A : Type) (q : St -> A -> Prop) : DS A :=
   bind (any St) (fun stf =>
     seq (set stf)
         (bind (any A) (fun a =>
             seq (ensure (q stf a)) (val a)))).
Implicit Arguments absPost [A].
```

We can prove that:

```
Lemma absPost_refined: forall (A: Type) (s: DS A) (q: St -> A -> Prop) (sti: St),
 (pre s sti /\ demon s q sti) -> refInEnv sti s (absPost q).
```

In other word, given a specification `s` of result type `A`, a precondition `p` of type `St->Prop` and a postcondition `q` of type `St->St->A->Prop`, we can interpret a proof of the Hoare triple "`{ p } s { q }`" as a proof of

```
refines s (bind get (fun sti => (seq (require (p sti)) (absPost (q sti)))))
```

To prove such a goal, it is sufficient to prove:

```
forall (sti:St), (p sti) ->   ((pre s sti)  /\  (demon s (q sti) sti)).
```

In particular, if we are interested in proving partial expressions with state-handling, such expressions can be expressed in a sub-language of specifications restricted to `val`, `bind`, `set`, `get` and `require` (to express partiality). Proofs of Hoare Logic for such expressions can thus be directly embed into the refinement theory presented here.

2.3.2 Semantics of DSL

In this paper, I study three kinds of Coq semantics for Dijkstra Specification Languages:

- An algebraic (or axiomatic) semantics, as introduced above, based on *refinement monads*.
- A (direct) denotational semantics, associating to each specification a pair of pre and post condition.
- A CPS semantics expressing a wp-calculus.

The first of this semantics is generally not equivalent to the two others (which can be proved equivalent). In particular, the algebraic semantics describe a class of languages: the models of the algebraic theory. Whereas the others *implements* a particular language (or model).

The semantics using pre/post gives actually simple models to refinement monads. For instance, for a pure refinement monad, it is straightforward to show that the following definitions satisfy the axioms:

```
Record Spec (A:Type): Type := {  pre: Prop ;  post: A -> Prop  }.
```

```
Definition DS : Type -> Type := Spec.

Definition refines (A: Type) (s1 s2: DS A): Prop
 := pre s2 -> pre s1 /\ (forall a : A, post s1 a -> post s2 a).

Definition val (A:Type) (a:A) : (DS A) := Build_Spec True (fun b => a = b).

Definition bind (A B:Type) (s1:DS A) (s2:A -> DS B): (DS B) :=
  Build_Spec (pre s1 /\ (forall a : A, post s1 a -> pre (s2 a)))
             (fun b => exists a : A, post s1 a /\ post (s2 a) b).

Definition any (A : Type) : (DS A) := Build_Spec True (fun a : A => True).

Definition require (p : Prop) : (DS unit) := Build_Spec p (fun _ => True).

Definition ensure (p : Prop) : (DS unit) := Build_Spec True (fun _ => p).
```

This model is intuitive and simple, but not very interesting in practice. Formulas produced for specifications are huge, and there is no partial evaluation of deterministic specifications.

2.3.2.1 Weakest-Precondition semantics

The weakest-precondition semantics is more powerful, because it expresses exactly in Coq the computational meaning of specifications. But due to the presence of non-determinism, there are two notions of weakest-preconditions: a notion reflecting the user point of view on the specification, and a notion reflecting the developer point of view. These point of view are expressed below using the pre/post semantics of specifications. Hence, let us assume `A` a type, `s` a specification returning a result of type `A` (i.e `s:(Spec A)`).

The user point of view of weakest-preconditions is *demonic*: she has no idea about how `s` is implemented, and which behavior it chooses on non-deterministic parts. Thus, given a postcondition `R` on the result of `s`, the *strong demonic weakest-precondition* of `s` for `R` is the necessary and sufficient condition which must hold for `R` to be satisfied however `s` is implemented. The strong demonic wp of `s` is `(sdemon s)` of type `(A->Prop)->Prop` and satisfies [10]:

```
forall (R:A->Prop),
    (sdemon s R) <-> ((pre s) /\ forall (a:A), (post s a)->(R a))
```

The developer point of view is slightly different and is said to be *angelic*: she knows the ways to implement `s`, and she would like to know how the precondition of `s` should be enforce in order to ensure that `R` holds. In other words, assuming that `(pre s)` holds, the *angelic weakest-precondition* of `s` for `R` is the necessary and sufficient condition which ensures that there exists a way to implement `s` such `R` holds on the results for this particular implementation.

```
(pre s)
  -> forall (R:A->Prop), (angel s R) <-> (exists a:A, (post s a) /\ (R a))
```

Let us remark, that with these definitions, we have the following properties:

```
  (sdemon s (fun _ => True)) <-> (pre s)
```

and

```
(pre S)
    -> forall (a:A), (angel s (fun b => (a=b)))<->(post s a)
```

[10] In subsection 2.3.1, we have introduce the liberal demonic wp `(demon s)`. The relation between `sdemon` and `demon` is: `forall (R:A->Prop), (sdemon s R)` $\Leftrightarrow$ `((pre s)` $\wedge$ `(demon s R))`

In classical logic, these two notions are dual using excluded-middle. For instance, `(angel s R)` could be defined as `not (sdemon s (fun a => (not (R a))))`.

In intuitionistic logic, these two notions are not dual. But, an interest of intuitionistic logic, is the ability to express and in a general way what is a "computable" specification. For instance, in intuitionistic logic, a specification `s` such `(pre s)->(angel s (fun a => True))` is said to be *valid*: if `(pre s)` holds, there exists a computable value `a:A`, such `(post s a)` is satisfied.

Actually, to simplify the definition of `angel` and have better properties, we assume `angel` to satisfy the following property:

```
forall (R:A->Prop), (angel s R) <-> (exists a:A, (angel s (fun b => a=b) /\ (R a)))
```

Hence, it can be proved that `angel` is monotonic (and `sdemon` also):

```
forall (R1 R2:A->Prop),
 (forall (a:A), (R1 a) -> (R2 a))
   (angel s R1) -> (angel s R2)
```

Instead of defining specifications using a `pre`/`post` pair, we can define them using a pair `sdemon`/`angel`. Hence a specification of type `(Spec A)` is now a pair `sdemon : (A->Prop)->Prop` and `angel : (A->Prop)->Prop` such:

```
  forall R : A -> Prop,
  (sdemon R <-> sdemon (fun _ => True) /\ (forall a : A, angel (eq a) -> R a))
  /\ (angel R <-> (exists a : A, angel (eq a) /\ R a)).
```

Actually, with such a definition, we can prove directly (without introducing abstract syntax for specifications) all standard properties of the wp-calculus. We detail this point in section 4.2 of chapter 4.

Refinement can be either defined, by translating the pre/post semantics:

```
(refines s1 s2)
   <-> ((sdemon s2 (fun _ => True)) -> (sdemon s1 (fun a => angel s2 (eq a)))).
```

But this definition is also equivalent to:

```
(refines s1 s2)
 <-> (forall R : A -> Prop,
         (sdemon s2 R -> sdemon s1 R) /\ (pre s2 -> angel s1 R -> angel s2 R).
```

At last, basic constructions of pure Dijkstra specifications can be coded such that:

```
forall (a:A) (R: A-> Prop),
   (sdemon (val a) R) = (R a)
/\ (angel (val a) R) = (R a).

forall (s1:(Spec A)) (s2:A->(Spec B)) (R: B->Prop),
   (sdemon (bind s1 s2) R) = (sdemon s1 (fun a:A => (sdemon (s2 a) R)))
/\ (angel (bind s1 s2) R) = (angel s1 (fun a:A => (angel (s2 a) R))).

forall (R: A->Prop),
    (sdemon (any A) R) = forall a:A, (R a)
/\  (angel (any A) R) = exists a:A, (R a).
```

```
forall (P:Prop) (R: unit->Prop),
    (sdemon (require P) R)=P /\ (R tt)
/\  (angel (require P) R)=(R tt).

forall (P:Prop) (R: unit->Prop),
    (sdemon (ensure P) R)=P -> (R tt)
/\  (angel (ensure P) R)=P /\ (R tt).
```

We see that on the deterministic part of the language (here `val` and `bind`), `sdemon` and `angel` perform deterministic computations. It is quite easy to prove that all these definitions are well-formed (they satisfy the constraints on `angel` and `sdemon`). It is also easy to prove that these definitions satisfy the axioms of refinement monad.

2.3.2.2 Small-steps simplifications in refinement proofs, using WP

We have just introduced how refinement is related to wp-calculus. Having a formula of refinement, it is thus possible to "simplify" it using the the equivalence below, and computing wps. As for the lemma `le'_le` of the Coq presentation (see page 1.2.4.2), the right-to-left direction of this equivalence allows "simplify" a refinement goal using wp-calculus.

```
(refines s1 s2)
   <-> ((sdemon s2 (fun _ => True)) -> (sdemon s1 (fun a => (angel s2 (eq a))))).
```

However, when the formula computed by wp is not trivially provable, it is practically not tractable (especially if `s1` or `s2` are defined by induction like `lookup`). Actually, we would like "small step simplifications" relating refinement formulas and wps.

For instance, to simplify proofs involving `bind` construction, we have the three following properties:

```
Lemma bind_simpl_left :
 forall (A B : Type) (s1 : Spec A) (s2 : A -> Spec B) (s3 : Spec B),
 sdemon s1 (fun a : A => refines (s2 a) s3)
  -> refines (bind s1 s2) s3.

Lemma bind_simpl_right :
 forall (A B : Type) (s1 : Spec A) (s2 : A -> Spec B) (s3 : Spec B),
  (pre s1 -> angel s1 (fun a => refines s3 (s2 a)))
   -> refines s3 (bind s1 s2).

Lemma bind_simpl_both :
 forall (A B : Type) (s1 s3 : Spec A) (s2 s4 : A -> Spec B),
  refines s1 s3
   -> (pre s3 -> sdemon s3 (fun a => refines (s2 a) (s4 a)))
  -> refines (bind s1 s2) (bind s3 s4).
```

The first formula `bind_simpl_left` indicates that in order to prove a formula of the form "`(refines (bind s1 s2) s3)`", we can prove "`(sdemon s1 (fun a:A => refines (s2 a) s3))`". If `s1` is simple enough, this last formula is simplified in formula where `s1` or `sdemon` do not appear any more. Hence, this lemma allows to perform a kind of partial evaluation of `s1` into the goal. The two others lemmas correspond respectively to a simplification of `bind` in the right part of the refinement formula, or in both part of the refinement formula.

2.3.3 Expressive power of DSL

In this section, we illustrate that using refinement is more expressive than both Hoare Logic and equational reasoning in monads. In particular, refinement allows to express specifications that mixes abstract and effective descriptions, in a much more convenient way than equational reasoning. But of course, equational properties can still be expressed exactly in the same way.

2.3.3.1 A dynamic calculus of pre and post conditions

A main interest of refinement theory with respect to monads, is the ability to put preconditions in programs. For instance, we can rewrite the preceding `decrN` example, with a precondition to forbid decrementing the variable when it is `O`.

```
Definition decrN: (DS nat->(DS unit)) :=
 (lambda (fun n:nat => forN n (fun i =>
                               bind get (fun m =>
                                    match m with
                                    | O    => require False
                                    | (S p) => set p
                                    end)))).
```

It is then possible to prove that:

```
Lemma decrN_spec:
 forall (n:nat), (refines (apply decrN (val n))
                         (bind get (fun m =>
                               (seq (require (le n m))
                                    (set (minus m n)))))).
```

On this example, we see that specifications have a dynamic meaning: setting a precondition at a control point of the program will have an influence on the global precondition of the program.

Hence, `require` behaves like a kind of run-time assertion. But, `require` is much more expressive than a run-time assertion, since we can use `require P` where `P` may be any undecidable property.

2.3.3.2 Grey-box specifications

Coming back to the preceding `lookup` example, we can express `lookup_non_pur_accept` in a more abstract way (i.e. less precise): we do not precise below which element of the list is returned. We specify here, that if there is an element in the list satisfying `p`, then `lookup` chooses a natural `n` and an element `a` such `a` is the n-th element of the list and satisfies `p`, and repeats `(S n)` times `k`, before to return `a`.

```
Lemma lookup_non_pur_accept_2:
 forall (A:Set) (test:A->(DS bool)) (p: A -> bool) (k:(DS unit)),
  (forall (a:A), (equiv (test a) (seq k (val (p a)))))
  -> forall (l:(list A)),
       (refines (lookup test l)
                (seq (require (exists a:A, (In a l) /\ (p a)=true))
                     (bind (any nat) (fun n =>
                           (seq (forN (S n) (fun _ => k))
                                (bind (any A) (fun a =>
                                      (seq (ensure ((p a)=true /\ (Is_nth n a l)))
                                           (val a))))))))
```

Hence, on the contrary to equational reasoning in monads, it is easy here to abstract behaviors using non-determinism. But on the contrary to standard Hoare Logic, these abstractions can be mixed with computational behaviors.

2.3.3.3 High-order refinement specifications

We can also establish the preceding result for a more general condition on `test`, replacing "`equiv`" by "`refines`" in the hypothesis "`(equiv (test a) (seq k (val (p a))))`". This condition can itself be put in a requirement clause. Hence, this lemma can be expressed as a high-order refinement specification:

```
Lemma lookup_non_pur_accept_3:
  forall (A:Set) (test:A->(DS bool)) (p: A -> bool) (k:(DS unit)) (l:(list A)),
    (refines (lookup test l)
             (seq (require (forall (a:A), (refines (test a) (seq k (val (p a))))))
             (seq (require (exists a:A, (In a l) /\ (p a)=true))
                  (bind (any nat) (fun n =>
                        (seq (forN (S n) (fun _ => k))
                             (bind (any A) (fun a =>
                                   (seq (ensure ((p a)=true /\ (Is_nth n a l)))
                                        (val a)))))))))).
```

This property is said to be a high-order refinement property, because one of the specification in the refinement assertion contains itself a refinement assertion.

Let us remark that high-order specifications are not necessarily monotonic. For instance the preceding specification:

```
 (seq (require (forall (a:A), (refines (test a) (seq k (val (p a))))))
  ... )
```

is monotonic with respect to `test`, but not with respect to `k` !

Even worse, a high-order specification can be not compatible with equivalence. This is the case if we write for instance:

```
 (seq (require (forall (a:A), (test a)=(seq k (val (p a)))))
  ... )
```

2.3.3.4 Dealing with potential non-termination, partial or total correctness

Dijkstra specification languages allow to express non-termination in a quite simple way. I present here informally the main ideas on a `while` operator. I define the `while` as the following:

```
Definition while (cond:DS bool) (body:DS unit) : DS unit
 := bind (any nat) (fun n =>
    seq (forN n (fun i =>
             (bind cond (fun b => seq (ensure (b = true)) body))))
        (bind cond (fun b => ensure (b = false)))).
```

In this definition, both `cond` and `body` may produce side-effects. The main properties of `while` are the following. First, it is a fixpoint [11] :

```
Lemma while_fixequiv: forall (cond:DS bool) (body:DS unit),
 (equiv (while cond body) (If cond (seq body (while cond body)) skip)).
```

[11] In order to prove this lemma, the `any_on_non_empty_type_can_be_lifted` property is needed, see page 54 section 2.4.1 below

And it is actually the most magic fixpoint (or smallest fixpoint with respect to `refines` order relation) to satisfy the preceding equation:

```
Lemma while_refines_fixspec: forall (cond:DS bool) (body fixspec: DS unit),
 refines (If cond (seq body fixspec) skip) fixspec
  -> refines (while cond body) fixspec.
```

Assuming to be on a state-handling DSL, we can derive from `while_refines_fixspec` lemma, the standard Hoare Logic rule for `while` in partial correctness. Assuming an invariant on the global state `inv: St -> Prop`, the abstract specification of `(while cond body)` is

```
Definition while_abs (cond:DS bool) (body:DS unit) (inv:St -> Prop): (DS unit)
 := (seq (bind get (fun st => require (inv st)))
      (seq (bind (any St) (fun tmp_st => set tmp_st))
           (bind cond (fun b =>
              bind get (fun new_st =>
                ensure ((inv new_st) /\ b=false)))))).
```

A necessary condition ensuring that `(while cond body)` refines `(while_abs cond body inv)` is the following:

```
Definition inv_preservation (cond:DS bool) (body:DS unit) (inv:St -> Prop): Prop :=
  forall (st:St), (inv st) ->
    refInEnv st (If cond body skip) (absPost (fun new_st result => (inv new_st))).
```

In other word, we can deduce from `while_refines_fixspec` the following lemma (using `while_abs` as `fixspec`) :

```
Lemma while_refines_while_abs:
 (inv_preservation cond body inv)
    -> (refines (while cond body) (while_abs cond body inv)).
```

Let us remark that this lemma is a bit more general than the standard Hoare Logic rule, since `cond` appears in `while_abs` (and hence, `while_abs` does not correspond to a Hoare specification). This choice has been made here in order to allow `cond` to write in the state. By forbidding `cond` to write in the state, with the following additional hypothesis:

```
forall (st:St), (demon (seq (set st) cond) (fun st' r' => st=st'))
```

we could derive the standard Hoare Logic rule. But it does not seem necessary in practice.

Let us remark however, that the lemma `while_refines_while_abs` may be not enough general, when dealing with an exception-handling DSL. Indeed, the `inv_preservation` hypothesis forbids `cond` and `body` to raise an exception (and thus `(while cond body)` do not raise an exception also). It could be interesting to relax this constraint, and have a more general rule.

The preceding definition of `while` does only ensure partial correctness: in particular, even if `cond` and `body` *"terminate"*, `while` may not *terminate*. Can we also find a setting for total correctness? The answer is yes, but to give a sense this question, we must first precise what is a *terminating specification*.

Termination is associated to a notion of computation. Thus, in order to define formally a notion of termination, we may imagine to use the notion of computation associated to weakest-preconditions. More formally, on the state-handling DSL, we can state that a specification `s: (DS A)` is *weakly-terminating* if and only if [12]

```
forall (st:St), (pre s st) -> (angel s (fun _ _ => True) st).
```

Indeed, because of the intuitionistic semantics of Coq, this definition ensures that for all `st:St` such `(pre s st)`, then there exists a computable value `a` and a computable state `st'`, such that `(post s st st' a)`.

However, the notion of weak-termination may be not very satisfying, because in presence of non-determinism, it does not oblige `s` to "terminate" whatever its internal choices. Consider, for instance, the following specification in a informal syntax. It satisfies this notion of termination, but if we interpret it as a non-deterministic program, it may not always terminate.

```
x:=5 ;
while x > 0 do
  bind y = (any true) in
  if y then x:=x-1 else skip
done
```

A simple solution is to forbid non-determinism. For instance, we can state that a specification `s: (DS A)` is *deterministic* if and only if

```
forall (st:St), (pre s st) ->
  exists st':St, exists a:A, forall (R:St->A->Prop), (angel s R st) <-> (R st' a).
```

Hence, if `s` is deterministic, for all `st:St` such `(pre s st)`, there exist a unique `a` and a unique `st'`, such `a` and `st'` are computable and such `(post s st st' a)`.

However, it could be interesting to prove that the specification below is always terminating, setting `x` to a natural below or equals to 3. The `(any nat)` in this specification could be refined as a keyboard input or as a pseudo-random generation.

```
x:=10 ;
while x > 3 do
  bind y=(any nat) in
  if y <= 2 then x:=y else x:=x-1
done
```

In chapter 3, I show how to interpret a sub-language of the state-handling DSL into a non-deterministic programming language. Roughly speaking, this sub-language contains a `while` operator and all basic constructions of the DSL, including `require` and `any`, but excluding `ensure`. Furthermore, `any` is restricted to non-empty types, in order to avoid magic behavior. For this language, I define a non-deterministic operational semantics (the non-determinism comes from the reduction rule of `any`). This allows to introduce a notion of *strong-termination*. Informally, a specification `s` in the sub-language strongly-terminates if and only if for all states `st`, there is no infinite computations (in the operational semantics) from the specification `s` in the initial state `st`. In other word, whatever are the non-deterministic choices in the evaluation of the program, computations terminates. By extension, a specification `s` is said strongly-terminating if and only if it is equivalent to a strongly-terminating specification of the sub-language.

I now introduce the `whileWF` operator, which can be used, for expressing total-correctness, for any of the above notion of termination: weak-termination, determinism or strong-termination.

[12] On the state handling monad, the angelic weakest-precondition is of type `(DS A)->(St->A->Prop)->St->Prop`

Given a relation `R: St->St->Prop`, we define here the property stating that this relation is a variant that decreases at each step of the loop:

```
Definition variant_decreasing (cond:DS bool) (body:DS unit)
                             (inv: St->Prop) (R: St->St->Prop)
 : Prop :=
   forall (st:St), (inv st) ->
     refInEnv st (bind cond (fun b => (seq (ensure (b=true)) body)))
                 (absPost (fun new_st r => (R new_st st))).
```

The `whileWF` operator is therefore defined by:

```
Definition whileWF (cond:DS bool) (body:DS unit)
                   (inv: St->Prop) (R: St->St->Prop)
 : DS unit :=
  (seq (bind get (fun st =>
             require ((inv_preservation cond body inv)
                       /\ (variant_decreasing cond body inv R)
                       /\ (well_foundedT R) /\ (inv st))))
       (while cond body)).
```

A fundamental property of the `whileWF` operator is the following:

```
Theorem P_whileWF_require:
  forall (cond:DS bool) (body:DS unit) (inv: St->Prop)
                          (R: St->St->Prop) (P: (DS unit) -> St -> Prop),
   (forall (st:St) (s1 s2:DS unit), (P s1 st) -> (eqInEnv st s1 s2) -> (P s2 st))
    -> (forall (st:St) (Q: St->Prop) (s:DS unit),
          ((Q st) -> (P s st))->(P (seq (require (Q st)) s) st))
     -> (forall (ws: (DS unit)),
             (forall (st:St), (P ws st))
             -> forall (st:St), (inv st) -> (P (If cond (seq body ws) skip) st))
   -> forall (st:St), (P (whileWF cond body inv R) st).
```

This property expresses a general invariant property about `whileWF`. Mainly, given a predicate `P: (DS unit)->St->Prop`, if `P` is compatible with `eqInEnv`, and if `P` is compatible with the notion of precondition, and if `P` is preserved by a step of the while-loop, then `P` holds for `(whileWF cond body inv R)`. In particular, we can directly deduce from `P_whileWF_require` the following lemma, showing that `whileWF` is a most abortive fixpoint (biggest fixpoint for refines order):

```
Lemma fixspec_refines_whileWF:
 forall (cond:DS bool) (body fixspec: DS unit) (inv: St->Prop) (R: St->St->Prop),
  refines fixspec (seq (bind get (fun st => require (inv st)))
                  (If cond (seq body fixspec) skip))
  -> refines fixspec (whileWF cond body inv R).
```

This property is proved by instantiating `P_whileWF_require` with `P` as

```
fun (s:DS unit) (st:St) => refInEnv st fixspec s.
```

Actually, `P_whileWF_require` allows also to prove that `whileWF` preserves weak-termination and determinism. Indeed, each of these notions is a property on specifications satisfying the hypothesis of the `P_whileWF_require`. For strong-termination however, the proof is more complex (see chapter 3).

In chapter 3, I propose a general fixpoint theory for DSL that extends the `while` and `whileWF` constructions presented here. Furthermore, I prove (in COQ) the correctness of DSL refinement semantics with respect to the non-deterministic programming language suggested above.

In conclusion, we see that DSL are sufficiently expressive to express non-termination, both with a partial-correctness and total-correctness point-of-view. In partial-correctness, non-termination corresponds to a magic behavior, while in total-correctness it is an abortive behavior[13]. The other noticeable aspect seen in this section, is that `while` and `whileWF` can be coded as high-order operators directly expressed in DSL. In other word, to express non-termination in both settings, the underlying weakest-precondition calculus does not need to be modified! This shows the great expressibility of high-order refinement calculus.

2.3.3.5 Lattice structure in DSL and comparison with Back's refinement calculus

Considering `refines` as an order, we can provide a lattice structure to DSL. We have already seen `magic` and `abort` which are respectively the absolute minimum and the absolute maximum of this order.

Given two specifications `s1` and `s2` of type `(DS A)`, we can build (in the algebraic semantics), the least upper bound called `or s1 s2` of type `(DS A)` as

```
Definition or (A : Type) (s1 s2 : DS A) : DS A :=
  If (any bool) s1 s2.
```

This operator satisfies the three following properties, showing that it is a least upper bound:

```
Lemma refines_or_intro_l : forall (A : Type) (s s1 s2 : DS A),
     refines s s1 -> refines s (or s1 s2).

Lemma refines_or_intro_r : forall (A : Type) (s s1 s2 : DS A),
   refines s s2 -> refines s (or s1 s2).

Lemma refines_or_elim : forall (A : Type) (s1 s2 s : DS A),
   refines s1 s -> refines s2 s -> refines (or s1 s2) s.
```

Of course, as usually, commutativity and associativity of `or` can be derived from these least upper bound property. In the algebraic semantics, `bind` satisfies the following distributivity rules over `or` :

```
Lemma bind_or_distr_r : forall (A B : Type) (s1 s2 : DS A) (s3 : A -> DS B),
 equiv (bind (or s1 s2) s3) (or (bind s1 s3) (bind s2 s3)).

Lemma bind_or_distr_l_2: forall (A B:Type) (s: DS A) (s1 s2: A -> (DS B)),
   (refines (or (bind s s1) (bind s s2)) (bind s (fun a => (or (s1 a) (s2 a))))).
```

If we admit the `any_on_non_empty_type_can_be_lifted` axiom (page 54), the converse property of this last one is provable,

```
Lemma bind_or_distr_l_1: forall (A B:Type) (s: DS A) (s1 s2: A -> (DS B)),
   (refines (bind s (fun a => (or (s1 a) (s2 a)))) (or (bind s s1) (bind s s2))).
```

[13] That explains why in a total-correctness setting like B, `pre` is also called `trm` for *"termination"*

This `or` operator is fundamental on a exception-handling DSL, because it allows to express Hoare-Logic specifications. Indeed, when dealing with exceptions, Hoare-Logic specifications have an additional postcondition about the potentially raised exceptions. For instance, if this postcondition is false, then no exception should be raised. If it is true, then any exception can be raised, but no exception may be raised, if the *normal* postcondition authorizes a result. Hence, on a purely-functional DSL with exception-handling, the postcondition part of Hoare-Logic specifications can be expressed as

```
(or (bind (any A) (fun a => seq (ensure (Q1 a) (val a))))
    (bind (any Exc) (fun e => seq (ensure (Q2 e) (raise e)))))
```

where `Exc` is the type of exception, `Q1: A->Prop` the *normal* postcondition, and `Q2: Exc->Prop` the *exceptional* postcondition.

At this step, we see that every DSL is semi-lattice. We can also build a greatest lower bound operator `and`. However, we have not a generic operator over DSLs like the preceding ones. Each DSL must thus define its own operator `and`. We consider here only the purely-functional case. In the purely-functional DSL, we define `and` operator as

```
Definition and (A: Type) (s1 s2:DS A): DS A :=
    (seq (require ((pre s1) \/ (pre s2)))
         (bind (any A) (fun a =>
               seq (ensure (((pre s1) -> (post s1 a)) /\
                            ((pre s2) -> (post s2 a))))
                   (val a)))).
```

It satisfies the following properties of greatest lower bounds:

```
Lemma and_elim_l: forall (A:Type) (s1 s2 s:DS A),
  (refines s1 s) -> (refines (and s1 s2) s).

Lemma and_elim_r: forall (A:Type) (s1 s2 s:DS A),
  (refines s2 s) -> (refines (and s1 s2) s).

Lemma and_intro: forall (A:Type) (s1 s2 s:DS A),
  (refines s s1) -> (refines s s2) -> (refines s (and s1 s2)).
```

Let us remark, that `bind` does not distribute over this operator very well. We have only the following properties:

```
Lemma bind_and_distr_r_1: forall (A B:Type) (s1 s2: DS A) (s: A -> (DS B)),
   (refines (bind (and s1 s2) s) (and (bind s1 s) (bind s2 s))).

Lemma bind_and_distr_l_1: forall (A B:Type) (s: DS A) (s1 s2: A -> (DS B)),
   (refines (bind s (fun a => (and (s1 a) (s2 a)))) (and (bind s s1) (bind s s2))).
```

The converse of property `bind_and_distr_r_1` is false. For instance, `s1` being `(val 0)`, `s2` being `(val 1)` and `s` being **fun** `x => (require (x=0))` Then `(bind (and s1 s2) s)` is equivalent to `(magic unit)` whereas `(and (bind s1 s) (bind s2 s))` is equivalent to `skip`.

The converse of property `bind_and_distr_l_1` is also false. For instance, `s` being `(any nat)`, `s1` being **fun** `x => (ensure (x=0))` and `s2` being **fun** `x => (ensure (x=1))`, then

```
bind s (fun a => (and (s1 a) (s2 a)))
```

is equivalent to `(magic unit)` where as `(and (bind s s1) (bind s s2))` is equivalent to `skip`.

We recover here the basic operators of Back and von Wright [BvW98]. In particular, the lattice structure of DSL allows to apply Tarski's theorem: smallest and greatest fixpoints exist for every monotonic transformation. (see chapter 3 for details). Thus, it seems that refinement calculus can be derived from intuitionistic logics without loss of fundamental properties.

On the contrary, constructive logic give to my theory two important aspects which are not present in Back's one. First, I have a powerful notion of structural recursion directly included in the weakest-precondition calculus. Second, I can give a simple meaning to high-order specifications and high-order refinement. It seems that having high-order procedures in refinement calculi based on classical logic requires to a complex encoding (see [Utt92] pages 73-75).

2.4 Embedding refinement calculus in type theory

In this section, I try to report my experiment of implementing a refinement prover in Coq. This experiment is itself detailed chapter 4. Coq sources can be found on [Bou05]. Here, I deal with the two following issues: how the three preceding semantics can be applied and how they should be improved in order to have a satisfying implementation.

From the implementation point-of-view, the pre/post semantics has no interest (but it is certainly the most intuitive semantics). The weakest-precondition semantics allows to apply small-steps (or big-steps) simplifications to refinement goals. Hence, how combine the algebraic semantics and the weakest-precondition semantics in a implementation?

2.4.1 Interest of the algebraic semantics

A big interest of the algebraic semantics is that it is sufficient to express the fixpoint theory developed chapter 3. Thus, this theory is directly applicable in any DSL.

The main interest of the algebraic semantics is that monads have been well studied, and in particular how models of specific monads can be combined to construct models for new specific monads, using for instance generic *monad transformers*. I will not detail here the theory of monad transformers, but only give the general picture.

2.4.1.1 Presentation of monad transformers

Informally, a monad transformer is a function from monads into monads, which extends the underlying models with specific operations. Monad transformers are thus generic extensions of monads.

Given `Core` a monad[14], a monad transformer for `Core` is a monad (called the *extern* monad) enriched with a `lift` operation

```
lift : forall A : Type, (Core.K A) -> K A.
```

satisfying the following properties:

```
lift_compat :
  forall (A : Type) (x y : Core.K A), Core.equiv x y -> equiv (lift x) (lift y).
lift_val :
  forall (A : Type) (a : A), equiv (lift (Core.val a)) (val a).
lift_bind :
  forall (A B : Type) (x : Core.K A) (f : A -> Core.K B),
    equiv (lift (Core.bind x f)) (bind (lift x) (fun a => lift (f a))).
```

[14] I define here implicitly monads as Coq modules, but they could also be defined as Coq records. I thus use the "dot notation" to access the operators of `Core`.

We can provide monad transformers for state-handling and exception-handling. Actually, the construction of these monad transformers is a generalization of the corresponding monads. For instance, the state-handling monad is defined by:

```
Definition K (A : Type)
 := St -> Core.K (prodT St A).

Definition equiv (A : Type) (x y : St -> Core.K (prodT St A))
 := forall s : St, Core.equiv (x s) (y s).

Definition val (A : Type) (a : A) (s : St)
 := Core.val (pairT s a).

Definition bind (A B : Type) (m : K A) (f : A -> K B) (s : St)
 := Core.bind (m s) (fun p => f (sndT p) (fstT p)).

Definition lift (A : Type) (m : Core.K A) (s : St)
 := Core.bind m (fun a => Core.val (pairT s a)).

Definition set (s : St) : K unit
 := fun _ => Core.val (pairT s tt).

Definition get : K St
  := fun s => Core.val (pairT s s).
```

This monad transformer has the following properties: `get` and `set` commute with lifted computations.

```
Lemma get_lift :
 forall (A B : Type) (m : Core.K A) (f : St -> A -> K B),
   equiv (bind get (fun s => bind (lift m) (fun a => f s a)))
         (bind (lift m) (fun a => M.bind get (fun s => f s a))).

