LIF

Université de Provence
Centre de Mathématiques et d'Informatique

CNRS – Université de Provence – Université de la Méditerranée

Vérification de code mobile ou embarqué

E. Bremont, M. Daoudou, S. Ravelomanana

19 juin 2004

Vérification de code mobile ou embarqué

E. Bremont, M. Daoudou, S. Ravelomanana

Sous la direction de Solange Coupet-Grimal

Université de Provence – Université de la Méditerranée

CMI, 39 Rue F. Joliot-Curie, 13453 Marseille France

{ebremont, mdaoudou, sravelom}@capucine.univ-mrs.fr

Résumé. Dans le monde d'aujourd'hui, de plus en plus d'applications s'échangent du code pour leur fonctionnement. Ce code, généralement échangé sous forme pré-compilé (bytecode) peut être affecté, soit accidentellement (erreur de programmation ou de transmission), soit intentionnellement (hackers,...). Le but de ce travail est de procéder à deux types de vérifications : une vérification de type et une vérification sur la forme des expressions manipulées, ce qui permettra, à terme et sous certaines conditions, de donner une borne sur la taille mémoire et le temps d'exécution nécessaire au code.

Mots clés. sécurité, vérification de bytecode, algorithme de Kildall,...

Contents

1 Introduction

Des transactions bancaires aux téléphones cellulaires en passant par les jeux sur consoles mobiles, de plus en plus d'applications requièrent l'utilisation de **code embarqué**. Ce code, échangé généralement sous une forme précompilée (**bytecode**), peut être naturellement le produit de la compilation d'un programme source écrit dans un langage donné (cartes à puce, ...), mais peut aussi être d'une provenance inconnue (jeux en réseaux, ...). Si dans le premier cas, on ne doute pas (ou peu) de sa correction, et ce, du fait - on peut le supposer - qu'il a été généré par un compilateur correct, il n'est pas garanti, en revanche, dans le deuxième cas que, la *main* l'ayant produit ne soit pas *malicieuse* ou du moins qu'elle ne se soit pas trompée. Il convient donc, avant de l'exécuter, de s'assurer qu'il est exempt d'erreurs et que son exécution sur la plate-forme hôte ne mettra pas en cause le bon fonctionnement de cette dernière, tant au niveau du **temps de calcul** qu'au niveau des **ressources mémoires utilisées**.

Il est démontré que si le code d'une fonction passe certaines vérifications avec succès, le temps et la taille mémoire nécessaires à son exécution peuvent être majorés polynômialement par rapport à la taille de ses arguments. Ces vérifications sont de quatre sortes ; outre l'analyse statique standard (vérification de *type*), une deuxième analyse, dite de *forme*, peut être faite sur le code. Les résultats obtenus à l'issue de ces deux premières vérifications sont ensuite combinés d'abord avec la vérification de *taille* - pour trouver une majoration de la taille mémoire nécessaire -, et enfin avec la vérification de *terminaison* pour montrer qu'on peut majorer le temps d'exécution du code.

Pour cela, le fournisseur du code l'accompagnera également d'annotations, notamment les certificats des fonctions qu'il aura définies dans son programme. Ces annotations seront confrontées au bytecode lors du chargement de ce dernier en mémoire, afin de l'authentifier.

Après avoir défini le langage qui sert de support à ces vérifications (section 2), la section 3 présentera un compilateur pour ce langage, en particulier, le fonctionnement du synthétiseur de types. La partie 4, quant à elle, définira une machine virtuelle permettant d'exécuter le bytecode généré. Enfin, on présentera dans la section 5 l'algorithme de Kildall, qu'on utilisera ensuite dans les sections 6.1 et 6.2 pour mener les deux premiers types de vérifications, objectifs de ce TER.

2 Le langage étudié

On se place dans le cadre d'un petit langage fonctionnel typé du premier ordre. Dans ce langage, un programme est une suite de déclarations de types, suivie par des définitions de fonctions (éventuellement mutuellement

récursives).

Définition 1 *Un identificateur est une suite de caractères alphanumériques, commençant par une lettre alphabétique.*

2.1 Les types

Un type est défini par ses constructeurs, chacun étant accompagné des types auxquels il s'applique.

Définition 2 *Soient $T_1, T_2, ..., T_n$ et T des types. Un **constructeur** de type T et construit sur $T_1, T_2, ..., T_n$ est une application de $T_1 * T_2 * \ldots * T_n$ dans T.*

Remarque 1 Dans cette définition, n peut valoir 0. Dans ce cas, le constructeur ne prend aucun argument et est dit constant.

Remarque 2 Les types étant définis récursivement, on peut avoir $T = T_i$ pour certaines valeurs de i.

Syntaxe de définition d'un type

La déclaration d'un type est introduite par le mot clé **type** suivi de son nom et de ses constructeurs séparés par le symbole *pipe*(|).

Exemple 1 (**type** et **of** sont des mots clés du langage)

```
type tnat = Z | S of tnat;;
type env = Nil | C of tnat * env;;
```

Dans le premier cas, on définit le type `tnat` des entiers naturels: un entier est soit zéro (`Z`), soit le successeur d'un entier (`S(tnat)`). Ainsi, $S(S(S(S(Z))))$ représente l'entier 4. Le deuxième cas définit le type `env` des listes d'entiers: Nil représente la liste vide, tandis que $C(n, ls)$, où n est de type `tnat` et ls de type `env`, est la liste obtenue en insérant n en tête de ls. Ainsi, la liste $[2; 0; 1; 2]$ sera représentée par $C(S(S(Z)), C(Z, C(S(Z), C(S(S(Z)), Nil))))$.

