Université de Provence (Aix-Marseille I)

Mémoire présenté en vue de l'obtention d'une
Habilitation à Diriger des Recherches

Des Preuves et des Programmes

Solange COUPET-GRIMAL

soutenu le 10 septembre 2004 devant le jury composé de

Monsieur	François	DENIS	Rapporteur
Monsieur	Gérard	HUET	Rapporteur
Monsieur	Jean-François	MONIN	Rapporteur
Monsieur	Roberto	AMADIO	Examinateur
Monsieur	Alain	COLMERAUER	Examinateur
Monsieur	Gilles	KAHN	Examinateur
Monsieur	Pierre	LESCANNE	Examinateur

Remerciements

Je tiens à exprimer ici toute ma reconnaissance à Gérard Huet. L'accueil qu'il m'a reservé il y a quelques années dans son groupe de travail à l'INRIA Rocquencourt, ses encouragements, l'intérêt qu'il a manifesté pour mon travail ont été pour moi déterminants. A nouveau aujourd'hui, il a accepté d'évaluer ce manuscrit.

Je remercie également très vivement François Denis et Jean-François Monin d'avoir bien voulu étudier ce document, faire des remarques et rédiger un rapport.

Je suis très honorée de la présence de Gilles Kahn dans ce jury et je l'en remercie.

J'ai eu le privilège de faire mes premiers pas en recherche sous la direction d'Alain Colmerauer. Il m'a donné le goût de l'esthétique, de l'élégance, de la concision dans le domaine des preuves et des calculs. Je suis heureuse qu'il soit présent à cette soutenance et je l'en remercie.

Dès sa nomination dans mon Université, Roberto Amadio m'a proposé des collaborations et m'a notamment amenée à m'intéresser de plus près à la sémantique. Je le remercie pour son dynamisme et sa participation à ce jury.

Je remercie aussi Pierre Lescanne pour tous nos échanges scientifiques au fil du temps et pour avoir accepté de faire partie de ce jury.

Je tiens de plus à adresser tous mes remerciements à Christine Paulin-Mohring, Yves Bertot et leurs équipes respectives du LIP à l'ENS-Lyon et de l'INRIA Sophia Antipolis dans lesquelles j'ai séjourné lors de congés sabbatiques et détachements. J'ai passé parmi eux, dans un excellent environnement scientifique, des moments propices à l'étude qui furent décisifs pour l'avancement de mes travaux.

Merci également à Robert Pasero, à Traian Muntean, à Karl Schlechta et à tous les membres de l'équipe Move, qui m'ont aidée en relisant ce mémoire, certains de mes articles, et par de fructueuses discussions.

Merci enfin à Bruno Durand pour son action aux commandes du Laboratoire d'Informatique Fondamentale de Marseille et à Chantal Ravier et Martine Quessada pour leur disponibilité, leur patience et leur efficacité.

Il y a dans les bois des arbres fous d'oiseaux

Paul Eluard

Table des matières

Des Preuves et des Programmes

1 Introduction

Le problème de la "mécanisation" du raisonnement mathématique, formulé par Hilbert, est à l'origine d'une extraordinaire révolution technologique. La naissance de l'informatique est en effet l'aboutissement d'une réflexion abstraite sur la nature de l'activité du mathématicien, d'ailleurs souvent considérée avec un certain scepticisme par celui-ci : je veux parler des travaux de Gödel, Church, Turing datant des années 30 sur la cohérence, la complétude et la décidabilité des mathématiques. Depuis, la théorie de la démonstration a mis en évidence l'essence commune des preuves et des calculs. Cette constatation explique les nombreuses retombées informatiques de la recherche en logique, elle-même nourrie des problèmes concrets posés par l'informatisation de la société.

L'interdépendance des deux domaines s'est manifestée notamment dans les premiers travaux visant à faire évoluer les ordinateurs du statut de super machines à calculer à celui de machines intelligentes, et dans l'émergence de la Programmation en Logique. Prolog fut créé au début des années 1970 par Alain Colmerauer. Ce fut avec Lisp l'un des langages de l'Intelligence Artificielle, utilisé en particulier pour le développement des moteurs d'inférences des systèmes experts ou encore le traitement automatique des langues naturelles. C'était initialement un démonstrateur de théorèmes fondé sur le principe de résolution de Robinson [Ro65]. Un programme Prolog est en fait une suite finie d'axiomes sous forme de clauses de Horn. Exécuter un tel programme revient à dériver une formule (le but) à partir des axiomes qui constituent le programme. Précisons que le modèle théorique de Prolog, donné en 1976 par Kowalski et Van Emden [KV76], a par la suite été modifié par Alain Colmerauer. Il a proposé un modèle basé sur l'algèbre des termes rationnels [Col82a] avec un algorithme d'unification sans "occur-check" (lié aux travaux de Gérard Huet [Hu76]) puis il l'a généralisé à la programmation en logique avec contraintes.

La simulation automatique, même partielle, du raisonnement humain était un programme très ambitieux. De façon plus ciblée, on s'est intéressé à la

vérification automatique de démonstrations mathématiques. Cette question s'est posée quand l'omniprésence de l'informatique dans les réalisations technologiques modernes a rendu indispensable de garantir la fiabilité des systèmes *critiques*; il s'agit de logiciels et de matériel dont les défaillances pourraient provoquer de grosses pertes économiques, voire humaines, dans des domaines comme l'avionique, l'informatique médicale, bancaire, le commerce électronique... Les méthodes de test traditionnelles se sont révélées insuffisantes et difficiles à automatiser. D'une part tester un système complexe, comme l'informatisation d'une ligne de métro par exemple, coûte très cher (plusieurs hommes/années). D'autre part, ces méthodes n'assurent en général pas que toutes les erreurs ont été détectées.

Une autre démarche consiste à spécifier au préalable le système que l'on veut vérifier dans une logique ad hoc, dont la sémantique est rigoureusement définie; puis à dériver dans cette logique les formules exprimant les propriétés de correction. Si les preuves de ces formules ne font pas nécessairement appel à des mathématiques profondes, elles sont presque toujours complexes, souvent parsemées de pièges logiques, fastidieuses et sujettes à erreurs. Des outils de démonstration automatique peuvent alors avantageusement remplacer le mathématicien. Cependant, non seulement leur rayon d'action est limité au domaine du décidable, mais ils se heurtent également à des problèmes d'explosion combinatoire. Dès lors, on ne peut éviter les preuves "à la main". Pour les raisons évoquées précédemment et dans un souci de rigueur absolue, il est souhaitable que ces preuves soient elles-mêmes vérifiées par des systèmes informatiques. Les preuves prennent ainsi la place jusqu'alors détenue par les programmes dont on souhaitait prouver la correction, en devenant elles-mêmes les objets de l'étude.

Les logiciels d'aide à la preuve sont de deux sortes. On distingue les *canevas logiques* (*logical frameworks*), méta-langages servant de plateformes communes pour spécifier divers systèmes déductifs dans le but de vérifier la validité des déductions. Une autre approche consiste à construire un système basé sur une logique donnée une fois pour toute : ce sont les *assistants de preuve*. Un petit nombre d'entre eux s'imposent actuellement dans la communauté scientifique, et parmi ceux-là Coq, issu des travaux en théorie des types de Thierry Coquand et Gérard Huet et développé à l'Inria.

Voici donc le cadre général dans lequel s'est située ma recherche, de la programmation en logique aux méthodes formelles et à la théorie des types.

2 Contribution à la Programmation en Logique

[1] Prolog n'est pas vraiment un *canevas logique*. En effet, la spécification d'une logique particulière dans le méta-langage conduit à coder au préalable la

[1] Références [1, 2, 7, 3, 21] de la liste de publications

syntaxe des termes, des formules et donc en particulier les variables logiques. Les termes du premier ordre de Prolog ne permettent pas de représenter naturellement des notions de portée ou de liaison de variables logiques. La solution consistant à représenter une variable objet par une variable Prolog et à réaliser les substitutions au moyen d'unifications est contraire à la sémantique de Prolog et génératrice d'erreurs. Par ailleurs, un codage correct des variables par des constantes du langage requiert une implémentation explicite du renommage. La méthode de De Bruijn [DB72] demande, elle, une implémentation explicite de la substitution. Une autre solution consiste à utiliser des *syntaxes abstraites d'ordre supérieur* dans lesquelles les formules logiques sont représentées par des termes du λ-calcul simplement typé.

2.1 Représentation de termes d'ordre supérieur

2.1.1 Codage comme arbres infinis rationnels

Cela m'a amenée à proposer [1, 3, 7, 21] une implantation du λ-calcul en Prolog, dans laquelle les classes d'α-équivalence des λ-termes sont codées par des arbres infinis rationnels. Les variables liées ne sont pas nommées : elles sont représentées par une "référence" à l'opérateur d'abstraction qui les lie (Fig. 1). Renommages et substitutions sont alors réalisés correctement par l'uni-

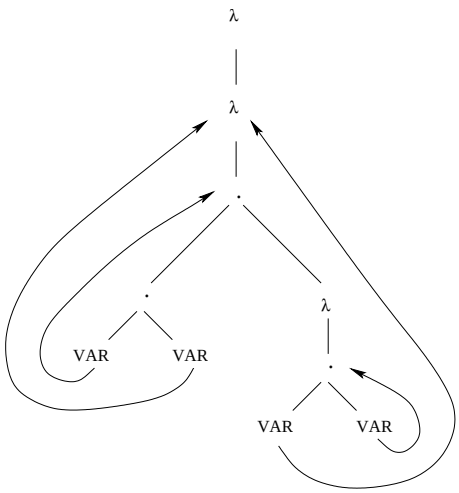

FIG. 1 – Le terme $\lambda xy\ ((y\ x)\ (\lambda y\ (x\ y)))$ sous forme d'arbre infini

fication Prolog. Ce codage est défini formellement dans [1]. On y donne un algorithme permettant d'obtenir l'arbre infini représentant un λ-terme. On y établit également un résultat d'adéquation en prouvant que deux λ-termes sont α-équivalents si et seulement s'ils ont même codage.

L'algorithme de normalisation proposé tient en quelques lignes. Il s'appuie sur les notions de *résiduel* et de *réduction complète*. La notion de *résiduel* exprime ce qu'il advient des radicaux d'un λ-terme lorsque l'on contracte l'un d'entre eux. Une *réduction complète* d'un ensemble F de radicaux d'un terme est une suite de contractions telle que chaque radical contracté est un résiduel d'un élément de F et telle que le terme obtenu après cette suite de conversions ne contient plus aucun résiduel de F ([Ch41, CF58]). L'algorithme consiste, dans une stratégie d'appel par valeur, à transformer un terme par réductions complètes successives, l'ensemble choisi étant toujours l'ensemble de tous les radicaux, et ce jusqu'à ce que l'expression obtenue ne contienne plus aucun radical. Dans le programme ci-après, le prédicat *reduire(<e,e">,nil)* est satisfait si et seulement si *e"* est obtenue à partir de *e* par réduction complète de l'ensemble des radicaux de *e*.

```
normalisation(e,e') →reduire(<e,e">,nil) recommencer(e, e", e') ;

recommencer (e,e,e) → ;
recommencer(e, e", e') → dif(e, e") normalisation(e", e') ;


reduire(<var(m), a>, s) -> dans (<var(m), a>, s) ;
reduire(<c,c> , s) -> constant(c) ;
reduire(<lambda(m),lambda(m')>,s) →reduire(<m, m'>, <var(m), var(m')>.s) ;
reduire(<lambda(m).a, m'>, s) → reduire(<a, a'>, s)
                                reduire(<m, m'>, <var(m), a'>.s) ;
reduire(<f.a, f'.a'>, s) → dif-de-lambda(f)
                           reduire(<f, f'>,s)
                           reduire(<a,a'>,s) ;
```

La correction de cet algorithme est prouvée dans [1, 21]. Reposant sur une stratégie d'appel par valeur, il ne s'applique qu'à des termes fortement normalisables. Toutefois cette limitation n'est pas gênante dans la mesure où les applications potentielles d'une telle implantation restent dans le domaine de λ-calculs simplement typés qui satisfont cette condition. C'est ainsi que, dans le cadre d'une étude sur le traitement automatique des langues naturelles, ce programme a pu être intégré dans un système d'interrogation en Français d'une banque de données.

2.1.2 Application au traitement automatique des langues naturelles

L'analogie entre analyse syntaxique et dérivation de formules est un des fondements de la recherche en traitement automatique des langues naturelles (TALN) : en décorant d'attributs du premier ordre (*exprimant des accords grammaticaux par exemple*) les règles des grammaires hors contexte, on obtient des grammaires dites logiques pouvant être directement traduites en clauses de Horn. Prolog est ainsi un produit dérivé de recherches en TALN ([Col92]).

De nombreux travaux sur la représentation de la sémantique du langage naturel en Programmation en Logique s'appuient sur la notion de *quantificateur à trois branches*, introduite par Colmerauer dans [Col82b]. Par exemple, la sémantique de la phrase *le ministre travaille* est la formule *existe(x, ministre(x), travaille(x))*. Cette approche a été reprise par beaucoup d'autres chercheurs, retravaillée en diverses extensions et solutions alternatives [DS88, DS88, GLSS92].

En m'appuyant sur la représentation des termes d'ordre supérieur exposée au paragraphe précédent, j'ai pu traiter ce problème en restant dans le cadre classique de la logique du premier ordre [1, 7, 3]. Cette approche est illustrée par la réalisation d'un système d'interrogation en Français d'une banque de données décrivant les relations européennes au XVII^{ième} siècle. Voici une présentation synthétique du langage traité.

Syntaxe et sémantique. La syntaxe du sous-ensemble du Français étudié est définie par la grammaire hors contexte du tableau 1 et le lexique du tableau 2. La sémantique (à la Montague [Mo74]) d'une phrase du langage est une formule du calcul des prédicats codée par un λ-terme. La sémantique de toute phrase dérivée d'un non terminal N est un terme obtenu par combinaison des termes sémantiques de ses composants syntaxiques. Pour chaque phrase exprimant une interrogation de la banque, le λ-terme ainsi obtenu est normalisé à l'aide de l'algorithme précédent, avant d'être évalué. De cette évaluation résultent les réponses à la requête.

Considérons par exemple la phrase *Charles épouse Thérèse*. Sa structure syntaxique est représentée sur la figure 2. Le tableau 4 décrit les règles "à la Prolog"

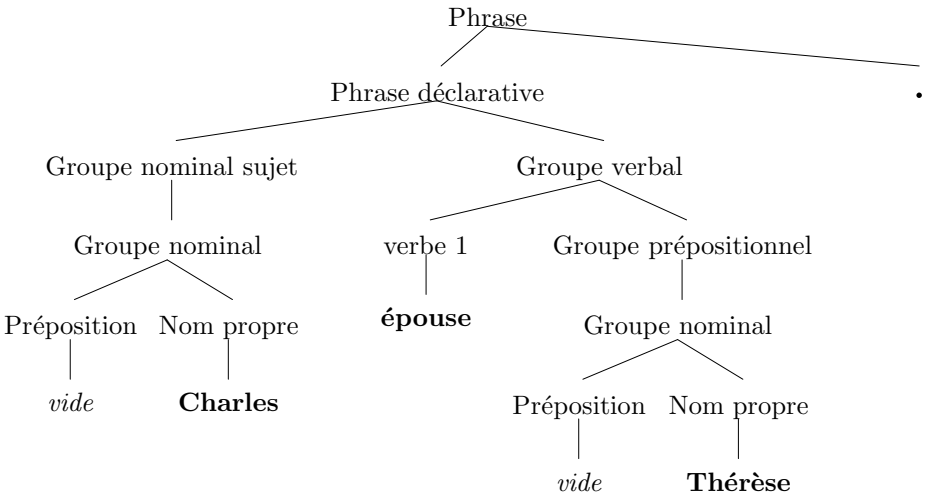

FIG. 2 – Structure syntaxique de la phrase *Charles épouse Thérèse*

permettant de construire, pendant l'analyse syntaxique de la phrase, le λ-terme

<*phrase*> → <*phrase interrogative*> **?** | <*phrase déclarative*> **.**
<*phrase déclarative*> → <*groupe nominal sujet*> <*groupe verbal*>
<*phrase interrogative*> →
qui <*groupe verbal*> |
qu'est-ce-qui <*groupe verbal*> |
<*préposition*> <*pronom interrogatif*> <*phrase moins*> |
<*préposition*> <*groupe nominal interrogatif*> <*phrase moins*> |
<*quel est le*> <*nom commun 1*> <*groupe prépositionnel*> |
<*quel est le*> <*nom commun 0*> <*relative*> |
est-ce-que <*phrase déclarative*> |
<*groupe nominal sujet*> <*groupe verbal*>
<*groupe nominal*> →
<*préposition*> <*nom propre*> |
<*préposition article*> <*nom commun 0*> <*relative*> |
<*préposition article*> <*nom commun 1*> <*groupe prépositionnel*>
<*groupe prépositionnel*> → <*groupe nominal*>
<*préposition article*> → **du** | **des** | **au** | **aux** | <*préposition*> < *article*>
<*préposition*> → **de** | **d'** | **à** | <*vide*>
<*vide*> →
< *article*> → <*article indéfini*> | <*article défini*> | <*autre*>
<*article indéfini*> → **un** | **une** | **des**
<*article défini*> → **le** | **l'** | **la** |**les**
<*autre*> → **tout** | **toute** | **chaque**
<*groupe verbal*> →
<*verbe être*> <*redoublement*> <*attribut*> |
<*verbe 0*> <*redoublement*> |
<*verbe 1*> <*redoublement*> <*groupe prépositionnel*>
<*redoublement*> → <*vide*> | **-t-il** | **-t-elle** | **-t-ils** | **-t-elles** | **-il** | **-elle** | **-ils** | **-elles**
<*attribut*> →
<*adjectif 0*> |
<*adjectif 1*> <*groupe prépositionnel*> |
<*article indéfini*> <*nom commun 0*> |
<*article défini*> <*nom commun 1*> <*groupe prépositionnel*>
<*nom propre*> → <*nom propre féminin*> | <*nom propre masculin*>
<*nom commun 0*> → <*nom commun 0 féminin*> | <*nom commun 0 masculin*>
<*nom commun 1*> → <*nom commun 1 féminin*> | <*nom commun 1 masculin*>
<*relative*> → <*préposition pronom*> <*phrase moins*> |<*vide*>
<*préposition pronom*> → **qui** | **que** | **à qui** | **dont**
<*phrase moins*> →
<*groupe verbal*> |
<*verbe 1*> <*groupe nominal sujet*> |
<*verbe être*> <*adjectif 1*> <*groupe nominal sujet*>|
<*groupe nominal sujet*> <*groupe verbal moins*> |
<*groupe nominal moins*> <*groupe verbal*>
<*groupe verbal moins*> →
<*verbe 1*> <*redoublement*> |
<*verbe 1*> <*groupe prépositionnel moins*> |
<*verbe être*> <*redoublement*> <*attribut moins*>
<*groupe prépositionnel moins*> →
<*groupe nominal moins*> |
à <*groupe nominal moins*>
<*groupe nominal moins*> → <*article défini*> <*nom commun 1*>
<*attribut moins*> → <*adjectif 1*> | <*article défini*> <*nom commun 1*>
<*groupe nominal interrogatif*> →
<*quel*> <*nom commun 0*> |
<*quel*> <*nom commun 1*> <*groupe prépositionnel*>|
<*article défini*> <*nom commun 1*> <*préposition*> <*pronom interrogatif*>
<*quel est le*> → <*quel*> <*verbe être*> <*article défini*>

