Université de Paris-Sud
CENTRE D'ORSAY

Rapport scientifique présenté pour l'obtention
d'une Habilitation à Diriger des Recherches

Jean-François MONIN

France Télécom R&D

Contribution aux méthodes formelles pour le logiciel

Soutenu le jeudi 4 avril 2002
devant un jury constitué par :

M.	HUET	Gérard	Président
M.	BEZEM	Marc	Rapporteur
M.	KAHN	Gilles	Rapporteur
M.	SHANKAR	Natarajan	Rapporteur
M.	BEAUQUIER	Joffroy	Examinateur
Mme	KIRCHNER	Hélène	Examinatrice
Mme	PAULIN-MOHRING	Christine	Examinatrice
M.	SIFAKIS	Joseph	Examinateur

Table des matières

Ce mémoire parcourt les travaux de recherche de l'auteur. Ces derniers portent sur quelques aspects de ce qu'il est convenu d'appeler les méthodes formelles : l'analyse et la conception de programmes Prolog, le logiciel de vérification de protocoles Véda, la preuve de programmes comportant des exceptions et les systèmes temps-réel.

Le chapitre 1 contient une rapide introduction au domaine (pour plus de détails on pourra se reporter à [99]) ainsi que quelques conclusions personnelles.

L'utilisation de Prolog dans le cadre de Véda a été relatée dans ma thèse de doctorat [90]. Les résultats présentés étaient alors justifiés semi-formellement. En fait la théorie des types et plus particulièrement le calcul des constructions inductives (CCI) [112] aurait permis de formaliser directement la plupart des raisonnements effectués. Le chapitre 2 reprend les principes utilisés à la lumière de CCI et revient sur l'interprétation de certains programmes à l'aide de la logique linéaire, une idée qui restera à développer.

Les preuves de programmes comportant des exceptions font l'objet du chapitre 3. Le seul ajout par rapport aux publications existantes est l'exemple du codage de Huffman.

Le chapitre 4 porte sur une étude de cas : un petit algorithme réactif (le contrôle de conformité ABR), qui a été traité par différentes approches. C'est aussi l'un des éléments fondateurs du projet RNRT CALIFE que j'anime actuellement. Ce sera l'occasion de discuter quelques uns des modèles utilisés pour décrire des systèmes temps-réel.

Chapitre 1

La vérification des systèmes logiciels

Quiconque a programmé se rend vite compte de l'immensité des situations potentiellement parcourues par la (ou les) machine(s) supportant l'exécution d'un logiciel tant soit peu réaliste. S'assurer qu'un programme se comporte comme prévu dans tous les cas est une tâche ardue, encore trop souvent hors de portée des moyens techniques à notre disposition ou incompatible avec les contraintes économiques ou sociales (formation des informaticiens, délais de livraison, etc.).

Pour prendre la mesure de la difficulté, il faut d'abord se souvenir que les systèmes considérés, essentiellement *discrets*, n'ont pas la stabilité des systèmes analogiques traités par les mathématiques traditionnelles : la moindre variation dans le code peut donner lieu à un changement disproportionné du résultat. Pour ne citer qu'un exemple célèbre, la substitution du point à la virgule de l'instruction Fortran `DO 10 I = 1,5` transforme une boucle en affectation, et c'est ainsi que l'une des premières sondes spatiales aurait été perdue. De tels risques sont sans doute atténués par les redondances introduites à dessein dans les langages modernes – au niveau de la syntaxe et par le biais du typage – ainsi que par les analyses statiques comme la détection de code mort. La difficulté essentielle reste : chaque pas d'exécution est un bouleversement en puissance. Le comportement d'un programme se résume à une succession de choix dont l'issue est en général imprévisible. En vue d'une analyse complète il faut appréhender toutes les possibilités, faire face à l'explosion combinatoire des cas à considérer.

Des outils adaptés, conceptuels et pratiques, sont donc nécessaires. Nous nous intéresserons ici à ceux qui ne font aucune concession à la rigueur, s'appuyant sur des formulations ayant un sens précis et utilisant le raisonnement mathématique : on les appelle les *méthodes formelles*. En voici un aperçu.

1.1 La démarche de vérification

On a tout d'abord besoin d'un modèle du système sur lequel on souhaite travailler : il peut s'agir d'un dispositif matériel ou d'un composant logiciel et de leur environnement. Schématiquement, il est nécessaire de décrire l'espace d'états dans lequel évolue le système ainsi que la dynamique de ce dernier – les changements d'états et leurs causes. On étudie ensuite les propriétés de ce modèle. Certaines sont génériques, par exemple l'impossibilité d'atteindre un état puits (dont on ne ressort pas) à partir de la configuration initiale. Mais les propriétés les plus significatives dépendent généralement du problème considéré et doivent être formulées explicitement. Il faut alors être en mesure de vérifier, automatiquement ou non, que les propriétés attendues sont satisfaites sur le modèle. Si ce n'est pas le cas, il se peut que le système considéré soit effectivement incorrect, mais aussi que le modèle ne reflète pas bien la réalité, que les propriétés soient mal formulées ou que la vérification soit défectueuse. Les trois dernières possibilités restent d'ailleurs à envisager dans tous les cas.

1.2 Fidélité du modèle

Établir des modèles est une activité traditionnelle dans les sciences telles que la physique ou la chimie. Un modèle fournit une grille d'analyse d'un pan de la réalité, sa fidélité est jugée en examinant les conséquences qui en découlent et en les mettant à l'épreuve de l'expérience.

La situation se présente différemment en informatique : il s'agit généralement, non de modéliser la réalité, mais de la modeler. Ainsi une spécification en Estelle (de nos jours, en LDS [66]) sert de texte de référence à la construction d'un dispositif ou d'une couche de protocole. Même si ceci n'est pas toujours conforme à la pratique – les spécifications sont informelles pour la plupart – les langages de spécification formelle sont conçus en ce sens. Au delà du monde des télécommunications, on observe que B [2] est utilisé exactement de cette manière.

On retrouve cependant la démarche de la physique dès que l'on s'intéresse à l'environnement du dispositif modélisé. Le modèle complet prend en compte, explicitement ou non, des hypothèses indiquant comment l'environnement interagit avec le système. Dans le cas d'un programme séquentiel représenté par une fonction, l'interaction se manifeste simplement sous forme d'une précondition. Un programme réactif donne lieu à des hypothèses sur les temps de réaction respectifs du système et de l'environnement. Ainsi, afin de simplifier la conception et le raisonnement, on suppose dans les langages synchrones comme Estérel [16], que le système réagit instantanément, c'est-à-dire plus vite que son environnement. Cette hypothèse a été reprise dans le

cadre de la vérification de l'algorithme de conformité de l'ABR[1]. Elle revient, plus précisément, à admettre que :

- la mise à jour de `ACR`, lors d'un événement programmé par l'échéancier, est plus rapide que le traitement d'une cellule en provenance de l'utilisateur ;
- la prise en compte d'une cellule RM en provenance du réseau (la partie compliquée de l'algorithme) est effectuée avant l'arrivée de la cellule RM suivante et a une durée inférieure à τ_3.

Pour prendre un autre exemple, les modèles établis dans le cadre de la tolérance aux fautes font des hypothèses sur le rythme d'arrivée des fautes et la nature de ces dernières [126].

Pour comprendre la portée exacte d'un modèle, il est donc important de mettre en évidence aussi explicitement que possible les hypothèses considérées. Leur exactitude peut être contrôlée par des tests.

1.2.1 Influence de l'usage

Un modèle peut avoir divers usages : la vérification de propriétés fonctionnelles (celles auxquelles je me suis surtout intéressé), mais aussi la communication au sein d'un projet, l'analyse de propriétés quantitatives, stochastiques, l'établissement de jeux de test, le support à l'animation en phase de mise au point et bien d'autres. Du fait de cette variété d'usages, un modèle peut revêtir plusieurs formes différentes. Cela soulève un problème de cohérence entre ces dernières. Pour traiter cette question, une idée naturelle consiste à passer par un formalisme pivot : dans le cadre du projet européen RACE SPECS [38] il s'agissait d'une algèbre de processus nommée CRL [2] ; plus récemment, le projet ALTARICA [117] a repris une démarche analogue autour des systèmes de transitions étiquetées.

1.2.2 Influence des outils

Les outils disponibles, conceptuels ou logiciels, ont une influence déterminante sur l'élaboration d'un modèle. Ainsi, pour l'analyse de performances, un modèle à base de files d'attentes s'impose, parce qu'il est possible de conduire des calculs appropriés dans ce formalisme et que des algorithmes efficaces de simulation existent. En vérification, on retrouve ce phénomène avec de nombreuses techniques automatiques. La démarche utilisée consiste essentiellement à réduire le problème posé à un problème connu, tel que : la résolution d'équations booléennes, le calcul d'un point fixe dans un domaine fini, la décision de formules habitant un fragment de l'arithmétique. Il n'est pas nécessaire de présenter le modèle directement sous l'une de ces formes :

[1] *Available Bit Rate* : une capacité normalisée pour l'ATM.

[2] *Common Representation Language* ; par la suite ce formalisme a été réduit et a pris le nom de μCRL [58].

le succès du *model checking* repose en grande partie sur la traduction de systèmes d'automates en conjonctions de formules booléennes représentées de manière compacte, comme les BDD. Malgré tout, le modèle subit des biais afin de respecter les contraintes dictées par la théorie (décidabilité) ou la pratique (consommation mémoire). Par exemple on approximera le modèle de façon à borner à l'avance le nombre d'états qu'il peut atteindre.

Par ailleurs, la frontière entre le décidable et l'indécidable est parfois très étroite, voir par exemple les résultats exposés dans [22]. La tâche de modélisation et d'interprétation demande alors un soin particulier. Il faut choisir les abstractions, voire les approximations appropriées : celles qui satisfont les outils utilisés et conduisent à des résultats pertinents.

1.2.3 Modèles mathématiques

Les approches qui s'appuient d'abord sur un langage expressif (et non sur un algorithme d'analyse) souffrent moins de ces inconvénients. Comme en mathématiques, on définit progressivement les notions utiles au modèle à partir de concepts élémentaires. On peut au besoin démontrer au passage des lemmes afin de contrôler que les notions définies sont conformes à l'intuition [43]. C'est ainsi, par exemple, que nous avons procédé pour l'étude de l'algorithme de conformité de l'ABR.

Reste à choisir un cadre mathématique convenable : pour appréhender la complexité des situations réelles, il faut pouvoir construire des ensembles et des fonctions arbitrairement complexes. La théorie des ensembles [61, 46] a toute la flexibilité désirable. La théorie des types permet un meilleur contrôle des énoncés sans – sous sa forme moderne [41, 51, 80, 135] – sacrifier la puissance d'expression.

Une telle approche peut également venir en complément d'une approche automatisée afin de justifier formellement les simplifications effectuées sur un modèle ou de préciser leur portée exacte. Ceci est exploré avec des environnements comme HOL [54, 55], Isabelle [113, 115], Coq [134, 64] et est appliqué aux systèmes temporisés dans le projet RNRT Calife (voir 4.6.2.

On notera que l'utilisation de la théorie des ensembles dans des méthodes formelles comme TLA [73, 1], Z [118, 131, 138] ou B [2] est assez différente : elle est au service d'une démarche délimitée. En B par exemple, elle sert à définir le cadre de travail (la spécification, les substitutions généralisées et le raffinement).

1.2.4 Modèles proches du programme

Pour terminer, mentionnons qu'un modèle sur lequel il est parfois possible de travailler est le programme lui-même. Aller à ce niveau de détail peut être contre-productif en particulier lorsqu'on veut simplement valider le principe d'un algorithme, mais c'est pertinent pour du code critique. Les

techniques d'extraction de programme peuvent être placées dans cette perspective : l'utilisateur construit incrémentalement le programme désiré. Nous y reviendrons au chapitre 3.

1.3 Formulation des propriétés

1.3.1 Formulation opérationnelle et formulation axiomatique

On peut distinguer deux façons de décrire les propriétés attendues d'un système. L'une est d'effectuer une comparaison avec un système de référence décrit dans le même langage. Dans cette catégorie on peut ranger l'utilisation de la bisimulation [84, 85], la technique des observateurs introduite par Roland Groz dans Véda [68, 70] dès 1985 et le produit synchronisé avec un automate [8, 9]. L'autre est d'utiliser un langage logique : logique des prédicats, calcul des constructions [41], logique temporelle [36, 73, 116, 120], μ-calcul [10, 23], etc. Bien entendu, il ne s'agit que d'une différence de présentation, puisque même dans la première catégorie, on vise une propriété logique exprimant « *le système A se comporte comme le système B* ». Il peut y avoir deux raisons pour préférer une telle formulation :

- une technique de vérification adaptée est disponible ;
- cette formulation est la plus claire.

Notons que le μ-calcul (une logique temporelle) a été utilisé avec succès chez Ericsson pour la vérification d'un composant de SGBD distribué écrit en Erlang [11]. Les formules utilisées sont non triviales (elles comprennent des alternances de points fixes), mais on peut les lire comme un programme fonctionnel de référence.

1.3.2 Influence des outils

Les techniques de vérification disponibles ont une influence non seulement sur le modèle, ainsi que nous l'avons vu au 1.2), mais aussi sur l'expression des propriétés. Prenons le cas de la vérification de l'algorithme de conformité de l'ABR par des techniques de *model checking*. Schématiquement, il s'agit de vérifier qu'une valeur observable délivrée par un dispositif est toujours au moins aussi grande que le maximum de certaines quantités reçues par ce dispositif au cours d'un intervalle de temps lui-même variable (voir chapitre 4). Lors d'une première tentative, l'équipe du LaBRI a cherché à reproduire la propriété désirée au moyen d'un automate effectuant continuellement les calculs de maximum et codé aussi naïvement que possible, de façon à écarter les doutes quant à la fidélité de la transcription. Cette approche s'est malheureusement heurtée à une explosion du nombre d'états. La tentative suivante a été couronnée de succès [25], au prix d'un abaissement des ambitions quant au codage de la propriété : l'automate simule une version optimisée du calcul de maximum, d'ailleurs issue de la preuve initiale.

1.3.3 Démarche générale

Ceci illustre un fait qui semble général : énoncer la propriété à vérifier exige d'autant plus de doigté que le formalisme choisi est limité en capacité d'expression. Or cela tend à se produire plus souvent avec l'automatisation de la vérification. Pour bénéficier des outils de vérification automatique, on n'échappe pas à une réflexion serrée sur la formulation de la propriété désirée. Si la propriété attendue ne se présente pas directement en termes simples, il faut introduire les définitions nécessaires à l'élaboration du niveau d'abstraction convenable, relier une propriété exprimée aussi clairement que possible et sa forme acceptable en entrée d'un outil automatique. Il s'agit d'une démarche mathématique, d'où l'intérêt, là encore, d'un environnement supportant la formalisation de mathématiques générales pour apporter des garanties complémentaires.

1.4 Fiabilité de la vérification

1.4.1 Problèmes rencontrés

Vérifier qu'un système satisfait une propriété, fondamentalement, c'est démontrer un théorème. On sait, en théorie, mener à bien ce travail dans le cas des programmes séquentiels depuis les années 1960 [48, 63]. Cependant, la distance est grande entre un raisonnement semi-formel esquissé en première approximation et une démonstration complète. La quantité d'inférences à manipuler devient rapidement décourageante ; les détails s'avèrent fastidieux, de sorte qu'une erreur se glisse facilement à la longue. Pour des vérifications plus ambitieuses (programmes volumineux, concurrents, objets, temps-réel, etc.) le problème se pose avec encore plus d'acuité, les opportunités pour tomber dans un piège subtil se présentent souvent ! Un exemple a été mentionné par John Rushby à FM'99 : il s'agissait d'une preuve d'algorithme publiée par un chercheur de grande renommée (L. Lamport) dans un journal tout aussi renommé (IEEE), reposant sur cinq lemmes et un théorème principal ; la démonstration de ce dernier et celle de quatre des lemmes étaient légèrement erronées – mais pouvaient être réparées.

On a retrouvé ce phénomène au niveau de l'expérience de l'algorithme de conformité de l'ABR : ma première preuve réutilisait indûment, vers la fin, un raisonnement effectué plus haut – même si là encore, le résultat final était correct.

Même si le cheminement principal s'avère souvent correct, une démonstration rédigée sur le papier laisse subsister un doute. Il serait aventureux de s'y fier totalement, surtout dans un contexte industriel et pour des composants d'importance critique. Cela ne doit pas être compris comme un procès intenté aux capacités de telle ou telle personne : des mathématiciens de génie ont aussi commis des bévues.

Il faut en outre prendre en compte l'évolution des logiciels. À chaque modification il faut s'assurer que les raisonnements effectués sur la version précédente sont toujours valides, ou les adapter si besoin. Une tâche bien fastidieuse...

1.4.2 Nécessité d'un support automatisé

Il est donc nécessaire que les tâches de vérification soient supportées par des outils d'automatiques, eux-mêmes fiables. Ceci ne peut se faire entièrement automatiquement dans le cas général du fait de limitations bien connues, soit strictes (indécidabilité), soit pratiques (complexité). Idéalement, on aimerait borner l'intervention de l'utilisateur à la partie créative du travail et que les détails soient calculés par la machine. Avec les assistants de preuve, la « partie créative » réside dans les lignes essentielles du raisonnement suivi. Dans les approches fondées sur des outils automatiques, elle consiste à établir un modèle et à formuler la propriété désirée de telle sorte que le problème se trouve dans une classe décidable ou semi-décidable. La vérification se déroule alors en deux phases : un travail de préparation qui n'est pas couvert par l'outil automatique, puis l'invocation de l'outil sans aucune interaction de l'utilisateur ou presque, suivant les outils[3]. En ce qui concerne l'automatisation, il reste plus de chemin à parcourir pour les assistants de preuve : ils ont l'ambition de traiter la totalité de la vérification. Un courant de recherche actuel, représenté notamment par PVS [108, 107], attaque précisément ce problème.

1.4.3 Fiabilité des outils de preuve

Une autre courant a mis l'accent sur la question de la fiabilité des outils de support. L'architecture des assistants issus de la tradition de LCF [56] est organisée autour d'un petit noyau écrit très soigneusement au crible duquel sont passées toutes les démonstrations effectuées. Cela est possible lorsque la logique utilisée se réduit elle-même à un nombre très restreint d'éléments primitifs. Par exemple, la logique de Coq repose essentiellement sur un seul connecteur logique ($\forall$), une forme très générale de récurrence et une notion interne de réduction. Une telle architecture est ouverte : un programme annexe spécialisé dans la recherche de démonstrations relatives à un domaine précis pourra être mis en service à la demande de l'outil principal sans que soit remise en cause l'intégrité logique de ce dernier puisqu'en dernière instance, le noyau contrôle la justesse des démonstrations proposées.

La fiabilité de ces assistants à la preuve reposant entièrement sur celle de leur noyau, ce dernier fait l'objet d'une grande vigilance. Mieux : lorsque

[3] Le *model checking* est entièrement automatique. D'autres procédures fondées sur la réécriture, mises en œuvre dans Spike [20, 21] par exemple, peuvent demander des interventions comme l'introduction d'un lemme.

l'on dispose d'une logique vraiment puissante, pourquoi ne pas formaliser et vérifier mécaniquement le fonctionnement du noyau ? Cet objectif a été atteint par B. Barras dans le cas du calcul des constructions inductives [13], ouvrant la possibilité que le noyau d'une version ultérieure de Coq soit obtenu par extraction.

1.4.4 Vers une synthèse

On aimerait à terme disposer du meilleur de tous ces mondes et les recherches en cours semblent prometteuses. Ainsi, dans le projet RNRT Calife, C. Alvarado et Q.H. Nguyen ont mis en place des procédures de communication permettant à Coq de déléguer la recherche de démonstrations d'égalités à Elan, un outil très efficace dans ce domaine [19]. Le passage à l'échelle se fait, sans concession à la sécurité, grâce à l'emploi d'une technique appelée la réflexion, qui diminue considérablement le nombre d'inférences à contrôler (elles sont remplacées par des réductions bien moins coûteuses).

1.5 Bilan

Au delà des possibilités ou des limitations des techniques, qui évoluent sans cesse, j'aimerais défendre l'idée que l'impact des méthodes formelles ne se borne pas à l'histoire de ses échecs ou de ses succès. Il réside aussi, de manière plus profonde, dans le développement d'une attitude plus scientifique face aux problèmes posés. Cela me paraît digne d'être souligné car trop souvent l'informatique n'est perçue que comme une mosaïque de technologies.

Certes, elle se pratique le plus souvent de manière expérimentale ou empirique, faute de mieux. Cependant, l'informaticien – comme tout un chacun – a tout avantage à adopter une démarche scientifique, consistant d'abord à élaborer des modèles et à les mettre en cause. Comme dans toute science, l'étape de modélisation consiste à choisir un niveau d'abstraction approprié, à effectuer des simplifications. Certaines sont salutaires – seules les facettes pertinentes doivent être retenues. D'autres sont rendues nécessaires par les limitations des outils conceptuels ou expérimentaux disponibles. Il convient de bien garder à l'esprit la distance qui subsiste entre tout modèle et la réalité qu'il représente.

La formalisation de tout cela, à un degré plus élaboré, relève des mathématiques appliquées : poser des définitions et démontrer des théorèmes. Cette activité est clairement présente lorsque l'on fait appel à un assistant de preuve. Dans un environnement supporté par des outils de vérification automatique, la construction d'un modèle et la formulation des exigences conduisent également à un travail d'abstraction et à des justifications, même si elles sont informelles.

En conclusion, la vertu pédagogique des méthodes formelles me paraît mériter la plus grande attention dans l'enseignement de notre discipline.

Chapitre 2

De la logique pour un compilateur

Mes premiers travaux au CNET se sont effectués au sein d'une équipe dont le thème d'étude était l'ingénierie des protocoles à partir de spécifications formelles. C'est ainsi qu'a été conçu l'outil de vérification par simulation Véda [1]. Les idées fondatrices de ce projet sont dues à Claude Jard et Roland Groz[2]. Ma contribution a porté sur l'architecture et sur la réalisation de Véda [12, 68, 69, 70]. En dehors des techniques de programmation Prolog évoquées plus bas, j'ai formalisé et mis au propre le mécanisme de génération de code, dont les principes sont simples – parcourir des termes de gauche à droite en produisant du texte, mais qui se complique lorsqu'on prend en compte la compilation séparée. Le code cible est alors engendré par pièces devant être assemblées ultérieurement ; il en résulte un procédé d'ordonnancement fondé sur un calcul statique de flux d'informations, dont la correction est justifiée au moyen de quelques définitions et théorèmes. Ces derniers sont *ad-hoc* et on n'y reviendra pas ici. Ils ont cependant constitué ma première expérience des méthodes formelles où apparaissait l'utilité de disposer d'un cadre assez large pour permettre le développement d'arguments mathématiques spécifiques à la demande.

Les outils issus de ce travail se sont avérés utiles pour la maintenance de Véda (de manière à suivre la normalisation à l'ISO du langage d'entrée, Estelle), une tentative d'outiller l'évaluation des performances de protocoles [139] et plus tard la réalisation, par Marc Phalippou, du premier prototype de TVéda, un outil de génération de jeux de tests.

Véda a été industrialisé et réécrit dans des langages plus conventionnels par Vérilog. Il n'est plus maintenu depuis plusieurs années, ayant évolué

[1] VÉrification D'Algorithmes distribués.

[2] En particulier la complémentarité d'une technique par simulation, couvrant partiellement les comportements d'un modèle fin, et d'une technique par *model checking*, couvrant exhaustivement les comportements d'un modèle grossier : cela donne deux techniques qui augmentent la confiance que l'on peut avoir dans un système.

(complété par d'autres sources) vers Object-Geode, qui prend en entrée du LDS à la place de Estelle. La version industrialisée de Véda était cependant encore utilisée chez GemPlus à la fin des années 90 pour des protocoles liés aux cartes à puce.

Choix de Prolog pour Veda

Le choix de Prolog pour les travaux relatés ci-dessus mérite discussion : pourquoi ne pas employer d'emblée les techniques de compilation traditionnelles, qui ne présentaient aucun risque d'inefficacité ? En dehors du désir ludique de chercher les limites d'une technique séduisante en théorie, d'après les travaux de Colmerauer et Warren, mais peu explorée en pratique, il se trouve que le langage à analyser était en cours d'élaboration à l'ISO ; il était important d'en suivre les évolutions. À cet effet, l'expression concise et calquée sur la grammaire du langage d'entrée apparaissait un atout décisif pour Prolog.