2.2 Les fonctions

La définition d'une fonction est introduite par le mot clé **function**, suivi du nom de la fonction et de ses paramètres mis entre parenthèses. Ensuite vient le corps de la fonction, qui peut être soit une expression, soit un filtrage.

Remarque 3 Une fonction prend au moins un argument.

Définition 3 *Une **variable** est un identificateur commençant par une lettre minuscule.*

2.2.1 Les expressions

Une expression est soit une *variable*, soit un *constructeur* appliqué éventuellement à des expressions, soit enfin une *fonction* appliquée à des expressions.

Ainsi, $x, S(x), f(x_1, x_2, Z)$ sont des expressions valides.

2.2.2 Les filtrages

Définition 4 *Un **motif** est soit une variable, soit un constructeur appliqué éventuellement à d'autres motifs.*

La syntaxe d'un filtrage est la suivante (**match** et **with** sont des mots clés du langage):

match x **with**
$m_1 \rightarrow exp_1|$
$m_2 \rightarrow exp_2|$
.....
$m_p \rightarrow exp_p$

Cette construction du filtrage consiste à *essayer* d'identifier successivement la variable x avec les motifs $m_1, m_2, ..., m_p$ (dans cet ordre). En cas de réussite avec le motif m_k (voir la définition 5 ci-dessous), alors, le filtrage est arrêté et l'expression exp_k est renvoyée. Si au contraire, aucune identification ne réussit, une erreur (une *exception* en fait) est retournée pour le filtrage.

Définition 5 *On dit que le motif m **filtre(ou identifie)** x si et seulement si l'une des trois conditions suivantes est vérifiée:*

1. *m est une variable*
2. *x est affectée à $C(p_1, p_2, \ldots, p_n)$, m est de la forme $C(m_1, m_2, \ldots, m_k)$, C étant un constructeur, les p_i et m_i étant des motifs tels que pour tout i, le motif m_i filtre p_i.*

Extension du filtrage : Au lieu de filtrer une variable x avec un motif m, on étend le filtrage à celui d'une liste de n variables par une liste de n motifs. Dans ce cas, l'identification réussit si et seulement si, pour tout i le i-ième motif *filtre* la i-ième variable (au sens de la définition 5).

Voici un exemple de fonction, `add`, qui calcule la somme de deux entiers positifs; elle utilise le type `tnat` défini dans la section 2.1 et un filtrage sur son premier argument.

Exemple 2

```
function add(x,y) = match x with
Z -> y // si x = 0 add(x,y)=y
|
S(u) -> add(u, S(y)) //add(u+1,y)=add(u,y+1)
;;
```

3 Un compilateur pour le langage

Le compilateur (fcompile.ml) génère, à partir d'un fichier source, d'une part le bytecode associé et d'autre part, un fichier de signatures récapitulant les prototypes de toutes les fonctions et des types définis dans le source.
Le bytecode se présente comme une suite d'instructions de la machine virtuelle (*cf.* section 4), qui possède 6 instructions: Load, Build, Branch, Return, Stop et Call.
Le synthétiseur de types a pour but de générer le prototype (type des arguments et type de retour) d'une fonction à partir de son code source, et par suite, de générer le fichier signatures.
Dans notre analyse, nous considérons deux sortes de types :

- les types définis explicitement dans le programme (on dira par la suite que ces types sont connus)
- les types polymorphes, c'est-à-dire qui dépendent de certains paramètres

Dans la suite, on dira que les types définis sont plus **précis** que les types polymorphes, ces derniers étant plus **généraux**.

Définition 6 *Un **type polymorphe** est une variable de type qui peut être instanciée par des types différents, selon son utilisation.*

Définition 7 *Une **fonction est polymorphe** si au moins un des types de ses arguments ou son type de retour est polymorphe.*

Définition 8 *Une **affectation de types** est un couple de types (t, t'), dont le membre droit t' est au moins aussi précis que le membre gauche. Un tel couple sera noté quelque fois $t \leftarrow t'$.*

Définition 9 *On dit que deux types sont **compatibles** si et seulement si l'une des conditions suivantes est vérifiée :*

- *les deux types sont connus et ils sont égaux.*
- *au moins un des deux est une variable de type.*

Algorithme de compatibilité (entre les arguments effectifs d'une fonction g et le certificat de g)

Soit $(args, rt) = ([t_1, t_2, \ldots, t_m], t_0)$ la signature de g, c'est-à-dire que g est de type $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_m \rightarrow t_0$; soit en outre ls la liste des types des arguments effectifs de g et n sa taille. On considère une liste d'affectations de types *subst*, initialement vide.

La vérification de compatibilité entre $(args, rt)$ et ls peut être décrite par l'algorithme suivant :

Compatibility(args, rt) ls = VerifyCompatibility(args, rt) ls [] avec

VerifyCompatibility $(args, rt)$ ls $subst$:
si $n \neq m$, alors erreur.
sinon
si $args = []$ alors
si $\exists a, (rt, a) \in subst$ alors a
sinon rt
sinon si $args$ est de la forme $(a :: l_1)$ et ls de la forme $(b :: l_2)$, alors
si $(a = b)$ alors VerifyCompatibility (l_1, rt) l_2 $subst$
sinon
si a est une variable de type et si $\exists c, (a, c) \in subst$ alors
si $b = c$ alors erreur
sinon VerifyCompatibility (l_1, rt) l_2 $(a, b) :: subst$

Définition 10 *L'**unification** de deux types retourne le plus précis des deux, s'ils sont compatibles. Dans le cas contraire, l'unification retourne une erreur.*

