Compilation
Fiche de Révisions:
P-Machine

10 avril 2018

1 Calcul de l'arbre sémantique

1.1 Package 1&2 : variables globales et typage

1.1.1 Phase descendante : récupérer/renomer les variables globales et leur type

Le plus symple est de faire plusieurs passes (le plus rapide est de tout faire en même temps, mais c'est pas facile) :

— Tout d'abord on descend dans l'arbre en maintenant deux listes : la liste $V :: [String * Type]$ des variables croisées jusque là et la liste $R :: [String * String]$ des remplacement à faire :

$$
\begin{array}{rll}
\langle \texttt{Var}\ x : T\rangle & \mapsto V := (x,T) : V; & \text{si } x \notin dom(V) \\
& \quad \texttt{return}\ (\texttt{Var}\ x : T) & \\
& \mapsto V := (z,T) : V; & \text{si } x \in dom(V) \text{ et } z \notin V \\
& \quad R := (x,z) : R; & \\
& \quad \texttt{return}\ (\texttt{Var}\ z : T) & \\
\langle < \texttt{VARIABLE} > (x)\rangle & \mapsto \texttt{return} < \texttt{VARIABLE} > (x) & \text{si } (x, \_) \notin dom(R) \\
& \mapsto \texttt{return} < \texttt{VARIABLE} > (R(x)) & \text{si } x \in dom(R) \\
\langle c_1; c_2\rangle & \mapsto \texttt{return}\ (\langle c_1\rangle; \langle c_2\rangle) & \\
\langle \texttt{begin}\ p\ \texttt{end}\rangle & \mapsto \texttt{return}\ (\texttt{begin}\langle p\rangle\texttt{end}) & \\
\langle \texttt{var}\ x : \texttt{Array}\ [e] of T\rangle & \mapsto e' := \langle e\rangle\ \texttt{dans} & \text{si } x \notin dom(V) \\
& \quad V := (x, List(T)) : V; & \\
& \quad \texttt{return}\ (\texttt{Var}\ x : \texttt{Array}\ [e'] of T) & \\
& \mapsto e' := \langle e\rangle\ \texttt{dans} & \text{si } x \in dom(V) \text{ et } z \notin V \\
& \quad V := (z,T) : V; & \\
& \quad R := (x,z) : R; & \\
& \quad \texttt{return}\ (\texttt{Var}\ z : \texttt{Array}\ [e'] of T) & \\
& \ldots\ldots &
\end{array}
$$

où on note :

$$
\begin{array}{rll}
dom([(x_1,z_1);...;(x_n,z_n)]) := \{x_1,...,x_n\}, & & \\
\quad [(x_1,z_1);...;(x_n,z_n)](x_i) := z_i & \text{si pour tout } j < i, x_j \neq x_i. &
\end{array}
$$

— Puis on remonte les expressions avec une fonction de typage :

$$
\begin{array}{rll}
Type(x) := V(x) & & \\
Type(< \texttt{NUM} >) := \texttt{Integer} & & \\
Type(< \texttt{BOOL} >) := \texttt{Boolean} & & \\
Type(e_1 + e_2) := T & \text{si } Type(e_1) = Type(e_2) = T \in \{\texttt{Int}, \texttt{Real}, \texttt{String}\} & \\
\mapsto \texttt{erreur} & & \text{sinon} \\
Type(e_1 \&\& e_2) := \texttt{Boolean} & \text{si } Type(e_1) = Type(e_2) = \texttt{Boolean} & \\
\mapsto \texttt{erreur} & & \text{sinon} \\
\ldots & &
\end{array}
$$

Il faut en profiter pour vérifier les conditionelles et les tableaux :
— Sur $\texttt{If}\ e\ \texttt{Then}\ c_1\ \texttt{Else}\ c_2$, is faut vérifier que $Type(e) = \texttt{Boolean}$,
— Sur $\texttt{While}\ e\ \texttt{Do}\ c$, is faut vérifier que $Type(e) = \texttt{Boolean}$,
— Sur $t[e]$, is faut vérifier que $Type(e) = \texttt{Integer}$,
— Sur $\texttt{Array}\ [e] of T$, is faut vérifier que $Type(e) = \texttt{Integer}$,

1.2 Package 3 : variables locales

Tout d'abord, il faut faire les étapes précédentes, mais en incluant les fonctions : on fait en sorte qu'il n'y ai pas deux fonctions/procédures/variables de même noms ; puis on type en vérifiant les types d'entré et de sortie des fonctions.

