Preuve formelle d'un récupérateur de mémoire pour cartes à puce *

Solange Coupet-Grimal †
Université de Provence, Marseille

10 juillet 2003

Résumé

Cet article présente un travail de recherche en informatique, mené dans le cadre d'une collaboration entre l'Université de Provence, l'INRIA et la société Gemplus. Il illustre l'interaction entre mathématique et informatique. Des résultats récents de la recherche en logique y sont utilisés concrètement dans une application industrielle. Plus précisément, nous montrons comment spécifier et démontrer formellement la correction d'un récupérateur de mémoire embarqué sur cartes à puce. Nous montrons également comment vérifier automatiquement la preuve de correction avec l'assistant de preuve Coq, dont nous présentons brièvement les fondements logiques.

Mots Clés Méthodes formelles, spécification, vérification, théorie des types, logiques temporelles, récupération de mémoire.

1 Introduction

On propose, sur le thème *mathématiques et autres disciplines* de parler d'un travail de recherche en Informatique, mené dans le cadre d'une collaboration régionale entre l'Université de Provence, la société Gemplus et l'INRIA. Il illustre de façon significative l'interaction entre mathématiques et informatique. Des résultats récents de la recherche en logique y sont utilisés concrètement dans des applications industrielles.

Avec l'omniprésence de l'informatique dans les réalisations technologiques modernes il est devenu crucial de garantir la fiabilité des systèmes *critiques* ; il s'agit de logiciels et de matériel dont les défaillances pourraient provoquer de

*Travail en partie financé par l'action COLOR *Clé* entre INRIA Sophia Antipolis et l'Université de Provence.

†Laboratoire d'Informatique Fondamentale de Marseille, CMI, 39 rue Joliot-Curie, F-13453, Marseille, France. Solange.Coupet@cmi.univ-mrs.fr.

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. 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.

On propose d'illustrer cette démarche par un problème concret, lié à la technologie des cartes à puce. Les *SmartCards* sont des cartes à puce multi-applications. Divers programmes, chargés sur ces cartes après leur émission, peuvent s'y exécuter en parallèle. C'est ainsi que, lors d'un achat, une application peut débiter un compte bancaire tandis qu'une autre crédite d'un certain nombre de points un compte fidélité. Ces applications sont écrites en JavaCard, un sous-ensemble du langage Java spécialement dédié aux cartes à puce. Ce langage permet par ailleurs l'allocation dynamique de mémoire : un programme peut, selon les besoins de son exécution, demander davantage de mémoire que ce qui était prévu au départ. Généralement ce genre de dispositif est accompagné d'un programme chargé de détecter et de récupérer les cellules mémoire qui, après avoir été dynamiquement allouées et utilisées, redeviennent inutiles. Il s'agit d'un problème particulièrement sensible dans le domaine des cartes à puce qui ne possèdent que peu d'espace mémoire, ce qui freine considérablement le développement d'applications. C'est ainsi qu'on a été amené à étudier un algorithme de récupération de mémoire et, ce qui était indispensable dans ce domaine hautement critique, à le vérifier formellement.

Dans la section 2, on décrit brièvement l'algorithme et on montre comment le spécifier formellement. Dans la section 3, on présente une logique permettant de raisonner formellement sur des propriétés temporelles et on ébauche, dans ses grandes lignes, la preuve de correction. Enfin, en section 4, on explique comment ces preuves formelles peuvent être automatiquement vérifiées dans le système Coq, avant de conclure en section 5.

2 Un algorithme de récupération de mémoire

Il s'agit donc d'implanter sur la carte un programme qui détecte et libère les cellules mémoire affectées à une application en cours d'exécution et abandonnées après utilisation. Le récupérateur de mémoire tourne en parallèle avec l'application : de temps en temps, l'application *laisse la main* au récupérateur qui en profite pour exécuter quelques pas de programme avant de rappeler à son tour l'application... Cet *entrelacement* des actions des deux composants de système est indéterministe et se poursuit indéfiniment. En voici, informellement, son fonctionnement.

2.1 Description informelle

Voici comment agit sur la mémoire de la carte, le système composé de l'application de l'utilisateur et du récupérateur de mémoire.