Plus précisément, les clauses Prolog, usuellement présentées comme des spécifications logiques exécutables, étaient surtout un moyen élégant d'exprimer des manipulations de termes, ce qui s'avèrait particulièrement utile dans la programmation d'outils de transformation de structures syntaxiques. Cependant, on se heurtait à des problèmes de performances dès qu'il s'agissait d'analyser des textes de plusieurs centaines ou milliers de lignes écrits dans un langage comme Estelle, dont la grammaire avait une taille de l'ordre du double de celle de Pascal. L'enjeu était de recourir le moins possible au retour arrière en exploitant l'autre mécanisme calculatoire typique de Prolog : l'unification. Ainsi ont été systématiquement utilisées les *listes différentielles* ou des structures analogues. Elles sont souvent présentées comme un jeu de pointeurs [35], mais le procédé utilisé pour Véda recourait à une formulation intermédiaire plus propre issue du λ-calcul.

Plus généralement, il fallait constamment garder à l'esprit la complexité en temps ou en mémoire des programmes Prolog. Les raisonnements permettant d'évaluer la complexité et de justifier la correction des programmes [37, 87, 89, 92, 90] étaient alors menés informellement.

Le résultat s'est avéré satisfaisant, comme en témoigne la diffusion de Véda dans une dizaine de laboratoires français, à des fins de recherche et d'enseignement : la « compilation »[3] d'un millier de lignes prenait environ deux minutes sur une machine Multics, ce qui à l'époque restait acceptable.

Raisonner formellement sur des programmes Prolog

Le principal obstacle rencontré à l'époque pour formaliser les raisonnements effectués était l'absence de notions maniables et d'outils de support appropriés. Le programmeur Prolog n'a officiellement accès qu'à une vision

[3] Véda ne produisait pas du code machine, mais du Pascal.

extensionnelle du résultat : une réponse sous forme de substitution. Dans les faits, le processus de formation des arbres de preuve lui importe tout autant. Traditionnellement ([72, 132] par exemple), Prolog est présenté à partir de la procédure de résolution de Herbrand-Robinson [124, 32], où les arbres de preuve comportent des nœuds étiquetés par des clauses. Pour obtenir une caractérisation plus précise il faut en outre un opérateur de point fixe [77].

Or il est bien connu que la règle de résolution correspond à une coupure en calcul des séquents [51] et que le fragment de la logique utilisé dans Prolog est intuitionniste : les clauses de Horn peuvent être vues comme des séquents intuitionistes. On peut alors se placer dans le cadre de la déduction naturelle, pour y gagner une notion de preuve plus élémentaire que celle du calcul des séquents [119]. Cela permet d'économiser la notion de résolvante et de raisonner directement sur la formation d'arbres de preuve qui, au demeurant, me semblent correspondre à ceux qui sont naturellement appréhendés par le programmeur.

Il paraît aujourd'hui très naturel d'assimiler les programmes Prolog à des définitions inductives du calcul des constructions inductives, à condition de ne pas se laisser perturber par les notions liées au contrôle. Ainsi, nous ignorerons le coupe-choix, considérant que son usage sain peut être représenté par une négation.

Présentation du chapitre

Le reste de ce chapitre vise à illustrer comment formaliser, au moins en principe, les raisonnements effectués dans le cadre de Véda. Au passage sera introduit le calcul des constructions inductives (CCI), qui nous sera utile dans les chapitres suivants. Seuls des exemples élémentaires seront considérés.

Nous commençons par des rappels sur Prolog (2.1) suivant le point de vue ci-dessus ainsi que sur CCI (2.2). Après une parenthèse au sujet d'un fameux slogan de R. Kowalski qui avait contribué à la popularité de Prolog (2.3), nous abordons au 2.4 le raisonnement sur des programmes. A titre de perspectives, la section 2.5 esquisse un lien à développer entre Prolog et la logique linéaire, qui est un bon candidat pour expliquer les techniques de programmation telles que les listes différentielles.

2.1 Prolog

Rappelons que la programmation en logique consiste à décrire un univers au moyen de formules de la logique du premier ordre et d'en rechercher les conséquences répondant à une requête donnée. Pour illustrer les notations utilisées ici, voici la relation qui est vraie entre trois listes lorsque la troisième est la concaténation des deux premières. Le vocabulaire utilisé comprend la constante 0-aire nil, le symbole de fonction binaire « . » utilisé sous forme

infixe, avec un élément à gauche et une liste à droite, et le symbole de prédicat conc dont les trois arguments sont des listes.

$$\forall l \quad \mathsf{conc}(\mathsf{nil}, l, l). \tag{2.1}$$

$$\forall x, l_1, l_2, l_3 \quad \mathsf{conc}(l_1, l_2, l_3) \Rightarrow \mathsf{conc}(x.l_1, l_2, x.l_3). \tag{2.2}$$

Quelques requêtes possibles sont :

$$\mathsf{conc}(1.2.3.\mathsf{nil}, 4.5.\mathsf{nil}, 1.2.3.4.5\mathsf{nil}) \tag{2.3}$$

$$\exists\, l \quad \mathsf{conc}(1.2.3.\mathsf{nil}, 4.5.\mathsf{nil}, l) \tag{2.4}$$

$$\exists u, v \quad \mathsf{conc}(1.2.3.\mathsf{nil}, u, v) \tag{2.5}$$

Remarquons que l'ensemble des réponses à (2.5) est décrit sous forme d'un seul couple $\langle u, 1.2.3.u\rangle$, où u est quantifié universellement. Cela peut s'interpréter en disant que le mécanisme interne de Prolog contient alors une fonction qui attend une liste close pour rendre une réponse complètement close.

Un autre programme typique qui nous servira d'illustration est la recherche d'un chemin dans un graphe, c'est-à-dire le calcul de la clôture transitive clotrans d'une relation arc.

$$\forall x \quad \mathsf{arc}(x, y) \Rightarrow \mathsf{clotrans}(x, y). \tag{2.6}$$

$$\forall x, y, z \quad \mathsf{clotrans}(x, y) \wedge \mathsf{clotrans}(y, z) \Rightarrow \mathsf{clotrans}(x, z). \tag{2.7}$$

2.1.1 La résolution : calcul = recherche de démonstrations

La *procédure de résolution* de Herbrand et Robinson est un mécanisme opératoire capable de calculer toutes ces réponses – et bien d'autres – de manière uniforme. Il faut que la formule constituant le programme soit préalablement mise sous forme d'une conjonction de clauses, qui sont des disjonctions de littéraux quantifiées universellement. Dans le cas de Prolog, on se restreint à celles qui ne comportent qu'un seul littéral positif (dites *de Horn*), que l'on peut donc présenter sous la forme

$$\forall x \ldots A_1 \wedge \ldots \wedge A_n \Rightarrow B.$$

Les deux ingrédients principaux de la procédure de résolution sont :

- la reprise sur échec, ou retour arrière (*backtracking*) qui permet d'énumérer les solutions ;
- l'algorithme d'unification qui calcule la plus petite substitution égalisant deux termes.

D'un point de vue logique, le calcul construit un arbre de preuve dont la racine est la formule donnée en but[4]. Pour fixer les idées, voici deux exemples

[4]La procédure de résolution est traditionnellement présentée selon le point de vue sémantique, comme la recherche d'un modèle (syntaxique) falsifiant la conjonction des clauses du programme et de la négation du but. Girard et Miller ont observé que le point de vue de la théorie de la démonstration est tout aussi pertinent.

d'arbres de preuve. Le premier correspond à (2.3) et (2.4). Il ne comporte qu'une branche.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{conc}(\mathsf{nil}, 4.5.\mathsf{nil}, 4.5.\mathsf{nil})}}{\mathsf{conc}(3.\mathsf{nil}, 4.5.\mathsf{nil}, 3.4.5.\mathsf{nil})}}{\mathsf{conc}(2.3.\mathsf{nil}, 4.5.\mathsf{nil}, 2.3.4.5.\mathsf{nil})}}{\mathsf{conc}(1.2.3.\mathsf{nil}, 4.5.\mathsf{nil}, 1.2.3.4.5.\mathsf{nil})}$$

Pour notre second exemple, nous nous donnons une relation arc comportant les clauses $\mathsf{arc}(1,2)$, $\mathsf{arc}(2,3)$ et $\mathsf{arc}(3,5)$.

$$\dfrac{\dfrac{\dfrac{\mathsf{arc}(1,2)}{\mathsf{clotrans}(1,2)} \quad \dfrac{\mathsf{arc}(2,3)}{\mathsf{clotrans}(2,3)}}{\mathsf{clotrans}(1,3)} \quad \dfrac{\mathsf{arc}(3,5)}{\mathsf{clotrans}(3,5)}}{\mathsf{clotrans}(1,5)}$$

La construction s'effectue progressivement, à partir de la racine, par essai des différentes clauses pouvant démontrer une feuille qui n'est pas un axiome du programme, s'il en existe une (un axiome est une clause constituée d'un unique littéral positif, comme arc dans notre dernier exemple). En Prolog, le choix de la feuille se fait en profondeur d'abord, de gauche à droite.

2.1.2 La résolution pour des formules du premier ordre

Dans le cas général, il faut considérer des arbres ayant des formules munies de variables. Celles-ci sont quantifiées globalement sur l'arbre complet. La confrontation d'une feuille F à une clause C conduit à tenter de calculer une substitution des variables de l'arbre courant et de la clause C de sorte que F et le littéral positif P de C soient identiques. Cette procédure, appelée *l'unification* fournit en fait la substitution la plus générale possible en parcourant simultanément F et P. Par exemple si F est $\mathsf{conc}(1.2.3.\mathsf{nil}, u, v)$ on choisira la clause (2.2) et la substitution $x := 1, l_1 := 2.3.\mathsf{nil}, l_2 := u, v := 1.l_3$. De proche en proche on construit l'arbre de preuve suivant correspondant à la requête (2.5).

$$\dfrac{\dfrac{\dfrac{}{\mathsf{conc}(\mathsf{nil}, u, u)}}{\mathsf{conc}(3.\mathsf{nil}, u, 3.u)}}{\mathsf{conc}(2.3.\mathsf{nil}, u, 2.3.u)}}{\mathsf{conc}(1.2.3.\mathsf{nil}, u, 1.2.3.u)}$$

Le programmeur Prolog est parfaitement conscient de ces mécanismes, qui fondent l'interprétation procédurale d'un ensemble de clauses :

- ces dernières deviennent des procédures récursives avec retour arrière ;
- le passage de paramètres s'effectue par unification.

2.2 Le calcul des Constructions Inductives

Il est souvent nécessaire de mener des raisonnements sur des programmes Prolog, y compris sur leur exécution. Nous avions souvent procédé ainsi pour

Véda, de manière informelle ou semi-formelle. La notion principale est bien sûr celle d'arbre de preuve. Celle qui nous convient se formalise bien dans la théorie de types de Martin-Löf [80], ou dans la partie *types inductifs* du Calcul des Constructions Inductives (CCI) [40, 112]. C'est également dans le cadre de CCI que nous nous placerons au chapitre 3 à propos de la preuve de programmes fonctionnels.

Le calcul des constructions inductives est à la fois une logique, définie par des règles d'inférence, et un langage de programmation purement fonctionnel typé statiquement. Il est mis en œuvre dans l'outil Coq [134] ainsi que dans Lego [78]. Il comporte un λ-calcul typé, le calcul des constructions [39, 41], et des objets arborescents définis inductivement – chaque type d'arbre est décrit exhaustivement par ses constructeurs.

2.2.1 Types de données simples

Ici encore nous introduisons les notations sur quelques exemples. Le type des entiers naturels est défini au moyen de deux constructeurs O et S, tandis que celui des listes d'entiers possède deux constructeurs nil et cons.

Inductive nat : Set :=
| O : nat
| S : nat → nat.

Inductive listnat : Set :=
| nil : listnat
| cons : nat → listnat → listnat.

La notation $x : T$ indique que x est de type T. Ainsi O est de type nat qui est lui même de type Set. Les x tels que $x : T$ sont appelés les habitants de T. L'application d'une fonction (ou d'un constructeur) est notée $(f\ x)$. Elle associe à gauche : $(f\ x\ y) = ((f\ x)\ y)$. De manière cohérente $A \to B \to C = A \to (B \to C)$ est le type d'une fonction f qui, appliquée à x de type A, rend une fonction de B vers C. Nous continuerons parfois à employer la notation usuelle $f(x)$, $f(x, y)$... lorsque tous les arguments d'une fonction sont connus. ainsi que $x.u$ à la place de $(\mathsf{cons}\ x\ u)$. Les constantes entières seront notées 1,2,... au lieu de (S O), (S (S O))...

Le type des listes peut être paramétré par le type A des éléments :

Inductive list [A :Set] : Set :=
| nil : (list A)
| cons : A → (list A) → (list A).

Ainsi, on retrouve le type des listes d'entiers en écrivant (list nat), mais on dispose aussi du type des listes de listes d'entiers (list (list nat)), etc.

2.2.2 Fonctions et procédure de réduction

CCI contient un λ-calcul, on peut par exemple définir le double successeur

$$\mathsf{ds} \stackrel{\text{déf}}{=\!=} \lambda n\,(\mathsf{S}\,(\mathsf{S}\,n)). \tag{2.8}$$

On peut aussi définir des fonctions par cas sur un argument, par exemple l'addition et la concaténation :

$$\mathsf{plus} \stackrel{\text{déf}}{=\!=} \lambda nm\ \mathbf{case}\,n\,\mathbf{of}\,\mathsf{O} \to m \mid (\mathsf{S}\,p) \to (\mathsf{S}\,(\mathsf{plus}\,p\,m)). \tag{2.9}$$

$$\mathsf{app} \stackrel{\text{déf}}{=\!=} \lambda uv\ \mathbf{case}\,u\,\mathbf{of}\,\mathsf{nil} \to v \mid x.l \to x.(\mathsf{app}\,l\,v). \tag{2.10}$$

En l'occurrence ce sont des définitions de plus petits points fixes. Elles sont acceptées car elles comportent un argument privilégié qui diminue structurellement dans tout appel récursif.

Parmi les termes de type nat on aura donc, en plus de O, (S O), (S (S O))... des expressions comme (ds O). C'est alors qu'intervient la *procédure de réduction* ou *procédure d'évaluation*, qui itère l'expansion des constantes définies, les choix pour **case** et les β-réductions. Elle ne peut aboutir qu'à des entiers formés à partir de S et de O. Cela justifie d'ailleurs que pour définir une fonction prenant en argument un x de type nat (ou plus généralement T), il suffise de considérer les cas où x est formé à partir des constructeurs de nat (respectivement de T). Une expression qui ne peut être réduite est dite *normale*. C'est en particulier le cas si elle est formée seulement au moyen de constructeurs.

2.2.3 Arbres de preuve et types dépendants

Les arbres de preuve de Prolog peuvent être représentés par des termes normaux de CCI. Cela se voit immédiatement sur des clauses propositionnelles :

$$\mathsf{B} \Rightarrow \mathsf{A}.$$
$$\mathsf{C}(1) \wedge \mathsf{D} \wedge \mathsf{C}(2) \Rightarrow \mathsf{A}.$$

Le type des arbres de preuve pour A se définit inductivement en prenant un constructeur par clause[5] :

```
Inductive A : prop :=
  | clause1A : B → A
  | clause2A : (C 1) → D → (C 2) → A
```

[5] On distingue usuellement **Prop**, le type des propositions et **Set**, le type des types de données comme **nat**. Ceci est lié au mécanisme d'*extraction* que nous aborderons au chapitre suivant. Ici nous prenons prop $\stackrel{\text{déf}}{=\!=}$ **Set** au lieu de prop $\stackrel{\text{déf}}{=\!=}$ **Prop** car nous voulons effectuer des calculs sur les arbres de preuve.

Dans le cas général, les clauses comportent des variables. Prenons l'exemple de la concaténation. En première approximation, nous pouvons dire que pour chaque liste v, nous avons un constructeur Cnil_v qui fabrique l'arbre de preuve $\mathsf{conc}(\mathsf{nil}, v, v)$. De même, pour chaque quadruplet (x, u, v, w), nous avons une règle de construction

$$\frac{\mathsf{conc}(u, v, w)}{\mathsf{conc}(x.u, v, x.w)}$$

identifiée par un constructeur $\mathsf{Ccons}_{x,u,v,w}$. Plus formellement, la notation indicée représente une application et les constructeurs Cnil et Ccons sont en fait des noms donnés aux clauses (2.1) et (2.2).

```
Inductive conc : list → list → list → prop :=
  | Cnil :  ∀v:(listA)  (conc nil v v)
  | Ccons : ∀x:A  ∀uvw:(listA)
            (conc u v w) → (conc (cons x u) v (cons x w)).
```

On peut dès lors raisonner et calculer formellement sur les arbres ainsi formés de la même manière que sur des structures de données arborescentes : évaluer leur hauteur, etc.

On voit au passage que l'implication $A \Rightarrow B$ se traduit par le type d'un constructeur (plus généralement : d'une fonction) qui prend un arbre de preuve habitant A et rend un arbre de preuve habitant B. De même, une preuve de $\forall x{:}A\ B(x)$ est une fonction qui calcule à partir de tout habitant x de A un arbre de preuve de $B(x)$. Les propositions apparaissent comme des types dont les habitants sont des arbres de preuve : c'est l'isomorphisme de Curry-Howard. Un type comme $B(x)$, qui dépend de la valeur de x, est appelé type dépendant.

2.2.4 CCI comme logique

Il est facile de définir la conjonction et la disjonction au moyen d'un type inductif paramétré :

```
Inductive and [A,B :Prop] : Prop :=
  | and_intro : A → B → (and A B).
Inductive or [A,B :Prop] : Prop :=
  | or_intro_left : A → (or A B)
  | or_intro_right : B → (or A B).
```

Le seul connecteur logique primitif de CCI est $\rightarrow$, (identifié à $\Rightarrow$) ou plus exactement $\forall$, qui est la version dépendante de $\rightarrow$. En effet, $(\forall x : A)\, B(x)$ s'écrit $A \rightarrow B$ si B ne dépend pas de x. Ce connecteur est régi par les règles d'introduction et d'élimination usuelles.

$$\frac{\Gamma, x{:}A \vdash e : B(x)}{\Gamma \vdash (\lambda x{:}A)\, e : (\forall x{:}A)B(x)} \qquad \frac{\Gamma \vdash f : (\forall x{:}A)B(x) \quad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B(t)}$$

On a également une règle disant que si t est une preuve de A et que A s'évalue en A', alors t est aussi une preuve de A'. Dans le détail, il faut préciser les règles de formation des contextes et des types : le lecteur est renvoyé à [134].

Par rapport à Prolog, on notera que CCI est une logique d'ordre supérieur : on peut quantifier sur des fonctions, des prédicats et des types. Cette richesse est importante pour la spécification et la capacité à formaliser le raisonnement mathématique. Par ailleurs CCI est typé. Ce n'est pas le cas de Prolog, mais la plupart des programmes Prolog que l'on rencontre en pratique sont typables, comme ceux que nous considérons ici.

2.2.5 Quelques propriétés de CCI

Le calcul des constructions inductives a plusieurs propriétés utiles : le type d'une expression reste inchangé après réduction (*subject reduction*), étant donné une expression e et un type T, on peut déterminer mécaniquement si e a pour type T (le système de type est décidable) ; l'évaluation d'une expression se termine toujours (normalisation forte [137]). Les deux dernières propriétés sont liées : il peut être nécessaire de réduire T pour savoir si e est de type T. Pour assurer la normalisation forte, les fonctions définies récursivement doivent respecter une contrainte appropriée que nous avons indiquée. Cependant le λ-calcul lui-même est une source potentielle de calculs sans fin, c'est alors le typage qui garantit la terminaison des calculs. La normalisation forte garantit en particulier que tout objet clos dont le type est inductif se réduit effectivement en un arbre formé au moyen des seuls constructeurs de ce type. Ainsi, pour la cohérence logique de CCI, il suffit d'observer qu'il n'y a aucune preuve normale de l'absurde (type inductif ayant zéro constructeur) pour en déduire qu'il n'y a aucune preuve de l'absurde.

L'aspect calculatoire de CCI se traduit par le fait qu'il s'agit d'une logique constructive, dans la tradition intuitionniste. Ainsi, en réduisant une preuve de $A \vee \neg A$, on peut extraire soit une preuve de A soit une preuve de $\neg A$. Un habitant de $(\forall x{:}T)\, B(x) \vee \neg B(x)$ est donc une procédure de décision pour la propriété B. En particulier, affirmer le principe du tiers exclu aurait pour conséquence que toutes les propriétés des entiers sont décidables (du moins celles qui sont exprimables dans CCI, ce qui est bien trop).

2.3 La logique comme langage de programmation

La section 2.1 nous a présenté, pour résumer, une procédure de recherche automatique de démonstrations comme un calculateur universel dans lequel les programmes sont des formules logiques. Voilà qui paraissait prometteur en matière de méthodes formelles : si le programme, c'est la propriété, sa correction n'est plus à prouver !

2.3.1 Le slogan de Kowalski

Cette approche a des limitations :
- sur le principe, même s'il est garanti que toute solution respecte la relation exprimée par la requête, on peut s'intéresser à d'autres propriétés du résultat ;
- seule la correction partielle est traitée, rien n'est dit sur la terminaison ;
- sur le plan de l'expression logique :
  - toutes les formules logiques ne sont pas acceptables comme programmes, loin s'en faut ;
  - toute formule logique qui est une conséquence d'un programme logique donné n'est pas acceptable comme requête. Par exemple on ne peut demander si nil est élément neutre à droite de la concaténation définie par (2.1) et (2.2), ou si cette opération est associative.

Certaines restrictions sont de nature logique. Ainsi l'emploi des connecteurs est limité du fait de la forme clausale employée. Bien que toute formule puisse être mise sous cette forme, cette transformation peut altérer considérablement la lisibilité d'un programme en tant que spécification logique.

Les limitations liées au contrôle sont peut-être encore plus importantes du point de vue de programmeur. Ce dernier souhaite prévoir la terminaison et la complexité en temps et en espace de ses programmes. En l'occurrence, il s'agit d'appréhender l'exploration d'un espace de preuves. Cela n'est généralement pas trivial et a une influence directe sur le style de programmation. Remarquons d'ailleurs que la restriction aux clauses de Horn, qui n'est pas neutre sur le plan logique, trouve sa justification dans un problème de contrôle : la procédure originelle de Herbrand-Robinson traite des clauses générales, mais d'une part elle est beaucoup moins efficace que la SLD-résolution esquissée plus haut, d'autre part le sens opérationnel des programmes y est obscur.

Sur notre exemple, le programme clotrans, bien qu'exprimant clairement la notion de clôture transitive, a une fâcheuse tendance à explorer des chemins de longueur arbitrairement grande n'ayant aucune chance d'aboutir. Ceci vient de la stratégie SLD, en profondeur d'abord et de gauche à droite. Même si la recherche ne s'enlise pas dans une boucle infinie, il arrive fréquement que de nombreuses tentatives infructueuses soient effectuées avant qu'une bonne voie soit explorée. Pour remédier à ce problème, on peut songer à exprimer séparément le contrôle, c'est-à-dire le choix du littéral à résoudre ou de la clause à utiliser. C'est le slogan de R. Kowalski, emblèmatique des espoirs mis initialement dans la programmation en logique [72] :

$$\text{algorithme} = \text{logique} + \text{contrôle}.$$

Mais l'expression même d'un contrôle séparé devient rapidement délicate. Prolog a conservé une certaine simplicité en ne mettant en œuvre qu'un mécanisme de contrôle rudimentaire intégré dans les clauses, le coupe-choix.

2.3.2 Des propriétés opérationelles dans la logique

En fait il est souvent possible d'améliorer le comportement à l'exécution d'un programe logique en trouvant une formulation logiquement équivalente. Et ce n'est pas seulement une question de réordonner les clauses ou leurs littéraux. (Le « contrôle », au sens ci-dessus, revient souvent à effectuer un tel réordonnancement opportunément.) Il est préférable d'écrire la clôture transitive :

$$\forall x \quad \mathsf{arc}(x, y) \Rightarrow \mathsf{chemin}(x, y). \tag{2.11}$$

$$\forall x, y, z \quad \mathsf{arc}(x, y) \wedge \mathsf{chemin}(y, z) \Rightarrow \mathsf{chemin}(x, z). \tag{2.12}$$

Les combinaisons de concaténations sont également intéressantes à cet égard. Pour calculer la concaténation de trois listes, on s'aperçoit vite qu'en termes de temps de calcul (ou de taille d'arbre de preuve) il faut effectuer l'association à droite :

$$\forall u, v, w, l, r \quad \mathsf{conc}(v, w, l) \wedge \mathsf{conc}(u, l, r) \Rightarrow \mathsf{conc3d}(u, v, w, r). \tag{2.13}$$

plutôt qu'à gauche :

$$\forall u, v, w, l, r \quad \mathsf{conc}(u, v, l) \wedge \mathsf{conc}(l, w, r) \Rightarrow \mathsf{conc3g}(u, v, w, r). \tag{2.14}$$

Le point intéressant est qu'on peut sans inconvénient changer l'ordre des appels dans (2.13) :

$$\forall u, v, w, l, r \quad \mathsf{conc}(u, l, r) \wedge \mathsf{conc}(v, w, l) \Rightarrow \mathsf{conc3d}(u, v, w, r). \tag{2.15}$$

Par exemple, au cours de l'élaboration d'un arbre de preuve pour la formule $\mathsf{conc3d}(1.2.3.\mathsf{nil}, 4.5.\mathsf{nil}, 6.\mathsf{nil}, r)$, la résolution du premier littéral de (2.15) correspond à la requête (2.5), ce qui donne l'étape intermédiaire :

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{conc}(\mathsf{nil}, l, l)}{\mathsf{conc}(3.\mathsf{nil}, l, 3.l)}}{\mathsf{conc}(2.3.\mathsf{nil}, l, 2.3.l)}}{\mathsf{conc}(1.2.3.\mathsf{nil}, l, 1.2.3.l)} \quad \dfrac{?}{\mathsf{conc}(4.5.\mathsf{nil}, 6.\mathsf{nil}, l)}}{\mathsf{conc3d}(1.2.3.\mathsf{nil}, 4.5.\mathsf{nil}, 6.\mathsf{nil}, 1.2.3.l)}$$

La résolution du sous-but restant aboutit au résultat en emboîtant $l = 4.5.6.\mathsf{nil}$ dans le dernier argument.

