Rapport de DEA d'Informatique
Université de Nice Sophia-Antipolis

Système de Preuves pour un Langage Purement Fonctionnel

François-René Rideau Đặng-Vũ Bân

8 septembre 1997

Résumé

Nous décrivons un système de preuve de programmes pour un langage fonctionnel pur se rapprochant de Scheme, utilisant un formalisme exprimé en terme de typage, et réalisé en un évaluateur réflexif écrit en Scheme.

Par des preuves simples, nous figurons le style et la puissance d'expression du système, dont nous discutons brièvement et incomplètement les propriétés théoriques.

Table des matières

1 Motivations

1.1 La problématique en perspective

Face à l'augmentation du nombre d'étudiants devant appréhender l'informatique, au développement des concepts et savoir-faire à enseigner, il reste des outils qui malgré les avancées technologiques dans le domaine, tardent à arriver entre d'autres mains que celles des chercheurs les plus pointus: des systèmes formels combinant effectivement programmes et preuves, exécution efficace et analyse expressive, puissance d'expression au niveau de base comme au niveau méta.

Indépendamment de leur attrait théorique, de tels systèmes permettraient dans la pratique une analyse semi-automatique de la sémantique (plutôt que la seule syntaxe) des programmes, pour pouvoir en établir la justesse, ou en corriger les erreurs. Les cas les plus courants et les moins intéressants seraient traités automatiquement par à des heuristiques développées grâce à une expérience qui s'accumulerait lentement mais sûrement; l'utilisateur pourrait alors se concentrer sur les cas non-triviaux, en ayant accès à toutes les ressources d'un système puissant.

En conséquence, enseignants et étudiants seraient libérés du fardeau du rabâchage des bases en classes entières, laissant aux uns et aux autres la possibilité de travailler seul, sans les contraintes d'un groupe, et de s'occuper à d'activités plus élevées à la place. Les applications industrielles pourraient quant à elles bénéficier d'une alliance inédite d'efficacité et de fiabilité, avec un investissement bien plus rentable en ressources humaines, et de grandes facilités pour la validation des programmes.

Dans cette optique à *très* long terme, ce stage se proposait plus modestement de définir les fondements d'un système de preuves formelles à venir pour un sous-ensemble pur du langage Scheme, utilisé par les étudiants de l'Université de Nice come par ceux de nombreuses autres universités de par le monde.

Pour simplifier les choses, nous désignerons dorénavant par foorp le système en cours de réalisation.

1.2 Le projet en situation

La première étape dans ce stage fut de situer foorp par rapport aux nombreux autres qui existent déjà ou sont en cours de développement.

Le système projeté ferait partie de la famille des systèmes de déduction fondés autour d'un langage fonctionnel d'ordre supérieur. La théorie sous jacente à chacun de ces langages est une version ou une autre du λ-calcul [5], premier système complet de calcul inventé (quoiqu'on n'en ai eu la preuve qu'après l'invention des machines de Turing), et un excellent compromis entre réalisabilité pratique et analisabilité théorique.

Cette famille de systèmes peut être assez nettement subdivisée en deux. D'une part se trouvent des systèmes historiquement développés autour de langages de la famille LISP [21], où les termes se réduisent sans contrainte de type; les plus éminents parmi ces systèmes sont le nqthm (sans quanteurs) de Bob Boyer et J Strother Moore [7], et son successeur direct, ACL2 [19], fondé sur un sous-ensemble pur de CommonLISP. D'autre part, les systèmes écrits autour d'un langage de la famille ML, où les termes sont contraints à décrire une algèbre de types, ce sont les successeurs plus ou moins directs de LCF [17]: HOL [15], Isabelle [23], Coq [10], Nuprl [12], etc.

Empressons-nous de remarquer que si, dans ces systèmes, le méta-langage utilisé pour réaliser le système est un parent très proche de celui décrit par le système, ce n'est pas par une quelconque nécessité théorique: n'importe quel langage assez expressif pourrait servir à réaliser n'importe lequel de ces systèmes. Cependant, pour des raisons pratiques d'homogénéité conceptuelle et implémentationnelle du système, c'est le plus souvent le cas; cela facilite d'ailleurs accessoirement un raisonnement réflexif du système sur une partie de lui-même [20].

