Typage du bytecode Caml

Sébastien Ailleret

Inria Rocquencourt - Projet Moscova
sebastien.ailleret@inria.fr

Résumé

Typer le bytecode Caml permet d'assurer à un utilisateur que le code compilé qu'il possède s'exécutera sans erreur, et ceci sans aucune connaissance du code source du programme ni de la façon dont il a été compilé.

Dans les langages de programmation fortement typés, la sûreté de l'exécution d'un programme est garantie par les propriétés de typage du langage source. Toutes les vérifications étant entièrement statiques, les informations de type sont logiquement effacées à la compilation car inutiles ensuite. En revanche, en présence de code mobile, il est impératif de garantir la correction du programme lors de sa réception, car on ne dispose plus du code source dont la validité assurait la sûreté. Pour ce faire, notre approche consiste à ajouter au bytecode quelques informations de type, qui, sans rien dévoiler du code source du programme, permettent de vérifier rapidement qu'un code compilé possède les mêmes garanties de sûreté que le langage source.

Ce travail présente un système de type pour le bytecode Caml, des résultats de sûreté pour le typage considéré ainsi qu'une implémentation pour un sous-ensemble de Caml comprenant les références et les types concrets paramétrés, et qui permet une vérification rapide du code.

Introduction

OCaml [5] est un langage fonctionnel fortement typé, ce qui signifie que les programmes bien typés sont garantis de s'exécuter sans erreur de typage. Ces erreurs sont importantes, car leur absence implique en particulier l'absence d'erreur de segmentation. Comme le typage est entièrement statique, les types ne sont pas conservés dans le bytecode, car une vérification lors de l'exécution est inutile. Cette approche se révèle très efficace dans le cadre de programmes compilés par la personne qui les exécute, mais est peu appropriée à la programmation distribuée. En effet, lorsque l'on reçoit des programmes, téléchargés sur l'Internet par exemple, on désire, avant de les exécuter, avoir des garanties sur la sûreté de leur exécution. C'est ce que fournit Java et qui a fait son succès en tant que langage de programmation du web. Le bytecode OCaml, par contre, ne garantit aucune propriété. En effet, en forçant le compilateur à ne pas vérifier le typage, ou en modifiant directement le bytecode, il est possible d'avoir une erreur de segmentation lors de l'exécution de bytecode OCaml. Dans l'état actuel, il n'est donc pas possible d'assurer la sûreté de l'exécution de code mobile Caml. Deux techniques ont été envisagées pour résoudre ce problème. La première consiste à faire migrer le code source, mais cette technique n'est pas satisfaisante, parce que de nombreuses personnes ne tiennent pas à diffuser le code source de leurs programmes. De plus, la mobilité est alors compliquée par le fait que par le fait qu'elle doit s'accompagner de multiples compilations, ce qui perd du temps et empêche de nombreuses optimisations. L'autre approche [12] consiste à utiliser un compilateur qui signe les programmes qu'il compile, assurant ainsi qu'ils s'exécuteront sans erreur, car produits par un compilateur sûr. Cette technique a le double inconvénient d'imposer l'utilisation d'un unique compilateur et de faire reposer la sécurité sur l'honnêteté et l'inviolabilité de l'instance centralisée qui produit le compilateur.

Dans cet article, nous décrivons un système de type pour le bytecode Caml qui permet de vérifier la sûreté d'un programme sans connaître son origine. L'idée de base consiste à ne pas effacer toutes les informations de type connues lors de la compilation, mais d'en garder suffisamment dans le bytecode pour assurer sa validité. Un vérificateur s'assure ensuite que les informations de type présentes dans le bytecode sont bien conformes au code, ce qui prouve la sûreté de l'exécution. Ce mécanisme ne repose que sur la correction du vérificateur : il ne nécessite aucune confiance en un tiers, ni même dans les informations de type fournies avec le bytecode qui sont toutes vérifiées. De plus, le code source du vérificateur est disponible, ce qui permet de le vérifier et de le compiler soi-même pour une sécurité maximale. Cette idée de garantir la sûreté d'un programme compilé en l'accompagnant d'une preuve a été tout d'abord présentée par George Necula et Peter Lee, dans leur article *Proof Carrying Code* [10]. Greg Morrisett, David Walker, Karl Crary et Neal Glew ont ensuite simplifié cette idée en remplaçant la preuve par une information de typage, dans leur article *From System F to Typed Assembly Language* [9].

Détecter les erreurs de typage est très important en présence de code mobile. Tout d'abord, la sûreté du typage assure une certaine cohérence du programme : les entiers et les pointeurs ne sont pas mélangés, et on évite une trop grande anarchie dans l'arithmétique des pointeurs. Ce point est important, car il assure que le programme s'exécutera jusqu'à son terme sans erreur de segmentation. De plus, la sûreté du typage est l'un des prérequis essentiel pour la majorité des politiques de sécurité, car elle interdit au programme d'essayer de modifier le code qui gère ses permissions [7].

L'originalité de notre approche est de se baser sur le bytecode OCaml, sans modifier le jeu d'instructions, en se contentant d'y rajouter quelques informations de type. Le bytecode généré peut donc être exécuté par l'interprète standard d'OCaml (`ocamlrun`). De plus, la taille des informations de type ajoutées au bytecode est très faible, grâce à un système de vérification qui permet d'inférer la majorité des types.

Dans la section 1, nous présentons la machine virtuelle utilisée. La section 2 décrit le système de type utilisé pour typer le bytecode. La section 3 présente rapidement Taml, une implémentation de compilateur et de vérificateur pour le système décrit dans les deux premières sections. Enfin, la section 4 présente les travaux apparentés et les développements en cours.

1. Présentation de la machine virtuelle et du bytecode

Cette première section présente la structure du cœur de la machine virtuelle ainsi que les instructions qui permettent de la manipuler. Cette machine virtuelle est celle d'OCaml. En effet, le bytecode que nous générons utilise le jeu d'instructions de la machine virtuelle d'OCaml.

1.1. La machine virtuelle

La machine virtuelle que nous présentons est une abstraction de celle d'OCaml, qui est une évolution de la machine Zinc [6]. Le sous-ensemble présenté ici permet de traiter Caml avec références. Utiliser une abstraction au lieu de la véritable machine virtuelle permet de simplifier la présentation, mais n'élude aucune difficulté de typage. Les omissions concernent essentiellement des optimisations pour la rapidité d'exécution du bytecode.

On dispose donc d'une machine à pile qui comporte les éléments suivants :

- Un accumulateur ac : c'est l'unique registre de calcul de la machine virtuelle. Il permet d'éviter trop de manipulations de la pile.
- Une pile de calcul cs : elle est composée de deux parties : une partie locale loc qui contient une liste de valeurs utilisées par la fonction en cours d'exécution, et une autre partie up utilisée pour les retours de fonction qui contient soit rien (si l'on est dans la fonction principale), soit les valeurs de pc, env et cs qui seront utilisées pour le prochain retour de fonction.
- Un environnement env : c'est une liste de valeurs qui correspond aux variables libres de la fonction

en cours d'exécution.