Ces astuces sont bien connues parmi les programmeurs Prolog. Indiquons pour conclure que le slogan « algorithme = logique + contrôle » est complètement inefficace sur de tels exemples : l'expression logique a un fort impact sur les performances alors que l'ordre des littéraux n'en a aucun.

2.4 Raisonner sur des programmes Prolog

Le temps nécessaire pour construire un arbre de preuve dépend de sa taille ainsi que du nombre d'utilisations infructueuses du retour arrière, qui est un mécanisme puissant mais espiègle. Le programmeur est donc amené à se poser des questions telles que :

- la résolution d'un but termine-t-elle pour la stratégie SLD ? ou quand termine-t-elle ?
- combien de voies inutiles seront explorées avant la bonne ?
- combien y a-t-il d'arbres de preuve pour un but donné ?
- quelle est la taille des arbres de preuve correspondant à un but ?

Il s'agit là de *mesurer* un programme logique. Plus constructivement, il veut ensuite *l'améliorer* : comment faire en sorte que la résolution termine, comment diminuer l'utilisation du retour arrière, diminuer la taille des arbres de preuve, assurer la terminaison ? La formulation la plus naturelle, par exemple (2.14) pour conc3 et (2.7) pour clotrans, n'est pas toujours la plus efficace, ce qui conduit à des problèmes de *vérification* :

- comment s'assurer de l'équivalence de deux programmes ?

2.4.1 Prolog et CCI

Se placer dans le cadre du calcul des constructions inductives pour formaliser des raisonnements sur les programmes Prolog semble *a priori* une idée assez naturelle : dans les deux cas une spécification est une proposition et les notions d'arbre de preuve utilisées sont semblables.

Prolog et CCI diffèrent évidemment sur la richesse de la logique sous-jacente. C'est d'ailleurs parce qu'elle est suffisamment pauvre dans le cas de Prolog, que rechercher automatiquement des démonstrations y prend un sens calculatoire significatif : une proposition devient un programme de calcul d'arbres de preuve de cette proposition, alors qu'en CCI elle reste un type et que le calcul des arbres de preuve s'exprime explicitement, en programmation fonctionnelle.

Considérons par exemple un programme Prolog comportant pour seule clause $\mathsf{P} \Rightarrow \mathsf{P}$. La recherche d'une preuve de P boucle sans fin. Dans CCI, c'est à l'utilisateur d'essayer de construire un habitant de P, qui est le type défini inductivement par un unique constructeur $\mathsf{Shadok} : \mathsf{P} \to \mathsf{P}$. Naturellement, c'est impossible : P est vide. Cela se démontre dans CCI en définissant une fonction (d'ailleurs très concise) de P vers *False*.

Contrairement à CCI, Prolog n'offre aucun moyen de manipulation formelle d'arbres de preuve : le programmeur n'a accès qu'à leurs racines – et à d'autres objets arborescents comme $S(0).S(0).\mathsf{nil}$.

Notons une particularité importante des arbres de preuve construits par la procédure de résolution de Prolog : il sont toujours *normaux*.

Cependant, si l'on veut raisonner sur le processus de recherche, il faut

aussi considérer des arbres incomplets, notion qui n'a pas de sens en CCI proprement dit (même si on manipule de tels arbres en mode interactif sous Coq). On peut coder des arbres incomplets en introduisant des constructeurs ad-hoc ou en utilisant des λ-abstractions, mais les manipulations sont bien moins aisées. Dans la suite nous nous limiterons donc aux résultats de la procédure de résolution.

2.4.2 Mesures de complexité

Il n'est pas simple, dans le cas général, de quantifier le temps nécessaire à la construction d'un arbre de preuve, en tenant compte des échecs gérés par le retour arrière. Si on considère une clause

$$B(\mathbf{x}) \wedge C(\mathbf{y}) \wedge D(\mathbf{z}) \Rightarrow A(\mathbf{v}),$$

où $\mathbf{v}$, $\mathbf{x}$, $\mathbf{y}$ et $\mathbf{z}$ représentent des n-plets de variables non nécessairement disjoints, on peut avoir des situations où une substitution proposée par B pour $\mathbf{x}$ sera mise en échec par D, mais le retour arrière commencera par épuiser inutilement les possibilités de C. Des travaux ont été effectués pour analyser automatiquement le flot d'information et éviter ce genre de phénomène. Mesurer le temps de calcul dans ces conditions est du même ordre de difficulté. Le programmeur doit de toute façon vite apprendre à faire en sorte que cela ne se produise pas (ou presque pas).

Si on suppose que le contrôle (au sens du slogan de Kowalski) est au mains d'un oracle qui choisit parmi les clauses possibles celles qui mènent à un succès ; ou encore, si on suppose que le programmeur a écrit un programme « convenablement » déterministe, il reste de toute façon un travail inévitable : la construction de l'arbre de preuve. La taille de cet arbre est donc une mesure de la complexité en espace du programme considéré et une mesure *dans le meilleur cas* de sa complexité en temps[6].

Plus exactement, nous mesurons la complexité d'un programme Prolog par une fonction reliant la taille des arguments d'un prédicat à la taille d'un arbre de preuve pour ce prédicat. La taille d'un arbre est naturellement la somme des tailles de ses sous-arbres, plus un. Cela se formalise sans difficulté, notons simplement l'utilisation du filtrage sur des types dépendants (d'où les arguments anonymes de Ccons, qui représente le constructeur de listes noté de manière infixe « . » auparavant). On a par exemple :

```
Fixpoint taille [u,v,w :list ; C :(conc u v w)] : Z :=
  Cases C of
    (Cnil v) ⇒ '1'
  | (Ccons ____cu) ⇒ '(taille cu) + 1'
  end.
```

[6] Il est clair que si par exemple la complexité d'un programme Prolog est polynomiale en espace, elle sera au pire NP en temps, mais la portée pratique de ce constat est réduite.

On montre alors par récurrence structurelle, non sur les listes en argument, mais sur la formation des arbres de preuve :
Lemma long_taille :
(u,v,w :list) (C :(conc u v w)) '(taille C)=(long u) + 1'.

On montre tout aussi aisément que la taille d'un arbre de preuve pour (conc3g u v w r) est (long u) * 2 + (long v) + 3 tandis que pour (conc3d u v w r) elle est (long u) + (long v) + 3, ou encore que les programme de renversement suivants sont respectivement de complexité quadratique et linéaire.

$$\mathsf{renvnaif}(\mathsf{nil}, \mathsf{nil}). \tag{2.16}$$

$$\forall x, u, v, r \quad \mathsf{renvnaif}(u, v) \wedge \mathsf{conc}(v, x.\mathsf{nil}, r) \Rightarrow \mathsf{conc}(x.u, r). \tag{2.17}$$

$$\forall v \quad \mathsf{renv}(\mathsf{nil}, v, v). \tag{2.18}$$

$$\forall x, u, v, w \quad \mathsf{renv}(u, x.v, w) \Rightarrow \mathsf{renv}(x.u, v, w). \tag{2.19}$$

2.4.3 Preuves de propriétés de programmes

De même qu'en programmation traditionnelle, on est amené à comparer des programme logiques, par exemple renv avec renvnaif, clotrans avec chemin ou conc3g avec conc3d. Typiquement, étant donné deux prédicats $P_1(\mathbf{x})$ et $P_2(\mathbf{x})$, on souhaite démontrer

$$\forall \mathbf{x} \quad P_1(\mathbf{x}) \Rightarrow P_2(\mathbf{x}) \ \wedge \ P_2(\mathbf{x}) \Rightarrow P_1(\mathbf{x}). \tag{2.20}$$

D'autres propriétés peuvent être intéressantes à démontrer, afin d'augmenter la confiance que l'on accorde au programme. Par exemple, la transitivité de chemin ou encore l'associativité de conc, dont une expression simple est :

$$\forall u, v, w, r \quad \mathsf{conc3g}(u, v, w, r) \Rightarrow \mathsf{conc3d}(u, v, w, r) \wedge \tag{2.21}$$

$$\mathsf{conc3d}(u, v, w, r) \Rightarrow \mathsf{conc3g}(u, v, w, r) \tag{2.22}$$

En fait, même si on ne s'intéresse qu'à l'équivalence de programmes il est souvent nécessaire de s'appuyer sur des propriétés intrinsèques. Ainsi, on a besoin de la transitivité de chemin pour montrer que chemin et clotrans sont équivalents.

Comme dans les études de complexité, on procède par récurrence sur la formation des arbres de preuve : par l'isomorphisme de Curry-Howard, prouver (constructivement)

$$\forall \mathbf{x} \quad P_1(\mathbf{x}) \Rightarrow P_2(\mathbf{x})$$

revient à exhiber une fonction qui transforme un arbre de preuve pour $P_1(\mathbf{x})$ en un arbre de preuve pour $P_2(\mathbf{x})$. Cependant, ce sont là des principes de

récurrence relativement sophistiqués, dont la formulation exacte demande beaucoup de soin. L'aide d'un assistant comme Lego, Coq ou Alf est indispensable pour éviter les fausses manœuvres.

Remarquons que même dans le contexte de Prolog, l'ordre supérieur intervient facilement dans les raisonnements. Ainsi, la démonstration de l'équivalence entre chemin et clotrans ne dépend pas de la définition de arc, ce qui conduit à quantifier universellement ce prédicat. En fait les interpréteurs/compilateurs de Prolog sortent du cadre au premier ordre dans lequel ils sont théoriquement confinés car ils offrent généralement la possibilité de prendre une relation en paramètre, par exemple :

$$\forall A, x \quad A(x,y) \Rightarrow \mathsf{chemin}(A,x,y). \tag{2.23}$$

$$\forall A, x, y, z \quad A(x,y) \wedge \mathsf{chemin}(A,y,z) \Rightarrow \mathsf{chemin}(A,x,z). \tag{2.24}$$

La procédure de résolution continue à fonctionner de la même manière, à condition que A soit convenablement instancié au moment de l'appel. Le lemme de complétude de chemin par rapport à clotrans s'énonce alors :

Theorem cl_ch : (a :A→A→prop; x,y :A) (clotrans a x y) ⇒ (chemin a x y).

Sur un exemple aussi réduit, une simple récurrence sur $\mathsf{clotrans}(a,x,y)$ suffit. Pour des exemples à peine plus compliqués, on a besoin de raisonner par *inversion*, un principe qui indique les différentes possibilités effectives de démontrer $P_1(\mathbf{x})$ où P_1 est un prédicat défini inductivement. Par exemple une preuve de $\mathsf{conc}(x.u,v,r)$ ne peut être obtenue que par application de Ccons sur une preuve de $\mathsf{conc}(u,v,w)$, en imposant $r = x.w$. Cela semble « évident » et est utilisé constamment dans les argumentations informelles, par exemple pour justifier une propriété aussi utile que l'associativité de conc. Là encore, une formalisation précise peut être menée à bien avec le support d'un outil.

Une autre manière de procéder, qui convient bien lorsque l'un des arguments – généralement le dernier – est une fonction des autres, consiste précisément à expliciter cette fonction et à ramener les propriétés du prédicat à celles de la fonction. Par exemple :

Lemma conc_app : (u,v,w :list) (conc u v w) → w = (app u v).

Lemma app_conc : (u,v :list) (conc u v (app u v)).

L'associativité de app s'énonce et se démontre beaucoup plus facilement que celle de la relation conc. Il faut d'ailleurs remarquer que l'utilisation de fonctions calculées simplifie souvent les raisonnements et l'expression des théorèmes : en présence d'une relation fonctionnelle il est beaucoup plus agréable de désigner son résultat par $f(x)$ que de bavarder au sujet d'un unique y vérifiant $R(x,y)$. Ainsi, dans l'exemple du renversement, on introduit :

$$\mathsf{rvn} \stackrel{\text{déf}}{=} \lambda u\ \mathbf{case}\ u\ \mathbf{of}\ \mathsf{nil} \to \mathsf{nil} \mid x.l \to \mathsf{app}\ (\mathsf{rvn}\ l)\ x.\mathsf{nil} \tag{2.25}$$

$$\mathsf{rv} \stackrel{\text{déf}}{=} \lambda uv\ \mathbf{case}\ u\ \mathbf{of}\ \mathsf{nil} \to v \mid x.l \to \mathsf{rv}\ l\ x.v \tag{2.26}$$

ce qui permet de formaliser le lien entre la version naïve et la version efficace :

Theorem rvn_renvnaif : (u :list) (renvnaif u (rvn u)).

Theorem renvnaif_rvn : (u,v :list) (renvnaif u v) ⇒ v = (rvn u).

Theorem rv_renv : (u,v :list) (renv u v (rv u v)).

Theorem renv_rv : (u,v,w :list) (renv u v w) ⇒ w = (rv u v).

Theorem rv_rvn : (u,v :list) (rv u v) = (app (rvn u) v).

Pour résumer, CCI offre un cadre pour raisonner formellement sur des programmes Prolog. Leur vérification, de même que celle des programmes fonctionnels, bénéficie de notions claires pour les variables et les valeurs manipulées.

2.4.4 Application : recherche de démonstrations en Coq

Les techniques ci-dessus ont servi dans le cadre de la preuve de l'algorithme de conformité pour le protocole ABR. Sa formalisation dans Coq a conduit à démontrer des inégalités dans un ensemble muni d'une addition et d'une relation d'ordre total compatible avec celle-ci. On trouve typiquement des buts comme $\Gamma \vdash a < d$ où Γ contient un certain nombre d'inégalités, parmi lesquelles $a \leq b$, $c < d$, $b + \tau < u$ et $u \leq c + \tau$. Les hypothèses de transitivité et de régularité sont suffisantes, mais comme la preuve de correction de l'ABR requiert quelques dizaines de vérifications de cette nature, il est bien utile de les automatiser.

La tactique `EAuto` de Coq vient à point nommé : elle effectue une recherche d'arbres de preuve à la Prolog. On reconnaît ici le calcul de la fermeture transitive d'une relation : pour que la recherche soit efficace, il faut que la transitivité soit exprimée dans le style de chemin (2.12) et non de clotrans (2.7). Dans notre cas, les inégalités peuvent en outre être larges ou strictes, ce qui donne quatre formes de transitivité. Cependant on peut factoriser l'exploration. Il suffit, pour démontrer une inégalité large, de rechercher une chaîne d'inégalités quelconques et, pour démontrer une inégalité stricte, de rechercher une chaîne d'inégalités comportant une inégalité stricte. Deux formulations de la transitivité ont donc été considérées. La première, simple et convaincante, est exprimée dans (2.29) et dans les conséquences analogues sur $<$ et $\leq$ que l'on peut tirer des axiomes (2.27) à (2.32) de la figure 2.1. La seconde est adaptée à la recherche de preuves, dans le style de chemin. Elle est représentée par les clauses de la figure 2.2, qui doivent être comprises comme des définitions inductives (ainsi, $\preccurlyeq^*$ est la clôture transitive et réflexive de $\preccurlyeq$). Bien entendu on démontre d'une part l'équivalence de $\leq$ et $\preccurlyeq^*$, d'autre part celle de $<$ et $\prec^+$ [98].

La régularité de l'addition aurait pu être traitée de la même manière, en complétant les clauses de chemin. Je me suis contenté d'intégrer la régularité à la relation arc, en faisant en sorte que le cas le plus problématique soit

$$(\forall t : \texttt{DD}) \quad x \leq x. \tag{2.27}$$
$$(\forall t_1, t_2 : \texttt{DD}) \quad t_1 \leq t_2 \Rightarrow t_2 \leq t_1 \Rightarrow t_1 = t_2. \tag{2.28}$$
$$(\forall t_1, t_2, t_3 : \texttt{DD}) \quad t_1 \leq t_2 \Rightarrow t_2 \leq t_3 \Rightarrow t_1 \leq t_3. \tag{2.29}$$
$$(\forall t_1, t_2 : \texttt{DD}) \quad t_1 \leq t_2 \vee t_2 \leq t_1. \tag{2.30}$$
$$(\forall t_1, t_2 : \texttt{DD}) \quad t_1 = t_2 \vee \neg(t_1 = t_2). \tag{2.31}$$
$$< \stackrel{\text{déf}}{=} \lambda x, y : \texttt{DD}\ x \leq y \wedge \neg x = y \tag{2.32}$$
$$(\forall t, x, y : \texttt{DD}) \quad x \leq y \Rightarrow x + t \leq y + t. \tag{2.33}$$
$$(\forall x, t, s : \texttt{DD}) \quad t \leq s \Rightarrow x + t \leq x + s. \tag{2.34}$$
$$(\forall t, x : \texttt{DD}) \quad (x + t) - t = x. \tag{2.35}$$
$$(\forall t, x : \texttt{DD}) \quad x \leq (x - t) + t. \tag{2.36}$$
$$(\forall t, x, y : \texttt{DD}) \quad x \leq y \Rightarrow x - t \leq y - t. \tag{2.37}$$
$$(\forall t, s, x : \texttt{DD}) \quad s \leq t \Rightarrow x - t \leq x - s. \tag{2.38}$$
$$(\forall x, y, z, t : \texttt{DD}) \quad z + t \leq y \Rightarrow x \leq y - t \Rightarrow x + t \leq y. \tag{2.39}$$

Dans (2.39), z est nécessaire car on ne suppose pas que `DD` est muni d'un zéro. Cet axiome peut, en présence des autres, être remplacé par :

$$(\forall t, x, z : \texttt{DD}) \quad z + t \leq x \Rightarrow x = (x - t) + t. \tag{2.40}$$

FIG. 2.1 – Axiomes pour les dates et les durées.

considéré en dernier (celui où, pour démontrer que x est plus petit que y, on cherche un c tel que $x + c$ est plus petit que $y + c$).

En pratique, cette stratégie a fonctionné. Le confinement de l'espace de recherche pour les preuves s'observe aux nombreuses invocations, dans les scripts, de `EAuto` avec un argument plus grand que 10 ou 20 – cet argument, qui vaut 5 par défaut, indique la profondeur maximale de la recherche. Par ailleurs la réponse vient immédiatement ou très rapidement, en tout cas beaucoup plus rapidement qu'en utilisant les axiomes de la figure 2.1. Par exemple, on gagne un facteur supérieur à vingt pour démontrer $a < d$ à partir de $a \leq b$, $c < d$, $b + t < u$ de $u \leq c + t$.

Notons que la façon normale de procéder aurait été d'écrire une procédure de décision ou de semi-décision pour cette théorie, qui est à la fois plus faible que l'arithmétique de Presburger (pas de 0, de 1 ni de successeur) et plus faible que l'arithmétique réelle (rien n'est dit sur la continuité). Mais cela aurait été bien plus coûteux que les quelques dizaines de lignes de Coq mises en œuvre ici. En outre, dans notre cas de figure, la taille des buts reste

$$\preccurlyeq_{\texttt{false}} \stackrel{\text{déf}}{=\!=} \leq . \tag{2.41}$$

$$\preccurlyeq_{\texttt{true}} \stackrel{\text{déf}}{=\!=} < . \tag{2.42}$$

$$(\forall b : \texttt{bool})(\forall x, y : \texttt{DD}) \quad x \preccurlyeq_b y \Rightarrow x \preccurlyeq^+ y. \tag{2.43}$$

$$(\forall b : \texttt{bool})(\forall x, y, z : \texttt{DD}) \quad y \preccurlyeq_b z \Rightarrow x \preccurlyeq^+ y \Rightarrow x \preccurlyeq^+ z. \tag{2.44}$$

$$(\forall x : \texttt{DD}) \quad x \preccurlyeq^* x. \tag{2.45}$$

$$(\forall x, y : \texttt{DD}) \quad x \preccurlyeq^+ y \Rightarrow x \preccurlyeq^* y. \tag{2.46}$$

$$(\forall x, y : \texttt{DD}) \quad x \preccurlyeq_{\texttt{true}} y \Rightarrow x \prec^+ y. \tag{2.47}$$

$$(\forall b : \texttt{bool})(\forall x, y, z : \texttt{DD}) \quad y \prec_b z \Rightarrow x \prec'_b y \Rightarrow x \prec^+ z. \tag{2.48}$$

$$(\forall x, y : \texttt{DD}) \quad x \preccurlyeq^* y \Rightarrow x \prec'_{\texttt{true}} y. \tag{2.49}$$

$$(\forall x, y : \texttt{DD}) \quad x \prec^+ y \Rightarrow x \prec'_{\texttt{false}} y. \tag{2.50}$$

FIG. 2.2 – Traitement optimisé de la transitivité.

modeste.

2.5 De la logique linéaire dans Prolog

Nous avons vu qu'il est possible de raisonner formellement sur des programmes Prolog grâce à des notions claires de variables et de valeurs, qui ne sont pas perturbées par des références ou des pointeurs.

Il reste cependant à examiner les techniques de programmation les plus typiques de Prolog, celles qui sont basées sur des structures qui se complètent progressivement par substitution au cours d'une résolution. Les plus connues sont les « listes différentielles » ou d-listes. Par exemple, le couple $\langle u, 1.2.3.u\rangle$ est une d-liste qui représente la liste 1.2.3.nil.

Les explications qui en sont données dans la littérature sur la programmation en logique ne nous ont jamais paru très satisfaisantes, allant d'une présentation intuitive à base de structures incomplètes munies d'un pointeur vers la partie à compléter [35] à de simples indications pragmatiques [132].

Un premier point est de démontrer qu'un programme avec d-listes est équivalent au programme avec listes ordinaires correspondant. Nous avons vérifié que les techniques précédentes s'appliquent encore. Mais cela ne suffit pas à comprendre les fondements de cette technique et à la généraliser. Dans [90, 92] nous avons suggéré que ces structures pouvaient être considérées comme une forme dégénérée de λ-expressions, soumises à des contraintes relevant de la logique linéaire. Nous revenons ici sur les idées principales de ce

travail et développons quelques points, essentiellement sur des exemples. Un traitement plus exhaustif serait nécessaire, il sera laissé dans les perspectives.

2.5.1 Structures incomplètes et λ-termes

Dans la définition de conc3d en (2.15), le couple constitué de l et r au premier appel de conc peut être vu comme une d-liste. Un second exemple est en (2.18) et (2.19) pour le programme renv. Une manière de l'analyser est de regarder le deuxième argument comme un accumulateur, mais on peut aussi considérer que renv produit un couple $\langle v, w\rangle$. Intuitivement, le second élément du couple représente une liste incomplète tandis que le premier élément indique la partie manquante, ce qui apparaît mieux sur la formulation suivante.

$$\forall v \quad \mathsf{renv}(\mathsf{nil}, v, v). \tag{2.51}$$

$$\forall x, u, v, w \quad \mathsf{renv}(u, v_2, w) \wedge v_2 = x v_1 \Rightarrow \mathsf{renv}(x u, v_1, w). \tag{2.52}$$

Voici un exemple plus substantiel extrait de ma thèse. Rappelons tout d'abord qu'étant donné un langage hors-contexte, il lui correspond en Prolog un analyseur descendant récursif où chaque non-terminal U est traduit par une relation entre deux listes de symbole lexicaux et où chaque règle est traduite par une clause exprimant une relation de Chasles, suivant le modèle suivant (les U_i sont des non-terminaux et les t_j sont des terminaux).

$$U \quad ::= \quad U_1\ \mathsf{t}_2\ U_3\ U_4\ \mathsf{t}_5$$

se traduit par :

$$\begin{aligned}&\forall l_0, l_1, l_2, l_3, l_4, l_5\\ &\qquad U_1(l_0, l_1) \wedge \mathsf{consomme}(\mathsf{t}_1, l_1, l_2) \wedge\\ &\qquad U_3(l_2, l_3) \wedge U_4(l_3, l_4) \wedge \mathsf{consomme}(\mathsf{t}_5, l_4, l_5) \quad \Rightarrow \quad U(l_0, l_5)\end{aligned}$$

Il y a autant de clauses pour U que d'alternatives dans la grammaire considérée. Chacune exprime la consommation d'une suite de symboles lexicaux reconnus par le non-terminal U au début de l_0. Le prédicat consomme est défini par :

$$\forall l, t \quad \mathsf{consomme}(t, tl, l) \tag{2.53}$$

On peut ensuite considérer des grammaires plus élaborées, dans lesquelles les non-terminaux comportent des arguments logiques supplémentaires. Cette technique a été inventée par Alain Colmerauer sous le nom de *grammaires de métamorphoses* dès l'origine de Prolog. Dans ce qui suit nous identifions règles et clauses : nous omettons les deux arguments supplémentaires, la consommation d'un symbole lexical est spécifiée par la mise

entre quotes de ce dernier, et celle d'une chaîne vide par ε. Pour alléger, les quantifications universelles sont omises, conformément à l'usage.