TAB. 1 – Grammaire du langage

<*pronom interrogatif*> → **qui** | **que** | **quoi**
<*quel*> → **quel** | **quelle** | **quels** | **quelles**
<*verbe être*> → **est** | **sont**
<*nom propre masculin*> →
Louis XIII | **Henri IV** | **Gaston d'Orléans** | **Charles I** | **Charles II** | **Philippe IV** | **Charles** | **Richelieu** | **Barbin** | **Marillac** | **Luynes** | **Olivares** | **Louis XIV** | **Buckingham**
<*nom propre féminin*> →
Marie de Médicis | **Henriette de France** | **Elisabeth** | **Anne d'Autriche** | **Montpensier** | **Thérèse** | **La Grande Mademoiselle** | **France** | **Espagne** | **Angleterre**
<*nom commun 0 masculin*> → **homme** | **pays** | **ministre** | **roi** | **hommes** |**ministres** | **rois**
<*nom commun 0 féminin*> → **femme** | **femmes** | **reine** | **reines**
<*nom commun 1 masculin*> →
enfant | **enfants** | **fils** | **cousin** | **cousins** | **oncle** | **oncles** | **frère** | **frères** | **père** | **pères** | **ministre** |**ministres** | **mari** | **maris** | **roi** | **rois** | **favori** | **favoris**
<*nom commun 1 féminin*> →
sœur | **sœurs** | **femme** | **femmes** | **mère**|**mères** | **fille** | **filles** | **cousine** | **cousines** | **tante** | **tantes** |**reine** | **reines** | **favorite** | **favorites** **favoris**
<*verbe 0*> → **gouverne** | **gouvernent** | **complote** | **complotent**
<*verbe 1*> → **épouse** | **épousent**
<*adjectif 0*> → **séduisant** | **séduisante** | **séduisants** | **séduisantes**
<*adjectif 1*> →
marié | **mariée** | **mariés** | **mariées** | **mécontent** | **mécontentes** | **satisfait** | **satisfaite** | **satisfaits** | **satisfaites** | **fidèle** | **fidèles** | **favori** | **favorite** | **satisfaite** | **favoris** | **favorites** | **amoureux** | **amoureuse** | **amoureuses**

TAB. 2 – Lexique

sémantique représenté sur la figure 3. Ce dernier est ensuite normalisé en le terme *((épouse Charles) Thérèse)*.

Typage. Les λ-termes sémantiques sont, de façon implicite, simplement typés. Ils sont donc fortement normalisables, ce qui assure la terminaison de l'algorithme de normalisation. Un morphisme ψ associe un type à chaque non-terminal de la grammaire. Le terme sémantique de toute phrase dérivée d'un non terminal N est de type $\psi(\mathrm{N})$. Le tableau 3 indique le typage choisi pour les non-terminaux ayant une valeur sémantique.

Voici un exemple d'interrogation un peu plus complexe. A la question :

De quel pays l'homme dont Louis XIII épouse la sœur est-il le roi ?

Le système répond :

L'Espagne.

Discussion. Le typage du tableau 3 fait intervenir deux types de base : o pour les propositions et i pour les *individus*. Remarquons qu'en affectant le type $(i \to o) \to o$ aux groupes nominaux et le type $i \to o$ aux groupes verbaux, la

o	:	$phrase\ declarative$
$i \to o$	:	$nom\ commun\ 0,\ nom\ commun\ 0\ masculin,\ nom\ commun\ 0\ feminin$ $verbe\ 0$ $adjectif\ 0$ $phrase\ moins$ $groupe\ verbal$
$i \to (i \to o)$	:	$nom\ commun\ 1,\ nom\ commun\ 1\ masculin,\ nom\ commun\ 1\ feminin$ $verbe\ 1$ $adjectif\ 1$ $groupe\ verbal\ moins$
$(i \to o) \to o$	:	$nom\ propre,\ nom\ propre\ masculin,\ nom\ propre\ feminin$ $groupe\ nominal,\ groupe\ nominal\ sujet$
$(i \to o) \to (i \to o) \to o$	:	$article,\ article\ defini,\ articleindefini, autre$ $preposition\ article$
$(i \to o) \to (i \to o) \to i \to o$	:	$quel$
$(i \to o) \to i \to o$	:	$relative$ $groupe\ nominal\ interrogatif$
	:	
$(i \to (i \to o)) \to i \to o$	:	$groupe\ prepositionnel$
$i \to (i \to o) \to o$	:	$groupe\ nominal\ moins$

TAB. 3 – Typage des non terminaux

phrase (O) → phrase-declarative(O).
phrase-declarative (N.V) → groupe-nominal(N) groupe-verbal(V).
groupe-verbal (P.V) → verbe1(V) groupe-prepositionnel(P).
verbe1 (C.P) → verbe1-lexique(P) combinateur-binaire(C).
verbe1-lexique (epouse) → .
groupe-prepositionnel (C.N) → groupe-nominal(N) combinateur-groupe-prep(C).
groupe-nominal (N) → nom-propre(N).
nom-propre (C.N) → nom-propre-lexique(N) combinateur-nom-propre(C).
*nom-propre-lexique (***Charles***) → .*
*nom-propre-lexique (***Thérèse***) →.*
combinateur-binaire ((λpλiλj (p.j).i)) → .
combinateur-groupe-prep (λnλpλi (n.(λj (p.j).i))) → .
combinateur-nom-propre (λiλp (p.i))→ .

TAB. 4 – Règles de construction pour le terme de la fig. 3

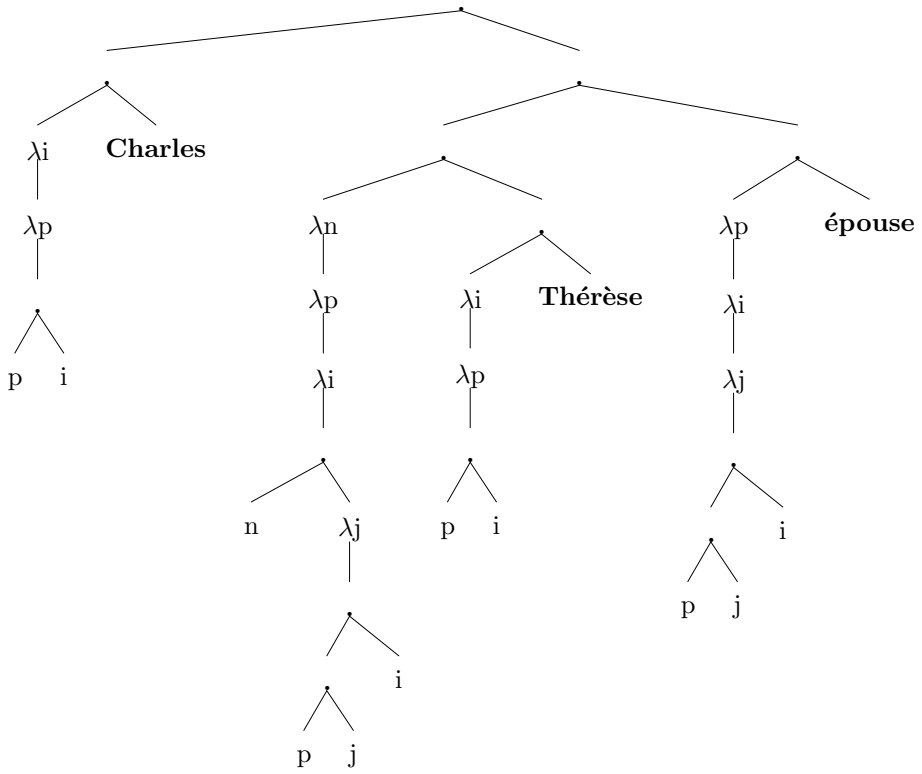

FIG. 3 – Sémantique de la phrase *Charles épouse Thérèse*

sémantique d'une phrase est obtenue en passant le groupe verbal en argument au groupe nominal. La quantification du sujet domine ainsi celle du groupe verbal. Une phrase de la forme *Un ministre participe à chaque conseil*, est alors comprise comme "un seul et unique ministre participe à tous les conseils". Mais la phrase peut aussi être interprétée par le fait que chaque conseil comporte la participation d'un ministre (qui peut différer d'un conseil à l'autre) : le quantificateur existentiel du sujet est alors dominé par le quantificateur universel qui apparaît dans le groupe verbal. On peut aisément généraliser la méthode proposée dans cet exercice d'école pour prendre également en compte ce genre d'interprétation. Il suffit que le groupe nominal soit passé en argument au groupe verbal et que celui-ci est alors affecté du type $((i \rightarrow o) \rightarrow o) \rightarrow o$. Les deux interprétations possibles de la phrase peuvent alors être définies à partir de deux interprétations distinctes du groupe verbal. Elles correspondent à deux manières distinctes de combiner les sémantiques du verbe et de son complément.

La sémantique considérée ici est une sémantique *statique* à la Montague, dans laquelle le sens d'une phrase est une valeur de vérité. Par son aspect forte-

ment compositionnel, elle a représenté une avancée majeure dans l'interprétation des langues naturelles. Cependant, elle permet difficilement la prise en compte du contexte, pourtant essentielle à la compréhension du discours. Depuis, des approches *dynamiques* comme les *Théories de la Représentation du Discours* (DRT)[Ka81, KR93] et la *Logique Dynamique des Prédicats* (DPL) [GS91] se sont focalisées sur ce problème ardu, tout en veillant à conserver la compositionnalité. Le concept central de *vérité* y est remplacé par celui *d'information* : le sens d'une phrase est sa capacité à modifier le contexte courant, vu comme un état d'information. La connection entre *information* et *vérité* reste toutefois essentielle. Divers logiciels, comme Illico [PS98], actuellement développé au Laboratoire d'Informatique Fondamentale de Marseille, s'inspirent de ces théories. Illico est écrit en PrologII+ et requiert une implantation des λ-termes en Prolog : le problème est donc toujours d'actualité.

2.1.3 Codage en λ-Prolog

Il est clair que l'extension du méta-langage des termes du premier ordre de Prolog à des termes d'ordre supérieur comme ceux du λ-calcul simplement typé permet une spécification bien plus aisée des syntaxes abstraites. C'est ce qui a été fait dans le langage λ-Prolog [MNS87]. Les clauses de Horn de Prolog y sont remplacées par les *formules de Harrop héréditaires* qui admettent l'implication et la quantification universelle dans les buts. Avec Olivier Ridoux, nous avons étendu l'étude précédente en proposant et comparant divers codages du λ-calcul en λ-Prolog [3].

La façon la plus directe de procéder est de représenter les λ-termes objets par les termes de λ-Prolog. Renommages et substitutions peuvent être alors réalisés respectivement par l'α-conversion et la β-conversion, "cablées" dans le langage. Les types sont dans ce cas automatiquement vérifiés. On peut également utiliser les termes de λ-Prolog pour représenter explicitement les abstractions et les applications sans nommer les variables. Ce choix permet d'exercer des contrôles sur les termes autres que ceux de λ-Prolog. L'utilisateur doit alors programmer la réduction, mais peut pour cela s'appuyer sur le mécanisme de substitution du système, comme suit.

```
kind lt type.
type '.' lt → lt → lt.
type lambda ( lt → lt)→ lt.

type (normalisation, reduire) lt → lt → o.
type recommencer lt → lt → lt → o.

normalisation E E_1 :- reduire E E_2, recommencer E E_2 E_1.

recommencer E E E.
recommencer E E_2 E_1 :- dif E E_2, normalisation E_2 E_1.
```

```
reduire C C :- constant C.                                    (1)
reduire (lambda M) (lambda M_1) :-
        pi x\(constant x => reduire (M x) (M_1 x)).           (2)
reduire (lambda M).A M_1 :-
        reduire A A_1, reduire (M A_1) M_1.                   (3)
reduire F.A F_1.A_1 :-
        dif_from_lambda F, reduire F F_1, reduire A A_1.      (4)
```

La clause (2) fait intervenir une quantification universelle pour accéder au corps d'une abstraction et une implication pour qualifier les variables universelles. Ceci permet de parcourir les abstrations en évitant les captures. La clause (3) utilise la β-reduction de λ-prolog pour implémenter la substitution au niveau objet. La liste de substitutions qui apparaît en argument de *reduire* du programme en PrologII, n'est plus ici nécessaire.Cette approche n'effectue bien sûr aucune vérification de type. Une autre solution consiste à tirer partie du polymorphisme du langage en déclarant :

```
kind lt type → type.
type '.' (lt A → B) → (lt A) → (lt B).
type lambda ( (lt A)→ (lt B))→ (lt A→ B).
```

La vérification de types est alors assurée par λ-Prolog.

A ces trois codages possibles, correspondent trois notions d'unification des termes très différentes. Dans la version en PrologII, il s'agit de l'unification des termes rationnels. Dans l'implémentation directe en λ-Prolog, il s'agit de l'$\alpha\beta\eta$-équivalence. Enfin, dans la représentation explicite, les termes sont unifiés modulo l'α-équivalence. On pourra se reporter à [3] pour une reflexion détaillée sur les mérites respectifs de ces diverses approches.

Cette étude a été appliquée à nouveau au même système d'interrogation en Français de banque de données. Nous avons introduit à cette occasion une forme de grammaire bien adaptée à une traduction automatique en un analyseur en λ-Prolog. Mon point de vue personnel après cette étude est que, malgré sa plus grande expressivité, l'utilisation de λ-Prolog se révèle moins confortable que celle de PrologII. Conçu comme un canevas logique, il est moins naturel comme langage de programmation et requiert souvent des démarches inhabituelles au programmeur.

2.2 Automates

J'ai également proposé, dans le domaine plus délimité des langages formels, un second exemple d'algorithme basé sur l'unification des arbres infinis ration-

nels [1, 2]. Il produit l'automate d'états finis déterministe et minimal d'un langage défini par une expression rationnelle. Ce problème a de multiples solutions. L'algorithme de Thomson [Th68], produit un automate non déterministe. Brzozowski dans [Br64] propose une méthode très élégante basée sur la notion d'expressions rationnelles dérivées pour obtenir un automate déterministe (AFD). Cette méthode s'applique aux expressions rationnelles étendues comportant des opérateurs comme l'intersection et la différence. L'algorithme de Mac Naughton et Yamada [MY60] construit un AFD (non minimal) en utilisant le "marquage" d'une expression rationnelle (deux occurrences d'un même symbole sont alors considérées comme des symboles distincts), mais il ne s'applique pas à l'intersection et la différence. Ces deux dernières approches ont inspiré diverses méthodes hybrides [BS86, An96, CZ01, Ch01] qui produisent efficacement des automates non détermistes. Elles ne s'appliquent pas aux expressions rationnelles étendues.

Ma solution repose sur la représentation d'un langage par un arbre infini (Fig 4). En supposant par exemple que l'alphabet d'entrée comporte deux symboles 0 et 1, un langage est représenté par un arbre binaire infini dont les nœuds sont étiquetés par F (final) ou NF (non-final). Le squelette de cet arbre représente l'ensemble des mots finis sur l'alphabet $\{0, 1\}$, chaque mot étant associé à un chemin fini partant de la racine avec la convention que la descente vers le sous-arbre gauche correspond au symbole 0 et la descente vers un sous-arbre droit correspond au symbole 1. Un mot appartient au langage si et seulement si le nœud auquel aboutit le chemin qui le caractérise est étiqueté par F.

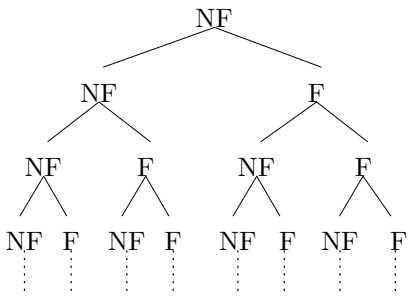

FIG. 4 – L'arbre du langage défini par $(0 + 1)^*1$

Cette représentation en extension a l'avantage d'être canonique, et si l'on peut dire opérationnelle puisqu'elle induit un algorithme de reconnaissance des mots du langage. L'arbre est en fait un automate déterministe, dont les états sont les sous-arbres, l'état initial est l'arbre global, les transitions sont étiquetées par un symbole ou l'autre selon qu'elles vont vers la gauche ou la droite. En remarquant que tout sous-arbre de racine q représente le langage dérivé du langage initial par le mot menant de la racine au nœud q, on en déduit l'équivalence des trois propositions ci-dessous :

– l'arbre est rationnel
– le langage est rationnel
– l'automate a un nombre fini d'états

De plus, le nombre d'états de l'automate étant dans ce cas exactement le nombre de ses langages dérivés, cet automate est minimal.

L'algorithme construit par induction structurelle sur une expression rationnelle, l'arbre rationnel du langage qu'elle définit. La représentation minimale de cet arbre est automatiquement calculée par Prolog. Par rapport aux travaux cités, la solution proposée présente les avantages suivants :

– elle produit l'automate déterministe minimal,
– elle s'applique aux expressions rationnelles étendues,
– manipulant des structures de données de haut niveau, elle est courte, élégante et donc facile à prouver,
– la sortie du programme Prolog est agréablement représentée par un diagramme.