Le grand avantage des systèmes à la LCF est que la hiérarchie de types peut aussi servir, à travers l'isomorphisme de Curry-De Bruijn-Howard [29], à exprimer les propositions du système, unifiant types et propositions, termes et preuves, typage et vérification. Les types implicites peuvent aussi servir à aider les heuristiques de recherche de preuve, et filtrer les erreurs les plus communes. Cependant, le choix de pouvoir efficacement (et donc calculablement) inférer les types des termes conduit à ne pouvoir exprimer atomiquement toutes les contraintes que l'on pourrait souhaiter à travers le seul type d'un objet. Si la plupart du temps, les contraintes de types sont naturelles, elles peuvent donc devenir un simple surcoût redondant dans certains cas.

Les systèmes à la Boyer-Moore utilisent aussi l'inférence de type, sans que cette inférence serve magiquement de routine de vérification de preuve, mais sans obliger à introduire à tout bout de champ de nouveaux types quand l'injection canonique fait l'affaire, ni rendre redondantes spécifications explicites et implicites. De plus, une autre sorte de "magie" devient facilement accessible, qui consiste à écrire des méta-programmes dans le langage cible et à les interner dans le système de preuve après vérification [8] [20].

Le fait d'avoir choisi comme langage cible un sous-ensemble de Scheme [9], parent minimaliste de la famille LISP, n'est pas innocent: Scheme, langage utilisé auprès des étudiants, ne possède pas encore de tel système formel; le sous-ensemble ciblé, avec des ambitions d'abord limitées à des preuves par induction élémentaires sur des structures récursives de listes, est cependant assez simple pour pouvoir être attaqué au cours d'un stage de DEA. Le sujet est donc circonscrit, sans préjudice de développements ultérieurs qui feraient du système formel un outil effectif pour des étudiants lambda de l'Université.

1.3 Approche choisie

Ce stage étant un travail préliminaire pour un système de preuve à venir, les problèmes d'interface utilisateur n'ont pas du tout été attaqués: d'autres recherches [6] semblent en effet démontrer que des techniques génériques permettent de gérer agréablement l'interaction homme-machine dans un assistant de preuve, de façon essentiellement indépendante du système sous-jacent.

L'efficacité à ce stade expérimental n'a été considéré qu'en ce qu'elle affectait la faisabilité des calculs assez élémentaires effectués.

L'insistance a donc été dans la recherche d'un cadre *sémantique* qui serait le mieux adapté au langage cible et aux développements qui pourraient être effectués ultérieurement, de façon à ce que le résultat potentiel ait une originalité ou un apport qui ne serait pas inutilement redondant avec les systèmes déjà existant et couronnés de succès.

Il a d'abord été brièvement envisagé de créer dans Coq une théorie du sous-ensemble souhaité de Scheme. Mais il est vite apparu qu'il n'était pas faisable alors de gérer agréablement en Coq les symboles, qui sont un type de base dans tous les langages LISP; de plus, la modélisation de Scheme était un cas typique où le système de type de Coq devenait un poids inutile. Dans ce cas encore, utiliser comme méta-langage un langage proche du langage cible semblait un choix naturel. Le choix le plus évident fut le Scheme standard [9].

Restait à déterminer un cadre formel original qui fasse du système projeté plus qu'un parent pauvre de Boyer-Moore. Or, une particularité des prouveurs de Boyer-Moore, est que de même que ses cousins fondés sur une hiérarchie de type à la ML, il n'acceptait de définir une fonction qu'avec une preuve de sa terminaison inéluctable, en trouvant un bon ordre structurel pour lequel les appels récursifs de la fonction sont strictement décroissant. Dans tous ces systèmes, une telle condition était nécessaire à la consistence du reste logique. Une piste d'originalité, était donc de trouver un formalisme qui puisse s'appliquer à des fonctions ne terminant pas. Le système obtenu, qui ne pourrait certes pas rivaliser sur leur terrain avec ses ainés, aurait néanmoins un domaine d'application propre, où il pourrait être respecté.

Or, il se trouve que Scheme est particulièrement bien adapté à ce genre d'interprétation sémantique: son noyau est simplement du λ-calcul avec appel par valeur sans typage, dont on définit habituellement la sémantique par l'observation de terminaison ou non-terminaison dans les contextes bien-formés. On choisit alors la terminaison elle-même comme indicateur de vérité.

Comme dans les systèmes fondés sur des algèbres de types, les connections logiques seront exprimées par des "types" unions et intersections; l'implication sera alors exprimée par un opérateur de comparaison de type: on aura $a \lhd b$ si et seulement si le type de a est inclus dans celui de b[1]. C'est cette dernière opération qui donne au système sa capacité de raisonnement logique d'ordre supérieur, et fait toute l'expressivité (et la semi-indécidabilité) du système.

