Rapport de synthèse de l'article

Java Generics are Turing Complete

Marie Laporte [1]
Patrick Chen [1]

Date de soumission de l'article : 09/11/2017

Résumé: Ce papier se veut être un rapport sur l'article de Radu Grigore[4] posant le problème suivant : "Au sein du langage Java, le contrôle de type est-il décidable ?". Plusieurs articles traitent de sujets similaires, mais aucun ne prouve réellement, de manière formelle, que la réponse au problème est soit vraie soit fausse. En effet, les articles de Kennedy and Pierce 2007, Wehr and Thiemann 2009, Tate et al. 2011, prouvent l'indécidabilité du contrôleur de type dans des langages particuliers, mais autorisent des modèles interdits en Java au cours des réductions (exemple : l'héritage multiple qui est interdit en Java). Nous allons ici présenter et commenter l'article de Radu Grigore prouvant que le contrôleur de type est bien indécidable en particulier pour le langage Java.

1 Introduction

Nous savons que le contrôleur de type dans plusieurs langages est indécidable, mais l'est-il dans le cas du langage Java ? C'est ce que démontre l'article de Radu que nous allons expliciter ici.

Definition (Problème décidable)**.** Un problème est dit **décidable**, si il existe un algorithme, une procédure mécanique qui termine en un nombre fini d'étapes, qui le décide, c'est-à-dire qui réponde par oui ou par non à la question posée par le problème.

Afin de prouver l'indécidabilité du langage Java, on va avant tout démontrer les deux théorèmes suivants :

Théorème 1. Il est indécidable si t est un sous-type de t' (notation : $t <: t'$) selon une table de classe donnée.

Théorème 2. Soit G une grammaire non contextuelle décrivant un langage $\mathcal{L} \in \Sigma^*$ dans un alphabet Σ de noms de méthodes. On peut construire des définitions d'une classe Java, un type T, et des expressions Start et Stop de telle sorte que le code

$$T\ell = Start.f^{(1)}().f^{(2)}()...f^{(m)}().Stop()$$

est correctement typé si et seulement si $f^{(1)}().f^{(2)}()...f^{(m)}() \in \mathcal{L}$. De plus, les définitions de classe sont de taille polynomiales à la taille de G, et le code Java peut avoir un contrôle de type en temps polynomial à la taille de G.

2 Sous-typage

Dans l'ensemble de cette partie, nous allons commencer par démontrer le **théorème 1**. Elle sera introduite par quelques rappels de notions importantes sur les classes génériques de Java, ainsi que quelques définitions liées au sous-typage.

2.1 Rappels sur les classes génériques de Java

De façon général, la généricité dans un langage de programmation permet à toute structure ayant des données un typage quelquonque et variable de ceux-ci. Il peut bien sûr y avoir des restrictions de tout type quant à la nature du typage quelquonque. Le lien entre la généricité et le sous-typage n'est pas trivial.

Comme précisé dans l'article on peut prendre l'exemple de $Integer$ et $Number$. En effet, $Number$ n'est pas un sous-type de $Integer$, mais l'inverse est vrai ($Number$ étant une classe abstraite, on ne peut donc pas insérer des $Number$ dans une liste de $Integer$). Si on considère des méthodes Java manipulant par exemple une liste de type <? **super** Integer> (supporté par $Integer$ et $Number$), Il y aura une différence au niveau de la vérification du typage pour deux methodes qui feraient la même chose mais qui auraient les types de retour suivants : <? **extends** Number> et <x **extends** Number> avec x comme type. Ceci est dû au fait que $Integer$ est un sous-type de <? **extends** Number>.

En Java, rappelons également que si B est un sous type de A alors <? **extends** B > <: <? **extends** A> mais que <? **super** A> <: <? **super** B>. Plus globalement, lorsque nous utilisons le type générique en Java, il va falloir jongler avec les notions de <? **extends** A> ou bien <? **extends** B>, et les notions de covariance et contravariance (remplacer le typage par ? extends ou ? super).