Les variables locales (les arguments des fonctions et les variables déclarées dans le corps de la fonction) sont difficile à traiter : elles peuvent être introduite plusieurs fois sans que l'on puisse écraser la précédente.

Pour éviter ca, on va utiliser des notations à la De Bruijn : les variables $<$ `VARIABLE` $>(x)$ locales sont remplacées par `Local(x,i,j)` où

- i est un indice représentant la fonction/procédure où est introduite la variable,
- j est un indice représentant le rang d'introduction de la variable dans la fonction/procédure en question.

On utilise pour ca un mapping $B :: [String * Int * Int]$ et on applique la procédure :

$$
\begin{array}{rll}
\langle <\texttt{VARIABLE}>(x)\rangle & \mapsto \texttt{return} <\texttt{VARIABLE}>(x) & \text{si } x \notin dom(B) \\
\langle <\texttt{VARIABLE}>(x)\rangle & \mapsto \texttt{return}\ Local(x,i,j) & \text{si } (x,i,j) \in B \\
\langle \texttt{Var}\ x : t;\rangle & \mapsto \texttt{return}\ (\texttt{Var}\ x : t); & \text{si } x \text{ est globale} \\
\langle \texttt{Var}\ x : t;\rangle & \mapsto B := (x, 0, max(B)+1) : B; & \text{si } x \text{ est locale} \\
 & \quad \texttt{return}\ (\texttt{VarLoc}\ x : \langle t\rangle); & \\
\langle \texttt{Procedure}\ f(x_1 : T_1, ..., x_n : T_n); & & \\
d; \texttt{Begin}\ p\ \texttt{End};\rangle & \mapsto B' := copie(B)\ \texttt{dans} & \\
 & \quad B := incrB & \\
 & \quad B := (x_1, 0, 1) : \cdots : (x_n, 0, n) : B; & \\
 & \quad d' := \langle d\rangle\ \texttt{dans} & \\
 & \quad p' := \langle p\rangle\ \texttt{dans} & \\
 & \quad B := B' & \\
 & \quad \texttt{return}\ (\texttt{Procedure}\ f(x_1 : T_1, ..., x_n : T_n) & \\
 & \qquad d';\ \texttt{Begin}\ p'\ \texttt{End}) & \\
 & \ldots\ldots &
\end{array}
$$

où max est le rang maximum pour la fonction en cours (d'indice 0) et $incr$ va incrémenter le i de tous les triplets :

$$
\begin{aligned}
max[(x_1,i_1,j_1)...(x_n,i_n,j_n)] &:= max(0, max\{j_k \mid i_k = 0\}) \\
incr[(x_1,i_1,j_1)...(x_n,i_n,j_n)] &:= [(x_1,i_1+1,j_1)...(x_n,i_n+1,j_n)]
\end{aligned}
$$

2 La P-machine

2.1 Sa sémentique

2.1.1 Package 1

Deux bandes mémoire indépendantes :

— Une bande de lecture `CODE` dont la taille `codemax` est arbiraire (elle fait la taille du code donné...). Chaque case contien une instruction de P-code.

— Une bande de calcul[1] `RAM` de taille fixée `max`. Chaque case est typé et contien un entier `(i)`, un booléen `(b)`, un flotant `(r)` ou une adresse `(a)`. Attention, il peut (et il va) y avoir des cases de types diférents sur la même bande.

Trois zones délimités par deux pointeurs dans la RAM :

— On définit un pointeur `SP` pour *Stack Pointer*.

— La partie initial `RAM[0..SP]` de la RAM est la pile, ou `STACK`. Il s'agit d'une pile/fifo/liste, c'est à dire que l'on a un accès qu'au sommet de la pile.

— On définit un pointeur `HP` pour *Heap Pointer*. Pour le Package 1, ce pointeur fixé et est choisi statiquement : c'est `max` moins le nombre de variables dans le programme.

— La partie final `RAM[HP..max]` de la RAM est le tas ou `HEAP`. Il s'agit de la zonne où sont stoquées les valeurs mutables.