De façon duale des systèmes précédents, où le type figurait la proposition et un terme l'habitant sa preuve, ici, le terme figurera la proposition, et son appartenance à un type non-vide sera la preuve.

Comme en logique intuitionniste, il n'y aura pas d'indicateur a priori de

1. On aurait pu utiliser $a \subset b$ pour symboliser cette inclusion de types, malheureusement, l'inclusion de types sera aussi isomorphe pour nous à l'implication logique, et la convention opposée $a \supset b$ est déjà utilisé en logique pour indiquer l'implication.

contre-vérité, mais on pourra affirmer terminaison à l'appui la négation d'un terme. La consistence viendra de ce que certaines expressions, indécidable, pourront n'être ni positivement affirmable ni positivement infirmable. Néanmoins toute proposition exprimable dans un des système précédent sera décidable.

L'originalité par rapport à Boyer-Moore est donc que l'évaluation de termes sera une opération beaucoup plus puissante, capable d'exprimer un processus intrinsèquement non-terminant comme le système interactif de preuve lui-même en entier. Cela rendra particulièrement attractif le développement des aspects méta-circulaires de l'évaluateur, qui sont déjà une raison pour laquelle les diverses moutures de Scheme sont si appréciées par rapport aux autres langages fonctionnels.

L'idée est donc de transformer en atout expressif l'absence de typage statique riche qui sinon se contenterait d'obsurcir la tâche des programmeurs et métaprogrammeurs.

2 Description du formalisme

2.1 Syntaxe

Le système foorp manipule un sur-ensemble du noyau applicatif de Scheme. A la base se trouvent les λ-abstractions et variables, permettant de former des expressions à partir d'objets primitifs internes au système: nombres, symboles, fonctions de base (addition, etc).

Expression symbolique	x
Syntaxe utilisateur	*x*
Valeurs prises	valeur de la variable x dans l'environnement courant
Type décrit	type précédemment nommé

Expression symbolique	$f[x_1 \ldots x_n]$
Syntaxe utilisateur	`(f x1 ... xn)`
Valeurs prises	application de fonction à une liste d'arguments
Type décrit	spécialisation de type

Expression symbolique	$\lambda[x_1 \ldots x_n].e$
Syntaxe utilisateur	`(lambda (`*x1* `...` *xn*`) e)`
Valeurs prises	λ-abstraction, forme irréductible
Type décrit	type produit dépendant

Expression symbolique	« x »
Syntaxe utilisateur	`'`*x*
Valeurs prises	citation litérale (forme irréductible)
Type décrit	type singleton

On étend alors le système pour obtenir des types union et intersection, assortis d'éléments neutres pour ces opérations, extension du formalisme déjà utilisé par Robert Givan [14] pour inférer en temps polynomial des propriétés de programmes écrits dans un dialecte de LISP.

Expression symbolique	$x \cup y$
Syntaxe utilisateur	`(either` *x y*`)`
Valeurs prises	choix non-déterministe de valeur
Type décrit	type union

Expression symbolique	$x \cap y$
Syntaxe utilisateur	`(both` *x y*`)`
Valeurs prises	valeurs conjointes de deux expressions
Type décrit	type intersection

Expression symbolique	$\top$
Syntaxe utilisateur	`(top)`
Valeurs prises	toutes les valeurs de l'univers
Type décrit	type top (univers)

Expression symbolique	$\perp$
Syntaxe utilisateur	`(bottom)`
Valeurs prises	aucune valeur
Type décrit	type bottom (vide)

Puis quelques constructeurs de base (facultatifs):

Expression symbolique	(x, y)
Syntaxe utilisateur	`(cons` *x y*`)`
Valeurs prises	paire (x, y)
Type décrit	produit cartésien de types

Expression symbolique	« [] »
Syntaxe utilisateur	`'()`
Valeurs prises	liste vide
Type décrit	type singleton particulier

Et le dernier mais non le moindre:

Expression symbolique	$f \lhd g$
Syntaxe utilisateur	`(<%` f g`)`
Valeurs prises	$\top$ si le type de f est inclus dans celui de g, $\bot$ sinon
Type décrit	type magique de sous-typage

Ainsi armé, on peut faire tous les calculs imaginables. Par exemple, la quantification universelle $\forall[x].P[x]$ peut s'exprimer par $(\lambda[x].\top) \lhd P$.

2.2 Extensions syntaxiques

Pouvoir exprimer en faisant de la macro-expansion manuelle, c'est bien; laisser le système faire la macro-expansion à sa place, c'est mieux, d'autant plus s'il sait faire la macro-expansion inverse lors de l'affichage!