Introduisons un type concret arexp des expressions arithmétiques, avec les opérateurs binaires moins, mult, puiss et pour terminer simplement une constante C. Le programme Prolog suivant parcourt une chaîne de symboles en construisant un habitant de arexp. Nous verrons plus loin l'explication des arguments x, y, z, etc.

$$\text{'C'} \quad \mathsf{ke}(\mathsf{C}, x, x, y, y, e) \Rightarrow \mathsf{expr}(e) \tag{2.54}$$

$$\text{'\^{}'} \quad \text{'C'} \quad \mathsf{ke}(\mathsf{puiss}(x_1, \mathsf{C}), x_2, y_1, y_2, z_1, z_2) \Rightarrow \mathsf{ke}(x_1, x_2, y_1, y_2, z_1, z_2) \tag{2.55}$$

$$\text{'*'} \quad \text{'C'} \quad \mathsf{ke}(\mathsf{C}, x_2, \mathsf{mult}(y_1, x_2), y_2, z_1, z_2) \Rightarrow \mathsf{ke}(x, x, y_1, y_2, z_1, z_2) \tag{2.56}$$

$$\text{'-'} \quad \text{'C'} \quad \mathsf{ke}(\mathsf{C}, x_2, x_2, y_2, \mathsf{moins}(z_1, y_2), z_2) \Rightarrow \mathsf{ke}(x, x, y, y, z_1, z_2) \tag{2.57}$$

$$\varepsilon \quad \Rightarrow \mathsf{ke}(x, x, y, y, z, z) \tag{2.58}$$

Les points à remarquer sont les suivants :

- le parcours de la chaîne est déterministe, étant piloté par la lecture des symboles d'opérations arithmétiques ;
- il n'y a pas de problème de récursion gauche ;
- il n'y a pas non plus de prédicats correspondant à des non-terminaux intermédiaires.

Et pourtant, la grammaire analysée est bien celle que l'on attend, avec trois niveaux de précédence et des opérations associant à gauche. Intuitivement, ce programme prépare trois niveaux d'emboîtements qui sont composés convenablement, de sorte qu'à l'exécution l'unification fait tout le travail. Chaque niveau est représenté par un couple d'arguments, par exemple $\langle x, \mathsf{puiss}(...\mathsf{puiss}(x, \mathsf{C}), ..., \mathsf{C})\rangle$ pour le premier niveau. Il est plus clair d'exprimer ces emboîtements avec des λ-expressions :

$$\text{'C'} \quad \mathsf{ke}(k_x,\, k_y,\, k_z) \Rightarrow \mathsf{expr}(k_z(k_y(k_x(\mathsf{C})))) \tag{2.59}$$

$$\text{'\^{}'} \quad \text{'C'} \quad \mathsf{ke}(k_x,\, k_y,\, k_z) \Rightarrow \mathsf{ke}(\lambda x\, k_x(\mathsf{puiss}(x, \mathsf{C})),\, k_y,\, k_z) \tag{2.60}$$

$$\text{'*'} \quad \text{'C'} \quad \mathsf{ke}(k_x,\, k_y,\, k_z) \Rightarrow \mathsf{ke}(\lambda x\, x,\, \lambda y\, k_y(\mathsf{mult}(y, k_x(\mathsf{C}))),\, k_z) \tag{2.61}$$

$$\text{'-'} \quad \text{'C'} \quad \mathsf{ke}(k_x,\, k_y,\, k_z) \Rightarrow \mathsf{ke}(\lambda x\, x,\, \lambda y\, y,\, \lambda z\, k_z(\mathsf{moins}(z, k_y(k_x(\mathsf{C}))))) \tag{2.62}$$

$$\varepsilon \quad \Rightarrow \mathsf{ke}(\lambda x\, x,\, \lambda y\, y,\, \lambda z\, z) \tag{2.63}$$

Le programme ke rend trois fonctions indiquant respectivement à quelles puissances mettre, par quoi multiplier successivement et que retrancher successivement à une valeur donnée. On peut vérifier que tout arbre syntaxique en argument de la racine d'un arbre de preuve de (2.59) correspond à la grammaire désirée.

2.5.2 Suppression des λ

Bien entendu, la recherche d'arbres de preuve pour les clauses (2.59) à (2.63), qui ne sont pas au premier ordre, est en dehors des capacités de

Prolog. Une idée serait de passer à λ-Prolog [81, 106], une possibilité apparue plus récemment que Véda– les premières mises en œuvre efficaces [27] datent seulement des années 90. Notre propos, dans [90, 92], n'était cependant pas de changer de langage de programmation, mais de définir un cadre conceptuel pour mieux comprendre et dériver des programmes efficaces écrits en Prolog ordinaire. Même ainsi, nous allons voir que λ-Prolog est à la fois trop puissant et trop grossier, contrairement au fragment sans exponentielles de la logique linéaire.

Dans [90], le programme Prolog légal (2.54) - (2.58) est dérivé à partir de la formulation fonctionnelle (2.59) - (2.63) de la manière suivante :

- dans chaque clause, chaque variable k de type $A \to B$ est remplacée par un couple de variables fraîches $\langle x_1, x_2 \rangle$;
- de même, les expressions $\lambda x\ t$ sont remplacées par des couples $\langle x, t \rangle$;
- les β-réductions sont effectuées d'avance : typiquement, $\langle x_1, x_2 \rangle(t)$ est remplacé par x_2 en propageant l'égalité $x_1 = t$ dans le reste de la clause.

En appliquant ce procédé sur (2.61) on obtient :

$$\text{'*'}\quad \text{'C'}\quad \mathsf{ke}(\langle \mathsf{C}, x_2 \rangle, \langle \mathsf{mult}(y, x_2), y_2 \rangle, \langle z_1, z_2 \rangle) \Rightarrow \mathsf{ke}(\langle x, x \rangle, \langle y, y_2 \rangle, \langle z_1, z_2 \rangle).$$

Pour terminer il reste à retirer uniformément les constructeurs de couples, ce qui ne change rien d'essentiel. On peut vérifier formellement (voir annexe A) que les arbres de preuve obtenus par (2.59) - (2.63) démontrent les mêmes formules $\mathsf{expr}(e)$ que (2.54) - (2.58) . Pour reprendre un petit exemple bien connu, le programme de renversement efficace

$$\mathsf{renv}(\mathsf{nil}, v, v) \tag{2.64}$$

$$\mathsf{renv}(u, x.v, w) \Rightarrow \mathsf{renv}(x.u, v, w) \tag{2.65}$$

s'obtient de la même façon à partir de l'écriture suivante :

$$\mathsf{renv}(\mathsf{nil}, \lambda v\, v). \tag{2.66}$$

$$\mathsf{renv}(u, f) \Rightarrow \mathsf{renv}(x.u, \lambda v\, (f\, x.v)). \tag{2.67}$$

En invoquant ce programme avec pour premier argument une liste (de longueur) close u, on obtient la fonction qui concatène son argument v au renversement de u. Le programme fonctionnel correspondant serait une variante de (2.26) qui n'attend pas que v soit connu pour parcourir u :

$$\mathsf{rv} \stackrel{\text{déf}}{=} \lambda u\ \mathbf{case}\, u\, \mathbf{of}\, \mathsf{nil} \to \lambda v\, v \mid x.l \to \mathbf{let}\, f = \mathsf{rv}\ l\, \mathbf{in}\, \lambda v\, (f\ x.v) \tag{2.68}$$

2.5.3 Contraintes de linéarité

J'ai indiqué dans [90] que cette traduction n'est possible que pour des fonctions linéaires, c'est-à-dire ayant un type exprimable dans le fragment

sans exponentielles de la logique linéaire [52] (voir l'annexe B). C'est en particulier le cas si dans chaque clause les variables de type $A \to B$ (ou plutôt $A \multimap B$ désormais) ont exactement deux occurrences, dont une en position de simple argument dans un littéral de queue et l'autre, éventuellement appliquée à un argument, en tête de clause. Ainsi, le programme qui concatène v et w au renversement de u sera dérivé à partir de (2.70) et non de (2.69).

$$\mathsf{renv}(u, f) \Rightarrow \mathsf{renv2nok}(u, v, w, f(v), f(w)). \tag{2.69}$$

$$\mathsf{renv}(u, f_1) \wedge \mathsf{renv}(u, f_2) \Rightarrow \mathsf{renv2}(u, v, w, f_1(v), f_2(w)). \tag{2.70}$$

2.5.4 Application : construction de programmes par morphisme

On peut donc concevoir des programmes Prolog en utilisant des fonctions à condition de respecter les contraintes de linéarité. Une autre manière de voir les d-listes est alors de s'appuyer sur le morphisme entre les listes munies de la concaténation et les fonctions munies de la composition.

Definition phi := [l :list] (eta (app l)).

Lemma phi_morph : (u,v :list) (phi (app u v)) = (comp (phi u) (phi v)).

Le lemme clef est une variante (plus forte) de l'associativité de app :

Lemma fonc_assoc :
(u,v :list) [w :list](app u (app v w))=[w :list](app (app u v) w).

On peut alors écrire un programme qui utilise la concaténation sur les listes sans se soucier du manque d'efficacité, puis obtenir une version efficace en passant au morphisme. Le résultat final s'obtient en appliquant la fonction construite à nil. Ce procédé de construction s'applique également aux programmes fonctionnels.

2.5.5 Perspectives

Il conviendrait de justifier les intuitions exposées ci-dessus. Voici une piste possible. Considérons une relation $R(\langle x, y\rangle)$ définie par un ensemble de clauses, de telle sorte que $R(\langle x, y_1\rangle)$ et $R(\langle x, y_2\rangle)$ entraîne toujours $y_1 = y_2$. Considérons ensuite un littéral $R(f)$ en queue d'une clause contenant par ailleurs une occurrence de $f(t)$. Lorsque ce littéral devient un sous-but au cours d'une exécution, tous les couples $\langle x, y\rangle$ dénotés par la relation R sont potentiellement disponibles. Ainsi, dans tous les arbres de preuve construits avec succès, t et $f(t)$ satisfont $R(\langle t, f(t)\rangle)$. Mais ces couples ne sont pas disponibles tous ensemble : si R est définie par une énumération de clauses $R(\langle x_i, y_i\rangle)$ où x_i et y_i sont des termes clos, seul l'un des couples $\langle x_i, y_i\rangle$ est utilisé dans l'arbre de preuve à l'occurrence considérée. On reconnaît le comportement de la conjonction additive &.

Cette idée prend son véritable intérêt lorsque R est décrite par une clause quantifiée universellement[7], comme

$$\forall x \quad R(\langle x, 1.2.3.x\rangle) \tag{2.71}$$

pour prendre un exemple simple : une infinité de couples sont disponibles et surtout, la sélection se fait non par retour arrière mais par unification de x à l'argument (t) auquel cette fonction linéaire est appliquée.

On peut combiner l'effet de & et de $\forall$. Par exemple pour le renversement, en employant une syntaxe par filtrage[8] :

$$\mathsf{renv}(\lambda^{\circ}\mathsf{nil}\,\lambda^{\circ}v\,v). \tag{2.72}$$

$$\mathsf{renv}(g) \Rightarrow \mathsf{renv}(\lambda^{\circ}x.u\,\lambda^{\circ}v\,(g\,u\,x.v)). \tag{2.73}$$

En assimilant le type de nil à $\mathbf{1}$ et celui des listes de la forme $x.u$ où x est de type A et u de type list, à $A\otimes$list, on a list $= \mathbf{1}\oplus A\otimes$list. On peut alors proposer les typages suivants, en identifiant $\mathbf{1}\multimap Q \,\&\, A\otimes\mathsf{list}\multimap Q$ avec $\mathbf{1}\oplus A\otimes\mathsf{list}\multimap Q$ (la syntaxe est décrite à l'annexe B) :

$$\dfrac{\dfrac{\dfrac{\mathsf{nil}:\mathbf{1};\ v:\mathsf{list}\vdash v:\mathsf{list}}{\mathsf{nil}:\mathbf{1}\vdash\lambda^{\circ}v\,v:\mathsf{list}\multimap\mathsf{list}}}{\vdash\lambda^{\circ}\mathsf{nil}\,\lambda^{\circ}v\,v:\mathbf{1}\multimap\mathsf{list}\multimap\mathsf{list}} \quad \dfrac{\dfrac{\vdash g:\mathsf{list}\multimap\mathsf{list}\multimap\mathsf{list}}{\vdots}}{\vdash\lambda^{\circ}x.u\,\lambda^{\circ}v\,(g\,u\,x.v):A\otimes\mathsf{list}\multimap\mathsf{list}\multimap\mathsf{list}}}{\vdash\lambda^{\circ}\mathsf{nil}\,\lambda^{\circ}v\,v\,[]\,\lambda^{\circ}x.u\,\lambda^{\circ}v\,(g\,u\,x.v):\mathsf{list}\multimap\mathsf{list}\multimap\mathsf{list}}$$

Le sous-arbre représenté par les points de suspension se détaille comme suit :

$$\dfrac{\dfrac{\dfrac{\vdash g:\mathsf{list}\multimap\mathsf{list}\multimap\mathsf{list}\quad u:\mathsf{list}\vdash u:\mathsf{list}}{u:\mathsf{list}\vdash g\,u:\mathsf{list}\multimap\mathsf{list}} \quad \dfrac{x:A\vdash x:A\quad v:\mathsf{list}\vdash v:\mathsf{list}}{x:A;v:\mathsf{list}\vdash x.v:\mathsf{list}}}{x:A;u:\mathsf{list};v:\mathsf{list}\vdash(g\,u\,x.v):\mathsf{list}}}{x.u:A\otimes\mathsf{list};v:\mathsf{list}\vdash(g\,u\,x.v):\mathsf{list}}}{x.u:A\otimes\mathsf{list}\vdash\lambda^{\circ}v\,(g\,u\,x.v):\mathsf{list}\multimap\mathsf{list}}}{\vdash\lambda^{\circ}x.u\,\lambda^{\circ}v\,(g\,u\,x.v):A\otimes\mathsf{list}\multimap\mathsf{list}\multimap\mathsf{list}}$$

L'objet rendu par renv s'interprète alors comme un arbre de preuve infini pour une formule de logique linéaire. Une question ouverte $\mathsf{renv}(g)$ conduit à l'énumération d'une infinité de réponses. En revanche si g est préalablement appliquée à une liste close u (ou simplement de longueur close) le mécanisme de filtrage de Prolog et la propagation des substitutions se traduisent par un processus d'élimination des coupures qui guide l'exécution le long d'une branche finie, tout en préparant une liste r comprenant les éléments de u dans l'ordre inverse ; la réponse est alors une fonction linéaire « uniforme », qui à l'appel place immédiatement son argument à la fin de r.

[7] Rappelons qu'en logique linéaire, $\forall$ généralise la conjonction additive.

[8] Nous autorisons des expressions de la forme λ<motif>E, où E peut contenir des variables libres apparaissant dans <motif>. Une fonction ainsi définie ne peut être appliquée qu'à des arguments respectant le motif.

Il est difficile de reproduire exactement ce comportement dans un langage fonctionnel à évaluation stricte. Le programme (2.68), qui est assez proche, réduit préalablement les analyses par cas, mais au lieu de préparer une clôture $\lambda x\, r_1. \ldots r_n.x$ (qui à l'appel commence de toute façon par recopier n exemplaires de cons), il construit une suite de clôtures qu'il faudra évaluer une à une : ce mécanisme est plus général mais moins efficace dans le cas dégénéré où les fonctions sont destinées à être appliquées une seule fois.

Avant de terminer, il convient de noter que la logique linéaire d'origine possède la propriété de normalisation forte. Si nous voulons rendre compte des comportements infinis de la même manière que ci-dessus, nous avons besoin d'une extension appropriée, qui pourrait d'ailleurs compenser l'absence d'exponentielles. Cela est laissé pour étude ultérieure.

2.6 Conclusion

Natarajan Shankar m'a récemment indiqué que les optimisations génériques effectuées dans les implémentations actuelles de Prolog donnent de bons résultats dans l'implantation de différents algorithmes de démonstration automatique. Il suffit d'exprimer les idées essentielles de ces derniers, le code obtenu semble très compétitif avec celui qui est issu d'une rédaction directe en langage procédural [130]. Ces bénéfices se retrouveraient-ils dans le domaine de la compilation ? Pour Véda, il paraissait en tout cas indispensable de bien comprendre les différences opérationnelles entre des formulations logiquement équivalentes, en particulier de limiter autant que possible le non-déterminisme à l'exécution. Il nous a fallu tout d'abord raisonner sur la dénotation des programmes. Le calcul des constructions inductives aurait été un outil adapté à la formalisation de tels raisonnements. Pour analyser et concevoir des programmes efficaces, nous nous sommes par ailleurs servis de la logique linéaire. Notre objectif pratique, la réalisation du compilateur Véda, a ainsi été atteint.

Cet exemple n'a pas été suivi à ma connaissance. De fait, si l'on a besoin d'écrire un analyseur syntaxique efficace dans un esprit analogue – programme proche de la grammaire, manipulation aisée d'arbres – et si de surcroît on a éliminé le non-déterminisme, le choix de Caml, qui de plus est typé, semble actuellement s'imposer.

Pour revenir à notre emploi de la logique linéaire, notons qu'il se distingue de celui qui est relaté dans d'autres travaux comme [82, 31]. Dans ceux-ci il s'agit de proposer au programmeur des concepts qu'il manipulera à travers de nouvelles constructions syntaxiques intégrées à un langage de programmation tel que λ-Prolog ou Lolli. Notre cible reste ici la version ordinaire de Prolog. On peut considérer qu'un calcul fonctionnel issu de la logique linéaire est *déjà* présent dans la procédure de résolution. Sommairement, une conjonction de clauses forme des objets habitant un type conjonctif additif tandis que la quantification universelle construit des fonctions calculant par unification.

Chapitre 3

Extraction de programmes avec exceptions

Il est assez simple d'exprimer ce qu'est la correction d'un programme purement fonctionnel par rapport à une spécification logique. On se donne
- la définition d'une fonction f,
- un prédicat $P(x_1, ...)$ sur les entrées $x_1, ...$ (la précondition),
- une relation $R(x_1, ..., y)$ entre les entrées et la sortie attendue

et il s'agit de démontrer

$$\forall x_1, ... \quad P(x_1, ...) \Rightarrow R(x_1, ..., f(x_1, ...)).$$

Il s'agit là de correction partielle, oublions momentanément les questions de terminaison. L'*extraction de programmes* est une technique qui permet d'extraire une telle fonction f à partir d'une démonstration constructive de

$$\forall x_1, ... \quad P(x_1, ...) \Rightarrow \exists y R(x_1, ..., y).$$

Dans un cadre adéquat, la terminaison de $f(a_1...)$ est en outre assurée lorsque la précondition $P(a_1...)$ est satisfaite.

Les programmes purement fonctionnels ont deux qualités particulièrement agréables :
- nous avons la notion de valeur immuable, habituelle en mathématiques ; nous ne sommes pas perturbés par les changements potentiels ou effectifs du contenu de cases mémoire ;
- en particulier, la valeur d'une expression ne dépend que de la valeur de ses sous-expressions – l'ordre d'évaluation n'intervient pas.

Ainsi le raisonnement mathématique usuel s'applique directement, on économise toutes sortes de complications rencontrées dans les programmes impératifs et cela se retrouve dans la formalisation des démonstrations.

Dans la réalité, on ne s'en débarrasse pas aussi facilement : les vrais programmes ont de temps à autre un effet sur un état global ou sur les entrées/sorties. Certains algorithmes (tri en place par exemple) n'ont de sens

que dans le cadre de la programmation impérative. On peut cependant modéliser ces aspects en utilisant la notion de monade, disponible dans le langage purement fonctionnel Haskell [136]. Nous renvoyons le lecteur à [47] pour une extension récente de cette approche à des langages de la famille de ML. Si l'environnement visé convient à l'exécution de programmes fonctionnels, il est bien entendu possible et même recommandé de confiner l'utilisation des traits impératifs, de façon à limiter les problèmes afférents. Dans le cas de ML, il reste cependant un trait fondamental qui lui a été intégré dès sa conception : les exceptions. Elles portent d'ailleurs mal leur nom : elles sont utilisées aussi bien pour signaler une anomalie – exceptionnelle ! – que pour rendre un résultat ordinaire.

L'effet de la levée d'une exception s'explique aisément en termes opérationnels : le contrôle passe directement au gestionnaire d'exception (**try** ... **with** ... en Ocaml, **handle** en syntaxe SML) et la pile d'exécution est vidée en cohérence. Cependant, justifier un programme fonctionnel avec exceptions en termes de compteur ordinal et de pile d'exécution serait impraticable. On perdrait d'emblée le bénéfice des qualités évoquées ci-dessus pour la programmation fonctionnelle, y compris, vraisemblablement, pour les parties d'un programme non concernées par une exception.

Heureusement, il est possible de travailler sur une représentation purement fonctionnelle mettant en œuvre la notion de *continuation*. Les continuations ont en fait été inventées pour formaliser les sauts de contrôle des langages impératifs – le fameux **goto** – dans le cadre de la sémantique dénotationnelle [133]. Elles ont par la suite été employées pour définir des opérateurs de contrôle plus exotiques : retour arrière, pseudo-parallélisme,... (Felleisen, Filinsky, Danvy).

Les continuations posent quelques difficultés de typage. Cependant, le mécanisme d'exception n'offre pas toute la puissance des continuations : tout programme avec exception se traduit à l'aide de continuations, mais pas l'inverse. Nous verrons comment démontrer formellement des propriétés de correction dans un cadre fortement typé, celui du calcul des constructions inductives. D'un point de vue pratique, nous nous placerons dans le cadre de l'extraction de programme définie dans ce calcul, ce qui règle incidemment la question de la terminaison.

3.1 L'extraction de programmes

3.1.1 Idée générale

Examinons à nouveau la spécification donnée plus haut pour introduire l'extraction de programmes.

$$\forall x_1, \ldots \quad P(x_1, \ldots) \Rightarrow \exists y R(x_1, \ldots, y). \tag{3.1}$$

D'après l'isomorphisme de Curry-Howard, une démonstration de (3.1) n'est rien d'autre que la définition d'une fonction F prenant pour arguments $x_1...$ avec un argument supplémentaire ρ_P qui est une preuve de $P(x_1, ...)$, et qui rend une valeur y avec une preuve ρ_R que y satisfait la post-condition désirée.

On exige bien entendu que la valeur de y ne dépende pas de ρ_P. Cela est assuré en Coq par la distinction entre les deux univers **Prop** et **Set**. Étant donné des arguments $a_1...$ et un argument α pour la précondition, $F(a_1, ...\alpha)$ se réduit donc à un couple de la forme $\langle F_y(a_1, ...), F_R(a_1, ...\alpha)\rangle$.

Le premier élément $F_y(a_1, ...)$ dénote la valeur recherchée, tandis que le second $F_R(a_1, ...\alpha)$ est une preuve que la postcondition est satisfaite. C'est du point de vue de l'évaluation que ces deux composantes ont un statut différent : le propos du programme est juste de réduire la première, inutile de normaliser la seconde. On peut donc voir cette dernière comme du code mort.

En résumé, la fonction désirée $f = F_y$ est obtenue en effaçant de F les termes de preuve comme ρ_P ou ρ_R. Si la démonstration de (3.1) est effectuée dans un cadre où les fonctions ont la propriété de normalisation forte, la terminaison de f est en outre assurée lorsqu'elle est employée sous la précondition P.

L'outil technique qui permet de définir et formaliser complètement l'extraction est la notion de *réalisateur*, qui remonte à Kleene – ses utilisations initiales étaient sans rapport avec la programmation. L'extraction de programme utilisée dans Coq est issue des travaux de C. Paulin [111, 110].

3.1.2 L'extraction en pratique

Le travail de l'utilisateur consiste donc à démontrer un but tel que (3.1). Il peut procéder de plusieurs manières. S'il connaît d'avance la fonction f, il peut la proposer d'emblée au moyen de la tactique `Realizer`. L'outil calcule alors les obligations de preuve qui doivent être démontrées. Généralement l'utilisateur doit annoter f par des variants et des invariants supplémentaires.