Elle n'a cependant pas l'efficacité d'implémentations beaucoup plus concrètes (comme dans [Ch91] par exemple : les automates y sont représentés par des structures de données de très bas niveau en C).

3 Méthodes formelles

3.1 De Prolog à Coq

Si l'exécution d'un programme Prolog consiste à prouver une formule dans une logique intuitionniste, cette preuve n'est jamais construite explicitement, elle n'existe pas en tant qu'objet. Cependant, la sémantique déclarative d'un programme P repose sur la notion d'extraction d'un résultat à partir d'une telle preuve : étant donnée une formule (but) B à prouver, on calcule la substitution s la plus générale telle que $P \vdash s(B)$. λ-Prolog procède d'une démarche analogue, et grâce à l'extension des termes du premier ordre à ceux du λ-calcul simplement typé, on pouvait espérer que ce langage contribue à unifier les paradigmes de programmation en logique et de programmation fonctionnelle. Mais en fait, ce ne fut pas vraiment le cas. En effet, le méta-langage ne comporte pas d'opérateur de point fixe, ni de conditionnelle, ce qui limite considérablement son expressivité. De plus, la β-conversion n'est pas utilisée pour exécuter de vrais programmes.

Ces notions (programmation fonctionnelle, programmation en logique, dérivations intuitionnistes, objets preuves, extraction du contenu calculatoire d'une

preuve) trouvent harmonieusement leur place dans le Calcul des Constructions Inductives (CIC), sous-jacent à l'assistant de preuves Coq [Coq01].

Le Calcul des Constructions, conçu par Gérard Huet et Thierry Coquand [CH85], est un λ-calcul d'ordre supérieur obtenu en rajoutant au système F de Jean-Yves Girard [Gi71] des types dépendants. Il a par la suite été enrichi par Christine Paulin-Mohring [Pa96] de types inductifs, définis dans un style " à la Prolog". Puis Eduardo Giménez [Gi96a, Gi96b] y a rajouté les types co-inductifs qui permettent de spécifier des objets infinis (plus généraux que les termes rationnels de Prolog).

C'est donc un langage fonctionnel de forte expressivité pour la spécification de systèmes informatiques. Mais c'est également, par la correspondance de *Curry-Howard*, une logique intuitionniste d'ordre supérieur, permettant de raisonner sur les spécifications. Les preuves sont elles-mêmes des objets du langage, de même nature que les programmes. Elles peuvent ainsi être compilées, sauvegardées, et donc réutilisées sans avoir à être "rejouées". De plus, les types sont classés en deux *sortes* jumelles, *Prop* et *Set*, selon qu'ils sont interprétés par une proposition ou par un ensemble. Ceci permet d'obtenir des programmes certifiés par extraction automatique de la partie calculatoire (de type *Set*) d'un terme de preuve, en oubliant le reste, considéré comme un simple commentaire logique. Très récemment, le concept de *code mobile porteur de preuve* (*proof carrying code*) a apporté un regain d'intérêt aux objets preuves (cf. section 3.5).

Coq n'a pas été conçu dans le but d'exécuter des programmes, mais plutôt de les spécifier et de raisonner correctement sur les spécifications (on peut en particulier spécifier très aisément des programmes Prolog sans opérateur de contrôle et manipuler les arbres de preuve [Mo02]). Aussi serait-il abusif de laisser penser qu'il peut avantageusement remplacer un "vrai" langage de programmation fonctionnelle ou logique. Par exemple, il ne permet que des définitions de fonctions totales et uniquement la récursion structurelle. Les preuves ne sont pas en général produites automatiquement. Priorité a été donnée à faire de Coq un assistant de preuve très sûr. D'une part, il est étayé par de solides bases logiques. D'autre part, il satisfait le *critère de De Bruijn* en ce sens que sa validité repose sur un programme de petite taille, dont la correction est aisée à établir : le *vérificateur de types*. Contrairement à des approches plus pragmatiques dans lesquelles on prend le parti de simplifier la tâche de l'utilisateur en mettant à sa disposition de nombreux outils relativement complexes et dont on n'est pas complètement sûr qu'ils soient corrects, les concepteurs de Coq ont choisi la fiabilité.

La contre-partie de cette exigence et des potentialités de cet outil est sa difficulté d'utilisation. Au moment où j'ai commencé à m'y intéresser, Coq n'était d'ailleurs pas vraiment sorti du cercle des logiciens. La plupart des exemples significatifs de développements étaient empruntés à la logique. Mon but était donc de mener une réflexion sur l'utilisation du Calcul des Constructions Induc-

tives dans la pratique informatique.

L'idée était notamment de ne pas se restreindre à n'utiliser qu'une partie du système, dénominateur commun à tous les assistants de preuve du marché pour finaliser, vaille que vaille, une étude de cas. Je souhaitais expérimenter avec soin toutes ses particularités, qui suggéraient des idées originales de modélisation et de vérification. Les *types dépendants*, par exemple, de maniement délicat et dont certains niaient l'intérêt. L'aspect un peu futuriste de *l'extraction* de programmes certifiés à partir de preuves. Les *types co-inductifs*, qui furent implantés en Coq par la suite. *L'ordre supérieur* également, dont je voulais tirer partie en proposant des axiomatisations de haut niveau (systèmes de numération, automates, logiques, cardinaux, ...) conduisant à des méthodologies générales et réutilisables, dont le bien fondé serait ensuite établi par des études de cas convaincantes.

3.2 Vérification de systèmes matériels

[2] La vérification de circuits a été l'une des motivations originelles et l'un des premiers domaines d'application des systèmes d'aide à la preuve, avec les travaux de Gordon avec HOL [Go86, CGM87] et de Hunt [Hu89] avec Nqthm. C'est donc sur ce thème qu'a débuté mon travail, en collaboration avec Line Jakubiec, dont j'ai encadré la thèse.

3.2.1 Types dépendants, extraction et circuits combinatoires

Le premier volet de cette étude a été consacré à l'expérimentation des types dépendants. Ils sont réputés d'utilisation délicate. Une des raisons en est qu'il n'y a pas, dans le CIC, de règle de forme :

$$\frac{p : A(t) \quad q : (t{=}s)}{p : A(s)}$$

Une telle règle (l'égalité considérée ici est l'égalité de Leibniz) rendrait en effet le vérificateur de types indécidable. Il est dès lors impossible de coder dans le calcul des énoncés pourtant naturels. Considérons par exemple le type des listes de longueur `n`, paramétré par un ensemble de référence `E`. Il est défini par :

```
Inductive list : nat → Set :=
                 nil :(list O)|
                 cons :(∀n :nat) E →(list n) →(list (n+1)).
```

On ne peut exprimer qu'un terme `l` de type `(list n)` est égal à la liste vide `nil` sous l'hypothèse `n = 0`. En effet, le terme `l = nil` n'est pas typable car :

[2] Références [11, 27, 12, 14, 25, 6] de la liste de publications

– `l` est de type `(list n)`,
– `nil` est de type `(list 0)`,
– l'égalité porte uniquement sur des termes de types identiques.

Très souvent de jolies idées visant à intégrer un maximum d'information dans les types doivent être abandonnées à cause de ce genre de difficultés, qui peuvent par ailleurs apparaître au cours du processus de preuve, donc bien après le stade de la spécification. Cependant, l'intérêt des types dépendants pour la spécification de circuits n'est plus à démontrer [HDL90, HD92, AL92]. En permettant un certain nombre de contrôles précoces (au moment de la vérification du type des spécifications), ils évitent de perdre beaucoup de temps à raisonner sur des spécifications invalides. Il était donc intéressant de mener une étude pratique pour Coq. Pouvait-on utiliser ces types de façon significative ou bien tout cela restait-il au stade des idées élégantes et des intentions ? Nous répondons positivement à la première question et produisons de substantielles librairies visant à alléger le travail des vérificateurs utilisant Coq.

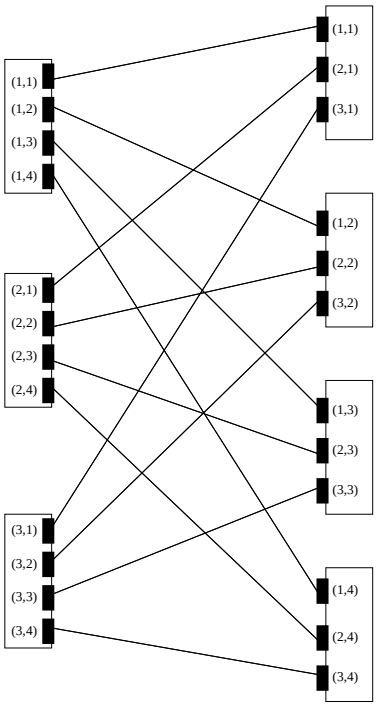

FIG. 5 – Une connexion transposée

Ports et connexions. En premier lieu, nous nous sommes attachées à spécifier finement les interconnexions de circuits au moyen de listes dépendantes. En effet,

contrairement aux techniques automatiques de *model checking*, la vérification formelle interactive avec un assistant de preuve offre la possibilité de raisonner sur des circuits dont la taille est variable. Si le nombre de ports d'entrée (par exemple) n'est pas fixé, l'entrée du circuit est représentée par une structure de données dont la longueur peut varier, c'est-à-dire une liste. L'emploi de listes dépendantes permet alors d'indiquer le nombre de ports au moyen d'un paramètre figurant dans le type de l'entrée. Certaines erreurs de spécification sont ainsi détectées très tôt ce qui peut faire gagner un temps précieux. Ces erreurs de spécification pour des circuits complexes, comme l'exemple traité dans la section suivante, ne sont pas anecdotiques. Voici une illustration de la façon dont elles peuvent être circonscrites par l'emploi systématique de listes dépendantes.

Considérons un circuit composé de `n` modules identiques à `m` ports. On modélise ses ports au moyen d'une liste de longueur `n` dont les éléments sont eux-mêmes des listes de longueur `m` de valeurs booléennes. Les connexions entre de tels dispositifs sont alors décrites par des fonctions sur les listes dépendantes. La figure 5 représente le cas particulier d'une *connexion transposée* entre un circuit ayant 3 sorties composées de 4 booléens et un second circuit possédant 4 entrées de 3 booléens chacunes. Une telle connexion revient en fait à calculer la matrice transposée d'une matrice 3x4. De façon plus générale elle est spécifiée par une fonction de type :

```
(∀n,m :nat)(list (list Bool n) m) →(list (list Bool m) n).
```

Cette fonction est définie par récursion sur `n` et fait appel à deux fonctions s'appliquant à une liste de listes, et retournant respectivement la liste de leurs têtes et la liste de leurs queues.

Dans ce style de codage, la distinction entre la phase de spécification et la phase de vérification n'est plus clairement établie puisque certaines preuves doivent être fournies au moment de la spécification. Voici un exemple, qui bien qu'un peu technique, contribue à illustrer le maniement des types dépendants.

Exemple. Supposons que l'égalité de Leibniz soit decidable sur l'ensemble `E`. On peut définir un prédicat inductif `member` exprimant qu'un élément `e` de type `E` apparaît dans une liste dépendante `l` :

```
Inductive member : E→(∀n :nat)(list n)→Set :=
    eq_hd : (∀n :nat)(∀l :(list n))
            (member e (n+1) (cons n e l))|
    in_tl : (∀e' :E)(∀n :nat)(∀l :(list n))
            (member e n l) → (member e (n+1) (cons n e' l)).
```

On peut alors construire une fonction `remove` qui ôte d'une liste `l` la première occurrence d'un élément qui est dans `l`. Une telle fonction prend en argument

non seulement l'élément `e` à supprimer, un entier naturel `n`, et une liste `l` de longueur `(n+1)`, mais aussi une preuve `p` que `e` est dans `l`. La fonction a donc pour type :

```
(∀e :E)(∀n :nat)(∀l :(list (n+1)))(member e (n+1) l)→(list n).
```

Elle est de la forme :

```
fix remove.(λe :E)(λn :nat)
Cases n of
         O     => (λl : (list 1))(λp :(member e 1 l))  nil|
         (m+1) => (λl : (list (n+1))) (λp :(member e (n+1) l))
                  if e = (head n l) then (tail (n+1) l)
                  else (cons m
                             (head n l)
                             (remove e m (tail (n+1) l) q)).
```

Dans l'appel récursif ci-dessus, le terme `q` est une preuve que `e` est dans la queue de `l`. Il est construit à partir d'un lemme préalable établissant que *si un élément de la liste* `l` *n'est pas égal à la tête de* `l`*, alors il est dans sa queue.* Preuves et programmes sont ainsi étroitement imbriqués.

Nous avons construit une bibliothèque très complète sur les listes dépendantes [27, 28]. Cette bibliothèque est largement utilisée dans ce qui suit. Nous l'avons complétée par des axiomatisations de systèmes de numération et d'architectures arithmétiques linéaires, qui font appel à des types dépendants divers et variés.

Systèmes de numération et architectures arithmétiques linéaires. Les systèmes de numération interviennent dans la vérification de circuits arithmétiques. Nous avons axiomatisé dans le CIC des systèmes de numération en faisant fortement appel aux listes dépendantes et aux sous-ensembles de la forme $\{$`x :A | (P a)`$\}$. Un système de numération est défini par une base `b` de type $\{$`x :nat | x>0`$\}$. Dans ce système, un chiffre est de type $\{$`x :nat | x < b`$\}$ et un *numeral* est une liste de chiffres de longueur `n`. Un numéral `N` désigne un entier naturel `v` défini par récursion structurelle sur `N` et on montre que $v < b^n$. On peut alors associer à tout numéral de type `(list chiffre n)` sa valeur `V` qui est un terme de type $\{x : nat \mid x < b^n\}$.

Ce travail a servi de base à une étude de cas suggérée dans [HDL90, HD92]. C'est un exemple de petite taille dont l'intérêt réside dans l'utilisation intensive des divers types dépendants des axiomatisations précédentes. Il s'agit de spécifier et de vérifier une classe de circuits combinatoires (circuits arithmétiques à architecture régulière). Ces circuits sont composés d'une connexion de cellules identiques à quatre ports portant une retenue entrante, une retenue sortante, deux entrées x_i et y_i (Fig. 6). Globalement, une telle connexion peut être vue

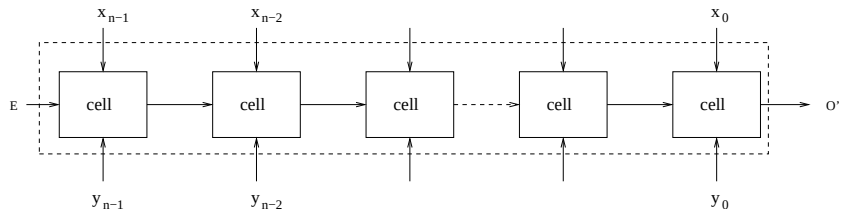

FIG. 6 – Exemple d'architecture arithmétique linéaire.

comme un circuit ayant une retenue entrante, une retenue sortante et deux entrées qui sont les numéraux `x` (constitué de la liste des x_i) et `y` (constitué de la liste des y_i). Elle est spécifiée par une relation inductive `Connexion` de type :

```
Connexion :(∀n :nat) A →(list B n)→ (list C n) → A → Prop
```

à l'aide d'une relation `cell` décrivant une cellule.

Nous montrons alors un théorème de factorisation sur les relations `R` de type :

```
R :(∀b :nat) A → {x :nat | x<b} → {x :nat | x<b} →A →Prop
```

Ce théorème établit que si la relation `R` est *propre* et *factorisable* [11], alors $(R\ b^n)$ est *itérativement implémentable*, c'est-à-dire qu'elle peut être implémentée par une connexion de `n` cellules, chacune réalisant la relation `(R b)`. Il permet de vérifier toute une classe d'architectures linéaires réalisant des relations arithmétiques propres et factorisables (comparaison, addition, soustraction, multiplication par une constante, division par une constante, reste modulo une constante ...).

Nous nous sommes également appuyées sur ce théorème de factorisation pour expérimenter le mécanisme d'extraction. Pour cela, il est énoncé sous une forme existentielle. Plus précisément, en supposant que `R` est propre et factorisable, on montre que pour tout élément `a :A`, pour tout entier `n` et pour tous numéraux `X` et `Y` de longueur `n` :

```
(∃ a' :A) (R b^n a V_X V_Y a')
```

où V_X et V_Y, de type $\{x : nat \mid x<b^n\}$ sont les valeurs respectives `X` et `Y`. Comme la logique sous-jacente est constructive, le terme de preuve de ce théorème contient un algorithme qui calcule en fonction de `a`, un témoin `a'` satisfaisant $(R\ b^n\ a\ V_X\ V_Y\ a')$. Ainsi, on peut extraire automatiquement une description fonctionnelle certifiée d'un circuit réalisant $(R\ b^n)$ [11, 27].

3.2.2 Coinduction, automates et circuits séquentiels

[3] Quand nous avons voulu étendre notre étude vers une axiomatisation générale des circuits séquentiels synchrones, il nous est apparu que celle-ci dépendrait essentiellement de la façon dont serait codé l'historique des valeurs portées par les fils. Généralement, les pas de temps sont représentés par des nombres entiers et les suites infinies de valeurs sont vues comme des fonctions sur ces entiers. Dans ce contexte, un circuit est lui-même modélisé comme une fonction du temps. Nous avons opté pour une approche plus algébrique, nous permettant de nous affranchir des références au temps. Elle est basée sur la co-induction, spécialement adaptée au raisonnement sur des objets infinis.

La co-induction dans le CIC. Rappelons brièvement les principes qui sous-tendent le raisonnement par co-induction. Les prédicats sur un ensemble `X` peuvent être identifiés avec les parties de `X` qu'ils caractérisent. L'ensemble $\mathcal{P}$`(X)` de ces parties, ordonné par l'inclusion, est un treillis complet. Le théorème de Tarski établit que tout opérateur monotone Φ `:` $\mathcal{P}$`(X)`$\rightarrow$ $\mathcal{P}$`(X)` possède un plus grand point fixe défini par :

$$\nu\Phi = \bigcup\{A \in \mathcal{P}(X) \mid A \subset \Phi(A)\} \quad .$$

Pour prouver qu'un prédicat défini comme un plus grand point fixe $\nu\Phi$ est satisfait par un élément `x` de `X`, il suffit ainsi de trouver un *invariant* `A` tel que :

- `A` $\subset$ Φ`(A)`
- `A(x)`

Cette méthode de preuve imprédicative est le principe de co-induction de Park.