— La partie central `RAM[SP+1..HP+1]` de la RAM est la mémoir libre. On ne sais pas ce qu'il y a dedans et on n'y accède jamais (à moins de faire grossir la pile).

Plus un pointeur `PC` sur le code.

La machine lit une instruction sur `CODE[PC]` et l'exécute, ce qui peut changer `SP`, `HP`, `PC` et `RAM`, mais pas `CODE`. Une fois l'instruction effectuée, elle continue jusquà ce que `PC` ne sorte de `CODE`.

Si à n'importe quel moment `PC` ou `SP` est négatif, on crash ; mais ca ne devraient pas pouvoir arriver.

Par contre, si `HP` devient inférieur à `SP` on crash et celà poura arriver : il s'agit d'un *stack overflow*.

Dans les instructions suivantes, il y a des conditions de la forme $q \in [\texttt{HP+1}, \texttt{max}]$; si elles ne sont pas vérifiées, la machine crash. Il s'agit d'un *seqfault*.

2.1.2 Package 2

Un seul changement : le pointeur `HP` n'est plus statique : il va diminuer lorsque l'on alloue dynamiquement un tableau ou un pointeur.

Attention, normalement le pointeur `HP` peut réaugmenter lors de la libération de pointeur, mais on n'en parle pas ici. Celà veut dire que notre machine a automatiquement des fuites de mémoire et doit être redémarer lorsque la RAM est pleine...

1. Appelée STORE dans *Compiler Design*

2.1.3 Package 3

On dispose ici d'un nouveau pointeur : `MP` qui pointe le moment où l'on a appelé la dernière procédure/fonction.

2.2 Les instructions de P-code

2.2.1 Package 1

Dans le tableau suivant, `T` est une variable de type, il désigne `(i)`, `(b)`, `(r)` ou `(a)`. Tandit que `N` est une variable de type pour les numériques, il peut représenter `(i)`, `(r)` ou `(a)`.

Instruction	sémentique	type du sommet de pile	type du résultat	autre condition
and	RAM[SP-1] := RAM[SP-1] and RAM[SP]; SP-- PC++	(b,b)	(b)	
or	RAM[SP-1] := RAM[SP-1] or RAM[SP]; SP-- PC++	(b,b)	(b)	
not	RAM[SP] := not RAM[SP]; PC++	(b)	(b)	
equ T	RAM[SP-1] := RAM[SP-1] $=_T$ RAM[SP]; SP-- PC++	(T,T)	(b)	
geq T	RAM[SP-1] := RAM[SP-1] $\geq_T$ RAM[SP]; SP-- PC++	(T,T)	(b)	
leq T	RAM[SP-1] := RAM[SP-1] $\leq_T$ RAM[SP]; SP-- PC++	(T,T)	(b)	
ges T	RAM[SP-1] := RAM[SP-1] $>_T$ RAM[SP]; SP-- PC++	(T,T)	(b)	
les T	RAM[SP-1] := RAM[SP-1] $<_T$ RAM[SP]; SP-- PC++	(T,T)	(b)	
neq T	RAM[SP-1] := RAM[SP-1] $\neq_T$ RAM[SP]; SP-- PC++	(T,T)	(b)	

Instr-uction	sémentique	type du sommet de pile	type du résultat	autre condition
add N	RAM[SP-1] := RAM[SP-1] $+_{\mathtt{N}}$ RAM[SP]; SP-- PC++	(N,N)	(N)	
sub N	RAM[SP-1] := RAM[SP-1] $-_{\mathtt{N}}$ RAM[SP]; SP-- PC++	(N,N)	(N)	
mul N	RAM[SP-1] := RAM[SP-1] $*_{\mathtt{N}}$ RAM[SP]; SP-- PC++	(N,N)	(N)	
div N	RAM[SP-1] := RAM[SP-1] $/_{\mathtt{N}}$ RAM[SP]; SP-- PC++	(N,N)	(N)	
neg N	RAM[SP] := $-$RAM[SP]; PC++	(N)	(N)	
ido T q	SP++; RAM[SP] := RAM[q] PC++		(T)	$q \in$ [HP+1, max]
idc T q	SP++; RAM[SP] := q PC++		(T)	$Type(q) =$ T
sro T q	RAM[q]:= RAM[SP] SP--; PC++	(T)		$q \in$ [HP+1, max]
ujp q	PC := q			$q \in [0,$ codemax]
fjp q	if RAM[SP] then PC++ else PC := q			$q \in [0,$ codemax]
wrt T	affiche RAM[SP] sur le terminal ; SP--;	(T)		