Algorithme d'unification :
Unification x y =
si $x = y$, retourne x
sinon si x est une variable de type, retourne y
sinon si y est une variable de type, retourne x
sinon erreur

Définition 11 *A un instant donné, l'**environnement de types** est l'ensemble associant les variables connues à leurs types pendant l'analyse d'un bloc.*

3.1 Mise en oeuvre

Le synthétiseur de types (fsig.ml) prend en argument une fonction (son nom et son bytecode) et produit son prototype (si elle est bien typée), c'est-à-dire le type le plus général de ses arguments et son type de retour. Lors du calcul de ce dernier, des contraintes sont générées sur les types rencontrés. Ensuite, on en déduit les types les plus généraux, si le système obtenu est soluble. Ces contraintes sont de la forme :

contraintes $\equiv (variabledetype, type)$ ou $(variabledetype, variabledetype)$

A chaque ajout d'une nouvelle contrainte, la fonction **add_equation** (cf. sous-section 3.3) permet de réduire et de résoudre le système.

3.2 Génération des contraintes et calcul du type de retour

Au départ, les arguments ont le type le plus général possible ; c'est pourquoi, dans l'implémentation, à chaque argument est associé une variable de type (initialisation de l'environnement).

Le corps d'une fonction est :

– Soit une expression qui peut être :

  - une variable
  - un constructeur avec ses arguments éventuels
  - une fonction avec ses arguments

– Soit un filtrage constitué d'une variable, d'un motif auquel on identifie la variable, et de deux corps (le premier s'exécute si l'identification réussit, l'autre en cas d'échec); le premier bloc sera appelé bloc positif et le deuxième bloc négatif. Par la suite, ce filtrage sera noté $\mathbf{Match}(v, m, body_1, body_2)$.

Nous allons suivre cette organisation pour montrer, cas par cas, comment se déroule la génération de contraintes et comment le type de retour est calculé. Tout d'abord, il est à noter que le type de retour d'une fonction est le type de son corps.

- Si le corps de la fonction est une expression e :

  Le type de celle-ci est le type de retour de la fonction. e peut avoir plusieurs formes :

  – si e est une variable v : dans ce cas, le type de e est celui de v (type pris dans l'environnement courant)
  – si $e = c(e_1, \ldots, e_n)$ (où c est un constructeur et chacun des e_i une expression) : si les types inférés (des arguments du constructeur) sont égaux aux types attendus, alors le type de e est celui du constructeur c, sinon il y a une erreur de typage.
  – si $e = f(e_1, \ldots, e_n)$ (où f est un symbole fonctionnel) : le type de e est le type de retour de f, si celle-ci est bien typée et que les types des arguments $e_1, \ldots, e_n$ sont compatibles avec la signature de f. Pour cela, on cherche le prototype de f dans la liste des signatures précédemment calculées. S'il n'y est pas, on suppose que le prototype de f est le plus général possible. Puis on le recalcule afin de préciser ses types.

- Si le corps de la fonction est un filtrage :

  Soient v la variable à filtrer, m le motif auquel on identifie v, $body_1$ le bloc positif et $body_2$ le bloc négatif. Les contraintes induites directement par le principe d'exécution d'un filtrage sont :

  – la variable v est de même type que le motif m (puisqu'on les identifie)
  – les deux blocs $body_1$ et $body_2$ sont de même type. En effet, si le filtrage réussit, le type de retour de la fonction sera égal au type

de $body_1$. Dans le cas contraire, le type de retour de la fonction sera égal au type de $body_2$. Or le type de retour pour une fonction donnée est unique.

Résumons les contraintes obtenues lors de l'analyse :
Analyse de **Match** $(v, m, body_1, body_2)$

- type de v =type de m
- type de *body1* = type de *body2*

Analyse de l'expression $c(e_1, ..., e_n)$, où c est un constructeur

- Soient $t_1, \ldots, t_n$ les types des arguments de c (dans la déclaration des types du programme source). Alors pour tout i, t_i = type de e_i

Analyse de l'expression $f(e_1, ..., e_n)$, où f est un symbole fonctionnel

- Soient $t_1, \ldots, t_n$ les types des arguments de f (dans la liste des signatures). Alors pour tout i, t_i = type de e_i

On obtient donc un système d'équations correspondant aux contraintes générées. Il s'agit alors de le transformer en un système d'affectations de types. Ainsi, à la fin de la résolution de ce système, on aura obtenu, pour le type de chaque argument et pour le type de retour, le type le plus général possible tout en étant assez précis pour respecter l'ensemble des contraintes. Par ailleurs, les affectations obtenues à partir des contraintes sont :
Pour un filtrage **Match**(v, m, b_1, b_2) :

- Si m est de type connu (*ie*, ce n'est pas une variable de type) alors :
  - → si le type de v est connu et que les types de v et m ne sont pas les mêmes, alors le bloc est mal typé.
  - → si le type de v est une variable de type alors on rajoute la contrainte *type de* $v \leftarrow$*type de* m
- Si le type de m est une variable de type : soient t_1 et t_2 les types de v et m et soit t le résultat de l'unification entre t_1 et t_2. Si $t_1 = t$, alors, on rajoute la contrainte $t_2 \leftarrow t_1$, sinon ($t_2 = t$) on rajoute la contrainte $t_1 \leftarrow t_2$.
- Si le type de b_1 et le type de b_2 sont deux types connus et différents, alors la fonction est mal typée.

  Sinon : soient t_1 et t_2 les types des blocs b_1 et b_2 et soit t le résultat de l'unification entre t_1 et t_2.
  - si $t_1 = t$, alors, on rajoute la contrainte $t_2 \leftarrow t_1$
  - si $t_2 = t$, alors, on rajoute la contrainte $t_1 \leftarrow t_2$.