Lemma set_lift :
 forall (A B : Type) (s : St) (m : Core.K A) (f : A -> K B),
   equiv (seq (set s) (bind (lift m) (fun a => f a)))
         (bind (lift m) (fun a => seq (set s) (f a))).
```

Monad transformers may be used as building blocks, which can be combined to build new monads. Indeed `lift` allows to embed some specific operators of the core monad into the extern monad. However, through this embedding some specific operators may loose their properties in the extern monad. For instance, the embedding of state-handling operators (`get` and `set`) preserve their properties, thus the extern monad is also a state-handling monad. But, the embedding of the error raising operator do not preserved its basic property. This impose some constraints on the order on which the transformers are applied.

As DSL are special cases of monads, it is possible to define models for them, *almost* automatically using such monad transformers. For instance, starting from a basic model of DSL (i.e. satisfying the properties mentioned above), it is possible to get a model for DSL with state handling, applying the usual state-handling transformer. And then, to have a model for exception-handling, applying the usual exception-handling transformer. Like for error raising, some properties of DSL are not automatically transmitted to the extern monad in application of monad transformers. Thus, this properties must be proved at "hand" knowing the implementation of the extern monad.

2.4.1.2 Drawbacks of the algebraic semantics

Actually, the algebraic semantics has also several drawbacks. First, if basic operators and basic properties are common to all DSL, their refinement theories are very different. Developing these theories is often very tedious because proofs are purely deductive (equational reasoning). Even if fixpoints can be defined generically over DSLs, derived rules for fixpoints must be inferred and proved at hand.

Second, algebraic semantics is not complete for reasoning on high-order programs. As mentionned section 2.3.3.4, properties of the high-order function `while` need the following extra-property stating that a `any` on a non-empty type can be permuted with any subexpression `s` that do not use its result.:

```
any_on_non_empty_type_can_be_lifted :
 forall (A B C : Type) (s : DS A) (f : A -> B -> DS C),
 (NonEmpty B) ->
  refines (bind s (fun a => bind (any B) (fun b => f a b)))
          (bind (any B) (fun b => bind s (fun a => f a b))).
```

Typically, this property is not needed in first order reasoning because `s` is then a ground term of a given monad, and this property is then provable by induction on ground terms. But, it is often needed in high-order reasoning, `s` being for instance the formal parameter of a function. We can still imagine to add this lemma as an axiom. But, a cousin of this property is still missing:

```
any_when_pre_can_be_lifted :
  forall (A B C : Type) (s : DS A) (f : A -> B -> DS C),
  refines s (any A) ->
  refines (bind s (fun a => bind (any B) (fun b => f a b)))
          (bind (any B) (fun b => bind s (fun a => f a b))).
```

With respect to the previous one, only the hypothesis is changing. This hypothesis could be written `(pre s)` if we lie in the "pure" DSL. But, in a state-handling DSL, this property needs to be generalized into:

```
any_when_pre_can_be_lifted:
  forall (A B C : Type) (s : DS A) (f : A -> B -> DS C),
  refines s (bind (any St) (fun stf => seq (set stf) (any A)))
   -> refines (bind s (fun a => bind (any B) (fun b => f a b)))
              (bind (any B) (fun b => bind s (fun a => f a b))).
```

Thus, it seems not easy here to expressed this property in a simple but sufficiently general way. A solution lies maybe in the work of Schröder and Mossakowski ([SM03]). Using monads, they express a "generic" Hoare proof system, independent of the programming language. Their system is proved to be sound, but not to be complete. However, their proof system looks stronger (but more complicated) than the one derived of my axiomatization.

Furthermore, as mentioned section 2.2.4.3, monad theory does not provide a sensible management of the universe of effects. Universes of effects are global in monads, whereas in real programming languages, they are local, in order to provide a real modularity. Hence, monads do not reflect the composition of effect universes. In practice, algebraic semantics is less interesting than it looks at first glance.

2.4.2 Overview of the current implementation

However, the idea of monad transformers can still be useful to design the architecture of the implementation, even if they are not formalized in the implementation. They give the idea to design the desired refinement theory by successive layer.

The first layer is defined by the module `FSL` (for Functional Specification Layer). It implements a purely functional DSL as suggested paragraph 2.3.2.1 page 42. The core of wp-calculi is defined at this level.

The second layer is `ISL` (for Imperative Specification Layer). It is the result of the application of the state-handling monad transformer to `FSL`. At this level, we do not need to know exactly how constructions of `FSL` are implemented. The wp-calculi of the `ISL` is implicitly simulated in `FSL` (via the state-handling monad transformation). Furthermore, all constructions of `FSL` are simply lifted into `ISL`, using the `lift` operation above for state-handling monad transformer. Implementation of `ISL` is thus relatively independent of the implementation of `FSL`, excepted in some proofs which use wp-simplifications of `FSL`.

It is planed to have a third layer `ESL` (for Exceptional Specification Layer) which will be the result of the application of the exception-handling monad transformer to `ISL`. Some experiments in the algebraic semantics show that adding exception raising, and exception catching present no technical difficulties.

In `ISL`, I have tried to generalize the monad approach in order to allow a local state space, without making the denotations of `ISL` too heavy with respect to their corresponding notations in imperative language. I propose a simple algebraic description of *places*, which allows to denote locations of the global state in `ISL`. Places can be implemented as operations on tuples. But this can be done outside of `ISL`. The idea is that the implementation of each places will be built by the translator from concrete syntax to denotations in Coq.

In `ISL` implementation, the main task is not to design the wp-computations. The main task is to find and prove the simplification rules (like those presented page 2.3.2.2) for refinement proofs in each layer. Indeed, on the contrary to provers for Hoare logic or for B, my wp-calculus is rather naive. But, as explained previously, I want to design a prover in which wp-simplifications can be invoked by small-steps during refinement proofs. The main work is to find which “small-steps” can be proposed to a user. For `ISL`, I have shown that powerful rules can be expressed when refinement goals are expressed using `refInEnv` instead of `refines`. For `ESL`, refinement goals must take an other form.

In `ISL`, there are some embryonic tactics which try combine some simplification rules in order to simplify the current goal. There is left a lot of work in order to get a usable tool.

2.4.3 Similarities & differences between type theory and refinement calculus

To conclude this chapter, I celebrate the marriage of type theory and refinement calculus, by emphasizing their complementarity, figure 2.1. Hence, refinement calculus could be for Coq what monads have been to Haskell: a pure way to interact with the impure side of the world.

Type theory	**Refinement calculus**
Suitable as logical foundations of mathematics	Assume the existence of a pre-existing logic
A logic for general theorem proving	A logic for program proving only
Handles only purely functional programs	Many programming paradigms may be supported
Specifications are abstract algorithms	
Programs are special cases of specifications	
Proofs mix deductions and computations	
Evaluation of specifications is lambda-calculus and rewriting rules	Evaluation of specifications is weakest-preconditions calculi
Deduction rules form a sequent-calculus	Deduction rules form a refinement-calculus

Figure 2.1: Complementarity of type theory and refinement calculus

Chapter 3

Fixpoints and total correctness

This chapter shows section 3.1 that Tarski's theorem can be applied in DSLs to build most magic (smallest) and most abortive (biggest) fixpoints. In section 3.2, I build a well-founded recursion operator `fixF`: the monotony hypothesis of Tarski is strengthened with a decreasing condition to ensure additional properties of fixpoints (unicity, preservation of determinism and weak-termination, ...). In section 3.3, deriving `whileWF` from `fixF`, I show that conditions of section 3.2 allow to establish the properties presented section 2.3.3.4. Then, I use these conditions to prove the total-correctness of a state-handling DSL with respect to a non-deterministic programming language. More precisely, this language allows first-order partially-defined expressions with non-determinism, state-handling and unbound iteration.

3.1 Tarski's theorem in DSLs

We have seen in chapter 2 how Coq structural recursion can be embedded in DSLs. In this section, I present a general theory to build fixpoints in DSLs. In fact, this is an application of Tarski's theorem.

I study here both recursive functions computing specifications, and recursive specifications. For instance, a while-loop is a special case of recursive specification.

3.1.1 Introduction

Let me study the following recursive equation. I consider here that `equiv` denotes the equivalence of refinement (both sides of the equation refines each other).

```
equiv even (or (val O) (bind even (fun n => val (S (S n)))))
```

First, this equation admits a solution: `(abort nat)`. Trivially, this is the most abortive solution (which is refined by all other solutions). Let me remark that `(magic nat)` is not a solution since `(val O)` does not refine `(magic nat)`. Second, this equation admits a most magic solution (which refines all other solutions): `(bind (any nat) (fun n => val (plus n n)))`.

Is this situation happening for any recursive equation on specifications[1] ? The answer is no. Let us consider the function `absurd_eq`

```
Definition absurd_eq (x:(DS unit)):=
 (seq (require (refines (abort unit) x))
 (seq (ensure (refines x (magic unit))
      x))).
```

[1] In a first step, I consider recursive equations such the left-side is a single variable

Let us assume a term `absurd:(DS unit)` such `(equiv absurd (absurd_eq absurd))`. We can then prove that `(abort unit)` refines `absurd` and that `absurd` refines `(magic unit)`. Then, `(abort unit)` refines `(magic unit)`. This is false.

Tarski's theorem ensures the existence of (smallest and biggest) fixpoints as soon as recursive equations are monotonic. A recursive equation is actually viewed as a function `F: (DS A) -> (DS A)` (where `A` is the type of the result), the parameter of the function being the recursive object to construct. The monotony condition is simply:

```
Hypothesis F_monotony: forall (x y:(DS A)), (refines x y) -> (refines (F x) (F y)).
```

3.1.2 Tarski's theorem in constructive type theory

In set theory, Tarski's theorem applies on complete lattice: preorders such that each subset E of the lattice admits a greatest lower bound and a least upper bound[2].

Here, we must adapt this notion of complete lattice to type theory. The notion of subset does not exist, but we can translate it by considering a predicate P instead, where P represents the subset of elements satisfying P.

Hence, if `A` is a type, `R` a relation on `A`, and `glb` of type (`A`→`Prop`) → `A`, the following structure represents a predicate defining `glb` has a greatest lower bound operator with respect to `R`.

```
Record is_glb (A:Type) (R: A → A → Prop) (glb: (A→Prop) → A) : Prop := {
  glb_lb: forall (P:A→Prop) (x:A), (P x) → (R (glb P) x) ;
  glb_glb: forall (P:A→Prop) (x:A),
     (forall (y:A), (P y) → (R x y)) → (R x (glb P))
}.
```

A lower semi-lattice is a preorder `R` with a greatest lower bound operator `glb`. By definition, `glb` is unique up to the equivalence associated to the preorder.

```
Record inf_semilattice (A:Type) (R: A → A → Prop) (glb: (A→Prop) → A) : Prop := {
  isemlat_preord:> (preorder R) ;
  isemlat_glb:> (is_glb R glb)
}.
```

On lower semi-lattice, Tarski's theorem ensures the existence of smallest fixpoints. I define below Tarski's smallest fixpoint operator `sfix` and its main properties in a Coq section.

```
Section Smallest_Fixpoints.

Variable A: Type.
Variable R: A → A → Prop.
Variable glb: (A → Prop) → A.

Hypothesis semi_lattice: (inf_semilattice R glb).

Section SFix_Def.
```

I denote `F` the function whose fixpoint is being computed. This function is assumed to be monotonic.

```
Variable F: A → A.
Hypothesis F_monotony: forall (x y:A), (R x y) → (R (F x) (F y)).
```

Below, I introduce the predicate `prefix` defining `x` as a prefixpoint.

[2] These bound may not be in the subset E

Definition `prefix (x:A) : Prop := (R (F x) x).`

The smallest fixpoint of `F`, denoted here `sfix`, is the greatest lower bound of the subset of prefix-points.

Definition `sfix:= glb prefix.`

By the `glb_lb` property of `glb`, `sfix` is lower than any prefixpoint `x`.
Lemma `sfix_smallest`: **forall** `(x:A), (R (F x) x) → (R sfix x).`

As a consequence, if `x` is a prefixpoint, then by `F_monotony`, `(F sfix)` is lower than `F x`. By transitivity, `(F sfix)` is lower than `x`. As this last proposition holds for any prefixpoint `x`, by the `glb_glb` property of `glb`, `(F sfix)` is lower than `sfix`. Thus, `sfix` is a prefixpoint.
Lemma `sfix_prefix: (R (F sfix) sfix).`

Applying `F_monotony` to `sfix_prefix`, we have that `(F sfix)` is a prefixpoint. Thus, by property `sfix_smallest`, `sfix` is lower than `F sfix`. Thus, `sfix` is a postfixpoint.
Lemma `sfix_postfix: (R sfix (F sfix)).`

End `SFix_Def.`

The fixpoint operator `sfix` is itself monotonic.
Section `SFix_monotony.`

Variable `F1 F2: A → A.`
Hypothesis `F1_monotony`: **forall** `(x y:A), (R x y) → (R (F1 x) (F1 y)).`
Hypothesis `F2_monotony`: **forall** `(x y:A), (R x y) → (R (F2 x) (F2 y)).`

Lemma `sfix_monotony`: (**forall** `(x:A), (R (F1 x) (F2 x))) → (R (sfix F1) (sfix F2)).`

End `SFix_monotony.`

End `Smallest_Fixpoints.`

I give a last property allowing to relate two fixpoints, when their equations are themselves related. Hence, given the following diagram:

$$\begin{tikzcd} \mathtt{A1} \arrow[r, "\mathtt{G}"] \arrow[d, "\mathtt{F1}"'] & \mathtt{A2} \arrow[d, "\mathtt{F2}"] \\ \mathtt{A1} \arrow[r, "\mathtt{G}"'] & \mathtt{A2} \end{tikzcd}$$

If the right path `F2∘G` is pointwise lower than the left path `G∘F1`, then `(sfix F2)` is lower than `(G (sfix F1))`. Section 3.1.4 give examples showing that this property is useful in proofs. An example also shows that when the left path is pointwise lower than the right path, then `(G (sfix F1))` may not be lower than `(sfix F2)`.

Section `SFix_Morph.`

Variable `A1 A2: Type.`
Variable `R1: A1 → A1 → Prop.`
Variable `R2: A2 → A2 → Prop.`
Variable `glb1: (A1 → Prop) → A1.`
Variable `glb2: (A2 → Prop) → A2.`

Hypothesis `semi_lattice1: (inf_semilattice R1 glb1).`

Hypothesis `semi_lattice2: (inf_semilattice R2 glb2).`

Variable `F1: A1 → A1.`
Hypothesis `F1_monotony:` **forall** `(x y:A1), (R1 x y) → (R1 (F1 x) (F1 y)).`

Variable `F2: A2 → A2.`
Hypothesis `F2_monotony:` **forall** `(x y:A2), (R2 x y) → (R2 (F2 x) (F2 y)).`

Variable `G: A1 → A2.`
Hypothesis `G_monotony:` **forall** `(x y:A1), (R1 x y) → (R2 (G x) (G y)).`

```
Lemma sfix2_R_G_sfix1:
 (forall (x: A1), (R2 (F2 (G x)) (G (F1 x))))
     → (R2 (sfix R2 glb2 F2) (G (sfix R1 glb1 F1))).
```

End `SFix_Morph.`

I now introduce biggest fixpoints using a dual principle. Given a preorder `R`, its dual is the preorder (**fun** `x y ⇒ R y x`).

```
Lemma dual_preorder:
  forall (A:Type) (R:A→A→Prop),
    (preorder (fun x y ⇒ R y x)) → (preorder R).
```

An upper semi-lattice is a preorder `R` with a least upper bound operator `lub`, that is a great lower bound operator for the dual of `R`.

```
Record sup_semilattice (A:Type) (R:A → A → Prop) (lub: (A→Prop) → A) : Prop := {
  ssemlat_preord:> (preorder R) ;
  ssemlat_lub:> (is_glb (fun x y ⇒ R y x) lub)
}.
```

Hence, an upper semi-lattice is a lower semi-lattice on the dual preorder.

```
Lemma dual_ssemilat:
  forall (A:Type) (R:A→A→Prop) (lub: (A→Prop) → A),
    (sup_semilattice R lub) → (inf_semilattice (fun x y ⇒ R y x) lub).
```

Thus, given an upper semi-lattice for a preorder `R` and a least upper bound `lub`, then the biggest fixpoint `bfix` is the smallest fixpoint for the dual: `sfix` (**fun** `x y ⇒ R y x`) `lub`.

3.1.3 Complete lattices in DSLs

In this subsection, I show how apply Tarski's constructions for recursive specifications and recursive functions on specifications.

3.1.3.1 Biggest fixpoints for specifications

In the algebraic semantics, for any type `A`, `refines` is a preorder on `(DS A)`. The least upper bound can be built by:

```
Definition lub (A:Type) (P:(DS A)->Prop) : (DS A)
 := bind (any (DS A)) (fun sp =>
          seq (ensure (P sp)) sp).
```

Let us remark that in this definition, we use a high-order feature of the language, since `(any (DS A))` is of type `(DS (DS A))`. Hence, we can instantiate the preceding `bfix` using:

```
Definition bfixDS (A:Type) (F: (DS A) -> (DS A)) : (DS A) :=
   bfix (refines (A:=A)) (lub (A:=A)) F.
```

Hence if `F` is monotonic, `(bfixDS F)` is a most abortive fixpoint.

3.1.3.2 Symmetric DSLs

In order to have a greatest lower bound, we need to extend the algebraic semantics of DSLs: this leads to the notion of *symmetric DSL*. A symmetric DSL is a DSL with a `sync` operator which satisfies:

```
sync: forall (A B:Type), (A -> (DS B)) -> (DS B).

sync_elim: forall (A B:Type) (s1: A -> (DS B)) (s2:(DS B)) (a:A),
  (refines (s1 a) s2) -> (refines (sync s1) s2).

sync_intro: forall (A B:Type) (s1:(DS B)) (s2: A -> (DS B)),
  (forall (a:A), (refines s1 (s2 a))) -> (refines s1 (sync s2)).
```

This operator is thus the dual of `fun (A B:Type) (s: A -> DS B) => bind (any A) s`. It is a generalization of `and` operator presented section 2.3.3.5.

"Standard" DSLs are actually symmetric. For instance, for purely functional DSL, `sync` is given by:

```
Definition sync (A B:Type) (s :A -> (DS B)) :=
  seq (require (exists a:A, (pre (s a))))
       bind (any B) (fun b =>
         seq (ensure (forall (a:A), (pre (s a)) -> (post (s a) b)))
             (val b)).
```

Or equivalently, we have the two following properties:

```
sync_pre: forall (A B:Type) (s: A -> (DS B)),
  (pre (sync s)) <-> (exists a:A, (pre (s a))).

sync_post:  forall (A B:Type) (s: A -> (DS B)) (b:B),
    (post (sync s) b) <-> forall (a:A), (pre (s a)) -> (post (s a) b).
```

In the imperative DSL, `sync` is trivially derived from the purely functional version:

```
Definition sync (A B:Type) (sp: A -> (DS B)) : (DS B)
 := fun (st:St) => FSL.sync (fun x:A => (sp x st)).
```

3.1.3.3 Smallest fixpoints for specifications

Using `sync`, construction of smallest fixpoints is very similar to construction of biggest fixpoints. The greatest lower bound is simply defined by:

```
Definition glb (A:Type) (P:(DS A)->Prop) : (DS A)
 := sync (fun sp => seq (require (P sp)) sp).
```

The smallest fixpoint operator is defined by:

```
Definition sfixDS (A:Type) (F: (DS A) -> (DS A)) : (DS A) :=
   sfix (refines (A:=A)) (glb (A:=A)) F.
```

Hence if `F` is monotonic, `(sfixDS F)` is a most magic fixpoint.

3.1.3.4 Fixpoints for functions

Let me assume `A` and `B` two types, and `F: (A -> DS B) -> (A -> DS B)` a monotonic function:

```
F_monotony:
  forall (f g: A -> (DS B)),
   (forall (x:A), (refines (f x) (g x)))
      -> forall (x:A), (refines (F f x) (F g x)).
```

`F` admits a smallest and a biggest fixpoint. Indeed, Tarski's theorem can be applied on type `A -> DS B`. I consider the preorder `refinesF` defined by:

```
Definition refinesF (f1 f2: A -> DS B) :=
  forall (x:A), refines (f1 x) (f2 x).
```

And greatest lower bound is given by:

```
Definition glbF (P:(A -> DS B)->Prop) (x:A) : (DS B)
 := sync (fun sp => seq (require (P sp)) (sp x)).
```

Symmetrically, least upper bound is given by:

```
Definition lubF (P:(A -> DS B)->Prop) (x:A): (DS B)
 := bind (any (A -> DS B)) (fun sp =>
          seq (ensure (P sp)) (sp x)).
```

In the following, I note `sfixF F` (respectively `bfixF F`) the smallest (resp. biggest) fixpoint of `F`.

3.1.4 Examples of recursive specifications

I now illustrate the operator `sfixDS` on some concrete examples. First, let me come back to the preceding example of `even`. I introduce the recursive equation:

```
Definition even_eq (ev: (DS nat))
 := (or (val O) (bind ev (fun n => (val (S (S n)))))).

Definition even : DS nat := sfixDS even_eq.
```

As `even_eq` is monotonic, `even` is a smallest fixpoint. Similarly, I define `odd` as a smallest fixpoint:

```
Definition odd : DS nat
  := sfixDS (fun od => or (val 1) (bind od (fun n => (val (S (S n)))))).
```

I can prove that:

```
Lemma odd_refines_Seven: refines odd (bind even (fun n => (val (S n)))).
```

Indeed, applying `sfix2_R_G_sfix1` with `G` being `fun x => bind x (fun n => (val (S n)))`, it is suffisant to prove:

```
forall x : DS nat,
refines
  (or (val 1)
      (bind (bind x (fun n : nat => val (S n))) (fun n => val (S (S n)))))
  (bind (or (val O) (bind x (fun n : nat => val (S (S n)))))
        (fun n => val (S n)))
```

This is trivially true. The converse holds also:

Lemma `Seven_refines_odd: refines (bind even (fun n => (val (S n)))) odd.`

However, the proof is a bit less simple, because we can not directly apply `sfix2_R_G_sfix1`. The solution is to first show that:

```
  refines even (or (val 0) (bind odd (fun n => (val (S n))))).
```

This is derived from `sfix2_R_G_sfix1`. Then, it suffices to prove:

```
refines
  (bind (or (val 0) (bind odd (fun n : nat => val (S n)))) (fun n => val (S n)))
  odd
```

Applying `sfix_prefix` on `odd`, it suffices to prove:

```
refines
  (bind (or (val 0) (bind odd (fun n : nat => val (S n)))) (fun n => val (S n)))
  (or (val 1) (bind odd (fun n => (val (S (S n))))))
```

This is trivially true.

The preceding example gives an application of the use of `sfix2_R_G_sfix1`. As announced, I give now a counter-example showing that the converse of this property does not necessarily hold. In the property, let me instantiate `F1` and `F2` with `ID: (DS unit) -> (DS unit) := fun x => x`. Let me instantiate `G` with `ABORT: (DS unit) -> (DS unit) := fun x => (abort unit)`. Clearly, `ID` and `ABORT` are monotonic. Furthermore, the left path of the diagram refines the right path.

```
  forall (x: DS unit), refines (ID (ABORT x)) (ABORT (ID x)).
```

However, `(refines (ABORT (sfixDS ID)) (sfixDS ID))` is false, because `(sfixDS ID)` is equivalent to `(magic unit)`, and `(abort unit)` does not refines `(magic unit)`.

Now, I prove a property announced in the introduction of this section: `even` is the non-deterministic computation of an even number

Lemma `even_refines_ext: refines even (bind (any nat) (fun n => val (n+n))).`

Lemma `ext_refines_even: refines (bind (any nat) (fun n => val (n+n))) even.`

The first property is proved using the fact that `even` is a smallest fixpoint. The second property can be simplified in:

```
  forall (n:nat), refines (val (n+n)) even.
```

This is proved by induction on `n` using the `sfix_prefix` property.

There are also examples of recursive specifications in imperative DSLs. In particular, the function `(fun (wh:DS unit) => If cond (seq body wh) skip)` admits `while` as smallest fixpoint and `whileWF` as biggest fixpoint (see sections 2.3.3.4 and 3.3.1).

3.2 Well-founded recursive functions

In this section, I study an other way to construct fixpoints for recursive functions in DSLs. Recursive calls in the fixpoint equation are now assumed to decrease with respect to a well-founded relation. (the equation is still assumed monotonic). Fixpoints of such equations have more properties: they are unique (thus, they are both smallest and biggest fixpoints), and they preserve termination.

The theory presented below is based on the notion of well-foundation defined in the standard library of Coq (see page 18). The proof of the following lemmas are done in the algebraic semantics.[3] I define below the fixpoint operator `fixF` and its main properties in a Coq section.
Section Fix.

Let me assume `A` the type for argument and `B` the type of the result. The resulting fixpoint will have type `A → (DS B)`
Variable `A B: Type`.

I denote `F` the function whose fixpoint is being computed. This function is assumed to be monotonic.
Variable `F: (A → (DS B)) → (A → (DS B))`.

```
Hypothesis F_monotony:
  forall (f g: A → (DS B)),
   (forall (x:A), (refines (f x) (g x)))
      → forall (x:A), (refines (F f x) (F g x)).
```

I assume a well-founded relation `R` on arguments, and I assume that recursive calls via `F` are decreasing the argument. This hypothesis is expressed by the `call_decreasing` hypothesis below.
Variable `R: A → A → Prop`.

```
Definition call_decreasing :=
  forall (f:A → (DS B)) (x:A),
      (refines (F f x)
               (F (fun y ⇒ (bind (ensure (R y x)) (fun _ ⇒ (f y))))
                   x)).
```

```
Hypothesis Rwf : (well_foundedT R).
```

3.2.1 Basic definition and properties

I first define the iterate of `F` by induction on the accessibility predicate of `x` via `R`. In this definition, I benefit of the opportunity to use proofs as values. Indeed, in recursive calls to `iterWF`, a proof that recursive argument `y` decreases with respect to `x` is chosen non-deterministically. If such a proof does not exist, the resulting specification is magic. By definition, when `x` is a minimal element of `R`, such a proof does not exist.

```
Fixpoint iterWF (x:A) (r: AccT R x) {struct r} : (DS B)
  := (F (fun y ⇒ bind (any (R y x)) (fun h ⇒ (iterWF (AccT_inv r y h))))
        x).
```

The following property shows that `iterWF` does actually not dependent on the proof of accessibility of `x` via `R`. Furthermore, such a proof always exists, because `R` is well-founded. The property below is proved by induction on r1.

```
Lemma iterWF_non_accT_dep:
 forall (x:A) (r1 r2: AccT R x),
```

[3]The Coq code presented here is actually automatically extracted using coqdoc.

```
    (refines (iterWF r1) (iterWF r2)).
```

Hence, I define the fixpoint `fixF` of `F` as below.
Definition `fixF (x:A): (DS B) := (iterWF (Rwf x)).`

First, I prove the following lemma by induction on `r`. The proof uses the properties `F_monotony` and `any_refines`.

```
Lemma iterWF_refines_F:
  forall (x:A) (r: AccT R x),
      (refines (iterWF r) (F fixF x)).
```

As a direct consequence, we get that `fixF` is a postfixpoint of `F`.

```
Lemma fixF_postfix:
   forall (x:A), (refines (fixF x) (F fixF x)).
```

I now prove that `fixF` refines any fixpoint of `F`

```
Lemma fixF_mostmagic:
  forall (fixO: A → (DS B)),
    (forall (x:A), (refines (F fixO x) (fixO x)))
     → forall (x:A), (refines (fixF x) (fixO x)).
```

Until now, I have not yet use the `call_decreasing` property. Let me first assume it.
Hypothesis `F_decreasing: call_decreasing.`

I show now that `fixF` is a fixpoint of `F`. As `fixF` is a prefixpoint, it suffices to prove that it is also a postfixpoint. Here, the `F_decreasing` hypothesis is used.

```
Lemma fixF_prefix:
   forall (x:A), (refines (F fixF x) (fixF x)).
```

As a consequence, `fixF` is refined by any iteration of `F`.

```
Lemma iterfixF_prefix:
  forall (n:nat) (x:A),
     (refines (iter n F fixF x) (fixF x)).
```

3.2.2 Invariant scheme

I now prove general invariant properties, satisfied by `fixF`.
Section `FixFInv.`

I assume a predicate `inv` compatible with observational equivalence via `inv_compat` hypothesis
Variable `inv: (A → (DS B)) → A → Type.`

```
Hypothesis inv_compat: forall (x:A) (s1 s2: A → (DS B)),
   (inv s1 x) → (refines (s1 x) (s2 x)) → (refines (s2 x) (s1 x)) →
      (inv s2 x).
```

I prove that if `inv` is preserved via `F`, then it holds for `fixF`. This property is proved by induction on the accessibility of `x` via `R`. It is then a direct consequence of the fixpoint property of `fixF`

```
Lemma fixF_rect:
 (forall (f: A → (DS B)) (x:A),
    (forall (y:A), (R y x) → (inv f y))
        → inv (F f) x)
  → forall (x:A), inv fixF x.
```

From this lemma, I now derive two fundamental and dual theorems about `fixF`. As we will see below, these theorems express at a very general level that `fixF` is respectively a most abortive and a most

magic fixpoint (and hence, the unique fixpoint). Like `P_whileWF_require` theorem page 49 (which is actually deduced from this one), the first theorem below allows to prove that `fixF` preserves weak-termination and determinism. The idea in both these theorems is to simplify (by weakening) the hypothesis of the preceding lemma, requiring that `inv` is preserved by `F`. In counterpart, we add in each theorem a new hypothesis, stating respectively that `inv` is compatible preconditions, and that `inv` is compatible with guards. The proofs of these theorems use `inv_compat` and respectively the fact that
`equiv (F f x) (F (fun y ⇒ (bind (require (R y x)) (fun _ ⇒ (f y)))) x)`
and
`equiv (F f x) (F (fun y ⇒ (bind (ensure (R y x)) (fun _ ⇒ (f y)))) x)`
which are two consequences of `F_decreasing` hypothesis.

```
Theorem inv_fixF_require:
  (forall (x:A) (Q: A → Prop) (s:A → (DS B)),
     ((Q x) → (inv s x)) → (inv (fun y ⇒ (bind (require (Q y)) (fun _ ⇒ (s y)))) x))
  → (forall (f: A → (DS B)), (forall (x:A), (inv f x))
            → forall (x:A), inv (F f) x)
  → forall (x:A), inv fixF x.

Theorem inv_fixF_ensure:
  (forall (x:A) (Q: A → Prop) (s: A → (DS B)),
     ((Q x) → (inv s x)) → (inv (fun y ⇒ (bind (ensure (Q y)) (fun _ ⇒ (s y)))) x))
  → (forall (f: A → (DS B)), (forall (x:A), (inv f x))
            → forall (x:A), inv (F f) x)
  → forall (x:A), inv fixF x.

End FixFInv.
```

These two lemmas have two consequences: `fixF` is a most abortive fixpoint and a most magic fixpoint (it allows an other proof of the preceding `fixF_mostmagic` property).

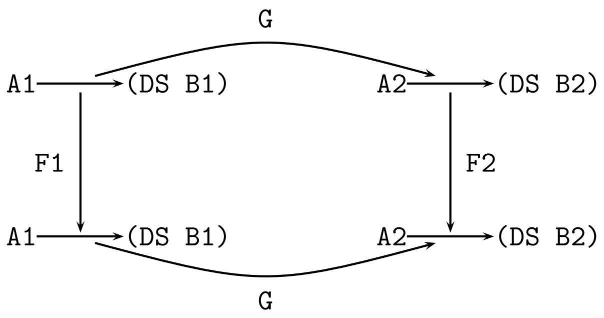
```
Lemma fixF_mostabort: forall (fixO: A → (DS B)),
   (forall (x:A), (refines (fixO x) (F fixO x)))
    → forall (x:A), (refines (fixO x) (fixF x)).

End Fix.
```

3.2.3 Other properties of `fixF`

Property `fixF_mostmagic` implies that `(fixF F Rwf)` is pointwise equivalent to `(sfixF F)`. Thus, extentionaly this fixpoint does not depend on `Rwf` nor `R`. Furthermore `fixF` is also a monotonic operator.

Because of `fixF_mostabort`, `(fixF F Rwf)` is also pointwise equivalent to `(bfixF F)`. Hence, the property `sfix2_R_G_sfix1` can be applied as `sfixF` or `bfixF`. In other words, given the following commuting diagram:



Then, `(fixF F2 Rwf2)` and `(G (fixF F1 Rwf1))` are pointwise equivalent.

3.3 Total correctness

In this section, I prove in Coq the total-correctness of refinement calculus for state-handling DSL, with respect to a sub-language of this DSL, where specifications can be interpreted as first-order, partially-defined and non-deterministic expressions. In particular, the evaluation of these expressions may never return a value, because of a non-terminating iteration or an abortion (such a division by zero). Informally, the total-correctness of refinement calculus, means that given a expression `p`, an initial state `st`, and `ps` an interpretation of `p` as a Dijkstra specification, such `pre ps st`, then all strategies of computations ultimately reduce `p` in `st` to some value `a` and some state `st'` such `(seq (set st') (val a))` can be interpreted as a Dijkstra specification refining `(seq (set st) ps)`. Furthermore, there still exists such a strategy.

3.3.1 Proving properties of `whileWF`

Before to present the total correctness result, I briefly sketch how properties of `fixF` presented previously allow to prove those of `whileWF` described section 2.3.3.4. The main idea is to construct a `whileWF2` operator observationally equivalent to `whileWF` but derived from `fixF`. As explained chapter 2, if `FSL.DS` denotes the type constructor of pure Dijkstra specifications, then the type constructor of imperative specifications is given by

```
Definition DS (A:Type) := St -> FSL.DS (prodT St A).
```

where `St` is the type of the global state. I will use this representation of imperative specifications to define `whileWF2`.

Assuming `cond:(DS bool)`, `body:(DS unit)` and `inv:St->Prop` (see notations of section 2.3.3.4 page 49), I define `(while_loop cond body inv)` as a step of the while-loop. Here, at each step, the current state is required to satisfy the invariant.