2.2 Le système de typage

L'article reprend dans l'essentiel les normes de l'article de Kennedy et Pierce 2007. On peut en rappeller ici quelque notions :

- le symbole $<:$ s'utilise dans un contexte de table de classe, qui est un emsemble de règles d'héritage. Cette relation de sous-typage ce traduit par :

$$\frac{At_1 \dots t_m <::^* Ct'_1 \dots t'_n \qquad t''_1 <: t'_1 \quad \dots \quad t''_n <: t'_n}{At_1 \dots t_m <: Ct''_1 \dots t''_n} \quad (1)$$

  Si $t1 <: t2$ on dit que $t2$ est supertype de $t1$.

- Par abus de language $<:: *$ est la fermeture transitive réflexive.
- Java n'autorisant pas l'héritage multiple, nous avons que si $t <:: *Ct_1..t_m$ et $t <:: *Ct'_1...t'_m$ alors $t_i = t'_i$.

2.3 Machines de Turing

La démonstration du **théorème 1** va passer par une réduction au problème d'arrêt d'une machine de Turing.

Definition (Problème de l'arrêt)**.** Le **problème de l'arrêt** d'une machine est le problème de décision qui détermine, à partir d'une description d'un programme informatique, si le programme s'arrête ou non.

On utilise cette réduction puisqu'il a été démontré que **le problème de l'arrêt est indécidable**.[2] De plus, un tel programme informatique sera excécutable sur une machine de Turing, défini de la manière suivante :

Definition (Machine de Turing)**.** Une **machine de Turing** est un modèle abstrait du fonctionnement des appareils mécaniques de calcul, tel un ordinateur et sa mémoire. Une machine de Turing comporte les éléments suivants :

- Un **ruban infini** divisé en cases consécutives. Chaque case contient un symbole parmi un alphabet fini.
- Une **tête de lecture/écriture** qui peut lire et écrire les symboles sur le ruban, et se déplacer vers la gauche ou vers la droite du ruban.
- Un **registre d'état** qui mémorise l'état courant de la machine de Turing.
- Une **table d'actions** qui indique à la machine quel symbole écrire sur le ruban, comment déplacer la tête de lecture, et quel est le nouvel état, en fonction du symbole lu sur le ruban et de l'état courant de la machine. Si aucune action n'existe pour une combinaison donnée d'un symbole lu et d'un état courant, la machine s'arrête.

On peut représenter une telle machine $\mathcal{T}$ par un **automate** $(Q, q_1, q_H, \Sigma, \delta)$ où Q est un ensemble d'états, q_1 l'état initial, q_H l'état d'arrêt, Σ un alphabet (= un ensemble de lettres), et $\delta \colon Q \times \Sigma_\perp \to Q \times \Sigma \times \{L, R, S\}$ une fonction de transition avec L pour un décalage à gauche sur le ruban et R pour un décalage à droite sur le ruban (S correspond à aucun décalage).

Une **configuration** de la machine sera représentée par un tuple (q, α, b, γ) avec $q \in Q$ l'état courant, $\alpha \in \Sigma^*$ la partie gauche du ruban, $b \in \Sigma_\perp$ le symbole courant, et $\gamma \in \Sigma^*$ la partie droite du ruban. Une machine va donc avoir plusieurs étapes d'excécutions en changeant de configuration. Par exemple, si on a $\delta(q, b) = (q', b', L)$, alors $(q, \alpha a, b, \gamma) \to (q', \alpha, a, b'\gamma)$. Si le ruban donné au départ est α_1, alors on aura une séquence d'excécutions commençant par la configuration $(q_1, \epsilon, \perp, \alpha_1)$. Si l'état q_H est atteint par $\mathcal{T}$, alors $\mathcal{T}$ s'arrête sur α_1. Cependant, comme on l'a dit précédement, le problème de l'arrêt de machine est indécidable par le théorème de Turing suivant :

Théorème 3 (Turing)**.** Il est indécidable si une machine de Turing $\mathcal{T}$ s'arrête sur α_1.

2.4 Machines sous-typées

Comme précisé dans l'article, la classe Z a une arité de zéro (son nombre d'arguments). Une machine sous-typée est donc definie comme suis :

$$C_1 C_2 ... C_m Z <: D_1 D_2 ... D_n Z.$$

On peut ainsi définir quelques règles qui correspondent à des exécutions de machines sous-typées. Par exemple, les machines sous-typées sont déterministes puisque l'héritage multiple n'est pas autorisé en Java. L'éxecution de ses règles forme une preuve partielle pour les réductions. On peut ainsi définir des tables de classes et effectuer une réduction sur une expression de la forme $A <:: B$. Il est important de noter que $C_1 ... C_m Z <: D_1 ... D_n Z$ si et seulement si l'étape d'éxecution arrive au point d'arrêt. L'article montre un exemple, avec une requête donnée, une éxecution de la reduction en utilisant des règles particulières à suivre.