Dans une démarche synthétique, calculatoire au sens de Dijkstra, l'utilisateur peut aussi découvrir (ou faire mine de découvrir) f, en se servant du mode interactif comme d'un éditeur de programme : `Intro` construit une λ-abstraction, `Induction` correspond à un opérateur de récursion, etc.

Mon travail sur l'extraction de programmes avec exceptions [93, 94, 96] se situe dans cette seconde perspective. Il donne à l'utilisateur la possibilité d'invoquer une fonction se comportant comme **raise**, pourvu que le but initial (3.1) – ou plus généralement le but courant – ait une forme appropriée, qui au demeurant s'obtient à partir d'un but quelconque en invoquant une fonction correspondant à **try** ... **with** ... (voir plus bas). Derrière le jeu de primitives ainsi manipulé se cachent des continuations.

3.2 Les continuations

3.2.1 Les continuations explicites

Au sens propre, une continuation est une fonction qui sera appliquée au résultat d'un calcul afin de continuer le traitement. Une continuation est obtenue en capturant un contexte d'évaluation. Considérons par exemple une fonction $f : A_1 \to A_2 \to B_1 \times B_2$ définie ainsi

$$f = \lambda\, x_1\, x_2\ \ldots\ \mathbf{let}\, b_1 = \ldots\ \mathbf{in}\ \mathbf{let}\, b_2 = \ldots\ \mathbf{in}\, \langle b_1, b_2 \rangle \tag{3.2}$$

utilisée dans le contexte suivant :

$$\mathbf{let}\, \langle b_1, b_2 \rangle = f\, a_1\, a_2\, \mathbf{in}\, (\ldots b_1 \ldots b_2 \ldots b_1 \ldots) \tag{3.3}$$

Dans un style par continuation, la fonction f prend un argument supplémentaire k qui représente la continuation et sera remplacée par :

$$f' = \lambda\, x_1\, x_2\, k\ \ldots\ \mathbf{let}\, b_1 = \ldots\ \mathbf{in}\ \mathbf{let}\, b_2 = \ldots\ \mathbf{in}\, k\, b_1\, b_2 \tag{3.4}$$

On remplace alors (3.3) par

$$f'\, a_1\, a_2\, (\lambda y_1 y_2\ \ldots y_1 \ldots y_2 \ldots y_1 \ldots) \tag{3.5}$$

De la même façon, b_1 et b_2 peuvent se calculer dans un style par continuation :

$$f' = \lambda\, x_1\, x_2\, k\ \ldots b_1'(\lambda z_1\, b_2'(\lambda z_2\ (k\, z_1\, z_2))) \tag{3.6}$$

et ainsi de suite. On passe ainsi d'un style appelé le *style direct* vers un style dit *CPS* (*continuation passing style*).

Quel est le gain de toutes ces transformations ? On a d'abord évité la construction suivie de la désintégration du couple $\langle b_1, b_2 \rangle$. Mais le point essentiel est ailleurs : le programmeur a l'opportunité d'utiliser l'argument de continuation de f', b_1', etc. comme bon lui semble. Une continuation pourra par exemple :

- être tout simplement ignorée ;
- être communiquée à des sous-expressions, ce qui permet de réutiliser un contexte pour y placer un nouveau calcul.

3.2.2 Les continuations de haut niveau

L'usage explicite de continuations permet de contrôler finement les évaluations, mais les expressions se compliquent vite : il suffit de substituer (3.6) dans (3.5) pour s'en rendre compte. Certains langages de programmation comme Scheme et SML proposent des constructions de haut niveau donnant accès aux continuations à la demande. L'opérateur de contrôle général le plus répandu est **callcc**, qui capture le contexte d'évaluation courant. Cela

permet, au niveau du programme source, de confiner l'utilisation des continuations et il revient au processus de compilation d'effectuer la traduction systématique du style direct vers des expressions en CPS.

Mais malgré les optimisations auxquelles se prête cette approche, elle entraîne un surcoût à l'exécution des programmes : les procédés de compilation fondés sur les continuations [6] se sont avérés moins efficaces que les technologies basées sur des machines à pile [75]. Les continuations ne se justifient donc que pour des programmes qui seraient difficiles à écrire sans ces dernières, ce qui est peu fréquent.

Finissons sur une note d'espoir, offerte par Internet. Il n'est pas commode de programmer un serveur Web gérant correctement les sollicitations en provenance du navigateur telles que la duplication d'une session ou le retour arrière. Christian Queinnec a proposé à cet effet un jeu de primitives exprimées au moyen de **callcc** [121].

3.2.3 Typage et preuves

La complexité qui surgit en présence de continuations est un bon argument pour appuyer leur emploi sur des preuves. Suivant la correspondance de Curry-Howard, cela revient à trouver un type aux fonctions décrites.

En ce qui concerne les opérateurs de contrôle, T. Griffin a observé que **callcc** prenait comme type $((P \to Q) \to P) \to P$ [57], tautologie classique connue sous le nom de loi de Pierce, qui a la propriété remarquable de ne pas être démontrable en logique intuitionniste. Autrement dit, **callcc** donne un sens calculatoire à une proposition classique. Cela était inattendu car la logique classique n'est pas constructive au sens suivant. Pour que l'élimination des coupures, en tant que mécanisme de calcul sur des preuves, ait un contenu calculatoire utile, il faut que certaines preuves puissent représenter des structures de données et donc disposer de propositions qui, en tant que types, contiennent au moins deux habitants non convertibles. C'est précisément ce qui fait défaut à la logique classique [51].

Laissons de côté l'interprétation du **callcc** et considérons le typage d'un programme contenant des continuations explicites. Une fonction f, de type $A \to B$ en style direct, prend le type $A \to (B \to X) \to X$ en CPS, où $B \to X$ est le type de la continuation et X est le type du résultat final. Pour de véritables preuves de correction, les types choisis pour A, B, X sont des types dépendants plus informatifs que les types simples, exactement comme dans la démarche usuelle utilisée en style direct. Cela a été bien illustré par Pierre Castéran dans [28] sur le calcul du produit des feuilles d'un arbre binaire, avec levée d'une exception dès qu'une feuille nulle est rencontrée. On commence par définir, pour tout arbre t, le type du résultat qui est un entier égal au produit des feuilles de t :

$$\mathsf{RESU}(t) = \{n{:}\mathsf{nat} \mid (\mathsf{produit}\ t\ n)\}$$

En prenant ci-dessus $A = t$, $B = \mathsf{RESU}(t)$ et $X = \mathsf{RESU}(t)$, on retrouve le but démontré par Pierre Castéran :

$$\forall t\!:\!\mathsf{tree}\ \ (\mathsf{RESU}(t) \to \mathsf{RESU}(t)) \to \mathsf{RESU}(t). \tag{3.7}$$

Il suffira, pour finir, d'invoquer le programme obtenu avec en premier argument un arbre t quelconque et en second argument la continuation identité $\lambda x\, x$. La démonstration de (3.7) s'effectue par récurrence structurelle sur t, mais en dissociant au préalable les occurences de t apparaissant dans le résultat final X. Autrement dit on se donne un arbre initial t_0 et on démontre par récurrence sur t :

$$\forall t\!:\!\mathsf{tree}\ \ (\mathsf{RESU}(t) \to \mathsf{RESU}(t_0)) \to \mathsf{RESU}(t_0). \tag{3.8}$$

A cet effet on suit exactement les mêmes étapes que pour démontrer $\forall t\!:\!\mathsf{tree}\ \ \mathsf{RESU}(t)$, mais au lieu d'un programme en style direct on obtient sa version en CPS. Une dernière modification dans la forme du but permet de rendre immédiatement un résultat final nul lorsqu'on peut le justifier.

$$\forall t\!:\!\mathsf{tree}\ \ (\mathsf{RESU}(t) \to \mathsf{RESU}(t_0)) \to (\mathsf{nul}(t) \to \mathsf{RESU}(t_0)) \to \mathsf{RESU}(t_0). \tag{3.9}$$

En fait la traduction du style direct vers le CPS correspond exactement, au niveau du typage, à une traduction par double négation, procédé utilisé par Gödel, Kolmogoroff et d'autres logiciens pour transformer des démonstrations classiques en démonstrations intuitionnistes [105] : la conclusion ainsi démontrée est obtenue à partir de l'original par insertion d'une quantité suffisante de $\neg\neg$ et il est clair que cela donne une formule classiquement équivalente.

Pour mieux comprendre, il convient d'examiner à nouveau X dans le type de f ci-dessus. Dans une traduction systématique en CPS, on utilise le même X sur l'ensemble du programme. Cela conduit à supprimer toute hypothèse sur X, qui devient une formule arbitraire ou tout simplement l'absurde $\bot$. Sur notre exemple, le type du résultat de f devient $\neg\neg B$.

Il était en un sens regrettable de préciser X dans la spécification de f, puisque X est *a priori* étranger à f. Cela soulevait un problème de modularité : comment spécifier f indépendamment de son contexte d'utilisation ? Prendre uniformément $X = \bot$ résout radicalement la question, au prix d'une interprétation nettement moins commode du résultat (faut-il considérer que son type est $\bot$? ou plutôt $\neg\neg B$?).

Une possibilité serait de raisonner directement dans un cadre constructif proche de la logique classique tel que LC [53] ou, plus proche de la syntaxe fonctionnelle, le $\lambda\mu$-calcul [109].

En résumé, il est en théorie possible de prouver la correction de fonctions avec continuations mais si on veut rester dans le cadre confortable des outils d'extraction existants, qui se placent dans une logique intuitionniste, l'énoncé même du théorème de correction s'annonce compliqué. Sur le plan pratique, le jeu en vaut-il la chandelle ?

3.3 Quelques mots sur les monades

Les monades trouvent leur origine dans la théorie des catégories. Elles ont été introduites en informatique par E. Moggi [86] et popularisées par P. Wadler [136]. Leur propriété essentielle est de réintroduire une séquentialité dans des calculs fonctionnels – alors que les programmes purement fonctionnels sont insensibles à l'ordre d'évaluation. Comment cela est-il possible ? En simplifiant, chaque application $f(e)$ est traduite par une expression plus explicite qui est soit $\mathbf{bind}\, f ... \mathbf{bind}\, e ...$ soit $\mathbf{bind}\, e ... \mathbf{bind}\, f ...$: l'opérateur **bind** a certes une définition fonctionnelle (dont le détail dépend des effets désirés) mais il n'y a pas de raison pour qu'il commute. Les deux traductions produisent alors des valeurs différentes.

Une transformation a lieu au niveau du typage : le type A est remplacé par $\mathrm{M}\, A$ qui intègre les données sur lesquelles ont lieu des effets supplémentaires. On dispose d'opérateurs spécifiques à M construisant des valeurs de type $\mathrm{M}\, A$ et, dans tous les cas, de l'opérateur **unit** qui construit un habitant de $\mathrm{M}\, A$ à partir d'une valeur pure de type A. Il est alors facile de transformer une fonction de type $A \to B$ en une fonction de type $A \to \mathrm{M}\, B$. Pour l'appliquer à une valeur de type $\mathrm{M}\, A$ on utilise l'opérateur **bind** dont le type est $\mathrm{M}\, A \to (A \to \mathrm{M}\, B) \to \mathrm{M}\, B$.

Les opérateurs **unit** et **bind** satisfont quelques lois algébriques simples dont nous ne ferons pas usage ici.

3.4 Les exceptions

Contrairement aux mécanismes généraux de continuations, les exceptions sont efficaces et sont utilisées intensivement. Quelques exemples :

- détection d'une anomalie lors d'un appel système ;
- codage d'une fonction partielle, (division euclidienne, unification de deux termes) ;
- accélération des calculs (dans l'exemple de Castéran ci-dessus, une exception est levée dès qu'une feuille nulle est trouvée) ;
- construction de termes avec partage maximum des parties communes (*sharing transducers* de G. Huet) ;
- algorithmes de recherche avec retour arrière (codage de Huffman, voir ci-dessous).

Leur utilisation rend les programmes sensibles à l'ordre d'évaluation, comme cela se voit immédiatement sur l'exemple suivant :

$$\mathbf{try}\, \langle\, \mathbf{raise}\ \mathrm{exc}(1), \mathbf{raise}\ \mathrm{exc}(2) \rangle\ \mathbf{with}\ \mathrm{exc}(n)\ \to\ \langle n, n \rangle.$$

Mais elles sont moins générales que le **callcc** : on peut coder $\mathbf{raise}\, \mathrm{exc}(n)$ et $\mathbf{try}\, E\ \mathbf{with}\ \mathrm{exc}(n)\ \to\ F$ avec **callcc**, mais pas l'inverse. Cela laisse un espoir de fournir une technique effective de preuve de programmes avec

Definition Mx :=
$\lambda C^{\mathsf{Prop}}\, \lambda A^{\mathsf{Set}}$
$\forall X\colon \mathsf{Set}\ \ \forall P\colon \mathsf{Prop}\ \ \forall e\colon P \to X\ \ (C \to P) \to (A \to X) \to X.$

Definition Mx_unit :
$\forall C\colon \mathsf{Prop}\ \ \forall A\colon \mathsf{Set}\ \ A \to \mathrm{Mx}\, CA :=$
$\lambda C^{\mathsf{Prop}}\, \lambda A^{\mathsf{Set}}\, \lambda a^{A}\ \ \lambda X^{\mathsf{Set}}\, \lambda P^{\mathsf{Prop}}\, \lambda e^{P\to X}\, \lambda i^{C\to P}\ \ \lambda k^{A\to X}\, ka.$

Definition Mx_raise :
$\forall C\colon \mathsf{Prop}\ \ \forall A\colon \mathsf{Set}\ \ C \to \mathrm{Mx}\, CA :=$
$\lambda C^{\mathsf{Prop}}\, \lambda A^{\mathsf{Set}}\, \lambda c^{C}\ \ \lambda X^{\mathsf{Set}}\, \lambda P^{\mathsf{Prop}}\, \lambda e^{P\to X}\ \ \lambda i^{C\to P}\, \lambda k^{A\to X}\, e(i\, c).$

Definition Mx_try :
$\forall C\colon \mathsf{Prop}\ \ \forall A\colon \mathsf{Set}\ \ \mathrm{Mx}\, CA \to \forall X\colon \mathsf{Set}\ \ (A \to X) \to (C \to X) \to X :=$
$\lambda C^{\mathsf{Prop}}\, \lambda A^{\mathsf{Set}}\, \lambda m^{\mathrm{Mx}\, CA}\ \ \lambda X^{\mathsf{Set}}\, \lambda k^{A\to X}\, \lambda e^{C\to X}\ \ (mX\ C\, e\, (\lambda p^{C}\ p)\, k).$

Definition Mx_bind :
$\forall A, A'\colon \mathsf{Set}\ \ \forall C, C'\colon \mathsf{Prop}$
$\mathrm{Mx}\, CA \to (A \to \mathrm{Mx}\, C'A') \to (C \to C') \to \mathrm{Mx}\, C'A' :=$
$\lambda A^{\mathsf{Set}}\ \ \lambda A'^{\mathsf{Set}}\ \ \lambda C^{\mathsf{Prop}}\ \ \lambda C'^{\mathsf{Prop}}\ \ \lambda m^{\mathrm{Mx}\, CA}\ \ \lambda f^{A\to \mathrm{Mx}\, C'A'}\ \lambda j^{C\to C'}$
$\lambda X^{\mathsf{Set}}\, \lambda P^{\mathsf{Prop}}\, \lambda e^{P\to X}\ \ \lambda i^{C'\to P}\, \lambda k^{A'\to X}$
$(mXP\, e\, (\lambda c^{C}\ i(j\, c))\, (\lambda a^{A}\ \ f\, a\, XP\, e\, i\, k)).$

FIG. 3.1 – Primitives d'accès aux exceptions

exceptions. La figure 3.1, issue de [96], fournit un jeu de fonctions primitives définies au moyen de continuations – et masquant ainsi l'usage de ces dernières – dont les types ont été conçus pour régler le problème de modularité mentionné au 3.2.3, tout en conservant une notion utilisable de type du résultat. Leurs noms sont choisis dans le vocabulaire des monades utilisé par P. Wadler [136] et elles s'utilisent de façon analogue[1] :

- si le type d'un résultat intermédiaire est A, il est remplacé par un type de la forme $\mathrm{Mx}\, CA$, où C conditionne la levée d'une exception ;
- Mx_unit construit une valeur de type $\mathrm{Mx}\, CA$ à partir d'une valeur de type A ;
- Mx_bind $m\ (\lambda a\, E)j$ correspond à $\mathbf{let}\, a = m\, \mathbf{in}\, E$ et permet de séquentialiser un calcul ; le rôle de j est de justifier la propagation d'une exception potentiellement levée dans le calcul de m.

Par ailleurs les primitives Mx_raise et Mx_try sont dédiées aux exceptions :

[1] Les monades manipulent des types de la forme MA, alors qu'ici nous avons un paramètre supplémentaire C.

- il est possible de retourner une valeur de type $\mathrm{Mx}\,CA$ en levant une exception au moyen de Mx_raise, si la proposition C est vérifiée ;
- Mx_try est l'interface entre un calcul « normal » et un calcul dans des types de la forme $\mathrm{Mx}\,CA$. Plus précisément, pour calculer un résultat de type R, on calculera un habitant de $\mathrm{Mx}\,CA$, puis on l'appliquera en prenant $X = R$ et en fournissant une continuation normale de type $A \to R$ ainsi qu'un habitant de R lorsque C est vérifiée.

Pour ne pas compliquer la présentation, on considère ici le cas où les exceptions ne transportent pas d'autre information que le fait d'avoir été levée. L'extension aux exceptions usuelles, qui transportent une valeur prise dans un type somme, ne pose pas de problème [94][2].

Notons qu'ici le type X du résultat est quantifié localement : il est déterminé au moment de l'appel de Mx_try. C'est un moyen terme entre quantifier sur X globalement (partage maximum, perte de la modularité) et quantifier sur X aux extrémités (perte d'information) : $\forall X\ X$ est une définition de $\bot$.

Le type $\mathrm{Mx}\,CA$ est très proche de la disjonction telle qu'elle est définie dans le système F (le λ-calcul typé au second ordre). En fait la définition suivante, analogue à celle de $C \vee A$ dans le système F, est équivalente :

Definition Mx :=
$\lambda C^{\mathsf{Prop}}\ \lambda A^{\mathsf{Set}}$
$\forall X : \mathsf{Set}\ \ (C \to X) \to (A \to X) \to X.$

Les primitives Mx_unit, Mx_bind, Mx_try et Mx_raise se définissent alors de manière légèrement plus simple. L'intérêt de la formulation en figure 3.1 est de séparer clairement les arguments dans **Set** et ceux dans **Prop** : l'exception e de type $P \to X$ est transmise telle quelle, tandis que le raisonnement justifiant sa propagation est représenté à part (à l'aide de c, i et j dans Mx_bind).

3.5 Utilisation

En Coq il est possible de construire un programme en mode interactif, en étant guidé par le type du résultat, au moyen de tactiques comme `Apply` qui indique l'application d'un terme. Dans ce cadre, les primitives précédentes donnent accès aux exceptions. Par exemple pour lever une exception, il suffit d'invoquer `Apply` Mx_raise. Pour les applications « ordinaires » il est nécessaire d'expliciter le séquencement au moyen de Mx_bind, ce qui revient à employer systématiquement le **let ... in** de ML. Le développement n'est donc pas plus compliqué que d'habitude.

[2]Nous nous limitons cependant aux programmes pour lesquels le type des exceptions est fixé statiquement. En ML le type des exceptions est déterminé dynamiquement, mais nous considérons que ceci relève d'une capacité – absente de CCI – à définir des sommes extensibles, capacité qui peut être considérée indépendamment de son usage pour les exceptions.

Par extraction on obtient un programme ayant le comportement souhaité en CPS (voir les exemples ci-dessous). Ces programmes fonctionnent, mais on pourrait les décompiler pour retrouver l'expression en style direct. C'est d'ailleurs dans cette optique que les primitives comme Mx_raise n'ont pas été expansées. En revanche si on les expanse on aboutit à un programme récursif terminal (*tail recursif*).

3.6 Imprédicativité du typage proposé

L'analogie avec la disjonction du système F fait apparaître l'imprédicativité du typage proposé : $\mathrm{Mx}\,CA$ est de la forme $\forall X : \mathsf{Set}\ \ldots$ et est lui-même de type **Set**, ce qui permet de prendre $X = \mathrm{Mx}\,CA$. Cela se produit lorsqu'on applique Mx_try pour construire un résultat dont le type est déjà sous la forme $\mathrm{Mx}\,CA$.

Lors de l'extraction, il est donc tout à fait possible que le programme extrait ne soit pas typable en ML. C'est le cas pour des fonctions (traduites en continuations) définies par point fixe, de la forme

$$\textbf{let rec}\, f\ x = \ldots\ \textbf{try}\ \ldots f\ y \ldots\ \textbf{with}\ \ldots$$

Cela se produit sur deux des exemples considérés ci-dessous. Bien entendu, comme le programme extrait est correct par construction, cela veut simplement dire qu'il faut désarmer la vérification de types de ML, par exemple au moyen de Obj.magic en Ocaml.

Ajoutons que cette situation « étrange » s'est rarement produite dans les développements Coq. La technique proposée ici permet de construire systématiquement des programmes Ocaml corrects mais non typables en Ocaml.

3.7 Exemples

Les exemples ci-dessous sont disponibles dans les contributions Coq.

3.7.1 Unification de termes du premier ordre

Unifier deux termes consiste à rendre la substitution la plus générale qui les égalise s'il en existe une, ou un échec sinon. L'algorithme le plus simple procède par récursion en parcourant simultanément les deux termes. J. Rouyer en a donné une version obtenue par extraction à partir d'un développement d'environ 2800 lignes [125]. Dans cette version, la fonction rend une somme échec + subst, ce qui oblige à analyser le résultat de chaque appel récursif pour, dans un cas, propager l'échec et dans l'autre, construire une nouvelle substitution ou rendre un échec.

La version avec exception rend simplement une substitution dans subst. En cas d'échec, une exception est levée et elle est attrapée directement au

```
let encode a t =
    let rec lookup = function
      Leaf(x) →if x = a then [] else raise Not_found
    | Node(t1,t2) →try L::lookup t1 with Not_found →R::lookup t2
    in lookup t
```

FIG. 3.2 – Codage de Huffman en syntaxe Ocaml

```
let lookup a =
  let rec lkup = function
    Leaf b →if b = a then mx_unit [] else mx_raise prop
  | Node (t1, t2) →Obj.magic
      mx_try
        (lkup t1) (fun l1 →mx_unit (L :: l1))
        (fun _ →
          (mx_bind (lkup t2) (fun l2 → mx_unit (R :: l2))) prop)
  in lkup
let encode a t _ = mx_try (lookup a t) (fun x →x) (fun _ → [])
```

FIG. 3.3 – Codage de Huffman extrait en Coq

niveau du premier appel, et c'est seulement là qu'est élaboré un habitant de échec + subst. Il s'est avéré possible d'obtenir cette version à partir de celle de J. Rouyer en ne changeant que le strict nécessaire, soit une centaine de lignes (voir [94] ou [96] pour une discussion plus détaillée).

Cet exemple ne comporte qu'un seul **try** ... **with** ... qui se trouve à l'extérieur de l'appel à une fonction récursive. L'imprédicativité de Mx n'est donc pas mise à contribution, le programme extrait est typable en ML. Ce n'est pas le cas des deux exemples suivants.

3.7.2 Codage de Huffman

L'algorithme en figure 3.2 construit le chemin conduisant à une lettre a dans un arbre binaire (qui est conçu de sorte que les lettres les plus fréquentes sont situées plus près de la racine que les autres). La liste de directions (L ou R) obtenue est le code de cette lettre dans la compression de Huffman. L'algorithme est utilisé sous la précondition que a est dans l'arbre.

Intuitivement, si la lettre n'a pas été trouvée dans le sous-arbre gauche, ceci est indiqué par une exception qui est attrapée ; la branche droite est alors essayée et en cas de nouvel échec l'exception est automatiquement propagée.

```
let rec core_cop = function
        Leaf(n) →raise Identity
      | Nd(Blue, t1) →try let v1 = core_cop t1 in Nd(Red, v1)
                      with Identity →Nd(Red, t1)
      | Nd(x, t1) →Nd(x, core_cop t1)
let eff_cop t = try core_cop t with Identity →t.
```

FIG. 3.4 – Transducteur avec partage en syntaxe Ocaml

Cependant mieux vaut oublier cet argument opérationnel pour la preuve formelle. Il suffit de démontrer les théorèmes suivants en suivant l'algorithme, les obligations de preuve engendrées se déchargent très simplement.

lookup : $\forall a$:A $\forall t$:tree (Mx (elsewhere a t) $\{l$:Id | (path a l $t)\}$).
not_elsewhere : $\forall a$:A $\forall t$:tree $\forall l$:Id (path a l $t) \to$ (elsewhere a $t) \to \bot$.
encode : $\forall a$:A $\forall t$:tree $(\exists l$:Id | (path a l $t)) \to \{l$:Id | (path a l $t)\}$.