- Un pointeur de code pc : il pointe vers la prochaine instruction à exécuter.
- Un tas h : dans notre description, il servira uniquement à stocker les valeurs contenues dans les références. Il est représenté par une fonction qui associe des valeurs aux références.

Les différentes valeurs que peut contenir la machine virtuelle sont :

- *void* : la valeur par défaut (`()` en Caml).
- Une référence : pointeur vers une valeur dans le tas.
- Une fermeture : elle contient une fonction (une étiquette l qui indique le début du code), un environnement env, une liste $args$ des arguments déjà fournis (en cas d'application partielle) et le nombre r d'arguments encore attendus. Une fermeture est notée $clos(l,env,args,r)$.

On voit que ces valeurs sont très similaires à celles de Caml. La machine virtuelle les utilise de la façon suivante : un `let` est interprété en poussant la valeur calculée sur la pile. Si l'on désire ensuite réutiliser cette valeur, on va la rechercher dans la pile. La pile est aussi utilisée pour les appels de fonction. On place tous les arguments sur la pile, la fonction dans l'accumulateur et on appelle l'instruction `APPLY`. Un mécanisme permet alors de sauver la position du pointeur de code et l'environnement que l'on devra retrouver au retour de la fonction, puis de leur affecter les valeurs qui correspondent à la fonction que l'on applique. Lorsque l'on rencontre l'instruction `RETURN`, on récupère les valeurs sauvées et on continue l'exécution avec le résultat de l'application dans l'accumulateur.

1.2. Le bytecode

Pour agir sur la machine virtuelle, on dispose d'instructions de bytecode. Celles-ci peuvent être séparées en 2 classes : celles qui effectuent des opérations sur les données, et celles qui gèrent les appels de fonction. Les sous-sections suivantes contiennent la liste des instructions disponibles, ainsi que leur action sur la machine virtuelle.

Le jeu d'instructions de la machine virtuelle est présenté sous forme de règles de réduction. Celles-ci sont présentées en indiquant à gauche la valeur de l'instruction à la position pc dans le code, et à droite la modification de la machine virtuelle lors de l'exécution de l'instruction. La notation utilisée pour désigner une machine virtuelle est ac,cs,env,pc,h, où cs est de la forme (loc,up). Le champ pc de la machine virtuelle est omis lorsque sa valeur est uniquement incrémentée de 1, les autres champs sont omis lorsqu'ils sont invariants. On utilise les notations usuelles de Caml pour représenter les listes.

Actions sur la pile

Les instructions suivantes ont une sémantique très simple : elles recopient ou effacent des valeurs de la machine virtuelle. `ACC n` recopie une valeur de la pile dans l'accumulateur et `ENVACC n` fait de même avec une valeur de l'environnement. `PUSH` permet d'ajouter au sommet de la pile la valeur présente dans l'accumulateur, et `POP` supprime la valeur au sommet de la pile. `CONST0` met dans l'accumulateur la valeur *void*.

Les 3 dernières instructions montrent comment le tas est utilisé pour gérer les références.

`ACC n`	$ac,([v_0;\ldots;v_n]@loc,up) \longrightarrow v_n,([v_0;\ldots;v_n]@loc,up)$
`ENVACC n`	$ac,[v_0;\ldots;v_n]@env \longrightarrow v_n,[v_0;\ldots;v_n]@env$
`PUSH`	$ac,(loc,up) \longrightarrow ac,(ac :: loc,up)$
`POP`	$ac,(x :: loc,up) \longrightarrow ac,(loc,up)$
`CONST0`	$ac \longrightarrow void$
`MAKEREF`	$ac,h \longrightarrow l,h \cup \{l : ac\}$ où l est une référence fraîche
`GETREF`	$l,h \cup \{l : v\} \longrightarrow v,h \cup \{l : v\}$
`SETREF`	$l,(x :: loc,up),h \cup \{l : v\} \longrightarrow void,(loc,up),h \cup \{l : x\}$

Appels de fonctions

Cette section présente les instructions utilisées pour gérer les appels de fonction. La convention est de passer les arguments sur la pile et de récupérer le résultat d'une application dans l'accumulateur.

`CLOSURE l n r` crée une fermeture dont le code est pointé par `l`, qui prend sur la pile et dans l'accumulateur la valeur de ses `n` variables libres, et qui attend ensuite `r` arguments. `APPLY n` applique la fonction qui est dans l'accumulateur à `n` arguments placés sur la pile. Enfin, `RETURN` indique la fin du corps d'une fonction et retourne vers la fonction appelante ; l'adresse de retour, l'environnement et la pile sont récupérés dans la partie de la pile prévue à cet effet.

Les appels de fonction sont compliqués par l'aspect fonctionnel du langage. En effet, si une fonction `f` a un type `int` $\rightarrow$ `int` $\rightarrow$ `int`, il se peut que cette fonction prenne un entier et renvoie une fonction. Dans ce cas, `f 2 3` doit appliquer 2 à `f`, puis 3 au résultat précédent. Malheureusement, au moment de la compilation, on ne connaît que le type de la fonction, et pas le nombre d'arguments qu'elle attend réellement. La solution consiste donc à passer à la fonction les 2 arguments dans tous les cas, et lors de l'exécution, de traiter un `RETURN` comme un `APPLY` si la fonction n'a pas consommé tous les arguments. Pour savoir si une fonction a traité tous les arguments qui lui ont été passés, il suffit de regarder si la partie locale de la pile est vide. En effet, une fonction doit effacer elle-même tous ses arguments avant de retourner son résultat. Une pile non vide indique donc une sur-application. Un cas typique d'application de cette règle correspond à l'évaluation de `snd (2, fun x -> x) 3`.

`CLOSURE l n r`	$v_1,([v_2;\ldots;v_n]@loc,up) \longrightarrow clos(l,[v_1;\ldots;v_n],[],r),(loc,up)$
`APPLY n` $(si\ n < r)$	$clos(l,cenv,cargs,r),([v_1;\ldots;v_n]@loc,up)$ $\longrightarrow clos(l,cenv,cargs@[v_1;\ldots;v_n],r-n),(loc,up)$
`APPLY n` $(si\ n \geq r)$	$clos(l,cenv,cargs,r),([v_1;\ldots;v_n]@loc,up),env,pc$ $\longrightarrow void,(cargs@[v_1;\ldots;v_n],(pc+1,env,(loc,up))),cenv,l$
`RETURN`	$ac,([],(ra,senv,cs)),env,pc \longrightarrow ac,cs,senv,ra$
`RETURN` $(si\ ea \geq r)$	$clos(l,cenv,cargs,r),([v_1;\ldots;v_{ea}],up),env,pc$ $\longrightarrow void,(cargs@[v_1;\ldots;v_{ea}],up),cenv,l$
`RETURN` $(si\ 0 < ea < r)$	$clos(l,cenv,cargs,r),([v_1;\ldots;v_{ea}],(ra,senv,cs)),env,pc$ $\longrightarrow clos(l,cenv,cargs@[v_1;\ldots;v_{ea}],r-ea),cs,senv,ra$

Voici un petit programme très simple :

```
let f x = x in
let g x = f x in
g ()
```

Sa traduction est en trois parties : l'une définit la fonction principale (à partir du label `main`), une autre la fonction `f`, et la dernière la fonction `g`.

```
Label main:                 Label f:
    CLOSURE f 0 1               ACC0
    PUSH                        POP
    CLOSURE g 1 1               RETURN
    PUSH                    Label g:
    CONST 0                     ACC0
    PUSH ACC1                   PUSH
    APPLY 1                     ENVACC1
    POP                         APPLY 1
    POP                         POP
    RETURN                      RETURN
```

Remarque On dispose aussi de l'instruction de marquage `Label`. Elle permet de définir un point d'entrée de fonction, et n'a aucun effet sur la machine virtuelle. Cette instruction est accompagnée d'informations sur le type de la fonction (nombre d'arguments attendus, type des variables libres et type de la fonction). Ce sont ces informations qui distinguent le bytecode TAML du bytecode OCAML.