Pour une expression $c(e_1, e_2, \ldots, e_n)$:

Soient $\tau_1, \ldots, \tau_n$ les types respectifs de $e_1, ..., e_n$, et soient $t_1, \ldots, t_n$ les types des arguments de c (récupérés dans la déclaration des types du programme source). On sait alors que $t_1, \ldots, t_n$ sont des types connus. Par conséquent, ils sont forcément plus précis que $\tau_1, \ldots, \tau_n$. Les égalités se traduisent par : $\forall i, 1 \leq i \leq n, \tau_i \leftarrow t_i$.

Pour une expression $f(e_1, ..., e_n)$:

Soient $t_1, \ldots, t_n$ les types des arguments de la fonction f (récupérés dans la liste des signatures de fonctions). Soient $\tau_1, \ldots, \tau_n$ les types respectifs de $e_1, ..., e_n$. Alors : $\forall i, 1 \leq i \leq n$,

$\rightarrow$ si t_i et τ_i sont connus et différents alors l'expression est mal typée

$\rightarrow$ sinon : soit u_i l'unification de t_i et τ_i

- si $t_i = u_i$ alors on rajoute la contrainte $\tau_i \leftarrow t_i$
- sinon on rajoute la contrainte $t_i \leftarrow \tau_i$

3.3 Résolution du système d'affectations

La résolution du système d'affectation se fait à chaque arrivée d'une nouvelle contrainte à l'aide de la fonction **add_equation**. A chaque ajout, on propage (par la fonction **apply_equation**) l'information apportée par celle-ci à toute la liste, afin d'obtenir une solution à ce système.

Préliminaires :

On définit la fonction **transitive** par:

transitive (t, t') (t_1, t_2) $ls =$

- Si $t_2 \neq t$ alors rajouter (t_1, t_2) à ls
- Sinon si $t' \neq t_1$ alors rajouter(t_1, t') à ls

On définit également la fonction **propa** par:

propa (t, t') $[(t_1, t'_1); \ldots; (t_n, t'_n)] =$
$res \leftarrow []$
Pour i de n à 1 faire
  $transitive$ (t, t') (t_i, t'_i) res
fin pour
retourner res

Algorithme apply_equation

Supposons que l'on souhaite rajouter le couple (x_1, x_2) à la liste d'affectation ls. Il y a plusieurs cas à distinguer :

apply_equation (t, t') $ls =$

- Si $t = t'$ alors retourner ls

- Sinon si $\forall (v, v') \in ls, v \neq t'$ alors $up \leftarrow t'$

  sinon soit up le second membre de la première occurence d'un couple (t, t'')

  retourner (t, up) ::(**propa** (t, up) ls

Et enfin, la fonction **add_equation**: elle rajoute une contrainte au système d'affectations déjà obtenu et propage la contrainte:

add_equation (t, t') $ls =$

- Si $t = t'$ alors retourner ls

- Sinon si $\forall (v, v') \in ls, v \neq t$ alors retourner **propa** (t, t') ls

  sinon soit t'' le second membre de la première occurence d'un couple (t, t''')

  $up \leftarrow$ **unification**(t', t'')

  $ls_1 \leftarrow$ **apply_equation** (t, up) ls

  $ls_2 \leftarrow$ **apply_equation** (t', up) ls_1

  $ls \leftarrow$ **apply_equation** (t'', up) ls_2

  retourner ls

4 La machine virtuelle

Le but de la machine virtuelle (notée par la suite VM) est de calculer les valeurs de chacune des expressions données.
Elle dispose de six instructions : Branch, Build, Call, Load, Stop, Return. Cela peut paraître peu, voire insuffisant, surtout comparé au très grand nombre d'instructions que peuvent comporter certaines plate-formes telles que Java (184 instructions), mais cela est largement suffisant pour développer les principes de vérifications qui nous intéressent.

En effet, malgré cet aspect simpliste, elle regroupe les principales fonctions que l'on peut attendre : branchement conditionnel, appel de fonctions, etc... et cela est parfaitement adapté à la mise en exergue des conséquences d'une modification malicieuse de bytecode.

La machine virtuelle, que nous considérons ici, travaille uniquement sur une pile, appelée pile d'exécution. Avant l'évaluation du bytecode b_f de chaque fonction f, on empile ses arguments. Puis, lors du traitement, la pile sera modifiée après analyse de chaque instruction de b_f, et ce, en suivant le schéma ci-dessous :

1. Branch (c, j) : *branchement conditionnel (traduction du match)*

   Identifie le sommet de la pile avec le constructeur c. Si le branchement réussit, c'est-à-dire que le sommet de la pile est de la forme

$c(a_1, \ldots, a_n)$, on dépile l'élément au sommet, puis on empile les arguments $a_1, \ldots, a_n$ de c et l'instruction suivante est exécutée, sinon, on saute à l'instruction j.

2. Build (c, n) : *appel à un constructeur c d'arité n*

   Les n arguments $(arg_1, \ldots, arg_n)$ nécessaires à l'appel du contructeur sont dépilés (si la taille de la pile est insuffisante, la VM renverra une erreur). Ensuite, la valeur $c(arg_1, \ldots, arg_n)$ est empilé.

3. Call(f, n) : *appel à une fonction f d'arité n*

   On dépile les n éléments $(a_1, \ldots, a_n)$ en sommet de pile (si la pile n'est pas assez grande, la VM renvoie une erreur) puis on empile le résultat de l'évaluation de $f(a_1, \ldots, a_n)$, celle-ci étant effectuée sur une nouvelle pile d'exécution.