Plutôt que d'utiliser un système de macros de trop bas niveau, on utilise une extension conservative des définitions habituelles: le motif d'une macro est de la forme `(nom sousmotifs...)` où chaque sous-motif peut être soit un nom de variable, soit un lui-même être de la forme `(nom sousmotifs...)`. Le corps de la macro est tel celui d'une autre λ-expression; mais à l'endroit (syntaxique) de l'appel d'une macro, les arguments correspondant à des motifs fonctionnels sont remplacés par une λ-expression dont le corps est l'expression donnée en argument, et les paramètres formels sont ceux donnés dans le motif de la macro.

Voici un exemple simple. Au lieu d'écrire `(lambda () x)`, on pourra écrire `(delay x)`, avec la définition suivante de `delay`:

```
(define (delay (x)) x)
```

De même, le motif de l'argument du corps d'une boucle `for` à la C pourra être de la forme `(body exit continue)`, de façon que l'on puisse donner en paramètre au corps de la boucle les fonctions de continuation d'échappement `exit` et `continue`.

Finalement, pour pouvoir créer de nouvelles formes quantificatrice, il est permis pour une macro de prendre un motif formel en paramètre, ainsi que des formes utilisant ce motif formel. Voir ci-dessous la définition de la quantification universelle.

Les formes dérivées suivantes sont définies:

Expression symbolique	$true[\ldots]$
Motif utilisateur	`(true .` x`)`
Expansion	`(top)`
Signification	converge si et seulement si tous les arguments donnés convergent

Expression symbolique	$false[\ldots]$
Motif utilisateur	`(false .` x`)`
Expansion	`(bottom)`
Signification	ne converge jamais, nonobstant tout effort logiquement consistent

Expression symbolique	$\hat{x}$
Motif utilisateur	`(delay (`x`))`
Expansion	x
Signification	protection contre l'évaluation

Expression symbolique	$\check{x}$
Motif utilisateur	`(`x`)`
Expansion	`(`x`)`
Signification	fin de protection contre l'évaluation

Expression symbolique	$\forall[x_1 \ldots x_n].e$
Motif utilisateur	`(forall` *formals* `(`*body* `,@`*formals*`))`
Expansion	`(<% (lambda` *formals* `(top))` *body*`)`
Signification	quantification universelle

Expression symbolique	$x \lhd_{[x_1 \ldots x_n]} y$
Motif utilisateur	`(<<%` *formals* `(`x `,@`*formals*`) (`y `,@`*formals*`))`
Expansion	`(forall` *formals* `(<% (`x `,@`*formals*`) (`y `,@`*formals*`)))`
Signification	sous-typage universellement quantifié

Expression symbolique	$x = y$
Motif utilisateur	`(=% (`x`) (`y`))`
Expansion	`(both (<% (`x`) (`y`)) (<% (`y`) (`x`)))`
Signification	égalité de types

Expression symbolique	$x =_{[x_1 \dots x_n]} y$
Motif utilisateur	`(==% `*`formals`*` (`*`x`*` ,@`*`formals`*`) (`*`y`*` ,@`*`formals`*`))`
Expansion	`(forall `*`formals`*` (=% (`*`x`*` ,@`*`formals`*`) (`*`y`*` ,@`*`formals`*`)))`
Signification	égalité de types universellement quantifié

Expression symbolique	$\epsilon[x_1 \dots x_n].e$
Motif utilisateur	`(epsilon `*`formals e`*`)`
Expansion	`(apply (lambda `*`formals e`*`) (top))`
Signification	témoin d'existence

Expression symbolique	$\exists[x_1 \dots x_n].e$
Motif utilisateur	`(exists `*`formals e`*`)`
Expansion	`(true (epsilon `*`formals e`*`))`
Signification	affirmation d'existence

Expression symbolique	$\Downarrow_{[x_1 \dots x_n]} e$
Motif utilisateur	`(converges `*`formals e`*`)`
Expansion	`(forall `*`formals`*` . (true (`*`e`*` ,@`*`formals`*`)))`
Signification	convergence universellement quantifiée d'une expression

Expression symbolique	$e \Longrightarrow v$
Motif utilisateur	`(converges-to (`*`e`*`) `*`v`*`)`
Expansion	`(<% v (`*`e`*`))`
Signification	convergence d'une expression vers une valeur