La mémoire La partie de la mémoire occupée par le programme est organisée sous forme d'un graphe orienté appelé *tas*. On y distingue un nœud particulier, la *racine*, qui symbolise la mémoire statique, occupée de façon permanente et allouée une fois pour toute en début d'exécution. Les autres nœuds du tas représentent les cellules qui ont été allouées dynamiquement au cours de l'exécution du programme. Il y a un arc entre deux nœuds n et m, si n fait référence au contenu de m. Les cellules inoccupées sont dites *libres* : elles ne font pas partie du tas et elles sont disponibles pour des demandes d'allocation dynamique ultérieures.

Le programme de l'utilisateur agit sur le *tas* (il est pour cela appelé le *mutateur* dans le jargon des informaticiens). Il le modifie :

1. soit en ajoutant un arc entre deux nœuds accessibles à partir de la racine,
2. soit en supprimant un arc entre deux nœuds accessibles à partir de la racine (ce faisant, un nœud peut devenir inaccessible ; c'est alors une cellule poubelle, toujours occupée mais inutile puisque le programme ne peut plus y accéder),
3. soit, en allouant dynamiquement une cellule libre, qui est alors rajoutée au *tas* comme fils de la racine et perd ainsi son statut de cellule libre.

Le récupérateur de mémoire On l'appelle aussi le **GC**, comme *glaneur de cellules* et aussi *garbage collector*. Son but est de détecter les cellules poubelles et de les libérer. Pour cela il parcourt le graphe à partir de la racine en *marquant* les nœuds sur lesquels il passe avec 3 couleurs : blanc, gris et noir de la façon suivante. *Initialement, tous les nœuds du tas sont blancs, sauf la racine qui est grise.* Le récupérateur peut effectuer les actions suivantes :

4. S'il y a des cellules grises, on est en phase de *marquage*. L'action consiste à noircir une cellule grise et griser ses fils blancs.

5. S'il n'y a plus de cellules grises, mais qu'il y a des cellules blanches, on est en phase de *balayage*. L'action consiste à libérer une cellule blanche.
6. S'il n'y a plus ni cellules blanches ni cellules grises, un cycle du GC vient de se terminer. L'action consiste à réinitialiser les couleurs du *tas*, en grisant la racine et blanchissant toutes les autres.

En fait, durant la phase de marquage, une cellule est noire si elle a été visitée, ainsi que tous ses fils. Elle est grise si elle a été visitée, mais ses fils pas nécessairement, elle est blanche si elle n'a pas été visitée. Par suite, la phase de marquage s'achève quand il ne reste plus de nœuds gris dans le graphe. A ce moment là, les cellules accessibles sont noires, les cellules blanches sont les cellules poubelles. Commence alors la phase de balayage qui libère ces dernières. On peut se convaincre, et c'est un invariant du programme, qu'*il n'y a jamais d'arc d'une cellule noire vers une cellule blanche.*

Mise en parallèle du GC et du mutateur On conçoit que les temps de réponse de la carte à puce doivent être très courts. De plus, il faut que la carte continue à fonctionner correctement en cas d'arrachage intempestif. On ne peut envisager de stopper le programme de l'utilisateur le temps d'exécuter un cycle complet de récupération de mémoire. On rend donc la récupération de mémoire *incrémentale* en permettant un *entrelacement* indéterministe des actions du mutateur et de celles du GC. C'est ce qui fait toute la subtilité et la difficulté du problème. Les actions du mutateur doivent être légèrement modifiées. Ainsi, dans le cas du rajout d'un arc d'un nœud n vers un nœud m (action (1)), si n est noir et m est blanc, le mutateur doit changer la couleur de m et le rendre gris pour respecter l'invariant. De même, en cas d'allocation de mémoire, le nœud rajouté à la racine est coloré en noir.

Correction du système Le système doit impérativement satisfaire une propriété dite de *sûreté*. Elle s'énonce comme suit :

Une cellule libérée est toujours inaccessible de la racine

Il est clair que la libération de cellules encore utilisées par le programme, provoquerait un comportement erratique. La propriété de sûreté peut être trivialement satisfaite si le GC ne libère jamais aucune cellule. Bien sûr, ce n'est pas le but poursuivi. Il est donc intéressant d'établir également la propriété, dite de *vivacité*, suivante :

Toute cellule inaccessible sera libérée au bout d'un temps fini.