2. Typage

La vérification du bytecode est effectuée par une semi-inférence de type. La difficulté, que nous décrivons dans la section 2.6, vient du fait que dans le bytecode, rien ne différencie les entiers des booléens ou des types concrets. Il faut donc rajouter au bytecode quelques informations de type pour rendre possible la vérification. Nous avons choisi de nous limiter aux types des fonctions, que l'on accole à l'étiquette qui indique le début du code de la fonction. On donne en fait le type de la fonction proprement dite, le nombre d'arguments qu'elle attend ainsi que le type de ses variables libres. Dans l'exemple précédent, on aurait donc les définitions suivantes :

- `Label f (1) : 'a -> 'a []`, 1 indiquant le nombre d'arguments attendus par la fonction.
- `Label g (1) : 'a -> 'a ['b -> 'b]`, le type de la variable libre f est indiqué entre crochets.

Cette technique possède de plus l'énorme avantage de permettre de faire la vérification du bytecode en une unique passe linéaire sans avoir à faire de recherche de point fixe. En effet, toutes les boucles sont codées par des appels récursifs de fonctions, ce qui permet de disposer du point fixe (le type de la fonction) sans effectuer de coûteux calculs.

Les règles de typage que l'on utilise modélisent le comportement de chaque instruction sur la machine virtuelle, en établissant une relation entre les types présents avant et après l'exécution de l'instruction. Certaines règles de typage peuvent rappeler des règles de typage de Caml, avec plusieurs différences notables toutefois. Tout d'abord, Le traitement de la généralisation ne peut pas être fait comme en Caml, car le bytecode ne contient aucune information qui permette de savoir quand généraliser (comme le fait le `in` en Caml). L'option choisie ici consiste donc à généraliser à tout instant. De plus, on ne dispose pas d'information sur l'expansivité des expressions traitées, qui indique si des références peuvent être créées lors de l'évaluation d'une expression. On traite donc le problème des références polymorphes de la façon suivante : au lieu de choisir si l'on généralise ou non une expression sur la base d'un critère syntaxique, on marque toutes les variables de type à ne pas généraliser, c'est-à-dire celles qui risquent de faire partie du type d'une référence. On peut ensuite généraliser toutes les autres variables.

2.1. Définition des types

Les types utilisés pour typer les valeurs de la machine virtuelle sont assez proches de ceux utilisés par Caml. Cependant, on utilise une notation unifiée pour définir les types et les schémas de type, ce qui simplifie les règles de sous-typage (qui expriment en particulier l'instanciation et la généralisation).

Vérifier le bytecode fonction par fonction permet une simplification intéressante : Comme on ne passe jamais de frontière de fonction, toutes les variables de types proviennent soit d'un des arguments de la fonction, soit d'une définition précédente dans le corps de la fonction. Si on note les premières $\underline{\alpha}$, on sait, grâce à des conditions dans les règles de typage de l'application et de la création de référence, que toutes les autres sont généralisables. Il est donc inutile de noter les schémas de type avec un symbole $\forall$, car on connaît implicitement le nom de toutes les variables généralisées. Les types utilisés ont donc la définition suivante :

$t ::= unit$	
α	variable généralisée
$\underline{\alpha}$	variable non généralisée (aussi appelée constante)
$t\ ref$	référence
$t \rightarrow t$	fonction

On dispose des fonctions fv et cv, qui, à tout type, associent respectivement la liste de ses variables généralisées et celles de ses variables non généralisées.

On dispose aussi d'une fonction *constinst*, qui, à tout type, en associe un autre dans lequel les variables de type généralisées ont été substituées par des variables non généralisées fraîches.

Remarque Les variables non généralisées ci-dessus ne correspondent pas à celles que l'on rencontre en Caml, dans la définition `let f = ref (fun x -> x)` par exemple. Ces dernières sont en fait inutiles, car en instanciant toutes ces variables par $unit$, on a toujours un type correct pour ce programme.

Sous-typage

On dispose d'une règle de sous-typage qui correspond à la fois à l'instanciation d'un schéma et à la généralisation d'un type.

Définition 1 (Règle de sous-typage) $t \leq t[\alpha_1 \leftarrow t_1; \ldots; \alpha_n \leftarrow t_n]$

Cette relation de sous-typage est une relation réflexive et transitive.

Propriété 2 *La relation $\leq$ de sous-typage vérifie les propriétés suivantes :*

- $t \leq t$
- *si* $t_1 \leq t_2$ *et* $t_2 \leq t_3$*, alors* $t_1 \leq t_3$

Démonstration : La première propriété est évidente, il suffit d'utiliser la substitution identité.

Pour la deuxième proposition, on note $t_2 = t_1 s$ et $t_3 = t_2 s'$. On a $s = [\alpha_i \rightarrow \tau_i]$, et on peut supposer que cette substitution réaffecte toute les variables de t_1 (au besoin, on complète s par l'identité). On peut alors démontrer par récurrence sur la structure de t_1 que $t_3 = t_1[\alpha_i \rightarrow \tau_i s']$, ce qui est le résultat attendu. $\square$

2.2. Type de la partie locale de la machine virtuelle

La partie locale de la machine virtuelle correspond à celle qui peut être accédée pendant l'exécution d'une fonction. Ceci correspond à une machine virtuelle dans laquelle la pile est réduite à sa partie locale. Son type est donc défini par la donnée des éléments suivants :

- le type de l'accumulateur.
- le type des valeurs de la partie locale de la pile.

– le type des valeurs de l'environnement.
– le type attendu pour la valeur de retour de la fonction courante.

Un tel type est donc noté $T = t,[t_0;\ldots;t_n],[\tau_0;\ldots;\tau_m],t_r$, où t, les t_i, les τ_i et t_r sont des types de valeurs.

On peut facilement définir une notion de sous-typage pour les types de machines virtuelles à partir de la notion de sous-typage définie à la section 2.1 :

$$\begin{gathered}\tau,[\tau_0;\ldots;\tau_n],[t_0;\ldots;t_m],\tau_r \leq \tau',[\tau'_0;\ldots;\tau'_n],[t'_0;\ldots;t'_m],\tau'_r\\ \Longleftrightarrow \tau \leq \tau' \ et\ \tau_i \leq \tau'_i \ et\ t_i = t'_i \ et\ \tau_r = \tau'_r\end{gathered}$$

2.3. Type d'un programme

Un programme est bien typé si l'on peut associer à chacune de ses instructions un type local de machine virtuelle (qui indique les valeurs attendues dans la machine virtuelle avant l'exécution de l'instruction), et si, pour chaque instruction, on respecte une relation entre son entrée et sa sortie.