2.5 Première preuve

Afin de prouver le **théorème 1** grâce à une machine de Turing $\mathcal{T}$, l'article va construire les types t_1 et t_2, et une table de classe tel que $t_1 <: t_2$ si et seulement si $\mathcal{T}$ s'arrête en α_1 (cela impliquera donc que $t_1 <: t_2$ est bien indécidable puisque l'arrêt est indécidable). Une telle table a été définie à l'aide de plusieurs nouvelles classes à introduire.

- Pour chaque état $q_s \in Q$, on aura 6 classes Q_s^{wL}, Q_s^{wR}, Q_s^L, Q_s^R, Q_s^{LR}, et Q_s^{RL} qui indiqueront que nous sommes en train de simuler l'état Turing s. Une classe

Q_s^{w*} (w pour $wait$) indique que la tête de la machine sous-typée n'est pas à la mêmes positions que la tête de la machine de Turing simulée. S'il n'y a pas de w, alors elles ont les mêmes positions de tête. Une classe Q_s^L (resp. Q_s^R) indique que la tête de la machine sous-typée bouge à gauche (resp. à droite). La classe Q_s^{LR} indique que la machine sous-typée bougait à gauche mais va maintenant tourner à droite.

- Pour chaque lettre $a \in \Sigma \cup \{\#\}$ avec $\#$ une lettre fraiche qui permettra de délimiter le ruban, il y a une classe L_a qui représente des symboles de la machine sous-typée.
- Des classes M^L et M^R qui marquent la position de la tête de la machine de Turing (à gauche ou à droite du marqueur).
- Une classe E qui représente la fin d'un marqueur de ruban.
- Une classe N permettant la covariance.

A l'aide de l'ensemble de ces classes, on peut déterminer la table suivante :

$Q_s^{\mathsf{wL}} x <:: M^{\mathsf{L}} N Q_s^{\mathsf{L}} x$	for $q_s \in Q$	$Q_s^{\mathsf{L}} x <:: L_a N Q_{s'}^{\mathsf{wL}} M^{\mathsf{L}} N L_b N x$	for $\delta(q_s, a) = (q_{s'}, b, \mathsf{L})$
$Q_s^{\mathsf{wL}} x <:: M^{\mathsf{R}} N Q_s^{\mathsf{wL}} M^{\mathsf{R}} N x$	for $q_s \in Q$	$Q_s^{\mathsf{L}} x <:: L_a N Q_{s'}^{\mathsf{wL}} M^{\mathsf{R}} N L_b N x$	for $\delta(q_s, a) = (q_{s'}, b, \mathsf{S})$
$Q_s^{\mathsf{wL}} x <:: L_a N Q_s^{\mathsf{wL}} L_a N x$	for $q_s \in Q$ and $a \in \Sigma \cup \{\#\}$	$Q_s^{\mathsf{L}} x <:: L_a N Q_{s'}^{\mathsf{wL}} L_b N M^{\mathsf{R}} N x$	for $\delta(q_s, a) = (q_{s'}, b, \mathsf{R})$
$Q_s^{\mathsf{wL}} x <:: E Q_s^{\mathsf{LR}} N x$	for $q_s \in Q \setminus \{q_{\mathsf{H}}\}$	$Q_s^{\mathsf{L}} x <:: L_\# N Q_{s'}^{\mathsf{wL}} L_\# N M^{\mathsf{L}} N L_b N x$	for $\delta(q_s, \bot) = (q_{s'}, b, \mathsf{L})$
$Q_{\mathsf{H}}^{\mathsf{wL}} x <:: E E Z$		$Q_s^{\mathsf{L}} x <:: L_\# N Q_{s'}^{\mathsf{wL}} L_\# N M^{\mathsf{R}} N L_b N x$	for $\delta(q_s, \bot) = (q_{s'}, b, \mathsf{S})$
$E x <:: Q_s^{\mathsf{LR}} N Q_s^{\mathsf{wR}} E E x$	for $q_s \in Q$	$Q_s^{\mathsf{L}} x <:: L_\# N Q_{s'}^{\mathsf{wL}} L_\# N L_b N M^{\mathsf{R}} N x$	for $\delta(q_s, \bot) = (q_{s'}, b, \mathsf{R})$

FIGURE 1: Une table de classe simulant une telle machine de Turing $\mathcal{T}$

L'existence d'une telle table de classe permet alors de prouver le **théorème 1**.