Toutefois, l'entrelacement des actions du *mutateur* et du GC étant indéterministe, on peut imaginer par exemple que le GC ne soit jamais appelé, auquel cas la propriété de vivacité ne peut évidemment pas être satisfaite. En fait, une propriété de vivacité se prouve le plus souvent sous une hypothèse dite d'*équité* : dans ce problème, elle stipule que le GC doit être appelé *infiniment souvent* (c'est à

dire qu'il n'existe pas d'instant à partir duquel il est oublié à tout jamais). En fait, à la fin du cycle de marquage, un nœud inaccessible peut être blanc ou, s'il est devenu inaccessible après avoir été marqué, noir. Dans le premier cas, il est libéré durant la phase de balayage qui suit. Dans le second cas, il devient blanc quand le marquage est ré-initialisé à la fin du cycle, et dans ce cas, il est libéré au prochain cycle. Il est donc essentiel que les cycles de récupération de mémoire se succèdent indéfiniment pour assurer la libération de toute cellule devenue inaccessible.

2.2 Spécification formelle

Ce système GC-mutateur, tel qu'il vient dêtre présenté, doit être maintenant décrit formellement dans un langage mathématique. Voici la démarche adoptée. On introduit divers objets et notions comme suit.

• *Cellule* est un ensemble quelconque représentant les cellules mémoires.

• *racine* est un élément particulier de *Cellules* désignant la racine du tas.

• $Couleurs = \{Noir,\ Gris,\ Blanc,\ Libre\}$ désigne l'ensemble des quatre couleurs possibles pour une cellule.

• Un *marquage* est alors une fonction $m :\ Cellules \rightarrow Couleurs$.

• $Contrôle = \{Marquage,\ Balayage,\ Mutateur\}$ est un ensemble de trois états de contrôle indiquant la phase du processus en cours d'exécution.

• Un tas est une relation binaire t sur les cellules. Pour toutes cellules c_1 et c_2, $t(c_1,\ c_2) = vrai$ si et seulement s'il y a un arc de c_1 vers c_2.

• L'état courant du système est alors un triplet σ=*(t, m, ctl)* où t est le tas courant, m est son marquage, *ctl* est un élément de l'ensemble *Contrôle*.

• L'*état initial* est l'état dans lequel le contrôle est *Mutateur*, le tas est réduit au nœud *racine* qui est grise. Toutes les autres cellules sont libres.

• On définit pour chaque action possible du système (par exemple *supprimer un arc*, ou bien *libérer une cellule blanche*), une relation binaire sur les états appelée *transition*. La relation de transition associée à une action a est notée $\xrightarrow{a}$. Par exemple, si a désigne l'action *supprimer un arc*, la relation $\xrightarrow{a}$ est définie par :

$\sigma_1 \xrightarrow{a} \sigma_2$ *si et seulement si* $\sigma_1 = (t,\ m,\ Mutateur)$, $\sigma_2 = (t',\ m,\ Mutateur)$ *et les deux tas t et t' sont partout identiques sauf sur un couple de cellules* $(c_1,\ c_2)$ *pour lequel* $t(c_1, c_2) = vrai$ *et* $t'(c_1,\ c_2) = faux$.

• Une exécution du système est alors une suite infinie d'états

$$(\sigma_0,\ \sigma_1, \ldots,\ \sigma_i,\ \sigma_{i+1}, \ldots)$$

telle que σ_0 est l'état initial et pour tout i, l'état σ_{i+1} se déduit de σ_i par l'une des transitions du système.

• le prédicat être *accessible* peut alors être défini de façon inductive comme suit :

- *la racine est* accessible,
- *tout nœud dont un père dans le graphe est* accessible *est lui-même* accessible.

Le système étant ainsi décrit, il reste à exprimer par des formules les propriétés à prouver. En particulier, il faut spécifier formellement le fait que toute exécution *équitable* du système satisfait les conditions de *sécurité* et de *vivacité*. Il est clair que toutes ces propriétés sont à forte connotation temporelle. Il faut être capable d'exprimer formellement par exemple la locution *infiniment souvent*. Il existe pour cela diverses logiques dites temporelles et nous présentons l'une d'entre elles au paragraphe suivant.