Les règles de typage utilisent les notations suivantes :

– P est le programme que l'on cherche à typer. C'est un tableau dont les éléments sont les instructions qui le composent, notées $P(i)$.
– V est le type du programme. C'est un tableau dont les éléments, $V(i)$, sont des types locaux de machine virtuelle.

La première règle de typage est celle qui rassemble les relations pour chaque instruction du programme et donne les conditions initiales. Les règles suivantes modélisent l'effet sur le typage de chaque instruction de bytecode en donnant une relation entre l'état de la machine virtuelle avant et après l'exécution de l'instruction. Dans toutes ses règles, le fait d'avoir $V(i+1) \geq T$ signifie qu'après chaque instruction, on peut généraliser ou instancier tous les types présents dans la machine virtuelle.

$$\frac{P(0) = \texttt{Label 1}\ []\ \texttt{unit} \rightarrow \tau \qquad \forall i.P,V,i \vdash P(i)}{P : V}$$

Actions sur la pile

La première des règles ci-dessous concerne les points d'entrée des fonctions. Elle indique la façon dont une fonction reçoit ses arguments sur la pile, et son environnement dans la partie de la machine virtuelle prévue à cet effet.

La règle suivante indique qu'un `ACC n` prend un élément dans la pile et le recopie dans l'accumulateur. Il en est donc de même pour les types. Les règles suivantes sont relativement similaires.

La règle concernant `MAKEREF` est assortie d'une condition qui assure que la référence ne cache pas de polymorhisme, en vérifiant qu'elle ne contient pas de variable généralisée.

$$\frac{V(i) = V(i+1) \geq unit,[\tau_1;\ldots;\tau_m],[t_1;\ldots;t_n],\tau_r \qquad constinst(\tau) = \tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \tau_r}{P,V,i \vdash \texttt{Label m } [\texttt{t}_1;\ldots;\texttt{t}_\texttt{n}]\ \tau}$$

$$\frac{V(i) = tac,[\tau_0;\ldots;\tau_n]@tcs,tenv \qquad V(i+1) \geq \tau_n,[\tau_0;\ldots;\tau_n]@tcs,tenv}{P,V,i \vdash \texttt{ACC n}}$$

$$\frac{V(i) = tac,tcs,[\tau_0;\ldots;\tau_n]@tenv \qquad V(i+1) \geq \tau_n,tcs,[\tau_0;\ldots;\tau_n]@tenv}{P,V,i \vdash \texttt{ENVACC n}}$$

$$\frac{V(i) = \tau,tcs,tenv \qquad V(i+1) \geq \tau,\tau :: tcs,tenv}{P,V,i \vdash \texttt{PUSH}} \qquad \frac{V(i) = \tau,t :: tcs,tenv \qquad V(i+1) \geq \tau,tcs,tenv}{P,V,i \vdash \texttt{POP}}$$

$$\frac{V(i) = tac,tcs,tenv \qquad V(i+1) \geq unit,tcs,tenv}{P,V,i \vdash \texttt{CONST0}}$$

$$\frac{V(i) = \tau,tcs,tenv \qquad V(i+1) \geq \tau\ ref,tcs,tenv \qquad fv(\tau) = \emptyset}{P,V,i \vdash \texttt{MAKEREF}}$$

$$\frac{V(i) = \tau\ ref,tcs,env \qquad V(i+1) \geq \tau,tcs,env}{P,V,i \vdash \texttt{GETREF}}$$

$$\frac{V(i) = \tau\ ref,\tau :: tcs,tenv \qquad V(i+1) \geq unit,tcs,tenv}{P,V,i \vdash \texttt{SETREF}}$$

Appels de fonctions

Les règles suivantes traitent de la création et de l'utilisation des fonctions. La première concerne la création d'une fermeture.

La deuxième règle concerne l'application. On ne s'intéresse ici qu'au résultat de l'application sans se soucier des étapes entre l'appel de fonction et le retour du résultat, qu'on ne peut de toute façon pas connaître au moment du typage. Tout comme `MAKEREF`, cette expression a pu créer des références, on vérifie donc que son type ne contient pas de variable libre.

La dernière règle indique les conventions pour les retours de fonction. Tous les arguments fournis sur la pile et traités doivent avoir été effacés et le résultat est placé dans l'accumulateur.

$$\frac{V(i) = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau,[\tau_1;\ldots;\tau_n]@tcs,tenv \qquad V(i+1) \geq \tau,tcs,tenv \qquad fv(\tau) = \emptyset}{P,V,i \vdash \texttt{APPLY n}}$$

$$\frac{\begin{array}{c} P(l) = \texttt{Label r } [\texttt{t}_1;\ldots;\texttt{t}_\texttt{n}]\ \tau \qquad cv(t_i) = \emptyset \\ V(i) = t_1,[t_2;\ldots;t_n]@tcs,tenv \qquad V(i+1) \geq \tau,tcs,tenv \end{array}}{P,V,i \vdash \texttt{CLOSURE l n r}} \qquad \frac{V(i) = \tau_r,[],tenv,\tau_r}{P,V,i \vdash \texttt{RETURN}}$$

2.4. Type d'une valeur

Les règles de typage de la section précédente n'ont de sens que si l'on est capable de donner un type aux valeurs qui se trouvent dans la machine virtuelle. Expliciter ce typage nécessite un peu de gymnastique pour être capable de typer des valeurs récursives à travers le tas.

Étant donnée une fonction H, qui à des locations associe des types sans variable (généralisée ou non), on définit la relation $P{,}H \vdash v : t$ de la façon suivante :

$$P{,}H \vdash void : unit \qquad\qquad P{,}H \vdash l : H(l)\ ref \qquad\qquad \frac{P{,}H \vdash v : \tau \qquad \tau \leq \tau'}{P{,}H \vdash v : \tau'}$$

$$\frac{P(l) = Label\ r\ T_{env}\ \tau_f \qquad \tau_f \leq \tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \tau \qquad P{,}H \vdash env : T_{env} \qquad P{,}H \vdash x_i : \tau_i}{P{,}H \vdash clos(l{,}env{,}[x_1; \ldots ; x_k]{,}r-k) : \tau}$$

On dit ensuite qu'une fonction H est une abstraction d'un tas h, noté $P \vdash h : H$, si pour toutes les locations l définies dans h, on a $P{,}H \vdash h(l) : H(l)$. Ceci implique en particulier que le domaine de h est inclus dans celui de H.

Ceci nous permet de définir la notion de typage réellement intéressante, notée $P{,}H{,}h \vdash v : \tau$, par :

$$P{,}H{,}h \vdash v : \tau \quad \Longleftrightarrow \quad P \vdash h : H \ \ et \ \ P{,}H \vdash v : \tau$$

Dans la suite, on utilise la notation $H \sqsupset H'$ pour indiquer que H et H' coïncident sur le domaine de H', ce qui signifie que H est un enrichissement de H'.