2.2.2 Package 2

Instr-uction	sémentique	type du sommet de pile	type du résultat	autre condition
ixa q	RAM[SP-1] := RAM[SP-1] + q∗RAM[SP]; SP--; PC++	(a,i)	(a)	q∗RAM[SP] ≤ RAM[RAM[SP-1]] RAM[SP] > 0
dpl T	SP++ RAM[SP] := RAM[SP-1] PC++	(T)	(T,T)	
ddp T	SP++ RAM[SP] := RAM[SP-2] PC++	(T,T')	(T,T',T)	
ind T	RAM[SP] := RAM[RAM[SP]] PC++	(a)	(T)	$Type$(RAM[SP]) = T
sto T	RAM[RAM[SP-1]]:= RAM[SP] SP := SP−2; PC++	(a)	(a,T)	
new	HP := HP − RAM[SP]; RAM[RAM[SP-1]] := HP SP := SP−2; PC++	(a,i)		
nwt	HP := HP − RAM[SP] − 1; RAM[HP] := RAM[SP]; RAM[RAM[SP-1]] := HP SP := SP−2; PC++	(a,i)		

2.2.3 Package 3

Dans le tableau suivant, on utilise $base(p, a)$ définit par :
$base(0, a) := a$
$base(p + 1, a) := base(p, \texttt{RAM}[a + 1])$
il s'agit d'un déréférencement successif pour remonter à la déclaration d'une variable.

Instr-uction	sémentique	commentaires
iod $T\ p\ q$	SP++; RAM[SP] := RAM[$base(p,$ MP$) + q$]; PC++;	
ida $p\ q$	SP++; RAM[SP] :=$base(p,$ MP$) + q$; PC++;	
str $T\ p\ q$	RAM[$base(p,$ MP$) + q$]:= RAM[SP]; SP−−;	
mst p	RAM[SP+2] :=$base(p,$ MP$)$; RAM[SP+3] := MP SP := SP+4; PC++	[2]
cup $p\ q$	MP := SP $-\ (p+3)$ RAM[MP+3] := PC PC :=q	
retp	SP := MP-1 PC := RAM[MP + 3] MP := RAM[MP + 2]	
retf	RAM[MP] := RAM[SP] SP := MP PC := RAM[MP + 3] MP := RAM[MP + 2]	

3 L'encodage

3.1 Environement

3.1.1 Package 1

S'il y a n variables définies dans un programme, on réserve les n dernières adresses de CODE et on fixe HP = codemax $- n$. On assicie aussi à chaque nom de variable x, son type Type(x) et une adresse Adresse$(x) \in$ [HP + 1, codemax] parmis ces n cases réservées.

3.1.2 Package 2

Dans le package 2, la taille des structures à réserver sur le tas n'est pas connue car on ne connait pas statiquement la taille des tableaux et strings. Du coup, il va faloir réserver dynamiquement de la place pour les tableaux.

2. Dans la machine historique, vous trouverez un pointeur en plus à stoquer : EP. Il est utilisé pour minimiser le nombre de tests à faire pour vérifier que la machine n'est pas en bout de la RAM. Ici on simplifie

3.1.3 Package 3

On commence par (α-)renomer les fonctions/procédures dont les noms apparaissent plusieurs fois (dans différent scopes ou avec l'une qui cache l'autre). De même avec les variables locales.

Pour chaque fonction/procédure p, on récupère, en fin de compilation, la ligne l_p où elle est introduite. En atendant on utilise l_p comme une variable.

3.2 Règles

3.2.1 Package 1

Expressions

On rappel que l'on travail maintenant avec l'AST du programme et non le string d'entré. Par exemple, si on écrit $e_1 \neq e_2$, celà fait référence à l'AST de racine $\neq$ et avec comme fils les arbres e_1 et e_2.