```
Definition while_loop (cond:(DS bool)) (body:(DS unit)) (inv:St -> Prop)
  : (DS unit) -> (DS unit)
  := fun (wh: DS unit) =>
     seq (bind get (fun st => require (inv st)))
         (If cond (seq body wh) skip).
```

Assuming that `FSL.fixF` is the `fixF` operator instantiated on the pure DSL, I define `while2` as the `FSL.fixF`-fixpoint of `(while_loop cond body inv)`. I thus use the representation of `(DS unit)` as `St -> FSL.DS (prodT St unit)`.

```
Definition while2 (cond:(DS bool)) (body:(DS unit))
                  (inv:St -> Prop) (R: St->St->Prop) (Rwf:(well_foundedT R))
  : DS unit := (fixF (while_loop cond body inv) Rwf).
```

Under `inv_preservation` and `variant_decreasing` hypotheses, `while_loop` is monotonic and satisfies `call_decreasing`. Thus, `while2` inherits the properties of `fixF`. Using the uniqueness of fixpoints, I prove the following equivalence:

```
Lemma while_equiv_while2:
  forall (cond:DS bool) (body:DS unit) (inv: St->Prop)
         (R: St->St->Prop) (Rwf:(well_foundedT R)),
    (inv_preservation cond body inv)
   -> (variant_decreasing cond body inv R)
    -> (equiv (seq (bind get (fun st => require (inv st))) (while cond body))
              (while2 cond body inv R Rwf)).
```

Now, I define `whileWF2`. The main problem here is to instantiate `while2` on some proof `Rwf` that `R` is well-founded. The trick is first to require that `R` is well-founded, and then to choose non-deterministically such a proof. As noticed section 3.2.3 the result of `fixF` does not depend on the proof of well-foundation.

```
Definition whileWF2 (cond:DS bool) (body:DS unit)
                    (inv: St->Prop) (R: St->St->Prop)
 : DS unit :=
 (seq
     (require ((inv_preservation cond body inv)
               /\ (variant_decreasing cond body inv R)
               /\ (well_foundedT R)))
     (bind (any (well_foundedT R)) (fun Rwf =>
           (while2 cond body inv Rwf)))).
```

From the previous remark (`while2` does not depend on `Rwf`) and from the `while_equiv_while2` equivalence, it is easy to see that `whileWF2` and `whileWF` are equivalent. Then, `P_whileWF_require` is directly derived from `inv_fixF_require`.

3.3.2 Total correctness in presence of non-determinism

The purpose of this section is to sketch how I formalize the total correctness theorem. As usual in presence of non-determinism, the operational semantics is a small-steps semantics. In other word, my semantics is based on a binary relation `leadsto` which describes one-step of computation. `leadsto` relates two couples formed of a program and a state. Hence, `leadsto` (`p`,`st`) (`p'`,`st'`) indicates that in the state `st`, `p` can be computed in `p'`, the resulting state being `st'`. It is non-deterministic in the sense that for a given (`p`,`st`), there are maybe several couple (`p'`,`st'`) such the relation holds.

A strategy of computation can be defined as a sequence of couple, successively in relation via `leadsto`. Given of a couple (`p`,`st`), stating that all strategies of computations terminate can be expressed as (`AccT` (**fun** `y x` ⇒ `leadsto x y`) (`p`,`st`)).

In order to express arbitrary but finite number of steps of computations, we define the binary relation `leadstostar` as the reflexive and transitive closure of `leadsto`.

In the following, I give the definition (`starRel R`) of the reflexive and transitive closure of a relation `R`. I prove that if I can decide whether some element is minimal for `R`, and if for a given `x`, (`AccT R x`) then there exists a minimal element `y` such (`starRel R y x`).

For (**fun** `y x` ⇒ `leadsto x y`), the minimal element are couple (`p`,`st`) such `p` is a value. And of course, being a value is a decidable property of programs. In other word, the total correctness can be formalized as the two properties:

- `leadstostar` induces a special case of refinement.
- (`AccT` (**fun** y x ⇒ `leadsto` x y) (p,st)) holds when `p` can be interpreted as a DSL specification `ps` such (`pre ps st`).

I prove now the announced property: accessibility implies the existence of a path to reach a minimal element if minimality is decidable. First, I give an inductive definition of (`starRel R`).

```
Inductive starRel (A: Type) (R: A → A → Prop) : A → A → Prop :=
  | starRel_refl: forall (x:A), (starRel R x x)
  | starRel_next:
      forall (x y z:A), (R x y) → (starRel R y z) → (starRel R x z).
```

The three following lemmas shows that (`starRel R`) is the smallest relation that is reflexive, transitive and that includes `R`

```
Lemma starRel_trans:
  forall (A:Type) (R: A → A → Prop) (x y z:A),
   (starRel R x y) → (starRel R y z) → (starRel R x z).
```

```
Lemma starRel_includes:
  forall (A:Type) (R: A → A → Prop) (x y:A), (R x y) → (starRel R x y).
```

```
Lemma starRel_smallest:
 forall (A:Type) (R P: A → A → Prop),
   (forall x:A, (P x x))
   → (forall (x y:A), (R x y) → (P x y))
   → (forall (x y z:A), (P x y) → (P y z) → (P x z))
    → forall (x y:A), (starRel R x y) → (P x y).
```

This small lemma shows that permutation of arguments is preserved via `starRel` (usually `leadsto` relation is written in the other way).

```
Lemma starRel_swap:
  forall (A:Type) (R: A → A → Prop) (x y:A),
    (starRel R x y) → (starRel (fun y x ⇒ R x y) y x).
```

Here is the announced property of `AccT`.

```
Theorem AccT_findmin:
 forall (A:Type) (R: A → A → Prop)
  (R_min_dec: forall (x:A), { forall (y:A), ~(R y x) } + { exists y:A, (R y x) })
   (x:A), (AccT R x) → (exists y:A, (starRel R y x) ∧ forall (z:A), ~(R z y)).
```

3.3.3 A non-deterministic programming language and its semantics

In this subsection, I define the abstract syntax of a non-deterministic expression language. The role of this syntax is mainly to give a support to express non-deterministic reduction rules on a sub-language of DSL and the associated notion of strong-termination. This sub-language contains all basic constructions except `ensure`, and a `any` restricted to non-empty types. Moreover, I explicitly introduce a `while` operator to express non-terminating iteration. In particular, this language of expressions contains a `require` operator which allows to express partial expressions like division by zero or access out of the bounds of an array (see the semantics below). Non-determinism is also a useful feature to give a semantics to "real" programming language aspects, such: non-initialized variables, pseudo-random computations or undetermined order of evaluation in some subexpressions. It also allows to give a very abstract view of interactions with environment, like keyboard inputs.

I interpret this programming language for ISL, a state-handling DSL formally defined chapter 4. The main particularity of this DSL, is that it allows to deal with a local state. But I do not use this feature here, and use it as a standard DSL. At this stage, the reader need only to know that the type of an ISL specification is of the form (`DS St A`) where `St` is the type of its local state and `A` the type of the result.

Thus, I first assume `St` the type of the global state, and interpret programs as ISL specifications in (`DS St`).

Variable `St`: `Type`.

I define the abstract syntax of the expression language by the `prog` inductive type below.

A main characteristic of this abstract syntax is that it is *very* abstract:

- the type of expressions `prog` is parametrized by the type of the result. Hence, it is a typed syntax. Furthermore, I interpret values as Coq values (and types as Coq types).
- variables and bindings (in the `bind` construction) are simulated by Coq functions. This avoids to explicitly redefine a notion of variable substitution.
- the definition of `bind` introduced what I call a *capture of structure* by analogy with a capture of variable. Indeed in an expression "`bind p1 p2`", `p2` is a function depending on the result of `p1`. But the syntactic structure of `p2` also depends on this result. Hence, `p2` can be any Coq function, performing pattern-matching or structural recursion. For instance, even if the syntax does not introduce an "if-then-else", I can define it as
  `bind cond` (**fun** `b` ⇒ **if** `b` **then** `pt` **else** `pe`) where `cond` is of type (`prog bool`) and `pt`, `pe` are of type (`prog A`) for any type `A`. Hence, the computational meaning of this syntax is defined by the operational semantics I give below, but also implicitly, by Coq reduction. However, this rules does not interfere with the result given here, since at our level, Coq reduction is deterministic and terminating. Let me remark however that by using a more complicated definition, I could avoid this capture of structure. But it does not seem necessary here. On the contrary, this capture of structure enriches the meaning of this language.

Here is the inductive definition

Inductive `prog`: `Type` → `Type` :=
| `val`: **forall** (`A`:`Type`), `A` → (`prog A`)
| `bind`: **forall** (`A B`:`Type`), (`prog A`) → (`A` → (`prog B`)) → (`prog B`)
| `require`: `Prop` → (`prog unit`)
| `any`: **forall** (`A`:`Type`), (`NonEmpty A`) → (`prog A`)
| `get`: (`prog St`)
| `set`: `St` → (`prog unit`)
| `while`: (`prog bool`) → (`prog unit`) → (`prog unit`).

Let me note that this syntax allows only *first-order* expressions. Indeed, a high-order operator such `lambda` below is *not correct* because it provokes a universe inconsistency.
Definition `lambda` (`A B`:`Type`) (`f`:(`A` → (`prog B`))): `prog` (`A` → (`prog B`)) := (`val f`).

It is possible to extend the syntax in order to support high-order expressions, but it requires to introduce an abstract syntax for types: it thus a bit more complex and I do not consider that here.

I now introduce the small-steps (operational) semantics for *prog*. As there is no syntax of "if-then-else" in the programming language, I use the above trick to express one computation step of `while`. Hence, I define

```
Definition unfold_while (cond:(prog bool)) (body:(prog unit)): (prog unit)
 :=(bind cond (fun b ⇒ if b then (bind body (fun _ ⇒ (while cond body))) else (val tt))).
```

The predicate `leadsto` defines one computation step on the language. [4] A computation transforms the expression involved but also the global state. More precisely, if `b` is a boolean (`leadsto p st p' st' b`) means that it is possible to computate a step from `p` in the state `st` to `p'` in the state `st'`. The boolean `b` indicates whether the computation has aborted: if it is `true`, a `require` clause has been violated. Because of `any` and `require`, this semantics is non-deterministic. In particular, a `require` expression may not abort even if its clause has been violated. [5]

```
Inductive leadsto: forall (A:Type), (prog A) → St → (prog A) → St → bool → Prop :=
| leadsto_bind:
   forall (A B:Type) (p1a p1b:(prog A)) (p2: A → (prog B)) (st1 st2:St) (ab:bool),
      (leadsto p1a st1 p1b st2 ab)
         → (leadsto (bind p1a p2) st1 (bind p1b p2) st2 ab)
| leadsto_bind_val:
   forall (A B:Type) (a:A) (p:A → (prog B)) (st:St),
                    (leadsto (bind (val a) p) st (p a) st false)
| leadsto_abort:
   forall (P:Prop) (st:St),
      ~P → (leadsto (require P) st (require P) st true)
| leadsto_require:
   forall (P:Prop) (st:St), (leadsto (require P) st (val tt) st false)
| leadsto_any:
   forall (A:Type) (An:NonEmpty A) (a:A) (st:St), (leadsto (any An) st (val a) st false)
| leadsto_get:
   forall (st:St), (leadsto get st (val st) st false)
| leadsto_set:
   forall (st1 st2:St), (leadsto (set st2) st1 (val tt) st2 false)
| leadsto_while:
   forall (cond:(prog bool)) (body:(prog unit)) (st:St),
     (leadsto (while cond body) st (unfold_while cond body) st false).
```

Here are some explanations about this rules:

- `leadsto_bind` are `leadsto_bind_val` are the two possible reduction rules for a `bind` expression. `leadsto_bind` propagates the reduction applicable on the left subexpression.

- `leadsto_abort` and `leadsto_require` give the semantics of `require`. When, the required property is not satisfied, the program may (but is not obliged to) abort. If the property is satisfied, the only applicable rule is `leadsto_require`, and `require` reduces to `skip`.

[4] As explained above, this computation-step may contain arbitrary Coq computations.

[5] Alternatively, I could also force a [require] expression to abort if the clause is not satisfied. The correctness theorem would be preserved (only a few intermediate lemmas need to be adjusted). However, this semantics seems less interesting than the one presented here since the computations may stuck on a non-decidable property.

- `leadsto_any` means that `any` reduces to any value `a` of the (non-empty) type, without changing the state.
- `leadsto_get` and `leadsto_set` mean respectively that `get` returns the current state without modifying it whereas `set` erases the value of the state with its parameter.
- `leadsto_while` means that `while` simply reduces to `unfold_while` without changing the state.

I have proved some trivial but fundamental properties of values: "to be a value" is a decidable property of programs, and values are the only irreducible expressions via `leadsto`. Let me remark, that this last assertion holds because `any` are applied to non-empty types, and there is no `ensure` construct in the language. In other word, we have the following lemma:

```
Lemma leadsto_values:
  forall (A:Type) (p:(prog A)) (st:St),
     { (exists a:A, p=(val a)) ∧ forall (p':(prog A)) (st':St), ~(leadsto p st p' st' false)} +
              { exists p':(prog A), exists st':St, (leadsto p st p' st' false)}.
```

I now define `leadstostar` as kind of reflexive and transitive closure of `leadsto`, using an inductive definition. Actually `leadstostar` is not exactly a reflexive and transitive closure, because in case of abortion, the reduction stops.

```
Inductive leadstostar (A:Type): (prog A) → St → (prog A) → St → bool → Prop :=
 | leadstostar_refl: forall (p:(prog A)) (st:St), (leadstostar p st p st false)
 | leadstostar_next_false: forall (p1 p2 p3: (prog A)) (st1 st2 st3:St) (b:bool),
     (leadsto p1 st1 p2 st2 false) →
       (leadstostar p2 st2 p3 st3 b) → (leadstostar p1 st1 p3 st3 b)
 | leadstostar_next_true: forall (p1 p2: (prog A)) (st1 st2:St),
     (leadsto p1 st1 p2 st2 true) → (leadstostar p1 st1 p2 st2 true).
```

Actually, restricted on the non-aborting cases, `leadstostar` is the reflexive and transitive closure of `leadsto` as stated by the following lemma:

```
Lemma leadstostar_starRel:
  forall (A:Type) (p1 p2:(prog A)) (st1 st2:St),
    (leadstostar p1 st1 p2 st2 false) ↔
        (starRel
             (fun (y x:(prodT (prog A) St)) ⇒
                       leadsto (fstT x) (sndT x) (fstT y) (sndT y) false)
             (pairT p2 st2) (pairT p1 st1)).
```

Thus, the picture of the theorem sketched section 3.3.2 is recovered when considering non-aborting computations. As we will see below, under the hypotheses of the correctness theorem, there are only non-aborting computations.

3.3.4 Statement of the total-correctness theorem

I now introduce a predicate `semDS` relating a program `p` to its ISL interpretation `ps`. The important point is that I interpret the `while` operator as `whileWF`, because I am here interested in *total* correctness. Other operators are directly interpreting by their corresponding one in ISL. Let me remark that `semDS` relation is not a function because in the case of the `while` operator, several interpretations are possible, depending on the invariant.

Let me now explain some syntactic details about specifications in ISL (see chapter 4 for more details):

- (`Val a`)%`ISL`, (`Req P`)%`ISL` and (`Any A`)%`ISL` are respectively the ISL notations for the usual (`val a`), (`require P`) and (`any A`).
- (`ISL.bind allPl pf1 pf2`) is the ISL notation for (`bind pf1 pf2`). The term `allPl` means here that `pf1` works on the global state `St`.
- (`!allPl`)%`ISL` and (`allPl :<- st`)%`ISL` are respectively reading and writing in `allPl` the global state. Thus, they are respectively the ISL notation for `get` and `set st`.

```
Inductive semDS: forall (A:Type), (prog A) → (DS St A) → Prop :=
| val_proof: forall (A:Type) (a:A), (semDS (val a) (Val a)%ISL)
| bind_proof: forall (A B:Type) (p1:(prog A)) (p2: A → (prog B))
   (pf1:(DS St A)) (pf2: A → (DS St B)),
     (semDS p1 pf1) → (forall (x:A),(semDS (p2 x) (pf2 x)))
       → (semDS (bind p1 p2) (ISL.bind allPl pf1 pf2))
| require_proof: forall (P:Prop), (semDS (require P) (Req P)%ISL)
| any_proof: forall (A:Type) (An:NonEmpty A), (semDS (any An) (Any A)%ISL)
| get_proof: (semDS get (!allPl)%ISL)
| set_proof: forall (st:St), (semDS (set st) (allPl :<- st)%ISL)
| while_proof: forall (cond:(prog bool)) (body:(prog unit))
   (inv: St → Prop) (R: St → St → Prop)
   (condpf: (DS St bool)) (bodypf: (DS St unit)),
     (semDS cond condpf)
      → (semDS body bodypf)
        → (semDS (while cond body) (whileWF condpf bodypf inv R)).
```

Hence, I can now formally write the statement of the theorem of this section. In this statement, `seq` of ISL is simply denoted by the infix symbol "-;".

```
Theorem semDS_correct:
  forall (A:Type) (p:(prog A)) (ps:(DS St A)) (st:St),
   (semDS p ps)
     → (pre (ps st))
        → (forall (p':(prog A)) (st':St) (b:bool),
              (leadstostar p st p' st' b) →
                 b=false
                 ∧ (exists ps':(DS St A),
                        (semDS p' ps') ∧ (refInEnv st ((allPl :<- st') -; ps') ps))%ISL)
           ∧ (AccT (fun (y x:(prodT (prog A) St)) ⇒
                        leadsto (fstT x) (sndT x) (fstT y) (sndT y) false)
                     (pairT p st)).
```

This theorem states that if the initial state satisfies the precondition of the initial expression, then computations can not abort. We thus recover the picture described section 3.3.2. Hence, the theorem is mainly divided in two parts. First, the left side of the conjunction in the conclusion corresponds to the preservation of the refinement semantics along arbitrary `leadstostar` reductions. And second, the right side of the conjunction corresponds to termination of an arbitrary sequence of `leadsto` reductions. In the following, I name the left-side as the *preservation property* and the right-side as the *termination property*.

I give the intermediary steps of the proof in the next subsection. Mainly, I first need to prove the preservation, because this property is used in the proof of the termination. Indeed, in this last proof, I need to prove that the state remains in the precondition of the computed expression, along reductions. This property is a direct consequence of the preservation property.

3.3.5 Main steps to prove this result

I now explain some details about the Coq proof of this result, because this proof has an interest by the Coq technique involved. Indeed, the preceding result has been proved in the ground theory of Coq without any variant of Streicher's K axiom! It is quite surprising, because there are many type dependencies in the preceding inductive definitions. The inversion of these kind of inductive definitions requires usually this axiom. For instance, with the previous definition of `prog`, a simple statement such (`val a`)=(`val b`) → `a=b` is not provable without K axiom.

In this proof, I have simply redefine `leadsto`, `leadstostar` and `semDS` using a continuation passing style. I still use inductive types, but the continuation style give me a better inversion. This approach has thus two advantages. First, it avoids to add an axiom. And more important (because, it is known that this axiom is consistent with Coq theory), it avoids the tedious proofs I have when each inversion require to invoke `inj_pairT2` (a consequence of K axiom). Actually, by using such a continuation passing style, we recover some kinds of weakest-precondition calculi.

Actually, the continuation technique used here is quite similar to the introduction of "evaluation contexts" usually associated with such operational semantics.

3.3.5.1 Step 1: Reformulating `leadsto`, `leadstostar` and `semDS` using angelic WP.

Hence, I first begin by defining a kind of angelic weakest-precondition associated to `leadsto`. Informally, given an expression `p` of type (`prog A`) and an initial state `st`, and a predicate `R` of type ((`prog A`) → `St` → `Prop`), I define `areduce p st R` as the weakest-precondition ensuring that there exists `p'` and `st'` such (`leadsto p st p' st' false`) and (`R p' st'`). I do not consider here aborting computations, since they can not happen under the hypotheses of the correctness theorem. Formally, here is the definition of `areduce`:

```
Inductive areduce: forall (A:Type), (prog A) → St → ((prog A) → St → Prop) → Prop :=
| areduce_bind_val:
   forall (A B:Type) (a:A) (p:A → (prog B)) (st:St) (R: (prog B) → St → Prop) ,
       (R (p a) st) → (areduce (bind (val a) p) st R)
| areduce_bind_nonval:
    forall (A B:Type) (p1:(prog A)) (p2: A → (prog B)) (st:St) (R: (prog B) → St → Prop),
       (areduce p1 st (fun p' st' ⇒ (R (bind p' p2) st')))
          → (areduce (bind p1 p2) st R)
| areduce_require: forall (P:Prop) (st:St) (R:(prog unit) → St → Prop),
     (R (val tt) st) → (areduce (require P) st R)
| areduce_any:
     forall (A:Type) (An:NonEmpty A) (a:A) (st:St) (R:(prog A) → St → Prop),
        (R (val a) st) → (areduce (any An) st R)
| areduce_get:
     forall (st:St) (R:(prog St) → St → Prop),
         (R (val st) st) → (areduce get st R)
| areduce_set:
```

```
    forall (st1 st2:St) (R:(prog unit) → St → Prop),
       (R (val tt) st2) → (areduce (set st2) st1 R)
| areduce_while:
    forall (st:St) (cond: (prog bool)) (body: (prog unit)) (R:(prog unit) → St → Prop),
      (R (unfold_while cond body) st)
        → (areduce (while cond body) st R).
```

Actually, `areduce p st R` could be alternatively defined as a fixpoint. Defining it as an inductive type provides an elimination scheme that simplifies some proofs: indeed, its case analysis pattern is different from `prog`. I will now construct a predicate `areduceinv` exactly equivalent to `areduce`, but such inversion of `areduce` corresponds to a reduction step on `areduceinv`. But first, let me introduce a predicate `is_val` indicating whether a program is a value or not.

```
Definition is_val (A:Type) (p:(prog A)) : Prop :=
  match p with
  | (val _ a) ⇒ True
  | _ ⇒ False
  end.
```

Of course, `is_val` is trivially a decidable predicate.

```
Lemma is_val_dec: forall (A:Type) (p:(prog A)), { (is_val p) } + { ~(is_val p) }.
```

Given a program `p: (prog A)`, a predicate `R: A → Prop` and a proof `H:(is_val p)`, I build the weakest-condition `(v_inv p R H)` such `R` holds for the underlying value of `p`. Hence, `v_inv` is an inverting function for `is_val` predicate.

```
Definition v_inv (A:Type) (p:(prog A)): (A → Prop) → (is_val p) → Prop
 := match p in (prog A) return (A → Prop) → (is_val p) → Prop with
     | (val A a) ⇒ fun R H ⇒ (R a)
     | x ⇒ fun R absurd ⇒ match absurd return Prop with end
     end.
```

`v_inv` satisfies the following principle:

```
Lemma v_inv_ind: forall (A:Type) (P: (prog A) → Prop) (R: A → Prop),
  (forall (a:A), (R a) → (P (val a)))
   → forall (p:(prog A)) (H:(is_val p)), (v_inv p R H) → (P p).
```

This allows now to define `areduceinv`. In the `bind` case, I proceed by a case analysis on `(is_val_dec p1)`. Alternatively, I could also directly proceed by case analysis on `p1`. But, in proofs, it is more convenient to invert using `is_val_dec`. Indeed, this introduces only two cases, instead of seven cases (six cases, being exactly treated in the same way).

```
Definition areduceinv (A:Type) (p:(prog A)) (st:St): ((prog A) → St → Prop) → Prop :=
  match p in (prog A) return ((prog A) → St → Prop) → Prop with
  | (val A a) ⇒ fun R ⇒ False
  | (bind A B p1 p2) ⇒ fun R ⇒
      match (is_val_dec p1) with
      | (left H) ⇒ (v_inv _ (fun a ⇒ (R (p2 a) st)) H)
      | _ ⇒ areduce p1 st (fun p' st' ⇒ R (bind p' p2) st')
      end
  | require P ⇒ fun R ⇒ (R (val tt) st)
  | (any A p) ⇒ fun R ⇒ exists a:A,(R (val a) st)
  | get ⇒ fun R ⇒ (R (val st) st)
```

```
  | (set v) ⇒ fun R ⇒ (R (val tt) v)
  | (while cond body) ⇒ fun R ⇒ (R (unfold_while cond body) st)
  end.
```

As announced, areduceinv and areduce are equivalent. The first lemma below, is proved by induction on areduce, while the second is proved by induction on p.

```
Lemma areduce_areduceinv:
 forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
   (areduce p st R) → (areduceinv p st R).
```

```
Lemma areduceinv_areduce:
 forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
   (areduceinv p st R) → (areduce p st R).
```

The following lemmas prove that ~(is_val p) is equivalent to **forall** (st:St), (areduce p st (**fun** _ _ ⇒ True)). In this case, p is said *reducible*. Let me first remark, that without the non-emptiness condition on any programs, the first lemma below would be false. Second, the second lemma is more general than the right side of the equivalence. But, this trivial property is crucial in the following. It gives a way to generalize a reducibility hypothesis into a nice inductive property (see checkDS_preservation_aux below).

```
Lemma non_val_reducible: forall (A:Type) (p:(prog A)),
  ~(is_val p) → forall (st:St), (areduce p st (fun _ _ ⇒ True)).
```

```
Lemma areduce_nonval: forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
  (areduce p st R) → ~(is_val p).
```

Now, I prove an important property of areduce. It is an angelic weakest-preconditions, and thus, it admits an angelic normal form. I prove this lemma on areduceinv instead of areduce. The proof is made by induction on p. Let me remark, that both sides of the equivalence must be proved simultaneously.

```
Lemma areduce_angelwp:
 forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
   (areduceinv p st R) ↔
    exists p':(prog A), exists st':St,
      ((areduceinv p st (fun p0 st0 ⇒ p0=p' ∧ st0=st')) ∧ (R p' st')).
```

A standard consequence of such a normal form property, is the weakening rule.

```
Lemma areduce_weaken:
 forall (A:Type) (p:(prog A)) (st:St) (P Q:(prog A) → St → Prop),
   (areduce p st P) →
     (forall p' st', (P p' st') → (Q p' st')) →
        (areduce p st Q).
```

As areduce is an angelic wp for leadsto, I now define areducestar an angelic wp for leadstostar.

```
Definition areducestar (A:Type) (R:(prog A) → St → Prop) (p:(prog A)) (st:St)
 := exists n:nat, (iter n (fun R p st ⇒ (areduce p st R)) R p st).
```

The three following lemmas are trivially derived from areducestar definition.

```
Lemma areducestar_refl: forall (A:Type) (R:(prog A) → St → Prop) (p:(prog A)) (st:St),
  (R p st) → (areducestar R p st).
```

```
Lemma areduce_areducestar: forall (A:Type) (R:(prog A) → St → Prop) (p:(prog A)) (st:St),
  (areduce p st R) → (areducestar R p st).
```

```
Lemma areduce_areducestar_trans:
```

```
forall (A:Type) (R:(prog A) → St → Prop) (p:(prog A)) (st:St),
  (areduce p st (areducestar R)) → (areducestar R p st).
```

The two following lemmas correspond to the normal form property of angelic weakest-preconditions. Here, the property has been split in two lemmas, because the two sides of the equivalence can be proved independently. In both cases, the proof is done by induction on the `n` of the `areducestar` in hypothesis.

```
Lemma areducestar_unfold: forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
   (areducestar R p st) →
    exists p':(prog A), exists st':St,
      ((areducestar (fun p0 st0 ⇒ p0=p' ∧ st0=st') p st) ∧ (R p' st')).
```

```
Lemma areducestar_fold:
 forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop) (p':(prog A)) (st':St),
  (areducestar (fun p0 st0 ⇒ p0=p' ∧ st0=st') p st)
    → (R p' st') → (areducestar R p st).
```

Of, course, I also derive a weakening lemma from the two preceding lemmas.

```
Lemma areducestar_weaken:
 forall (A:Type) (p:(prog A)) (st:St) (P Q:(prog A) → St → Prop),
   (areducestar P p st) →
     (forall p' st', (P p' st') → (Q p' st')) →
        (areducestar Q p st).
```

I now introduce a decomposition lemma to prove a property such (`areducestar R (bind p1 p2) st`). In particular, I want to reduce the proof of such a property, to a proof of (`areducestar R' p1 st`) for some `R'` that is to find. At a first step, I prove the decomposition lemma for `R'` being (`fun p' st' ⇒ (areducestar R (bind p' p2) st')`). The proof is done by induction on the `n` of the `areducestar` hypothesis.

```
Lemma areducestar_bind:
  forall (A B:Type) (p1:(prog A)) (p2:A → (prog B)) (st:St)
    (R: (prog B) → St → Prop),
   (areducestar (fun p' st' ⇒ (areducestar R (bind p' p2) st')) p1 st)
      → (areducestar R (bind p1 p2) st).
```

Then I prove the decomposition lemma for `R'` being (`fun p' st' ⇒ (exists a:A, p'=(val a) ∧ (areducestar R (p2 a) st'))`). The proof is simply a weakening on the preceding lemma.

```
Lemma areducestar_bind_val:
  forall (A B:Type) (p1:(prog A)) (p2:A → (prog B)) (st:St)
    (R: (prog B) → St → Prop),
   (areducestar (fun p' st' ⇒
        (exists a:A, p'=(val a) ∧ (areducestar R (p2 a) st'))) p1 st)
     → (areducestar R (bind p1 p2) st).
```

I now define `checkDS` predicate, which is the angelic wp corresponding to `semDS`. On the contrary of `areduce`, I define it by a fixpoint. Indeed, the corresponding inductive predicate is not correct, because of a non-positive occurrence in the `bind` case. Furthermore, the corresponding elimination scheme, would not provide a case analysis different from the one of `prog`.

```
Fixpoint checkDS (A:Type) (p:(prog A)) {struct p}: ((DS St A) → Prop) → Prop :=
  match p in (prog A) return ((DS St A) → Prop) → Prop with
  | (val A a) ⇒ fun R ⇒ (R (Val a)%ISL)
  | (bind A B p1 p2) ⇒ fun R ⇒
```

```
      (exists pf2: A → (DS St B),
        (checkDS p1 (fun pf1 ⇒
          (R (ISL.bind allPl pf1 pf2))
           ∧ forall (a:A), (checkDS (p2 a) (fun pf2' ⇒ pf2'=(pf2 a))))))
  | require P ⇒ fun R ⇒ (R (Req P)%ISL)
  | (any A An) ⇒ fun R ⇒ (R (Any A)%ISL)
  | get ⇒ fun R ⇒ (R (!allPl)%ISL)
  | set v ⇒ fun R ⇒ (R (allPl :<- v)%ISL)
  | (while cond body) ⇒
       fun R ⇒ (checkDS cond (fun c ⇒ (checkDS body (fun b ⇒
          exists inv, exists W, (R (whileWF c b inv W))))))
  end.
```

Hence, `checkDS` satisfies a normal form property for angelic weakest-precondition. As usual, I derive from this lemma a weakening lemma.

```
Lemma checkDS_angelwp:
 forall (A:Type) (p:(prog A)) (R:(DS St A) → Prop),
   (checkDS p R) ↔
    (exists ps:(DS St A), (checkDS p (fun ps0 ⇒ ps0=ps)) ∧ (R ps)).
```

3.3.5.2 Step 2: proving a reformulation of the preservation property.

I now want to prove the first part of the theorem: the preservation of the refinement semantics along arbitrary `leadstostar` reduction. In order to express the notion of *arbitrary reductions*, I introduce a demonic weakest-precondition for `leadstostar`, called `dreducestar`. I first define a demonic weakest-precondition for `leadsto`, called `dreduce`. Construction of `dreduce` and `dreducestar` is dual to `areduce` and `areducestar` respectively. Hence, the only differences between `areduce` and `dreduce` are the treatment of `any` and `require`. For `any`, `dreduce` imposes that the postcondition holds for *all* values of the type. For `require`, `dreduce` requires a proof of the required property.

```
Inductive dreduce: forall (A:Type), (prog A) → St → ((prog A) → St → Prop) → Prop :=
| dreduce_bind_val:
   forall (A B:Type) (a:A) (p:A → (prog B)) (st:St) (R: (prog B) → St → Prop) ,
       (R (p a) st) → (dreduce (bind (val a) p) st R)
| dreduce_bind_nonval:
    forall (A B:Type) (p1:(prog A)) (p2: A → (prog B)) (st:St) (R: (prog B) → St → Prop),
       (dreduce p1 st (fun p' st' ⇒ (R (bind p' p2) st')))
         → (dreduce (bind p1 p2) st R)
| dreduce_require: forall (P:Prop) (st:St) (R:(prog unit) → St → Prop),
     P → (R (val tt) st) → (dreduce (require P) st R)
| dreduce_any:
     forall (A:Type) (An:NonEmpty A) (st:St) (R:(prog A) → St → Prop),
        (forall (a:A), (R (val a) st)) → (dreduce (any An) st R)
| dreduce_get:
     forall (st:St) (R:(prog St) → St → Prop),
         (R (val st) st) → (dreduce get st R)
| dreduce_set:
     forall (st1 st2:St) (R:(prog unit) → St → Prop),
        (R (val tt) st2) → (dreduce (set st2) st1 R)
| dreduce_while:
```

```
  forall (st:St) (cond: (prog bool)) (body: (prog unit)) (R:(prog unit) → St → Prop),
   (R (unfold_while cond body) st)
    → (dreduce (while cond body) st R).
```

As for `areduce`, I define a `dreduceinv` predicate equivalent to `dreduce`, and such inversion of `dreduce` corresponds to reduction of `dreduceinv`.