3 La Logique Linéaire Temporelle

Les logiques temporelles proposent des outils pour formaliser des raisonnements impliquant des notions liées au temps. Il s'agit d'opérateurs temporels s'appliquant à des prédicats définis sur des exécutions infinies de programmes. Une recherche active dans ce domaine a fait émerger ces dernières années plusieurs formalismes logiques, dont les mérites comparés ont été l'objet de vives discussions. Nous présentons ci-après l'un d'entre eux, suffisamment expressif pour l'étude de cas qui nous intéresse, et proche de l'intuition. Il s'agit de la Logique Temporelle Linéaire (LTL) [7].

3.1 La logique

Voici les principaux opérateurs temporels de LTL. Dans ce qui suit, P, Q et R désignent des prédicats sur les suites infinies d'états.

L'opérateur *prochain* exprime qu'une propriété P va être satisfaite à l'instant suivant.

• **L'opérateur** *prochain* **:** $\bigcirc$

$\bigcirc$P$(\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots)$:= P$(\sigma_1, \ldots, \sigma_n, \ldots)$.

L'opérateur *toujours* exprime qu'une propriété P est satisfaite par tous les suffixes d'une exécution, c'est-à-dire à chaque instant.

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

$\Box$P($\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$) := $\forall$i$\in \mathbb{N}$ P($\sigma_i, \ldots, \sigma_n, \ldots$)

L'opérateur *finalement* exprime qu'elle sera satisfaite au bout d'un temps fini.

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

$\Diamond$P($\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$) := $\exists$i$\in \mathbb{N}$ P($\sigma_i, \ldots, \sigma_n, \ldots$)

L'opérateur *jusqu'à ce que* exprime que P est satisfaite jusqu'à ce que, au bout d'un temps fini, Q soit satisfaite.

- **L'opérateur** *jusqu'a ce que* **:** $\mathcal{U}$

P$\mathcal{U}$Q($\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$) :=
$\exists$i$\in \mathbb{N}$ ($\forall$j$\in$ {0, ...,i-1} P($\sigma_j, \ldots, \sigma_n, \ldots$))$\wedge$Q($\sigma_i, \ldots, \sigma_n, \ldots$)

L'opérateur *A moins que* est plus faible, puisqu'il n'exige pas que Q soit finalement satisfaite.

- **L'opérateur** *A moins que* **:** $\mathcal{W}$

P$\mathcal{W}$Q($\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$) :=
($\exists$i$\in \mathbb{N}$ ($\forall$j$\in$ {0, ...,i-1} P($\sigma_j, \ldots, \sigma_n, \ldots$))$\wedge$ Q($\sigma_i, \ldots, \sigma_n, \ldots$)) $\vee$
$\forall$i$\in \mathbb{N}$ P($\sigma_i, \ldots, \sigma_n, \ldots$).

De ces opérateurs de base, on peut en dériver d'autres. Ainsi, on peut se convaincre facilement que σ *satisfait infiniment souvent* P s'exprime par

$$\Box \diamond P(\sigma).$$

On établit alors des règles, qui sont autant d'outils pour le raisonnement. Par exemple l'idempotence de l'opérateur *toujours* s'exprime comme suit :

$$\forall P,\ \forall \sigma,\ \Box P(\sigma) \rightarrow \Box\Box P(\sigma)$$

Dans l'étude de cas qui nous intéresse, les propriétés de sûreté, de vivacité et d'équité s'expriment par des formules de cette logique. Par souci de simplicité, nous ne rentrerons pas dans les détails techniques ici, car les formules obtenues sont assez complexes.

3.2 Preuve de correction

3.2.1 Sûreté

Pour la propriété de sûreté, on considère le prédicat P défini sur les suites infinies d'états σ par :

Dans le premier état de σ, si un nœud est accessible, alors il n'est pas libre

Il faut donc montrer que pour toute exécution σ du système, la formule $\Box P(\sigma)$ est satisfaite. On prouve pour cela que la condition *si un nœud est accessible, alors il n'est pas libre* est vraie sur l'état initial, et qu'elle est conservée par chaque transition du système *GC-Mutateur*. Il faut pour cela établir un invariant plus fort. Il s'agit de la conjonction des cinq propriétés suivantes :

1. La racine est grise ou noire
2. Si le contrôle est *balayage*, alors il n'y a pas de nœud gris
3. Il n'y a pas d'arc d'un nœud noir vers un nœud blanc
4. Si un nœud est accessible, alors il n'est pas libre
5. S'il n'y a pas de nœuds gris, alors tout nœud accessible est noir

3.2.2 Vivacité

La preuve de vivacité est plus compliquée et nous ne faisons que l'esquisser. L'important est d'en comprendre l'esprit.