Étant donné un AST a et un environement , on définit le codage $[\![a]\!]$ comme suit

$$
\begin{array}{lll}
[\![e_1 =_T e_2]\!] := [\![e_1]\!]; & [\![e_1 +_{\texttt{Int}} e_2]\!] := [\![e_1]\!]; & [\![e_1 +_{\texttt{Real}} e_2]\!] := [\![e_1]\!]; \\
\quad [\![e_2]\!]; & \quad [\![e_2]\!]; & \quad [\![e_2]\!]; \\
\quad \textbf{equ}\ T; & \quad \textbf{add}\ i; & \quad \textbf{add}\ r; \\
[\![e_1 \neq_T e_2]\!] := [\![e_1]\!]; & [\![e_1 -_{\texttt{Int}} e_2]\!] := [\![e_1]\!]; & [\![e_1 -_{\texttt{Real}} e_2]\!] := [\![e_1]\!]; \\
\quad [\![e_2]\!]; & \quad [\![e_2]\!]; & \quad [\![e_2]\!]; \\
\quad \textbf{neq}\ T; & \quad \textbf{sub}\ i; & \quad \textbf{sub}\ r; \\
\quad \ldots & \quad \ldots & \quad \ldots
\end{array}
$$

$$
[\![\texttt{<VAR>}(x)]\!] := \textbf{ido}\ \texttt{Type}(x)\ \texttt{Adresse}(x);
$$

Commandes

$$
\begin{array}{ll}
[\![x := e]\!] := & [\![e_1]\!]; \\
& \textbf{sro}\ \texttt{Type}(x)\ \texttt{Adresse}(x);
\end{array}
$$

$$
\begin{array}{ll}
[\![\texttt{If}\ e\ \texttt{Then}\ c_1 \texttt{Else}\ c_2]\!] := & [\![e]\!]; \\
& \textbf{fjp}\ l_1; \\
& [\![c_1]\!]; \\
& \textbf{ujp}\ (l_2 + 1); \\
& \begin{array}{l} l_1: \\ l_2: \end{array} [\![c_2]\!];
\end{array}
$$

où l_1 est la ligne où commence $[\![c_2]\!]$; et l_2 est la ligne où il finit.

$$
\begin{aligned}
[\![\texttt{While}\ e\ \texttt{Do}\ c]\!] \ := \ & {}^{l_1:}[\![e]\!]; \\
& \textbf{fjp}\ (l_2+1); \\
& [\![c_1]\!]; \\
& l2 : \textbf{ujp}\ (l_1);
\end{aligned}
$$

où l_1 est la ligne où commence $[\![e]\!]$; et l_2 est la ligne de **ujp** (l_1);.

$$
\begin{aligned}
[\![\texttt{For}\ x := e_1\ \texttt{To}\ e_2\ \texttt{Do}\ c]\!] \ := \ & [\![e_1]\!]; \\
& \textbf{sro}\ \texttt{n}\ \texttt{Adresse}(x); \\
& {}^{l_1:}[\![e_2]\!]; \\
& \textbf{ido}\ \texttt{n}\ \texttt{Adresse}(x); \\
& \textbf{les}\ \texttt{n}; \\
& \textbf{fjp}\ (l_2+1); \\
& [\![c_1]\!]; \\
& \textbf{ido}\ \texttt{n}\ \texttt{Adresse}(x); \\
& \textbf{idc}\ \texttt{n}\ 1; \\
& \textbf{add}\ \texttt{n}; \\
& \textbf{sro}\ \texttt{n}\ \texttt{Adresse}(x); \\
& l2 : \textbf{ujp}\ (l_1);
\end{aligned}
$$

où $\texttt{Type}(x) = \texttt{n}$, où l_1 est la première ligne de $[\![e_2]\!]$; et l_2 est la ligne de **ujp** (l_1).

$$
\begin{aligned}
[\![\texttt{Writeln}_\texttt{T}(e)]\!] \ := \ & [\![e]\!]; \\
& \textbf{wrt}\ \texttt{T};
\end{aligned}
$$

$$
[\![\texttt{Begin}\ p\ \texttt{End}]\!] \ := [\![p]\!];
$$

Remarque : Les déclarations ne créent pas de code dans le package 1.