```
Definition dreduceinv (A:Type) (p:(prog A)) (st:St): ((prog A) → St → Prop) → Prop :=
  match p in (prog A) return ((prog A) → St → Prop) → Prop with
  | (val A a) ⇒ fun R ⇒ False
  | (bind A B p1 p2) ⇒ fun R ⇒
      match (is_val_dec p1) with
      | (left H) ⇒ (v_inv _ (fun a ⇒ (R (p2 a) st)) H)
      | _ ⇒ dreduce p1 st (fun p' st' ⇒ R (bind p' p2) st')
      end
  | require P ⇒ fun R ⇒ P ∧ (R (val tt) st)
  | (any A An) ⇒ fun R ⇒ forall a:A,(R (val a) st)
  | get ⇒ fun R ⇒ (R (val st) st)
  | (set v) ⇒ fun R ⇒ (R (val tt) v)
  | (while cond body) ⇒ fun R ⇒ (R (unfold_while cond body) st)
  end.
```

I also prove that `dreduce` satisfies the normal form property corresponding to demonic weakest-precondition. As usual, this normal form involves the corresponding angelic weakest-precondition, which is here `areduce`.

```
Lemma dreduce_demonwp:
 forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
   (dreduceinv p st R) ↔
    (dreduceinv p st (fun _ _ ⇒ True)) ∧
    (forall (p':(prog A)) (st':St),
       (areduceinv p st (fun p0 st0 ⇒ p0=p' ∧ st0=st') → (R p' st'))).
```

This normal form property also implies a weakening property.

```
Lemma dreduce_weaken:
 forall (A:Type) (p:(prog A)) (st:St) (P Q:(prog A) → St → Prop),
   (dreduce p st P) →
     (forall p' st', (P p' st') → (Q p' st')) →
        (dreduce p st Q).
```

The property (`dreduce p st` (**fun** `_ _` ⇒ `True`)) implies that `p` is not a value. But the converse is not true, since (`deduce` (`require False`) `st` (**fun** `_ _` ⇒ `True`)) is false. But as implied by `checkDS_preservation_aux` below, if `p` is reducible and `st` is in the precondition of `p`, then (`deduce p st` (**fun** `_ _` ⇒ `True`)) holds.

```
Lemma dreduce_not_isval: forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
  (dreduce p st R) → ~(is_val p).
```

Now, I express a weak version of the preservation: "refinement is preserved by an arbitrary `leadsto` reduction". More formally, the following lemma states that given `p`, `st` and `ps` satisfying the hypotheses of the correctness theorem, then if `p` is reducible then for all p' and st' such (`leadsto p st p' st' false`) then (`bind` (`set st'`) (**fun** `_` ⇒ `p'`)) can be interpreted as a specification refining `ps`. In the proof of this lemma, the hypothesis "`p` *reducible*" is generalized as (`areduce p st R`) for some `R`. Hence, the proof is performed by induction on this hypothesis. It presents no particular difficulty, but the possibility to "invert" `dreduce` easily is very helpful here.

```
Lemma checkDS_preservation_aux:
 forall (A:Type) (p:(prog A)) (st:St) (ps:(DS St A)),
 ~(is_val p) →
   (checkDS p (fun ps0 ⇒ ps0=ps))
        → (pre (ps st))
         → (dreduce p st (fun p' st' ⇒
               (checkDS p' (fun ps'' ⇒
                  (refInEnv st ((allPl :<- st') -; ps'') ps)%ISL)))).
```

I now define `dreducestar`. The main idea in this definition, is that the postcondition `R` must be preserved by all iteration of `dreduce`, but only while programs are reducible. Indeed, if I have not introduced this restriction, then `dreducestar` would never be provable.

```
Definition dreducestar (A:Type) (R:(prog A) → St → Prop) (p:(prog A)) (st:St)
 := forall n:nat, (iter n (fun R p st ⇒ ~(is_val p) → (dreduce p st R)) R p st).
```

The basic property of `dreducestar` is to be the demonic weakest precondition associated to `areducestar`, as stated by the two following lemmas. The first lemma is proved by induction on the `n` of the `areducestar` hypothesis, while the second lemma is proved by induction on the `n` of the `dreducestar` in the goal. In both lemmas, I simply apply the corresponding relations between `dreduce` and `areduce`.

```
Lemma dreducestar_unfold:
 forall (A:Type) (p p':(prog A)) (st st':St) (R:(prog A) → St → Prop),
   (dreducestar R p st) → (areducestar (fun p0 st0 ⇒ p0=p' ∧ st0=st') p st)
      → (R p' st').
```

```
Lemma dreducestar_fold:
 forall (A:Type) (p:(prog A)) (st:St) (R:(prog A) → St → Prop),
  (dreducestar (fun p' _ ⇒ True) p st)
  → (forall (p':(prog A)) (st':St),
        (areducestar (fun p0 st0 ⇒ p0=p' ∧ st0=st') p st) → (R p' st'))
   → (dreducestar R p st).
```

Below is the reformulation of the preservation property that I now prove. The proof is a simple induction on the `n` of the `dreducestar`. The recursive case is a direct consequence of `checkDS_preservation_aux`.

```
Lemma checkDS_preservation:
 forall (A:Type) (p:(prog A)) (st:St) (ps:(DS St A)),
      (checkDS p (fun ps0 ⇒ ps0=ps)) → (pre (ps st))
        → (dreducestar (fun p' st' ⇒
               (checkDS p' (fun ps'' ⇒
                  (refInEnv st ((allPl :<- st') -; ps'') ps)%ISL))) p st).
```

The lemma below is direct consequence of `checkDS_preservation`. It shows that if the initial state satisfies the precondition of the initial expression, then in any reduction, the derived state satisfies the precondition of the corresponding derived expression.

```
Lemma checkDS_pre_preservation:
 forall (A:Type) (p:(prog A)) (st:St),
    (checkDS p (fun ps ⇒ pre (ps st)))
          → (dreducestar
                  (fun p' st' ⇒ (checkDS p' (fun ps' ⇒ pre (ps' st'))))
                  p st).
```

3.3.5.3 Step 3: proving a reformulation of the termination property

I first introduce the relation `reduce` on which I prove the accessibility property. This is a special case of `areduce`.

```
Definition reduce (A: Type) (y x: (prodT (prog A) St))
  := (areduce (fstT x) (sndT x) (fun p' st' ⇒ p'=(fstT y) ∧ st'=(sndT y))).
```

I first prove the accessibility values. It is trivial because, values have no predecessor for `reduce`.

```
Lemma AccT_val:
  forall (A:Type) (a:A) (st:St),
    (AccT (reduce (A:=A)) (pairT (val a) st)).
```

I then prove that `bind` preserves accessibility of `reduce`. Indeed, intuitively, if the left subexpression of the `bind` is accessible, it will always be reduced to a value. And then, the accessibility of the `bind` is deduced from the accessibility of the right subexpression. This property is thus proved by induction on the accessibility of left subexpression (or more formally the hypothesis (`AccT (reduce (A:=A)) x`) in the following lemma).

```
Lemma AccT_bind_preserv:
  forall (A:Type) (x:(prodT (prog A) St)),
    (AccT (reduce (A:=A)) x) →
     forall (B:Type) (p2: A → (prog B)),
      (forall (a:A) (st':St),
         (areducestar (fun p' st0 ⇒ p'=(val a) ∧ st0=st') (fstT x) (sndT x))
            → (AccT (reduce (A:=B)) (pairT (p2 a) st')))
         → (AccT (reduce (A:=B)) (pairT (bind (fstT x) p2) (sndT x))).
```

If a state `st` satisfies the precondition of an expression `bind p1 p2`, then it satisfies also trivially the precondition of `p1`.

```
Lemma checkDS_pre_bind_1:
 forall (A B:Type) (p1:(prog A)) (p2:A → (prog B)) (st:St),
    (checkDS (bind p1 p2) (fun ps ⇒ pre (ps st)))
    → (checkDS p1 (fun ps ⇒ pre (ps st))).
```

If a state `st` satisfies the precondition of an expression `bind p1 p2`, then if for any value `a` and state `st'`, such `p1` in the state `st` can be reduced in `a` and `st'`, then `st'` is in the precondition of (`p2 a`). This lemma is a consequence of `checkDS_preservation`.

```
Lemma checkDS_pre_bind_2:
 forall (A B:Type) (p1:(prog A)) (p2:A → (prog B)) (st:St) (a:A),
    (checkDS (bind p1 p2) (fun ps ⇒ pre (ps st)))
    → (dreducestar
         (fun p' st' ⇒
            p'=(val a) → (checkDS (p2 a) (fun ps ⇒ pre (ps st'))))
         p1 st).
```

I now prove the reformulation of the termination theorem. It is proved by induction on `p`. The cases `val`, `require`, `get` and `set` corresponds to the `AccT_val` lemma. The `bind` case uses the property `checkDS_pre_bind_1` and `checkDS_pre_bind_2` in order to discharge the hypotheses of the induction hypotheses and then apply `AccT_bind_preserv`. The `while` case is the most technical: it uses the preconditions of `whileWF`. In particular, it performs an induction over the variant and applies `AccT_bind_preserv` in order to prove that the unfolding of `while` terminates.

```
Lemma checkDS_termination:
 forall (A:Type) (p:(prog A)) (st:St),
```

```
    (checkDS p (fun ps ⇒ pre (ps st)))
        → (AccT (reduce (A:=A)) (pairT p st)).
```

3.3.5.4 Last step: proving the first formulation of the theorem

I have now to prove the correspondence between `semDS` and `checkDS` on one hand, and `leadstostar` and `areducestar` on the other hand. The two following lemmas correspond to this first part. The first is simply proved by induction on `semDS`. And the second is proved by induction on `p`.

```
Lemma semDS_checkDS: forall (A:Type) (p:(prog A)) (ps:(DS St A)),
  (semDS p ps) → (checkDS p (fun ps' ⇒ ps'=ps)).

Lemma checkDS_semDS: forall (A:Type) (p:(prog A)) (ps:(DS St A)),
  (checkDS p (fun ps' ⇒ ps'=ps)) → (semDS p ps).
```

I now establish a correspondence between `leadsto` and `areduce`. The first lemma below is proved by induction on `leadsto` and the second by induction on `areduce`.

```
Lemma leadsto_areduce:
  forall (A:Type) (p1 p2:(prog A)) (st1 st2:St),
    (leadsto p1 st1 p2 st2 false)
        → (areduce p1 st1 (fun p' st' ⇒ p'=p2 ∧ st'=st2)).

Lemma areduce_leadsto:
  forall (A:Type) (p1 p2:(prog A)) (st1 st2:St),
    (areduce p1 st1 (fun p' st' ⇒ p'=p2 ∧ st'=st2))
      → (leadsto p1 st1 p2 st2 false).
```

This equivalence is extended to `leadstostar` and `areducestar`. The first lemma below is proved by induction on `leadstostar` and the second is proved by induction on the `n` in the `areducestar` hypothesis.

```
Lemma leadstostar_areducestar:
  forall (A:Type) (p1 p2:(prog A)) (st1 st2:St),
    (leadstostar p1 st1 p2 st2 false)
        → (areducestar (fun p' st' ⇒ p'=p2 ∧ st'=st2) p1 st1).

Lemma areducestar_leadstostar:
  forall (A:Type) (p1 p2:(prog A)) (st1 st2:St),
    (areducestar (fun p' st' ⇒ p'=p2 ∧ st'=st2) p1 st1)
      → (leadstostar p1 st1 p2 st2 false).
```

I prove that a `leadsto` computation can not abort when the initial state is in the precondition of the initial expression. Then, I extend this result to `leadstostar` computations.

```
Lemma leadsto_nonabort:
   forall (A:Type) (p1 p2:(prog A)) (st1 st2:St) (b:bool),
    (leadsto p1 st1 p2 st2 b) → (checkDS p1 (fun ps ⇒ pre (ps st1)))
        → b=false.

Lemma leadstostar_nonabort:
   forall (A:Type) (p1 p2:(prog A)) (st1 st2:St) (b:bool),
    (leadstostar p1 st1 p2 st2 b) → (checkDS p1 (fun ps ⇒ pre (ps st1)))
        → b=false.
```

At last, I prove the theorem. I deduce directly the left side from `checkDS_preservation` and the preceding correspondences. Let me remark that `dreducestar_unfold` and `areducestar_unfold` allows to establish how `dreduce` can be expressed in terms of `leadstostar`. The right side is directly deduced from `checkDS_termination`.

```
Theorem semDS_correct:
  forall (A:Type) (p:(prog A)) (ps:(DS St A)) (st:St),
   (semDS p ps)
     → (pre (ps st))
        → (forall (p':(prog A)) (st':St) (b:bool),
              (leadstostar p st p' st' b) →
                 b=false
                 ∧ (exists ps':(DS St A),
                       (semDS p' ps') ∧ (refInEnv st ((allPl :<- st') -; ps') ps))%ISL)
           ∧ (AccT (fun (y x:(prodT (prog A) St)) ⇒
                       leadsto (fstT x) (sndT x) (fstT y) (sndT y) false)
                    (pairT p st)).
```

The technique involved in this theorem shows that CPS techniques and more precisely weakest-preconditions can be useful to invert complex inductive predicates. However a simpler (but weaker) version of this theorem could be probably be stated using a big-step semantics instead of a small-step one, using coinduction to express non-termination as exposed in “cite{Leroy-coindsem}.

3.3.6 Conclusion

We have proved here a total-correctness theorem in Coq for a particular DSL, and a particular programming language. The main interest of this proof is to show that the conditions presented page 2.3.3.4 ensure termination of programs with a non-deterministic semantics.

Chapter 4

Toward a refinement prover in Coq

In this chapter, I describe the prototype of a refinement prover for Coq. This prototype is still under construction. The fundamental choice of its design is that most of the job should be done by the standard version of Coq. In particular, interactive proofs must be done directly using Coq. First, this is a way to ensure a maximal compatibility with Coq language and libraries. Second, this is a way to have a "safe" prover. Third, as explained in chapter 2, refinement proofs in DSL can not be split in a wp-calculus part which computes a formula where denotations of programs have disappeared, and a proof of this formula. Because of high-order functions, refinement proofs in DSL mixes deduction steps and wp-computations. Thus, the kernel of the refinement prover is a Coq library that introduces denotations for DSL constructions, lemmas and tactics that helps in building refinement proofs, and some syntactic macros that helps in reading and writing DSL specifications.

4.1 Design principles of the Coq refinement prover

4.1.1 Interface issues of the prover

One of the technical challenge of this program is to deal with local sets of effects in expressions. For instance, imperative expressions presented chapter 2 are defined on a global state. But that is not usable in practice. We want thus to provide imperative expressions on a local state. This introduces in expressions many state conversions, like the embedding of an expression for a given state into a "bigger" state. Unfortunately, these conversions may quickly pollute the code of expressions.

I develop in this chapter some ideas to deal with this problem. These ideas are largely inspired by Filliâtre thesis [Fil99b, Fil99a, Fil03a].

- The first idea is that the really tedious point for a user is to *write* these conversions explicitly. Thus, I propose a statical analysis using typing with effects to infer these conversions. This analysis is described section 4.6. It is not yet implemented, but this analysis should be integrated in a front-end, external to standard Coq. This front-end would translate the DSL specifications written in a concrete syntax into Coq terms.

- The second idea is to forbid aliases using the typechecking of the front-end. Proof systems for imperative programs with aliases seem not yet mature. Proof of programs with aliases requires to reason explicitly on equality and difference of addresses in memory. Furthermore, in practice, algorithms which really use aliases, require to express graph properties on the whole memory like accessibility, sharing, disjointness, linearity, absence of cycles, ... It seems thus necessary to provide dedicated specifications constructs, and dedicated simplification procedures to help the user. On the contrary, forbidding aliases allows to make a proof system both simpler and more efficient. Each state of the program is denoted by a unique identifier. No notion of memory address is needed. States and their identifier are identified.

- The third idea is that DSL denotations in Coq must not be to far from the concrete syntax. Indeed, as the user will build interactive refinement proofs with Coq prover, intermediate specifications generated by the prover must remain readable. On the contrary to Filliâtre [Fil03a], we do not want to denote DSL terms in the purely functional monad, but directly in a imperative monad. This means in particular that the Coq library for DSL must provide denotations of states and state conversions.

- An other idea is that states will be finally denoted in Coq by tuples. Because of the non-aliasing principle, all computations involving states (affectation, dereference, parameter passing of references, state conversions) can be performed by Coq during proofs without the help of the user.

- At last, at the library level of Coq it seems interesting to have a more abstract representation of states and state conversions. This abstraction of states may present several advantages. First, it should allow to write the library independently of the way that states and states conversions are finally implemented. Second, implementations of conversion functions between state can be partly produced by the front-end. This may both improve efficiency of the prover and readability of goals. For instance, one can thus imagine to name the conversions functions, such the Coq denotation is not polluted by their encoding. The current abstract representation for states is described section 4.3.1.

4.1.2 Extending state-handling monads with local state handling

I present here how I extend the usual monad with state-handling for dealing with local states. The design of this extension is lead by the following principle: refinement proofs in Coq must be as less polluted by state conversions as possible. Furthermore, as explained before, for modularity reasons, I want to keep partly abstract the conversions mechanism at the level of the Coq library. This allows also to keep close to the simple theory seen until now.

This extension of the monadic description leads to modifies some operators of monads and add new operators. The main modification concerns `bind`. This modification is based on the following remark: in a bind-expression "`bind s1 s2`" working on a state of type `St`, `s1` is very often of the form "`f x1 x2... xn`" where `f` is a previously defined function (and "`xi`" are functional variables). And because `f` has been previously defined, it does not work on all the state of type `St` but only on a subset: `f` does not work with new mutables variables that have been introduced after it. I say that `f` works on a *place* of `St`. Hence, in the new definition of `bind`, I impose `s2` to work on `St`, but `s1` may work on a place of `St`. Hence `bind` admits three parameters: a place `v`, an expression `s1` working on `v`, and a expression `s2` working on `St` and parametrized by the result of `s1`. Given `x` a result of `s1`, if `(s2 x)` works only on a place `v2` of `St`, it is possible to *embed* `s2` on `St` using `bind v2 (s2 x) (fun a => val a)` (where `(fun a => val a)` works here on `St`). This last construction corresponds actually as a new derived operator called `embed`, which parameters are here `v2` and `(s2 x)`.

At last, I need a new operator called `block`, parametrized by a specification, say `s`. This operator allows to introduce a new mutable variable, which is not visible outside of the block. This operator is defined section 4.3.4.

4.1.3 Architecture of the Coq library

As explained in section 2.4.1, the architecture of the library is not based on the algebraic semantics, but has been inspired from it. Currently, it is a two-level architecture. But it should evolute in a three-level architecture.

[Un graphe de dépendances des paquetages]

4.2 Purely functional wp-calculus

The concepts and intuitions underlying the definitions now introduced are largely presented in subsection 2.3.2. I try however to recall briefly these ideas for each new definitions.

4.2.1 Module `WP`: Definitions and properties of weakest-preconditions.

4.2.1.1 Definition of weakest-preconditions.

On functional expressions, weakest-preconditions are of type (`A` → `Prop`) → `Prop` where `A` is the type of the result returned by the expression. Weakest preconditions are kind of continuations semantics: given a post-condition on the result, they returned a precondition ensuring that the post-condition hold on some result of the expression. In presence, of non-determinism, there are two families of preconditions: demonic that ensure the post-condition hold for all possible results of the expression, and angelic that ensure there exists a result such the post-condition holds. A pair of demonic and angelic weakest-precondition satisfies the following `is_wp_pair` predicate.

```
Definition is_wp_pair (A : Type) (sdemon angel : (A → Prop) → Prop) :=
  forall R : A → Prop,
  (sdemon R ↔ sdemon (fun _ ⇒ True) ∧ (forall a : A, angel (eq a) → R a))
  ∧ (angel R ↔ (exists a : A, angel (eq a) ∧ R a)).
```

A functional Dijkstra specification is simply such a pair.

```
Record DS (A : Type) : Type :=
  {
   sdemon : (A → Prop) → Prop;
   angel : (A → Prop) → Prop;
   DScorrect : is_wp_pair sdemon angel
  }.
```

In the following, I introduce syntactic macros. If `s` is a specification, (`sdemon s` (**fun** _ ⇒ `True`)) is called the precondition of `s` and is noted (`pre s`). For a given `a`, (`angel s` (`eq a`)) is called the postcondition that `s` returns `a`, and is noted (`post s a`).

```
Notation "'pre' s" := (sdemon s (fun _ ⇒ True)) (at level 70, s at level 0).
Notation "'post' s a":= (angel s (eq a)) (at level 70, s at level 0, a at level 0).
```

Here are some direct reformulations of the definitions. These lemmas shows that a specification is uniquely determined by a its `pre`/`post` couple. Such a couple is traditionally called the "normal form" of the specification in refinement calculus.

```
Lemma sdemon_fold :
 forall (A : Type) (s : DS A) (R : A → Prop),
 pre s →
 (forall a : A, post s a → R a) →
 sdemon s R.
```

```
Lemma sdemon_unfold :
 forall (A : Type) (s : DS A) (R : A → Prop) (a : A),
 sdemon s R → (post s a) → R a.
```

```
Lemma angel_fold :
 forall (A : Type) (s : DS A) (R : A → Prop) (a : A),
 post s a → R a → angel s R.
```

```
Lemma angel_unfold :
 forall (A : Type) (s : DS A) (R : A → Prop),
 angel s R →
 exists a : A, (post s a) ∧ R a.
```

4.2.1.2 Basic properties of weakest-preconditions

The monotony of `sdemon` and `angel` with respect to logical implication is directly derived from these properties.

```
Lemma sdemon_weaken :
 forall (A : Type) (s : DS A) (R1 R2 : A → Prop),
 sdemon s R1 → (forall a : A, R1 a → R2 a) → sdemon s R2.
```

```
Lemma angel_weaken :
 forall (A : Type) (s : DS A) (R1 R2 : A → Prop),
 angel s R1 → (forall a : A, R1 a → R2 a) → angel s R2.
```

Sometimes, I need a finner weakenning property for `angel`, using `pre` and `post`.

```
Lemma angel_weaken_pre :
 forall (A : Type) (s : DS A) (R1 R2 : A → Prop),
 pre s
 → angel s R1
  → (forall a:A, (post s a) → R1 a → R2 a)
   → angel s R2.
```

I directly derive from `sdemon_weaken` the following properties on `sdemon`. Of course, these properties are also be proved for `angel` using `angel_weaken`.

```
Lemma sdemon_and_distr: forall (A:Type) (s:(DS A)) (R1 R2:A → Prop),
   (sdemon s (fun a:A ⇒ (R1 a) ∧ (R2 a))) → (sdemon s R1) ∧ (sdemon s R2).
```

```
Lemma sdemon_or_lift: forall (A:Type) (s:(DS A)) (R1 R2:A → Prop),
  (sdemon s R1) ∨ (sdemon s R2) → (sdemon s (fun a:A ⇒ (R1 a) ∨ (R2 a))).
```

```
Lemma sdemon_forall_distr:forall (A B:Type) (s:(DS A)) (R:B → A → Prop),
  (sdemon s (fun a:A ⇒ forall (b:B),(R b a))) → forall (b:B), (sdemon s (R b)).
```

```
Lemma sdemon_exists_lift:forall (A B:Type) (s:(DS A)) (R:B → A → Prop) (b:B),
  (sdemon s (R b)) → (sdemon s (fun a:A ⇒ (exists b:B, (R b a)))).
```

Using the definition of `sdemon` and `angel`, it is possible to prove the converse of the properties `sdemon_and_distr`, `angel_or_lift` and `angel_exists_lift`.

```
Lemma sdemon_and_lift :
 forall (A : Type) (s : DS A) (R1 R2 : A → Prop),
 sdemon s R1 → sdemon s R2 → sdemon s (fun a : A ⇒ R1 a ∧ R2 a).
```

```
Lemma angel_or_distr :
 forall (A : Type) (s : DS A) (R1 R2 : A → Prop),
 angel s (fun a : A ⇒ R1 a ∨ R2 a) → angel s R1 ∨ angel s R2.
```

```
Lemma angel_exists_distr :
 forall (A B : Type) (s : DS A) (R : B → A → Prop),
 angel s (fun a : A ⇒ exists b : B, R b a) → exists b : B, angel s (R b).
```

The converse of `sdemon_forall_distr` do not hold. Indeed, assuming **forall** (b:B), `sdemon s (R b)`, then `sdemon (`**fun** `a : A ⇒` **forall** `b : B, R b a)` may not be true. Because `B` could be empty and `pre s` be `False`. By excluding this bad case, I can however prove two weaker versions of this property.

```
Lemma sdemon_forall_lift1 :
 forall (A B : Type) (s : DS A) (R : B → A → Prop),
 pre s →
 (forall b : B, sdemon s (R b)) →
 sdemon s (fun a : A ⇒ forall b : B, R b a).
```

```
Lemma sdemon_forall_lift2 :
 forall (A B : Type) (s : DS A) (R : B → A → Prop),
 (NonEmpty B)
  → (forall b : B, sdemon s (R b)) →
 sdemon s (fun a : A ⇒ forall b : B, R b a).
```

4.2.1.3 Refinement in the functional specification language.

I now define the refinement relation between specifications. There are at least three equivalent definitions. I choose one, and shows the equivalence with the two others.

Here is the first definition of the refinement. Let me note that in intuitionnistic logic, both sides of the conjonction are necessary, whereas in classical logic the left part is sufficient (in classical logic, `angel` can be expressed in terms of `sdemon` using a double negation).

```
Definition refines (A : Type) (s1 s2 : DS A) :=
  forall R : A → Prop,
  (sdemon s2 R → sdemon s1 R) ∧ (pre s2 → angel s1 R → angel s2 R).
```

With this definition, it is almost trivial to prove that refinement is a pre-order.

```
Lemma refines_refl : forall (A : Type) (s : DS A), refines s s.
```

```
Lemma refines_trans :
 forall (A : Type) (s1 s2 s3 : DS A),
 refines s1 s2 → refines s2 s3 → refines s1 s3.
```

I now show that this definition is equivalent to the traditional one, expressed in terms of `pre/post`. The proof only applies the relation between `sdemon/angel` and `pre/post`.

```
Lemma refines_prepost :
 forall (A : Type) (s1 s2 : DS A),
 refines s1 s2 ↔
 pre s2
  → pre s1 ∧ (forall a : A, post s1 a → post s2 a).
```

I now prove the equivalence with a third definition. For more convenience, the equivalence is split into two lemmas. This definition is actually only a reformulation of the preceding ones, using `sdemon` and `angel` instead of `pre/post`. With respect with the first definition, this definition avoids a quantification over all postconditions. In particular, the lemma `refines_fold` can be interpreted as a proof rule allowing to simplify refinement proofs using wp-calculus.

```
Lemma refines_unfold :
 forall (A : Type) (s1 s2 : DS A),
 refines s1 s2 →
   pre s2 → sdemon s1 (fun a ⇒ post s2 a).
```

```
Lemma refines_fold :
 forall (A : Type) (s1 s2 : DS A),
 (pre s2 → sdemon s1 (fun a ⇒ post s2 a)) →
 refines s1 s2.
```

This lemma is a little proof rule (in refinement proofs) that allows to introduce in the context the precondition of the specification being refined.

```
Lemma refines_pre_intro :
 forall (A : Type) (s1 s2 : DS A),
 (pre s2 → refines s1 s2) → refines s1 s2.
```

4.2.2 Module FSL: primitive operators of functional specifications

This module defines the primitive operators of functional specifications. These primitive operators are defined directly using a pair of weakest-preconditions. In the following, the proofs that these pairs satisfy the `is_wp_pair` condition are named with a suffix "`_correct`". Others operators of FSL (like recursion operators) are built using these primitive operators.

The primitive operators are split in two categories: those which are primitive in the sense of the algebraic semantics: `val`, `bind`, `any`, `require` and `ensure`. And those which are defined as primitive to increase the efficiency of wp-calculus: `abort`, `magic` and `or`.

4.2.2.1 primitive operators of the algebraic semantics.

I first introduce `val` and `bind` operators.

```
Definition val (A : Type) (a:A) : DS A
 := Build_DS (sdemon:=fun R ⇒ R a) (angel:=fun R ⇒ R a) (val_correct a).
```

```
Definition bind (A B : Type) (s1:DS A) (s2:(A → DS B)) : DS B
 := Build_DS
        (sdemon:=fun R ⇒ sdemon s1 (fun a ⇒ sdemon (s2 a) R))
        (angel:=fun R ⇒ angel s1 (fun a ⇒ angel (s2 a) R))
        (bind_correct s1 s2).
```

With the two following lemmas, I check that `bind` satisfies its expected `pre`/`post` semantics.

```
Lemma bind_pre : forall (A B : Type) (s1 : DS A) (s2 : A → DS B),
 pre (bind s1 s2) ↔
   pre s1 ∧ (forall a : A, post s1 a → pre (s2 a)).
```

```
Lemma bind_post : forall (A B : Type) (s1 : DS A) (s2 : A → DS B) (b : B),
 (post (bind s1 s2) b) ↔
   (exists a : A, (post s1 a) ∧ (post (s2 a) b)).
```

Refinement is monotonic with respect to `bind`

```
Lemma bind_monotony : forall (A B : Type) (s1 s3 : DS A) (s2 s4 : A → DS B),
 refines s1 s3 →
  (forall a : A, refines (s2 a) (s4 a)) →
   refines (bind s1 s2) (bind s3 s4).
```

This lemma corresponds to a rule for small-steps simplifications in refinement proofs, see section 2.3.2.2. A goal `refines (bind s1 s2) (bind s3 s4)` such `s1` refines `s3`, can be simplified in (`pre s3 → sdemon s1 (fun a : A ⇒ refines (s2 a) (s4 a))`). If `s1` is sufficiently simple, this gives a simple formula.

```
Lemma bind_simpl_both_l : forall (A B : Type) (s1 s3 : DS A) (s2 s4 : A → DS B),
 refines s1 s3 →
  (pre s3 → sdemon s1 (fun a : A ⇒ refines (s2 a) (s4 a))) →
    refines (bind s1 s2) (bind s3 s4).
```

The three following lemmas also correspond to various small-steps simplifications rules.

```
Lemma bind_simpl_both : forall (A B : Type) (s1 s3 : DS A) (s2 s4 : A → DS B),
 refines s1 s3 →
  (pre s1 → sdemon s1 (fun a : A ⇒ refines (s2 a) (s4 a))) →
    refines (bind s1 s2) (bind s3 s4).

Lemma bind_simpl_left : forall (A B : Type) (s1 : DS A) (s2 : A → DS B) (s3 : DS B),
 sdemon s1 (fun a : A ⇒ refines (s2 a) s3) →
  refines (bind s1 s2) s3.

Lemma bind_simpl_right : forall (A B : Type) (s1 : DS A) (s2 : A → DS B) (s3 : DS B),
 (pre s1 → angel s1 (fun a ⇒ refines s3 (s2 a))) →
  refines s3 (bind s1 s2).
```

I now prove the three monadic properties relating `bind` and `val`. Currently, I have not yet introduced the equivalence relation associated to `bind`. Thus, the three properties correspond to the six lemmas below.

```
Lemma bind_val_l_ref1 : forall (A B : Type) (a : A) (s1 : A → DS B),
 refines (bind (val a) s1) (s1 a).

Lemma bind_val_l_ref2 : forall (A B : Type) (a : A) (s1 : A → DS B),
 refines (s1 a) (bind (val a) s1).

Lemma bind_val_r_ref1 : forall (A : Type) (s1 : DS A),
 refines (bind s1 (fun a ⇒ val a)) s1.

Lemma bind_val_r_ref2 : forall (A : Type) (s1 : DS A),
 refines s1 (bind s1 (fun a ⇒ val a)).

Lemma bind_assoc_ref1 : forall (A B C : Type) (s1 : DS A) (s2 : A → DS B) (s3 : B → DS C),
 refines (bind (bind s1 s2) s3)
   (bind s1 (fun a ⇒ bind (s2 a) s3)).

Lemma bind_assoc_ref2 : forall (A B C : Type) (s1 : DS A) (s2 : A → DS B) (s3 : B → DS C),
 refines (bind s1 (fun a ⇒ bind (s2 a) s3))
   (bind (bind s1 s2) s3).
```

Here is the definition of the `require` operator.

```
Definition require (p:Prop) : DS unit
 := Build_DS (sdemon:=fun R ⇒ p ∧ R tt) (angel:=fun R ⇒ R tt) (require_correct p).
```

With the two following lemmas, I check that `require` satisfies its expected `pre`/`post` semantics.

```
Lemma require_pre : forall p : Prop,
  pre (require p) ↔ p.