3 Chaînage de méthodes

On va à présent prouver **le théorème 2** en montrant que l'ont peut implémenter un générateur de parseur pour des chaînages de méthodes.

Definition (Chaînage de méthodes)**.** Une désignation chaînée ou **chaînage de méthodes** (fluent pattern) consiste à agir en une seule instruction sur plusieurs méthodes du même objet, dans un but de plus grande lisibilité.

Par exemple, on utilisera $f().g().h()$ (chaînage) plutôt que $f(); g(); h();$ (séquentiel). On considérera que le chaînage de méthodes est plus lisible puisque l'ordre dans lequel sont appellées les fonctions est limité par le contrôleur de type. L'article propose

un constructeur de chaînage de méthode proposé par Gil et Levy 2016 :

Théorème 4 (Gil et Levy 2016)**.** Soit G une grammaire **déterministe** non contextuelle décrivant un langage $\mathcal{L} \in \Sigma^*$ dans un alphabet Σ de noms de méthodes. On peut construire des définitions d'une classe Java, un type T, et des expressions Start et Stop de telle sorte que le code

$$T\ell = Start.f^{(1)}().f^{(2)}()...f^{(m)}().Stop()$$

est correctement typé si et seulement si $f^{(1)}().f^{(2)}()...f^{(m)}() \in \mathcal{L}$.

Il s'agit du même théorème que le **théorème 2** à l'exception de la condition de déterminisme de la grammaire et de l'efficacité du contrôleur de type. On va pouvoir utiliser le **théorème 1** afin de montrer que ce déterminisme n'est en fait pas nécessaire. De plus, on garantira que le code généré soit polynomial en taille de grammaire, et enfin que le code puisse être typé en temps polynomial.

3.1 Classe constructrice

On va souhaiter avoir une classe constructrice déterminant les expressions Start et Stop, ainsi que toutes les autres méthodes $f^{(m)}()$. Pour cela, on reprend le typage présenté dans l'article :

$$\overbrace{\underbrace{ZEENL_{\#}NM^{\mathsf{L}}}_{Start}\underbrace{NL_{f^{(1)}}}_{f^{(1)}()}\ldots\underbrace{NL_{f^{(m)}}}_{f^{(m)}()}\underbrace{NL_{\#}Q_{\mathsf{I}}^{\mathsf{wR}}}_{Stop}}^{S} \blacktriangleleft \overbrace{EEZ}^{T}$$

Où les types suivant sont notés :

$$S := Q_{\mathsf{I}}^{\mathsf{wR}}L_{\#}NL_{f^{(m)}}N\ldots L_{f^{(1)}}NM^{\mathsf{L}}NL_{\#}NT$$
$$T := EEZ$$

Ici, $Start.f^{(1)}().f^{(2)}()...f^{(m)}().Stop()$ construit le type S. On va alors pouvoir implémenter, à l'aide des classes génériques de Java, une classe abstraite B nous permettant de construire très facilement le type S :

```
abstract class B<x> {
  static B<ML<?super N<?super
           Lhash<?super N<?super
           E<?super E<?super Z>>>>>>> start;
  abstract QWRstart<?super
           Lhash<?super N<?super x>>> stop();
  abstract B<La<?super N<?super x>>> a();
  abstract B<Lb<?super N<?super x>>> b();
  abstract B<Lc<?super N<?super x>>> c();
}
```

FIGURE 2: Classe constructrice B.