Programmes

$$
\begin{aligned}
[\![c_1; c_2; ...; c_n]\!] \ := \ & [\![c_1]\!]; \\
& [\![c_2]\!]; \\
& \cdots \\
& [\![c_n]\!];
\end{aligned}
$$

3.2.2 Package 2

Mutables

$$[\![x]\!] \ := \ \textbf{idc n } \texttt{Adresse}(x);$$

$$\begin{aligned}[\![m[e]]\!] \ := \ & [\![m]\!]; \\ & \textbf{ind a} \\ & [\![e]\!]; \\ & \textbf{idc n } 1 \\ & \textbf{add n}; \\ & \textbf{ixa } 1\end{aligned}$$

$$\begin{aligned}[\![m^{\wedge}]\!] \ := \ & [\![m]\!]; \\ & \textbf{ind a}\end{aligned}$$

Expression

$$[\![\texttt{Nil}]\!] \ := \ \textbf{idc a } 0;$$

$$\begin{aligned}[\![Mutable_T(m)]\!] \ := \ & [\![m]\!]; \\ & \textbf{ind T}\end{aligned}$$

Commande

$$\begin{aligned}[\![m := e]\!] \ := \ & [\![m]\!]; \\ & [\![e_1]\!]; \\ & \textbf{sto Type}(m);\end{aligned}$$

Pour t un type de tableau :

$$\begin{aligned}[\![\texttt{Var } x : t]\!] \ := \ & \textbf{idc aAdresse}(x); \\ & [\![t]\!];\end{aligned}$$

Types tableaux

pour T un type de base :

$$\begin{aligned}[\![\texttt{Array}[e]\ \texttt{of}\ T]\!] \ := \ & [\![e]\!]; \\ & \textbf{nwt};\end{aligned}$$

pour t un type de tableau :

$$\begin{aligned}[\![\texttt{Array}[e]\ \texttt{of}\ t]\!] \ := \ & \textbf{dpl}\ \texttt{a}; \\ & [\![e]\!]; \\ & \textbf{nwt}; \\ & \textbf{dpl}\ \texttt{a}; \\ & \textbf{ind}\ \texttt{T} \\ l_1 : \ & \textbf{dpl}\ \texttt{a}; \\ & \textbf{idc}\ \texttt{n}0; \\ & \textbf{equ}\ \texttt{n}; \\ & \textbf{fjp}\ (l_2 + 1) \\ & \textbf{ddp}\ \texttt{a}; \\ & \textbf{ddp}\ \texttt{n}; \\ & \textbf{ixa}\ 1 \\ & [\![t]\!]; \\ & \textbf{idc}\ \texttt{n}1; \\ & \textbf{sub}\ \texttt{n}; \\ l2 : \ & \textbf{ujp}\ l_1\end{aligned}$$

3.2.3 Package 3

Déclarations de procédures, fonctions et variables locales

$$\begin{aligned}& [\![\texttt{Procedure}\ p(x_1 : T_1, ..., x_n : T_n); d;\ \texttt{Begin}\ p\ \texttt{End};]\!] \\ := \quad & {}^{l_p}[\![d]\!]; \\ & [\![p]\!]; \\ & \textbf{retp};\end{aligned}$$

$$\begin{aligned}& [\![\texttt{Function}\ f(x_1 : T_1, ..., x_n : T_n) : T; d;\ \texttt{Begin}\ p\ \texttt{End};]\!] \\ := \quad & {}^{l_f}[\![d]\!]; \\ & [\![p]\!]; \\ & \textbf{retf};\end{aligned}$$

$$[\![\texttt{VarLocal}\ (x, i, j) : T;]\!] := \quad \textbf{ido}\ T\ 0;$$

Appels de procédures et variables

$$\begin{aligned}[\![\texttt{Appel}\ (p, i)(e_1, ..., e_n)]\!] := \quad & \textbf{mst}\ i; \\ & [\![e_1]\!]; \\ & \dots \\ & [\![e_n]\!]; \\ & \textbf{cup}\ n\ l_p;\end{aligned}$$

$$[\![\texttt{Local}\ (x, i, j)]\!] := \quad \textbf{iod}\ (Type(x))\ i\ (j + 4)$$

$$\begin{aligned}[\![\texttt{Local}\ (x, i, j) := e;]\!] := \quad & [\![e]\!]; \\ & \textbf{str}\ (Type(x))\ i\ (j + 4)\end{aligned}$$