Dans [Coq93] Thierry Coquand suggère une autre approche dans le cadre de la théorie des types reposant sur des définitions co-inductives au moyen d'un ensemble fini de constructeurs spécifiant des règles d'introduction. Il souligne ainsi élégamment la dualité entre induction et co-induction, puisque les types inductifs sont également définis au moyen de constructeurs [Pa96]. Par exemple, les suites infinies d'éléments d'un ensemble de référence `E` sont introduites par le constructeur `Cons` du type co-inductif suivant :

`Stream` = ν`t.(cons`^{Stream} `: E`$\rightarrow$`t`$\rightarrow$`t).`

On peut également définir les suites finies ou infinies par le type :

`List` = ν`t.(nil :t, cons`^{List} `: nat`$\rightarrow$`t`$\rightarrow$`t`).

Les termes fonctionnels à valeurs dans un type co-inductif peuvent être interprétés :

- soit comme des programmes récursifs calculant paresseusement des objets infinis,

[3] Références [12, 14, 25, 6]de la liste de publications

– soit comme des preuves utilisant en hypothèse la proposition qu'elles établissent (ce qui correspond à un appel récursif).

A tout type co-inductif σ est associé un destructeur $\texttt{case}^{\sigma}$. Pour le type *Stream* par exemple, il est typé comme suit :

$$\texttt{case}^{Stream} \texttt{ : Stream} \rightarrow \texttt{(E}\rightarrow \texttt{ Stream} \rightarrow \tau) \rightarrow \tau$$

La tête et la queue d'une liste infinie sont ainsi calculées par les termes suivants :

$$\texttt{hd= } \lambda\texttt{s.(case}^{\texttt{Stream}} \texttt{ s } \lambda\texttt{h}\lambda\texttt{t.h)} \qquad\qquad \texttt{tl= } \lambda\texttt{s.(case}^{\texttt{Stream}} \texttt{ s } \lambda\texttt{h}\lambda\texttt{t.t).}$$

La fonction `db` qui duplique les éléments de son argument peut alors être définie par :

$$\texttt{fix db.}\lambda\texttt{s.(cons}^{Stream} \texttt{ (hd s) (cons}^{Stream} \texttt{ (hd s) (db (tl s)))).}$$

Toutefois, un terme récursif bien formé doit satisfaire une condition syntaxique dite *de garde*, introduite dans le calcul comme condition de bord de sa règle de typage. Cette condition est duale de celle qui, dans le cas d'un terme fonctionnel défini par induction structurelle sur un argument bien fondé, exige que les appels récursifs aient lieu *sur un composant strict* de l'argument. Dans le cas co-inductif, la condition de garde impose que les appels récursifs aient lieu *sous un constructeur*. Elle traduit en fait une propriété de *contraction* de l'opérateur dont on cherche le point fixe et assure ainsi l'existence et l'unicité (sous forme canonique) de la solution. Avec Roberto Amadio, nous avons d'ailleurs proposé une méthode pour intégrer dans le typage cette condition (voir section 3.3).

L'approche suggérée dans [Coq93] est plus intuitive que le principe d'induction de Park, puisqu'elle ne nécessite pas la recherche d'un invariant. Elle a été implantée en Coq avec quelques variations dûes à l'aspect imprédicatif du CIC [Gi96a, Gi96b]. Nous y faisons largement appel pour la preuve de circuits où l'emploi des types co-inductifs se révèle très agréable.

Description co-inductive des architectures séquentielles et de leurs comportements. Nous spécifions donc les flux de valeurs par des termes du type `Stream`. Un registre est ainsi simplement modélisé par le constructeur $\texttt{cons}^{Stream}$ de ce type, noté `Cons` pour plus de simplicité dans ce paragraphe. Le problème est alors d'éviter de ré-introduire par la suite les paramètres temporels. Il ne peut être question, par exemple, de représenter les comportements des circuits par des chronogrammes, ni de prétendre accéder à l'élément d'indice `t` d'une suite infinie. Notre choix s'est porté sur une modélisation par des machines à états finis qui produisent un flux de sortie en réponse à un flux d'entrée : les automates de Moore et Mealy [Mo56, Me55].

Ces automates sont codés par des *fonctions récursives s'appliquant sur des streams co-inductives*. Plus précisément, nous associons à un automate de Moore

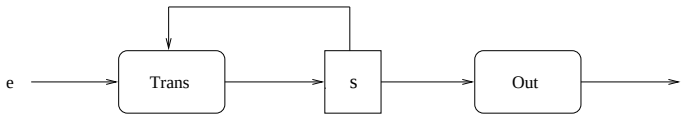

FIG. 7 – Automate de Moore

par exemple (Fig. 7), une fonction qui prend en argument un état courant `s` et d'un flux d'entrée `e`. Si `Out` et `Trans` sont respectivement les fonctions de sortie et de transition, elle est codée par un point fixe comme suit :

`(fix Moore)(λe)(λs) (Cons (Out s) (Moore e' (Trans` $\mathtt{e}_0$ `s))).`

Ici, `e'` désigne le flux d'entrée, privé de son premier élément $\mathtt{e}_0$.

Notons que ce codage est *très général*. En effet, aucune hypothèse n'est faite sur l'ensemble des états. Il s'applique donc à des machines ayant un nombre infini d'états. De plus, les fonctions de transition et de sortie peuvent être des algorithmes complexes.

Ceci nous permet de modéliser aisément au moyen de ces automates à la fois les structures et les comportements, considérés de ce point de vue comme des descriptions de la même entité, représentée à des niveaux d'abstraction différents. L'*uniformité de la modélisation* et le fait d'éviter les références au temps ne sont pas les seuls avantages de cette approche. Ces automates sont effectivement des représentations synthétiques, clairement décrites par des diagrammes de transition de petite taille, mais suffisamment abstraites pour porter beaucoup d'information. Comparés aux outils de bas niveau comme les chronogrammes, ils permettent d'exprimer de façon plus naturelle (et donc plus fiable) les spécifications informelles des concepteurs.

Nous introduisons par ailleurs une notion *d'équivalence sur les automates*, de telle sorte que prouver la correction d'un circuit consiste à établir l'équivalence de son automate structurel et de son automate comportemental. La relation d'équivalence étant co-inductive, la preuve est donc établie par co-induction. Elle repose sur un unique lemme générique qui est en fait un *schéma de preuve* sous-jacent à toute preuve de correction et qui capture une fois pour toutes les aspects temporels du raisonnement.

Par ailleurs, l'ensemble des automates peut être muni de règles de composition [HU79, Bo67]. Nous axiomatisons dans CIC cette *structure algébrique*. De ce fait, les architectures peuvent être décrites de façon *modulaire* et la vérification d'un circuit complexe peut être déduite de celles de composants plus simples.

Nous établissons également que l'équivalence que nous avons définie sur les

automates est une *congruence* pour les règles de composition. Ceci conduit à des preuves hiérarchiques dans lesquelles un composant pre-prouvé d'une architecture modulaire est remplacé par son comportement.

Enfin, nous introduisons les types dépendants chaque fois qu'ils contribuent à une meilleure fiabilité des spécifications. Ils sont utilisés conjointement avec les types co-inductifs pour coder par exemple un signal de `n` booléens par une `stream` de listes de booléens de longueur `n`.

Travaux connexes. De nombreux travaux ont été menés sur le thème de la preuve de circuits avec des assistants de preuve et je ne mentionnerai ici que les plus proches de cette étude. Un codage imprédicatif des *streams* en théorie des types est utilisée dans [Pa95] pour prouver un multiplicateur. Toutefois le circuit y est représenté comme une fonction d'un paramètre de temps, ce qui fait perdre son intérêt à la spécification co-inductive. Dans [MJ96], les *streams* sont spécifiées en PVS comme un type de données non vide défini par des axiomes sur un constructeur et un accesseur. Cette axiomatisation est alors utilisée pour prouver un circuit tolérant aux pannes par co-induction basée sur des bisimulations. Le système *Lava* développé à l'Université de Chalmers est basé sur une représentation fonctionnelle des circuits en Haskell. Ils peuvent être simulés par un interpréteur et vérifiés par un système de démonstration basé sur la logique propositionnelle. Il permet uniquement des circuits de taille fixée. La vérification des circuits séquentiels se fait par induction sur le temps. Plus récemment, dans [BH01] Boulmé et Hamon ont spécifié dans Coq un langage synchrone de flots de données. Ils utilisent des types dépendants co-inductifs pour coder des *streams* dans lesquelles il manque des éléments et ils appliquent le paradigme *d'horloges-comme-des-types* pour exprimer des contraintes de synchronisation statiques avec une forme restreinte de types dépendants.

Nous avons mis en œuvre notre méthodologie pour vérifier un circuit de taille et de complexité significatives, le *Fairisle ATM Switch Fabric*.

3.2.3 Une étude de cas : l'ATM Switch Fabric

L'intérêt de ce circuit est multiple. D'une part il nous a permis de justifier nos choix en évitant l'écueil des exemples trop académiques. Il s'agit d'un vrai circuit, conçu et réalisé à l'Université de Cambridge par le *System Research Group*. Le fait que nos méthodes de spécification et de vérification s'adaptent parfaitement et systématiquement aux différentes parties du circuit atteste du bien fondé de notre démarche. De plus, ce circuit a été largement utilisé par la communauté scientifique, ce qui nous a permis de mener une étude comparative pertinente.

Le circuit. Le *Fairisle ATM* est un réseau expérimental qui fonctionne selon un mode de transfert asynchrone (*Asynchronous Transfert Mode*). Il est composé d'une série de *Switch Elements* identiques, simples et rapides. Chaque machine

du réseau est connectée à un switch par lequel elle communique avec les autres machines.

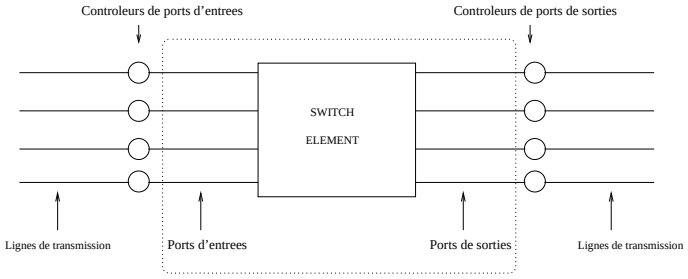

FIG. 8 – Vue extérieure d'un switch

Le *Fairisle Switch Fabric* (Fig. 8) est donc le composant essentiel du réseau (élément d'aiguillage). Il est composé de lignes de transmission assurant la communication du circuit avec l'environnement extérieur. Sur ces lignes, des contrôleurs de ports gèrent l'entrée et la sortie des données en créant des files d'attente et des connexions virtuelles (protocole) entre la source et la destination. Il y a 4 ports d'entrée et 4 ports de sortie. Les données à injecter dans le switch se présentent sur les ports d'entrée et sont destinées aux contrôleurs de ports de sortie via ces ports de sortie. Les contrôleurs de ports d'entrée sont informés de succès ou de l'échec concernant l'envoi de leur données, ce qui permet la gestion des files d'attente.

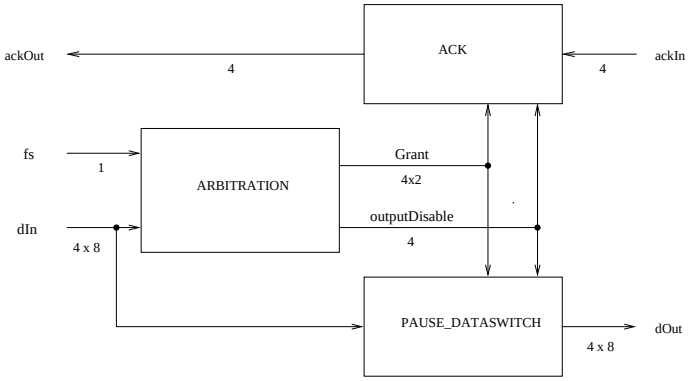

FIG. 9 – Description structurelle d'un switch

La structure d'un switch est décrite Fig. 9. En fait, vis-à-vis de son environnement extérieur, le circuit a un comportement asynchrone dans le sens où il

n'y a aucun moyen de savoir à quel moment des données vont se présenter sur les ports d'entrée. Mais en réalité, il a un comportement cyclique, cadencé par un signal d'entrée *fs* indiquant le début d'un cycle. Lorsque *fs* prend la valeur 1, le circuit teste si des données se trouvent sur les ports d'entrée (ces ports sont dits *actifs*). Si l'un au moins des ports est actif, les données sont injectées dans le switch afin d'atteindre la destination requise. Lorsque plusieurs ports d'entrée souhaitent envoyer leurs données sur un même port de sortie, il y a conflit. Le rôle du switch est alors d'éviter les collisions en effectuant un décodage de priorité, puis un arbitrage selon l'algorithme *Round-Robin*. Cadencement, décodage de priorité et arbitrage sont réalisés par le composant ARBITRATION. Après arbitrage, l'unité ACK envoie les accusés de reception aux ports d'entrée dont la requête a été satisfaite, et l'unité PAUSE_DATASWITCH réalise effectivement l'aiguillage.

Les composants ACK et PAUSE_DATASWITCH sont essentiellement combinatoires et relativement simples. Nous nous sommes donc attachées à la vérification de la partie ARBITRATION (Fig. 10). Elle joue en effet un rôle essentiel dans le switch, son comportement est non trivial, elle est composée de plusieurs autres unités elles-mêmes structurées en divers composants, et comporte de nombreux éléments mémorisants. L'unité FOUR_ARBITER par exemple est la composition en parallèle de quatre sous-modules identiques et réalise l'arbitration pour les quatre ports de sortie. L'unité TIMING comporte des registres et des boucles et PRIORITY_DECODE est combinatoire.

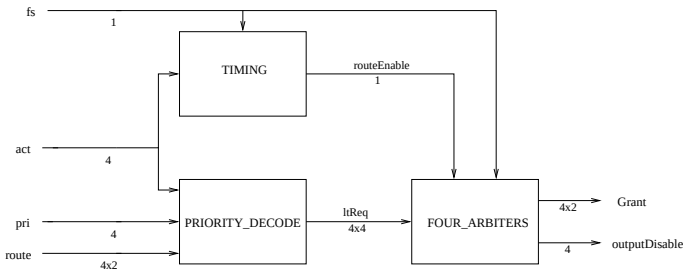

FIG. 10 – L'unité *arbitration*

L'ensemble des fichiers relatifs à la preuve de hardware est constitué d'environ 13.500 lignes de code [28], dont 6000 sont dédiées à la vérification de l'unité arbitration. On trouvera les détails de la vérification dans les articles joints au document.

Travaux connexes. L'ATM switch fabric a été (et est encore) largement utilisé comme *benchmark* par la communauté internationale travaillant sur la preuve de circuits. C'est le travail de Paul Curzon avec l'assistant de preuve

HOL [Cu94a, Cu94b], qui se rapproche le plus de ce que nous avons fait. Cependant, les spécifications comportementales y sont des propositions universellement quantifiées sur plusieurs entiers naturels représentant des pas de temps. Les preuves de correction consistent à établir une implication entre les descriptions structurelles et ces propositions comportementales. Ceci est fait en général au moyen de plusieurs inductions imbriquées. De plus, cette implication n'est pas prouvée directement, mais nécessite l'introduction d'une description comportementale intermédiaire, plus concrète et plus proche de la structure du circuit. Notre approche est plus simple et plus claire. Notons tout d'abord que nous établissons des équivalences, qui sont par ailleurs des congruences. Ces équivalences résultent directement de l'application d'un schéma de preuve général (et ne nécessitent pas de structures intermédiaires). L'effort se concentre en fait sur une preuve d'invariant, sans référence au temps. On trouvera une comparaison plus détaillée dans [6].

Dans [SK95], Schneider et Kropf utilisent Mephisto, un outil basé sur HOL. Bien qu'ils aient automatisé la vérification de sous-modules de bas niveau, ils ne font pas une vérification du circuit global. D'autres approches [LT98] proposent des procédés d'abstraction pour éviter l'explosion des états, mais là encore, ils ne permettent pas de vérifier la totalité du circuit. Dans [GR96] Garcez et Rosenstiel vérifient uniquement certaines propriétés sur des spécifications de bas niveau, en utilisant HSIS, un outil de *model checking*. Dans [TSC99] les auteurs utilisent les MDG (Multiple Decision Graphs) et manipulent des automates de plus bas niveau et plus gros que les notres. La preuve automatique nécessite toutefois un travail à la main préalable et aucun résultat n'est réutilisable. On trouvera des comparaisons sur ces divers travaux dans [TCL98, TC99]. Plus récemment Kort, Tahar et Curzon ont proposé une méthode hybride utilisant à la fois HOL et MDG [KTC03]. L'utilisateur doit donner à HOL une description modulaire de circuit et fournir une description comportementale de ses sous-modules. Puis, on tente une preuve automatique avec MDG. Si elle provoque une explosion combinatoire, le théorème de correction en HOL est décomposé en un certain nombre de sous-buts correspondant aux sous-modules, que l'on essaie à nouveau d'établir avec MDG. Bien que plus automatisée que la notre, cette approche présente quelques inconvénients. Tout d'abord, HOL doit supposer que MDG est correct. Ensuite, elle requiert un travail préalable à la main (codage de bas niveau des données destinées à MDG, décomposition du circuit et description des comportements des sous-modules). Enfin, le processus avec MDG peut ne pas terminer, et dans ce cas, il faut utiliser à la main des heuristiques.

On peut résumer cette étude en concluant que notre démarche est certainement la plus générale et qu'une grande partie de ce travail est réutilisable. Bien que non automatique, le processus de preuve permet de se focaliser sur les aspects combinatoires, les considérations temporelles se déduisant immédiatement d'un lemme général. Le fait que la partie relative à l'étude de cas soit relativement brève souligne l'efficacité de cette méthodologie fortement modulaire

reposant de plus sur des descriptions particulièrement abstraites et compactes.