Une mesure sur les états On définit une mesure sur les états comme la somme du nombre de nœuds gris et du nombre de nœuds blancs. On démontre que cette mesure décroît strictement à chaque action de marquage et de balayage. La mesure s'annule à la fin de chaque cycle de récupération de mémoire. On démontre un résultat dit de terminaison :

Pour toute exécution équitable, la mesure s'annule infiniment souvent

Rappelons que dans cette étude cas, *équitable* signifie que le GC est appelé infiniment souvent. Le résultat de terminaison établit que chaque cycle de récupération de mémoire se termine au bout d'un temps fini et qu'il est suivi par un autre cycle. Il est à noter que l'énoncé ci-dessus fait fortement appel aux opérateurs temporels. Il est prouvé par induction.

Principales étapes de la preuve de vivacité Considérons un nœud n non accessible :

- soit il est libéré *au bout d'un temps fini* (CQFD)
- soit, *au bout d'un temps fini* un état est atteint dans lequel n est resté non libre et non accessible et dans lequel il n'y a plus de nœuds gris (fin de la phase de marquage)

A ce moment-là, n ne peut être que blanc ou noir.

- s'il est blanc, il sera libéré *au bout d'un temps fini* (au cours de la phase de balayage en cours d'exécution)
- s'il est noir, il va rester noir et non accessible *jusqu'à ce que* il n'y ait plus de nœuds blancs (fin de la phase de balayage et du cycle courant)

Dans ce dernier cas, après ré-initialisation du marquage, il devient blanc et tous ses ancêtres sont libres ou blancs. Ceci persiste *jusqu'à* la fin de la phase de marquage. Par suite, n sera libéré au cours de la phase de balayage qui suit, donc *au bout d'un temps fini*.

La preuve formelle tient compte bien sûr de tous les détails. Les étapes évoquées ci-dessus sont autant de formules de LTL à établir. Il y en a exactement neuf, de la forme :

$$A(n,\ \sigma) \Rightarrow A(n,\ \sigma)\mathcal{U}B(n,\ \sigma)$$

dans lesquelles n désigne un nœud et σ une exécution. On prouve que ces formules sont toujours *toujours* vraies, sous des hypothèses d'équité convenables. Toutes les preuves sont faites par induction et reposent sur le fait que la mesure s'annule infiniment souvent.

On peut prendre conscience à cette brève présentation, que la démonstration n'est pas très simple. Il est facile, dans une telle preuve *papier-crayon* (dans le jargon des méthodes formelles), d'oublier des étapes, de faire de petites erreurs d'inattention qui peuvent s'avérer fatales. Aussi, après avoir prouvé le programme, est-il bon de *prouver la preuve*. Il s'agit du même type de démarche, si ce n'est que l'objet de l'étude n'est plus le programme, mais sa preuve elle-même. Mais au fait, quelle différence entre une preuve et un programme ? Aucune, nous disent les logiciens.

4 Vérifier les preuves

Il ne peut être question dans le cadre de cet article de faire un exposé rigoureux sur la très fameuse correspondance de Curry-Howard. L'histoire se passe dans un système formel, appelé λ-calcul, destiné à exprimer les algorithmes sous forme fonctionnelle.

4.1 Lambda-calcul et programmation fonctionnelle

Dans un tel système, on distingue des types et des termes. Les types peuvent être interprétés par des ensembles et les termes par des fonctions.

Les types sont, soit des types de base $A,\ B\ \ldots$, soit des types de la forme $A \to B$, pouvant être interprétés par l'ensemble des fonctions de A dans B.