Expression symbolique	$x\vert_{[x_1 \dots x_n]} P$
Motif utilisateur	`(| `*`formals`*` (`*`x`*` ,@`*`formals`*`) (`*`P`*` ,@`*`formals`*`))`
Expansion	`(epsilon `*`formals`*` (both (`*`x`*` ,@`*`formals`*`) (`*`P`*` ,@`*`formals`*`)))`
Signification	contrainte existentielle sur une expression

Expression symbolique	$\neg x$
Motif utilisateur	`(not (x))`
Expansion	`(<% (x) (bottom))`
Signification	affirmation de la non-terminaison de x

Expression symbolique	$\bar{x}$
Motif utilisateur	`(non (x))`
Expansion	`(| (y) (y) (<% (both (x) (y)) (bottom)))`
Signification	valeurs prouvablement non atteintes par une expression

Expression symbolique	Y
Motif utilisateur	`Y`
Expansion	`(lambda (F) ((lambda (x) (x x)) (lambda (x) (F (x x)))))`
Signification	opérateur de point fixe

Expression symbolique	$boolean\%[x]$
Motif utilisateur	`(boolean% (x))`
Expansion	`(<% (x) (<% (top) (x)))`
Signification	booléanité de x

Expression symbolique	x *implements* y
Motif utilisateur	`(implements% (x) (y))`
Expansion	`(both (<% (x) (y)) (<% (true (y)) (true (x))))`
Signification	réalisation partielle booléanité de y par x

Notons qu'en présence de telles macros, les seules formes syntaxiques spéciales réellement nécessaires sont le λ et l'application de fonction, car l'on pourrait exprimer $a \lhd b$ par $\hat{a} \unlhd \hat{b}$ avec une fonction primitive $\unlhd$ qui comparerait les fonctions a et b données par valeur, et convergerait vers toute valeur si on peut prouver que le graphe de a est inclus dans celui de b, et divergerait sinon. De même, intersection et union de types pourrait être exprimées par intersection et union de fonction de graphe donné. Ces primitives pourraient d'ailleurs être étendues ou non des fonctions nullaires vers toute fonction. Cela cadre bien avec le fait que «protéger une expression par un lambda» est une façon naturelle de «représenter» une expression par une valeur, que l'on peut déconstruire par un `epsilon`.

2.3 Types et Cotypes

Il y a plusieurs manières d'appréhender la sémantique des expressions obtenues.

La plus classique, la sémantique dénotationnelle, consiste à décrire un évaluateur associant de façon non forcément univoque des valeurs à chaque expression. On appelle alors *type* d'une expression l'ensemble des valeurs qu'elle peut prendre, ce qui peut être étudié du point de vue de l'interprétation abstraite [11]. l'ensemble des valeurs possibles pour une expression. On pourra figurer l'évaluateur par un système de réécriture [1], les valeurs étant des expressions irréductibles.

La seconde, la sémantique observationnelle, consiste à définir un ensemble de contextes valides qui peuvent opérer sur les expressions, de façon que l'on puisse observer pour toute expression donnée et tout contexte donné si l'expression converge dans ledit contexte. On appelle alors *cotype* d'une expression l'ensemble des contextes valides qui la font converger.

Pour un choix judicieux de l'évaluateur et des contextes d'observation, on a isomorphisme entre types et cotypes.

En effet, dès que l'on peut pour toute valeur v, il y a un contexte qui pour toute expression e observe la terminaison de « v » $\lhd$ e, l'algèbre des cotypes est au moins aussi riche que celle des types.

Réciproquement, si les règles de convergence des contextes suivent fidèlement les règles du système de réécriture décrivant l'évaluation des expressions, et décrivent donc l'ensemble des preuves possibles, alors l'algèbre des cotypes ne peut pas être strictement plus riche que celle des types.

Pour que la sémantique soit semi-décidable, il «suffit» de s'arranger pour que l'ensemble des contextes valides soit récursivement énumérable, et la terminaison d'un terme dans un contexte décidable.

Mais que se passe-t-il si après avoir choisi la sémantique précédente, on considère une sémantique d'observation de second ordre, par laquelle on observe avec pour ensemble de contexte celui des expressions, pour tout contexte Γ et toute expression e la terminaison selon la première sémantique de $\Gamma[e]$? On obtient une algèbre de cotypes d'ordre 2 qui sera strictement plus riche que la précédente: en effet, par les résultats usuels d'incalculabilité, certains objets, disons $\hat{x}$ et $\hat{y}$, seront indiscernables par la première sémantique; cependant, il sera impossible de le prouver, et donc, le contexte $\lambda[\hat{y}].\hat{x}[] \lhd \hat{y}[]$ distinguera $\hat{x}$ de $\hat{y}$! C'est le phénomène connu par lequel x et y ont avoir le même type, tandis que (x, x) et (x, y) ont des types différents [26].