Lemma require_post : forall (p : Prop) (a : unit),
  (post (require p) a) ↔ True.
```

The two following lemmas state respectively that `skip` refines any `require`-expression. and that `require p` refines `skip` if `p` holds.

```
Lemma require_refined_tt : forall p : Prop,
 refines (val tt) (require p).

Lemma require_refines_tt : forall p : Prop,
  p → refines (require p) (val tt).
```

Here is the definition of `ensure`:

```
Definition ensure (p:Prop) : DS unit
 := Build_DS (sdemon:=fun R ⇒ p → R tt) (angel:=fun R ⇒ p ∧ R tt) (ensure_correct p).
```

With the two following lemmas, I check that `ensure` satisfies its expected `pre`/`post` semantics.

```
Lemma ensure_pre : forall p : Prop,
  pre (ensure p) ↔ True.

Lemma ensure_post : forall (p : Prop) (a : unit),
  post (ensure p) a ↔ p.
```

The two following lemmas state respectively that any `ensure`-expression refines `skip`. and that `skip` refines `ensure p` if `p` holds.

```
Lemma ensure_refines_tt : forall p : Prop,
  refines (ensure p) (val tt).

Lemma ensure_refined_tt : forall p : Prop,
  p → refines (val tt) (ensure p).
```

The two following lemmas recall that refinement corresponds to a weakening on preconditions and a strengthening on postconditions.

```
Lemma require_refines_require : forall p1 p2 : Prop,
  (p2 → p1) → refines (require p1) (require p2).

Lemma ensure_refines_ensure : forall p1 p2 : Prop,
  (p1 → p2) → refines (ensure p1) (ensure p2).
```

The four following lemmas are particular cases of `bind_simpl_right` and `bind_simpl_left`.

```
Lemma require_refined : forall (A : Type) (p : Prop) (s1 s2 : DS A),
 (p → refines s1 s2) →
 refines s1 (bind (require p) (fun _ ⇒ s2)).

Lemma ensure_refines : forall (A : Type) (p : Prop) (s1 s2 : DS A),
 (p → refines s1 s2) →
 refines (bind (ensure p) (fun _ ⇒ s1)) s2.

Lemma require_refines : forall (A : Type) (p : Prop) (s1 s2 : DS A),
 p → refines s1 s2 →
 refines (bind (require p) (fun _ ⇒ s1)) s2.

Lemma ensure_refined: forall (A : Type) (p : Prop) (s1 s2 : DS A),
 p → refines s1 s2 →
```

```
 refines s1 (bind (ensure p) (fun _ ⇒ s2)).
```

I now define any.

```
Definition any (A : Type) : DS A
  := Build_DS
       (sdemon:=fun R ⇒ forall a : A, R a)
       (angel:=fun R ⇒ exists a : A, R a)
       (any_correct (A:=A)).
```

With the two following lemmas, I check that any satisfies its expected pre/post semantics.

```
Lemma any_pre : forall A : Type,
  pre (any A) ↔ True.

Lemma any_post : forall (A : Type) (a : A),
 (post (any A) a) ↔ True.
```

It is easy to prove the three following lemmas. In particular, the second one is a particular case of bind_simpl_right, whereas the third one is a particular case of bind_simpl_left.

```
Lemma any_refined : forall (A : Type) (a : A),
  refines (val a) (any A).

Lemma any_refined_bind : forall (A B : Type) (s1 : A → DS B) (s2 : DS B) (a : A),
 refines s2 (s1 a) → refines s2 (bind (any A) s1).

Lemma any_refines : forall (A B : Type) (s1 : A → DS B) (s2 : DS B),
 (forall a : A, refines (s1 a) s2) →
   refines (bind (any A) s1) s2.
```

With the properties proved above, it is easy to check that the properties of the algebraic semantics are satisfied. Furthermore, we have also proved that these definitions are equivalent to those of the pre/post semantics.

4.2.2.2 Important properties of any operator.

I detail now some properties of any.

```
Lemma any_val_refined_unfold : forall (A B : Type) (f : A → DS B) (s : DS A),
 refines s
   (bind (any A) (fun a ⇒ bind (f a) (fun _ ⇒ val a))) →
 (forall a : A, pre (f a)) → sdemon s (fun a ⇒ angel (f a) (fun _ ⇒ True)).

Lemma any_val_refined_fold : forall (A B : Type) (f : A → DS B) (s : DS A),
 ((forall a : A, pre (f a)) → sdemon s (fun a ⇒ angel (f a) (fun _ ⇒ True))) →
 refines s
   (bind (any A) (fun a ⇒ bind (f a) (fun _ ⇒ val a))).

Lemma any_on_non_empty_type_can_be_lifted : forall (A B C : Type) (s : DS A) (f : A → B →
DS C),
 (NonEmpty B) →
 refines (bind s (fun a ⇒ bind (any B) (fun b ⇒ f a b)))
   (bind (any B) (fun b ⇒ bind s (fun a ⇒ f a b))).

Lemma any_when_pre_can_be_lifted : forall (A B C : Type) (s : DS A) (f : A → B → DS C),
 pre s →
 refines (bind s (fun a ⇒ bind (any B) (fun b ⇒ f a b)))
   (bind (any B) (fun b ⇒ bind s (fun a ⇒ f a b))).
```

4.2.2.3 primitive operators for "efficiency"

Here, I define `abort`, `magic` and `or` as primitive operators, because it allows to have better wp-simplifications for these constructions.

```
Definition abort (A : Type) : DS A
 := Build_DS (sdemon:=fun R ⇒ False) (angel:=fun R ⇒ False) (abort_correct (A:=A)).

Lemma refines_abort :
 forall (A : Type) (s : DS A), refines s (abort A).

Definition magic (A : Type) : DS A
 := Build_DS (sdemon:=fun R ⇒ True) (angel:=fun R ⇒ False) (magic_correct (A:=A)).
Lemma refines_magic :
 forall (A : Type) (s : DS A), refines (magic A) s.


Definition or (A:Type) (s1 s2:(DS A)) : (DS A)
:= Build_DS (sdemon:=(fun R ⇒ (sdemon s1 R) ∧ (sdemon s2 R)))
            (angel:=(fun R ⇒ (angel s1 R) ∨ (angel s2 R)))
            (or_correct s1 s2).
Lemma or_pre: forall (A:Type) (s1 s2:(DS A)), (pre (or s1 s2)) ↔ (pre s1) ∧ (pre s2).

Lemma or_post: forall (A:Type) (s1 s2:(DS A)) (a:A),
 (post (or s1 s2) a) ↔ (post s1 a) ∨ (post s2 a).


Lemma refines_or_intro_l: forall (A:Type) (s1 s2 s:(DS A)),
  (refines s s1) → (refines s (or s1 s2)).

Lemma refines_or_intro_r: forall (A:Type) (s1 s2 s:(DS A)),
  (refines s s2) → (refines s (or s1 s2)).

Lemma refines_or_elim: forall (A:Type) (s1 s2 s:(DS A)),
  (refines s1 s) → (refines s2 s) → (refines (or s1 s2) s).
```

Here is the usual definition of `or` using `bind` and `any`. It allows to inherits properties of `bind`, but has less powerful wp.

```
Definition or_x (A : Type) (s1 s2:DS A) :=
  (bind (any bool) (fun b ⇒ if b then s1 else s2)).

Lemma refines_or_x: forall (A : Type) (s1 s2:DS A),
 (refines (or s1 s2) (or_x s1 s2)).

Lemma refines_x_or: forall (A : Type) (s1 s2:DS A),
 (refines (or_x s1 s2) (or s1 s2)).

Lemma or_monotony :
 forall (A : Type) (s1 s2 s3 s4 : DS A),
 refines s1 s3 →
 refines s2 s4 → refines (or s1 s2) (or s3 s4).
```

```
Lemma refines_or_assoc_l :
 forall (A : Type) (s1 s2 s3 : DS A),
 refines (or (or s1 s2) s3) (or s1 (or s2 s3)).

Lemma refines_or_assoc_r :
 forall (A : Type) (s1 s2 s3 : DS A),
 refines (or s1 (or s2 s3)) (or (or s1 s2) s3).

Lemma refines_or_sym :
 forall (A : Type) (s1 s2 : DS A),
 refines (or s1 s2) (or s2 s1).

Lemma bind_or_distr_r_1:
  forall (A B:Type) (s1 s2: DS A) (s: A → (DS B)),
   (refines (bind (or s1 s2) s) (or (bind s1 s) (bind s2 s))).

Lemma bind_or_distr_r_2:
  forall (A B:Type) (s1 s2: DS A) (s: A → (DS B)),
   (refines (or (bind s1 s) (bind s2 s)) (bind (or s1 s2) s)).

Lemma bind_or_distr_l_1:
 forall (A B:Type) (s: DS A) (s1 s2: A → (DS B)),
   (refines (bind s (fun a ⇒ (or (s1 a) (s2 a)))) (or (bind s s1) (bind s s2))).

Lemma bind_or_distr_l_2:
 forall (A B:Type) (s: DS A) (s1 s2: A → (DS B)),
   (refines (or (bind s s1) (bind s s2)) (bind s (fun a ⇒ (or (s1 a) (s2 a))))).
```

4.2.3 Modules FSLTarski et FSLFixF: fixpoints and well-founded recursion

I do not detail these module here. They have been respectively detailed section 3.1 and 3.2.

4.3 Extending the wp-calculus with state handling

In this section, I first introduce the abstract description of mutable variables. Using these description, I extend the monadic operators to deal with local states using the principles described section 4.1.2. Hence, I build a specification language for state-handling.

4.3.1 Module `Place`: Abstract management of "states"

This module describes a model of state conversions using places. Informally, a place is typically a mutable variable. And a set of place is also a place. Given a type `Env` of environments and a type `A`, (`Place Env A`) is the type of places of type `A` in the environment of type `Env`. In typical use of this module, `A` and `Env` are uples.

4.3.1.1 Definition and examples of places

In this definition, places are defined by two functions `read` and `write` that allow respectively to read the value of the place in the environment, and to write a new value for the place in the environment, creating thus a new environment (a functional denotation of places). The relation between read and write are described by some invariant properties.

```
Record Place (Env A: Type) : Type := {
  read: Env → A ;
  write: Env → A → Env ;
  read_write: forall (e:Env) (x:A), (read (write e x))=x ;
  write_read: forall (e:Env), (write e (read e))=e ;
  write_write: forall (e:Env) (x y:A), (write (write e x) y)=(write e y)
}.
```

I first introduce allPl, the place designing the whole environment. The read function is identity and the write function is **fun** _ x ⇒ x. The Coq definition of this function is interactive.
Definition allPl (A: Type): (Place A A).
Implicit Arguments allPl [A].

I now introduce nonePl, the empty place. The read function is **fun** x ⇒ tt and the write function is **fun** x _ ⇒ x.
Definition nonePl (A: Type): (Place A unit).
Implicit Arguments nonePl [A].

I now introduced the composition of two places, which returns a place.

- read function is **fun** e ⇒ (read v2 (read v1 e))
- write function is **fun** e a ⇒ (write v1 e (write v2 (read v1 e) a)).

Definition liftPl (Env A B: Type) (v1:(Place Env A)) (v2:(Place A B)): (Place Env B).

The composition of two places v1 and v2 is simply noted v1 # v2.
Notation "v1 # v2" := (liftPl v1 v2) (at level 20, right associativity).

The left member of a pair is a place. read is simplify the first projection. write is **fun** e x ⇒ (pairT x (sndT e)).
Definition pairLPl (A B:Type): (Place (prodT A B) A).

Symetrically, the right member of pair is also a place.
Definition pairRPl (A B:Type): (Place (prodT A B) B).

The permutation of a pair is also a place. read is (**fun** p ⇒ (pairT (sndT p) (fstT p))) and write is (**fun** _ p ⇒ (pairT (sndT p) (fstT p))).
Definition pairSPl (A B: Type): (Place (prodT A B) (prodT B A)).

4.3.1.2 Equality of places

I first prove that write function of places is injective on its rightmost argument.
Lemma write_right_inj: **forall** (Env A:Type) (v:(Place Env A)) (e:Env) (x y:A),
(write v e x)=(write v e y) → x=y.

I now introduce an equality relation between places. Two places are equals if and only if their respective write functions are extensionally equals.
Definition eqPl (Env A:Type) (v1 v2:(Place Env A)) : Prop :=
forall (e:Env) (x:A), (write v1 e x)=(write v2 e x).

Lemma eqPl_refl: **forall** (Env A:Type) (v:(Place Env A)), (eqPl v v).

Lemma eqPl_sym: **forall** (Env A:Type) (v1 v2:(Place Env A)), (eqPl v1 v2) → (eqPl v2 v1).

Lemma eqPl_trans: **forall** (Env A:Type) (v1 v2 v3:(Place Env A)),
(eqPl v1 v2) → (eqPl v2 v3) → (eqPl v1 v3).

read functions of equal places are extensionally equals.
Lemma eqPl_read: **forall** (Env A:Type) (v1 v2:(Place Env A)) (e:Env),
(eqPl v1 v2) → (read v1 e)=(read v2 e).

liftPl is compatible with this equality
Lemma liftPl_compat: **forall** (Env A B:Type) (v1a v2a:(Place Env A)) (v1b v2b:(Place A B)),
(eqPl v1a v2a) → (eqPl v1b v2b) → (eqPl (v1a#v1b) (v2a#v2b)).

liftPl is associative with this equality
Lemma liftPl_assoc: **forall** (Env A B C:Type)
(v1:(Place Env A)) (v2:(Place A B)) (v3:(Place B C)),
(eqPl ((v1#v2)#v3) (v1#v2#v3)).

allPl is a neutral element for liftPl
Lemma liftPl_allPl_left: **forall** (Env A:Type) (v:(Place Env A)),
(eqPl (allPl#v) v).

Lemma liftPl_allPl_right: **forall** (Env A:Type) (v:(Place Env A)),
(eqPl (v#allPl) v).

nonePl is the unique place of type unit. In other words, all place of type unit are equals. Let us remark that this property does not hold with allPl because (pairSPl A A) could for instance have the same type than (allPl (A:=(prodT A A))).
Lemma nonePl_uniq_aux: **forall** (Env:Type) (v:(Place Env unit)),
(eqPl v nonePl).

Lemma nonePl_uniq: **forall** (Env:Type) (v1 v2:(Place Env unit)),
(eqPl v1 v2).

4.3.1.3 Bipartition of a state into two places.

We now define a bipartition of a state. Given a state Env, two places vL and vR of respective type (Place Env A) and (Place Env B), (sumPl vL VR) is the type of "proofs" that vL and vR form a bipartition of Env. These proofs introduce a function mkEnv allowing to construct a new value of type Env from a value of type A and a value of type B. Actually, (sumPl vL VR) implies that Env is isomorphic to (prodT A B). This is used in module ISLBlock.

```
Record sumPl (Env A B: Type) (vL:(Place Env A)) (vR:(Place Env B)) : Type :=
 {
   mkEnv: A → B → Env ;
   mkEnv2writeL: forall (e:Env) (x:A), (mkEnv x (read vR e))=(write vL e x) ;
   mkEnv2writeR: forall (e:Env) (x:B), (mkEnv (read vL e) x)=(write vR e x)
 }.
```

The three following lemmas shows that (mkEnv u) is like the constructor of a pair, where as (read vL) and (read vR) are the corresponding destructors.
Lemma mkEnv_readLR: **forall** (Env A B:Type) (vL:Place Env A) (vR:Place Env B)
(u:sumPl vL vR) (e:Env),

(mkEnv u (read vL e) (read vR e))=e.

Lemma readL_mkEnv: **forall** (Env A B:Type) (vL:Place Env A) (vR:Place Env B)
(u:sumPl vL vR) (x:A) (y:B), (read vL (mkEnv u x y))=x.

Lemma readR_mkEnv: **forall** (Env A B:Type) (vL:Place Env A) (vR:Place Env B)
(u:sumPl vL vR) (x:A) (y:B), (read vR (mkEnv u x y))=y.

Hence, by lemma mkEnv_readLR, we could write elements of Env in a "normal form" (mkEnv u x y). The two following lemmas, show how (write vL) and (write vR) can be defined using this normal form.

Lemma writeL_mkEnv: **forall** (Env A B:Type) (vL:Place Env A) (vR:Place Env B)
(u:sumPl vL vR) (x1 x2:A) (y:B),
(write vL (mkEnv u x1 y) x2)=(mkEnv u x2 y).

Lemma writeR_mkEnv: **forall** (Env A B:Type) (vL:Place Env A) (vR:Place Env B)
(u:sumPl vL vR) (x:A) (y1 y2:B),
(write vR (mkEnv u x y1) y2)=(mkEnv u x y2).

Of course, the two places (pairLPl A B) and (pairRPl A B) form a bipartition of (prodT A B).
Definition pairsumPl (A B:Type): (sumPl (pairLPl A B) (pairRPl A B)).

4.3.2 Module ISL: specifications with state and exception handling.

This module defines operators for a Dijsktra specification language with state and exception handling. Monad operators are extended to deal with local states.

The type of specification is also parametrized by St the local type of states and A the type of the result. This definition is the standard implementation of monad with state and exception handling.

Definition DS (St A:Type) := St → WP.DS (prodT St A).

This definition uses WP.DS the DS type defined in module WP. Assuming that st is a state and sp a specification, the term (pre (sp st)) is the precondition of sp on the initial state st. The same remarks applies for (sdemon (sp st)) and (angel (sp st)).

I first define refInEnv the relation of refinement in a environment, using refines relation of WP.

Definition refInEnv (A St: Type) (st:St) (sp1 sp2 : DS St A) : Prop :=
WP.refines (sp1 st) (sp2 st).

For a given st, (refInEnv st) is reflexive and transitive.
Lemma refInEnv_refl :
forall (A St: Type) (st: St) (sp : DS St A), refInEnv st sp sp.

Lemma refInEnv_trans :
forall (A St: Type) (st:St) (sp1 sp2 sp3 : DS St A),
refInEnv st sp1 sp2 → refInEnv st sp2 sp3 → refInEnv st sp1 sp3.

This lemma is the proof rule stating how a refinement goal can be simplified by a wp-computation.
Lemma refInEnv2wp :
forall (A St: Type) (st:St) (sp1 sp2 : DS St A),

```
 (pre (sp2 st) →
  sdemon (sp1 st) (fun a ⇒ post (sp2 st) a)) →
 refInEnv st sp1 sp2.
```

The following lemma is the converse of the preceding one.

```
Lemma refInEnv_from_wp :
 forall (A St: Type) (st:St) (sp1 sp2 : DS St A),
 refInEnv st sp1 sp2 →
 pre (sp2 st) →
 sdemon (sp1 st) (fun a ⇒ post (sp2 st) a).
```

In a refinement proof, the precondition of the refined specification can still be assumed.

```
Lemma refInEnv_pre_intro:
  forall (A St: Type) (st:St) (sp1 sp2 : DS St A),
   ((pre (sp2 st)) → refInEnv st sp1 sp2) → refInEnv st sp1 sp2.
```

Below, I define `val` and `raise` operators which corresponds to their homonym in the algebraic semantics.

```
Definition val (A St: Type) (a : A) : DS St A :=
  fun st ⇒ FSL.val (pairT st a).
```

I extend here `set` and `get` operators of monad, with a place parameter. Hence `set` and `get` are close to the usual constructions of programming language, allowing to affect a reference or to derefence.

```
Definition set (A St :Type) (v:(Place St A)) (x : A) : DS St unit :=
  fun st ⇒ FSL.val (pairT (write v st x) tt).
```

```
Definition get (A St:Type) (v:(Place St A)) : DS St A :=
  fun st ⇒ FSL.val (pairT st (read v st)).
```

The major change with respect to the usual monadic operators is `bind`.

```
Definition bind (A B St1 St2: Type) (v:(Place St1 St2)) (m : DS St2 A) (f : A → DS St1 B) :
  DS St1 B :=
  fun st ⇒
  FSL.bind (m (read v st))
    (fun p ⇒ let (st',a):=p in f a (write v st st')).
```

```
Lemma bind_monotony :
 forall (A B St1 St2: Type) (v1 v2:(Place St1 St2)) (sp1 sp2 : DS St2 A) (f1 f2 : A → DS St1 B)
(st : St1),
 (eqPl v1 v2) →
 refInEnv (read v2 st) sp1 sp2 →
 (forall (st2 : St2) (x : A), refInEnv (write v2 st st2) (f1 x) (f2 x)) →
 refInEnv st (bind v1 sp1 f1) (bind v2 sp2 f2).
```

Here is a notation for IF-Then-Else

```
Notation "'If' [ v ] pb 'Then' sp1 'Else' sp2" :=
  (bind v pb (bool_rect (fun _ ⇒ (DS _ _)) sp1 sp2))
  (at level 20, v, pb, sp1, sp2 at level 90): ISL_scope.
```

```
Lemma if_monotony :
 forall (A St1 St2 : Type) (v: (Place St1 St2)) (spb1 spb2 : DS St2 bool) (sp1 sp2 sp3 sp4 : DS
St1 A) (st : St1),
 refInEnv (read v st) spb1 spb2 →
```

```
(forall st2 : St2, refInEnv (write v st st2) sp1 sp2) →
(forall st2 : St2, refInEnv (write v st st2) sp3 sp4) →
refInEnv st
  (If[ v] spb1 Then sp1 Else sp3)%ISL
  (If[ v] spb2 Then sp2 Else sp4)%ISL.
```

bind is associative. It is expressed by the two following lemmas

```
Lemma bind_assoc_ref1 :
 forall (A B C St1 St2 St3: Type) (v1: (Place St2 St1)) (sp1 : DS St1 A)
  (v2: (Place St3 St2)) (sp2 : A → DS St2 B) (sp3 : B → DS St3 C) (st : St3),
 refInEnv st (bind v2 (bind v1 sp1 sp2) sp3)
   (bind (v2#v1) sp1 (fun a ⇒ bind v2 (sp2 a) sp3)).
```

```
Lemma bind_assoc_ref2 :
 forall (A B C St1 St2 St3: Type) (v1: (Place St2 St1)) (sp1 : DS St1 A)
  (v2: (Place St3 St2)) (sp2 : A → DS St2 B) (sp3 : B → DS St3 C) (st : St3),
   refInEnv st (bind (v2#v1) sp1 (fun a ⇒ bind v2 (sp2 a) sp3))
              (bind v2 (bind v1 sp1 sp2) sp3).
```

Now, let us see "Proof by step-by-step execution" lemmas

```
Lemma bind_simpl_left :
 forall (A B St1 St2: Type) (v:(Place St1 St2)) (sp1 : DS St2 A) (sp2 : A → DS St1 B) (sp3 : DS
St1 B) (st : St1),
 sdemon (sp1 (read v st))
   (fun p ⇒ let (st',a):=p in refInEnv st (bind allPl (set v st') (fun _ ⇒ sp2 a)) sp3)
    → refInEnv st (bind v sp1 sp2) sp3.
```

Here is a "pure" version of the preceding lemma (when "v" is nonePl)

```
Lemma bind_simplPure_left :
 forall (A B St: Type) (v:(Place St unit)) (sp1 : DS unit A) (sp2 : A → DS St B) (sp3 : DS St B)
(st : St),
 sdemon (sp1 tt)
   (fun p ⇒ refInEnv st (sp2 (sndT p)) sp3) → refInEnv st (bind v sp1 sp2) sp3.
```

```
Lemma bind_simpl_right :
 forall (A B St1 St2: Type) (v:(Place St1 St2)) (sp1 : DS St2 A) (sp2 : A → DS St1 B) (sp3 : DS
St1 B) (st : St1),
 (pre (sp1 (read v st)) →
  angel (sp1 (read v st))
    (fun p ⇒ let (st',a):=p in
         refInEnv (write v st st') (bind allPl (set allPl st) (fun _ ⇒ sp3)) (sp2 a)
     )) → refInEnv st sp3 (bind v sp1 sp2).
```

```
Lemma bind_simpl_both :
 forall (A B St1 St2: Type) (v1 v2:(Place St1 St2)) (sp1 sp3 : DS St2 A) (sp2 sp4 : A → DS St1
B) (st : St1),
 (eqPl v1 v2) →
 refInEnv (read v1 st) sp1 sp3 →
  (pre (sp1 (read v1 st)) →
   sdemon (sp1 (read v1 st))
     (fun p ⇒ let (st',a):=p in
```

```
                 refInEnv (write v1 st st') (sp2 a) (sp4 a)))
   → refInEnv st (bind v1 sp1 sp2) (bind v2 sp3 sp4).
```

Here are a few lemmas to simplify "set". There are symmetric lemmas for the right side of refines relation.

```
Lemma set_skip_left :
 forall (A St1 St2 St3: Type) (v1:(Place St1 St2)) (v2:(Place St2 St3)) (sp1 sp2 : DS St1 A) (st
: St1) (x:St3),
 x=(read (v1#v2) st) → refInEnv st sp1 sp2 → refInEnv st (bind v1 (set v2 x) (fun _ ⇒
sp1)) sp2.
```

```
Lemma set_swap_leftright:
 forall (A St1 St2 St3: Type) (v1:(Place St1 St2)) (v2:(Place St2 St3)) (sp1 sp2 : DS St1 A) (st
: St1) (x:St3),
   refInEnv (write (v1#v2) st x) sp1 (bind allPl (set allPl st) (fun _ ⇒ sp2))
     → refInEnv st (bind v1 (set v2 x) (fun _ ⇒ sp1)) sp2.
```

```
Lemma set_simplify_left :
 forall (A St1 St2 St3: Type) (v1:(Place St1 St2)) (v2:(Place St2 St3)) (sp1 sp2 : DS St1 A) (st
: St1) (x:St3),
 refInEnv st (bind v1 (set v2 x) (fun _ ⇒ sp1)) sp1 →
 refInEnv st sp1 sp2 →
 refInEnv st (bind v1 (set v2 x) (fun _ ⇒ sp1)) sp2.
```

Here is the lift operator from state transformer monad. It allows to define `any`, `require`, ... directly from FSL.

```
Definition lift (St A: Type) (wp : WP.DS A) : DS St A :=
  fun st ⇒ FSL.bind wp (fun a ⇒ FSL.val (pairT st a)).
```

```
Lemma bind_lift_changeV_equiv: forall (A B St1 St2 St3:Type)
 (v1:(Place St1 St2)) (v2:(Place St1 St3)) (wp:(WP.DS A)) (sp:A → (DS St1 B)) (st:St1),
  (refInEnv st (bind v1 (lift wp) sp) (bind v2 (lift wp) sp)).
```

`lift` and `set` commutate.

```
Lemma set_permut_lift_left:
  forall (A B St1 St2 St3:Type) (v1:(Place St1 St2)) (v2:(Place St2 St3))
    (wp: WP.DS A) (sp1: A → (DS St1 B)) (sp2: (DS St1 B)) (st:St1) (x:St3),
    (refInEnv st
      (bind nonePl (lift wp) (fun a ⇒ (bind v1 (set v2 x) (fun _ ⇒ (sp1 a))))) sp2)
   →
    (refInEnv st
      (bind v1 (set v2 x) (fun _ ⇒ (bind nonePl (lift wp) sp1))) sp2).
```

```
Definition abort (St A: Type) : DS St A := lift (FSL.abort A).

Definition magic (St A : Type) : DS St A := lift (FSL.magic A).

Definition require (St: Type) (p : Prop) : DS St unit := lift (FSL.require p).

Definition ensure (St: Type) (p : Prop) : DS St unit := lift (FSL.ensure p).

Definition any (St A: Type) : DS St A := lift (FSL.any A).

Lemma any_permut_bind :
```

```
forall (A B C St1 St2: Type) (v:(Place St1 St2)) (sp : DS St2 A) (f : A → B → DS St1 C)
(st : St1),
pre (sp (read v st)) ∨ (NonEmpty B) →
refInEnv st (bind v sp (fun a ⇒ bind nonePl (any B) (fun b ⇒ f a b)))
            (bind nonePl (any B) (fun b ⇒ bind v sp (fun a ⇒ f a b))).
```

4.3.3 Module ISLfriendly: Macros and tactics in the prover

```
Hint Resolve refInEnv_refl bind_val_r_ref1 bind_val_r_ref2: isl.
Hint Resolve bind_monotony if_monotony : isl.
Hint Resolve set_skip_left set_skip_right: isl.
```

4.3.3.1 Simple tactics to simplify refInEnv proofs using wp-calculus

Below, tactics ISL_simplify_all and ISL_Trivial apply wp to the whole goal.

```
Ltac ISL_simplify_all := intros; apply refInEnv2wp; simpl in ⊢ ×.
Ltac ISL_Trivial := try (ISL_simplify_all; intuition eauto ; fail).
```

Tactic ISL_pre_intro introduces the precondition of the right-side of the goal into hypothesis

```
Ltac ISL_pre_intro := apply refInEnv_pre_intro; simpl.
```

Tactic ISL_simplify_both corresponds to bind_simpl_both lemma

```
Ltac ISL_simplify_both :=
  apply bind_simpl_both; simpl in ⊢ ×; auto with isl place.
```

Roughly speaking, ISL_simplify_left corresponds to bind_simpl_left lemma. But, when the first part of the left expression is a set expression, it tries also to eliminate it. If it fails, then it uses set_swap_left_right to move the set expression on the right side of the goal.

```
Ltac ISL_skip_set := apply set_skip_left ; [ simpl; reflexivity | idtac ].
Ltac ISL_simplify_left :=
  (apply bind_simplPure_left || ISL_skip_set ||
    (apply set_simplify_left; [ ISL_Trivial; fail | idtac ] ) ||
      (apply set_swap_leftright;
         try (apply set_simplify_right; [ ISL_Trivial; fail | idtac ] )) ||
        (apply bind_simpl_left; simpl in ⊢ ×;
          try ISL_skip_set)); simpl.
```

Tactic ISL_simplify_right is quite symmetric to ISL_simplify_left, except that, when the right expression is a set expression, it is never moved on the left.

```
Ltac ISL_simplify_right :=
  (apply bind_simplPure_right ||
  (apply set_simplify_right ; [ ISL_Trivial; fail | idtac ] ) ||
      (apply bind_simpl_right; simpl in ⊢ ×;
        try (let z:= fresh "H" in (intro z; ISL_skip_set; generalize z; clear z)))); simpl.
```

4.3.3.2 Tactics using monotony to decompose the goal

```
Ltac ISL_enter_left := eapply bind_enter_left; simpl.
Ltac ISL_enter_right := eapply bind_enter_right; simpl.
Ltac ISL_monotony_left :=
  eapply seq_monotony_left; simpl .
```

```
Ltac ISL_monotony_right :=
  eapply seq_monotony_right; simpl .
Ltac ISL_monotony :=
  (apply if_monotony || apply bind_monotony); simpl; auto with isl place.
```

4.3.3.3 Tactics changing the goal using associativity

```
Ltac ISL_unassoc_left :=
  apply bind_left_unassoc_allPl1 || apply bind_left_unassoc_allPl2 ||
  apply bind_left_unassoc_nonePl2 || apply bind_left_unassoc.
Ltac ISL_unassoc_right :=
  apply bind_right_unassoc_allPl1 || apply bind_right_unassoc_allPl2 ||
  apply bind_right_unassoc_nonePl2 || apply bind_right_unassoc.
Ltac ISL_assoc_left :=
  apply seq_left_assoc_allPl1 || apply seq_left_assoc_allPl2 ||
  apply seq_left_assoc_nonePl2 || apply seq_left_assoc_nonePl1 ||
  apply bind_left_assoc_allPl1 || apply bind_left_assoc_allPl2 ||
  apply bind_left_assoc_nonePl2.
Ltac ISL_assoc_right :=
  apply seq_right_assoc_allPl1 || apply seq_right_assoc_allPl2 ||
  apply seq_right_assoc_nonePl2 || apply seq_right_assoc_nonePl1 ||
  apply bind_right_assoc_allPl1 || apply bind_right_assoc_allPl2 ||
  apply bind_right_assoc_nonePl2.
```

4.3.3.4 Syntactic Sugar

```
Delimit Scope ISL_scope with ISL.
Bind Scope ISL_scope with DS.

Notation "'Val' v" := (val v) (at level 90): ISL_scope.
Notation "'Abort'" := (abort _) (at level 90): ISL_scope.
Notation "v :<- x" := (set v x) (at level 90, right associativity): ISL_scope.
Notation "! v" := (get v) (at level 90): ISL_scope.
Notation "'Any' A" := (any A) (at level 90): ISL_scope.
Notation "'Req' p" := (require p) (at level 90): ISL_scope.
Notation "'Ensure' p" := (ensure p) (at level 90): ISL_scope.
Notation "'Ext' x => f" := (val (fun x => (val f))) (at level 90, f at level 0): ISL_scope.
Notation "'Skip'" := (val tt) (at level 90): ISL_scope.

Notation "'If' [ v ] pb 'Then' sp1 'Else' sp2" :=
  (bind v pb (bool_rect (fun _ => (DS _ _)) sp1 sp2))
  (at level 20, v, pb, sp1, sp2 at level 90): ISL_scope.

Notation "'If' pb 'Then' sp1 'Else' sp2" :=
  (bind allPl pb (bool_rect (fun _ => (DS _ _)) sp1 sp2))
  (at level 20, pb, sp1, sp2 at level 90): ISL_scope.

Notation "'If' / pb 'Then' sp1 'Else' sp2" :=
  (bind nonePl pb (bool_rect (fun _ => (DS _ _)) sp1 sp2))
  (at level 20, pb, sp1, sp2 at level 90): ISL_scope.