4. Load(i) : *chargement d'une valeur d'adresse i*

   La VM empile la i-ième valeur de la pile.

5. Stop : *arrêt de la machine*

   La VM arrête l'analyse du bloc de la fonction en cours d'exécution. Un cas typique est l'apparition d'une erreur.

6. Return : *retour de résultat*

   L'analyse du bloc en cours s'arrête et l'état courant est retourné.

5 Vérification du bytecode : algorithme de Kildall

Le code d'une fonction sera certifié correct(pour un test donné) si chacune de ses instructions l'est. Pour cela, un *état* est associé à chaque instruction du code, et on dira que la p-ième instruction est correcte si elle concorde avec le p-ième état du programme, cette concordance étant dictée par la *fonction de flot* **step** définie ci-dessous. Les états vont dépendre du type de vérification effectuée et seront donc précisés (instanciés) dans les sections 6.1 et 6.2. Pour l'heure, notons qu'il y a un état particulier, l'*état incohérent*, noté $\top$ et dont la signification est assez explicite.

5.1 Préliminaires

Dans la suite, on notera Σ l'ensemble des états et on appellera **programme**, le type des listes à n états, n étant le nombre d'instructions du code en cours de vérification.

On suppose que :

- $\leq$ est un ordre partiel sur Σ. Intuitivement, dire que $s_1 \leq s_2$ pour 2 états s_1 et s_2, cela signifie que s_1 est *au moins aussi cohérent* que s_2.

- **sup** est une loi interne binaire sur Σ, et $\mathbf{sup}(s_1, s_2)$ est la borne supérieure de s_1, s_2 (selon la loi $\leq$)
- $\forall s \in \Sigma, s \leq \top$ (tout état est plus cohérent que l'état inconsistant)
- Il n'existe aucune suite infinie strictement décroissante d'états $s_1 > s_2 > \ldots > s_n > \ldots$, où $>$ est la relation définie par: $\forall (s_1, s_2) \in \Sigma^2, (s_1 > s_2) \Longleftrightarrow (s_2 \leq s_1 \wedge s_1 \neq s_2)$. On notera au passage que $>$ est la relation *moins cohérent que*.

Introduisons quelques notations:

- *boolean* est le type des booléens, ne contenant que les éléments $true$ et $false$.
- $\mathbb{N}list$ est le type des listes d'entiers naturels et $\Sigma list$ celui des listes d'états. En particulier, le type **programme** est une partie de $\Sigma list$.
- Étant donné une liste ls et un élément e, $e :: ls$ est la liste obtenue en insérant e au début de ls.
- Soit ls une liste à m éléments et k un entier.
  - Si $k \in \{0, 1, ..., m-1\}, ls[k]$ est le $(k+1)$-ième élément de ls
  - Si $k \geq m, ls[k]$ est un paramètre arbitrairement fixé.

On définit alors:

- Le **graphe de flot** d'un bytecode bc est un graphe orienté F, dont les sommets sont les numéros des instructions de bc et tel que, pour tous sommets i et j de F, (i, j) est un arc de F si et seulement si l'instruction j peut être exécutée juste après l'instruction i.
- **succs**: $\mathbb{N} \rightarrow \mathbb{N}list$. Pour un noeud p du graphe de flot du bytecode en cours de vérification, $\mathbf{succs}(p)$ renvoie la liste des noeuds successeurs de p.
- **step**: $\mathbb{N} * \Sigma \rightarrow (\mathbb{N} * \Sigma)list$. Il s'agit de la *fonction de flot*. $\mathbf{step}(p, s)$ calcule la liste de tous les couples (k, s_k), où k est un successeur de p et s_k l'état obtenu en exécutant l'instruction p à partir de l'état s.
- **correct**: programme $* \mathbb{N} \rightarrow$ *boolean*. $\mathbf{correct}(ss, p) = true$ si et seulement si le p-ième état de ss est correct.
- **correctprog:** programme $\rightarrow$ *boolean*. Un programme est correct si et seulement si tous ses états sont cohérents et corrects:

  $\mathbf{correctprog}(ss) \Longleftrightarrow \forall p, 1 \leq p \leq n, ss[p] \neq \top \wedge \mathbf{correct}(ss, p)$.
- On étend la définition de la relation $\leq$ sur $\Sigma list$:

- $[] \leq []$
- $\forall (s_1, s_2) \in \Sigma^2, \forall (l_1, l_2) \in (\Sigma list)^2$,
  $(s_1 \leq s_2 \wedge l_1 \leq l_2) \Rightarrow (s_1 :: l_1 \leq s_2 :: l_2)$.

Définition 12 *Soit* ***bcv****: programme → programme, une fonction.* ***bcv*** *est un* ***vérificateur de bytecode*** *si et seulement si pour tout programme ss, si la vérification par* ***bcv*** *de ss est consistant, alors il existe un programme correct ts au plus aussi cohérent que ss :*

$\forall ss \in$ *programme,* $\top \notin (\textbf{\textit{bcv}}\, ss) \iff (\exists ts \in \text{programme}, ss \leq ts \wedge$ ***correcteprog***(ts).