Le programme obtenu par extraction est en figure 3.3, à l'introduction près de *Obj.magic* qui a été faite à la main. L'argument de mx_raise noté prop est la trace de la justification pour lever l'exception (il correspond à c de type C dans la définition de Mx_raise en figure 3.1). Il s'agit simplement de l'unique habitant du type unit.

Le problème de typage se présente ainsi sur cet exemple. Le type de lkup est de la forme[3] direction list $\to \alpha t$, avec

$$\alpha t = (\text{unit} \to \alpha) \to \text{unit} \to (\text{direction list} \to \alpha) \to \alpha,$$

sauf dans lkup t_1 où il est de la forme direction list $\to (\alpha t)t$. Le type désiré est en fait direction list $\to \forall\alpha\ \alpha t$, mais on est alors dans le système F, où l'inférence de type n'est plus décidable.

3.7.3 Un transducteur avec partage maximal

Considérons une fonction qui prend en entrée une donnée t – disons un arbre ou une liste pour fixer les idées – et qui rend un résultat r de même type, identique à t éventuellement à quelques modifications près. Pour économiser la mémoire, on souhaite partager autant que possible les représentations de t et de r. En particulier, dans le cas le plus simple où $t = r$, il doit y avoir partage complet, aucune cellule mémoire supplémentaire n'est nécessaire.

À cet effet, G. Huet a introduit dans les années 80 une technique appelée *sharing transducers*, dans laquelle une exception (appelée Identity) est levée

[3] En ML l'application au niveau des types est notée de manière postfixe : αt veut dire t appliqué à α.

```
let rec core_cop = function
    Leaf n →mx_raise
  | Nd(c, t1) →
    match c with
      Blue →Obj.magic
        mx_try
          (core_cop t1) (fun v1 →mx_unit (Nd(Red,v1)))
          (fun _ →mx_unit (Nd(Red,t1)))
    | Red →
        mx_bind (core_cop t1) (fun v1 →mx_unit (Nd(Red,v1))) prop
    | Yellow →
        mx_bind (core_cop t1) (fun v1 →mx_unit (Nd(Yellow,v1))) prop
let eff_cop t = mx_try (core_cop t) (fun x →x) (fun _ →t)
```

FIG. 3.5 – Transducteur avec partage extrait en Coq

lorsqu'on sait qu'il y a égalité entre le résultat et l'argument. Le partage est réalisé au moyen de la fonctionnelle suivante :

$$\textbf{let}\ \mathsf{share}\ f\ x = \textbf{try}\ f\ x\ \textbf{with}\ \mathsf{Identity} \rightarrow x$$

Il suffit alors de remplacer les appels de la forme $f\ x$ par $\mathsf{share}\ f\ x$ pour avoir l'effet désiré. Cela est illustré dans [96] avec le remplacement de Blue par Red dans une liste de couleurs. L'algorithme est en figure 3.4 (share y est déplié). On démontre sa correction (il rend le même résultat que la fonction def_cop définie de manière évidente) et qu'il effectue le partage demandé à partir de la spécification suivante, qui indique que si une exception est levée alors le résultat est identique à l'original et que réciproquement seule une valeur différente de l'original peut être rendue par calcul ordinaire.

$$\forall t{:}\mathsf{tree1}\ \ (\mathrm{Mx}\ \ t = (\mathsf{def\_cop}\ t)\ \ \{t'{:}\mathsf{tree1} \mid t' = (\mathsf{def\_cop}\ t)\ \wedge\ t' \neq t\}).$$

Le programme extrait est en figure 3.5. En toute rigueur, l'imprédicativité de Mx n'est pas indispensable sur cet exemple, mais pour une raison *ad-hoc*. Il se trouve que chaque branche du **try** ... **with** ... rend une valeur ordinaire et non une exception, on peut donc factoriser l'appel de Mx_unit de la manière indiquée en figure 3.6. Cette astuce ne s'applique pas en général, comme en témoigne le codage de Huffman ci-dessus.

3.8 Conclusion

Le procédé indiqué dans ce chapitre fournit une solution assez naturelle à utiliser pour qui veut définir des programmes fonctionnels munis d'ex-

```
Blue →
  mx_unit
    (mx_try
      (core_cop t1) (fun v1 →Nd(Red,v1))
      (fun   _ →(Nd(Red,t1))))
```

FIG. 3.6 – Variante prédicative

ceptions et corrects par construction. Les obligations de preuve sont les mêmes que si l'on utilisait un type somme *type normal* + *exception levée* à la place de *type normal*. La monade des exceptions de [136], par exemple, effectue une telle traduction. Son inconvénient est que dans le programme résultant, chaque calcul d'une valeur v de type T à partir d'une valeur v' de type T' se traduit systématiquement par l'analyse d'un élément de T + *exception levée*, le calcul proprement dit puis l'encapsulation du résultat v' dans T' + *exception levée*. Le développement de l'algorithme d'unification mentionné dans la section 3.7.1 a également été conçu de cette manière à l'origine.

Le mécanisme sous-jacent mis en œuvre ici repose en revanche sur des continuations, de sorte qu'en cas de levée d'exception le contrôle passe immédiatement au niveau du gestionnaire (**try** ... **with** ...) correspondant. Il est plus simple et limité que les techniques de traduction par double négation – il ne fournit pas de **callcc** . Il faut plutôt y reconnaître le codage de la somme en λ-calcul, connu dès l'invention de ce dernier par Church, et dont le typage imprédicatif a été proposé par Girard. Inversement, on peut considérer que la représentation des structures de données en λ-calcul s'interprète comme un calcul de continuations.

Chapitre 4

Le contrôle de conformité ABR

Ce chapitre est consacré à une petite *success story* des méthodes formelles : la preuve de correction d'un petit algorithme réactif et temps-réel, preuve indispensable à la normalisation d'un protocole appelé l'ABR (*Available Bit Rate*). Cette preuve a d'abord été faite au crayon [102, 103], quelques invariants significatifs ont même été intégrés au document de normalisation I.371 [67]. Elle a ensuite été formalisée ou reprise avec plusieurs outils, au CNET et dans d'autres laboratoires, le plus souvent dans le cadre du projet national FORMA, puis du projet RNRT Calife [24, 29] et a ainsi servi de terrain d'expérience pour différentes techniques de vérification automatique par *model checking*, classique [7] ou temporisé [15, 25, 26, 62], par réécriture et procédures de décision [50, 49, 127, 107] par des outils interactifs [30, 134, 29, 98, 100, 115] et même de conception par raffinements [2, 3].

Le principal auteur de l'algorithme est Christophe Rabadan [122]. A l'origine, le problème m'a été transmis par Francis Klay et c'est avec lui que les réflexions initiales ont été menées. Sa propre approche l'a conduit à proposer une suite d'algorithmes aboutissant progressivement, au choix, à celui qui était soumis ou à une version mieux conçue. Ces étapes facilitent la formalisation et des expériences ont été menées au moyen d'outils de réécriture [127, 128].

4.1 Contexte

L'ATM (*Asynchronous Transfer Mode*) est un mode de transmission d'informations inventé au CNET Lannion au début des années 80 – sous le nom de TTA (Technique de Transmission Asynchrone – dans le but de transporter des flux de caractéristiques violemment opposées (données, voix, video) sur les mêmes ressources physiques et sans perturbations perceptibles par les usagers. Ces ressources étant évidemment bornées il est impossible de garantir des critères de qualité de service – taux de pertes, gigue[1], délai de

[1] Déformation d'un signal périodique.

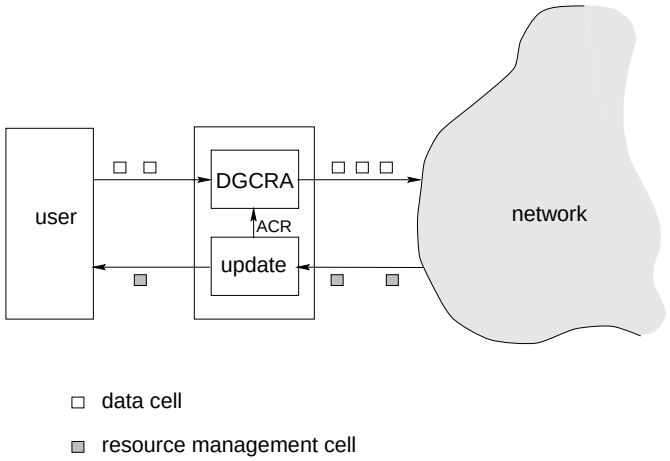

FIG. 4.1 – Le contrôle de conformité

transmission, sans un minimum d'interaction avec les équipements aux frontières du réseau. Dans la suite, ces derniers sont assimilés indifféremment à des terminaux ou à leur usager.

En ATM, chaque connexion est précédée d'une phase de négociation établissant la qualité de service garantie par le réseau en échange de paramètres décrivant le débit moyen et ses irrégularités du flux souhaité, paramètres acceptables ou non suivant l'état de congestion du réseau. Afin de protéger les autres connexions, ces paramètres sont ensuite contrôlés dans un dispositif situé entre le terminal et le réseau, appelé GCRA (*Generic Cell Rate Algorithm*) ou DGCRA (*Dynamic* GCRA). Il s'agit essentiellement d'assurer que le débit ne dépasse jamais ACR (*Allowed Cell Rate*), la valeur autorisée à instant donné, ce qui revient à contrôler la durée séparant l'émission de deux cellules successives[2]. Toute cellule en excès peut être supprimée.

Il existe différentes sortes de connexions ou *capacités de transfert*. Dans les plus simples, ACR est constant et l'algorithme de contrôle ne présente pas de difficulté particulière. La capacité ABR a été conçue pour les flux non prioritaires sujets à variations. Le débit peut augmenter si les ressources sont disponibles, mais doit diminuer (dans les limites du contrat négocié) si l'environnement l'exige.

Le problème de base se présente simplement (voir figure 4.1). Pour chaque connexion, le réseau calcule le débit admissible en fonction des demandes et des ressources disponibles. Le résultat est envoyé à l'usager dans des cellules de gestion dites RM (*Resource Management*) et ce dernier doit se conformer aussitôt au débit indiqué.

[2]Les cellules sont des paquets indivisibles, leur taille est fixée à 53 octets : 48 octets de données et 5 octets administratifs.

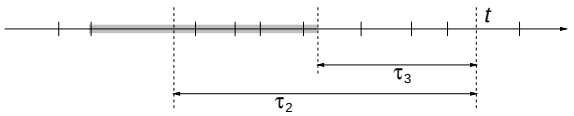

FIG. 4.2 – Cellules à prendre en compte à la date t

Mais comme il y a nécessairement un temps de réaction, dû à la distance entre le dispositif de contrôle et l'usager d'une part et à la traversée de couches matérielles et logicielles d'autre part, et que ce temps n'est connu qu'approximativement (il est compris entre deux bornes appelées τ_3 et τ_2 pour des raisons historiques ; ces bornes sont déterminées en début de connexion), n'importe lequel des débits transmis sur l'intervalle dessiné en figure 4.2 est susceptible d'avoir été utilisée par le terminal, en toute légitimité.

C'est donc la valeur maximum transportée par les cellules RM de cet intervalle qui doit être prise en compte au niveau du dispositif de contrôle. Calculer ce maximum Acr(s) à l'instant précis s où il prend effet prendrait trop de temps au regard de la fréquence des cellules RM et des débits considérés. Aussi le calcul est-il effectué d'avance et placé dans un échéancier : c'est le rôle du bloc `update` dans la figure 4.1.

Même ainsi, il apparaissait aux yeux des opérateurs et constructeurs élaborant les normes qu'un echéancier exact et bon marché consommerait trop de ressources (quelques dizaines à quelques centaines d'événements). L'idée était donc d'utiliser un échéancier limité à deux événements, programmé astucieusement de façon que l'écart entre la valeur délivrée `ACR` et la valeur idéale Acr soit le plus faible possible.

Différentes propositions ont été faites, avec des algorithmes allant d'une page à une dizaine de pages. Le problème épineux était alors de tenir la qualité de service promise : en aucun cas l'usager ne doit perdre de cellules légales. Autrement dit il fallait prouver qu'à tout instant s, `ACR`(s) est au moins aussi grand que Acr(s). C'est ce qui a été fait sur l'algorithme dit B', proposé par Christophe Rabadan et Annie Gravey [122]. Il est reproduit aux figures 4.3 et 4.4 dans le langage des commandes gardées de Dijkstra où, comme en `B`, les assignations simultanées sont séparées par le symbole « || ». L'algorithme comporte très peu de variables. En dehors de `ACR`, il y a `tfi` et `tla`, dates du premier et du dernier évènement de l'échéancier. Les estimations de `ACR` prévues pour ces dates sont respectivement `Efi` et `Ela`. La variable `Emx` contient le maximum de ces deux valeurs.

```
if t_k < tfi then
  if Emx ≤ ER_k then
    if tfi < t_k+τ3 then
      if t_k+τ3 < tla ∨ tfi = tla then
        Emx := ER_k || Ela := ER_k || tla := t_k+τ3
      else
        Emx := ER_k || Ela := ER_k
    else
      if ACR ≤ ER_k then
        Emx := ER_k || Efi := ER_k || Ela := ER_k
                    || tfi := t_k+τ3 || tla := t_k+τ3
      else
        Emx := ER_k || Efi := ER_k || Ela := ER_k || tla := tfi
  else
    if ER_k < Ela then
      Efi := Emx || Ela := ER_k || tla := t_k+τ2
    else
      Efi := Emx || Ela := ER_k
else
  if ACR ≤ ER_k then
    Efi := ER_k || Ela := ER_k || Emx := ER_k
                || tfi := t_k+τ3 || tla := t_k+τ3
  else
    Efi := ER_k || Ela := ER_k || Emx := ER_k
                || tfi := t_k+τ2 || tla := t_k+τ2
```

FIG. 4.3 – Algorithme I.371-B' (arrivée de la k^{e} cellule)

4.2 Démarche

Considérant que nous pouvions nous placer sous les hypohèses d'un système réactif synchrone [16], la démarche suivie a consisté à modéliser le système sous forme d'un système de transitions temporisé comportant :

- des transitions instantanées (dites *discrètes*), qui correspondent soit à l'échéance d'une mise à jour de `ACR` programmée dans l'échéancier, soit à l'arrivée d'une cellule RM avec mise à jour immédiate de l'échéancier ;
- des transitions (dites *continues* ou *temporisées*) où le temps s'écoule sans variation de l'état du système.

`ACR` := `Efi` || `tfi` := `tla` || `Efi` := `Ela` || `Emx` := `Ela`

Si `tfi` = t_k, exécuter d'abord cette transition
puis l'algorithme de la figure 4.3.

FIG. 4.4 – Algorithme I.371-B' (événement issu de l'échéancier)

C'est donc une adaptation des automates temporisés de Alur et Dill [5], sans contrainte sur l'espace des états [3] : chaque événement de l'échéancier est un couple comportant une date et une valeur de débit pour `ACR`. La date pourrait être modélisée par une horloge (une variable qui croît dans le temps avec une pente de 1, ou peut être remise à 0 par une transition discrète), mais le type de la composante de débit n'est pas fini – même s'il tient sur 48 bits, cette donnée est manifestement déplacée. L'idée n'était pas d'appliquer un outil de *model checking* sur une abstraction finie du système, mais démontrer que $\mathrm{Acr}(s) \leq \texttt{ACR}(s)$ est un invariant en utilisant la logique de Hoare.

4.2.1 Invariant

La principale difficulté dans ce genre de problème est de découvrir l'invariant inductif convenable. Il s'est vite avéré que les explications informelles existantes, essentiellement quelques diagrammes représentatifs de quelques « situations typiques », n'étaient d'aucun secours. L'algorithme lui-même n'est pas des plus simples à lire, j'ai renoncé à essayer de le comprendre.

Une difficulté moins usuelle pour la logique de Hoare est la prise en compte du temps, au sens quantitatif :

- $\mathrm{Acr}(s) \leq \texttt{ACR}(s)$ résulte d'actions effectuées sur l'état à des instants antérieurs à s ;
- dualement, les dates programmées dans l'échéancier portent sur des instants futurs ;
- en fait, sont importantes non seulement les dates programmées dans l'échéancier mais toutes les dates intermédiaires, car l'échéancier exact idéal comporte en général bien plus d'événements.

La première intuition qui a guidé la formulation de l'invariant était de voir un échéancier comme la définition d'une fonction $f(t)$ à comparer à $\mathrm{Acr}(t)$, ou encore de voir $\mathrm{Acr}(t)$ comme un échéancier idéal constant. Il est en outre très facile de formaliser Acr à partir de la figure 4.2.

Mais Acr n'est pas directement utilisable car la valeur $f_s(t)$ programmée à l'instant t dans l'échéancier de l'état s n'a aucune raison de respecter

[3] Une approche apparentée et sans doute plus proche de celle décrite ici est celle des systèmes de transition chronométrés (*clocked transition systems*) de Manna et Pnueli [18, 71, 79]

$$\texttt{utd} = \text{false} \;\Rightarrow\; s = \texttt{tfi} \qquad (\mathrm{I_{utd}})$$

$$\texttt{Emx} = \max(\texttt{Efi}, \texttt{Ela}) \qquad (\mathrm{I_{max}})$$

$$\texttt{Ela} = \mathrm{ER}_n \qquad (\mathrm{I_{Ela}})$$

$$\texttt{tfi} \leq \texttt{tla} \leq t_n + \tau_2 \qquad (\mathrm{I_{fil}})$$

$$(\texttt{tfi} = s \Rightarrow \texttt{utd} = \text{true}) \Rightarrow \texttt{tfi} \leq s \;\Rightarrow\; \forall t,\; s \leq t \;\Rightarrow\; \mathrm{Approx}(n,t) \leq \texttt{ACR} \qquad (\mathrm{I_{tfs}})$$

$$\texttt{ACR} < \texttt{Efi} \;\Rightarrow\; \texttt{tfi} \leq t_n + \tau_3 \qquad (\mathrm{I_{Et1}})$$

$$\texttt{Efi} < \texttt{Ela} \;\Rightarrow\; \texttt{tla} \leq t_n + \tau_3 \qquad (\mathrm{I_{Et2}})$$

$$\texttt{tfi} = \texttt{tla} \;\Rightarrow\; \texttt{Efi} = \texttt{Ela} \qquad (\mathrm{I_{ttE}})$$

$$\forall t \quad s \leq t < \texttt{tfi} \;\Rightarrow\; \mathrm{Approx}(n,t) \leq \texttt{ACR} \qquad (\mathrm{I_{Ub1}})$$

$$\forall t \quad \texttt{tfi} \leq t < \texttt{tla} \;\Rightarrow\; \mathrm{Approx}(n,t) \leq \texttt{Efi} \qquad (\mathrm{I_{Ub2}})$$

$$\forall t \quad \texttt{tla} \leq t \;\Rightarrow\; \mathrm{Approx}(n,t) \leq \texttt{Ela}\,. \qquad (\mathrm{I_{Ub3}})$$

FIG. 4.5 – Invariant de l'algorithme I.371-B'

l'inégalité voulue si t est tant soit peu éloigné de s : l'arrivée de la prochaine cellule RM remettra les prévisions en cause.

La comparaison est donc faite par rapport à la valeur prévisible compte-tenu des cellules RM connues. Plus précisément, on introduit une suite de fonctions $\mathrm{Approx}_n(t)$ qui est définie comme Acr mais en ne prenant en compte que les n premières cellules RM. On vérifie facilement que cette suite converge vers Acr quand n tend vers l'infini, et qu'elle est à jour pour la date courante.

Lemme 1 *En notant $n(s)$ le nombre de cellules reçues à l'instant s, on a* $\mathrm{Approx}_{n(s)}(s) = \mathrm{Acr}(s)$.

L'invariant est alors formulé au moyen de Approx (figure 4.5). Cette version [100] comporte une variable supplémentaire, *utd*, sur laquelle nous revenons plus bas. On retrouve l'invariant initial de [102, 103] en faisant $\texttt{utd} = \text{true}$. Le théorème désiré est une conséquence simple du lemme 1 et des invariants $\mathrm{I_{tfs}}$ et $\mathrm{I_{Ub1}}$ dans lesquels ont prend $t = s$: il suffit d'examiner les deux cas $\texttt{tfi} \leq s$ et $s < \texttt{tfi}$ (le dernier cas représente un échéancier vide). On peut aussi noter que les invariants $\mathrm{I_{Ub1}}$, $\mathrm{I_{Ub2}}$ and $\mathrm{I_{Ub3}}$ signifient que $\mathrm{Approx}(n,t) \leq f(t)$ pour $t \geq s$, où $f(t)$ est définie par : $f(t) = \texttt{ACR}$ pour $s \leq t < \texttt{tfi}$, $f(t) = \texttt{Efi}$ pour $\texttt{tfi} \leq t < \texttt{tla}$ et $f(t) = \texttt{Ela}$ pour $\texttt{tla} \leq t$.

Un lemme clef pour la démonstration est la caractérisation incrémentale suivante de Approx.

Lemme 2 *La fonction* Approx_n *est calculée à partir de* Approx_{n-1} *de la manière suivante :*

$t < t_n + \tau_3$	$t_n + \tau_3 \leq t < t_n + \tau_2$	$t_n + \tau_2 \leq t$
$\text{Approx}_{n-1}(t)$	$\max(\text{Approx}_{n-1}(t), ER_n)$	ER_n

On démontre alors :

Théorème 1 *A tout instant* s *on a* $\text{Acr}(s) \leq \texttt{ACR}$.

4.2.2 Modèle d'exécution

On n'échappe pas aux problèmes usuels de continuité à gauche des systèmes temporisés, qui se présentent lorsque le temps atteint la date d'exécution d'une transition discrète. Dans le modèle d'exécution l'invariant doit toujours être conservé afin que le raisonnement par récurrence sous-jacent soit valide.

Dans le cas de l'ABR, l'arrivée d'une cellule RM ne pose pas de problème : son effet sur Acr est dans le futur et la relation $\text{Acr}(s) \leq \texttt{ACR}$ n'est donc pas falsifiée. Elle peut l'être, en revanche, lorsque le temps atteint `tfi` : c'est précisément le but de la transition de l'échéancier de rétablir la situation.

Trois procédés différents ont été utilisés pour éviter que la récurrence soit brisée. La première version [102, 103] utilise un système temporisé « impur » : les transitions discrètes sont précédées d'une progression du temps de sorte que l'état intermédiaire où $t = \texttt{tfi}$ mais où les affectations de la figure 4.4 sont à faire n'est même pas considéré. Un inconvénient de cette approche est que la garde de ces transitions se complique : il faut indiquer que dans le laps de temps écoulé aucun événement ne peut se produire. Il s'agit d'une redondance par rapport aux transitions continues, qui ont précisément pour effet de faire avancer le temps sous cette condition. Cela pourrait aussi être une source de complications techniques s'il fallait effectuer un produit de plusieurs systèmes semblables.

La modélisation effectuée par Béatrice Bérard et Laurent Fribourg [25] et reprise dans un travail en commun [26] introduit dans l'état une composante supplémentaire qui a un effet très subtil : la condition d'observation, matérialisée par le franchissement d'une transition *snapshot* est inhibée exactement dans l'état éphèmère où l'inégalité demandée est falsifiée (voir la figure 4.9). En l'occurrence c'est précisément lorsque l'échéancier devient vide. La dichotomie transitions discrètes/transitions temporisées est alors parfaitement respectée.

Enfin, pour revenir au plus près de l'algorithme normalisé, la version présentée dans [100] introduit un booléen `utd` (*up to date*). L'idée sous-jacente est d'indiquer que le système est dans un état *stable* où, par définition, le temps peut progresser, au moyen d'un jeton qui circule entre le temps

$$\texttt{stb} = \text{true} \;\Rightarrow\; \texttt{ctl} = \text{true} \qquad (\mathrm{I_{utd}})$$

$$(\texttt{tfi} = s \;\Rightarrow\; \texttt{ctl} = \text{true} \;\Rightarrow\; \texttt{stb} = \text{true}) \Rightarrow \\ \texttt{tfi} \leq s \;\Rightarrow\; \forall t,\; s \leq t \;\Rightarrow\; \text{Approx}(n,t) \leq \texttt{ACR} \qquad (\mathrm{I_{tfs}})$$

FIG. 4.6 – Nouveaux invariants de l'algorithme I.371-B'

Transitions d'échéancier

$$\texttt{stb} = \text{false} \qquad (\mathrm{Gi_0})$$

$$s = \texttt{tfi} \qquad (\mathrm{Gi_1})$$

Arrivée d'une cellule RM

$$s = \mathrm{tb}_k \qquad (\mathrm{Ge_1})$$

$$\texttt{tfi} = \mathrm{tb}_k \;\Rightarrow\; \texttt{ctl} = \text{false} \qquad (\mathrm{Ge_2})$$

Transitions de temps

$$\texttt{stb} = \text{true} \qquad (\mathrm{Gt_0})$$

$$s \leq s' \qquad (\mathrm{Gt_1})$$

$$s < \texttt{tfi} \;\Rightarrow\; s' \leq \texttt{tfi} \qquad (\mathrm{Gt_2})$$

$$\forall i{:}\texttt{nat}\;\; s < \mathrm{tb}_i \;\Rightarrow\; s' \leq \mathrm{tb}_i \qquad (\mathrm{Gt_3})$$

FIG. 4.7 – Gardes des transitions

(`utd` = true) et le système discret (`utd` = false) : seule une transition discrète peut faire passer `utd` à true, marquant la terminaison des calculs de l'instant courant, et seule une transition temporisée peut effectuer l'inverse.