Cette classe a pour alphabet l'ensemble de ses méthodes à l'exception de Start et Stop (i.e $\Sigma = \{a, b, c\}$). Si l'on souhaite vérifier que le mot $abca$ est dans le language considéré, on pourra par exemple écrire

```
E<?super E<?super Z>> l = B.start.a().b().c().a().stop();
```

3.2 Grammaires et arbres de dérivation

Quelques notions élementaires liées aux grammaires vont maintenant être introduites avant de poursuivre. Soit Σ un **alphabet** fini composé de terminaux ou de lettres. Soit T l'ensemble des non terminaux. Par convention, on dénote les lettres en majuscule pour les **non terminaux**, et en minuscule pour les **terminaux**. On note également ϵ le **symbole vide**. Une **production** est une relation $A \rightarrow t_1, t_2...t_n$ et si A est un non terminal alors $t_1t_2...t_n$ est une chaîne de caractères. Une **grammaire** G est un ensemble de productions ayant comme élément de départ un non terminal S. Les dérivations qui conduisent de l'axiome à un mot terminal peuvent se regrouper en arbres, appelés **arbres de dérivation**. Chaque règle de grammaire peut être vue comme un nœud étiqueté par le non-terminal de la règle dont les fils sont étiquetés par les lettres du membre droit.

3.3 L'algorithme CYK

L'algorithme de CYK permet de resoudre le *membership problem* pour une chaîne de caractère s et une grammaire G données. CYK est un algorithme fortement récursif, il suit le principe suivant :

Soit la relation cyk definie inductivement $cyk(a, b)$.

(a) Si a est un non terminal, alors on continue avec $cyk(a, a)$.

(b) Si $A \to t_1, ...t_n$ est une production, a est une concaténation $a_1...a_n$, et $cyk(a_k, t_k)$ continue $\forall k \in \{1, ..., n\}$, alors $cyk(a, A)$.

Il y a plusieurs façon d'implémenter un tel algorithme. Mais l'approche évoquée est celle utilisant la programmation dynamique. Cet algorithme a une complexité assez grande en espace et temps.

3.4 Le language Simper

L'article introduit un nouveau language simple et impératif qu'il nomme **Simper**. Le but ici est d'implémenter les parseurs CYK dans ce language. On donne au language Simper la syntaxe suivante :

$$
\begin{array}{lll}
s \to \ell{:} \mid \underline{\mathsf{goto}}\,\ell \mid l \underline{:=} v \mid \underline{\mathsf{if}}\, c\, \underline{\{}s^*\underline{\}}\, \big(\underline{\mathsf{else}}\, \underline{\{}s^*\underline{\}}\big)^? \mid \underline{++}l \mid \underline{--}l \mid \underline{\mathsf{halt}} & & \text{statements (core)} \\
s \to \underline{\mathsf{while}}\, c\, \underline{\{}s^*\underline{\}} \mid \underline{\mathsf{switch}}\, v\, \underline{\{}\big(v\, \underline{\{}s^*\underline{\}}\big)^*\underline{\}} & & \text{statements (sugar)} \\
v \to l \mid r & & \text{values} \\
l \to id\big(\underline{[}v\big(\underline{,}v\big)^*\underline{]}\big)^? & & \text{left values} \\
r \to nat \mid \underline{\text{“}string\text{”}} \mid \underline{\mathsf{array}}\underline{[}v\big(\underline{,}v\big)^*\underline{]}\big(v\big) & & \text{right values} \\
c \to c\, \underline{\&\&}\, c \mid c\, \underline{||}\, c \mid v \underline{==} v \mid v \underline{!=} v & & \text{conditions}
\end{array}
$$

FIGURE 3: La syntaxe de Simper donnée par la grammaire suivante

Afin de simplifier l'écriture des parseurs, on a ajouté au language des tableaux nommés de taille arbitraire, un petit système de type, et du sucre syntaxique afin de rendre le tout plus lisible[2]. Il a été aussi évité d'ajouter la gestion des procédures et des opérations arithmétiques afin de faciliter la traduction vers une machine de Turing. Le système de type de Simper est le suivant :

$$
\frac{l : t \qquad v : t}{l := v \ : \ \text{unit}} \qquad
\frac{c : \text{bool} \qquad b_1 : \text{unit} \qquad b_2 : \text{unit}}{\text{if } c \,\{b_1\}\, \text{else}\, \{b_2\} \ : \ \text{unit}} \qquad
\frac{l : \text{nat}}{{+}{+}l \ : \ \text{unit}} \qquad
\frac{}{\mathsf{halt} \ : \ \text{unit}}
$$