Coq et la preuve de circuits : conclusion. Ce travail répond complètement aux questions que nous nous étions posées au départ. Nous avons démontré que Coq, *avec toutes ses caractéristiques*, est praticable dans le domaine significatif de la preuve de circuits. Nous avons également produit un bon nombre d'outils réutilisables, ce qui n'est pas sans intérêt pour un système tel que Coq, dont l'utilisation demande une expertise certaine. Le traitement des parties combinatoires reste cependant fastidieux et il serait intéressant de s'attacher maintenant à apporter davantage d'automatisation.

Enfin, notre volonté d'utiliser au maximum les potentialités du CIC (ordre supérieur, types dépendants, types inductifs et co-inductifs) nous a conduites à des procédés de modélisation et des stratégies de vérification élégants, ayant un intérêt propre, indépendamment de l'outil de preuve. La mise en œuvre d'une telle démarche dès l'étape de conception, par sa clarté et sa précision, peut aider à l'élaboration de systèmes fiables et performants.

3.2.4 Une algèbre pour les systèmes matériels synchrones

[4] Parallélement à ce travail, nous avons mené avec Jean-Luc Paillet et Michel Allemand (dont j'ai co-encadré la thèse) une étude dont le point de départ est dans [Pa87]. Nous avons proposé un langage typé comprenant notamment des opérateurs de composition séquentielle, de composition parallèle, de point fixe et un opérateur *passé P* donnant son nom au langage : le P-Calcul. Les expressions de ce langage sont interprétées dans une algèbre de fonctions sur des suites de valeurs indexées sur $\mathbb{Z}$. Une expression décrit la structure d'un circuit comme une interconnexion de composants, leur interprétation décrit son comportement.

On définit de plus sur ce langage un système de règles de réécriture préservant l'interprétation et permettant de ce fait de transformer algébriquement un circuit en un autre qui lui est équivalent. Nous avons établi la complétude de ce système. On en déduit que toute expression a une forme normale obtenue en propageant les éléments mémorisants (opérateur P) vers les entrées. Nous en déduisons un critère de reconnaissance de circuits correctement synchronisés, qui seuls possèdent une interprétation fonctionnelle que nous construisons alors effectivement (c'est en général la solution d'une équation de point fixe).

Cette étude a été complétée par une expérimentation sur le Larch Prover (LP) [GG91]. Il est basé sur des techniques de réécriture pour un sous-ensemble de la logique du premier ordre multi-sortée avec égalité. Il fait partie de cette classe d'outils qui suivent une démarche "pragmatique". Il est plus facile d'utilisation que Coq, mais beaucoup plus permissif. L'utilisateur définit des axiomatisations sous forme d'un ensemble de déclarations d'opérateurs typés, de règles

[4] Références [8, 9, 10] de la liste de publications

de réécriture, de déclarations de schéma d'induction, de règles de déduction. Un but est normalisé par réécriture. Si le résultat de la normalisation ne permet pas de conclure, l'utilisateur peut choisir de prouver le but obtenu par cas, par induction ou par contradiction. Nous avons dégagé une méthodologie visant à produire des théories LP à partir de la description de circuits en P-calcul et mené un certain nombre détudes de cas avec cet outil.

3.3 Sémantique des types co-inductifs

[5] Comme indiqué section 3.2.2, l'expérimentation des types co-inductifs de Coq a été le point de départ d'une étude menée en collaboration avec Roberto Amadio, dans le but de remplacer la condition syntaxique de garde pour les termes récursifs par une condition de typage. Nous proposons ainsi, dans l'article en référence, une extension du λ-calcul simplement typé aux types co-inductifs, que nous interprétons dans la catégorie des REP (relations d'équivalence partielles i.e. relations transitives et symétriques).

Interprétation du fragment simplement typé. On considére une algèbre combinatoire partielle *(D, s, k, .)*, dans laquelle on simule la λ-abstraction (à gauche ci-dessous) et on interprète dans un environnement ρ le λ-calcul pur (à droite ci-dessous).

λd.d = skk	$[\![x]\!]_\rho = \rho(x)$
λd.t = kt (*si d n'apparaît pas dans t)*	$[\![\lambda x.P]\!]_\rho = \lambda d.[\![P]\!]_{\rho[d/x]}$
λd.tt'= s(λd.t)(λ d.t')	$[\![PQ]\!]_\rho = ([\![P]\!]_\rho)([\![Q]\!]_\rho)$

On définit alors les combinateurs :

$\forall i \in \{1,\ 2\}$	$\pi_i\ =\ \lambda d.d(\lambda x_1 x_2.x_i)$	$j_i\ =\ \lambda d.\lambda y_1 y_2.(y_i\ d)$
$< d_1, d_2 >= \lambda p.(pd_1)d_2$	$\Pi\ = \lambda\phi_1\phi_2.\lambda d.\ < \phi_1 d,\ \phi_2 d >$	$\Sigma\ =\ \lambda\phi_1\phi_2.\lambda d.d\phi_1\phi_2$
$ev\ =\ \lambda d.(\pi_1 d)(\pi_2 d)$	$lam\ =\ \lambda\phi.\lambda dd'.\phi < d, d' >$	

ainsi que le produit, la somme et l'exponentiation de deux REP de la façon suivante :

$(d,\ e) \in A_1 \times A_2\ \Leftrightarrow\ \forall i \in \{1,\ 2\}\ (\pi_i d,\ \pi_i e) \in A_i$
$(d,\ e) \in A_1 + A_2\ \Leftrightarrow\ \exists i \in \{1, 2\}\ \exists e',\ d' \in D\ (d\ =\ j_i d',\ e\ =\ j_i e',\ (d', e') \in A_i)$
$(f,\ g) \in B^A\ \Leftrightarrow\ \forall d,\ e \in D\ ((d, e) \in A\ \Rightarrow\ (fd,\ ge) \in B)$

En désignant par $[d]_A$ la classe modulo A d'un élément d de D tel que $(d, d) \in A$, on considère la catégorie $\mathrm{REP}_\mathtt{D}$ dont :

- les objets sont les REP sur D,
- $f : A \to B$ est un morphisme $\Leftrightarrow\ \exists\phi \in D,\ \ f = [\phi]_{B^A}$

[5] Références [13] de la liste de publications

Le produit et l'exponentiation font de $\mathrm{REP}_{\mathtt{D}}$ une catégorie cartésienne fermée, munie d'un objet terminal $1 = \mathrm{D}\times\mathrm{D}$ et d'un objet initial $0 = \emptyset$. Le fragment simplement typé du calcul s'interprète canoniquement dans $\mathrm{REP}_{\mathtt{D}}$ comme dans toute catégorie cartésienne fermée [AC98] [AC98]. En ce qui concerne les types, on définit dans l'environnement η :

$[\![\mathtt{t}]\!] = \eta(\mathtt{t})$ *si* $\mathtt{t}$ *est un type de base* et $[\![\tau \to \tau']\!] = [\![\tau']\!]^{[\![\tau]\!]}$

Pour toute assignation de types $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$, on note $[\![\Gamma]\!] = 1 \times [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$. Les jugements de typage de la forme $\Gamma \vdash M : \tau$, sont alors interprétés par des morphismes $f : [\![\Gamma]\!] \to [\![\tau]\!]$ comme suit :

$$[\![\Gamma \vdash M : \tau]\!] = [\lambda d.[\![\overline{M}]\!]_{[(\pi_i\ d)/x_i]}]_{[\![\tau]\!]^{[\![\Gamma]\!]}} \tag{1}$$

$\overline{M}$ est le λ-terme pur sous-jacent à M et $[\![\overline{M}]\!]_{[(\pi_i\ d)/x_i]}$ désigne l'interprétation de ce terme dans D, la variable x_i étant interprétée par la $i^{ème}$ projection de d ($(\pi_i\ d)$ est une abréviation pour $\pi_1(\pi_2^{n-i}\ d)$).

Notre travail a donc consisté à étendre ce contexte aux types co-inductifs et aux termes récursifs.

Interprétation des types co-inductifs. L'idée de base est de les considérer comme des plus grands points fixes définis par approximations. En effet, d'après le théorème de Tarski, étant donné un opérateur monotone $\Phi : \mathcal{P}(\mathtt{X}) \to \mathcal{P}(\mathtt{X})$, si l'on pose pour tout ordinal α :

$\Phi^{\alpha} = \mathtt{X}$ si α=0,
$\Phi^{\alpha+1} = \Phi(\Phi^{\alpha})$,
$\Phi^{\alpha} = \bigcap_{\beta<\alpha} \Phi^{\beta}$ si α est un ordinal limite,

alors, $\exists\alpha\ \nu\Phi = \Phi^{\alpha}$.

Nous associons aux types `Stream` et `List` les opérateurs $\mathcal{F}_{Stream}$ et $\mathcal{F}_{List}$ définis par :

$$\mathcal{F}_{Stream}(A) = [\![nat]\!] \times A \text{ et } \mathcal{F}_{List}(A) = 1 + [\![nat]\!] \times A.$$

Ceci se généralise à tout type co-inductif σ : l'opérateur associé $\mathcal{F}_{\sigma}$ est monotone dès que les types de ses constructeurs satisfont certaines conditions syntaxiques de positivité. L'idée est d'interpréter σ comme le plus grand point fixe de $\mathcal{F}_{\sigma}$, calculé au moyen des approximations $\mathcal{F}_{\sigma}^{\alpha}$.

Interprétation des termes M. On prolonge la méthode employée pour le fragment simplement typé en interprétant les jugements de typage comme en (1). On associe pour cela à tout terme M un λ-terme pur $\overline{M}$ qui peut être vu comme une version compilée de M. Pour les constructeurs et les destructeurs, il

s'agit de combinateurs. Par exemple pour les constructeurs et les destructeurs des types `Stream` et `List`, on définit :

$$\overline{cons^{Stream}} \;=\; \lambda h,t.\;\lambda f.f < h,t > \qquad \overline{case^{Stream}} \;=\; \lambda s.\lambda f.\; f(\pi_1 s)(\pi_2 s)$$

$$\overline{nil} \;=\; \lambda f_n f_c.f_n \qquad \overline{cons^{List}} \;=\; \lambda h,t.\lambda f_n f_c.\; f_c < h,t >$$

$$\overline{case^{List}} \;=\; \lambda l.\lambda f_n f_c.\; l\; f_n\; (\lambda u.f_c(\pi_1 u)(\pi_2 u))$$

La *compilation* se prolonge à tous les autres termes par induction sur leur structure. On a :

$$\overline{x} \;=\; x \qquad \overline{\lambda x.M} \;=\; \lambda x.\overline{M} \qquad \overline{M\;N} \;=\; \overline{M}\;\overline{N} \qquad \overline{fix\;x.M} \;=\; Y(\lambda x.\overline{M})$$

où Y est l'opérateur de point fixe défini par $Y \;=\; \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

Considérons un terme clos récursif $fix\;x.M : \sigma$ où σ est un type co-inductif. Pour que le typage soit valide, il faut établir $(Y(\lambda x.\overline{M}), Y(\lambda x.\overline{M})) \;\in\; [\![\sigma]\!]$ et donc, σ étant interprété comme le plus grand point fixe de $\mathcal{F}_\sigma$, on envisage de prouver par induction transfinie que

$$\forall \alpha\; ordinal, (Y(\lambda x.\overline{M}), Y(\lambda x.\overline{M})) \in \mathcal{F}_\sigma^\alpha$$

Il faut en particulier prouver que :

$$\forall \alpha\; ordinal, (Y(\lambda x.\overline{M}), Y(\lambda x.\overline{M})) \in \mathcal{F}_\sigma^\alpha \Rightarrow (Y(\lambda x.\overline{M}), Y(\lambda x.\overline{M})) \in \mathcal{F}_\sigma^{\alpha+1}$$

Comme $Y(\lambda x.\overline{M}) = \overline{M}[Y(\lambda x.\overline{M})/x]$, la condition précédente est satisfaite dès que :

$$\forall \alpha, P \quad ((P,P) \in F_\sigma^\alpha \Rightarrow (\overline{M}[P/x], \overline{M}[P/x]) \in F_\sigma^{\alpha+1}) \tag{2}$$

Pour représenter cette condition dans la syntaxe, on introduit deux nouveaux types $\check{\sigma}$ et $\check{\sigma}^+$ associés à σ. La propriété de contraction exprimée par la condition (2) est traduite par un jugement de typage de la forme :

$$x : \check{\sigma} \vdash M : \check{\sigma}^+ \tag{3}$$

dans une interprétation des types paramétrée par les ordinaux dans laquelle :

$$[\![\check{\sigma}]\!]^\alpha \;=\; F_\sigma^\alpha \qquad \text{et} \qquad [\![\check{\sigma}^+]\!]^\alpha \;=\; F_\sigma^{\alpha+1}$$

Modification des règles de typage Les notions sémantiques sous-jacentes à ces nouveaux types conduisent aux règles de sous-typage suivantes :

$$\frac{}{\tau \le \tau} \qquad \frac{}{\sigma \le \check{\sigma}^+} \qquad \frac{}{\sigma \le \check{\sigma}} \qquad \frac{}{\check{\sigma}^+ \le \check{\sigma}} \qquad \frac{\tau_2 \le \tau_1 \quad \tau_1' \le \tau_2'}{\tau_1 \to \tau_1' \le \tau_2 \to \tau_2'}$$

On rajoute aux règles de typage des constructeurs des règles de contraction et à celles des destructeurs des règles duales. Pour le type *Stream* on rajoute ainsi :

$$\begin{array}{l} cons^{Stream} : nat \ \rightarrow \ \check{Stream} \ \rightarrow \ \check{Stream}^{+} \\ case^{Stream} : \ \check{Stream}^{+} \rightarrow (nat \rightarrow \check{Stream} \rightarrow \ \tau) \ \rightarrow \ \tau \end{array} \tag{4}$$

Quant aux points fixes, en notant $\vec{\tau} \rightarrow \sigma = \tau_1 \rightarrow \ldots \tau_m \rightarrow \sigma \quad (m \geq 0)$, ils se typent comme suit :

$$\frac{\begin{array}{c} T(\Gamma) \cup \{\tau_1 \ldots \tau_m\} \subseteq T \\ \Gamma, x : \vec{\tau} \rightarrow \sigma \vdash M : \vec{\tau} \rightarrow \sigma \\ \Gamma, x : \vec{\tau} \rightarrow \check{\sigma} \vdash M : \vec{\tau} \rightarrow \check{\sigma}^{+} \end{array}}{\Gamma \vdash \mathsf{fix}\ x.M : \vec{\tau} \rightarrow \sigma} \qquad \frac{\begin{array}{c} T(\Gamma) \cup \{\tau_1 \ldots \tau_m\} \subseteq T^{+} \\ \Gamma, x : \vec{\tau} \rightarrow \check{\sigma} \vdash M : \vec{\tau} \rightarrow \check{\sigma}^{+} \end{array}}{\Gamma \vdash \mathsf{fix}\ x.M : \vec{\tau} \rightarrow \check{\sigma}^{+}}$$

Dans ces règles, T désigne l'ensemble de types initial, sans occurrence de types de la forme $\check{\sigma}$ et $\check{\sigma}^{+}$. T^{+} est enrichi des types de la forme $\check{\sigma}$ et $\check{\sigma}^{+}$, fermé par le constructeur $\rightarrow$. De plus, $\check{\sigma}$ et $\check{\sigma}^{+}$ doivent apparaître en position positive dans les éléments de T^{+}. L'hypothèse

$$\Gamma, x : \vec{\tau} \rightarrow \check{\sigma} \vdash M : \vec{\tau} \rightarrow \check{\sigma}^{+} \tag{5}$$

remplace la condition de garde. La deuxième règle permet de typer des définitions récursives imbriquées.

Les types sont interprétés de la façon suivante :

$$[\![\tau]\!]^{\alpha} \ = \ [\![\tau]\!] \text{ si } \tau \text{ est un type de } T \qquad [\![\tau \rightarrow \tau']\!]^{\alpha} \ = \ [\![\tau]\!]^{\alpha [\![\tau']\!]^{\alpha}}$$

$$[\![\check{\sigma}]\!]^{\alpha} \ = \ \mathcal{F}_{\sigma}^{\alpha} \qquad [\![\check{\sigma}^{+}]\!]^{\alpha} \ = \ \mathcal{F}_{\sigma}^{\alpha+1}$$

Nous établissons les résultats suivants.

Validité Si $\Gamma \vdash M : \tau$ alors $(\overline{M}, \overline{M}) \in [\![\tau]\!]^{[\![\Gamma]\!]}$

Etant donnés M et N deux termes de type τ dans Γ, on note $\Gamma \models M = N : \tau$ la condition $\forall \alpha \ ordinal, ([\![M]\!], [\![N]\!]) \in [\![\tau]\!]^{\alpha [\![\Gamma]\!]^{\alpha}}$

Unicité du point fixe Si N et N' sont de type $\vec{\tau} \rightarrow \sigma$ dans Γ, et si M satisfait l'hypothèse (3) alors :

$$\left.\begin{array}{c} \Gamma \models N = M[N/x] : \vec{\tau} \rightarrow \sigma \\ \Gamma \models N' = M[N'/x] : \vec{\tau} \rightarrow \sigma \end{array}\right\} \Rightarrow \Gamma \models N = N' : \vec{\tau} \rightarrow \sigma$$

Normalisation Pour obtenir une théorie équationnelle décidable, on exhibe un système de réécriture dont on prouve des propriétés de confluence et de terminaison. Pour cela, comme proposé dans [Gi96b], `fix` n'est déplié que s'il apparaît

sous un `case`. On prouve alors les propriétés de *réduction du sujet* puis de *normalisation forte*. La preuve de normalisation repose sur la notion de candidats de réductibilité [GLT89].

Il faut mentionner que ces résultats ont été depuis généralisés et étendus aux types inductifs dans [BFGPU04] pour un système permettant une récursion plus générale que la récursion structurelle.