2.5. Type d'un état de la machine virtuelle

Un état de machine virtuelle est bien typé si il correspond à une étape de l'évaluation d'un programme correct. Les informations nécessaires pour s'assurer de cette propriété sont les instances de toutes les fonctions en cours d'évaluation (c'est-à-dire la fonction courante et toutes celles présentes dans la pile des retours). Cette connaissance permet de vérifier que tous les types présents dans la machine virtuelle sont en accord avec le type du programme, c'est-à-dire que toutes les valeurs retournées seront du type souhaité par la fonction appelante.

Une telle relation de typage est notée $P : V \models ac{,}(loc{,}up){,}env{,}pc{,}h : H{,}\Sigma$.

On utilise les notations suivantes, qui agissent sur les types locaux de machine virtuelle :

- $arg(tac{,}tcs{,}tenv{,}tr) = tac{,}tcs{,}tenv$
- $ret(tac{,}tcs{,}tenv{,}tr) = tr$
- Σ est une liste de substitutions de l'ensemble des constantes de type vers les types sans constante ni variable de type.

Formellement, la relation de typage est définie de la façon suivante :

$$return(P{,}V{,}loc{,}(){,}pc{,}H{,}\sigma{,}[])$$

$$\frac{ret(V(pc)\sigma) = \tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \tau \\ (H' \sqsupset H \quad P{,}H'{,}h \vdash loc : [\tau_1; \ldots ; \tau_k] \quad P{,}H'{,}h \vdash ac : \tau) \Rightarrow P : V \models ac{,}cs{,}env{,}pc'{,}h : H'{,}\Sigma}{return(P{,}V{,}loc{,}(pc'{,}env{,}cs){,}pc{,}H{,}\sigma{,}\Sigma)}$$

$$\frac{P{,}H{,}h \vdash ac{,}loc_1{,}env : arg(V(pc)\sigma) \qquad return(P{,}V{,}loc_2{,}up{,}pc{,}H{,}\sigma{,}\Sigma)}{P : V \models ac{,}(loc_1 @ loc_2{,}up){,}env{,}pc{,}h : H{,}(\sigma :: \Sigma)}$$

Le jugement *return* traduit le fait que le retour de fonction doit bien se passer, c'est-à-dire que si la fonction courante retourne une valeur en accord avec son type, la fonction appelante peut reprendre son exécution de façon correcte, et ainsi de suite.

Propriétés

L'intérêt du typage est de déterminer statiquement qu'un programme bien typé s'exécutera sans erreur. Ceci signifie qu'un état bien typé de la machine virtuelle se réduit vers un autre état bien typé de la machine virtuelle, et que la réduction peut toujours continuer, du moins jusqu'au `RETURN` final.

Théoreme 3 (Préservation du typage par réduction) *Si il existe* H *et* Σ *tel que* $P : V \models mv : H,\Sigma$ *et que* $mv \longrightarrow mv'$*, alors il existe* H' *et* Σ' *tel que* $P : V \models mv' : H',\Sigma'$.

Démonstration : La démonstration de ce théorème se fait par cas sur la règle de réduction utilisée.

Si la règle de réduction correspond à une action sur la pile, il est relativement aisé de voir que le nouvel état est toujours bien typé. En effet, seule la partie locale est affectée.

La règle de création de fermeture et l'application partielle ne modifient elles aussi que la partie locale de la machine virtuelle, ce qui simplifie sensiblement la preuve.

Les cas délicats sont ceux de l'application et du retour de fonction. En effet, il faut gérer l'empilement dans la pile des retours de l'état actuel et trouver la nouvelle substitution à appliquer pour le nouvel état. Cette nouvelle substitution s'obtient en composant la substitution due au sous-typage et celle de la fermeture courante et en prouvant que la composition est bien une substitution. Un élément important est de remarquer que la substitution σ est sans effet sur l'environnement, car par construction, il ne peut contenir de constante de type. □

Théoreme 4 (Progression) *Soit* $mv = ac,(loc,up),env,pc,h$ *et* K *tel que* $P : V \models mv : K$*. Alors, si on n'a pas* $P(pc) =$ `RETURN` *et* $(loc,up) = ([],[])$*, il existe* mv' *tel que* $mv \longrightarrow mv'$.

Démonstration : La démonstration de ce théorème se fait par cas sur l'instruction à exécuter ($P(pc)$). □

2.6. Autres types de données

Pour rendre le langage utilisable, il est essentiel de disposer d'autres types de base que *unit*. Les sous-sections suivantes présentent la façon dont les entiers, les booléens et les types concrets sont implémentés dans la machine virtuelle.

Les entiers

L'instruction de bytecode qui introduit une constante entière dans un programme est `CONST i`. C'est une généralisation de l'instruction `CONST0`, qui permet de mettre *val i* dans l'accumulateur. *val i* est une valeur de la machine virtuelle qui correspond à un entier. *void* est en fait la valeur *val* 0. Il convient donc de rajouter une règle de typage pour `CONST i` qui reflète le fait que la valeur ajoutée dans la machine virtuelle par cette instruction peut être un entier.

$$\frac{V(i) = tac,tcs,tenv \qquad V(i+1) \geq int,tcs,tenv}{P,V,i \vdash \texttt{CONST i}}$$

Il faut aussi bien sûr ajouter une règle pour le typage des valeurs de type `int`, qui reflète le fait qu'une variable de type entier est représentée en mémoire par *val i*.

$$P,h \vdash val\ i : int$$

On dispose enfin d'instructions pour manipuler les entiers (opérations arithmétiques et logiques en particulier), qui sont évidemment typables de façon très simple.

Les booléens

Les booléens sont eux aussi introduits par l'instruction `CONST i`. Il convient donc de rajouter les règles de typage nécessaires à la fois au niveau de l'instruction de bytecode `CONST i` et au niveau du typage des valeurs dans la machine virtuelle. Ces règles sont très faciles à écrire et très proches de celles introduites à la section précédente pour les entiers.

Les booléens sont naturellement liés avec les instructions de sauts conditionnels et inconditionnels. Il convient donc de rajouter des règles qui expriment le fait qu'après un saut, le type de la machine virtuelle est le même qu'avant le saut, quelque soit le saut effectué.