Notation "'Bind' x <-[ v ] sp1 'In' sp2" := (bind v sp1 (fun x => sp2))
```

```
  (at level 95, v, sp1 at level 90, right associativity): ISL_scope.
Notation "'Bind' x ← sp1 'In' sp2" := (bind allPl sp1 (fun x ⇒ sp2))
  (at level 95, sp1 at level 90, right associativity): ISL_scope.
Notation "'Bind' x <-/ sp1 'In' sp2" := (bind nonePl sp1 (fun x ⇒ sp2))
  (at level 95, sp1 at level 90, right associativity): ISL_scope.

Notation "[ v ] sp1 -; sp2" := (bind v sp1 (fun _ : unit ⇒ sp2))
  (at level 95, v, sp1 at level 90, right associativity): ISL_scope.
Notation "sp1 -; sp2" := (bind allPl sp1 (fun _ : unit ⇒ sp2))
  (at level 95, right associativity): ISL_scope.
Notation "sp1 /; sp2" := (bind nonePl sp1 (fun _ : unit ⇒ sp2))
  (at level 95, right associativity): ISL_scope.

Notation "f @- arg" := (bind allPl arg (fun x ⇒ bind allPl f (fun g ⇒ g x)))
  (at level 110, left associativity): ISL_scope.

Notation "[< st >] sp1 >==> sp2" := (refInEnv st sp1 sp2)
  (at level 200): ISL_scope.
```

4.3.4 Module ISLBlock: introduction of local mutable variables

In this module, I define operations to define blocks (introduction of local mutable variables).

I first define, a macro Embed.

```
Notation "'Embed' [ v ] sp" := (bind v sp (val (A:=_) (St:=_)))
  (at level 95, v, sp at level 90): ISL_scope.
Notation "'Embed' sp" := (Embed [allPl] sp)%ISL
  (at level 95, sp at level 90): ISL_scope.
```

Section ISLBlock.

Here, v is the "new" variable introduced in the block, bkv is the place of the "old" state. Hence, the state of the block is of type sumPl v bkv. In the future, it is maybe a good idea to simplify all this stuff, and to construct directly a pair of states.

```
Variable St St2 A: Type.
Variable v: Place St A.
Variable bkv: Place St St2.

Definition block (B: Type) (u:sumPl v bkv) (x:A) (sp: (DS St B)): (DS St2 B)
  := fun (st:St2) ⇒
       (FSL.bind (sp (mkEnv u x st)) (fun p ⇒
                         FSL.val (pairT (read bkv (fstT p)) (sndT p)))).
```

Here is a proof rule when block in the left part of a refinement goal. The refinement goal is simply lifted in a bigger state, where the new variable is simply not changed by the right part of the goal. A similar rule can be applied when block is in the right part of the goal.

```
Lemma block_enter_left:
  forall (B:Type) (u:sumPl v bkv) (x:A) (sp1: (DS St B)) (sp2:(DS St2 B)) (st:St2),
```

```
  ([< mkEnv u x st >] (Bind a ← sp1 In v :<- x -; Val a) >==> (Embed [bkv] sp2))%ISL
   → ([< st >] (block u x sp1) >==> sp2)%ISL.
```

The following rule expresses monotony of block w.r.t refinement.

```
Lemma block_monotony:
 forall (B:Type) (u:sumPl v bkv) (x:A) (sp1: (DS St B)) (sp2:(DS St B)) (st:St2),
  ([< mkEnv u x st >] (Bind a ← sp1 In v :<- x -; Val a) >==> (Bind a ← sp2 In v :<- x -;
Val a))%ISL
   → ([< st >] (block u x sp1) >==> (block u x sp2))%ISL.
```

In these rules for handling blocks, data-refinement misses.

End `ISLBlock`.

4.3.5 Module `AbsPost`: Hoare Specifications

In this module, I define operations to handle specifications a la Hoare in ISL.

A Hoare specification is a black-box specification: a specification in which the result or the final state is only determined by some predicates. I propose here two kind of Hoare specifications: Hoare specifications that may update the state, and purely functional Hoare specifications. Of course, the second ones are a special case of the first ones. As in section 2.3.1.2, Hoare specifications are decomposed into a precondition and an abstract postcondition.

4.3.5.1 Hoare specifications without exception raising.

With respect to section 2.3.1.2, I now consider that a such Hoare specification may work only on a place of the current state.

Section `UPD_ABSPOST`.

I assume `Exc` the type of exceptions, `St` the type of global state, `St2` the type of local state. `v` is the place on which works the abstract postcondition. `A` is the type of the returned result. `q` is the postcondition relating the final state and the result.

Variable `St St2: Type`.

Variable `v: (Place St St2)`.

Variable `A: Type`.

Variable `q: St2 → A → Prop`.

The abstract postcondition of `v` and `q` is defined as follows:

```
Definition upd_abspost: (DS St A)
  := (Bind result <-/ (Any A) In
               [v](Bind new_st <-/ (Any St2) In
                     (allPl :<- new_st) -;
                     (Ensure (q new_st result))) -;
                   (Val result))%ISL.
```

In the following lemmas, I assume an initial state `old_st`.

Variable `old_st: St`.

Here is a rule stating how to prove that a specification `pt` refines an abstract postcondition using wp-calculus. This rule assume that the current state `tmp_st` has been obtained from `old_st` only by modifications in `v`. Hence, the current goal is equivalent to (`refInEnv old_st (allPl :<- tmp_st -; pt) upd_abspost`) This rule generalizes the usual one which assumes `tmp_st=old_st`. This generalization is useful in proofs using small-steps simplifications, because goals of the form "`refInEnv old_st pt0 upd_abspost`" can be simplified before to use this rule.

```
Lemma updabspost2wp:
  forall (pt:(DS St A)) (tmp_st: St),
  tmp_st=(write v old_st (read v tmp_st)) →
  (sdemon (pt tmp_st)
   (fun p ⇒ let (new_st,a):=p in
      new_st = (write v old_st (read v new_st))
      ∧ (q (read v new_st) a)))
    → (refInEnv tmp_st pt upd_abspost).
```

This lemma corresponds to a "refinement in a context" rule. In a goal (`refInEnv old_st sp1 sp2`) such `sp1` satifies a postcondition `q`, it is possible to strengthen `sp1` with a clause ensuring that the result and the final state of `sp1` satisfy `q`.

```
Lemma intro_updabspost:
  forall (sp1 sp2:(DS St A)),
   (refInEnv old_st sp1 upd_abspost)
      → (refInEnv old_st (Bind a ← sp1 In
                           (Bind new_st ← !allPl In
                            (Ensure ((q (read v new_st) a)
                                     ∧ new_st=(write v old_st (read v new_st))))) -;
                          (Val a))
                       sp2)%ISL
  → (refInEnv old_st sp1 sp2)%ISL.
```

This rule is a generalization of the preceding one.

```
Lemma intro_bind_updabspost:
  forall (B:Type) (sp1:(DS St A)) (sp2: A → (DS St B)) (sp3: (DS St B)),
    (refInEnv old_st sp1 upd_abspost)
        → (refInEnv old_st (Bind a ← sp1 In
                              (Bind new_st ← ! allPl In
                                (Ensure ((q (read v new_st) a)
                                 ∧ new_st=(write v old_st (read v new_st))))) -;
                            (sp2 a))
                         sp3)%ISL
  → (refInEnv old_st (bind allPl sp1 sp2) sp3)%ISL.

End UPD_ABSPOST.
```

4.3.5.2 Pure Hoare specifications.

Given a type `A` and a predicate `q:A → Prop`, I now define `pure_abspost q` as a specification equivalent to (`upd_abspost _ nonePl (fun _ ⇒ q)`)). However, it is interesting to simplify the corresponding specification.

```
Section PURE_ABSPOST.
```

```
Variable St A: Type.

Variable q: A → Prop.

Definition pure_abspost: (DS St A)
  := (Bind result <-/ (Any A) In
            (Ensure (q result)) /;
                  (Val result))%ISL.
```

The rules of the preceding section can be specialized to this case. I do not give the details.
...

```
End PURE_ABSPOST.
```

4.3.5.3 Rules for Hoare specifications.

```
Lemma updabspost_monotony: forall (St St2 A:Type) (v1 v2:(Place St St2)) (q1 q2: St2 → A
→ Prop) (st:St),
  (eqPl v1 v2) →
   (forall (st2:St2) (a:A), (q1 st2 a) → (q2 st2 a)) →
    (refInEnv st (upd_abspost v1 q1) (upd_abspost v2 q2)).

Lemma updabspost_split: forall (St St2 A:Type) (v:(Place St St2)) (q1 q2: St2 → A → Prop)
  (st:St) (sp:(DS St A)),
   (refInEnv st sp (upd_abspost v q1))
     → (refInEnv st sp (upd_abspost v q2))
      →
    (refInEnv st sp (upd_abspost v (fun (stf:St2) (r:A) ⇒ (q1 stf r) ∧ (q2 stf r)))).

Lemma updabspost_splitforall: forall (St St2 A B:Type) (v:(Place St St2))
  (p: B → Prop) (q: B → St2 → A → Prop)
  (st:St) (sp:(DS St A)),
   (exists x:B, (p x)) →
   (forall (x:B), (p x) → (refInEnv st sp (upd_abspost v (q x)))) →
    (refInEnv st sp (upd_abspost v (fun (stf:St2) (r:A) ⇒ forall (x:B), (p x) → (q x stf r)))).

Lemma embed_updabspost_ref1: forall (St St2 St3 A:Type) (v1:(Place St St2)) (v2:(Place St2
St3))
 (q: St3 → A → Prop) (st:St),
  (refInEnv st (Embed [v1] (upd_abspost v2 q))%ISL (upd_abspost (v1#v2) q)).

Lemma bind_updabspost_ref1:
 forall (St St2 St3 A B:Type) (v1:(Place St St2)) (v2:(Place St2 St3))
  (q: St3 → A → Prop) (st:St) (sp: A → (DS St B)),
  (refInEnv st
      (bind v1 (upd_abspost v2 q) sp)
      (bind allPl (upd_abspost (v1#v2) q) sp)).

Lemma embed_updabspost_ref2: forall (St St2 St3 A:Type) (v1:(Place St St2)) (v2:(Place St2
St3))
 (q: St3 → A → Prop) (st:St),
  (refInEnv st (upd_abspost (v1#v2) q) (Embed [v1] (upd_abspost v2 q))%ISL).

Lemma bind_updabspost_ref2:
```

```
forall (St St2 St3 A B:Type) (v1:(Place St St2)) (v2:(Place St2 St3))
  (q: St3 → A → Prop) (st:St) (sp: A → (DS St B)),
  (refInEnv st
      (bind allPl (upd_abspost (v1#v2) q) sp)
      (bind v1 (upd_abspost v2 q) sp)).
```

```
Lemma embed_pureabspost_ref2: forall (St St2 A:Type) (v:(Place St St2)) (q: A → Prop)
(st:St),
  (refInEnv st (pure_abspost q)
               (Embed [v] (pure_abspost q))%ISL).
```

```
Lemma updabspost_liftPl_ref1: forall (St St2 St3 A:Type) (v1:(Place St St2)) (v2:(Place St2
St3))
 (q: St3 → A → Prop) (tmp_st old_st:St) (st:St2),
  tmp_st=(write v1 old_st st)
  → (refInEnv tmp_st
       (upd_abspost v1 (fun (e':St2) (x:A) ⇒ e'=(write v2 st (read v2 e'))
                              ∧ (q (read v2 e') x)))
       (upd_abspost (v1#v2) q)).
```

```
Lemma updabspost_liftPl_ref2: forall (St St2 St3 A:Type) (v1:(Place St St2)) (v2:(Place St2
St3))
 (q: St3 → A → Prop) (tmp_st old_st:St) (st:St2),
  tmp_st=(write v1 old_st st)
  → (refInEnv tmp_st
       (upd_abspost (v1#v2) q)
       (upd_abspost v1 (fun (e':St2) (x:A) ⇒ e'=(write v2 st (read v2 e'))
                              ∧ (q (read v2 e') x)))).
```

```
Unset Implicit Arguments.
```

4.3.6 Module ISLWhile: while operator

I do not detail this module here. This module introduce the construction of `while` and `whileWF` described sections 2.3.3.4 and 3.3.

4.4 Small examples

We have already seen other examples of high-order specifications: fixpoint operators such `whileWF`. Furthermore, the ISL library has been use to treat the simple `lookup` example (with high-order refinement reasoning) presented chapter 2. I do not detail it here. I also only mention the development of two others high-order examples: enumeration of binary trees and enumeration of prime number, using in both cases a continuation-based implementation (see [Bou05]).

I only present here a simple example, which do not use high-order functions, but which is a classical example of proof in Hoare logic: the sort of an array of size n using an $\mathcal{O}(n^2)$ algorithm.

The algorithm proceeds by successive permutations of the $k-th$ maximum with the element of index $n-k$ (indexes being between 0 and $n-1$). This example allows to check that the "practical complexity" of the proof remains the same than if I were use an interactive Hoare-logic prover.

First, I give ISL functions to access and store values in a array. I interpret here arrays as functions.

```
Section array.
```

```
Variable A: Type.

Record StArray : Type := {
  length: Z ;
  abs_array: (AbsArray A)
}.

Identity Coercion AbsArray2Fun: AbsArray >-> Funclass.
Coercion abs_array: StArray >-> AbsArray.

Definition array: (Place StArray (AbsArray A)).
```

Let us note that `access` function (and similarly `store` and `swap`) below, the array is passed "by variable" (and not "by copy"). This possible here, because places are first class values in our model. And user proofs will not involve aliases problem (e.g. user will not have to reason about equality or inequality of places) because these functions have a single place argument.

```
Definition access (Env: Type) (t:(Place Env StArray)) (i:Z) : (DS Env A) :=
   (Bind tv ← !t In
    If / (Val ((Zle_bool 0 i)%Z && (Zlt_bool i (length tv))))%bool
    Then (Val (tv i))
    Else Abort)%ISL.

Lemma access_ok:
  forall (Env: Type) (t:(Place Env StArray)) (i:Z) (e:Env),
   (0 ≤ i < (length (read t e)))%Z →
     ([< e >] (access t i) >==> (Val ((read t e) i)))%ISL.

Definition store (Env: Type) (t:(Place Env StArray)) (i:Z) (a:A): (DS Env unit) :=
   (Bind tv ← !t In
    If / (Val ((Zle_bool 0 i)%Z && (Zlt_bool i (length tv))))%bool
    Then
      t#array :<- (fun j ⇒ (if (Zeq_bool i j) then a else (tv j)))
    Else
      Abort)%ISL.

Lemma store_ok:
  forall (Env: Type) (t:(Place Env StArray)) (i:Z) (a:A) (e:Env),
   (0 ≤ i < (length (read t e)))%Z →
     ([< e >] (store t i a) >==>
        t#array :<- (fun j ⇒ (if (Zeq_bool i j) then a else (read t e j))))%ISL.

Definition swap (Env: Type) (t:(Place Env StArray)) (i j:Z): (DS Env unit) :=
  (Bind ti ← (access t i) In
    Bind tj ← (access t j) In
      (store t i tj) -; (store t j ti))%ISL.

Lemma swap_ok:
  forall (Env: Type) (t:(Place Env StArray)) (i j:Z) (e:Env),
    (0 ≤ i < (length (read t e)))%Z → (0 ≤ j < (length (read t e)))%Z
     → ([< e >] (swap t i j) >==>
          (upd_abspost (t#array)
          (fun new_t result ⇒ (Exchange (read t e) new_t i j))))%ISL.
```

```
Lemma swap_permutation:
  forall (Env: Type) (t:(Place Env StArray)) (i j:Z) (e:Env),
    (0 ≤ i < (length (read t e)))%Z → (0 ≤ j < (length (read t e)))%Z
    → ([< e >] (swap t i j) >==>
       (upd_abspost (t#array)
          (fun new_t result ⇒
             (permutation (read t e) new_t 0 ((length (read t e))-1)))))%ISL.

End array.
```

Maximum sort is then decomposed in two functions: `maximum` which computes the index of the maximum in a slice of an array, and `maxisort` that sorts the slice of an array. The array is a global variable shared by these two functions. It corresponds to `arrayglob` place defined below. The slice is represented by two indexes `a` (the first index of the slice) and `b` (the second index of the slice) which are parameters of these functions.

```
Definition arrayglob (A:Type) := (pairRPl A (StArray Z)).
Implicit Arguments arrayglob [A].

Section MaximumDef.
```

Function `maximum` introduces two local variables `ind` (an index, such that successive indexes accessed by `maximum` lies in the interval `a..ind`), and `res` (the index of the maximum in `a..ind`).

```
Record place_maximum: Type := {
  res: Z;
  ind: Z
}.
```

Place `vloc` corresponds to the pair of this two local variables and place `vglob` is the global array variable.

```
Definition vloc := (pairLPl place_maximum (StArray Z)).
Definition vdecl: (sumPl vloc arrayglob) := (pairsumPl place_maximum (StArray Z)).

Definition maximum (a b:Z) : (DS (StArray Z) Z) :=
 (block vdecl (Build_place_maximum a a)
    (Bind init_t ← !arrayglob In
    (whileWF
       (* condition to continue *)
       (Bind s ← !vloc In (Val (Zlt_bool (ind s) b)))
       (* body *)
       (Bind s ← !vloc In
          let nind := ((ind s)+1)%Z in
           let r := (res s) in
             If (Bind tr ← (access arrayglob r) In
                 Bind tn ← (access arrayglob nind) In
                   (Val (Zlt_bool tr tn)%Z))
             Then
               vloc :<- (Build_place_maximum nind nind)
             Else
               vloc :<- (Build_place_maximum (res s) nind))
       (* invariant *)
       (fun e ⇒
```

```
        let r:=(res (read vloc e)) in
        let i:=(ind (read vloc e)) in
        let t:=(read arrayglob e) in
          (a ≤ i ≤ b)%Z ∧ (a ≤ r ≤ b)%Z ∧ (Maximize t (t r) a i) ∧ t=init_t)
      (* variant *)
      (fun x y ⇒ (Zwf_up b (ind (read vloc x)) (ind (read vloc y))))
    ) -;
    Bind s ← !vloc In (Val (res s)))
  )%ISL.
```

```
Lemma maximum_spec:
  forall (a b:Z) (t:(StArray Z)),
    ((0 ≤ a ≤ b)%Z ∧ (b < (length t))%Z) →
      ([< t >] (maximum a b)
               >==> (pure_abspost (fun r ⇒ ((a ≤ r ≤ b)%Z
                                            ∧ (Maximize t (t r) a b%Z))))
      )%ISL.
```

```
End MaximumDef.
```

```
Section MaxisortDef.
```

Function `maxisort` introduces a local variable represented as `vloc2` place below. This local variable describes the slice !`vloc2..b` which is sorted at each step of the loop.

```
Definition vloc2 := (pairLPl Z (StArray Z)).
Definition vdecl2: (sumPl vloc2 arrayglob) := (pairsumPl Z (StArray Z)).
```

```
Definition maxisort (a b:Z) : (DS (StArray Z) unit) :=
 (block vdecl2 b
    (Bind init_t ← !arrayglob In
    (whileWF
      (* condition to continue *)
      (Bind i ← !vloc2 In (Val (Zle_bool a i)))
      (* body *)
      (Bind i ← !vloc2 In
       Bind r <-[arrayglob] (maximum a i) In
         (swap arrayglob i r) -;
         (vloc2 :<- (i-1)%Z))
      (* invariant *)
      (fun e ⇒
        let i:=(read vloc2 e) in
        let t:=(read arrayglob e) in
              ((a-1) ≤ i ≤ b)%Z
          ∧ ((i < b)%Z → Maximize t (t (i+1))%Z a i)
          ∧ (length t)=(length init_t)
          ∧ (Sorted t (i+1)%Z b)
          ∧ (permutation t init_t a b)
       )
      (* variant *)
      (fun x y ⇒ (Zwf a (read vloc2 x) (read vloc2 y)))
    )))%ISL.
```