3.4 Vérification de systèmes concurrents

[6] En ce qui concerne la vérification, j'ai élargi l'étude sur les systèmes matériels aux systèmes logiciels parallèles et communicants. Dans ce contexte, les exécutions se poursuivent le plus souvent indéfiniment, tout comme les circuits, après leur mise sous tension, reçoivent sur leurs ports d'entrée un flux infini d'impulsions, chacune d'entre elles les faisant passer d'un état à un autre à chaque *top* d'horloge. Une différence essentielle est cependant le non déterminisme inhérent à de nombreux systèmes concurrents, qui à partir d'un état initial bien déterminé, peuvent s'exécuter de plusieurs façons possibles. Un autre enrichissement est apporté par la communication, synchrone ou asynchrone, entre divers processus pouvant échanger des messages. Ainsi s'est développée une recherche active visant à fournir des langages et des logiques qui permettent de décrire formellement ces aspects et d'établir rigoureusement des énoncés codés par des formules. Tous ces formalismes s'appuient sur la notion de *systèmes de transitions infinis* dans lesquels les exécutions sont représentées par des suites infinies d'états (σ_0, σ_1, ..., σ_n, σ_{n+1} ...) dont chacun se déduit du précédent par une *action* (ou *transition*) du système. Les propriétés de correction s'expriment alors par des relations co-inductives sur les exécutions (simulations, bi-simulations) ou par des propriétés à connotation temporelle, incluant des expressions comme *toujours*, *au bout d'un temps fini*, *infiniment souvent* ... et précisément décrites par des formules de logiques temporelles. C'est ainsi que j'ai débuté cette étude par une formalisation dans le CIC des différentes notions d'équivalence et de congruence (faibles, fortes, observationnelles ...) pour les systèmes de transitions sous-jacents aux algèbres de processus [26] et en collaboration avec Christian Ene (dont j'ai encadré de DEA), une axiomatisation d'Occam en vue de construire des systèmes parallèles par transformations formelles certifiées. Puis je me suis intéressée aux logiques temporelles et à leur plongement dans le CIC.

3.4.1 Axiomatisation de la Logique Temporelle Linéaire

[7] Les diverses logiques temporelles se distinguent selon qu'elles sont propositionnelles, du premier ordre ou d'ordre supérieur, qu'elles incluent ou non des opérateurs passés, qu'elles sont linéaires (LTL) ou "à branchement" (CTL, CTL* ...), que la représentation du temps est discrète ou continue, qu'elles sont

[6] Références [26, 4, 29] de la liste de publications
[7] Références [4, 29] de la liste de publications

à points fixes ou non ... Dans le cas des systèmes à états finis, les formules temporelles peuvent être automatiquement vérifiées. Il existe pour cela un certain nombre d'outils dits de *vérification de modèles* (*model checking*) dont les plus répandus reposent sur des logiques "à branchement" pour des raisons de complexité. Dans le cas de systèmes ayant un nombre infini d'états, la vérification de formules est indécidable, et l'approche *vérification de modèles* doit être abandonnée au profit de la réalisation interactive des preuves au moyen de logiciels. D'où l'intérêt d'implanter les logiques temporelles dans des assistants de preuve. Mon choix s'est porté sur la Logique Temporelle Linéaire (LTL), d'une part parce qu'elle est plus naturelle, plus proche de l'intuition du concepteur et donc plus agréable à utiliser. De plus, associée à l'ordre supérieur, elle offre suffisamment d'expressivité pour couvrir, en pratique, toutes les applications. Enfin, la complexité du problème de la *vérification de modèles*, qui lui fait souvent préférer ses concurrentes, n'entre pas ici en ligne de compte.

Je propose un codage direct de LTL dans le CIC, dans lequel les exécutions infinies des programmes sont représentées par le type co-inductif `Stream` (cf. 3.2.2) des listes infinies d'états. Les opérateurs temporels sont alors codés par des types inductifs ou co-inductifs, selon que ce sont de plus petits ou de plus grands points fixes. Voici en illustration, le traitement des opérateurs *Always* et *Eventually*. Dans ce qui suit `P` désigne un prédicat sur le type `Stream`. Le codage dans le CIC de chaque opérateur est précédé par sa sémantique informelle.

- **L'opérateur** *Always* **:** $\Box$

$\Box$`P`$(\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots)$:= $\forall$`i`$\in I\!N$ `P`$(\sigma_i, \ldots, \sigma_n, \ldots)$

```
CoInductive □P: Stream → Prop :=
            C_always: (∀s:state)(∀σ:Stream)
                      (P (cons s σ)) →(□P σ)  →(□P (Cons s σ))).
```

- **L'opérateur** *Eventually* **:** $\Diamond$

$\Diamond$`P`$(\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots)$:= $\exists$`i`$\in I\!N$ `P`$(\sigma_i, \ldots, \sigma_n, \ldots)$

```
Inductive ◇P : stream → Prop :=
          ev_h : (∀σ:stream)(P σ) →(◇P σ) |
          ev_t : (∀s:state)(∀σ:stream)(◇P σ) →(◇P (cons s σ)).
```

Comme c'était le cas pour les systèmes matériels, on obtient ainsi des spécifcations très pures, dans lesquelles aucun paramètre de temps n'intervient. La structure des termes de preuves reflète fidèlement les opérateurs temporels apparaîssant dans la formule à prouver. Ces preuves sont plus compactes et plus claires que celles obtenues en codant les listes infinies par des fonctions d'un paramètre entier naturel représentant un pas de temps comme dans [Ru92, HC96]. Les règles de LTL correspondent à des lemmes exprimant des propriétés d'invariance, de terminaison, de monotonicité, prouvés par co-induction ou par in-

duction.

Toutefois, il faut mentionner que cette approche ne permet pas la définition d'opérateurs passés. On considère en fait une version "ancrée" de la logique, par opposition à la version "flottante". Dans cette dernière, l'interprétation des opérateurs est basée sur la notion de formules P satisfaites à une position i dans une exécution σ. On écrit $(\sigma,\ i)\ \models P$. Lorsque l'on se restreint à des formules sans opérateur passé, cette version équivaut à ancrer le paramètre i en 0, au lieu de l'autoriser à "flotter". On considère donc uniquement des modèles σ tels que P est satisfait à la position 0. Les formules temporelles sont ainsi les prédicats sur les exécutions et $(\sigma,\ 0)\ \models P$ est exprimé par $(P\ \sigma)$. Cette apparente limitation n'en est pas vraiment une, puisqu'il a été prouvé que ce fragment de LTL est expressivement complet.

Il est à noter que le μ-calcul, qui subsume LTL, CTL et CTL*, a été formalisé dans le CIC. Dans [Mi01], l'auteur présente le plongement dans Coq d'un système de preuve en déduction naturelle pour le μ-calcul propositionnel. Il utilise le CIC comme un *canevas logique* dans lequel il spécifie la syntaxe des formules et la notion de déduction, puis il prouve informellement que le système est correctement représenté. L'intérêt de ce travail est plus théorique que pratique et ne semble pas être utilisable pour la résolution d'un problème en grandeur nature. D'autres études ([Sp98, Ve00]) sont orientées vers la *vérification de modèles* et de ce fait ne s'appliquent qu'à des problèmes à états finis. La démarche que je propose, reposant sur un codage direct de LTL dans le CIC, permet de travailler beaucoup plus confortablement dans la logique sous-jacente au CIC, comme le démontre l'étude de cas relativement complexe qui suit.

3.4.2 Applications aux systèmes embarqués sur cartes à puce

[8] Ce travail a été mené dans le cadre d'une collaboration locale de recherche (*Action Color*) avec la société Gemplus et l'Inria. J'ai encadré pour cela trois étudiants sur des stages Inria. En voici le contexte.

La technologie des cartes à puce est actuellement en pleine expansion. Sont apparues récemment les SmartCards, cartes à puce multi-applications de nouvelle génération, qui permettent notamment le chargement d'applications après émission de la carte. Cette évolution de la technologie est accompagnée d'une intense recherche au niveau international, visant à garantir formellement toutes les propriétés de sécurité et de sûreté souhaitables dans ce domaine hautement critique. Les applications sont écrites en JavaCard, un sous-ensemble du langage Java spécialement dédié aux cartes à puce.

Nous avons prouvé dans le CIC, *la correction de l'algorithme de Kildall* utilisé pour vérifier le bytecode en entrée de la machine virtuelle JavaCard. La

[8] Référence [5] de la liste de publications

difficulté de l'étude provient du fait que l'algorithme est une fonction définie par récursion non structurelle. Il ne peut donc être codé que par un terme du CIC portant dans sa structure une preuve de sa terminaison. Pour cela il a été nécessaire de formaliser en Coq des notions non triviales d'ordres dans des treillis.

Bien que JavaCard permette l'allocation dynamique de mémoire, les SmartCards ne comportent pas pour l'instant de récupérateur de mémoire. Les objets persistants ne sont donc créés par les programmeurs qu'avec parcimonie, ce qui freine considérablement le développement d'applications. Les concepteurs travaillent actuellement à ce problème et c'est à leur demande que nous avons *étudié et vérifié formellement un récupérateur de mémoire* qui pourrait être un bon candidat pour JavaCard.

Nous avons proposé une variante d'un célèbre algorithme à trois couleurs de type *marquage-balayage* [Di78]. Il s'agit d'un algorithme incrémental, propriété nécessaire pour que le système supporte un éventuel arrachement intempestif de la carte. Cette condition permet également d'éviter les délais de réponse inacceptables (plus de quelques secondes) qui pourraient résulter d'une exécution séquentielle d'un cycle complet de récupération de mémoire.

Le système embarqué sur la carte est constitué de deux processus (le programme utilisateur et le récupérateur) qui s'exécutent "en parallèle", par entrelacement non déterministe de leurs actions. Nous le spécifions comme un système de transitions infini en LTL. Nous établissons qu'il est sûr, en ce sens que toute cellule libérée est une cellule morte (inaccessible). Nous prouvons également que toute cellule inaccessible est libérée au bout d'un temps fini à condition toutefois que le récupérateur de mémoire soit appelé infiniment souvent. Nous nous appuyons bien sûr très largement sur l'axiomatisation de LTL évoquée au paragraphe précédent.

La description formelle que nous donnons de l'algorithme est suffisamment abstraite pour recouvrir plusieurs réalisations possibles (une vérification ultérieure d'une réalisation concrète nécessiterait évidemment une preuve formelle de l'adéquation du niveau concret au niveau abstrait). Ainsi, bien qu'initialement motivée par la problématique des cartes à puces, cette étude peut s'appliquer à d'autres cas comme l'Internet par exemple, puisque nous ne supposons pas que la mémoire est finie.

Les travaux sur les preuves formelles de récupérateurs de mémoire font état de récupérateurs à deux couleurs [Ru94, HS97], de récupérateurs à trois couleurs séquentiels [Ja98, VG00], de récupérateurs à trois couleurs dont seul le marquage est incrémental [GBB99], de récupérateur par comptage de références [MD01]. Souvent, les auteurs ne s'intéressent pas à la vivacité. Cette étude est donc la seule à établir formellement la sûreté et la vivacité d'un récupérateur à trois couleurs dont les deux phases de marquage et de balayage sont incrémentales.

On a fait remarquer que les preuves de ce type d'algorithmes sont particulièrement délicates. L'historique des erreurs successives trouvées dans les preuves "à la main", dont les auteurs sont pourtant d'excellents scientifiques, est retracé dans [HS97]. Et l'on prend bien conscience à sa lecture, que l'investissement demandé par la mécanisation des preuves n'est pas vain.

En outre, nous avons été amenés, au cours de ce travail, à donner une *axiomatisation des ensembles finis* dans CIC, nécessaire pour prouver des résultats de terminaison. Contrairement à celle figurant dans les bibliothèques Coq standard [HK95] elle ne fait appel ni à l'axiome du tiers exclus, ni à l'axiome d'extentionnalité. Elle est basée sur la représentation des sous-ensembles finis d'un ensemble A par un prédicat caractéristique $P \circ M$ dont on distingue une partie calculatoire M qui est un programme de type $A \to B$ et une partie logique P de type $B \to Prop$, donc un prédicat sur l'ensemble B.

Notre preuve est donc entièrement constructive. Elle ne comporte qu'une seule hypothèse : la décidabilité de l'égalité sur l'ensemble des cellules de la mémoire. Il est clair que cette hypothèse sera satisfaite par n'importe quel modèle raisonnable de la mémoire d'un ordinateur.

Le développement comporte 10000 lignes de code et 450 lemmes, dont un tiers est relatif à la propriété de sûreté et le reste à la propriété de vivacité. Certaines parties ont pu être automatisées grâce à une tactique *à la Prolog*. Ce travail est décrit en détail dans [5].

3.5 Certification de code mobile (travail en cours)

[9] Ce travail prolonge et approfondit l'étude précédente sur la certification de code mobile. Il est mené en collaboration avec Roberto Amadio et Silvano Dal Zilio, dans le cadre de l'ACI CRISS.

Les codes mobiles récupérés par un système "hôte" (ordinateur, carte à puce, ...), peuvent provenir de sources non fiables voire malveillantes. Il est donc fondamental de maîtriser les problèmes de sécurité liés à cette technologie. On s'est jusqu'à présent beaucoup focalisé sur l'intégrité de l'environnement d'exécution, assurée en général par une analyse statique. Un autre point important est la prévention de fuites d'information d'un niveau de sécurité vers un autre moins élevé : elle résulte de preuves de non interférence. Il est également intéressant de pouvoir établir un contrôle des ressources mémoire utilisées par l'exécution.

Nous nous intéressons en particulier à la production automatique d'une borne de l'espace mémoire occupé pendant l'exécution du code. La plupart des

[9] Références [15, 19] de la liste de publications

travaux sur ce thème [BC92, Ho02, Jo97, Le94] traitent de langages fonctionnels du premier ordre. Ils ont pour point de départ la caractérisation des fonctions polynômiales par récursion bornée sur la notation, donnée par Cobham [Cob65], et proposent diverses techniques d'inférence pour certaines classes de programmes.

Nous travaillons donc sur un langage fonctionnel du premier ordre, pour lequel nous proposons trois sortes d'analyses : une analyse de types, une analyse de la taille des valeurs stockées en mémoire et une analyse assurant la terminaison. La première est maintenant classique et est notamment utilisée dans la plateforme Java ; la seconde est basée sur la notion de quasi-inteprétation introduite dans [MA00] ; la troisième fait intervenir des techniques empruntées au domaine de la réécriture mettant en œuvre des RPO (recursive path ordering). La combinaison des trois analyses permet d'obtenir une borne des ressources utilisées. Il a été en notamment prouvé qu'un programme admettant une quasi-interprétation polynômiale et dont la terminaison est établie à l'aide d'un RPO lexicographique, s'exécute en espace polynômial [BMM01]. Toutefois, c'est du bytecode (supposé provenir d'une source non sûre) que nous traitons et c'est donc à ce niveau que nous travaillons pour exploiter ces résultats et effectuer ces analyses.

3.5.1 Le langage de haut niveau et les instructions du bytecode

Le langage source est un langage fonctionnel du premier ordre, avec types (mutuellement) inductifs et fonctions définies par une suite de règles de filtrage de la forme : $f(p_1, \ldots, p_n) = e$, où e est une expression et $p_1, \ldots, p_n$ sont des filtres linéaires (une variable apparaît au plus une fois). Un exemple de programme évaluant des expressions booléennes est présenté sur la figure 11.

Chaque fonction est compilée en un code exécutable par une machine virtuelle. Lors de l'exécution, à chaque appel de fonction f, un bloc d'activation *(f, pc, P)* est mis en place : f est le nom de la fonction, *pc* le compteur de programme qui indique le numéro de l'instruction courante (initialement 0), et P est une pile de valeurs, initialisée avec les arguments de la fonction. Comme illustration, la figure 12 montre un bytecode possible pour la fonction *member* du programme de la figure 11, ainsi qu'une exécution symbolique de la fonction sur l'entier n et l'environnement l. Sur chaque ligne se trouve la valeur p du compteur ordinal, l'instruction correspondante et la pile des expressions (le sommet est à droite), sur laquelle s'exécute l'instruction de rang p.

On y distingue les instructions suivantes dont on décrit informellement la sémantique opérationnelle :

- L'instruction $Load\ (n)$ empile sur P l'élément de rang n, en comptant à partir du bas de la pile. *pc* est incrémenté.
- L'instruction $Branch(C, l)$ filtre l'élément x en sommet de pile avec le constructeur C d'arité n. Si le filtrage réussit, x est déconstruit et ses n

```
type  bool = T | F ;;
type  nat  = Z  | S of nat ;;
type  env  = Nil | C of nat * env ;;
type  form = Var of tnat
           |   Not of form
           |   Or of form * form
           |   Ex of nat * form ;;

not (T) = F        or (T,y) = T       eq (Z,Z)       = T
not (F) = T        or (F,T) = T       eq (Z,S(y))    = F
                   or (F,F) = F       eq (S(x),s(y)) = eq(x,y)

member (x,Nil)     = F
member (x,C(y,l1)) = or(eq(x,y),member(x,l1))

check (Var(x), l)    = member(x,l)
check (Not(f1), l)   = not(check(f1,l))
check (Or(f1,f2), l) = or(check(f1,l),check(f2,l))
check (Ex(x,f1), l)  = or(check(f1,l),check(f1, C(x,l)))

qbf (f) = check(f,Nil)
```

FIG. 11 – Un programme d'évaluation de formules booléennes, et le bytecode de la fonction *member*

arguments sont empilés à sa place, *pc* est incrémenté. Si le filtrage échoue, *pc* prend la valeur l.
- L'instruction $Build(C, n)$, où C est un constructeur d'arité n, prélève n valeurs $v_1, \ldots, v_n$ sur P et empile $C(v_1, \ldots, v_n)$. *pc* est incrémenté.
- L'instruction *Return* renvoie à l'environnement le résultat prélevé au sommet de la pile P. La fonction courante est désactivée.
- L'instruction *Stop* arrête l'exécution en cours.
- L'instruction $Call(g, n)$ empile, au dessus du bloc d'activation de la fonction appelante, un nouveau bloc d'activation pour l'exécution de la fonction g d'arité n. Le compteur ordinal y vaut 0 et la pile d'exécution y est constituée de n valeurs, prélevées au sommet de la pile de valeurs de la fonction appelante.