Pour conserver la dichotomie entre les deux sortes de transition, il faut considérer `utd` comme une composante propre au modèle d'exécution général mais étrangère au modèle du système étudié : `utd` intervient dans l'invariant mais n'est pas testée par le système (contrairement à la composante d'état de la solution de [25, 26]).

L'invariant principal qui s'ensuit de celui de la figure 4.5 est donc :

$$\texttt{utd} = \text{true} \;\Rightarrow\; \text{Approx}(n,s) \leq \texttt{ACR}\,. \qquad (\mathrm{I_{main}})$$

Pour démontrer le théorème 1 il faut donc vérifier la propriété de vivacité $\texttt{utd} = \text{false} \;\Rightarrow\; \Diamond_0(\texttt{utd} = \text{true})$, où $\Diamond_0 P$ signifie que P devient fatalement

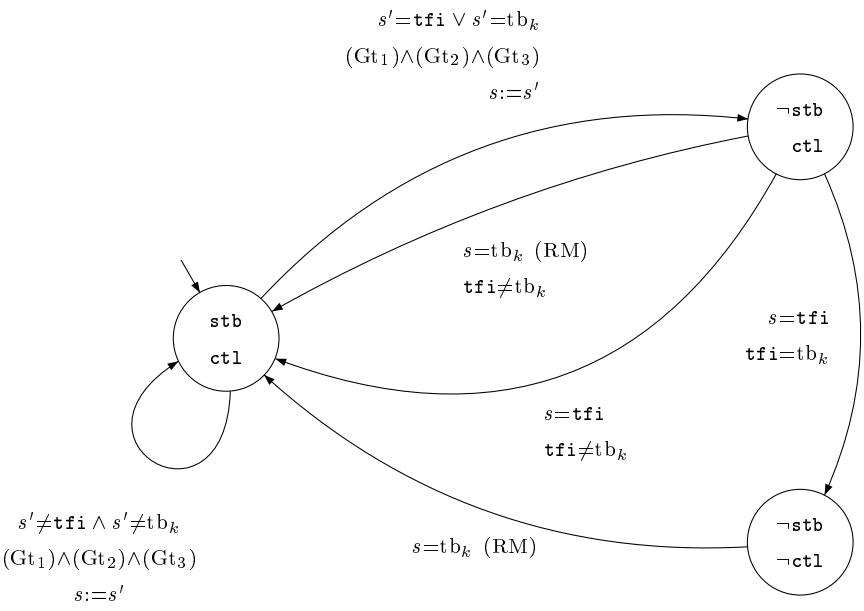

FIG. 4.8 – Transitions du système I.371-B' + temps

vrai à la date courante. Cela est trivial ici, compte tenu de I_{utd} et du fait que la transition d'échéancier établit `utd` = true.

Pour un système quelconque, cela correspond à l'obligation d'assurer la progression du temps au moyen d'un lemme de terminaison énonçant que toute suite de transitions (discrètes) débutant dans un état instable aboutit à un état stable.

Dans la version [100], le modèle d'exécution présenté ici n'est cependant pas tout à fait respecté : `utd` n'est utilisée que pour le tir d'un événement de l'échéancier. Cette optimisation fonctionne parce que l'arrivée d'une cellule RM n'a d'effet que sur des valeurs ultérieures de `ACR` (et Acr), mais on ne peut plus affirmer que le système est stable si et seulement `utd` = true. En particulier, si la date `tfi` coïncide avec l'arrivée d'une cellule RM, on a alors une réaction instantanée comportant deux transitions successives, entre lesquelles `utd` passe à true (le traitement de la cellule RM vient en second).

La présente synthèse a été l'occasion de pousser la démarche jusqu'au bout. La variable `utd` est remplacée par `stb` (*stable*), qui respecte strictement la politique du jeton. Il est alors nécessaire de matérialiser l'état fugitif intermédiaire qui apparaît lors d'une collision, par exemple au moyen d'un booléen de contrôle `ctl` qui ne fait que formaliser l'obligation (mentionnée dans la spécification) d'exécuter les deux transitions dans un ordre précis[4].

[4]Cette variable n'intervient donc pas dans les transitions temporisées. Sa valeur est

Une modification convenable de l'invariant est indiquée en figure 4.6. Les gardes des transitions internes (échéancier), externes (arrivée de la k$^{\text{e}}$ cellule RM) et de passage du temps sont dans la figure 4.7. Les deux dernières (Gt_2) et (Gt_3), indiquent que le temps ne peut franchir une date programmée (`tfi`) ou d'arrivée de cellule (tb_i avec i quelconque). La garde `stb` = false n'est utile que pour la transition interne. Le système de transition qui en résulte est schématisé en figure 4.8. Seules les variables `stb` et `ctl` y sont représentées. Le temps peut augmenter au cours d'une transition si, et seulement si, la source de cette dernière est l'état de gauche. Les arcs étiquetés par l'arrivée d'une cellule RM (respectivement par $s = \texttt{tfi}$) correspondent à l'exécution d'un même algorithme, celui de la figure 4.3 (respectivement 4.4).

4.3 Formalisation en Coq

La vérification de l'invariant ci-dessus n'est pas vraiment difficile mais couvre une dizaine de pages de calculs fastidieux, avec les risques que cela implique. Ils ont été vérifiés mécaniquement avec Coq [98, 100] – ainsi que l'ensemble de la démarche – ce qui a permis au passage de rectifier la démonstration de l'une des 80 obligations de preuve. La valeur du résultat principal ne réside pas dans sa profondeur mais dans la simplicité de son énoncé : tout tient dans le théorème 1 et dans la formulation suivante de Acr (qui n'est pas la plus élégante, mais c'était celle fournie par le comité de normalisation) :

$$\text{Acr}(t) = \max\{\text{ER}_i \mid i \in I(t)\}$$
$$i \in I(t) \quad \text{ssi} \quad (t - \tau_2 < t_i \leq t - \tau_3) \vee (t_i \leq t - \tau_2 < t_{i+1})$$
$$t_1 < t_2 < \ldots t_n < \ldots$$

On fait naturellement appel à l'ordre supérieur pour exprimer la caractérisation incrémentale de Approx (lemme 2). Le système lui-même est représenté par un espace d'états $\texttt{state}(s)$ et trois fonctions de transition de $\texttt{state}(s)$ vers $\texttt{state}(s')$. Le type $\texttt{state}(s)$ est un type *record* comprenant les composantes `ACR`, `tfi`, `Efi`, `tla`, `Ela` et `Emx`, ainsi que des composantes pour les invariants. Pour les transitions temporisées on a $s \leq s'$ et une garde assurant qu'aucun événement n'a lieu dans l'intervalle. Pour les transitions discrètes on a $s = s'$. Leur définition est exprimée en posant un but $\texttt{state}(s')$ sous l'hypothèse $\texttt{state}(s)$, en fournissant les valeurs de `ACR`...`Emx` appropriées, puis en résolvant les sous-buts engendrés pour les composantes correspondants aux invariants. Les obligations de preuve sont bien entendu les mêmes que celles obtenues auparavant à la main par calcul des plus faibles préconditions. Nous avons vu au 2.4.4 comment certaines étapes de vérification ont été automatisées.

---

toujours true, sauf dans l'état fugitif entre les deux transitions à exécuter consécutivement si une cellule RM arrive à la date `tfi`.

4.4 Autres expériences

4.4.1 Utilisations de B

Dans la mesure où une bonne partie de la vérification est un calcul de plus faible précondition, on pouvait penser que les outils de la méthode B [2] s'appliqueraient aisément. L'expérience a été tentée en 98 avec deux étudiants (G. Blorec et P. Chavin) mais n'a pas été concluante. Les difficultés principales trouvaient leur cause dans l'utilisation de la famille de fonctions Approx dans l'invariant : en B, une fonction est simplement un ensemble de couples, et il est préfèrable de ne considérer que des relations à domaine fini.

Par la suite, J-R. Abrial a repris le problème sous un angle complètement différent [3]. Il s'agissait de reconstruire par raffinements successifs une solution à la spécification intiale. C'est l'une des toutes premières expériences à l'origine du B événementiel [4]. Il en est résulté un algorithme, certes un peu différent de celui de la norme, mais nettement plus compréhensible et où les décisions conceptuelles sont claires.

4.4.2 Dans le cadre de FORMA

Plusieurs équipes ont essayé des outils de *model checking* ou d'inférence d'invariants dans le cadre du projet FORMA[5], qui réunissait des laboratoires académiques ou industriels autour d'applications issues des télécommunications, du nucléaire et de l'avionique.

Sans entrer dans les détails, l'ABR n'a pu être véritablement traité par les outils automatiques en prenant comme référence la spécification initiale. Lors d'une première tentative, au moyen de MEC [7] et Uppaal [15], l'équipe du LaBRI a cherché à reproduire la propriété désirée au moyen d'un automate effectuant continuellement les calculs de maximum et codé aussi naïvement que possible, de façon à minimiser les biais de transcriptions. Cette approche s'est malheureusement heurtée à une explosion du nombre d'états qui a limité la vérification à des configurations éloignées de la réalité (en termes de fréquences d'événements).

La tentative suivante (au LSV) a permis de retrouver automatiquement les invariants avec GAP et HyTech, en effectuant la comparaison par rapport à la version opérationelle de Acr – celle qui est exprimée dans Approx. Nous y revenons au 4.5.2.

Par ailleurs des recherches ont été menées sur la génération automatique de jeux de test guidée par une démonstration déductive.

[5]Ce projet, animé par J. Sifakis, était financé partiellement par le Ministère de la Recherche, le CNRS et la DGA.

4.4.3 Outils de réécriture et procédures de décision

Des outils de démonstration automatique ont été appliqués sur des algorithmes de contrôle de conformité manipulant des listes d'événements. Il s'agit de Spike [21], qui se situe dans la tradition des systèmes de réécriture et automatise la découverte d'invariants inductifs, et de PVS [108, 107], qui met en œuvre des procédures de décision puissantes sous le contrôle d'une démarche interactive. Les résultats sont décrits dans [128]. Pour maximiser les opportunités d'automatisation , la spécification a été rédigée en logique du premier ordre. Cependant, la preuve relatée a demandé une intervention humaine importante.

4.5 Synthèse des invariants avec GAP et HyTech

4.5.1 GAP

GAP [50] est une mise en œuvre de la procédure de Revesz [123] développée à l'ENS-DMI par Julian Richardson et Laurent Fribourg. Elle prend en entrée un programme Datalog [6] mettant en relation des entiers relatifs et calcule une formule finie caractérisant son plus petit point fixe. Les contraintes arithmétiques doivent être de la forme $X+c<Y$, $X+c\leq Y$, $c\leq X$, $X=Y$ ou $X=c$, où X, Y désignent des variables et c une constante. Elle procède par chaînage avant en ordonnant les contraintes, la terminaison est garantie par le lemme de Dickson.

En transcrivant sous cette forme le produit de la relation de transition de l'algorithme de conformité B' avec une relation de transition d'observation appropriée, Laurent Fribourg a engendré la formule caractérisant les invariants de cet ensemble et vérifié que ceux de la figure 4.5 en font partie [49].

Deux astuces méritent d'être notées. La première est spécifique à la procédure de décision de Revesz : les contraintes arithmétiques permises excluent les termes additifs qui apparaissent naturellement comme $s+\tau_2$. La vérification a donc porté sur une variante de B' où les additions sont éliminées, mais dont la correction implique celle de B'.

La seconde idée est assez classique et de portée plus générale, elle consiste à éliminer la quantification sur t dans les invariants $\mathrm{I_{tfs}}$, $\mathrm{I_{Ub1}}$, $\mathrm{I_{Ub2}}$ et $\mathrm{I_{Ub3}}$ en considérant t comme un paramètre. La relation d'observation est simplement la projection de la relation de transition de la fonction Approx sur une date cible t choisie arbitrairement. La mise à jour à effectuer porte alors sur une valeur simple Approx_t et non sur une fonction $\lambda t\,\mathrm{Approx}(t)$, ce qui permet de rester au premier ordre.

[6] Datalog est la variante de Prolog où les termes sont d'arité nulle. Par exemple chemin est un programme Datalog mais pas conc.

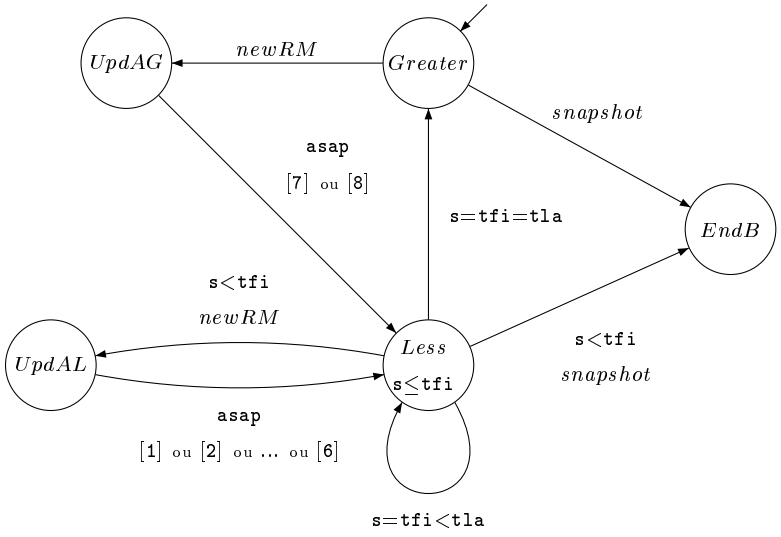

FIG. 4.9 – Modèle de B' en HyTech (source : LSV [25])

4.5.2 Hytech

La même idée a été utilisée avec HyTech [62], un outil de *model checking* sur des automates temporisés paramétriques provenant de l'équipe de T. Henzinger à l'université de Berkeley [25]. Les paramètres ont été utilisés pour travailler sur des représentations symboliques de τ_2 et τ_3 (en Uppaal il fallait prendre des instances numériques qui, pour des valeurs réalistes, contribuaient à l'explosion combinatoire). L'avantage de HyTech sur GAP est d'autoriser des sommes dans les contraintes arithmétiques, ce qui a permis de travailler sur le modèle schématisé en figure 4.9 qui est beaucoup plus proche de l'algorithme B' (les variables et les affectations ne sont pas explicitées). En revanche la terminaison n'est pas garantie, mais dans le cas de l'ABR les calculs ont convergé.

4.5.3 Mise en question de ce modèle

Le modèle précédent comporte une composante qui n'existe pas dans la norme : la « place » qui peut prendre les valeurs *Less*, *Greater*, *UpdAg*, *UpdAL*, et *EndB*. Ce n'est pas neutre car il ne s'agit pas seulement d'une caractérisation de l'état du système mais bien d'une composante de contrôle qui est utilisée pour prendre des décisions.

Les valeurs *UpdAg* et *UpdAL* sont bénignes, elles représentent des états fugitifs rendus nécessaires par une limitation du langage d'entrée de HyTech,

qui oblige à séparer la réception d'une cellule et la réaction de mise à jour. A vrai dire, on pourrait critiquer la présentation de l'algorithme faite dans les figures 4.3 et 4.4 pour la raison inverse : dans la norme I.371, les affectations sont séquentialisées, faisant apparaître des états intermédiaires qui ont disparu avec les affectations simultanées. Dans tous les cas la distance entre le modèle et la norme reste très modeste.

La valeur *EndB* est plus étrange. Rappelons que cet état provient de la projection de Approx sur sa seconde composante dans une perspective d'automatisation. Cela est possible car $\mathrm{Approx}(n,t)$ et $\mathrm{Approx}(n,t')$ sont indépendants si $t \neq t'$. Mais l'utilisation de *EndB* transforme un système vivant en un système qui sera bloqué lorsque la date d'observation cible sera atteinte. On se convainc intuitivement qu'il s'agit d'un artifice de vérification sans conséquence réelle.

En fait il est possible d'adapter la preuve Coq exposée au 4.3 à la preuve par invariants inductifs de [26] en supprimant *UpdAg*, *UpdAL* et *EndB*. Mais on ne peut se passer de *Less* et *Greater*, il y a donc introduction d'un booléen supplémentaire dans B' (sans lien clair avec `ctl`).

C'est pour éviter cette composante d'état *ad-hoc* que `utd` a été introduit dans [100], puis `stb` dans ce document. L'approche indiquée au 4.2.2 semble s'étendre à un système quelconque.

4.6 En guise de conclusion : le projet Calife

4.6.1 Complémentarité des approches précédentes

Les expériences relatées ci-dessus ont illustré – ou confirmé, s'il en était besoin – la complémentarité entre la vérification automatique et la vérification interactive.

Une logique puissante nous a donné les éléments pour exprimer l'énoncé à démontrer sous une forme convaincante (qui est particulièrement simple dans le cas de l'ABR) et pour formaliser la démonstration. Cela nous place d'emblée dans un cadre indécidable où il est inévitable que les lignes essentielles de la démonstration soient issues de l'intervention humaine. Mais les buts pouvant être résolus automatiquement sont assez simples (cela varie suivant les outils et les versions), ce qui conduit à un travail de décomposition non négligeable.

L'utilisation d'outils automatiques a automatisé la recherche d'étapes de démonstration beaucoup plus consistantes. Cela demande de reconnaître une classe de problèmes décidables pertinente, ainsi qu'un travail conceptuel important lors de l'étape de modélisation de façon à mettre le problème initial sous la forme adéquate. La question de la fidélité de cette dernière se pose alors avec acuité. Mais on peut très bien envisager d'étudier les abstractions effectuées au moyen d'un assistant interactif.

4.6.2 Vers une combinaison des approches

Le projet RNRT Calife[7] s'inscrit dans la mouvance actuelle des méthodes formelles visant à intégrer techniques automatiques et interactives. Il est organisé en six sous-projets, allant de travaux fondamentaux sur les techniques de spécification et vérification à des applications liées aux problèmes de qualité de service dans les télécommunications.

Sur le plan fondamental, un nouveau noyau logique de Coq est mis au point, intégrant de nouveaux résultats sur la réécriture à l'ordre supérieur et la modularité afin d'améliorer la puissance d'expression du système. Concernant l'automatisation des démonstrations, des techniques pour améliorer l'efficacité de procédures de décision au moyen de la *réflexion* sont développées (il s'agit de remplacer un procesus déductif par l'évaluation d'un λ-terme). C'est dans ce cadre qu'a été conçu une interface permettant de déléguer, en toute sécurité, des buts exprimés dans une théorie équationnelle à Elan, un moteur de réécriture très performant. Un autre axe concerne les techniques d'abstraction, qui permettent de réduire la complexité de la recherche des démonstrations. Les travaux sur le test démarrés dans FORMA s'y poursuivent également.

Plus près des applications, un modèle général pour des systèmes temporisés a été défini. Pour l'essentiel, il reprend le modèle classique de Alur et Dill en ne conservant que la sémantique d'exécution (avec la distinction entre transitions discrètes et continues), et en levant les contraintes sur le type des états qui avaient été introduites en vue de résultats de décidabilité. De la sorte, il devient possible de modéliser un système réelle librement et de raisonner formellement sur les abstractions effectuées lorsqu'on se ramène à un modèle restreint à dessein [24]. Des bibliothèques Coq et Isabelle formalisant ce document sont disponibles [29], les prochaines versions bénéficieront des nouveautés introduites dans les sous-projets plus fondamentaux.

Ces bibliothèques ont été expérimentées sur le contrôle de conformité ABR. Les applications visées actuellement portent sur des protocoles de diffusion (*multicast*) issus de l'IETF.

[7] Les partenaires de ce projet sont le LaBRI (université de Bordeaux), le LSV (ENS Cachan), le LRI (université de Paris-Sud), l'INRIA Rocquencourt et Lorraine, CRIL Technology FTR&D et, durant la première année, Alcatel.

Annexe A

Equivalence d'analyseurs

Au chapitre 2 nous avons indiqué comment dériver le programme Prolog d'analyse syntaxique (2.54) - (2.58) à partir d'une formulation fonctionnelle exprimée par (2.59) - (2.63). Cette annexe indique les étapes de la démonstration formelle de l'équivalence entre ces deux formulations. Le script complet est disponible sur requête à l'auteur.

Mutual Inductive exp : Set :=
| Cst : exp
| $Moins$: $exp \rightarrow exp \rightarrow exp$
| $Mult$: $exp \rightarrow exp \rightarrow exp$
| $Puiss$: $exp \rightarrow exp \rightarrow exp$.

Version fonctionnelle

Definition ide := $[x : exp]x$.

Inductive kf : $(exp \rightarrow exp) \rightarrow (exp \rightarrow exp) \rightarrow (exp \rightarrow exp) \rightarrow$ Set :=
| Fpu : $(kx, ky, kz : exp \rightarrow exp)\ (kf\ kx\ ky\ kz) \rightarrow$
$(kf\ [x](kx\ (Puiss\ x\ Cst))\ ky\ kz)$
| Fmu : $(kx, ky, kz : exp \rightarrow exp)\ (kf\ kx\ ky\ kz) \rightarrow$
$(kf\ ide\ [y](ky\ (Mult\ y\ (kx\ Cst)))\ kz)$
| Fmo : $(kx, ky, kz : exp \rightarrow exp)\ (kf\ kx\ ky\ kz) \rightarrow$
$(kf\ ide\ ide\ [z](kz\ (Moins\ z\ (ky\ (kx\ Cst)))))$
| $Feps$: $(kf\ ide\ ide\ ide)$.

Inductive $pexpf$: $exp \rightarrow$ Set :=
| Pf : $(kx, ky, kz : exp \rightarrow exp)\ (kf\ kx\ ky\ kz) \rightarrow (pexpf\ (kz\ (ky\ (kx\ Cst))))$.

Programme Prolog dérivé selon 2.5.2

Inductive ke : $exp \rightarrow exp \rightarrow exp \rightarrow exp \rightarrow exp \rightarrow exp \rightarrow$ Set :=

```
| Kpu : (x1, x2, y1, y2, z1, z2 : exp)
        (ke (Puiss x1 Cst) x2 y1 y2 z1 z2) → (ke x1 x2 y1 y2 z1 z2)
| Kmu : (x, x2, y1, y2, z1, z2 : exp)
        (ke Cst x2 (Mult y1 x2) y2 z1 z2) → (ke x x y1 y2 z1 z2)
| Kmo : (x, y, x2, y2, z1, z2 : exp)
        (ke Cst x2 x2 y2 (Moins z1 y2) z2) → (ke x x y y z1 z2)
| Keps : (x, y, z : exp) (ke x x y y z z).

Inductive pexp : exp → Set :=
| Pe : (x, y, e : exp) (ke Cst x x y y e) → (pexp e).
```

Équivalence entre kf et ke

```
Theorem kf_ke :
  (kx, ky, kz : exp → exp) (kf kx ky kz) →
  (x, y, z : exp) (ke x (kx x) y (ky y) z (kz z)).

Inductive trouve_ke [x1, x2, y1, y2, z1, z2 : exp] : Set :=
  Exke : (kx, ky, kz : exp → exp) (kf kx ky kz) →
         (kx x1) = x2 → (ky y1) = y2 → (kz z1) = z2 →
         (trouve_ke x1 x2 y1 y2 z1 z2).

Theorem ke_kf :
  (x1, x2, y1, y2, z1, z2 : exp) (ke x1 x2 y1 y2 z1 z2) →
  (trouve_ke x1 x2 y1 y2 z1 z2).
```

Équivalence entre les parseurs

```
Theorem pexpf_pexp : (e : exp) (pexpf e) → (pexp e).

Theorem pexp_pexpf : (e : exp) (pexp e) → (pexpf e).
```

Annexe B

Logique linéaire

Cette annexe a pour but de rappeler les notions de logique linéaire utilisées à la fin du chapitre 2 ainsi que les notations utilisées.

La logique linéaire [52] est une logique constructive due à Jean-Yves Girard. En tant que calcul, elle manipule primitivement des ressources devant être consommées exactement une fois. Si f est une « fonction linéaire » de A vers B, (son type est noté $A \multimap B$), et si x est une ressource de type A, on peut appliquer f à x et obtenir une ressource de type B. En termes logiques, cela s'exprime en démontrant le séquent $A, A \multimap B \vdash B$. Dans le processus, f et x sont nécessairement consommées : on ne peut pas démontrer $A, A \multimap B \vdash A \wedge B$ ou $A, A \multimap B \vdash A \multimap B$ par exemple.