2.4 Plongement Réflexif

foorp ne bénéficie pas de l'isomorphisme de Curry-De Bruijn-Howard entre types et propositions, termes et preuves. Mais à ce même instant, on a vu que les objets de preuves apparaissent naturellement dans la sémantique du système comme contextes valides dans lesquels essayer d'évaluer un terme. L'ensemble des contextes valides peut isomorphiquement être considéré comme celui des traces de la fonction d'évaluation du système. On a donc un isomorphisme entre termes et propositions, entre traces finies de la fonction d'évaluation et preuves.

En implémentant les constructeurs qui énumèrent l'ensemble de ces contextes valides, on obtient donc (selon la façon dont on s'y prend) un vérificateur ou un évaluateur métacirculaire du langage cible de foorp dans un sous-ensemble restreint au λ-calcul applicatif.

On est alors tenté de profiter autant que possible de la réflexivité du système. Une étude à faire serait de comparer le pouvoir d'expression du langage avec ou sans ajout de primitives de réflexivité, d'après les critères définis dans l'étude de Matthias Felleisen [13]. Il se pourrait cependant que l'apport de la réflexivité se trouve dans une mesure d'efficacité théorique comme pratique (en considérant le processus interactif de programmation), et non dans le simple ensemble de fonctions exprimables avec des ressources indéfinies.

2.5 Règles formelles d'évaluation

La règle du modus ponens pour $\lhd$ suffit, assortie d'un ensemble suffisant de schémas d'axiomes:

Règle	$\dfrac{\Gamma \vdash x \lhd y \qquad \Gamma \vdash x}{\Gamma \vdash y}$
Motif reconnu	y
Syntaxe utilisateur	`((MP-<% pf px) x)`
Signification	Modus Ponens pour $\lhd$, où pf est une preuve que $x \lhd y$ et px est une preuve de x

Ici auraient dû être insérées toutes les règles d'évaluations pour les formes de base du calcul.

La sémantique de plus petit point fixe pour Y est assurée par l'axiome suivant:

Règle	$\forall[F,f].(\forall[x].F[f][x]\lhd f[x])\lhd(\forall[x].Y[F][x]\lhd f[x])$
Motif reconnu	...
Syntaxe utilisateur	(...)
Signification	Induction pour Y

2.6 Consistence

Quid de l'argument de Gödel?

Il semble a priori aisé de produire dans ce système une phrase autocontradictoire à la Gödel:

```
(define G (not G))
```

Etant donné une telle instruction, le système va chercher à évaluer $Y(\lambda[G].\neg G)$ pour en assigner le résultat à G. Un tel résultat serait un terme autocontradictoire: si ce terme convergeait, c'est que le système pourrait prouver qu'il diverge.

Mais il n'y a aucune raison que le terme ci-dessus converge: il est tout à fait permis au système de ne jamais trouver de point fixe, et de continuer à tourner indéfiniment, faute de trouver la contradiction demandée par l'utilisateur. C'est la réaction (ou non-réaction) attendue du système, pour peu qu'il soit consistent. En faisant fonctionner le système réflexivement, on pourrait aussi attendre de lui qu'il se rende compte qu'il y a divergence, et qu'en conséquence il s'échappe vers un méta-système inférieur [28].

Similairement, l'instruction

```
(define (G) (not (G)))
```

affectera à G une fonction qui, si on l'évalue, tenterait vainement (espéronsle) de produire une phrase auto-contradictoire.

Tout ceci pour dire qu'un argument à la Gödel ne suffit pas à trouver trivialement une contradiction dans le système de preuve, car le système ne prétend pas à la complétude. Ceci bien sûr ne constitue pas en soi une quelconque preuve de consistence. Une preuve de consistence relative nécessiterait un plongement du système `foorp` dans un autre système auquel on ferait confiance.

3 Utilisation du Système

3.1 Listes

Voici quelques définitions et axiomes engendrant la théorie des listes:

$Axiom\ cons\text{–}def : \forall[a, b].true[(x, y)]$
$Axiom\ cons\text{–}inj : \forall[a, b, x, y].((a, b) \lhd (x, y)) \lhd ((a \lhd x) \cap (b \lhd y))$
$car[x]{::=}a|_{[a]}\exists[d].((a, d) \lhd x)$
$cdr[x]{::=}d|_{[d]}\exists[a].((a, d) \lhd x)$
$pair\%[x]{::=}\exists[a, d].((a, d) \lhd x)$
$Axiom\ nil\text{–}not\text{–}pair : \neg(pair\%\text{« [] »})$
$alist[]{::=}\text{« [] »} \cup (\top, alist[])$

Ici auraient dû être insérées des lemmes et définitions atteignant leur paroxysme dans la preuve de l'associativité de append, voire dans l'isomorphisme entre listes non vide, et arbres binaires modulo rotation des noeuds.