Dans ce contexte d'exécution symbolique (par opposition à exécution effective), à chaque instruction *Branch* est associée une substitution de la variable en sommet de pile, trace d'un filtrage réussi : cette substitution σ_p est indiquée en regard de chaque ligne de rang p.

```
member:
0:  Load(1);           n  l
1:  Branch("Nil",4);   n  l       l
2:  Build("F",0);      n  Nil                                          l=Nil
3:  Return;            n  Nil     F                                    l=Nil
4:  Branch("C",13);    n  l       l
5:  Load(0);           n  C(h,t)  h  t                                 l = C(h,t)
6:  Load(2);           n  C(h,t)  h  t  n                              l = C(h,t)
7:  Call("eq",2);      n  C(h,t)  h  t  n           h                  l = C(h,t)
8:  Load(0);           n  C(h,t)  h  t  eq(n,h)                        l = C(h,t)
9:  Load(3);           n  C(h,t)  h  t  eq(n,h)     n                  l = C(h,t)
10: Call("member",2);  n  C(h,t)  h  t  eq(n,h)     n     t            l = C(h,t)
11: Call("orb",2);     n  C(h,t)  h  t  eq(n,h)     member(n,t)        l = C(h,t)
12: Return;            n  C(h,t)  h  t  orb(eq(n,h),member(n, t))      l = C(h,t)
13: Stop;              n  l       l
14: Return;            Bot
```

FIG. 12 – Exécution symbolique de la fonction *member*

3.5.2 Exécution abstraite et exécution symbolique

Ce bytecode est supposé accompagné d'annotations. Il s'agit tout d'abord des signatures des fonctions et des constructeurs de type, en vue d'une analyse de type. Cette analyse consiste en une exécution abstraite du bytecode dans laquelle les valeurs sont remplacées par leur type (cf. figure 13).

Pour chaque instruction, elle s'assure de la compatibilité des types empilés avec l'instruction courante, en référence aux signatures des constructeurs de types et des fonctions du programme. En cas d'erreur, l'algorithme produit un état $\top$, traité comme le plus grand élément du treillis constitué par les états abstraits. Un état $\bot$ n'exprimant aucune contrainte sur les types, est le plus petit élément du treillis. On démontre que le résultat de l'exécution abstraite ne produit pas l'élément $\top$, si et seulement si le programme est correct en ce qui concerne le typage. Ceci assure par exemple qu'il n'y aura pas de tentative d'accès à des éléments de mémoire situés en dehors des bornes du tableau dans lequel est stockée la pile courante de valeurs. De même, chaque appel de fonction s'applique à des éléments correctement typés, chaque filtrage met en jeu une valeur et un constructeur du même type, chaque nouvelle valeur est construite avec un constructeur et des arguments dont les types sont cohérents ...

On peut aussi en déduire une indication utile au contrôle de ressources. En effet, après la vérification des types, on connaît exactement le nombre maximal de valeurs empilées pendant l'exécution d'une fonction : c'est la hauteur de pile maximale. Il suffit donc de savoir borner la taille de ces valeurs pour en déduire une borne de l'espace mémoire occupé par le bloc d'activation d'une fonction.

```
not   : bool -> bool            or     : (bool, bool) -> bool
eq    : (nat,  nat) -> nat      member : (nat,   env) -> bool
check : (form, env) -> bool     qbf    : form -> bool

member:
0:   tnat  env
1:   tnat  env   env
2:   tnat  env
3:   tnat  env   bool
4:   tnat  env   env
5:   tnat  env   tnat  env
6:   tnat  env   tnat  env   tnat
7:   tnat  env   tnat  env   tnat  tnat
8:   tnat  env   tnat  env   bool
9:   tnat  env   tnat  env   bool  tnat
10:  tnat  env   tnat  env   bool  tnat  env
11:  tnat  env   tnat  env   bool  bool
12:  tnat  env   tnat  env   bool
13:  tnat  env   env
14:  Bot
```

FIG. 13 – Exécution abstraite de la fonction *member*

Les deux exécutions, abstraite et symbolique, sont deux instances du célèbre algorithme d'analyse de flots de données dû à Kildall ([Ki73]) et déjà cité au paragraphe précédent. L'analyse symbolique est effectuée après que la vérification de type ait réussi. On démontre qu'elle ne produit pas d'erreurs si et seulement si le graphe de flot de la fonction est un arbre et qu'aucun appel de fonction ne précède un filtrage. Le résultat de la vérification symbolique est également utilisé pour le contrôle des ressources mémoire, comme expliqué dans le paragraphe suivant.

3.5.3 Contrôle de la taille des valeurs en mémoire

Le but est donc de borner la taille des valeurs en mémoire. On utilise pour cela des *quasi-interprétations* ([MA00, AM03]) de fonctions. Intuitivement, cet outil permet d'exprimer une borne de la taille du résultat en fonction de la taille des arguments. Plus précisément, on associe à chaque constructeur c et fonction f des fonctions q_{c}, q_f dont les arguments et la valeur sont des réels positifs et telles que :

(i) si c est une constante, q_{c} est constante et égale à 0,

(ii) si c est un constructeur d'arité $n \geq 1$ alors q_{c} est la fonction $(\mathbb{R}^+)^n \to \mathbb{R}^+$

définie par $q_{\mathsf{c}}(x_1, \ldots, x_n) = d + \Sigma_{i \in 1..n} x_i$, avec $d \geq 1$, et

(iii) si f est une fonction d'arité n, alors $q_f : (\mathbb{R}^+)^n \to \mathbb{R}^+$ est monotone et telle que pour tout $i \in 1..n$, $q_f(x_1, \ldots, x_n) \geq x_i$.

On étend cette notion à une expression quelconque en posant :

$$q_x = x, \qquad q_{\mathsf{c}(e_1,\ldots,e_n)} = q_{\mathsf{c}}(q_{e_1}, \ldots, q_{e_n}), \qquad q_{f(e_1,\ldots,e_n)} = q_f(q_{e_1}, \ldots, q_{e_n}).$$

On dit que q est une quasi-interprétation, si pour toute règle $f(p_1, \ldots, p_n) = e$ du programme, l'inégalité

$$q_{f(p_1,\ldots,p_n)} \;\geq\; q_e$$

est satisfaite.

Exemple. Une quasi-interprétation pour le programme de la figure 11 est donnée par :

$$q_S = q_{Var} = q_{Not} = x + 1 \qquad q_C = q_{Or} = q_{Ex} = x + y + 1$$

$$q_{not} = q_{qbf} = x \qquad q_{or} = q_{eq} = q_{mem} = max(x, y) \qquad q_{check} = x + y$$

La taille $|e|$ d'une expression e est définie par 0 si e est une constante et par $1 + \Sigma_{i \in 1..n} |e_i|$ if e est de la forme $\mathsf{c}(e_1, \ldots, e_n)$ ou $f(e_1, \ldots, e_n)$. On démontre alors que pour toute valeur v, $|v| \leq q_v$.

Borne de la taille des expressions en mémoire. On démontre que, lors de l'évaluation de $f_0(u_0, \ldots, u_n)$ par la machine virtuelle, la taille des expressions en mémoire est bornée par :

$$q_{f_0(u_0,\ldots,u_n)}$$

à condition que pour toute fonction f du programme, pour toute instruction p du bytecode de f, pour toute expression e figurant dans la pile de rang p dans l'exécution symbolique de la fonction sur les variables $x_0, \ldots, x_m$:

$$q_e \leq q_{f(\sigma_p(x_0),\ldots,\sigma_p(x_m))}$$

3.5.4 Majoration de l'espace occupé

Nous disposons pour tout programme vérifiant les conditions précédentes d'une borne :

$$b = q_{f_0(u_0,\ldots,u_n)}$$

des valeurs stockées en mémoire à un moment donné de l'exécution. Nous disposons également du nombre maximal de valeurs stockées dans chaque bloc d'activation : il s'agit de la hauteur maximale h des piles de valeurs de chaque

fonction. Comme nous l'avons vu, cette valeur h peut être calculée dès l'étape de vérification statique. Il suffit donc alors de produire un majorant M du nombre de blocs d'activation pendant l'exécution, pour borner la taille de l'espace mémoire, une borne étant dans ce cas $h \times b \times M$.

Pour établir la terminaison de l'évaluation d'une expression, on utilise une technique employée en réécriture. Il est en effet prouvé [Gra96] que si l'on suppose que les règles de définitions de fonctions $f(p_1, \ldots, p_n) = e$ sont orthogonales, la terminaison de la stratégie d'appels par valeur est équivalente à la terminaison du système de réécriture obtenu en les interprétant comme des règles de réécriture $f(p_1, \ldots, p_n) \to e$. Cette méthode est fondée sur les RPO *Recursive Path Ordering*) [BN98].

On suppose disposer d'une relation de pré-ordre $\leq_\Sigma$ sur les symboles de fonctions telle que si f appelle g, alors $g \leq_\Sigma f$ et donc si f et g sont mutuellement récursives, alors $f =_\Sigma g$. On suppose de plus dans ce dernier cas que f et g ont même arité. On étend le pré-ordre aux symboles de constructeurs, en supposant qu'ils sont incomparables entre eux et toujours inférieurs aux symboles de fonctions. On note $<_l$ le RPO associé quand on donne à chaque symbole de fonction le status lexicographique et à chaque constructeur le status produit.

Borne du nombre de blocs d'activation. On démontre que, si a est la plus grande arité de fonction du programme et si b majore la taille des expressions en mémoire, le nombre de blocs d'activation est borné par $M = b^a$ à condition que pour toute fonction f du programme, pour toute instruction p du bytecode de f, pour toute expression e figurant dans la pile de rang p dans l'exécution symbolique de la fonction sur les variables $x_0, \ldots, x_m$:

$$e <_l f(\sigma_p(x_0), .., \sigma_p(x_m)).$$

On en déduit que la somme des tailles des valeurs stockées en mémoire pendant l'exécution n'éxcède pas $h \times b \times b^a$. Cette borne est donc polynômiale en la taille des données du programme, dès que la quasi-interprétation de la fonction appelée est elle-même bornée par un polynôme.

3.5.5 Conclusion

Une étude sur le contrôle des ressources est menée dans le projet MRG [Sa01] et repose sur la notion de typage linéaire. Sur le même thème, Marion et Moyen [MM03] analysent les ressources d'une machine à compteurs par réduction à un certain type de réseaux de Petri. Ils ne travaillent que sur des entiers et la pile ne contient que des adresses de retour. Je travaille actuellement à l'implantation des algorithmes de vérification ainsi qu'à leur spécification et leur preuve en Coq.

4 Conclusion

Les travaux décrits dans ce document se situent à la charnière de la logique et de l'informatique. Ils trouvent leur inspiration et leur justification dans l'exploitation pratique, désormais en prise directe avec les besoins de l'entreprise, de résultats puissants mais pointus, issus d'études théoriques.

Mes premiers travaux en programmation en logique relèvent de cette préoccupation. La normalisation de λ-termes sous forme d'arbres rationnels a été appliquée au traitement des langues naturelles et intégrée dans un système d'interrogation en Français d'une banque de données.

La richesse du Calcul des Constructions Inductives en fait tout son intérêt, mais également toute sa difficulté. Certes, on peut arguer que le système Coq manque encore d'automatisation, de procédures de décision, de convivialité. Je reste cependant convaincue que, quelles que soient les futures les évolutions du système, on ne pourra espérer tirer pleinement avantage de ses spécificités dans une démarche naïve, non étayée par une bonne connaissance de la logique sous-jacente. La conduite de preuves formelles dans Coq demande du temps et de l'expertise. En revanche, le système se prêtant bien à des démarches très générales, une façon appréciable de corriger ces inconvénients est de mettre à disposition des utilisateurs, des logiques et théories pré-implantées. Une part importante de cette contribution réside ainsi dans diverses axiomatisations relatives aux *listes dépendantes, systèmes de numération, automates, logiques temporelles, treillis, ensembles finis*. Se situant à un haut niveau d'abstraction, elles couvrent un grand nombre d'applications potentielles dans des domaines très concrets : *circuits combinatoires, circuits séquentiels synchrones, algorithmes concurrents, récupération de mémoire, vérification de bytecode JavaCard...* Le praticien dispose ainsi d'une panoplie d'outils efficaces et prêts à l'emploi, mettant en œuvre les particularités d'un système logique puissant et expressif. Interfaces entre le système Coq et l'utilisateur, ces axiomatisations se devaient d'être claires et naturelles. De plus, elles induisent souvent entièrement la méthodologie de preuve. C'est particulièrement manifeste dans le cas du traitement des circuits séquentiels par des automates. Ces études ont donc été conduites dans le souci de produire des outils puissants et généraux, mais d'utilisation agréable et proches de l'intuition.

Elles ont été appliquées dans un contexte industriel, à des études de cas en grandeur réelle, impliquant mes étudiants. Qu'il s'agisse des circuits ou de l'algorithme de récupération de mémoire, elles illustrent de façon pertinente le bien fondé et la praticabilité des méthodologies proposées. Certains de ces travaux ont eu lieu dans le cadre d'une collaboration régionale avec la société Gemplus et l'INRIA Sophia-Antipolis. A cette occasion, j'ai été amenée à créer, former et diriger une petite équipe en partie financée par l'INRIA. Une partie de ces travaux a également pu être menée à bien lors d'un séjour d'un an que j'ai effectué comme chercheur détaché à l'INRIA.

J'ai été également amenée, au gré de collaborations, à aborder des problèmes complètement théoriques, comme la sémantique des types co-inductifs. C'est un aspect particulièrement stimulant de mon activité de recherche de pouvoir établir un pont entre les aspects les plus abstraits et les applications.

5 Références

[AC98] R. Amadio et P.L. Curien. Domains and Lambda-Calculi. Cambride University Press, 1998.

[AL92] M. Aagaard and M. Leeser. A Methodology for Reusable Hardware Proofs. *International Workshop on Higher Order Logic Theorem Proving and its Applications*, L. Claesen and M. Gordon editors, 1992.

[AM03] R. Amadio. Max-plus quasi-interpretations. *Typed Lambda Calculi and Applications, TCLA'03*, LNCS 2701, Springer, 2003.

[An96] V. Antimov. Partial Derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155, pp. 291-319, 1996.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2, pp. 97-110, 1992.

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[Be96] F. Benhamou and al. *Le manuel de Prolog IV*. PrologIA, Marseille, 1996.

[BFGPU04] G. Barthe, M. J. Frade, E. Gimnez, L. Pinto and T. Uustalu. Type-based termination of recursive definitions. *In Mathematical Structures in Computer Science*. To appear.

[BH01] S. Boulmé and G. Hamon. Certifying Synchrony For Free. *Logic for Programming, Aritificial Intelligence and Reasoning, LPAR 2001*, 2001, Havana, Cuba.

[BMM01] G. Bonfante, J.-Y. Marion and J.-Y. Moyen. On termination methods with space bound certifications. In *Andrei Ershov 4th International Conference "Perspectives of System Informatics"*, pp. 482-493, 2001.

[Bo67] T. L. Booth. Sequential machines and automata theory. Ed. John Wiley and Sons, Inc., 1967.

[Br64] J. A. Brzozowski. Derivatives of regular expressions. *Journal of ACM*, Vol 11, Num 4, pp. 481-494, 1964.

[BR93] P. Brisset and O. Ridoux. The Compilation of λProlog and its execution in Mali. *Technical Report*, Num 1831, INRIA, 1993.

[BS86] G. Berry and R. Sethi. From regular expressions to deterministic automata (note). *Theoretical Computer Science*, Vol 48, Num 1, pp. 117-126, 1986.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic.* Vol 1, North Holland, 1958.

[CGM87] A. Camilleri, M. Gordon, and T. Melham. Hardware Verification Using Higher Order Logic. *From HDL Descriptions to Guaranteed Correct Circuit Designs*, Elsevier Scientific Publishers, 1987.

[Ch41] A. Church. The Calculi of Lambda Conversion. *Annals of math. studies*, Vol 6, 1941, Princeton University Press.

[CH85] T. Coquand and G. Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. *European Conference on Computer Algebra, EUROCAL 85* , Linz, Austria. Springer-Verlag LNCS 203, 1985.

[Ch91] J.M. Champarnaud, G. Hansel. AUTOMATE, a Computing Package for Automata and Finite Semigroups. *JSC*, Vol 12, Num 2, pp. 197-220, 1991.

[Ch01] J.M. Champarnaud. Evaluation of three implicit structures to implement nondeterministic automata from regular expressions. *International Journal of Foundations of Computer Science*. Vol 13, Num 1, pp. 99-103, 2002.

[CKV83] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, bases théoriques et développements actuels. *TSI*, Vol 2, Num 4, 1983.

[Cob65] A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Logic, Methodology, and Philosophy of Science II, North Holland*, 1965.

[Col82a] A. Colmerauer. Prolog and infinite trees. *Logic Programming*, K. L. Clark et J. A. Tarnlund ed., Academic Press, 1982.

[Col82b] A. Colmerauer. An interesting subset of natural language. *Logic Programming*, K. L. Clark et J. A. Tarnlund ed., Academic Press, pp. 45-66 1982.

[Col84] A. Colmerauer. Equations and Inequations on Finite and Infinite Trees. *Fifth Generation Computer Systems*, pp. 85-99, 1984.

[Col90] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, Vol 33, Num 7, pp. 68-90, 1990.

[Col92] A. Colmerauer. Les systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. *Traitement Automatique des Langues*. Vol 33, Num 1-2, pp. 105-148, 1992, (publication d'un rapport technique de l'Université de Montréal, 1970).

[Coq01] The Coq development team. The Coq Proof Assistant Reference Manual - Version 7.1. *Technical Report*, LogiCal Project-INRIA, 2001.

[Coq93] T. Coquand. Infinite objects in type theory. *Types for proofs and programs, Types'93*, Series LNCS, Vol 806, pp. 63-78, 1993.

[CP92] C.-H. Chang and R. Paige. From regular expression to DFA's using compressed NFA's. *Theoretical Computer Science*, Num 178, pp. 1-36, 1997.

[Cu94a] P. Curzon. The Formal Verification of the Fairisle ATM Switching Element. *Rapport technique 329*, University of Cambridge, 1994.

[Cu94b] P. Curzon. Experiences Formally Verifying a Network Component. *Proceedings of the 9th Annual IEEE Conference on Computer Assurance*, IEEE Press, pp. 183-193, 1994.

[CZ01] J.M. Champarnaud, D. Ziadi. Computing an Equation Automaton of a Regular Expression in $O(s^2)$ Space and Time. *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001.* Jerusalem, Israel, LNCS 2089, pp. 157-168, 2001.