$$
\frac{c_1 : \text{bool} \qquad c_2 : \text{bool}}{c_1 \,\&\&\, c_2 \ : \ \text{bool}} \qquad
\frac{x : \text{array}\, n\, t \qquad e_1 : \text{nat} \quad \ldots \quad e_n : \text{nat}}{x[v_1, \ldots, v_n] \ : \ t} \qquad
\frac{s_1 : \text{unit} \quad \ldots \quad s_n : \text{unit}}{s_1 \ldots s_n \ : \ \text{unit}} \qquad
\frac{}{0 \ : \ \text{nat}}
$$

$$
\frac{v_1 : t \qquad v_2 : t}{v_1 == v_2 \ : \ \text{bool}} \qquad
\frac{v' : t \qquad v_1 : \text{nat} \quad \ldots \quad v_n : \text{nat}}{\mathsf{array}[v_1, \ldots, v_n](v') \ : \ \text{array}\, n\, t} \qquad
\frac{}{x \ : \ t} \qquad
\frac{}{\text{“}\texttt{foo}\text{”} \ : \ \text{sym}}
$$

FIGURE 4: L'ensemble des règles de typage de Simper

A l'aide de la syntaxe et du système de type de Simper, l'algorithme CYK a pu être retranscrit dans ce language avec cependant quelques petits ajouts dû au manque d'opérations arithmétiques possibles dans ce language, comblés par des **variables auxiliaires**.

```
sn := n   ++sn   T := array[sn,sn,15](0)
i := 0  si := 1  while i != n {
  switch input[i] {
    "a" { T[i,si,11] := 1 }
    "b" { T[i,si,12] := 1 }
    "c" { T[i,si,13] := 1 }
    "d" { T[i,si,14] := 1 }
  }
  ++i   ++si
}

k := 2  while k != sn {
  i := 0  ik := k  while ik != sn {
    j := i  ++j  while j != ik {
      // for Y → EG
      if T[i,j,5] == 1 && T[j,ik,9] == 1 {
        T[i,ik,3] := 1
      }
      ... eight other binary productions ...
      ++j
    }
    ++i   ++ik
  }
  i := 0  ik := k  while ik != sn {
    j := 0  while j != 11 { // 11 nonterminals
      // for S → X
      if T[i,ik,1] == 1 { T[i,ik,0] := 1 }
      ... seven other unary productions ...
      ++j
    }
    ++i   ++ik
  }
}
if T[0,n,0] == 1 { halt }
```

FIGURE 5: L'algorithme CYK retranscrit dans le language Simper

Ces variables vont rendre l'algorithme CYK plus compliqué à lire, mais heureusement elles n'influenceront pas l'éfficacité de l'algorithme en temps et en taille.

4 Un compilateur Java comme interpréteur

Grâce à notre petit algorithme CYK rédigé en Simper, on va chercher à montrer dans cette dernière partie comment un contrôleur de type de Java peut être utilisé comme **un interpréteur pour n'importe quel programme de Simper**. On va ici décrire la compilation d'un programme Simper en code Java. Le code Java typera correctemment si et seulement si le programme Simper donné atteint l'état d'arrêt. La compilation va se faire en deux phases :

1. Une traduction d'un programme Simper vers une machine de Turing.

2. Une traduction de machine de Turing vers du code Java.

Cette partie va également (enfin) donner la preuve du **théorème 2**.

4.1 Machines de Turing étendues

Nous avons besoin dans cette partie d'introduire la notion de machines de Turing étendues. Cela est du à l'existence du type nat dans le language Simper, puisque celui-ci autorise des entiers positifs de grande taille dont on ne peut pas connaitre le nombre de chiffres à l'avance. On aura donc parfois besoin d'étendre l'espace nécessaire pour stocker un nombre nat. Au lieu de procéder à un décalage du contenu du ruban, on va autoriser les machines de Turing étendues à pouvoir insérer des symboles comme opération primitive. L'article compare alors la définition (en tant qu'automate) d'une machine de Turing et d'une machine de Turing étendue : la seule différence se fait au niveau de la fonction de transition où le symbole courrant appartient à Σ^* une fois étendue au lieu de Σ. Une machine étendue pourra alors avoir symbole courant une lettre de l'alphabet ou bien le symbole vide. Cela nous amène alors à la proposition suivante :

Proposition. Il est possible de convertir une machine de Turing et une machine de Turing étendue avec une augmentation polynomiale en taille, mais tout en préservant le temps d'excécution en temps polynomial.