3.2 Vrai et Faux

On pourra utiliser la réflexivité pour passer d'un formalisme logique à l'autre entre celui fondé sur la terminaison et celui fondé sur les valeurs explicites vrai et faux (`#t` et `#f`).

On réservera le suffixe `%` pour le vocabulaire attaché au premier formalisme, fondamental pour le système, et le suffixe ? pour le vocabulaire attaché au second, plus classiquement employé.

Il reste à se souvenir que malgré tout, certaines expressions peuvent ne pas terminer et n'être par conséquent ni vrai ni fausses pour le second formalisme.

Ici auraient dû être insérées des lemmes, définitions, et tactiques et exemples pour étudier les avantages et les inconvénients, la puissance et les limitations du second formalisme

4 Réalisation en un programme

4.1 Choix d'implémentation

Dans l'espoir de pouvoir un jour appliquer les méthodes de preuves pour la vérification de méta-programmes (tactiques) internables par le système [18], voire à l'interpréteur lui-même [24], le système réalisé est écrit en Scheme, dans un style aussi fonctionnel (pur) que possible, sans s'aliéner la possibilité d'utiliser des effets de bords, notamment pour l'amorçage du système.

Pour des raisons de portabilité, et dans l'espoir de profiter pleinement des compilateurs Scheme existant et à venir le système se restreint autant que possible au langage défini par la norme R^4RS [9], avec des extensions quasi-universellement répandues: `define-macro` à la `defmacro` de CommonLISP, `format` à la CommonLISP.

La contrepartie de cette portabilité est que le programme ne peut s'appuyer que sur un plus grand dénominateur commun des fonctionnalités offertes par les différentes implantations du langage Scheme.

4.2 Fonctionnalités réalisées ou non

Un parseur a été écrit pour les expressions. Un évaluateur a été réalisé pour le sous-ensemble purement calculatoire qui constitue l'intersection de Scheme avec le langage cible du prouveur (i.e. le λ-calcul applicatif), avec possibilité d'interner des primitives Scheme.

La stratégie de réduction, avec appel strict et réécriture complète des textes substitués lors des β-réduction, fonctionne correctement, mais de façon catastrophiquement lente, car elle effectue avidement les substitutions requises.

L'évaluation de termes arbitraires, étant intrinsèquement parallèle et réflexive, n'a pas été réalisée encore.

Seules les tactiques de vérification de preuve les plus triviales ont été réalisées. Le vérificateur est encore incomplet.

Le développement de métaprogrammes d'analyse sémantique et de typage approximatif, de tactiques de résolution partielle, ou d'heuristiques de recherche de preuve n'a pas commencé.

Le peu qui est fait et ce qui s'y ajoutera par la suite sera librement disponible sur Internet sous License Publique Générale GNU depuis la page de l'auteur, ou par voie de courrier électronique.

4.3 Développements possibles à long-terme

Les choix d'implémentation contribuent tous à ralentir le système. Un investissement humain et temporel pourrait remédier à cette lenteur, mais les ressources requises faisaient défaut au cours du stage. Pour cette raison, seul un vérificateur a été réalisé.

Un évaluateur demandant virtuellement la vérification en parallèle de toutes les preuves possibles, une réalisation assez efficace pour être utilisable demanderait un ordonnanceur de tâches soigné, capable, par diverses heuristiques, de sélectionner les tactiques les plus susceptibles d'aboutir, et d'éliminer celles n'ayant aucune chance.

L'amélioration progressive du système se ferait par l'écriture de nouvelles tactiques ou heuristiques, que l'on pourrait réflexivement certifier dans le système avant de les interner [8, 20, 18, 24]. Pour exécuter efficacement les tactiques réflexivement certifiées correctes par le système, un évaluateur partiel voire un compilateur devrait être ajouter.