Voici la règle de typage pour l'instruction `BRANCHIF`. Les règles pour `BRANCHIFNOT` et `BRANCH` sont analogues.

$$\frac{V(i) = \mathit{bool},\mathit{tcs},\mathit{tenv} \qquad V(i+1) \geq \mathit{bool},\mathit{tcs},\mathit{tenv} \qquad V(l) \geq \mathit{bool},\mathit{tcs},\mathit{tenv}}{P,V,i \vdash \texttt{BRANCHIF l}}$$

Les types concrets

Les types concrets sont représentés en mémoire par un bloc étiqueté par un identifiant et contenant une liste de valeurs. Cet identifiant est un entier qui indique le numéro du constructeur par rapport à la liste donnée pour définir le type. Pour les manipuler, on dispose d'une instruction `MAKEBLOCK i j` qui crée un bloc de taille `i` et d'identifiant `j` en prenant les valeurs à inclure sur la pile, ainsi que d'une instruction `SWITCH`, qui est l'instruction de base qui permet de compiler un `match`. C'est une généralisation de `BRANCHIF`, qui fait un saut dépendant de l'identifiant de l'objet présent dans l'accumulateur. Enfin, on dispose de l'instruction `GETFIELD i`, qui récupère le i^{eme} champ du bloc présent dans l'accumulateur. On a donc les règles de réduction suivantes :

$$\begin{array}{rll}
\texttt{MAKEBLOCK i j} & v_1,([v_2;\ldots;v_i]@\mathit{loc},\mathit{up}) \longrightarrow \mathit{block}\ j : [v_0;\ldots;v_i],(\mathit{loc},\mathit{up}) & \\
\texttt{GETFIELD k} & \mathit{block}\ j : [v_1;\ldots;v_i],\mathit{cs} \longrightarrow v_k,\mathit{cs} & \mathit{si}\ 0 \leq k \leq i \\
\texttt{SWITCH}\ \texttt{l}_0 \ldots \texttt{l}_\texttt{k} & \mathit{block}\ j : [v_0;\ldots;v_i],\mathit{pc} \longrightarrow \mathit{block}\ j : [v_0;\ldots;v_i],l_j & \mathit{si}\ j \leq k
\end{array}$$

On étend de plus la définition des types de la manière suivante :

$$\begin{array}{lll}
t ::= \ldots & & \\
& \mathit{Concr}\ c & \text{type concret} \\
& \mathit{Block}\ i : [t_1,\ldots,t_n] & \text{n-uplet tagué}
\end{array}$$

c décrit un type concret. Il est défini par la donnée d'une liste de tags, chacun associé à une liste de type, ce qui correspond à la façon dont ils sont définis en Caml.

La définition du type *Block* est rendue nécessaire par la dualité des types concrets. En effet, une même valeur peut être vue selon les moments soit comme un type concret soit comme un n-uplet. L'exemple (artificiel) suivant explicite le problème à l'aide du type concret `list` :

```
let test l =
  match l with
  | [] -> []
  | a::b -> l @ b
```

`l` est bien de type `list`, et pourtant, l'accès à `b` est compilé comme un accès à `l`, suivi de l'instruction `GETFIELD`, ce qui suggère que `l` est un n-uplet. Il est évidemment nécessaire de rajouter une règle de

sous-typage qui spécifie qu'une valeur de type *Block* est aussi une valeur de type *Concr c* si la définition de c comprend une ligne identique à ce bloc. On a donc une nouvelle règle de sous-typage :
$Block\ j : [t_0; \ldots ; t_i] \leq Concr\ c$, si la définition de c comprend une ligne $j,[t_0; \ldots ; t_i]$.

Pour typer un `SWITCH`, on remplace donc dans chacune des branches le type de la valeur sur laquelle il discrimine par le type *Block* correspondant à la branche considérée.

Remarque On peut facilement étendre cette définition aux types concrets paramétrés.

3. Implémentation

Un prototype de compilateur de Caml vers du bytecode typé est disponible sur le web à l'adresse suivante : `http://pauillac.inria.fr/~ailleret/recherche/taml/`.

3.1. Vue d'ensemble

La version actuelle dispose déjà de nombreuses fonctionnalités : polymorphisme, récursivité, types concrets paramétrés, références, exceptions et gestion de la récursion terminale. Le compilateur produit de plus du bytecode compatible avec OCaml (ce qui signifie qu'il est exécutable par `ocamlrun`, ce dernier ignorant les informations de type), ce qui permet de disposer de nombreux outils développés pour OCaml, en particulier du très efficace interprète.

L'ensemble se décompose en deux parties distinctes : `tamlc`, qui compile un fichier source vers un fichier de bytecode typé, et `tamlv`, qui vérifie la validité du bytecode typé avant son exécution par `ocamlrun`.

La compilation vers le bytecode typé se différencie assez peu de la compilation classique de OCaml avec `ocamlc`. La principale différence réside dans le fait que l'on désire connaître le type de toutes les fonctions utilisées (ainsi que l'instance de chaque opérateur). Ceci impose de modifier l'arbre de syntaxe abstraite pour accoler un type aux fonctions et aux opérateurs. Ces informations de type permettent, lors de la génération de code, de se souvenir du type de toutes les fonctions et donc de les stocker dans le bytecode. Leur connaissance permet aussi de sélectionner l'instruction qui correspond le mieux à une primitive (on dispose d'instructions spécialisées pour les primitives de comparaison si l'on compare des entiers).

3.2. Algorithme de vérification de type

L'algorithme de vérification de type implémenté dans `tamlv` est la partie la plus importante de Taml. C'est non seulement le seul composant exécuté à l'exécution, ce qui impose une bonne efficacité, mais c'est surtout sur lui que repose toute la sûreté. En effet, une erreur dans le compilateur n'est pas dangereuse pour l'utilisateur final, car elle est rattrapée par le vérificateur, mais une erreur dans le vérificateur n'est rattrapée par personne, ce qui en fait une cible privilégiée pour les pirates.

L'algorithme de vérification de type fonctionne de façon très similaire à ce qui est décrit dans la section 2. On vérifie la fonction principale, puis récursivement toutes les fonctions qui peuvent être appelées : lors de la vérification, dès que l'on rencontre une instruction `CLOSURE`, on marque l'étiquette de la fonction créée comme à vérifier.

La vérification d'une fonction se fait en parcourant de façon linéaire son code. On garde en permanence la liste des positions à vérifier et le type à l'entrée de chacune de ses instructions. On traite en permanence la plus petite position jusqu'à ne plus rien avoir à traiter. On est en effet obligé de garder une liste de positions, car le traitement d'un branchement conditionnel crée deux nouvelles branches à vérifier. Le nombre de branches diminue lorsque l'on rencontre un `RETURN`, où l'on vérifie que l'état final est correct, ou lorsque la nouvelle position à vérifier est déjà dans la liste. Dans ce cas, on fusionne les types des différentes branches et on continue. La vérification s'achève lorsque l'on a plus de position à vérifier.

La principale difficulté vient du fait que certaines des règles de typage sont non déterministes. C'est le cas en particulier pour les règles qui concernent `CONST i` et `MAKEBLOCK i j`. La première peut créer une valeur de type *unit*, *bool* ou *int*. La deuxième peut créer un n-uplet ou une valeur de type concret. En fait, tous ces types sont reliés, car un constructeur constant pour un type concret est codé par un entier et non par un bloc, pour des raisons évidentes d'efficacité. On pourrait rajouter des informations de types sur tous ces constructeurs, mais la taille des informations de type deviendrait beaucoup plus grosse pour un gain relativement faible. Il est cependant indispensable de donner certaines informations de type pour que la fusion de deux types reste une semi-unification acyclique. On utilise donc une généralisation du type *Block* défini lors de la présentation des types concrets. On utilise le type suivant :

```
type type_t = ...
  | Block_t of int * (int * vtype list) list
```

Le premier entier détermine la valeur du plus grand constructeur constant, et la liste qui suit donne l'ensemble des constructeurs non constants possibles. On dispose ensuite des règles de sous-typage évidentes pour fusionner deux `Block_t`, ou un `Block_t` avec un entier, un booléen ou un type concret.