Les termes sont

– soit des variables $x,\ y, \ldots,$

– soit une expression de la forme $(f\ a)$ où f et a sont des termes et qui s'interprète comme *l'application* de la fonction f à un argument a,

– soit une expression de la forme $\lambda x.t$ où x est une variable et t un terme. Un tel terme est une *abstraction* et correspond à la notation fonctionnelle $x \mapsto t$ désignant la fonction qui à la variable x associe t.

Typage des termes. A chaque terme t est associé un type A (on écrit $t : A$). Les variables sont typées de façon arbitraire à l'aide d'assignation de types de la forme $H = \{x_1 : t_1, \ldots, x_n : t_n\}$ où les x_i sont des variables et les t_i sont des types. Les autres termes sont typés à l'aide des règles suivantes, écrites sous forme de séquents (les hypothèses sont en haut et la conclusion en bas). Elles tombent sous le sens : la première n'exprime rien d'autre que le fait qu'on obtient un élément dans un ensemble B en appliquant une fonction $f : A \to B$ à un élément d'un ensemble A :

$$\frac{H \vdash f : A \to B, H \vdash a : A}{H \vdash (fa) : B}. \tag{1}$$

$$\frac{H, x : A \vdash t : B}{H \vdash \lambda x.t : A \to B}. \tag{2}$$

Il est à noter qu'il n'y a pas de différence de nature entre les fonctions et les arguments de fonctions : une fonction peut-être elle-même l'argument ou le résultat d'une fonction d'ordre supérieur (comme la dérivation par exemple).

Les règles de réduction La plus importante est la β-conversion. Elle consiste à remplacer un *radical*, c'est-à-dire un terme de la forme $((\lambda x.t)\ a)$, par le terme noté $t[x \leftarrow a]$, obtenu en remplaçant dans t les occurrences libres de la variable x par l'argument a (ceci doit être accompagné d'éventuels renommages de variables liées pour éviter d'éventuels conflits de noms).

Le λ-calcul ainsi succintement présenté, est le paradigme de la programmation fonctionnelle. Il s'agit d'un système de même expressivité que les *machines de Turing*. Un terme représente ainsi un programme, un algorithme, une fonction *effective* (trois termes synonymes dans la suite). Exécuter un programme c'est réduire le terme du λ-calcul correspondant par β-conversions successives, jusqu'à ce qu'il ne contienne plus de radicaux. C'est ce que fait l'interpréteur du langage *Caml*, pour ne citer que celui-là.

4.2 La correspondance de Curry-Howard

Le λ-calcul a été introduit par Church [1] pour être interprété comme on vient de le présenter. Mais Curry et Howard [6] en ont donné une autre interprétation, parfaitement symétrique, dont un des aspects est l'analogie entre la règle du *modus ponens* et le fait qu'on obtient un élément dans un ensemble B

en appliquant une fonction $f : A \to B$ à un élément de A. Dans l'interprétation de Curry-Howard, un **type** est une **proposition** logique, et un **terme** de type A est une **preuve** de la propositon A. Le constructeur de type $\to$ est alors interprété par l'implication logique.

Les preuves considérées ici sont des dérivations dans un système de déduction naturelle. Dans un tel système, si H est un ensemble de formules et A est une formule, on exprime par un jugement de la forme $H \vdash A$ le fait que A se déduit des hypothèses H. Les connecteurs logiques ont une interprétation opérationnelle donnée par des règles d'introduction et d'élimination. Pour l'*implication*, la règle d'élimination correspond au principe du *modus ponens*.

$$\frac{H \vdash A \to B, H \vdash A}{H \vdash B}. \tag{3}$$

La régle d'introduction est la suivante :

$$\frac{H, A \vdash B}{H \vdash A \to B}. \tag{4}$$

Dans la conclusion de cette règle, A a disparu de l'ensemble des hypothèses, on dit qu'elle a été déchargée. La symétrie entre les règles 1 et 3 d'une part et les régles 2 et 4 d'autre part est flagrante. Le fait que l'hypothèse A soit déchargée dans le règle 4 correspond, dans la conclusion de la règle 2, à la liaison de la variable x par l'opérateur λ.