Un système vraiment performant pouvant rivaliser en vitesse avec les langages de programmation classique nécessiterait l'abandon du Scheme standard comme langage d'implémentation, pour éviter les *inversions d'abstractions* [3] comme l'émulation des environnements de première classe, du parallélisme [22], le ramasse-miette manuel, le "dispatch" manuel selon le type, etc. S'il est parfaitement possible de programmer élégamment tout ce qui précède en Scheme standard, c'est souvent au prix d'efforts démesurés eu égard à la complète inefficacité du résultat.

En fait, l'exposition au système de raisonnement de tous ces mécanismes (et la possibilité de réinterner des versions modifiées) permettrait le raisonnement sur ces mécanismes, leur modification et "optimisation" sémantiquement conservative, et réduiraient la compilation à de l'évaluation partielle.

De tels développements mèneraient à un système réflexif. Mais l'amorçage d'un tel système est un processus difficile [25], qui n'a pas pu être réalisé durant le stage.

5 Remerciements

Je tiens tout particulièrement à remercier ...

mes enseignants Michel Rueher, Jacques Chazarain, Bruno Martin, mes tantes Chantal et Linh Chi pour leur gentillesse et leur infinie patience.

```
 == Faré -=- (FR) François-René Rideau -=- (VN) Đặng-Vũ Bân -=- rideau@ens.fr ==
Join a project for a free reflective computing system! | 6 rue Augustin Thierry
TUNES is a Useful, Not Expedient System.               | 75019 PARIS     FRANCE
http://www.eleves.ens.fr:8080/home/rideau/Tunes/ -=- Reflection&Cybernethics ==
```


Références

[1] ??? *Rewriting Solving Proving.* ???, February 1997.

[2] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Program.* MIT-Press, 2nd edition, 1997.

[3] Henry G. Baker. Critique of DIN Kernel Lisp Definition Version 1.2. *Lisp and Symbolic Computation*, 4, 4:371–398, March 1992.

[4] Henry G. Baker and Carl Hewitt. The Incremental Garbage Collection of Processes. In *SIGPLAN Notices*, volume 12, 8, pages 52–59. ACM, August 1977.

[5] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1985. (more than you care to know about the λ-calculus).

[6] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by Pointing. Technical report, INRIA Sophia Antipolis, 199x.

[7] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

[8] Robert S. Boyer and J Strother Moore. Metafunctions: Proving them correct and using them efficiently as new Proof Procedures. In *The correctness problem in Computer Science*, pages 103–184. Academic Press, 1981.

[9] William Clinger and Editors Jonathan Rees. *Revised^4 Report on the Algorithmic Language Scheme*. ACM, July-September 1991.

[10] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual*, 1996.

[11] Patrick Cousot. Types as abstract interpretations. In *Proceedings of the 1997 Conference on Principles Of Programming Languages*, pages 316–331. ACM, 1997.

[12] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, October 1995.

[13] Matthias Felleisen. On the expressive power of programming languages. Technical report, Rice University, 1991.

[14] Robert Givan. Inferring Program in Polynomial Time. In *SAS96*, 1996.

[15] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, A theorem proving environment for higher order logic*. Press Syndicate of the University of Cambridge, 1993.

[16] Michael J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 199x.

[17] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Lecture Notes in Computer Science. Springer, 1978.

[18] Douglas J. Howe. Computational Meta-Theory in Nuprl. Technical report, Cornell University, March 1988.

[19] Matt Kaufmann and J Strother Moore. *ACL2 version 1.9*. Computational Logic, Inc, May 1997.

[20] Todd B. Knoblock and R. Constable. Formalized meta-reasoning in type theory. Technical report, Cornell University, March 1986.

[21] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*. North-Holland, 1967.

[22] Robin Milner. The Polyadic Pi-Calculus: a Tutorial. Technical report, University of Edinburgh, October 1991.

[23] Laurence C. Paulson. *Introduction to Isabelle, A theorem proving environment for higher order logic*. Press Syndicate of the University of Cambridge, 1996.

[24] Dino Petreshi and Salvatore Ruggieri. Verification of meta-interpreters. *Journal of Logic and Computation*, 1996.

[25] Jacques Pitrat. *Métaconnaissance, futur de l'intelligence artificielle*. Hermès, 1990.

[26] Bruno Poizat. *Cours de Théorie des Modèles*. Nur al-Mantiq Wal-Ma'rifah, 1985.

[27] Tim Sheard and Neal Nelson. Type safe abstractions using program generators. Technical report, OGI, 1995.

[28] Brian Cantwell Smith. Reflection and Semantics in LISP. In *Proceedings of the 1983 Conference on Principles Of Programming Languages*. ACM, 1983.

[29] Benjamin Werner. *Théories des Types*, February 1997.