4.2 D'un programme à une machine de Turing

Dans cette partie, l'auteur de l'article montre comment la bande de Turing est organisée[3], u'elle est l'idée principale pour la transformation en machine de Turing, et comment le compilateur est organisé. Le mécanisme d'une bande est la suivante : il y a 2 jetons sym et nat, l'un est borné l'autre non. On peut effectuer des opérations comme des comparaisons mais pas de propriétés d'additions, par exemple, pour former un nouvel entier. Chaque valeur donnée par les jetons sont une valeur de l'alphabet de base. De même, chaque variable est stockée dans une bande à part bornée, c'est le même mécanisme pour les listes où chaque élements a un délimiteur. Pour ce qui est de la conversion, il suffit de prendre le graphe symbolique du programme et d'en convertir chaque arc de noeud à un état dans une machine de Turing (ce qui donnerait un ensemble d'états). Nous pouvons parler de l'exemple d'une incrémentation, $l++$, où la machine pointe vers la bande l ,et incrémente une variable 11..10 en 00..01. Pour le cas d'une affectation, par exemple $l := m$, la machine de Turing pointe vers l puis la marque d'un $\perp$, et affecte dans cette bande la valeur de r. Selon de ce qu'est r, elle est stockée dans une zone fixée de la bande (celle-ci est ensuite marquée par un T).

Pour ce qui est du mécanisme du compilateur, on peut noter que celui-ci est fait en Ocaml, et suis un schéma traditionel quant au mode de fonctionnement d'un compilateur. Des fonctions de transformation sont cela dit utiles.

4.3 D'une machine de Turing à du code Java

On passe maintenant à l'étape de compilation de machines de Turing en code Java. On rappelle que l'on va utiliser dans cette étape la notion de machine de Turing étendue. Pour cela, une nouvelle table de classe pour une machine étendue a été proposée en modifiant quelques règles de la table de classe que l'on avait proposé précédement pour une machine de Turing simple. De plus, on a aussi été capable d'adapter le **théorème 1** aux machines de Turing étendues. La convertion devient alors triviale pour les machines de Turing étendues.

On peut alors bien compiler des programmes Simper en code Java, et un programme Simper s'arrêtera si et seulement si le code Java est correctement typé, pour n'importe quelle entrée. Pour le code java, l'entrée sera produite à l'aide d'une classe constructrice.

4.4 Efficacité

Cette dernière partie va compléter la preuve du **théorème 2** en discutant de l'éfficacité d'une telle compilation en terme de taille, et en terme de temps de contrôle du typage. En thérorie, c'est efficace, mais en pratique, non.

En théorie, L'utilisation de l'algorithme CYK combiné aux machine sous-typées se fait en temps polynomial avec malgré tout un degré polynomial relativement élevé (9). Il est encore possible d'améliorer les performances de ce coté là, mais l'auteur ne propose aucune piste.

En pratique cependant, le code généré en Java est de l'ordre de 1GiB, ce qui est de taille beaucoup trop grande pour le compilateur **javac** puisque le temps de compilation d'un tel code ne sera pas raisonnable. L'auteur de l'article propose cependant des alternatives afin de contourner se problème, à savoir de fabriquer à la main une machine de Turing reconnaissant un language ambigue afin de réduire la taille du code généré en Java. Il propose également de rajouter dans la syntaxe de Simper une opération primitive "**goto**" afin d'améliorer les performances de l'étape de compilation d'un programme Simper en machine de Turing.

5 Conclusion

On a montré que le contrôleur de type de Java peut être utilisé comme un interpréteur pour n'importe quel programme de Simper en passant par une traduction en machine de Turing. Le contrôleur de type de Java est donc indécidable grâce à la réduction au problème d'arrêt des machines de Turing. On peut alors également dire que le contrôleur de type de Java est **Turing-complet**, et donc qu'il a une puissance de calcul au moins équivalentes aux machines de Turing.

6 Références

1. Kennedy and Pierce On decidability of nominal subtyping with variance. In FOOL, 2007.
2. S. Wehr and P. Thiemann. On the decidability of subtyping with bounded existential types. In APLAS, 2009.
3. A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. J. of Math, 1936.
4. Radu Grigore Java generics are turing complete.2017.