Cette correspondance qui ne se limite pas uniquement à l'implication. On enrichit le système de types du λ-calcul en introduisant autant de constructeurs que nécessaire : la conjonction correspond à un produit cartésien, un prédicat sur un ensemble A n'est rien d'autre qu'une famille d'ensembles indexée par $A \ldots$ On obtient ainsi de puissantes logiques d'ordre supérieur et des langages de programmation très expressifs. Une preuve étant ainsi codée comme un λ-terme, il n'y a plus aucune distinction entre les preuves et programmes. De plus :

Vérifier une preuve, c'est vérifier un type

Plus précisément, vérifier que la preuve d'une proposition A est correcte revient à montrer que son type est A. Ceci peut être établi automatiquement par un logiciel s'appuyant sur des règles de typage comme 1 et 2.

4.3 Vérification avec l'assistant de preuves Coq

Ces travaux ont donné naissance à l'assistant d'aide à la preuve Coq [8]. Il s'agit d'un logiciel réalisé à l'INRIA et fondé sur un λ-Calcul d'ordre supérieur, le Calcul des Constructions [2]. Son système de types est très riche et permet notamment la définition de types dépendants de termes, de types inductifs pour exprimer des plus petits points fixes et des types co-inductifs pour les plus grands

points fixes.

Nous avons implanté la Logique Temporelle Linéaire dans ce système [4, 3], c'est-à-dire que nous avons décrit les opérateurs temporels comme des types du λ-calcul et nous avons prouvé un certain nombre de théorèmes correspondant aux règles de LTL. Nous avons également implanté une axiomatisation des ensembles finis, que nous avons utilisée pour compter le nombre de cellules blanches, grises, noires ... Ceci nous a permis de spécifier dans Coq le système *GC-Mutateur* et ses propriétés, et d'en vérifier la preuve de correction [5]. Pour donner quelques chiffres, l'étude de cas qui nous intéresse comprend 600 lemmes, une centaine de définitions, 10.000 lignes de code. Elle a demandé 6 mois de travail pour 1,5 chercheurs.

Il faut bien prendre conscience que dans ce type d'approche, les moindres détails d'une preuve, même les plus évidents, doivent être explicités. Dénombrer les éléments d'un ensemble requiert au préalable une théorie des ensembles finis. Les résultats arithmétiques les plus triviaux doivent avoir été établis. Heureusement, il existe de nombreuses bibliothèques Coq et les utilisateurs mettent sur le site internet [9] leurs contributions, permettant ainsi au plus grand nombre d'en bénéficier. Donc, tout n'est pas à construire à partir de rien. Ce type de travail nécessite une rigueur absolue mais assure une fiabilité maximale aux systèmes ainsi vérifiés. En effet, la validité de cette approche repose sur la bonne conception du vérificateur de types Coq. Or il s'agit d'un programme relativement court, basé sur un petit nombre de règles. Il est donc facile de se convaincre de sa correction.

Les industriels sont désormais fortement demandeurs dans le domaine des méthodes formelles et contribuent à dynamiser la recherche par de nombreux contrats passés avec les laboratoires.

5 Conclusion

Cette étude cas illustre de façon significative les retombées pratiques de la recherche dans un des domaines les plus abstraits des mathématiques : celui de la logique. Historiquement, cela n'a rien d'étonnant. C'est le problème de la "mécanisation" du raisonnement mathématique, formulé par Hilbert, qui est à l'origine de cette extraordinaire révolution technologique qu'est la naissance de l'informatique. Elle est 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.

Références

[1] Alonzo Church. *The Calculi of λ-Conversion*. Princeton University Press, 1941.

[2] Thierry Coquand and Gérard Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. *EUROCAL 85, Linz Springer-Verlag LNCS 203*, 1985.

[3] Solange Coupet-Grimal. An Axiomatization of Linear Temporal Logic. *The Coq Users's Contributions*, July 2002. http ://coq.inria.fr/contribs-eng.htlm.

[4] Solange Coupet-Grimal. An Axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions. *The Journal of Logic and Computation*, 2002. A paraître.

[5] Solange Coupet-Grimal and Catherine Nouvet. Verification of an Incremental Garbage Collector in Type Theory. *The Journal of Logic and Computation*, 2002.

[6] William A. Howard. The Formulae-as-types Notion of Construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, London, 1980. Academic Press.

[7] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[8] The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V7.1. Technical report, LogiCal Project-INRIA, 2001.

[9] The Coq Proof Assistant. http ://coq.inria.fr.