Si f utilise deux fois son argument, il faut l'expliciter dans son type qui devient $A \multimap A \multimap B$. Le typage est donc beaucoup plus précis que d'habitude, mais on a aussi besoin de typer des fonctions traditionnelles utilisant leur argument un nombre indéterminé de fois. Le type d'une ressource inépuisable de A est noté $!A$, l'opérateur de répétition « ! » est dit exponentiel. Un habitant de $!A$ peut être vu comme une valeur traditionnelle de type A, et on retrouve la flèche usuelle $A \to B$ en écrivant $!A \multimap B$. Cette décomposition de la flèche intuitionniste est historiquement issue d'une étude de la sémantique dénotationnelle du λ-calcul typé.

Dans le calcul des séquents correspondant, l'usage des règles de contraction et d'affaiblissement (respectivement duplication et effacement en lecture ascendante) est donc strictement contrôlé par les exponentielles. Il y a alors deux façons non équivalentes de définir la conjonction et la disjonction, suivant la façon de gérer le contexte (cela apparaîtra dans les règles de la figure B.1) : la forme multiplicative, qui sépare les ressources et la forme additive, qui les partage.

Les notions correspondant aux structures de données usuelles, produit et somme, sont respectivement représentées par la conjonction multiplicative $A \otimes B$ (juxtaposition d'une ressource de type A et d'une ressource de type B) et la disjonction additive $A \oplus B$.

Il est possible de consommer un produit comme suit :

$$A \otimes B, A \multimap B \multimap C \vdash C.$$

Pour consommer une somme, il faut un analogue de **if** ... **then** ... **else** ..., c'est la conjonction additive & qui joue ce rôle :

$$A \oplus B, (A \multimap C) \& (B \multimap C) \vdash C.$$

On peut voir $A \& B$ comme la superposition d'une ressource de type A et d'une ressource de type B. À partir d'un habitant de $A \& B$ on peut retrouver (et consommer) au choix l'habitant sous-jacent de A ou de B, mais seulement l'un des deux ! On a les théorèmes :

$$A \& B, A \multimap C \vdash C$$
$$A \& B, B \multimap C \vdash C$$
$$A \& B, (A \multimap C) \oplus (B \multimap C) \vdash C$$

La disjonction multiplicative $A \parr B$ est une version symétrique de $A \multimap B$ qui apparaît naturellement en logique linéaire classique : un séquent de la forme $A_1, A_2 \ldots \vdash B_1, B_2 \ldots$ est équivalent à $\vdash (A_1 \otimes A_2 \ldots) \multimap (B_1 \parr B_2 \ldots)$. On dispose d'une absurde $\bot$ et d'une « négation » linéaire *involutive* $A^\bot$ équivalente à $A \multimap \bot$ vérifiant $A, A^\bot \vdash$. Elle permet d'écrire les procédés de consommation ci-dessus de façon plus élégante :

$$A \oplus B, A^\bot \& B^\bot \vdash$$
$$A \& B, A^\bot \oplus B^\bot \vdash$$
$$A \otimes B, A^\bot \parr B^\bot \vdash$$
$$A \parr B, A^\bot \otimes B^\bot \vdash$$

Les connecteurs $\otimes$, $\oplus$, $\&$ et $\parr$ ont des éléments neutres qui sont respectivement $\mathbf{1}$, $\mathbf{0}$, $\top$ et $\bot$. Les quantificateurs du premier ordre se comportent de manière additive. On a aussi une équivalence entre $!(A \& B)$ et $!A \otimes !B$, d'où l'appellation « exponentielle ».

Pour notre propos au 2.5, le fragment intuitionniste sans exponentielles est suffisant. La figure B.1 donne les règles en calcul des séquents. Une syntaxe pour les termes propositionnels correspondants est donnée en figure B.2. Pour les règles d'introduction à gauche de $\otimes$ et & nous employons une syntaxe par filtrage. Les séquents manipulés sont donc de la forme $\Gamma \vdash t : A$ où Γ est une séquence de déclarations $p : T$, où p est un motif, c'est-à-dire une variable, ou un couple $x.y$ ou $x \,[]\, y$ où x et y sont des motifs.

$$\frac{}{A \vdash A} \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\text{CUT}$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\otimes_L \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\otimes_R$$

$$\frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\multimap_L \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\multimap_R$$

$$\frac{\Gamma, A \vdash C}{\Gamma, A\&B \vdash C}\&_{L_1} \quad \frac{\Gamma, B \vdash C}{\Gamma, A\&B \vdash C}\&_{L_2} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A\&B}\&_R$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\oplus_L \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\oplus_{R_1} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\oplus_{R_2}$$

$$\frac{\Gamma, [x := t]A \vdash C}{\Gamma, \forall x\, A \vdash C}\forall_L \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x\, A}\forall_R$$

$$\frac{\Gamma, A \vdash C}{\Gamma, \exists x\, A \vdash C}\exists_L \qquad \frac{\Gamma \vdash [x := t]A}{\Gamma \vdash \exists x\, A}\exists_R$$

$$\frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A}\mathbf{1}_L \qquad \frac{}{\vdash \mathbf{1}}\mathbf{1}_R$$

$$\frac{}{\bot \vdash}\bot_L \qquad \frac{\Gamma \vdash}{\Gamma \vdash \bot}\bot_R$$

$$\frac{}{\Gamma, \mathbf{0} \vdash A}\mathbf{0}_L \qquad \frac{}{\Gamma \vdash \top}\top_R$$

FIG. B.1 – Calcul des séquents linéaire intuitionniste

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma \vdash t : A \quad \Delta, x : A \vdash s : B}{\Gamma, \Delta \vdash [x := t]s : B}$$

$$\frac{\Gamma, x : A, y : B \vdash t : C}{\Gamma, xy : A \otimes B \vdash t : C} \qquad \frac{\Gamma \vdash t : A \quad \Delta \vdash s : B}{\Gamma, \Delta \vdash ts : A \otimes B}$$

$$\frac{\Gamma \vdash t : A \quad \Delta, y : B \vdash s : C}{\Gamma, \Delta, f : A \multimap B \vdash [y := (ft)]s : C} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda\!\!\!/ x\, t : A \multimap B}$$

$$\frac{\Gamma, x : A \vdash t : C}{\Gamma, x \,[\!]\, y : A\&B \vdash t : C} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash s : B}{\Gamma \vdash t \,[\!]\, s : A\&B}$$

$$\frac{\Gamma, y : B \vdash t : C}{\Gamma, x \,[\!]\, y : A\&B \vdash t : C}$$

$$\frac{\Gamma, x : A \vdash t : C \quad \Gamma, y : B \vdash s : C}{\Gamma, z : A \oplus B \vdash (\,\mathbf{case}\, z\, \mathbf{of}\, \mathrm{inl}(x) \Rightarrow t \mid \mathrm{inr}(y) \Rightarrow s) : C} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathrm{inl}(t) : A \oplus B}$$

$$\frac{\Gamma \vdash s : B}{\Gamma \vdash \mathrm{inr}(s) : A \oplus B}$$

FIG. B.2 – termes linéaires intuitionnistes

Bibliographie

[1] Martin Abadi. An Axiomatization of Lamport's Temporal Logic of Actions. Technical Report 65, Digital Equipment Corporation, Systems Research Centre, October 1990.

[2] J-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[3] J.-R. Abrial. Développement de l'algorithme ABR. Personal communication, 1999.

[4] Jean-Raymond Abrial. Event driven sequential program construction. École Jeunes chercheurs en programmation, March 2000.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, April 1994.

[6] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[7] A. Arnold. MEC : A system for constructing and analysing transition systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 117–132, Berlin, June 1990. Springer Verlag.

[8] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET « Les mathématiques de l'informatique »*, pages 35–68, 1982.

[9] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. études et recherches en informatique. Masson, 1992.

[10] André Arnold and Damian Niwiński. *Rudiments of the mu-calculus*. Elsevier, 2001.

[11] Thomas Arts and Mads Dam. Verifying a distributed database lookup manager written in erlang. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the development of Computing Systems*, volume 1708-1709 of *LNCS*, pages 682–700. Springer Verlag, 1999.

[12] J-M. Ayache, R. Groz, C. Jard, and J.-F. Monin. Des outils pour Estelle : du prototype au poste industriel. In *Journées francophones pour l'informatique*, janvier 1987. Liège.

[13] B. Barras. *Auto-valisation d'un systèmes de preuves avec familles inductives.* Thèse de doctorat, Université de Paris 7, 1999.

[14] Patrick Bellot, Jean-Philippe Cottin, and Jean-François Monin. Développement et validation de logiciels. Méthodes formelles. *Techniques de l'Ingénieur*, (12) :1–12, 1995.

[15] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. UPPAAL : a tool suite for the automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1996.

[16] Gérard Berry, Philippe Couronné, and Georges Gonthier. Programmation synchrone des systèmes réactifs : le langage ESTEREL. *Techniques et Sciences Informatiques*, 6(4) :305–315, 1987.

[17] Richard Bird and Philip Wadler. *Introduction to functional programming.* Prentice Hall, 1988.

[18] Nikolaj Bjorner, Zohar Manna, Henny Sipma, and Tomás Uribe. Deductive Verification of Real-time Systems using STeP. *Theoretical Computer Science*, 253 :27–60, 2001.

[19] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN : A logical framework based on computational systems. In *Proc. of the First Int. Workshop on Rewriting Logic*, volume 4. Elsevier, 1996.

[20] Adel Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE : An automatic theorem prover. In *1st Int. Conf. on Logic Programming and Automated Reasoning*, volume 624 of *LNAI.* Springer Verlag, 1992.

[21] Adel Bouhoula and Michaël Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2) :189–235, 1995.

[22] P. Bouyer, C. Dufourd, E. Fleury, , and A. Petit. Are timed automata updatable ? In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 464–479, Chicago, IL, USA, July 2000. Springer Verlag.

[23] Julian Charles Bradfield. *Verifying Temporal Properties of Systems.* Progress in Theoretical Computer Science. Birkhäuser, 1992.

[24] B. Bérard, P. Castéran, E. Fleury, L. Fribourg, J-F. Monin, C. Paulin, A. Petit, and D. Rouillard. Automates temporisés calife. Fourniture RNRT Calife F1.1, juillet 2000.

[25] B. Bérard and L. Fribourg. Automated verification of a parametric real-time program : the ABR conformance protocol. In *CAV'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999. To appear.

[26] B. Bérard, L. Fribourg, F. Klay, and J.-F. Monin. A compared study of two correctness proofs for the standardized algorithm of ABR conformance. *Formal Methods in System Design*, 2002. to appear ; also tech report ENS Cachan LSV-99-7.

[27] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of lambda-prolog : Prolog/mali. In *ILPS Workshop : Implementation Techniques for Logic Programming Languages*, 1994.

[28] P. Castéran. Pro[gramm,v]ing with continuations, a development in Coq. Coq contribution, 1993.

[29] P. Castéran, E. Freund, C. Paulin, and D. Rouillard. Bibliothèques Coq et Isabelle-HOL pour les systèmes de transitions et les p-automates. Fourniture RNRT Calife F5.3, juillet 2000.

[30] Pierre Castéran and Davy Rouillard. Reasoning about parametrized automata. In *RTS'2000*, Paris, 2000.

[31] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science – LICS'96*, pages 264–275, New Brunswick, New Jersey, 27–30 1996. IEEE Computer Society Press.

[32] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science Classics. Academic Press, 1973.

[33] Philippe Chavin and Jean-François Monin. Using Coq for specifying an invoicing system. In H. Habrias M. Allemand, C. Attiogbé, editor, *Proceedings of Comparing Systems Specification Techniques*, pages 19–34, Nantes, France, 1998. IRIN, Université de Nantes.

[34] Philippe Chavin and Jean-François Monin. An Abstract and constructive specification in Coq. In Marc Frappier and Henri Habrias, editors, *Software Specification Methods*, FACIT, chapter 13. Springer Verlag, 2000.

[35] K.L. Clark and S.-A. Tarnlund. A First Order Theory of Data and Programs. *Information Processing*, 77 :939–944, 1977.

[36] E.M Clarke, E. M. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications : a practical approach. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, 1983.

[37] M. Condillac. *Prolog, fondements et applications*. Dunod, 1986.

[38] SPECS RACE 1046 consortium. Definition of MR and CRL Version 2.1, SPECS-Semantics and Analysis, 1990.

[39] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation (formerly Information and Control)*, 76 :95–120, February/March 1988.

[40] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[41] Thierry Coquand and Gérard Huet. A theory of constructions. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *LNCS*. Springer Verlag, 1985.

[42] Guy Cousineau and Michel Mauny. *Approche fonctionnelle de la programmation*. Ediscience, 1995.

[43] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Technical report, SRI, Menlo Park, CA, April 1995. Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida.

[44] G. Doumenc and J.-F. Monin. The parallel abstract machine : A common execution model for FDTs. In *FME'93 : Industrial-Strength Formal Methods*, volume 670 of *LNCS*. Springer Verlag, 1993.

[45] F. du Castel, editor. *Les télécommunications*. Roger Levrault Int., 1993. contribution au chapitre informatique.

[46] Herbert B. Enderton. *Elements of set theory*. Academic Press, 1977.

[47] J.-C. Filliâtre. *Preuves de programmes impératifs en théorie des types*. Thèse de doctorat, Université de Paris-Sud, 1999.

[48] R. W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Sciences*, pages 52–66, 1967.

[49] L. Fribourg. A closed-form evaluation for extended timed automata. Research Report LSV-98-2, Lab. Specification and Verification, ENS de Cachan, Cachan, France, March 1998. 17 pages.

[50] L. Fribourg and J. Richardson. Symbolic Verification with Gap-Order Constraints. Technical Report LIENS-96-3, Ecole Normale Sipérieure, Paris, February 1996.

[51] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[52] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50 :1–102, 1987.

[53] Jean-Yves Girard. A new constructive logic : classical logic. *Mathematical Structures in Computer Science*, 1 :225–296, 1991.

[54] M. J. C. Gordon. HOL, a Proof Generating System for Higher-Order Logic. In C. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[55] M. J. C. Gordon and T. F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[56] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF : A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[57] T. Griffin. A formulae-as-types notion of control. In *Proc. 17th ACM Symp. on Principles of Programming Languages*. ACM, Orlando, 1990.

[58] Jan Friso Groote. The syntax and semantics of timed μCRL. In *216*, page 42. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 30 1997.

[59] R. Groz, M. Litime, J-F. Monin, M. Phalippou, C. Hervé, P. Riou, P. Cousin, and J. Le Compagnon. Elements of a C.A.S.E. environment for defining and generating test suites. In *Int. Work. on Prot. Test Syst*, Berlin, 1989.

[60] R. Groz and J.-F. Monin. Current applications of formal methods in France Telecom-CNET. British-french meeting on formal methods organized by Dassault Electronique, St Quentin en Yvelines, february 1995.

[61] P. R. Halmos. *Naive Set Theory*. Van Nostrand, Princeton, N.J., 1960.

[62] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH : A model checker for hybrid systems. *Lecture Notes in Computer Science*, 1254 :460–463, 1997.

[63] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[64] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant, a tutorial, V6.3. Technical report, INRIA Rocquencourt et CNRS-ENS Lyon, 1999.

[65] Gérard Huet. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison Wesley, 1990.

[66] International Organization for Standardization, Geneve. *Information Processing Systems - Open Systems Interconnection - Guidelines for the Application of ESTELLE, LOTOS and SDL*. ISO/IEC TR 10167.

[67] ITU-T. *Traffic control and congestion control in B-ISDN*. Recommendation I.371.1.

[68] C. Jard, R. Groz, and J.-F. Monin. Véda : a software Simulator for the Validation of Protocol Specifications. In L. Csaba, K. Tarnay, and T. Szentivanyi, editors, *COMNET'85, Computer Network Usage : Recent Experiences*, Budapest, 1986. North Holland.

[69] C. Jard, J.-F. Monin, and R. Groz. Experience in implementing X250 (a CCITT subset of Estelle). In M. Diaz, editor, *IFIP Workshop V on Protocol Specification Testing and Verification*. North Holland, 1986.

[70] C. Jard, J.-F. Monin, and R. Groz. Development of Veda, a Prototyping Tool for Distributed Algorithms. *IEEE Transactions on Software Engineering*, 14(3) :339–352, march 1988.

[71] Yonit Kesten, Zohar Manna, and Amir Pnueli. Verifying clocked transition systems. In *Hybrid Systems*, pages 13–40, 1995.

[72] Robert A. Kowalski. Algorithm = Logic + Control. *CACM*, 22(7) :424–436, 1979.

[73] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, May 1994.

[74] X. Leroy and P. Weis. *Le langage Caml.* InterEditions, 1993.

[75] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, 1990.

[76] Xavier Leroy. Programmation du système Unix en Caml light. RT 147, INRIA, Rocquencourt, 1993.

[77] J. W. Lloyd. *Foundations of Logic Programming, Second Extended Edition.* Springer-Verlag, 1993.

[78] Zhaohui Luo and Robert Pollack. LEGO proof development system : User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, 1992.

[79] Zohar Manna and Amir Pnueli. Clocked Transition Systems. In *Logic and Software Engineering*, pages 3–42. World Scientific Pub., 1996. Also Stanford CSD Technical Report STAN-CS-TR-96-1566.

[80] Per Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory. Bibliopolis, Napoli, 1984.

[81] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming*, LNCS 225, pages 448–462. Springer-Verlag, 1986.

[82] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994. IEEE Computer Society Press.

[83] R. Milner. A proposal for Standart ML. In *ACM Conf. on Lisp and Functional Programming*, 1987.

[84] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[85] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87 :209–220, 1991.

[86] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1) :55–92, July 1991.

[87] Jean-François Monin. Écriture d'un compilateur réel en Prolog. In S. Bourgault and M. Dincbas, editors, *Séminaire sur la programmation en logique*, Plestin, 1984.

[88] Jean-François Monin. Terminaison distribuée et groupes commutatifs. In J-P. Verjus and G. Roucairol, editors, *Parallélisme, communication et synchronisation.* éditions du CNRS, 1985.

[89] Jean-François Monin. A compiler written in Prolog : the Véda experience. In Deransart, Lorho, and Maluszynski, editors, *Programming Languages Implementation and Logic Programming*, number 348 in LNCS, Orléans, 1989. Springer Verlag.

[90] Jean-François Monin. *Programmation en logique et compilation de protocoles : le simulateur Véda.* Thèse d'université, Univ. de Rennes I, jan, 1989.

[91] Jean-François Monin. Langages, méthodes et outils pour la spécification et la programmation. Technical report, CNET, 1990.

[92] Jean-François Monin. Real-size compiler writing using Prolog with arrows. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 188–201, Paris, France, 1991. The MIT Press.

[93] Jean-François Monin. Certification de programmes avec exceptions : une approche modulaire. Actes des journées du GDR programmation, Lille, septembre 1994.

[94] Jean-François Monin. Extracting Programs with Exceptions in an Impredicative Type System. In B. Möller, editor, *Mathematics of Program Construction*, volume 947 of *LNCS*. Springer Verlag, 1995.

[95] Jean-François Monin. *Comprendre les Méthodes formelles, panorama et outils logiques.* CTST. Masson, 1996. Préface de G. Huet.

[96] Jean-François Monin. Exceptions considered harmless. *Science of Computer Programming*, 26 :179–196, 1996.

[97] Jean-François Monin. Formal methods in CNET, 15 years later. In *FemSys, Workshop in Formal Design of Safety Critical Embedded Systems*, Munich, Germany, april 1997.

[98] Jean-François Monin. Proving a real time algorithm for ATM in Coq. In E. Gimenez and C. Paulin-Mohring, editors, *Types for Proofs and Programs*, volume 1512 of *LNCS*, pages 277–293. Springer Verlag, 1998.

[99] Jean-François Monin. *Introduction aux méthodes formelles.* CTST. Hermès, 2000. Préface de G. Huet.

[100] Jean-François Monin. Proving the Correctness of the Standardized Algorithm for ABR Conformance. *Formal Methods in System Design*, 17(3) :221–243, december 2000.

[101] Jean-François Monin. *Understanding Formal Methods.* Springer Verlag, 2002. Translated with the help of M. Hinchey.

[102] Jean-François Monin and Francis Klay. Correctness Proof of the Standardized Algorithm for ABR Conformance. Note technique en latence, France Télécom, CNET, 1997.

[103] Jean-François Monin and Francis Klay. Correctness Proof of the Standardized Algorithm for ABR Conformance. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, volume 1708 of *LNCS*, pages 662–681. Springer Verlag, 1999.

[104] Jean-François Monin and Joseph Sifakis. *Application des techniques formelles au logiciel*, volume 20 of *Arago*, chapter II, Eléments de classification des méthodes formelles. OFTA, 1997.

[105] C. Murthy. An evaluation semantics for classical proofs. In *Proc. IEEE Symp. on Logic in Computer Science*. IEEE, 1991.

[106] Gopalan Nadathur and Dale Miller. An overview of λ PROLOG. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seatle, 1988. ALP, IEEE, The MIT Press.

[107] S. Owre, J. M. Rushby, and N. Shankar. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, 1993.

[108] S. Owre, J.M. Rushby, and N. Shankar. PVS : a prototype verification system. In *11th Conf. on Automated Deduction (CADE), LNAI 607*, pages 748–752. Springer Verlag, 1992.

[109] M. Parigot. Lambda-mu-calculus : an algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Logic Programming and Automated Reasoning : International Conference LPAR '92 Proceedings, St. Petersburg, Russia*, pages 190–201, Berlin, DE, 1992. Springer-Verlag.

[110] C. Paulin-Mohring. Extracting Fw's programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, January 1989.

[111] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, January 1989.

[112] Christine Paulin-Mohring. Inductive definitions in the system Coq : Rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664, pages 328–345. Springer-Verlag, Berlin, 1993.

[113] L. C. Paulson. Isabelle : the 700 next theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[114] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[115] Lawrence C. Paulson. *Isabelle : A Generic Theorem Prover*. Springer LNCS 828, 1994.

[116] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science (FOCS'77)*, pages 46–57, Providence, RI, USA, 1977.

[117] Gérald Point and Antoine Rauzy. Altarica : langage de modélisation par automates à contraintes. In J.J. Lesage, editor, *Modélisation des systèmes réactifs*, page 81 et s. Hermès, 1999.

[118] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. International Series in Computer Science. Prentice Hall, 1991.

[119] D. Prawitz. *Natural Deduction*. Almqvist and Wiksell, 1965.

[120] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. Int. Symp.on Programming*, volume 137 of *LNCS*, pages 337–351. Springer Verlag, 1982.

[121] Christian Queinnec. The Influence of Browsers on Evaluators. version préliminaire présentée aux JFLA, Pontarlier, 2001.

[122] Christophe Rabadan. L'ABR et sa conformité. NT DAC/ARP/034, CNET, 1997.

[123] P. Z. Revesz. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1) :117–149, August 1993.

[124] J.A. Robinson. A machine oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[125] J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Research Report 1795, INRIA-Lorraine, nov. 1992.

[126] John Rushby. Verification diagrams revisited : Disjunctive invariants for easy verification. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification, CAV'2000*, volume 1855 of *LNCS*, pages 508–520. Springer Verlag, Chicago, July 2000.

[127] M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical Verification of a Generic Incremental ABR Conformance Algorithm. Technical Report RT-3794, INRIA, 1999.

[128] M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical Verification of an Ideal ABR Conformance Algorithm. In *Conference on Computer Aided Verification*, number 1855 in LNCS, Chicago, USA, 15–19 July 2000. Springer Verlag.

[129] Philippe Schnoebelen, editor. *Vérification de logiciels*. Vuibert, 1999.

[130] Natarajan Shankar. personal communication, july 2001.

[131] J. M. Spivey. *The Z notation : A reference manual*. International Series in Computer Science. Prentice Hall, 1989.

[132] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[133] C. Strachey and C. Wadsworth. Continuations : A mathematical semantics for handling full jumps. Technical Report PRG-11, Programming Research Group, Oxford University, Oxford, U.K, 1974.

[134] The Coq Development Team. The Coq Proof Assistant Reference Manual. Technical report, INRIA, 2001.

[135] Simon Thomson. *Type Theory and Functional Programming.* International Computer Science Series. Addison Wesley, 1991.

[136] P. Wadler. The essence of functional programming. In *Proceedings of POPL'92*, pages 1–14, Albuquerque, New Mexico, 1992.

[137] Benjamin Werner. *Une théorie des constructions inductives.* PhD thesis, Université Paris 7, 1994.

[138] J. B. Wordsworth. *Software Development with Z.* International Computer Science Series. Addison Wesley, 1992.

[139] W. Zhang and J-F. Monin. Résultats et expériences sur un environnement de simulation pour l'évaluation de performances à partir d'Estelle. In R. Castanet and O. Rafiq, editors, *Colloque Francophone sur l'Ingénierie des Protocoles*, Bordeaux, 1988.