5.2 L'algorithme de Kildall

L'algorithme qu'on présente ici et qu'on utilise pour les vérifications de type et de forme est dû à G. Kildall, et il est prouvé que si la fonction de flot **step** est monotone, alors il s'agit d'un analyseur de flot. La monotonie signifie ici que si s_1 est plus cohérent que s_2, alors $\textbf{step}(s_1)$ est plus cohérent que $\textbf{step}(s_2)$. Mais avant de détailler l'algorithme, définissons quelques notions:

Définition 13 ***(Stabilité d'une instruction)***

stable*: programme* $* \mathbb{N} \rightarrow$ *boolean.*

Une instruction p est dite stable si et seulement si pour tout successeur q de p, le nouvel état associé à l'instruction q est au moins plus cohérent que l'ancien:

$\textbf{\textit{stable}}(ss, p) \iff \forall (q, s) \in \textbf{step}(p, ss[p]), s \leq ss[q]$

Définition 14 ***(Stabilité d'un programme)***

stableprog*: programme → boolean.*

Un programme est stable si et seulement si chacune de ses instructions est stable:

$\textbf{\textit{stableprog}}(ss) \iff \forall p, 1 \leq p \leq n$ $\textbf{\textit{stable}}(ss, p)$.

Définition 15 ***(analyseur de flots)***

Soit dfa: programme → programme. dfa est un ***analyseur de flot de données*** *si et seulement si pour tout programme ss, les 3 conditions suivantes sont vérifiées:*

- *dfa*(ss) *est un programme stable*
- $ss \leq$ *dfa*(ss) *: un programme est au moins aussi consistant que son image.*
- $\forall ts \in$ *programme,* $(ss \leq ts \wedge \textbf{\textit{stableprog}}(ts)) \Rightarrow$ *dfa*$(ss) \leq ts$ *: pour tout programme ts, si ts est stable et moins cohérent que ss, alors l'analyse par dfa de ss est plus cohérent que ts.*

Il peut alors être démontré que si les fonctions **stable** et **correct** vérifient la condition (C) ci-dessous, alors un analyseur de flot est un vérificateur de bytecode. Ainsi, si **stable** et **correct** vérifient (C) et que **step** est monotone, alors l'algorithme de Kildall sera un vérificateur de bytecode.

(C) : $\forall ss \in$ programme, $\forall p, 1 \leq p \leq n, (\top \notin ss \Rightarrow \textbf{correct}(ss, p)) \Longleftrightarrow \textbf{stable}(ss, p)$.

Dans l'algorithme ci-dessous, wl est l'ensemble des instructions dont les états associés ont changé à la dernière itération. On suppose qu'on a une fonction **replace** qui prend en paramètre une liste ls à m éléments, un entier $k \leq m$ et un élément elt et qui renvoie la liste obtenue à partir de ls, en remplaçant son k-ième élément par elt.

Algorithme Kildall

- Entrées:
  - un programme ss, i.e une liste de n états
  - un code $bcode$ de n instructions
  - une relation $\leq$ sur Σ
  - une loi sup sur Σ
  - une fonction de flot $step$
- Sortie: un programme $reslt$
- Spécifications: $reslt$ est obtenu à partir de ss en remplaçant les états associés aux instructions instables de $bcode$ par les états calculés à partir de leurs prédecesseurs.
- Méthode:

  // 1. Initialisation

  $wl \leftarrow \varnothing$

  $reslt \leftarrow ss$

  // Génération de la work list

  Pour chaque numéro d'instruction p dans $bcode$ faire

  Si p n'est pas stable alors $wl \leftarrow wl \cup \{p\}$

  fin pour

  Tant que $wl \neq \varnothing$ faire

  Extraire un numéro d'instruction p de wl

  // 2. Propagation

  $sucp \leftarrow step(ss, p)$ //calculer les successeurs de p

  Pour chaque couple (q, s) de $sucp$ faire

$up \leftarrow sup(ss[q], s)$

Si $up \neq ss[q]$ alors // q n'est donc pas encore stable

$wl \leftarrow wl \cup \{q\}$

$reslt \leftarrow replace(reslt, q, up)$

fin pour

fin tant que

retourner $reslt$

Fonctionnement de l'algorithme : wl est l'ensemble des numéros d'instructions non encore stables. L'idée est de partir d'un noeud instable et de calculer ses successeurs. S'il y en a parmi ceux-ci dont les états associés ont changé, c'est qu'ils ne sont pas encore stables et doivent donc être rajoutés à la work list. Le procédé est itéré jusqu'à ce que toutes les instructions soient stables. Ainsi, à la fin de l'algoritme(il est démontré qu'il termine), toutes les instructions de *bcode* sont stables.

6 Applications de l'algorithme de Kildall : vérifications de type et de forme

On vient de voir un algorithme générique permettant de *vérifier* du bytecode. Comme mentionné au 5, la notion de *correction* du code dépend du type de vérification qu'on effectue. Les deux types de vérifications consistent en une exécution symbolique de la machine virtuelle définie en 4, les symboles manipulés étant des types pour la vérification statique et des expressions pour la forme. Après avoir défini formellement la notion d'état, on donnera pour chacune des deux, la fonction de flot **step** permettant de propager les états dans le programme.

Un état est soit l'*état chaos*($\bot$), soit l'*état incohérent*($\top$), soit enfin une liste de symboles. On a donc :

$\Sigma = \{\bot, \top\} \cup \mathcal{S}$, où $\mathcal{S}$ est à préciser dans chaque type de vérification.