```
Lemma maxisort_spec:
```

```
forall (a b:Z) (t:(StArray Z)),
  ((0 ≤ a ≤ b)%Z ∧ (b < (length t))%Z) →
    ([< t >] (maxisort a b) >==>
      (upd_abspost (array Z)
        (fun new_t _ ⇒ ((Sorted new_t a b) ∧ (permutation new_t t a b)))))%ISL.
End MaxisortDef.
```

4.5 Future works on the library

There is left a lot of work. The “todo list” seems even too long to be completely done. However, I try here to briefly sketch such a list.

- Rules for data-refinement.[**REGARDER von WRIGHT**].
- Rules for exception. In particular, it remains to invent a way to have a modular code: introduce new exceptions in the context. This seems actually more simple than for states, because the set of exception only grows (it never decreases) because there is no “local” exception.
- Provide reasoning rules for equivalence of specifications.
- Optimization of tactics.
- Develop larger examples.

To try larger examples, it would be convenient to have a front-end allowing to write specifications using a concrete syntax. I develop this idea in the next section.

4.6 Type inference of state conversions

This section sketches a mechanism to help the user in writing imperative specifications. Actually, I only present here an algorithm to infer information describing state conversions. The correction of this algorithm is not given.

The inference algorithm presented here is directly inspired from Filliâtre’s thesis. In his work, Filliâtre proposes a type system to infer effects for a programming language close to mini-ML, which rejects programs with aliases, but allows references as parameters. Actually, the purpose of this type system in Filliâtre works is very close to my preoccupation: it is involved in the generation of proof obligations in Coq for Hoare Logic.

However, my objectives are different from his ones. This leads me to change the requirement on the type system.

- The distinction between read and modified variable is not crucial for me, whereas it is crucial in Hoare Logic. Indeed, in Hoare Logic, function specifications must declare the variable that are not modified. This is not necessary with refinement, because only variable that are modified by the (imperative) specification can be modified. Eliminating this distinction simplifies the type system, and especially the generated Coq terms.
- I need to express high-order functions with side-effects. Filliâtre system allows only a limited form high-order functions with side-effects. For instance, function f taking a function g as parameter must declare the *exact* state on which works g, and they can not take this state as a generic parameter. I relax here these constraints by a subtyping mechanism (hence, f can declare a bigger state than the effective state on which g works), and a quantification over state (a form of effect polymorphism).

4.6.1 UAML--: a simplified version of Filliâtre's system

In this section, I present a simplified version of Filliâtre's system. These simplifications are the following: I only consider programs (and I do not consider specifications), and I do not distinguish modified variables from needed variables. But, of course, I keep restrictions that forbids aliases. In the following, I name this system by UAML--.

4.6.1.1 Variables and states

As a variable in logic, a UAML-- variable names the result of a calculus. Hence, they can not be updated, they have no state. To represent states, UAML-- uses a special data-type called reference. An object of type $\mathbf{ref}(\tau)$ is a state which contents is of type τ. If x is a variable of type $\mathbf{ref}(\tau)$, the contents of the variable, denoted $!x$ is syntactically distinct from the variable x itself. In the following variables are named x, y or z.

Unlike ML, UAML-- does not allow aliasing: each state of the program is denoted by a unique variable. This restriction is guaranteed by typing rules. Concretely, restrictions on references are the following. As effective parameter a reference must be a variable and must not already be read or modified by the function to which it is applied. In other contexts, references must be new, or in other words, references must not be already denoted by a variable. References of reference are forbidden.

As their usage is very restricted with respect to ML, references in UAML-- looks like ordinary variables of Pascal. The main differences with respect to Pascal are the following. First, Pascal do not forbid aliases in parameter-passing by variable. Second, UAML-- allows to write functions that returns new references. This possibility seems interesting if we want to generalize references of UAML-- to arrays or objects: in OO language, these functions correspond to constructors of objects.

4.6.1.2 Types and typing contexts

UAML-- type system is based on effects (see [TJ92]). However, it uses only a very simple version of effects. In UAML--, an effect ϵ is a set of variables. It represents a set of states that are potentially accessed or modified by an expression.

Definition 4.6.1 (Value types and computation types) *Abstract syntax for types of values and types of computations is given by the following mutually recursive definition:*

$$\begin{array}{ll} \textit{values} & \tau ::= \beta \mid \forall x:\tau.\kappa \mid \mathbf{ref}(\tau) \\ \textit{computations} & \kappa ::= (\tau, \epsilon) \end{array}$$

where

- β is any base type. Among base type we assume a type `unit` and a type `bool`.
- $\forall x:\tau.\kappa$ is a functional type. The symbol x is a variable which is bound in κ.
- $\mathbf{ref}(\tau)$ is type of references of type τ.

Hence, UAML-- provides a restricted form of dependent-types. Indeed, the only substitution mechanism in this type system is the substitution of a variable by another variable. This is used to express reference-passing parameters. In other words, the dependencies in functional types like $\forall x:\tau.\kappa$ occur only on reference-passing parameters (i.e when τ is the reference type). Other kind of parameters are never bound in types. This is formalized in the definition of well-formed types (see definition 4.6.5). In examples of section 4.6.1.4, I use the notation $\tau \to \kappa$ to express the type $\forall x:\tau,\kappa$ where $x \notin FV(\kappa)$.

Definition 4.6.2 (Free variables of a type) *I now define the set of free variables of a type. It* *is defined by mutual induction on type of values and type of computations:*

$$\begin{array}{rclcrcl} FV(\beta) & \stackrel{def}{=} & \emptyset & \quad & FV(\forall x:\tau.\kappa) & \stackrel{def}{=} & FV(\tau) \cup (FV(\kappa)\backslash\{x\}) \\ FV(\mathbf{ref}(\tau)) & \stackrel{def}{=} & FV(\tau) & & FV((\tau,\epsilon)) & \stackrel{def}{=} & FV(\tau) \cup \epsilon \end{array}$$

Definition 4.6.3 (Substitutions on types) *Let x and y be variables. If ϵ is an effect, I define the substitution $\epsilon[x \leftarrow y]$ as:*

$$\begin{array}{rcl} \epsilon[x \leftarrow y] & \stackrel{def}{=} & \epsilon \text{ if } x \notin \epsilon \\ \epsilon[x \leftarrow y] & \stackrel{def}{=} & (\epsilon\backslash\{x\}) \cup \{y\} \text{ if } x \in \epsilon \end{array}$$

I now define the substitutions of x by y in types of values $\tau[x \leftarrow y]$ and in types of computations $\kappa[x \leftarrow y]$, by mutual induction on τ and κ.

$$\begin{array}{rcl} \beta[x \leftarrow y] & \stackrel{def}{=} & \beta \\ \mathbf{ref}(\tau)[x \leftarrow y] & \stackrel{def}{=} & \mathbf{ref}(\tau[x \leftarrow y]) \\ (\forall z:\tau.\kappa)[x \leftarrow y] & \stackrel{def}{=} & \forall z':\tau[x \leftarrow y].\kappa[z \leftarrow z'][x \leftarrow y] \text{ with } z' \notin FV(\kappa) \cup \{x, y\} \\ (\tau,\epsilon)[x \leftarrow y] & \stackrel{def}{=} & (\tau[x \leftarrow y], \epsilon[x \leftarrow y]) \end{array}$$

As usual in this kind of definitions, substitution under a bound variable may require a renaming of the bound variable in order to avoid capture. In the following rules, equality between types is thus implicitly α-equivalence: syntactic equality modulo renaming of bound variables. I do not formalize this point here.

Definition 4.6.4 (Typing contexts) *Typing contexts are defined as lists of couple variable/type. Their purpose is to store the type associated to each variable during type-checking.*

$$\Gamma ::= \emptyset \mid \Gamma; x : \tau$$

As usual, in a context "$\Gamma; x : \tau$", names introduced in Γ are bound in τ.
I define $x : \tau \in \Gamma$ as a predicate satisfying the following rules:

$$\frac{}{x:\tau \ \in \ \Gamma; x:\tau} \qquad \frac{x:\tau \in \Gamma \quad x \neq y}{x:\tau \ \in \ \Gamma; y:\tau'}$$

Furthermore, I define the set $Refs(\Gamma)$ of references in Γ.

$$Refs(\Gamma) \stackrel{def}{=} \{x \mid \exists \tau. x : \mathbf{ref}(\tau) \in \Gamma\}$$

I will now introduce well-formness judgment for types "$\Gamma \vdash \tau$ wf" and for contexts "Γ wf". Well-formness is a property of contexts which is preserved during typechecking. In particular, we need the following properties:

- if Γ wf and $\Gamma \vdash \tau$ wf then $FV(\tau) \subseteq Refs(\Gamma)$.
- if $\Gamma \vdash \mathbf{ref}(\tau)$ wf then τ is not a reference type.
- if $\Gamma; \Delta$ wf and $\Gamma \vdash \tau$ wf then $\Gamma; \Delta \vdash \tau$ wf.
- if Γ wf and $x : \tau \in \Gamma$ then $\Gamma \vdash \tau$ wf

In order to guarantee these two last properties, I forbid to hide references in well-formed contexts. To simplify the rules, I assume in the following that bound variables are not hiding references of the context (typically expressed by a constraint $x \notin Refs(\Gamma)$). This is not restrictive, because as explained above, these rules must be understood modulo α-equivalence. Hence, it is still possible to rename bound variables to satisfy these constraints.

Definition 4.6.5 (Types well-formness) *Judgments* $\Gamma \vdash \tau$ wf *and* $\Gamma \vdash \kappa$ wf *are mutually defined by:*

$$\frac{}{\Gamma \vdash \beta \text{ wf}} \qquad \frac{\Gamma \vdash \tau \text{ wf} \quad \tau \neq \mathbf{ref}(\_)}{\Gamma \vdash \mathbf{ref}(\tau) \text{ wf}} \qquad \frac{\Gamma \vdash \tau \text{ wf} \quad \Gamma; x : \tau \vdash \kappa \text{ wf} \quad x \notin Refs(\Gamma)}{\Gamma \vdash \forall x : \tau . \kappa \text{ wf}}$$

$$\frac{\Gamma \vdash \tau \text{ wf} \quad \epsilon \subseteq Refs(\Gamma)}{\Gamma \vdash (\tau, \epsilon) \text{ wf}}$$

Definition 4.6.6 (Context well-formness) *Well-formness judgment* Γ wf *is defined by:*

$$\frac{}{\emptyset \text{ wf}} \qquad \frac{\Gamma \text{ wf} \quad \Gamma \vdash \tau \text{ wf} \quad x \notin Refs(\Gamma)}{\Gamma; x : \tau \text{ wf}}$$

4.6.1.3 Expressions and type inference

I start here by defining the abstract syntax of the UAML--. This is a typed lambda-calculus with references. Constants represent predefined values such `true` and `false`. In the following, the type of each constant c is given by $Type(c)$.

Definition 4.6.7 (Expressions) *The abstract syntax of* UAML-- *expressions is given by the following definition:*

e	$::=$	c	constants
	$\mid$	x	variable
	$\mid$	$\mathbf{ref}\ e$	allocation of a reference
	$\mid$	$x := e$	affectation of a reference
	$\mid$	$!x$	dereference
	$\mid$	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	composition of expressions (x is bound in e_2)
	$\mid$	$\lambda x : \tau .\ e$	functional values (x is bound in e)
	$\mid$	$(e_1\ e_2)$	application of a function e_1 to its argument e_2
	$\mid$	$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	conditional

Syntactically, the set of free variables of an expression e is defined as usual. Given an expression e, two variables x and y, I need also to define a substitution operation $e[x \leftarrow y]$. This operation is defined as usual by renaming bound variables. The sequence is not present in this language, but we can define it as a macro. If $x \notin FV(e_2)$, we set:

$$e_1; e_2 \stackrel{def}{=} ((\lambda x : \texttt{unit}.e_2)\ e_1)$$

The operational semantics of this language corresponds to the standard semantics of ML. Evaluation is strict: actual parameters are evaluated before to be substituted in the body of the function. And values of references are location in a “heap”. In particular, passing a reference as parameter is passing a location of the heap. Formally, this semantics is given as a small-step semantics [WF94]. The type-system below restricts the usage of references (aliases are forbidden). Hence, for well-typed expressions, this semantics can be equivalently changed for an other, where:

- references are allocated in a “stack”: they live only in their lexical scope.
- alive references correspond to distinct locations of the stack.
- reference parameters are passed by copy: their contents is copied before the call into the formal parameter (a newly allocated reference), and copied back from the formal parameter into the actual one when the called function returns.

Definition 4.6.8 (Typing judgment) *Given an expression* e*, the typing judgment* $\Gamma \vdash e : \kappa$ *is defined by the rules figure 4.1*

$$\frac{}{\Gamma \vdash c : (\mathit{Type}(c), \emptyset)}\textbf{CONST} \qquad \frac{x : \tau \in \Gamma \quad \tau \neq \mathbf{ref}(\_)}{\Gamma \vdash x : (\tau, \emptyset)}\textbf{VAR}$$

$$\frac{x : \mathbf{ref}(\tau) \in \Gamma \quad \Gamma \vdash e : (\tau, \epsilon)}{\Gamma \vdash x := e : (\mathtt{unit}, \epsilon \cup \{x\})}\textbf{SET} \qquad \frac{x : \mathbf{ref}(\tau) \in \Gamma}{\Gamma \vdash !x : (\tau, \{x\})}\textbf{GET}$$

$$\frac{\Gamma \vdash e : (\tau, \epsilon) \quad \tau \neq \mathbf{ref}(\_)}{\Gamma \vdash \mathbf{ref}\ e : (\mathbf{ref}(\tau), \epsilon)}\textbf{NEW}$$

$$\frac{\Gamma \vdash \tau \text{ wf} \quad \Gamma; x : \tau \vdash e : \kappa \quad x \notin \mathit{Refs}(\Gamma)}{\Gamma \vdash \lambda x : \tau.e \ : \ \forall x : \tau.\kappa}\textbf{FUN}$$

$$\frac{\Gamma \vdash e_1 : (\forall x : \tau_2.(\tau_1.\epsilon_1), \epsilon) \quad \Gamma \vdash e_2 : (\tau_2, \epsilon_2) \quad \tau_2 \neq \mathbf{ref}(\_)}{\Gamma \vdash (e_1\ e_2) : (\tau_1, \epsilon \cup \epsilon_1 \cup \epsilon_2)}\textbf{APP}$$

$$\frac{\Gamma \vdash e_1 : (\forall x : \mathbf{ref}(\tau_2).(\tau_1, \epsilon_1), \epsilon) \quad y : \mathbf{ref}(\tau_2) \in \Gamma \quad y \notin \mathit{FV}((\tau_1, \epsilon_1)) \backslash \{x\}}{\Gamma \vdash (e_1\ y) : (\tau_1[x \leftarrow y], \epsilon \cup \epsilon_1[x \leftarrow y])}\textbf{APPREF}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \epsilon_1) \quad \tau_1 \neq \mathbf{ref}(\_) \quad \Gamma; x : \tau_1 \vdash e_2 : (\tau_2, \epsilon_2) \quad x \notin \mathit{Refs}(\Gamma)}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : (\tau_2, \epsilon_1 \cup \epsilon_2)}\textbf{LET}$$

$$\frac{\Gamma \vdash e_1 : (\mathbf{ref}(\tau_1), \epsilon_1) \quad \Gamma; x : \mathbf{ref}(\tau_1) \vdash e_2 : (\tau_2, \epsilon_2) \quad x \notin \mathit{Refs}(\Gamma) \cup \mathit{FV}(\tau_2)}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : (\tau_2, \epsilon_1 \cup (\epsilon_2 \backslash \{x\}))}\textbf{LETREF}$$

$$\frac{\Gamma \vdash e_1 : (\mathtt{bool}, \epsilon_1) \quad \Gamma \vdash e_2 : (\tau, \epsilon_2) \quad \Gamma \vdash e_3 : (\tau, \epsilon_3)}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : (\tau, \epsilon_1 \cup \epsilon_2 \cup \epsilon_3)}\textbf{COND}$$

Figure 4.1: Type system of UAML--

Let me explain some of these rules:

VAR This rule forbids expressions to return reference variables. Otherwise, aliases could be created by this way. The context in which a reference variable can be an expression is controlled in other rules.

APP There are two rules for application: **APP** is application to expressions which are not references.

APPREF As an effective parameter, a reference can only be a variable. The condition $y \notin FV((\tau_1, \epsilon_1)) \backslash \{x\}$ ensure that this variable is not already used by the function which applies to it. Otherwise, an alias would created.

LET There are two rules for let-expression: **LET** is the binding of an expression which is not a reference.

LETREF A let-expression on a reference variable x corresponds to the allocation of a new reference named x, and e_1 corresponds to the initialization of x. Other rules actually ensure that the reference initialized by e_1 is not used in e_1. The condition $x \notin \tau_2$ and the effect returned by the let-expression ensure that x can not escape from its lexical scope. It can thus be safely destruct at the end of this block.

Let me remark that when e_1 returns a value of type $\mathbf{ref}(\tau_1)$, $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ is not equivalent to $(\lambda x : \mathbf{ref}(\tau_1).e_2\ e_1)$. Indeed, when one form of these expression is typable, the other is not: in the first one, e_1 can not be a variable (because of rule **VAR**), whereas in the second one, e_1 must be a variable (rules **APP** and **APPREF**).

Proposition 4.6.1 (Basic properties of the type system) *Assuming a context* Γ *such* Γ wf*, an expression* e*, and a type* κ *such* $\Gamma \vdash e : \kappa$*, then:*

- $\Gamma \vdash \kappa$ wf. (κ is well-formed).
- *for all type* κ' *such* $\Gamma \vdash e : \kappa'$ *then* $\kappa = \kappa'$. (κ is unique).

Proposition 4.6.2 (Typechecking algorithm) *There exists an algorithm, parametrized by a well-formed context* Γ *and an expression* e*, which terminates and indicates either that there is no* κ *such* $\Gamma \vdash e : \kappa$*, or which returns* κ *such* $\Gamma \vdash e : \kappa$*.*

4.6.1.4 Examples

Example 4.6.1 Assuming a (well-formed) context Γ such $x : \mathbf{ref}(int) \in \Gamma$, then

$$\Gamma \vdash x :=!x + 1 : (\mathtt{unit}, \{x\})$$

Let me consider the expression

$$ex1 \overset{def}{=} \lambda r : \mathbf{ref}(int).r :=!r + 1; !r$$

In any (well-formed) context Γ, we have:

$$\Gamma \vdash ex1 : (\forall r : \mathbf{ref}(int).(int, \{r\}), \emptyset)$$

Example 4.6.2 Let me consider the expression

$$ex2 \overset{def}{=} y := (ex1\ x)$$

In any context Γ such $x : \mathbf{ref}(int) \in \Gamma$ and $y : \mathbf{ref}(int) \in \Gamma$, we have:

$$\Gamma \vdash ex2 : (\mathtt{unit}, \{x, y\})$$

Example 4.6.3 (no encapsulation in closure) Let me consider the expression

$$ex3 \overset{def}{=} \mathbf{let}\ x = \mathbf{ref}\ 0\ \mathbf{in}\ \lambda y : \mathtt{unit}.(ex1\ x)$$

There is no (well-formed) context Γ and no κ such $\Gamma \vdash ex3 : \kappa$. Indeed, for any Γ, we have:

$$\Gamma, x : \mathbf{ref}(int) \vdash \lambda y : \mathtt{unit}.(ex1\ x) : (\mathtt{unit} \to (int, \{x\}), \emptyset)$$

But, the **LETREF** rule forbids to return $ex3$ a type $\mathtt{unit} \to (int, \{x\})$, because x would escape its lexical scope. Consequently, in UAML-- (and in UAML-) it is not possible to encapsulate the state of function in a the closure. This is discussed section 5.2.

Example 4.6.4 (high-order expressions) Let me consider the expression

$$ex4 \overset{def}{=} \lambda r_2 : \mathbf{ref}(int).\lambda f : (int \to (int, \{r_2\})).r_1 := (f\ !r_1)$$

We have

$$r_1 : \mathbf{ref}(int) \vdash ex4 : (\forall r_2 : \mathbf{ref}(int).\forall f : (int \to (int, \{r_2\})).(\mathtt{unit}, \{r_1, r_2\}), \emptyset)$$

Hence, we have:

$$r_1 : \mathbf{ref}(int); r_2 : \mathbf{ref}(int) \vdash (ex4\ r_2\ (\lambda x : int.r_2 :=!r_2 + 1; x + 1)) : (\mathtt{unit}, \{r_1, r_2\})$$

Example 4.6.5 (no alias via parameter passing) Let me consider the expression

$$ex5 \stackrel{def}{=} (ex4\ r_1\ (\lambda x : int.r_1 :=!r_1 + 1; x + 1))$$

In the context $r_1 : \mathbf{ref}(int)$, $ex5$ is not typable. Indeed, the rule **APPREF** forbids the application of $ex4$ to r_1.

Example 4.6.6 (absence of effect subtyping) Let me consider the expression

$$ex6 \stackrel{def}{=} (ex4\ r_2\ \lambda x : int.x)$$

In the context $r_1 : \mathbf{ref}(int); r_2 : \mathbf{ref}(int)$, $ex6$ is not typable. To express this kind of application, we are obliged to write $(\lambda x : int.\ r_2 :=!r_2\ ;\ x)$ instead of the identity function. Hence,

$$r_1 : \mathbf{ref}(int); r_2 : \mathbf{ref}(int) \vdash (ex4\ r_2\ (\lambda x : int.\ r_2 :=!r_2\ ;\ x)) : (int, \{r_1, r_2\})$$

This limitation of UAML-- motivate the introduction of UAML-

4.6.2 UAML-: UAML-- with more high-order imperative functions

4.6.2.1 Motivations

The example $ex6$ shows a first default of UAML-- type system: functions as effective parameters must effectively use all references that are declared in the type of the formal parameter. The type discipline can be made more flexible using a subtyping relation generated from the inclusion between effects. For instance, we would have:

$$\forall x : int.(int, \emptyset) :> \forall x : int.(int, \{r_2\})$$

Using such a subtyping relation, we will now have:

$$r_1 : \mathbf{ref}(int); r_2 : \mathbf{ref}(int) \vdash (ex4\ r_2\ \lambda x : int.x) : (int, \{r_1, r_2\})$$

Furthermore, a high-order functions like $ex4$ must declare exactly the number and the type of the references that its functional argument f is allowed to use. We would like to have a more expressive type system, in which a high-order function like $ex4$ could take as argument the state on which its argument works. For instance, $ex4$ could be generalize in

$$ex7 \stackrel{def}{=} \lambda r_2 : *.\lambda f : (int \to (int, \{r_2\})).r_1 := (f\ !r_1)$$

such $r_1 : \mathbf{ref}(int) \vdash ex7 : (\forall r_2 : *.\forall f : (int \to (int, \{r_2\})).(\mathtt{unit}, \{r_1, r_2\}), \emptyset)$
The type "$*$" indicates that r_2 is an effect. Typing rules will ensure that such a variable can appear only in types or in effects. In the expression language, r_2 can be instantiated by an effect. For instance, we will have:

$$\begin{array}{l} r_1 : \mathbf{ref}(int); r_2 : \mathbf{ref}(int); r_3 : \mathbf{ref}(int) \\ \quad \vdash (ex7\ \{r_2, r_3\}\ (\lambda x : int.r_2 :=!r_3 + x\ ;\ !r_2)) : (\mathtt{unit}, \{r_1, r_2, r_3\}) \end{array}$$

Of course, typing rules of application restricts its usage in order to avoid aliases: the effect passed as effective parameter must have an empty intersection with the free variables of the result.

4.6.2.2 Types of UAML-

In the abstract syntax of types, the only modification with respect to UAML-- consists in adding a new type of values "$*$" :

$$\tau \stackrel{def}{=} \ldots \mid *$$

The definition of FV and of substitutions are the same, $*$ being a type without variable. I introduce a new kind of substitution which allows to substitute a variable by an effect. This new operation of substitution is needed to express the type of the application of a function of type $\forall x : *.\kappa$ to an effect ϵ.

Definition 4.6.9 (Substitution by an effect) *Let x a variable and ϵ be variables. If ϵ' is an effect, I define the substitution $\epsilon'[x \hookleftarrow \epsilon]$ as:*

$$\begin{aligned} \epsilon'[x \hookleftarrow \epsilon] &\stackrel{def}{=} \epsilon' \text{ if } x \notin \epsilon' \\ \epsilon'[x \hookleftarrow \epsilon] &\stackrel{def}{=} (\epsilon' \backslash \{x\}) \cup \epsilon \text{ if } x \in \epsilon' \end{aligned}$$

I now define the substitutions $\tau[x \hookleftarrow \epsilon]$ and $\kappa[x \hookleftarrow \epsilon]$ by mutual induction on τ and κ.

$$\begin{aligned} \beta[x \hookleftarrow \epsilon] &\stackrel{def}{=} \beta \\ *[x \hookleftarrow \epsilon] &\stackrel{def}{=} * \\ \mathbf{ref}(\tau)[x \hookleftarrow \epsilon] &\stackrel{def}{=} \mathbf{ref}(\tau[x \hookleftarrow \epsilon]) \\ (\forall z : \tau.\kappa)[x \hookleftarrow \epsilon] &\stackrel{def}{=} \forall z : \tau[x \hookleftarrow \epsilon].\kappa[x \hookleftarrow \epsilon] \text{ if } z \notin \{x\} \cup \epsilon \\ (\tau, \epsilon)[x \hookleftarrow \epsilon] &\stackrel{def}{=} (\tau[x \hookleftarrow \epsilon], \epsilon[x \hookleftarrow \epsilon]) \end{aligned}$$

Actually, $\kappa[x \leftarrow y] = \kappa[x \hookleftarrow \{y\}]$.

Definition of typing contexts is unchanged. But, the definition of $Refs(\Gamma)$ is now:

$$Refs(\Gamma) \stackrel{def}{=} \{x \mid x : * \in \Gamma \vee \exists \tau.x : \mathbf{ref}(\tau) \in \Gamma\}$$

As I do not want function to return values of type "$*$", I do not consider "$*$" as a well-formed type. Thus, well-formness satisfies now the following properties:

- if $\Gamma \vdash \mathbf{ref}(\tau)$ wf then τ is not a reference type and $\tau \neq *$.
- if Γ wf and $x : \tau \in \Gamma$ then $\tau = *$ or $\Gamma \vdash \tau$ wf.

Definition 4.6.10 (Well-formness) *Judgments "$\Gamma \vdash \tau$ wf", "$\Gamma \vdash \kappa$ wf" and "Γ wf" are defined by adding the two following rules to those of* UAML--*:*

$$\frac{\Gamma; x : * \vdash \kappa \text{ wf} \quad x \notin Refs(\Gamma)}{\Gamma \vdash \forall x : *.\kappa \text{ wf}} \qquad \frac{\Gamma \text{ wf} \quad x \notin Refs(\Gamma)}{\Gamma; x : * \text{ wf}}$$

Other rules are unchanged.

I introduce now the subtyping relation. Intuitively, this subtyping relation says that we can still pretend that an expression e uses a bigger state that the one found by the type system. Indeed, if x is defined reference in the context, the expression "$x := !x; e$" is observationally equivalent to e.

Definition 4.6.11 (Subtyping relation) *The subtyping relations "$\tau_1 :> \tau_2$" and "$\kappa_1 :> \kappa_2$" are mutually defined by the following rules:*

$$\frac{\tau_1 :> \tau_2 \quad \epsilon_1 \subseteq \epsilon_2}{(\tau_1, \epsilon_1) :> (\tau_2, \epsilon_2)}\textbf{SUBK}$$

$$\frac{}{\tau :> \tau}\textbf{SUBR} \qquad \frac{\tau_2 :> \tau_1 \quad \kappa_1 :> \kappa_2}{\forall x : \tau_1.\kappa_1 :> \forall x : \tau_2.\kappa_2}\textbf{SUBF}$$

Except SUBK rule which is characteristic of our type system, others rules are very common. As usual, for functional types, subtyping is contravariant on the parameter, and covariant on the result. For references, subtyping is equality (because dereference is covariant and affectation contravariant).

Definition 4.6.12 (Union and Intersection of types) *Given three types of values τ_1, τ_2, τ_3 and three types of computations κ_1, κ_2, κ_3, we define mutually recursively the four relations:*

- $\tau_1 \sqcup \tau_2 \Rightarrow \tau_3$ *indicating that τ_3 is the least supertype of τ_1 and τ_2.*
- $\tau_1 \sqcap \tau_2 \Rightarrow \tau_3$ *indicating that τ_3 is the biggest subtype of τ_1 and τ_2.*
- $\kappa_1 \sqcup \kappa_2 \Rightarrow \kappa_3$ *indicating that κ_3 is the least supertype of κ_1 and κ_2.*
- $\kappa_1 \sqcap \kappa_2 \Rightarrow \kappa_3$ *indicating that κ_3 is the biggest subtype of κ_1 and κ_2.*

These relations are defined by the following rules:

$$\frac{}{\tau \sqcup \tau \Rightarrow \tau} \qquad \frac{\tau_1 \sqcap \tau_2 \Rightarrow \tau_3 \quad \kappa_1 \sqcup \kappa_2 \Rightarrow \tau_3}{\forall x : \tau_1.\kappa_1 \sqcup \forall x : \tau_2.\kappa_2 \Rightarrow \forall x : \tau_3.\kappa_3} \qquad \frac{\tau_1 \sqcup \tau_2 \Rightarrow \tau_3}{(\tau_1, \epsilon_1) \sqcup (\tau_2, \epsilon_2) \Rightarrow (\tau_3, \epsilon_1 \cup \epsilon_2)}$$

$$\frac{}{\tau \sqcap \tau \Rightarrow \tau} \qquad \frac{\tau_1 \sqcup \tau_2 \Rightarrow \tau_3 \quad \kappa_1 \sqcap \kappa_2 \Rightarrow \kappa_3}{\forall x : \tau_1.\kappa_1 \sqcap \forall x : \tau_2.\kappa_2 \Rightarrow \forall x : \tau_3.\kappa_3} \qquad \frac{\tau_1 \sqcap \tau_2 \Rightarrow \tau_3}{(\tau_1, \epsilon_1) \sqcap (\tau_2, \epsilon_2) \Rightarrow (\tau_3, \epsilon_1 \cap \epsilon_2)}$$

We show that the announced properties of least supertype and biggest subtype hold.

Proposition 4.6.3 *Given three types of values τ_1, τ_2, τ_3 and three types of computations κ_1, κ_2, κ_3, the following properties holds:*

- *If $\tau_1 \sqcup \tau_2 \Rightarrow \tau_3$, then $\tau_1 :> \tau_3$ and $\tau_2 :> \tau_3$.*
- *If $\tau_1 \sqcap \tau_2 \Rightarrow \tau_3$, then $\tau_3 :> \tau_1$ and $\tau_3 :> \tau_2$.*
- *If $\kappa_1 \sqcup \kappa_2 \Rightarrow \kappa_3$, then $\kappa_1 :> \kappa_3$ and $\kappa_2 :> \kappa_3$.*
- *If $\kappa_1 \sqcap \kappa_2 \Rightarrow \kappa_3$, then $\kappa_3 :> \kappa_1$ and $\kappa_3 :> \kappa_2$.*

Proof.
The proof is done by mutual induction over the derivations.

□

Proposition 4.6.4 *Given two types of values τ_1, τ_2 such $\tau_1 :> \tau_2$ and two types of computations κ_1, κ_2 such $\kappa_1 :> \kappa_2$, the following properties holds*

- $\tau_1 \sqcup \tau_2 \Rightarrow \tau_1$.
- $\tau_1 \sqcap \tau_2 \Rightarrow \tau_2$.
- $\kappa_1 \sqcup \kappa_2 \Rightarrow \kappa_1$.
- $\kappa_1 \sqcap \kappa_2 \Rightarrow \kappa_2$.

Proof.
The proof is done by mutual induction over $\tau_1 :> \tau_2$ and $\kappa_1 :> \kappa_2$.

□

Proposition 4.6.5 *Given three types of values τ_1, τ_2, τ and three types of computations κ_1, κ_2, κ, the following properties holds:*

- *If $\tau_1 :> \tau$ and $\tau_2 :> \tau$ then there exists τ_3 such $\tau_1 \sqcup \tau_2 \Rightarrow \tau_3$ and $\tau_3 :> \tau$.*
- *If $\tau :> \tau_1$ and $\tau :> \tau_2$ then there exists τ_3 such $\tau_1 \sqcap \tau_2 \Rightarrow \tau_3$ and $\tau :> \tau_3$.*
- *If $\kappa_1 :> \kappa$ and $\kappa_2 :> \kappa$ then there exists κ_3 such $\kappa_1 \sqcup \kappa_2 \Rightarrow \kappa_3$ and $\kappa_3 :> \kappa$.*
- *If $\kappa :> \kappa_1$ and $\kappa :> \kappa_2$ then there exists κ_3 such $\kappa_1 \sqcap \kappa_2 \Rightarrow \kappa_3$ and $\kappa :> \kappa_3$.*

Proof.
The proof is done by induction over the first hypotheses. The **SUBR** case corresponds to the preceding proposition. Others are directly deduced from induction hypotheses.

□

4.6.2.3 New typing rules

The abstract syntax of expressions is enriched with the application of an expression to an effect:

$$e ::= ... \mid (e\ \epsilon)$$

The typing rules are now given figure 4.2. Only rules with name postfixed in "+" differs from those of UAML--. Rule **APPEFF** and **FUNEFF** are new.

$$\frac{}{\Gamma \vdash c : (Type(c), \emptyset)}\textbf{CONST} \qquad \frac{x : \tau \in \Gamma \quad \tau \neq \textbf{ref}(\_) \quad \tau \neq *}{\Gamma \vdash x : (\tau, \emptyset)}\textbf{VAR+}$$

$$\frac{x : \textbf{ref}(\tau_1) \in \Gamma \quad \Gamma \vdash e : (\tau_2, \epsilon) \quad \tau_2 :> \tau_1}{\Gamma \vdash x := e : (\texttt{unit}, \epsilon \cup \{x\})}\textbf{SET+} \qquad \frac{x : \textbf{ref}(\tau) \in \Gamma}{\Gamma \vdash !x : (\tau, \{x\})}\textbf{GET}$$

$$\frac{\Gamma \vdash e : (\tau, \epsilon) \quad \tau \neq \textbf{ref}(\_)}{\Gamma \vdash \textbf{ref}\ e : (\textbf{ref}(\tau), \epsilon)}\textbf{NEW}$$

$$\frac{\Gamma \vdash \tau \text{ wf} \quad \Gamma; x : \tau \vdash e : \kappa \quad x \notin Refs(\Gamma)}{\Gamma \vdash \lambda x : \tau.e \ : \ (\forall x : \tau.\kappa, \emptyset)}\textbf{FUN}$$

$$\frac{\Gamma \vdash e_1 : (\forall x : \tau_3.(\tau_1, \epsilon_1), \epsilon) \quad \Gamma \vdash e_2 : (\tau_2, \epsilon_2) \quad \tau_2 \neq \textbf{ref}(\_) \quad \tau_2 :> \tau_3}{\Gamma \vdash (e_1\ e_2) : (\tau_1, \epsilon \cup \epsilon_1 \cup \epsilon_2)}\textbf{APP+}$$

$$\frac{\Gamma \vdash e_1 : (\forall x : \textbf{ref}(\tau_2).(\tau_1, \epsilon_1), \epsilon) \quad y : \textbf{ref}(\tau_2) \in \Gamma \quad y \notin FV((\tau_1, \epsilon_1)) \backslash \{x\}}{\Gamma \vdash (e_1\ y) : (\tau_1[x \leftarrow y], \epsilon \cup \epsilon_1[x \leftarrow y])}\textbf{APPREF}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \epsilon_1) \quad \tau_1 \neq \textbf{ref}(\_) \quad \Gamma; x : \tau_1 \vdash e_2 : (\tau_2, \epsilon_2) \quad x \notin Refs(\Gamma)}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : (\tau_2, \epsilon_1 \cup \epsilon_2)}\textbf{LET}$$

$$\frac{\Gamma \vdash e_1 : (\textbf{ref}(\tau_1), \epsilon_1) \quad \Gamma; x : \textbf{ref}(\tau_1) \vdash e_2 : (\tau_2, \epsilon_2) \quad x \notin Refs(\Gamma) \cup FV(\tau_2)}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : (\tau_2, \epsilon_1 \cup (\epsilon_2 \backslash \{x\}))}\textbf{LETREF}$$

$$\frac{\Gamma \vdash e_1 : (\texttt{bool}, \epsilon_1) \quad \Gamma \vdash e_2 : (\tau_2, \epsilon_2) \quad \Gamma \vdash e_3 : (\tau_3, \epsilon_3) \quad \tau_2 \cup \tau_3 \Rightarrow \tau_1}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : (\tau_1, \epsilon_1 \cup \epsilon_2 \cup \epsilon_3)}\textbf{COND+}$$

$$\frac{\Gamma; x : * \vdash e : \kappa \quad x \notin Refs(\Gamma)}{\Gamma \vdash \lambda x : *.e \ : \ \forall x : *.\kappa}\textbf{FUNEFF}$$

$$\frac{\Gamma \vdash e_1 : (\forall x : *.(\tau_1, \epsilon_1), \epsilon_2) \quad \epsilon \subseteq Refs(\Gamma) \backslash (FV((\tau_1, \epsilon_1)) \backslash \{x\})}{\Gamma \vdash (e_1\ \epsilon) : (\tau_1[x \hookleftarrow \epsilon], \epsilon_2 \cup \epsilon_1[x \hookleftarrow \epsilon])}\textbf{APPEFF}$$

Figure 4.2: Type system of UAML-

I comment here these rules:

VAR+ Now, this rule also forbids to return state variable.

SET+, APP+ Subtyping hypotheses are added.

COND+ Infer the least supertype of the two branches.

FUNEFF It is need because **FUN** admit only well-formed types as parameter type. And $*$ is not well-formed. But the rule is similar to **FUN**.

APPEFF This rule adapts the rule **APPREF** for state variable. Elements of the effective effect must be in reference variable or state variable of the context which do not occur free in the computation returned by the function.

Many properties of UAML-- type system are preserved. In particular, properties 4.6.1 and 4.6.2 (page 116) still hold for UAML-. Furthermore, the subtyping relation opens the perspective to prove a subject-reduction property with respect to a small-steps semantics.

4.6.3 Interpretation of UAML- in Coq

[...] This interpretation is quite close to the Filliâtre's one. There are mainly two differences:

- the Coq term produced by Filliâtre's interpretation does not contain constants to represent monads construction (in other words, such constant are directly "unfolded" into their purely functional encoding during the translation).
- In Filliâtre's interpretation, successive values of a variable of reference are denoted by a renaming of this variable in the translation. Hence, Filliâtre interprets states as tuples only in a few places (mainly, in results of functions).

Thus, the interpretation is quite optimized: there are not too many operations on tuples. In counterpart, the syntactic structure of the source has (almost) disappear, and the produced function is hard to understand by "normal" users.

Interpretation of states and variables In implementations, states may be interpreted as tuple. However, in order to simplify this presentation, I use a record notation to represent this records. Hence, a state related to a variable x in an expression, will be represented as a field of same name in a record. I assume the following operations on records[**A FAIRE**].

I must translate variables of UAML- into Coq. I do not detail here how this translation is performed, and the translation of a variable x is simply denoted here by $\bar{x}$.

I now introduce the interpretation of UAML- types into Coq types. Dependent types of UAML-- are translated into non-dependent types. Quantification over "$*$" is interpreted as polymorphism.

Definition 4.6.13 (Interpretation of UAML- types) *We define the interpretation of well-formed UAML- types by mutual induction on the well-formness judgments.*

$$
\begin{array}{ll}
\text{base type} & [\beta]_\Gamma \stackrel{def}{=} \beta \\
\text{reference type} & [\mathbf{ref}(\tau)]_\Gamma \stackrel{def}{=} [\tau]_\Gamma \\
\text{effects type} & [\,*\,]_\Gamma \stackrel{def}{=} \textsf{Type} \\
\text{effects} & [\{x_1 \ldots x_n\}]_\Gamma \stackrel{def}{=} \{x_1 : \tau'_1; \ldots; x_n : \tau'_n\} \\
 & \text{where } \left\{ \begin{array}{l} \tau'_i = [\tau_i]_\Gamma \text{ if } x_i : \mathbf{ref}(\tau_i) \in \Gamma \\ \tau'_i = \bar{x}_i \text{ if } x_i : * \in \Gamma \end{array} \right. \\
\text{computations} & [(\tau, \epsilon)]_\Gamma \stackrel{def}{=} (\texttt{DS}\ [\epsilon]_\Gamma\ [\tau]_\Gamma). \\
\text{functional type (ref)} & [\forall (x : \mathbf{ref}(\tau)).\kappa]_\Gamma \stackrel{def}{=} [\kappa]_{\Gamma; x:\mathbf{ref}(\tau)}. \\
\text{functional type (non ref)} & [\forall (x : \tau).\kappa]_\Gamma \stackrel{def}{=} \textsf{forall}\ \bar{x} : [\tau]_\Gamma\textsf{,}\ [\kappa]_{\Gamma; x:\tau} \text{ if } \tau \neq \mathbf{ref}(\_).
\end{array}
$$

Definition 4.6.14 (Interpretation of typing contexts) *Interpretation of contexts* $[\Gamma]$ *is defined by induction on* Γ*:*

$$
\begin{array}{ll}
[\emptyset] \stackrel{def}{=} \emptyset & \\
[\Gamma; x : \mathbf{ref}(\tau)] \stackrel{def}{=} [Gamma] & \\
[\Gamma; x : \tau] \stackrel{def}{=} [\Gamma]; \bar{x} : [\tau] & \textit{if } \tau \neq \mathbf{ref}(\_).
\end{array}
$$

I now define an operation (called conversion) to represent renaming of variable on translated expressions and values in Coq.[**A detailler**]. This operation performs a permutation of the field in the tuple. Given a value type τ, and two variables x and y, and Coq expression V of type $[\tau]$, I define a conversion operation[1] $\mathbb{C}_{\tau,x,y}(V)$ returning a Coq expression of type $[\tau[x \leftarrow y]]$. In the same way, given κ and E of type $[\kappa]$, $\mathbb{C}_{\kappa,x,y}(E)$ returning a Coq expression of type $[\kappa[x \leftarrow y]]$.

Definition 4.6.15 (Conversion (interpretation of renaming)) *$\mathbb{C}_{\tau,x,y}(V)$ and $\mathbb{C}_{\kappa,x,y}(E)$ are defined by mutual induction on τ and κ.*

$$\mathbb{C}_{(\tau_1,\epsilon_1),x,y}(E) \stackrel{def}{=} \texttt{bind}\ \mathbb{P}_{\epsilon_1,x,y}\ E\ (\textbf{fun}\ x\ \Rightarrow (\texttt{val}\ \mathbb{C}_{\tau_1:>\tau_2}(x)))$$
$$\mathbb{C}_{\beta,x,y}(V) \stackrel{def}{=} V$$
$$\mathbb{C}_{\forall z:\mathbf{ref}(\tau).\kappa,x,y}(V) \stackrel{def}{=} \mathbb{C}_{\kappa,x,y}(V)$$
$$\mathbb{C}_{\forall z:\tau.\kappa,x,y}(V) \stackrel{def}{=} \textbf{fun}\ \bar{z} : [\tau]\ \Rightarrow \mathbb{C}_{\kappa,x,y}(V\ \mathbb{C}_{\tau,x,y}(\bar{z}))\ \textit{if}\ \tau \neq \mathbf{ref}(\_)$$

[Meme a faire chose avec $\mathbb{C}_{\kappa,x,\epsilon}$]

Given two value types τ_1, τ_2 such $\tau_1 :> \tau_2$ and a Coq expression V of type $[\tau_1]$, I define a conversion operation[2] $\mathbb{C}_{\tau_1:>\tau_2}(V)$ returning a Coq expression of type $[\tau_2]$. In the same way, given $\kappa_1 :> \kappa_2$ and E of type $[\kappa_1]$, $\mathbb{C}_{\kappa_1:>\kappa_2}(E)$ returning a Coq expression of type $[\kappa_2]$.

Definition 4.6.16 (Conversion (interpretation of subtyping)) *$\mathbb{C}_{\tau_1:>\tau_2}(V)$ and $\mathbb{C}_{\kappa_1:>\kappa_2}(E)$ are defined by mutual induction on the subtyping judgment.*

SUBK

$$\mathbb{C}_{(\tau_1,\epsilon_1):>(\tau_2,\epsilon_2)}(E) \stackrel{def}{=} \texttt{bind}\ \mathbb{P}_{\epsilon_1\subseteq\epsilon_2}\ E\ (\textbf{fun}\ x\ \Rightarrow (\texttt{val}\ \mathbb{C}_{\tau_1:>\tau_2}(x)))$$

SUBR

$$\mathbb{C}_{\tau:>\tau}(V) \stackrel{def}{=} V$$

SUBF

- *if* $\tau_1 \neq \mathbf{ref}(\_)$

$$\mathbb{C}_{\forall x:\tau_1.\kappa_1:>\forall x:\tau_2.\kappa_2}(V) \stackrel{def}{=} \textbf{fun}\ \bar{x} : [\tau_2]\ \Rightarrow \mathbb{C}_{\kappa_1:>\kappa_2}(V\ \mathbb{C}_{\tau_2:>\tau_1}(\bar{x}))$$

-

$$\mathbb{C}_{\forall x:\mathbf{ref}(\tau_0).\kappa_1:>\forall x:\mathbf{ref}(\tau_0).\kappa_2}(V) \stackrel{def}{=} \mathbb{C}_{\kappa_1:>\kappa_2}(V)$$

Definition 4.6.17 *Let Γ a well-formed Coq context. Let e a specification such $\Gamma \vdash e : (\tau, \epsilon)$. Given ϵ' such $\epsilon \subseteq \epsilon' \subseteq Refs(\Gamma)$, I now define the Coq term $[e]_\Gamma^{\epsilon'}$ of type $[(\tau,\epsilon')]_\Gamma$ by mutual induction on the typing judgment of e:*

CONST

$$[c]^{\epsilon'} \stackrel{def}{=} \texttt{val}\ \bar{c}$$

VAR+

$$[x]^{\epsilon'} \stackrel{def}{=} \texttt{val}\ \bar{x}$$

SET+

$$[x := e]^{\epsilon'} \stackrel{def}{=} \texttt{bind allPl}\ [e]^{\epsilon'}\ (\textbf{fun}\ \bar{x} \Rightarrow \texttt{set}\ \mathbb{P}_{\{x\}\subseteq\epsilon'}\ \bar{x})$$

GET

$$[!x]^{\epsilon'} \stackrel{def}{=} \texttt{get}\ \mathbb{P}_{\{x\}\subseteq\epsilon'}$$

NEW

$$[\mathbf{ref}\ e]^{\epsilon'} \stackrel{def}{=} [e]^{\epsilon'}$$

FUN We get $e \equiv \lambda x : \tau_0.e_1$ with $\Gamma \vdash \lambda x : \tau_0.e_1\ :\ (\forall x : \tau_0.(\tau_1, \epsilon_1), \emptyset)$. I distinguish two cases:

[1] This operation is actually defined in the meta-language.
[2] This operation is actually defined in the meta-language.

- *In the case* $\tau_0 \neq \mathbf{ref}(\_)$, *I set:*

$$[\lambda x : \tau_0.e_1]^{\epsilon'} \stackrel{def}{=} \mathtt{val}\ (\textsf{fun}\ \bar{x} : [\tau_0] \Rightarrow [e_1]^{\epsilon_1})$$

- *In the case* $\tau_0 = \mathbf{ref}(\tau_0')$, *I set:*

$$[\lambda x : \tau_0.e_1]^{\epsilon'} \stackrel{def}{=} \mathtt{val}\ ([e_1]^{\epsilon_1})$$

LET

$$[\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]^{\epsilon'} \stackrel{def}{=} \mathtt{bind}\ \mathtt{allPl}\ [e_1]^{\epsilon'}\ (\textsf{fun}\ \bar{x} \Rightarrow [e_2]^{\epsilon'})$$

LETREF

$$\begin{array}{l}[\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]^{\epsilon'} \stackrel{def}{=} \\ \quad \mathtt{bind}\ \mathtt{allPl}\ [e_1]^{\epsilon'}\ (\textsf{fun}\ \bar{x} \Rightarrow \mathtt{block}\ \mathbb{S}_{\epsilon',\{x\}}\ \bar{x}\ [e_2]^{\epsilon' \cup \{x\}})\end{array}$$

APP+

$$\begin{array}{l}[(e_1\ e_2)]^{\epsilon'} \stackrel{def}{=} \\ \quad \mathtt{bind}\ \mathtt{allPl}\ [e_2]^{\epsilon'} \\ \quad\quad (\textsf{fun}\ x_2 \Rightarrow \mathtt{bind}\ \mathtt{allPl}\ [e_1]^{\epsilon'}\ (\textsf{fun}\ x_1 \Rightarrow \mathbb{C}_{\forall x:\tau_3.(\tau_1,\epsilon_1):>\forall x:\tau_2.(\tau_1,\epsilon')}(x_1)\ x_2))\end{array}$$

APPREF

$$[(e_1\ y)]^{\epsilon'} \stackrel{def}{=} \mathtt{bind}\ \mathtt{allPl}\ [e_1]^{\epsilon'}\ (\textsf{fun}\ x_1 \Rightarrow \mathtt{val}\ \mathbb{C}_{(\tau_1,\epsilon_1),x,y}(x_1))$$

COND+

$$[\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]^{\epsilon'} \stackrel{def}{=} \mathtt{If}\ [e_1]^{\epsilon'}\ \mathtt{Then}\ \mathbb{C}_{(\tau_2,\epsilon'):>(\tau_1,\epsilon')}([e_2]^{\epsilon'})\ \mathtt{Else}\ \mathbb{C}_{(\tau_3,\epsilon'):>(\tau_1,\epsilon')}([e_3]^{\epsilon'})$$

FUNEFF *We get* $e \equiv \lambda x : *.e_1$ *with* $\Gamma \vdash \lambda x : *.e_1\ :\ (\forall x : *.(\tau_1, \epsilon_1), \emptyset)$.

$$[\lambda x : *.e_1]^{\epsilon'} \stackrel{def}{=} \mathtt{val}\ (\textsf{fun}\ \bar{x} : \textsf{Type} \Rightarrow [e_1]^{\epsilon_1})$$

APPEFF

$$[(e_1\ \epsilon_0)]^{\epsilon'} \stackrel{def}{=} \mathtt{bind}\ [e_1]^{\epsilon'}\ (\textsf{fun}\ x_1 \Rightarrow \mathbb{C}_{\kappa_1[x \hookleftarrow \epsilon_0],(\tau_1[x \hookleftarrow \epsilon_0],\epsilon')}(x_1\ [\epsilon_0]))$$

The correction of the interpretation of UAML-- *in monads has been proved by Filliâtre in [Fil99b, Fil99a]. I have not yet adapted his proof to my interpretation.*

Chapter 5

Perspectives of application

In this chapter, I study several way to continue this work. In section 5.1, I sketch what kind of case-studies will be interesting, when the prover introduced chapter 4 will be achieved. In section 5.2, I present ideas to extend this work with object-oriented features. Examples of this chapter are given using a pseudo-syntax extending the language UAML- of chapter 4, with DSL constructions and OCAML constructions.

5.1 Application of high-order impure specifications

I present here two examples of using high-order impure specifications. In the first example, high-order functions are used to express algorithms. In the second, they are used to allow code reusing.

5.1.1 Proofs of algorithms

For enumerating trees (or forests), continuation-style is well-adapted. Indeed, using this style, algorithms for enumerating trees can be expressed as simple recursive functions. Using this idea, J.-C. Filliâtre and F. Pottier present in [FP03] a high-order imperative implementation of Koda and Ruskey's algorithm for enumerating all ideals of a finite poset as a Gray code. Let me detail this. Given a partially ordered set $(E, \leq)$, an ideal is a subset I of E such for all element y of I, all predecessors of y (elements x of E such $x \leq y$) are in I. Assuming E is a finite, the algorithm of Koda and Ruskey enumerates all ideals as a Gray code: two successive ideals in the enumeration differ only of one element (present or absent). Such enumeration is optimal in the sense that the information to transmit between two successive item is minimal.

Technically, Koda and Ruskey's algorithm enumerate particular colorings of a finite forest. As E is finite, its structure of partial ordering can be represented as a finite forest. A subset S of E corresponds to the coloring of this forest using two colors on nodes: white when the corresponding element is in S and black when it is not. An ideal is thus a coloring, such that all descendants of a white node are white.

Using a continuation-based approach, Filliâtre and Pottier give an extremely concise OCAML implementation of the algorithm's core. Then, in a number of steps, they refine it with a first-order version. The efficiency of this last implementation is comparable to a C implementation given by Knuth. It seems an interesting challenge to formalize their complete proof in the theory of high-order refinement.

5.1.2 Property-based specifications versus model-based specifications

I distinguish here two style of specifications: property-based specifications and model-based specifications. The first ones, typically algebraic specifications, enumerate the properties that the software must fulfill. The second ones, typically B specifications [Abr96] or JML specifications [LBR04], describe a model, and software must have the same observational behavior than the model.

Generally, model-based specifications make proofs simpler whereas property-based specifications are more reusable. Indeed, having a model (partially abstract) allows to perform computations on this model. But, the properties satisfied by the software are encoded into this model and can not a priori be reused for a different model.

Let me give a small example. This example will also demonstrate that high-order refinement enriches first-order refinement by allowing more property-based specifications than first-order refinement. The aim of this example is to specify a function `f: nat -> unit` such that `(f n)` prints the `n`-first naturals satisfying a given property (for instance, the `n`-first prime number). To describe abstractly this problem, we may say that `(f n)` must print a list of natural `l` such a given property `(P n l)` is satisfied. The point now is to specify what is "to print a natural".

In a model-based approach, we use a model variable `screen`: a list representing the successive element sent to the screen. This model variable is private to function `print_nat`. Hence, `f` can not modify it directly:

```
screen: ref(list nat)

let print_nat x =
  screen := !screen @ [x]   (* x is inserted in the queue of screen *)
```

We can now specify that `(f n)` must refine the following specification:

```
let l = any(list nat) in
  ensure (P n l) ;
  screen := !screen @ l
```

In a property-based approach, instead of giving a model of what does `print_nat`, we express directly how `(f n)` must use `print_nat`. At the level of `f`, we only see `print_nat` as a parameter. Hence, instead of considering `f`, we consider `f'` the function abstracting `print_nat` in `f`, such `(f n)=(f' print_nat n)`. Using the type-system of proposed in chapter 4, we specify that `print_nat` works on a private state (hence in this type-system, `f'` is also parametrized by this private state). Then, we specify that `(f' print_nat n)` must refine:

```
let l = any(list nat) in
  ensure (P n l) ;
  List.iter print_nat l
```

Maybe that for some cases the proof of this refinement requires more informations on `print_nat`: `print_nat` do not raise an exception or is never aborting (its precondition is true). These properties can be expressed as preconditions of the specification. But, it seems to me that on most examples, such properties are not needed.

On this particular example, proofs in the property-based approach does not seem to be more difficult than in the model-based approach. In the property-based approach, we have however a more general view on `f` (or `f'`) because it is a enumerating function, and not only a printing function.

5.2 Object-oriented methodology

Object-oriented languages provide examples of widely used applications that combine high-order style and imperative style. In this section, I consider a weak form of object-orientation: the combination of state encapsulation and dynamic binding (high-order functions). In particular, I neglect usual feature of OO-languages such aliasing and open-recursion which are non-trivial to deal with.

Hence, in this section, I first explain what OO-encapsulation could bring to DSL languages. Then, conversely, I show that high-order refinement allows to specify some utilizations of callbacks (methods as value). As notice in [BW99], specifications of patterns using callbacks involve grey-box specifications. They can not be correctly expressed using standard Hoare logics. At the end, I develop these ideas on a more complex example.

5.2.1 Interest of "OO-extensions" in high-order refinement

In the UAML- type-system presented chapter 4, or even in its denotational semantics formalized by `ISL`, there is no notion of state-encapsulation as in ML.

5.2.1.1 The need of state-encapsulation

In ML, state encapsulation is performed using closure. For instance, we can define a function `incr` counting how many times it is called, and which prints the value of this counter:

```
let incr =
  let c = ref 0
  in fun () -> c := !c+1 ; print_nat !c
```

Hence, `incr` of type `unit -> unit` has an internal state which is initialized when it is defined, and which is only accessible by it. In the operational semantics, this state is a global state (allocated in the heap), which lives as long `incr` is living.

In UAML- such a definition is rejected in the **LETREF**-rule, because the `c` in the resulting type escape its scope. If we look how to denote such a function in the Coq interpretation, we see that `c` must be represented as a global variable which lives between two calls to `incr`.

Hence, UAML- must be extended to provide a way to encapsulate some global variable. I advocate here for the use of a OO-like encapsulation. Actually, I know only one other alternative: modules with abstract types. The object-oriented approach seems more expressive since object are first-class citizen whereas modules are not. Except this, the difference between the two approaches seems not very big, since the objects considered here are very weak.

5.2.1.2 A weak form of object-orientation

In the OO-approach I consider, classes are only a way to declare a set of closure sharing some global variables, which are not visible from outside. For instance, the `incr` function above, can be coded in Ocaml class:

```
class counter =
  object (self)
    val mutable c = 0
    method incr =
        c <- c+1 ;
        print_nat c
  end
```

Creating an object is creating the variables and the instantiated methods of the corresponding classes. Actually, with respect to the type-system presented chapter 4, objects would only be a generalization of references: object creation would be a generalization of the **NEW/LETREF** rules, and method calls would generalize the **GET/SET** rules. Indeed, when calling `incr`, the state which this function works with, is explicitly given in the syntax:

```
let o = new counter in
  ...
  o#incr
```

Actually, if we omit the encapsulation aspects, the operational aspects of such a `counter` class can be coded in the type-system of chapter 4:

```
let counter_new () = ref 0

let counter_incr c =
  c := !c+1 ; print_nat !c
```

Then, **let** `o` = **new** `counter` **in** ... is translated into

```
let o_state = counter_new () in
let o_incr = counter_incr o_state in
  ...
```

and, method calls `o#incr` are simply translated into `o_incr`. Passing `o` as parameter is translated in passing `o_state` and `o_incr` as parameters. Hence, in this simplified view of objects, we code operationally objects as a pair formed of:

- a reference on a record of fields.
- a record of methods, each method having the preceding reference as first parameter. A method call is only applying the method on the reference.

Let me remark that a such encoding allows to represent some form of dynamic binding. But, it does not allow open recursion (nor aliasing). This informal example shows that we can extend UAML- in order to have some OO-features.

5.2.1.3 B-machines as classes

After this step, an interesting extension is to embed concepts of B-machines (see [Abr96]) in this language. The idea is to consider B-machines as classes. Objects would be instances of machines. This would extend modularity of B, because currently, in B, there is only one instance by machine. Furthermore, instances of machine are not first-class values. But, B-machines introduce powerful concepts, which may be reused in OO proof-systems.

First, B introduces a notion of strong invariant. Translated in OO-languages, such an invariant is a property about the internal state of an object. It is satisfied all the time, excepted during method-calls on this object where it can be locally violated. This invariant is said *strong*, because syntactic restrictions ensure that it is sufficient to prove the preservation of the invariant *when the class introducing this invariant is defined*. Hence, every user of the corresponding objects can assume that their invariant holds without never proving them. This notion of strong invariant is opposed to weak invariants (of JML for instance), which are only syntactic sugar for pre and post-conditions of methods. Thus, users of an object with a weak-invariant must prove at each method call that the invariant holds, as it is a precondition of the call. But the correction of strong invariants requires a strong encapsulation of object attributes. In particular, they can only be modified using a method of the object that own them.

Second, B introduces a notion of refinement on machines. Translated in OO-languages, this can be considered as the logical part of subtyping. Refinement of machines involve a notion of gluing invariant, which relates the variables of the refined machine and the refining machine. In B, data-refinement is performed with this mechanism. Strong invariants are fundamental to ensure that machine refinements is compositional.

5.2.2 Interest of high-order refinement for OO-patterns

In the following example, I show that the (informal) OO language that I propose allows to implement some (simplified) design-patterns with callbacks.[GHJV95]

5.2.2.1 A very simple instantiation of observer pattern in Ocaml

Observer pattern is widely use in the implementation of interfaces (cf. as in Java API for instance). It aims to centralize the actions associated to events occurring on elements of the interface. Here, I consider a button with a method `press_or_release`, which is called each time the button is pressed or released by the user. Internally, the button as a state `is_pressed` indicating whether it is currently pressed or released.

The idea is to register all actions observing (or listening) the button in the button. Hence, each time the button is pressed or released, it can notify all its observers and communicate them its new state. Thus, observers of the button are parametrized by a boolean corresponding to the internal state of the button. The button registers them in a list called below `obs_l`. Each observer can be registered via the `add` method. The method `notifyAll` allows the button to notify all its current observers.

```
class button =
  object (self)

    val mutable obs_l: (bool -> unit) list = []
    val mutable is_pressed = false

    method notifyAll() =
      List.iter (fun obs -> obs is_pressed) obs_l

    method add new_obs =
      obs_l <- new_obs :: obs_l

    method press_or_release =
      is_pressed <- not (is_pressed) ;
      self#notifyAll()

  end
```

Once a button is created, we can dynamically add it some observer (below a function printing "On" or "Off" whether the button is pressed or not).

```
let b = new button;;

b#add (fun is_p -> print_string (if is_p then "On" else "Off"))
```

We can also use the preceding counter of page 126 to count how many times the button is pressed.

```
let c = new counter in b#add (fun is_p -> if is_p then c#incr)
```

5.2.2.2 Specifying this example

The preceding example may almost be expressed in the OO-language I have sketched informally section 5.2.1.2. The only problem is that the state on which the observers are allowed to work with, must be specified. This state can be however given as a parameter of `button` class. Informally, this give us:

```
class button (st:*) =
  object (self)

    val mutable obs_l: list (bool -> (unit, {st})) = []

    ...

  end
```

But, this parameter must be instantiated at the creation of the button. Hence, if we want to add the two preceding observers, we must create the counter `c` before the button `b`. Supposing that `screen` is the internal state of `print_string` and `print_nat`, we write:

```
let c = new counter
let b = new button { c, screen }
```

Now, we can dynamically add observers, but they must work on a state which is a subset of { `c, screen` }. More formally, their type must be a subtype of `bool -> (unit, { c, screen })`. This limit seriously the interest of the method `add`. Concretely, this means that all kind of observations must be known at the creation of the button. However, the class `button`, which parametrizes buttons by the set of their observers, seems still useful.

Now, what is a good specification of this class? Actually, it is already rather simple. But it seems interesting to let the notification order of observers undetermined. This would ensure that the behavior of the whole application does not depend on this order, and in particular, not on the order in which observers are registered. Thus, to express this, I assume that a logical predicate `List.is_perm` which means that two lists are permutation of each other, is defined. Then, the specification of `button` is the following class (only `notifyAll` is changing with respect to methods of `button`) :

```
class button_spec (st:*) =
  object (self)

    val mutable obs_l: list (bool -> (unit, {st})) = []
    val mutable is_pressed = false

    method notifyAll() =
      let l = any (list (bool -> (unit, {st}))) in
        ensure (List.is_perm l obs_l) ;
        List.iter (fun obs -> obs is_pressed) l

    method add (new_obs:list (bool -> (unit, {st}))) =
      obs_l <- new_obs :: obs_l

    method press_or_release =
      is_pressed <- not (is_pressed) ;
      self#notifyAll()

  end
```

Stating that `button` is an implementation of `button_spec` means that `button` refines `button_spec`.

5.2.3 A pseudo case-study

In this subsection, I propose a more complex example. This example still written in a pseudo-syntax is presented as a motivation for future works. First, I introduce an English specification of the "application". Second, I give some abstract specifications which are only partial with respect to the English specification, but which express some important aspects of the application. Third, I give the total specification of the application. At last, I give an example of implementation in OCAML.

5.2.3.1 English specification

The application to realize is a class managing a banking account which automatically pays external creditors. This class must provide two methods: an operation, called `credit`, allowing to credit the account, and an operation, called `debit`, allowing to transfer money from the account to an external creditor.

Hence, `credit` is simply parametrized by the amount to credit, whereas `debit` is parametrized by the amount to debit and a transfer operation. This operation, also parametrized by an amount, allows to pay the creditor who will receive this money. It is non-aborting except when the amount is zero (the operation may reject zero). The amount can be transfered in several times.

The internal balance of the account must be always non negative: debt can only be contracted with external creditors. Hence, during a debit operation, the transfer operation may be partly delayed. But, the global debt must be minimal: external creditors must be refunded as soon as the internal balance of the account is non zero.

5.2.3.2 Abstract specifications

As the total specification of this application is a bit complex, I formulate in a first step two weaker specifications that the total specification must enforce. The first is a model-based specification, whereas the second is rather property-based.

The first specification, class `account1`, describes the basic purposes of an account. It introduces a variable `global_balance` which represents the global balance of the system. It gives the specification of `debit` and `credit` with respect to this notion of `global_balance`. I also introduces a very abstract description of `debit` and `credit` with respect to transfer operations. The class `account1` is parametrized by `st` the state which transfer operations are allowed to work with. As `debit` and `credit` may call some transfer operation, they can modify this state. In the following, I require that the total specification refines this class.

In the pseudo-syntax below, **modify** is an operator of the language parametrized by an effect which modifies variables of this effect non-deterministically. Hence, if `x` is of type `ref(nat)` then **modify** $\{x\}$ is equivalent to `x:=`**any**`(nat)`. Let me remark, that if `st:*` then `st := ...` is not allowed by the type system (see figure 4.2 page 120).

```
class account1 (st:*) =
  object (self)


    val mutable global_balance: Z = 0
```

```
    method debit (amount:nat) (transfer: nat -> (unit,{st})) =
      require (
         forall (n:nat),
           ((transfer n) refines (require (n>0) ; modify {st}))
      ) ;
      global_balance <- global_balance - (Z_of_nat amount) ;
      modify {st}

    method credit (amount:nat) =
      global_balance <- global_balance + (Z_of_nat amount) ;
      modify {st}

end
```

Now, I express a property, called `refund_guarantee`, which external creditors are interested in: as soon as the global balance becomes non negative, they are ensured to be refunded. This property is parametrized by a state `st` and an other state `others` and an object `o` of type `account1 {st}`. It expresses that for all `amount`, and all `transfer`, in a sequence of a debit operation `o#debit amount transfer`, followed by an arbitrary sequence of credit operation on `o`, interleaved with arbitrary modifications of the global state (`others` included), as soon as the global balance becomes non negative then the sum of the value given to `transfer` is at least equals to `amount`.

In the specification below, expression **forN** `x` **do** `s` **done** repeats successively `x` times the expression `s` (where `x` is natural). Operator **choose** chooses non-deterministically a branch between the two expressions separated by "|".

```
property refund_guarantee (st others:*) (o: account1 {st})
=
 forall (amount: nat) (transfer: nat -> (unit,{st})),
 (forall (n:nat),
   ((transfer n) refines (require (n>0) ; modify {st})))
->
(
  o#debit amount transfer ;
  forN (any nat) do
    ensure o#global_balance < 0 ;
    choose
      o#credit (any nat)
    | modify {o,st,others}
  done ;
  ensure o#global_balance >= 0
)
refines
(
   let sum = ref 0 in
   forN (any nat) do
     choose
       let am = any nat in
         ensure 0 < am ;
         transfer am ;
         sum := !sum + am
```

```
      | modify {o,st,others}
    done ;
    ensure !sum >= amount
)
```

refund_guarantee which expresses a property on the trace of operation calls, looks like specification in event B-systems. Here the observed event is the call of transfer.

5.2.3.3 A total (model-based) specification

I propose now a model of the English specification, with the class account2 below. It introduces:

- the internal balance (variable balance) of the account, that must be non negative (a natural).
- the global external debt which must still be paid to external creditors.
- the list of external creditors, represented as a couple of an amount, the debt which is not yet refunded, and a transfer operation.

Here, invariants (strong, as in B, see section 5.2.1.3) give properties that are satisfied by an account. Invariant debt_definition expresses that debt is the sum of the amount in the creditors list (List.fold_left is the usual OCAML traversal on lists). Invariant transfer_spec expresses that all transfer operation of the creditors list are non-aborting when the transfered amount is non-negative (I assume List.for_all as a predefined predicate). Invariant debt_is_minimal expresses that the balance can not be positive if the debt is negative. To satisfy this constraint, operations debit and credit must transfer money to creditors as soon as it is possible.

This class introduces a private method refund which is called to perform all transfer operations in the class. This method expresses that each time an amount is transfered, the amount is subtracted from debt and balance. Let us remark that st is implicitly disjoint from the internal variables of the account. Hence, a transfer operation can not modify these internal variables.

At the beginning of operation debit, if balance is positive, the debt is zero (there are no creditors to pay). In this case, debit refunds directly (but maybe partly) its new creditor. If balance is zero, debit only insert the new creditor in the list of creditors. Invariant debt_is_minimal is thus preserved.

Operation credit, using function notify, performs a traversal of creditors. The purpose of notify is to share the value of balance between creditors. It transfers a part of this value to each creditors. The order in which elements of creditors are examined is chosen non-deterministically (using function permut). The amount transfered to each creditor is also chosen non-deterministically. Function notify returns the new value of the creditors list. This list is permuted non-deterministically before to be stored in creditors. At the end, credit ensure that debt_is_minimal property is preserved. This **ensure** clause forces the sharing policy between creditors to satisfy this property. Hence, at this level, preservation of debt_is_minimal is trivially proved. Thus refinements of credit which will determine the sharing policy will have to ensure this property. To prove the preservation of debt_definition and transfer_spec requires to show that:

- these two properties are invariant by permutation of lists.
- if these two properties hold for the list in the parameter of notify, they hold for the result of notify. The proofs are done by induction on the parameter.

```
class account2 (st:*) =
  object (self)

    val mutable balance: nat = 0
    val mutable debt: nat = 0
    val mutable creditors: (nat * (nat -> (unit,{st}))) list = []
```

```
  invariant debt_is_minimal:
    balance > 0 -> debt = 0

  invariant debt_definition:
    debt =
     List.fold_left (fun acc (amount,_) -> acc+amount) 0 creditors

  invariant transfer_spec:
     List.for_all (fun (_,transfer) ->
                    forall (n:nat),
                      ((transfer n) refines (require (n>0) ; modify {st})))

  method private refund (amount:nat) (transfer:nat -> (unit,{st})) =
    if amount > 0 then (
       transfer amount ;
       debt <- debt - amount ;
       balance <- balance - amount
    )

  method debit (amount:nat) (transfer: nat -> unit) =
    require (
       forall (n:nat),
         ((transfer n) refines (require (n>0) ; modify {st}))
    ) ;
    debt <- debt + amount ;
    let r = min balance amount in
      self#refund r transfer ;
      creditors <- (amount - r,transfer)::creditors

  method credit =
    let permut l =
       let pl = any ((nat * (nat -> (unit,{st}))) list) in
         ensure (List.is_perm l pl) ;
         pl
    and let rec notify all =
      match all with
      | [] -> []
      | (amount,transfer)::others ->
           let r = any (nat) in
             ensure r <= min balance amount ;
             self#refund r transfer ;
             (amount - r, transfer)::(notify others)
    in fun (amount:nat) ->
      balance <- balance + amount ;
      creditors <- permut (notify (permut creditors)) ;
      ensure (balance > 0 -> debt = 0)

end
```

Now, I advocate that this specification is "correct" with respect to the English specification. In particular, I relate the class `account2` to the abstract specifications above. Class `account2` refines `account1` under the gluing invariant: `global_balance = (Z_of_nat balance) - (Z_of_nat debt)`.

For `debit` the proof is trivial (but use invariant `transfer_spec`). For `credit`, it requires an induction over list (and invariants `debt_definition` and `transfer_spec`).

Using the fact that `account2` refines `account1`, we want to prove that for all `st`, for all `others` and all `o` of type `account2` st, property `refund_guarantee` {st} {others} `o` holds. The proof seems a bit complex: it requires an induction over the length of the left iteration, and then an induction over the list of creditors in `o`.

A last important requirement of the application is that the account must not create or loose money. In the refinement of `account1` by `account2`, I have already expressed that the couple `(balance,debt)` is correctly handled with respect to the notion of `global_balance`. Furthermore, invariant `debt_definition` also ensure that list `creditors` is correctly handled from this point of view. The property `refund_guarantee` guarantees that `creditors` do not do not loose money. It remains to check that the `account` do not transfer more money to creditors that it has to. Actually, in this formalization, this property seems hard to express abstractly. However, it is easy to check that the model specification `account2` satisfies it. A good point here is that this property is preserved by refinement: in particular, a class refining `account2` can not call more time (or less time) a transfer operation than `account2` allows.

5.2.3.4 An OCaml implementation

To achieve this example, I give an OCaml implementation of the specification above. This implementation, class `account3`, refines `account2` in fixing the refund policy of external creditors: first arrived is first refunded. Moreover, the implementation does not keep in `creditors` those who have been refunded. In other words, `account3` preserves the following invariant: all amount in the `creditors` are positive. Hence the gluing invariant between `account2` and `account3` is something like:

```
   account3.balance=account2.balance
/\ special_equiv account3.creditors account2.creditors
```

where `special_equiv` is an equivalence over list of creditors, which is invariant by permutation of elements and invariant by deletion of couples with a null amount. Of course, this implementation could be refined in a more efficient one, using predefined FIFO queues instead of lists.

```
class account3 =
  object (self)

    val mutable balance = 0
    val mutable creditors: (int * (int -> unit)) list = []

    method private refund (amount:int) (transfer:int -> unit) =
      if amount > 0 then (
         transfer amount ;
         balance <- balance - amount ;
      )

    method debit (amount:int) (transfer: int -> unit) =
      let r =  min balance amount in
        self#refund r transfer ;
        if r < amount then
          creditors <- creditors @ [(amount - r,transfer)]
```

```
    method credit =
      let rec notify all =
        match all with
        | [] -> []
        | (amount,transfer)::others ->
            let r = (min balance amount) in
              self#refund r transfer ;
              if r < amount then
                (amount - r, transfer) :: (notify others)
              else
                notify others
      in fun amount ->
        balance <- balance + amount ;
        creditors <- notify creditors

  end
```

Bibliography

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[AZ99] P. Audebaud and E. Zucca. Deriving Proof Rules from Continuation Semantics. *Formal Aspects of Computing*, 11(4):426–447, 1999.

[Bac78] R.-J. Back. *On the correctness of refinement steps in program development*. Ph.D. thesis, University of Helsinki, 1978.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Ph.D. thesis, Université Paris 7, 1999.

[BC04] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[BFM99] J.-P. Bodeveix, M. Filali, and C. Muñoz. A Formalization of the B-Method in Coq and PVS. In *Elec. Proc. of the B-User Group Meeting at FM'99*. 1999.

[BH01a] S. Boulmé and G. Hamon. Certifying Synchrony for Free. *LNCS*, 2250, 2001.

[BH01b] S. Boulmé and G. Hamon. A clocked denotational semantics for Lucid-Synchrone in Coq. Tech. rep., LIP6, 2001.
URL `http://www-spi.lip6.fr/lucid-synchrone/lucid_in_coq_report.ps`

[Bou97] S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software*, pp. 515–529. 1997.

[Bou05] S. Boulmé. *Higher-Order Refinement In Coq (reports and Coq files)*. Web page, 2005.
URL `http://www-lsr.imag.fr/users/Sylvain.Boulme/horefinement`

[BvW98] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[BW99] M. Büchi and W. Weck. The Greybox Approach: When Blackbox Specifications Hide Too Much. Tech. Rep. 297, Turku Center for Computer Science, 1999.
URL `http://www.tucs.fi/Publications/techreports/TR297.php`

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[Coq86] T. Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*. Cambridge, MA. IEEE PRESS, 1986.

[Coq04] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.0*. Logical Project, INRIA-Rocquencourt, 2004.
URL `http://coq.inria.fr/`

[Dij69] E. W. Dijkstra. Notes on Structured Programming. Tech. Rep. EWD249, Technical U. Eindhoven, The Netherlands, 1969.
URL `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[dR03] D. de Rauglaudre. *Camlp4, release 3.07*. Projet Cristal, INRIA-Rocquencourt, 2003.
URL `http://caml.inria.fr/camlp4/`

[Fil99a] J.-C. Filliâtre. A theory of monads parameterized by effects. Research Report 1367, LRI, Université Paris Sud, 1999.
URL `http://www.lri.fr/~filliatr/ftp/publis/monads.ps.gz`

[Fil99b] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Ph.D. thesis, Université Paris-Sud, 1999.
URL `http://www.lri.fr/~filliatr/ftp/publis/these.ps.gz`

[Fil03a] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[Fil03b] J.-C. Filliâtre. *The WHY verification tool*. LRI, 2003.
URL `http://why.lri.fr/index.en.html/`

[FP03] J.-C. Filliâtre and F. Pottier. Producing All Ideals of a Forest, Functionally. *Journal of Functional Programming*, 13(5):945–956, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[Gim94] E. Gimenez. Codifying guarded definitions with recursive schemes. Tech. Rep. 95-07, LIP ENS-Lyon, 1994.

[Gor88] M. J. C. Gordon. Mechanizing Programming Logics in Higher-Order Logic. In G. M. Birtwistle and P. A. Subrahmanyam, eds., *Current Trends in Hardware Verification and Automatic Theorem Proving*. Springer-Verlag, 1988.

[Har98] J. Harrison. Formalizing Dijkstra. In J. Grundy and M. Newey, eds., *TPHOLs'98*, vol. 1497 of *LNCS*, pp. 171–188. Springer-Verlag, 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[JW93] S. L. P. Jones and P. Wadler. Imperative functional programming. In *Proc. of the 20th ACM SIGPLAN-SIGACT symposium on POPL*. ACM Press, 1993.

[LBR04] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Tech. Rep. 98-06y, Iowa State University, Department of Computer Science, 2004. See `www.jmlspecs.org`.

[LH96] S. Liang and P. Hudak. Modular Denotational Semantics for Compiler Construction. In *Proc. of ESOP'96*, vol. 1058, pp. 219–234. Springer-Verlag, 1996.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[OYR04] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of the 31st ACM SIGPLAN-SIGACT symposium on POPL*, pp. 268–280. ACM Press, 2004.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Proc. of TLCA*, no. 664 in LNCS. 1993.

[Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74. IEEE Computer Society, 2002.

[SM03] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HasCasl. In *FASE*, vol. 2621 of *LNCS*. Springer-Verlag, 2003.

[TJ92] J.-P. Talpin and P. Jouvelot. The Type and Effect Discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pp. 162–173. IEEE Computer Society Press, 1992.

[Utt92] M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. Ph.D. thesis, University of New South Wales, Kensington, Australia, 1992.

[VGPA00] K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Proc. 6th Asian Computing Science Conference (ASIAN'2000), Penang, Malaysia, Nov. 2000*, vol. 1961, pp. 162–181. Springer, 2000.

[vW94] J. von Wright. Program refinement by theorem prover. In *6th Refinement Workshop*. Springer-Verlag, 1994.
URL `http://www.tucs.fi/publications/oldpubs/reports/A94-146.ps.Z`

[Wad99] P. Wadler. The marriage of effects and monads. In *Proc. of the ACM SIGPLAN ICFP'98*, vol. 34(1), pp. 63–74. 1999.

[Wer94] B. Werner. *Une Théorie des Constructions Inductives*. Ph.D. thesis, Université Paris 7, 1994.

[WF94] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[Wir71] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

Index of definitions and properties