Un mécanisme similaire est mis en place pour gérer les blocs mutables, qui peuvent être des références, des enregistrements mutables ou des tableaux.

On dispose donc d'un système d'inférence qui détermine le type de toutes les valeurs non fonctionnelles. Il est ici nécessaire de déterminer si les valeurs sont des entiers, des booléens ou des types concrets. En OCaml, ce problème n'existe pas car on dispose d'informations syntaxiques qui permettent de discriminer entre ces types. C'est ce manque d'information qui nous oblige à ajouter des informations de type au bytecode.

3.3. Performance et complexité

Comme le bytecode généré par `tamlc` est quasiment identique à celui généré par `ocamlc`, il est évident que la taille du code et la vitesse d'exécution sont les mêmes que pour OCaml. Il suffit donc de comparer la taille du code par rapport à la taille des types et la vitesse de vérification par rapport à la vitesse d'exécution pour se faire une idée du coût de la sûreté du bytecode.

Pour tester le compilateur, on utilise quelques petits programmes dont on mesure la taille du bytecode et les différentes vitesses d'exécution. Ces programmes sont :

- fib : programme minimal qui calcule la fonction de fibonacci. Deux versions ont été testées : l'une qui calcule `fib 5`, l'autre `fib 30`.
- erat : programme calculant les 100 premiers nombres premiers à base de listes pour garder en mémoire les prochains multiples des nombres premiers déjà trouvés.
- eval : petit interprète pour un petit sous-ensemble de Caml.

	fib	erat	eval
Code	132	844	2068
Type	11	73	256

TAB. 1 – *Taille du bytecode généré*

La première série de tests montre que la taille des types n'augmente que légèrement la taille du bytecode. Il est donc inutile de chercher à diminuer encore le nombre d'informations de types stockées, car le bénéfice serait très faible et risquerait d'augmenter la complexité du vérificateur.

	fib 5	fib 30	erat	eval
Compilation	1,14	1,14	2,14	4,54
Vérification	0,50	0,50	0,81	1,32
Exécution	0,98	278,84	6,42	-

TAB. 2 – *Vitesse d'exécution en ms*

La seconde série de tests nous donne un nouvel argument en faveur du bytecode par rapport au source compilé à la réception. Le temps de vérification est en effet beaucoup plus court que celui de la compilation, ce qui signifie qu'en plus de garantir une certaine confidentialité du code source, on gagne du temps de traitement au lancement de l'exécution sûre. De plus, ce phénomène s'accentue lorsque l'on améliore le compilateur, car la compilation se complexifie énormément lorsque l'on cherche à ajouter des constructions plus faciles à utiliser, alors que le bytecode reste le même (il peut même être simplifié par certaines optimisations).

On voit de plus que le temps de vérification est minime par rapport au temps d'exécution du programme, et ceci d'autant plus que le programme contient de boucles, ce qui n'est pas rare en réalité. De plus, une seule vérification suffit pour un nombre quelconque d'exécutions, ce qui diminue encore la part de la vérification par rapport à l'exécution : il suffit en fait d'une vérification par migration.

4. Autres approches et développements futurs

4.1. Travaux apparentés

Le précurseur en matière de bytecode typé est sans aucun conteste JAVA [4]. Dès 1995, avec l'émergence de l'Internet, Sun a compris l'intérêt du typage de bytecode pour diffuser sur l'Internet du code de façon sûre, sans avoir à distribuer le source du programme. Notre approche diffère de celle de JAVA car notre langage de base n'est pas orienté objet, mais fonctionnel et polymorphe, ce qui requiert un système de type plus élaboré. De plus, le typage est entièrement statique, ce qui améliore l'efficacité à l'exécution. En effet, on libère de la place en évitant de stocker des informations de type et on évite de perdre du temps à faire des vérifications de type en principe inutiles.

Certaines extensions de JAVA cherchent à lui adjoindre du polymorphisme. Cet ajout se fait toujours de façon beaucoup moins efficace que dans TAML, du fait de la simplicité imposée du système de type du bytecode JAVA. En effet, il est nécessaire soit d'ajouter des vérifications dynamiques de types [11], soit de dupliquer le code [1], ce qui ralentit la vérification de code.

Une approche plus récente est celle de TAL (Typed Assembly Language) [9]. Elle s'appuie sur un travail théorique complexe qui construit une transformation du système F vers un assembleur typé en utilisant un langage intermédiaire à base de continuations et typé grâce à des types existentiels. Cette approche a l'inconvénient de rendre la compréhension plus difficile, car plus éloignée du langage source (en particulier le langage cible ne dispose pas de la notion de fermeture). De plus, cette approche utilise une grande quantité d'informations de type dans le code compilé, car chaque introduction de valeur nécessite une indication sur son type. Une évolution de TAL [8] a l'intérêt de compiler directement vers de l'assembleur pour une machine réelle.

PCC (Proof Carrying Code) [10] est une tentative de généralisation de la notion de bytecode typé. Un programme n'est plus accompagné par des informations de type mais par une preuve qu'il respecte certaines propriétés. L'avantage théorique de ce système est que la propriété à prouver peut être infiniment plus variée que des propriétés de typage. Malheureusement, ce système nécessite des outils de vérification plus évolués que pour une vérification de type, et il est très difficile de générer automatiquement des preuves pour des propriétés plus fortes que le typage. De plus, des implémentations pour compiler du JAVA montrent que le vérificateur doit inclure de nombreuses règles qui sont en fait très spécifiques à JAVA.

4.2. Développements logiciels

Le développement de TAML est très récent, et de nombreux points sont encore à améliorer avant qu'il ne soit réellement utilisable par quelqu'un d'autre que son concepteur.

La première partie des améliorations concerne les fonctionnalités à ajouter au langage pour faciliter la programmation. Ces ajouts concernent principalement des fonctionnalités familières aux utilisateurs de Caml, mais non encore implémentées. La plus importante est la possibilité de gérer les modules, non seulement pour gérer les gros programmes, mais surtout pour faciliter l'utilisation des librairies standards. Ces dernières posent d'ailleurs un gros problème, car plusieurs d'entre elles font appel à des aspects non sûrs de Caml. Ils sont bien évidemment soigneusement utilisés, mais leur sûreté est difficile à prouver dans notre approche basée sur le typage. Enfin, il reste à améliorer le pattern-matching et à ajouter des extensions syntaxiques comme les enregistrements.

Un objectif plus ambitieux serait d'essayer de modifier le compilateur OCAML pour y intégrer TAML, dans le but de diffuser TAML comme un patch pour OCAML, ou mieux de l'intégrer totalement. Une telle approche aurait l'avantage de permettre une réutilisation de code au lieu d'écrire des fonctionnalités en double (analyse syntaxique, typage. . .). De nombreux problèmes sont malgré tout à résoudre, en particulier le traitement de toutes les particularités les plus avancées d'OCAML comme les objets, les variants ou les labels. Une telle intégration serait intéressante dans le cadre de la prochaine version de JOCAML [2], un langage basé sur OCAML qui fournit des primitives efficaces pour la programmation distribuée et la mobilité de code. Actuellement, JOCAML ne propose aucune garantie de typage propre au code mobile, ce qui le rend vulnérable aux erreurs dans le bytecode, et peut mettre en péril tous les programmes JOCAML en cours d'exécution sur une machine. La migration de bytecode typé, couplée à une vérification de type lors de chaque réception de programme, augmenterait donc grandement la sûreté du système.