On définit alors la relation $\leq$ par:

- $\forall s \in \Sigma, \bot \leq s$
- $\forall s \in \Sigma, s \leq \top$
- $\forall s_1, s_2 \in \Sigma \setminus \{\bot, \top\}, s_1 \leq s_2 \iff s_1 = s_2$

Puis la loi (commutative) *sup* par:

- $\forall s \in \Sigma, \sup(s, \bot) = s$
- $\forall s \in \Sigma, \sup(s, \top) = \top$

- $\forall s_1, s_2 \in \Sigma \setminus \{\bot, \top\}, \sup(s_1, s_2) = s_1$ si $s_1 = s_2$ sinon $\sup(s_1, s_2) = \top$

Et enfin, la fonction *succs* donnant la liste des successeurs d'une instruction (n est la longueur du bytecode en cours d'analyse):

Instruction p	$succs(p)$
Return	$[p]$
Stop	[]
Branch c j	si $p+1 < n$ et $j < n$ alors$[p+1; j]$ sinon $[p; p]$
Load j	si $p+1 < n$ alors$[p+1]$ sinon $[p]$
Build c m	si $p+1 < n$ alors$[p+1]$ sinon $[p]$
Call g m	si $p+1 < n$ alors$[p+1]$ sinon $[p]$

Par ailleurs, pour deux listes $l_1 = [a_1, \ldots, a_n]$ et $l_2 = [b_1, \ldots, b_n]$ de même taille, on définit **combine** (l_1, l_2) comme étant la liste de couples $[(a_1, b_1), \ldots, (a_n, b_n)]$.

6.1 Vérification de type

La vérification de type permet d'assurer la cohérence des types entre les signatures accompagnant le bytecode et les valeurs réellement manipulées. De plus, il permet de détecter les débordements mémoire dus à des opérations illégales (branchement à une adresse inexistante, chargement d'une variable à une adresse inconnue, ...)

Exemple 3 Considérons le cas de la fonction *succ* suivante (les types `tnat` et `env` sont définis en 2.1):

```
(* succ : prend en argument un entier x et renvoie x+1 *)
let succ (x) = S (x);;
```

Soit le byte code associé (obtenu après compilation de la fonction) :

```
Load 0;
Build (S,1);
Return;
```

Sans la vérification de type, le résultat de l'évaluation de $succ(Nil)$ serait $S(Nil)$, c'est-à-dire $Nil+1$, ce qui est évidemment incorrect sémantiquement. Pour ce type de vérification, les symboles manipulés sont des listes de types. Soient s l'état courant et p l'instruction à exécuter. La fonction de flot associée à la vérification de type est définie par :

step $(p, s) = combine(succsp, fstep(p, s))$, $fstep$ étant définie ci-dessous selon l'instruction à exécuter.

1. Branch c j :

   $fstep(p, s) =$

- si $s = \bot$ alors $[\bot; \bot]$
- si $s = \top$ alors $[\top; \top]$
- si $s \in \mathcal{S}$ alors
  - si s est de la forme $l@[rt]$ alors
    soit $(t_1, \ldots, t_n) \rightarrow t$ la signature de c
    si $rt = t$ alors $[l@[t_1, \ldots, t_n]; s]$
    sinon $[\top; \top]$
  - sinon $[\top; \top]$

2. Build $c\ n$:

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - si $s = \top$ alors $[\top]$
   - si $s \in \mathcal{S}$ alors
     - si s est de la forme $l@[ts_1, ..., ts_n]$ alors
       soit $(t_1, \ldots, t_n) \rightarrow t$ la signature de c
       si $\forall i, 1 \leq i \leq n, compatibles(t_i, ts_n)$ alors $[l@[t]]$
       sinon $[\top]$
     - sinon $[\top]$

3. Call $g\ n$:

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - si $s = \top$ alors $[\top]$
   - si $s \in \mathcal{S}$ alors
     - si s est de la forme $l@[ts_1, ..., ts_n]$ alors
       soit $s = (t_1, \ldots, t_n) \rightarrow t$, la signature de f
       $rt \leftarrow Compatibility\ s\ [ts_1, ..., ts_n]$
       s'il y a erreur alors $[\top]$
       sinon $[l@rt]$
     - sinon $[\top]$

4. Load j :

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - si $s = \top$ alors $[\top]$
   - si $s \in \mathcal{S}$ alors
     si s est de la forme $[ts_1, ..., ts_n]$ alors $[ts_1, ..., ts_n, ts_j]$
     sinon $[\top]$

5. Stop : $fstep(p, s) = []$

6. Return :

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - si $s = \top$ alors $[\top]$
   - si $s \in \mathcal{S}$ alors
     - si s est de la forme $l@[rt]$ alors
       soit $s = (t_1, \ldots, t_n) \rightarrow t$ la signature de f
       si $rt = t$ alors $l@[rt]$
       sinon $[\top]$
     - sinon $[\top]$

On peut introduire maintenant, la notion de correction d'une fonction : une instruction p est ***correcte*** par rapport à l'état courant σ si et seulement si $(\sigma \neq \top)$ et$(\top \notin fstep(p, \sigma))$

Soit f la fonction en cours d'analyse. Le programme (liste d'états) associé à f est **correct** si et seulement si toutes ses instructions sont correctes et que son premier état est de la forme $[t_1, \ldots, t_n]$ où n est l'arité de f, cette dernière prenant des paramètres de types $t_1, \ldots, t_n$.

6.2 Vérification de forme

Comme annoncé au début du paragraphe 6, la vérification de forme est axée sur la forme des expressions manipulées(et non pas sur les types). On suppose donc qu'une vérification de type a deja été faite sur le bytecode, et que toutes les fonctions définies sont bien typées.

Il s'agit donc de définir les états manipulés ainsi que la fonction de flot correspondant.