[DB72] N. G. De Bruijn. Lambda Calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag Math.*, Vol 34, pp. 381-392, 1972.

[Di78] E. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, E.F.M. Steffens. On the fly garbage collection : an exercise in cooperation. *Communications of the ACM* Vol 21, Num 11, pp. 966-975, 1978.

[DS88] V. Dahl, P. Saint-Dizier, Eds. *Proceedings of the 1st Workshop on Natural Language Understanding and Logic Programming.* North-Holland, Amsterdam, 1985.

[DS88] V. Dahl, P. Saint-Dizier, Eds. *Proceedings of the 2nd Workshop on Natural Language Understanding and Logic Programming.* North-Holland, Amsterdam, 1988.

[GBB99] H. Goguen, R. Brooksby, R. Burstall. An abstract approach to memory management. *In the proceedings of Types*, Vol 1956, pp. 148-161, 1999

[GG91] S. Garland and J.Guttag. A guide to LP, the Larch Prover. *DEC System Reseach Center*, Report 82. Palo Alto, 1991.

[Gi71] J.-Y. Girard. Une extension de l'interprétation de Godel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. *Proceedings of the Second Scandinavian Logic Symposium*, In J. E. Fenstad, editor, . North-Holland Publishing Co., pp. 63-92, 1971.

[Gi96a] E. Giménez. Codifying guarded definitions with recursive schemes. *BRA workshop on Types for Proofs and Programs*, Springer-Verlag LNCS 996, pp. 39-59, 1996.

[Gi96b] E. Giménez. Un calcul de constructions infinies et son application à la vérification de systèmes communicants. *Thèse d'université*, ENS-Lyon, 1996.

[GLSS92] A. Gal, G. Lapalme, P. Saint-Dizier, and H. Somers. Prolog for natural language processing. *Wiley*, New-York, 1992.

[GLT89] J.-Y. Girard, Y. Lafon, and P.Taylor. Proofs and Types. *Cambridge University Press*, 1989.

[Go86] M. Gordon. Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware. *Proceedings of the 1985 Edinburgh Workshop on VLSI Design : Formal Aspects of VLSI Design* Edited by G.Milne and P. A. Subrahmanyam, North Holland, pp. 153-177, 1986.

[GR96] E. Garcez, W. Rosenstiel. The Verification of an ATM Switching Fabric using the HSIS Tool. *IX Brazilian Symposium on the Design of Integrated Circuits*, 1996.

[Gra96] B. Gramlich. On proving termination by innermost termination. In *RTA, Int. Conf. on Rewriting Techniques and Applications*, LNCS 1103, pp. 93–107, 1996.

[GS91] J. Groenendijk and M. Stokhof. Dynamic Predicate Logic. *Linguistics and Philosophy*, Vol. 14, pp. 39-100, 1991.

[HC96] B. Heyd et P. Crégut. A Modular Coding of Unity in Coq. *The 1996 International Conference on Theorem Proving in Higher Order Logics, TPHOL'96*, pp. 251-266. Turku, 1996.

[HD92] F.K. Hanna and N. Daeche. Dependant types and formal synthesis. *Phil. Trans. R. Soc. Lond.*, Vol. 339, pp. 121-135, 1992.

[HDL90] F.K. Hanna, N. Daeche, and M. Longley. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, 16 (1), pp. 949-964, 1990.

[HK95] G. Huet and G. Khan. The Coq Standard Library. *http ://coq.inria.fr/library/SETS*, 1995.

[Ho02] M. Hofmann. The strength of non size-increasing computation. In *Proc. POPL*,pp. 260-269, 2002.

[HS97] K. Havelund, N. Shankar. A Mechanized Refinement Proof for a Garbage Collector. *Manuscrit non publié*, 1997.

[HU79] L. Hopcroft, L. Ullman. Introduction to automata theory, languages and computation. Addison-Wesley Publishing Company, 1979.

[Hu76] Gérard Huet. Résolution d'équations dans des langages d'ordre 1,2, ..., ω. *Thèse d'Etat*, Universit é Paris 7, 1976.

[Hu89] W.A. Hunt. Microprocessor Design Verification. *Journal of Automated Reasonning*, 5 (4), pp. 429-460, 1989.

[Ja98] P.B. Jackson. Verifying a Garbage Collector Algorithm. *11th International Conference in Theorem Proving in Higher Order Logics, TPHOL'98*, Series LNCS, Vol 1479,pp. 225-244, 1998.

[Jo97] N. Jones. *Computability and complexity, from a programming perspective*. MIT-Press, 1997.

[Ka81] H. Kamp. A theory of truth and semantic representation. *Formal Methods in the Study of Language*, Ed. Groenendijk et al., Mathematisch centrum, Amsterdam, pp. 277-322, 1981.

[Ki73] G.A. Kildall. A unified approach to global program optimization. *ACM Symposium on Principles of Programming Languages*, pp. 194-206, 1973.

[KR93] H. Kamp and U. Reyle. From Discourse to Logic. *Studies in Linguistics and Philosophy*. Vol.42, Kluwer Academic Publishers, Dordrecht,pp. 305-711, 1993.

[KTC03] S. Kort, S. Tahar, P. Curzon. Hierarchical formal verification using a hybrid tool. *International Journal Softw. Tools Technol. Transfer*, pp. 313-322, 4, 2003.

[KV76] R. Kowalski and M. Van Emden. The semantics of predicate logic as programming language. *Journal of ACM*, Vol 23, Num 4, pp. 733-743, 1976.

[La] Lava. *The Lava Homepage.* http ://www.cs.chalmers.se/∼koen/Lava, 2000.

[Le94] D. Leivant. Predicative recurrence and computational complexity i : word recurrence and poly-time. *Feasible mathematics II, Clote and Remmel (eds.)*, Birkhäuser, pp. 320-343, 1994.

[LLR93] S. Le Huitouze, P. Louvet, and O. Ridoux Logic Grammars and λProlog. *10th Int. Conference Logic Programming*, MIT Press, Cambridge, MA, pp. 64-79, 1993.

[LT98] J. Lu, S. Tahar. Practical Approaches to the Automatic Verification of an ATM Switch using VIS. *IEEE 8th Great Lakes Symposium on VLSI (GLS-VLSI'98)*, pp. 368-373, Lafayette, USA, 1998.

[MA00] J.Y. Marion. Complexité implicite des calculs, de la théorie à la pratique. *Mémoire d'habilitation à diriger les recherches*, Nancy, 2000.

[MD01] L. Moreau, J. Duprat. A Construction of Distributed Reference Counting. *Acta Informatica*, Vol 37, pp. 563-595, 2001.

[Me55] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34 :5, pp. 1045-1079, 1955.

[Mi01] M. Miculan. On the Formalization of the Modal μ-Calculus in the Calculus of Inductive Constructions. *Information and Computation*, 164 :1, pp. 199-231,2001.

[MJ96] P. S. Miner and S. D. Johnson. Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit. *Designing Correct Circuits*, Båstad, 1996.

[MM03] J.-Y. Marion, J.-Y. Moyen. *Termination and resource analysis of assembly programs by Petri Nets.* Technical Report, Université de Nancy, 2003.

[MNS87] D.A. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, Vol 51, pp. 98-105, 1991

[Mo56] E. F. Moore. Gedanken-Experiments on Sequential Machines. *Automata studies*, Princeton University Press, ed. C.E. Shannon and J. McCarthy, pp. 129-153, 1956.

[Mo74] R. Montague. Formal Philosophy : Selected Papers of Richard Montague. Yale University Press, New Haven, Co, USA, 1974.

[Mo02] J. F. Monin. Contribution aux mthodes formelles pour le logiciel. Habilitation à Diriger des Recherches, Univ. Paris-Sud, 2002.

[MY60] R. Mac Naughton and H. Yamada. Regular expressions and state graphs for automata. *IRE trans. on Electronic Computers*, Vol 9, Num 1, pp. 38-47, 1960.

[Ne97] G. Necula. Proof Carrying Code. *Proc. POPL*, ACM Press,PP. 106-119, 1997.

[Pa87] J.-L. Paillet. A functional model for descriptions and specifications of digital devices. *System Description and Design Tools*, IFIP WG 10.2 Workshop.

[Pa95] C. Paulin-Mohring. Circuits as Streams in Coq : Verification of a Sequential Multiplier. *Basic Research Action "Types"*,PP. 216-230, 1995.

[Pa96] C. Paulin-Mohring. Définitions Inductives en Théorie des Types d'Ordre Supérieur. *Habilitation à diriger les recherches*, Université Claude Bernard Lyon I, 1996.

[Pe91] F. Pereira. Semantic interpretation as higher-order deduction. *Logics in AI : European Workshop JELIA'90*, Jan van Eijck, editor. Num 478 in LNCS, pp. 78-96, 1991.

[PM90] R. Pareschi and D. Miller. Extending Definite Clause Grammars with Scoping Constructs. *International Conference in Logic Programming*, edited by David H. D. Warren and Peter Szeredi, pp. 373-389, MIT Press, 1990.

[PS98] R. Pasero and P. Sabatier. Illico : Un système générique pour la compréhension d'un sous-ensemble du français. *Rapport de Recherche LIM*, 192.

[Ro65] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of ACM*, Vol 12, Num 1, pp. 227-234, 1965.

[Ru92] D. Russinoff. A Verification System for Concurrent Programs Based on the Boyer-Moore Prover. *Formal Aspects of Computing*, Vol 4, pp. 597-611, 1992.

[Ru94] D. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects of Computing*, Vol 6, pp. 359-390, 1994.

[Sa01] D. Sannella. Mobile resource guarantee. IST-Global Computing research proposal, U. Edinburgh,2001. `http ://www.dcs.ed.ac.uk/home/mrg/`.

[SK95] K. Schneider, T. Kropf. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. *International Workshop on Higher Order Logic Theorem Proving and Its Applications : B-Track : Short Presentation*, pp. 89-104, 1995.

[Sp98] C. Sprenger. A Verified Model Checker for the Modal μ-Calculus in Coq. *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, LNCS 1384, pp. 167-183, 1998.

[TC99] S. Tahar, P. Curzon, Paul. Comparing HOL and MDG : a Case Study on the Verification of an ATM Switch Fabric. *Nordic Journal of Computing*, 6 :4, pp. 372-402, 1999.

[TCL98] S. Tahar, P. Curzon, J. Lu. Three Approaches to Hardware Verification : HOL, MDG and VIS Compared. *Formal Methods in Computer-Aided Design*, LCNS 1522, pp. 433-450, 1998.

[Th68] K. Thomson. Regular expression search algorithm. *Communications of the ACM*, Vol 11, Num 6, pp. 419-422, 1968.

[TSC99] S. Tahar, X. Song, E. Cerny, M. Langevin, O. Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18 :7, pp. 956-972, 1999.

[VG00] K.N. Verma, J. Goubault-Larrecq, S. Prasad, S. Arun-Kumar. Reflecting BDDs in Coq. *In 6th Asian Computing Science Conference (ASIAN'2000)*, LNCS 1961, pp. 162-181, 2000.

[Ve00] K.N. Verma. Reflecting symbolic model checking in Coq. *Mémoire de DEA*, DEA Programmation, Paris, September 2000, 43 pages.

6 Liste de Publications

Thèse

[1] Solange Coupet-Grimal. Deux arguments pour les arbres infinis en Prolog. *Thèse de doctorat de l'Université Aix-Marseille II*, 1989.

Revues

[2] Solange Coupet-Grimal. Prolog infinite trees and automata. *RAIRO : Informatique Théorique et Applications*, Vol 25, No 5, 1991, pp 397-418.

[3] Solange Coupet-Grimal and Olivier Ridoux. On the use of advanced logic programming features in computational linguistics. The *Journal of Logic Programming*,Vol 24, No 1-2, pp. 123-162, July-August. 1995.

[4] Solange Coupet-Grimal. An axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions. The *Journal of Logic and Computation*, Vol 13, No 6 pp. 801-813, Oxford University Press 2003.

[5] Solange Coupet-Grimal and Catherine Nouvet. Formal verification of an incremental garbage collector. *Journal of Logic and Computation*, Vol 13, No 6, pp. 815-833, Oxford University Press 2003.

[6] Solange Coupet-Grimal and Line Jakubiec. Circuit certification in type theory. *Formal Aspects of Computing*, à paraître.

Conférences avec Comité de Lecture

[7] Solange Coupet-Grimal. Représentation sémantique dans le traitement des langues naturelles en Prolog. *Secondes journées francophones de programmation en logique, JFPL'93*, Telkea, Nimes, 1993. ISBN 2-87717-034-9, pp. 69-91.

[8] Michel Allemand, Solange Coupet-Grimal, and Jean-Luc Paillet. FORMATH a system for modelling and proving circuits. *Correct Hardware Design and Verification Method*, CHARME'95, Frankfurt, 1995. Poster.

[9] Michel Allemand, Solange Coupet-Grimal, and Jean-Luc Paillet. A System for modelling and proving circuits. *European Design and Test Conference and Exhibition ED&TC'96*, IEEE Computer Society Press, pp. 605, 1996.

[10] Michel Allemand, Solange Coupet-Grimal, and Jean-Luc Paillet. A formal system for correct hardware design. *Advanced Technology Workshop, ATW'96*, Kluwer Academic Press, 1996, pp. 45-71

[11] Solange Coupet-Grimal and Line Jakubiec. Coq and Hardware Verification : a Case Study. *International Conference on Theorem Proving in Higher Order Logics, TPHOL'96*, Turku, Finlande, LNCS, Springer Verlag, Août 1996, pp. 125-139.

[12] Solange Coupet-Grimal, Paul Curzon, and Line Jakubiec. A Comparison of the Coq and HOL Proof Systems for specifying Hardware. Supplementary proceeding of *The 10th International Conference on Theorem Proving in*

Higher Order Logics, TPHOL'97, Murray Hill,New Jersey, USA, LNCS, Springer Verlag, Août 1997, pp. 63-78.

[13] Roberto Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory. *First International Conference on Foundations of Software Science and Computation Structures, FoSSaCS'98*, M.Nivat Ed.,LNCS 1378, Springer Verlag, Lisbon, Portugal, pp. 48-62.

[14] Solange Coupet-Grimal and Line Jakubiec. Hardware Verification using co-induction in COQ. *International Conference on Theorem Proving in Higher Order Logics, TPHOL'99*, Nice, France, LNCS, Springer Verlag, Septembre 1999, pp. 91-108.

[15] Roberto Amadio, Solange Coupet-Grimal, Silvano Dal Zilio, and Line Jakubiec. A Functional Scenario for Bytecode Verification of Resource Bounds. *the Annual Conference of the European Association for Computer Science Logic, CSL'O4*.

Workshops

[16] Solange Coupet-Grimal. Coq and formal validation of circuits. *Second Workshop EUROFORM*, Ile des Embiez, France, 1994.

[17] Solange Coupet-Grimal and Line Jakubiec. Vérification formelle de circuits avec Coq. In *Journées du GDR Programmation*, Lille, France, 1994.

[18] Solange Coupet-Grimal. Formal verification of an incremental garbage collector. *Journée AS Mobilité*, ENS-Lyon , Mars 2003.

[19] Roberto Amadio, Solange Coupet-Grimal, Silvano Dal Zilio, Line Jakubiec. Short Presentation : A Functional Scenario for Bytecode Verification of Space Bounds. *Space 2004, Second workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*. Venice, Italy.

[20] Roberto Amadio, Solange Coupet-Grimal. Analysis of a Guard Condition in Type Theory. *Théorie des preuves des types inductifs et coinductifs*. ACI Géocal, Marseille, Juin 2004.

Rapports de recherche

[21] Georges Blanc, Noëlle Bleuzen-Guernalec, Solange Coupet-Grimal. Kleene Fonctor and Logic Programming. *Rapport de recherche Laboratoire de Mathématiques de Marseille*, No 89-15, 1989.

[22] Solange Coupet-Grimal and Laurence Pierre. Recursive models for synchronous sequential devices. *Rapport de Recherche Imag-Artemis*, No 855-I, Juillet 1991.

[23] Michel Allemand, Solange Coupet-Grimal, and Jean-Luc Paillet. A functional algebra for circuit modelling and its implementation in LP. *Rapport de recherche LIM*, No 1995.099, Mars 1995.

[24] Michel Allemand, Solange Coupet-Grimal, and Jean-Luc Paillet. FORMATH :a system for modelling and proving circuits. *Rapport de recherche LIM*, No 1995.106, Avril 1995.

[25] Solange Coupet-Grimal and Line Jakubiec. Circuit certification in Type Theory. *Rapport du Laboratoire d'Informatique Fondamentale de Marseille*, RR LIF03-2002.

Contributions Coq

[26] Solange Coupet-Grimal. Equivalence notions on labelled transition systems. *Users' contributions, http ://coq.inria.fr/contribs-eng.html*, 1996.

[27] Solange Coupet-Grimal and Line Jakubiec. Verification and synthesis of hardware linear arithmetic structures. *Users' contributions, http ://coq.inria.fr/contribs-eng.html*, 1999.

[28] Solange Coupet-Grimal and Line Jakubiec. Vérification de circuits synchrones. *http ://cmi.univ-mrs.fr/∼solange/CONTRIB-HARDWARE/FAIRISLE*.

[29] Solange Coupet-Grimal. Linear Temporal Logic. *Users' contributions, http ://coq.inria.fr/contribs-eng.html*, 2002.

[30] Solange Coupet-Grimal and Catherine Nouvet. Formal Verification of an incremental garbage collector. *Users' contributions, http ://coq.inria.fr/contribs-eng.html*, 2002.

7 Articles joints

1. **Référence** [**2**] *Prolog infinite trees and automata.*

2. **Référence** [**7**] *Représentation sémantique dans le traitement des langues naturelles en Prolog.*

3. **Référence** [**3**] *On the use of advanced logic programming features in computational linguistics.*

4. **Référence** [**10**] *A formal system for correct hardware design.*

5. **Référence** [**6**] *Circuit certification in type theory*

6. **Référence** [**13**] *Analysis of a guard condition in type theory.*

7. **Référence** [**4**] *An axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions.*

8. **Référence** [**5**] *Formal verification of an incremental garbage collector.*

9. **Référence** [**15**] *A Functional Scenario for Bytecode Verification of Resource Bounds.*