La correction du vérificateur est un problème complexe, car les erreurs peuvent se produire à de nombreux niveaux. En effet, si la correction des règles de typage est essentielle, il faut aussi vérifier que l'algorithme utilisé est correct et que l'implémentation correspond effectivement à cet algorithme. Une telle correspondance est difficile à prouver en pratique, car le programme est assez important (plusieurs milliers de lignes) et sujet à de fréquentes modifications (amélioration de la vitesse d'exécution, ajout de nouvelles fonctionnalités. . .) qui demandent à chaque fois de refaire la preuve.

Une approche plus pragmatique nous incite donc à prouver la correction du typage et celle de l'algorithme, et à tester le vérificateur grâce à de nombreux exemples de programmes corrects et incorrects afin de vérifier son bon comportement sur ces exemples. Cette technique est en particulier celle utilisée par Sun pour tester la validité des vérificateurs de bytecode JAVA. Le projet Kimera [14] de l'université de Washington affine cette méthode en testant des programmes aléatoires sur plusieurs vérificateurs différents et en testant la cohérence des résultats.

La vérification de type permet d'exécuter du code avec la certitude qu'il n'y aura pas de problèmes à l'exécution. Si le mécanisme de vérification peut échouer en provoquant lui-même une telle erreur, le gain s'avère faible. . . On doit donc s'assurer que le vérificateur répond correctement quelle que soit l'entrée, et ne finit pas en provoquant une erreur ou une exception. Un point à surveiller particulièrement est la sérialisation des données (en particulier des informations de type) dans l'exécutable. Il faut donc s'assurer que la désérialisation est sûre.

Une idée à tester serait de créer des exécutables corrects, mais relativement complexes, d'en modifier au hasard quelques octets puis de les faire valider par le vérificateur. Une première étape s'assure que le vérificateur répond par oui ou non sans provoquer d'erreur, et une seconde exécute les programmes jugés corrects, et vérifie qu'ils ne produisent pas d'erreurs à l'exécution.

4.3. Développements théoriques

La préservation du typage par réduction et le théorème de progression garantissent que les règles de typage présentées sont sûres, mais ceci n'est qu'une première étape. Pour que ce système de type soit réellement utilisable, il reste plusieurs points importants à prouver.

Le premier concerne la complétude de ce système de type vis-à-vis de Caml. En effet, il est important de savoir que tout programme Caml peut être compilé en du bytecode TAML correct. Ceci signifie que l'on veut être sûr que notre système de type est au moins aussi expressif que celui de Caml. Un second point lui est très fortement lié : il concerne la décidabilité du système de type. On tient à être sûr que le vérificateur est capable de déterminer en temps fini si un programme est correct ou non.

Enfin, il est important que la théorie recouvre tous les aspects du langage présents dans le compilateur. En effet, actuellement, l'implémentation est en avance sur la théorie, ce qui fait que certains traits du langage (comme par exemple les exceptions) sont présents dans le compilateur et le vérificateur, mais ne sont nullement couverts par la théorie.

4.4. Applications

Les domaines d'application de TAML sont de manière générale tous les projets qui utilisent du code mobile. Le plus connu du grand public et le plus ancien est sans aucun doute les applets. Il serait donc intéressant de disposer d'un plugin qui permette d'exécuter de façon sûre du bytecode Caml à l'intérieur d'un navigateur, de façon plus sûre que ce que permet aujourd'hui MMM [12].

Un autre cadre où le bytecode typé est utile est celui des réseaux actifs. Un réseau actif est un réseau dans lequel les paquets s'orientent eux-mêmes dans le réseau, car ils contiennent, en plus des données, du code qui s'exécute lors de leur arrivée sur un routeur et détermine, entre autre, leur prochain saut dans le réseau. Un exemple de tel système est PLAN [3]. Il est plus efficace d'avoir du bytecode que du code source à exécuter. Mettre du bytecode typé dans le paquet, au lieu de code source qu'il faut typer puis interpréter, s'avèrerait donc intéressant au niveau des performances, ce qui est le problème majeur des implémentations actuelles de réseaux actifs.

Une autre application potentielle concerne les systèmes de calculs distribués sur l'Internet, utilisés entre autres pour tester la solidité de clés de cryptage (ECDL, RC5...) ou chercher des extraterrestres [13]. L'avantage d'utiliser du bytecode typé est qu'il permet aux utilisateurs d'être sûrs que leur bonne volonté ne se retournera pas contre eux. En effet, actuellement, les utilisateurs exécutent des programmes dont ils ignorent totalement le contenu, ce qui est un risque majeur pour leur sécurité. Pour ne pas perdre inutilement de la puissance de calcul, il importerait alors de disposer d'un compilateur à la volée, comme celui développé par Fabrice Le Fessant.

Conclusion

Notre système permet de rajouter des types au bytecode existant de OCAML pour que le bytecode conserve les propriétés de sûreté du langage source. Cet ajout a pu se faire de manière entièrement compatible avec le système existant, aidé par le fait que le bytecode OCaml manipule des abstractions de haut-niveau (fermetures en particulier). Le système de type créé pour l'occasion est très proche de celui utilisé par OCaml, ce qui rend sa compréhension relativement aisée. De plus la taille des informations rajoutées à l'exécutable est relativement faible, car on dispose d'un mécanisme qui infère la majorité des types.

Les développements en cours concernent un enrichissement des fonctionnalités et des applications périphériques, pour autoriser une utilisation maximale de Taml dans un environnement impliquant du code mobile. La compatibilité avec OCaml permet déjà de profiter de programmes très efficaces pour utiliser Taml, en particulier l'interprète qui est très finement optimisé.

Références

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *Conference on Object-Oriented Programming: Systems, Languages and Applications*. ACM Press, 1997.

[2] Sylvain Conchon and Fabrice Le Fessant. JoCaml: Mobile Agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99)*, pages 22–29, Palm Springs, California, October 1999.

[3] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[4] The Java language : http://java.sun.com/.

[5] Xavier Leroy. The Objective Caml system : http://caml.inria.fr/.

[6] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[7] Xavier Leroy and François Rouaix. Security properties of typed applets. In *25th symposium Principles of Programming Languages*, pages 391–403. ACM Press, 1998.

[8] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic typed assembly language, 1999.

[9] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *25th ACM SIGPLAN International Conference on Functional Programming*, pages 85–97, San Diego, CA, USA, January 1998. ACM Press.

[10] George Necula and Peter Lee. Proof-Carrying Code. Technical Report CMU-CS-96-165, Carnegie Mellon University, 1997.

[11] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *24th symposium Principles of Programming Languages*. ACM Press, 1997.

[12] François Rouaix. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28, pages 1365–1371. Elsevier, May 1996.

[13] Seti@home : http://setiathome.ssl.berkeley.edu/.

[14] E. Sirer, S. McDirmid, and B. Bershad. Kimera: A Java system security architecture, 1997.