6.2.1 Les états

Pour la vérification de forme, les symboles manipulés sont des expressions (au sens de la définition 2.2.1). Toutefois, on associe à chaque état une substitution, c'est à dire une liste de couples (v, e), où v est une variable et e une expression. Ces substitutions sont utiles pour mener les vérifications de taille et de terminaison. En notant $\mathfrak{S}$ l'ensemble des substitutions et E celui des expressions, on a ici : $\Sigma = \{\bot, \top\} \cup ((E\ list) * \mathfrak{S})$

6.2.2 La fonction de flot

La fonction de flot permet de propager les états des instructions instables vers leurs successeurs dans le graphe de flot. On rappelle également que cette

fonction est une exécution symbolique de la machine virtuelle. Il est donc naturel que la propagation dépende de l'instruction instable à considérer. On suppose que la machine se trouve dans un état $s \in \Sigma$, qu'on traite l'instruction p et on va donner les règles de propagation de s, ainsi que la définition de la correction de l'instruction p: dans ce qui suit, σ, σ_i sont des substitutions, e, e_i sont des expressions et ls une liste d'expressions. Pour deux listes l_1 et l_2, on notera $l_1@l_2$ la liste obtenue en concaténant les listes l_1 et l_2, l_2 étant en tête, et $\sigma_1 \circ \sigma_2$ représentera la composée(mathématique) des substitutions σ_1 et σ_2. On suppose l'existence d'une fonction **new_vars** qui prend en argument un entier k et qui renvoie k variables fraiches(non encore utilisées), et une fonction **arity** qui renvoie l'arité d'un constructeur.

fstep étant la fonction définie ci-dessous, la fonction de flot est définie par :

step(p, s)= $combine(succsp, fstep(p, s))$.

Comme dans le cas de la vérification de types, la fontion $fstep$ est définie par cas, selon l'instruction à exécuter :

1. Return :

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - sinon
     - si s est de la forme $ls@[e], \sigma$ alors $[s]$
     - sinon $[\top]$

2. Stop : $fstep(p, s) = []$

3. Load j :

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - sinon
     - si s est de la forme(ls, σ) alors
       si $j < n$ alors$[(ls@[ls[j]], \sigma)]$
       sinon $[\top]$
     - sinon $[\top]$

4. Build $c\ m$:

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - sinon
     - si s est de la forme $(ls@[e_1, e_2, \ldots, e_m], \sigma)$ alors $[(ls@[c(e_2, \ldots, e_m)], \sigma)]$

- sinon $[\top]$

5. Call $g\ m$:

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot]$
   - sinon
     - si s est de la forme $(ls@[e_1, e_2, \ldots, e_m], \sigma)$ alors $[(ls@[g(e_2, \ldots, e_m)], \sigma)]$
     - sinon $[\top]$

6. Branch $c\ label$:

   $fstep(p, s) =$

   - si $s = \bot$ alors $[\bot; \bot]$
   - sinon
     - si s est de la forme $(ls@[v], \sigma)$ alors
       soit $vars := \mathbf{new\_vars}(\mathbf{arity}(c))$
       soit $e := c(vars)$
       retourner $[(ls[v := e]@vars, [(v, e)] \circ \sigma); s]$
     - sinon
       si s est de la forme $(ls@[cons(ls_1)], \sigma)$ alors
       * si $cons = c$ alors $[(ls@ls_1, \sigma); s]$
       * sinon $[\bot; s]$

       sinon $[\top; \top]$

On peut alors définir **correct** comme suit : une instruction est bien formée si et seulement si l'état associé est cohérent et qu'aucun de ses états successeurs n'est incohérent. Et un programme est bien formé si tous ses états sont consistants et si le premier état (celui associé à la première instruction) est de la forme $([v_1, v_2, \ldots, v_m], id)$, où m est l'arité de la fonction en cours d'analyse et id la substitution identité.

7 Conclusion

Avec le développement des systèmes de télécommunications, il est très courant que du code soit échangé sous forme pré-compilé. Or, les personnes vectrices ou les moyens utilisés pour les échanges peuvent affecter (in)consciemment ce code. Il est donc de mise de le soumettre à des contrôles avant de le charger en mémoire sur le support hôte. Il devient donc essentiel d'établir des méthodes permettant de s'assurer de la validité du code transmis.

En s'appuyant sur l'algorithme de G. Kildall, on a développé deux types de vérifications sur le code : d'une part, la vérification de type, d'autre part celle de la forme des expressions manipulées.

Certes, ces deux analyses permettent de rejetter un grand nombre de programmes incorrects, et ainsi d'accroître considérablement la sûreté d'exécution; mais ils ne sauraient être considérées comme suffisantes, vu l'expansion du piratage informatique. Comme on l'a mentionné, d'autres tests pourraient encore être menés, en particulier les vérifications de terminaison et de taille, surtout si l'on sait qu'en général, les plate-formes sur lesquelles sont exécutés ces codes sont d'une capacité mémoire très réduite.

8 Références

[AMA04] R. Amadio. Synthesis of max-plus quasi-interpretations. Research report. January 2004.
[ACZJ04] R. Amadio and S. Coupet-Grimal and S. Dal Zilio and Line Jakubiec. A functional scenario for bytecode verification of resource bounds. Research report. January 2004
[Kil73] G. A. Kildall. A unified approach to global program optimization. Boston, MA, Octobre 1973.
[Ler01] X. Leroy. On-card bytecode verification for Java Card. 2001.
[LY99] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1999.
[MWCG99] G. Morriset, D. Walker, K. Crary and N. Glew. From System F to Typed Assembly Language. ACM Transactions on Programming Languages and Systems,1999.
[Nec97] G. Necula. Proof carrying code, 1997.
[Nip01] T. Nipkow. Verified Bytecode verifiers. Foundations of Software Science and Computation Structures. Springer-Verlag, 2001.
[Ray01] D. Rayside. A generic worklist algorithm for graph reachability problems in program analysis. University of Waterloo, 2001.
[San01] D. Sannella. Mobile resource guarantee. Ist-global computing research proposal, U. Edinburgh, 2001. http://www.dcs.ed.ac.uk/home/mrg/.