DANIEL GODBOUT

CONVERSION DE PROGRAMMES DE L'IMPÉRATIF AU DÉCLARATIF

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise Maîtrise en informatique
pour l'obtention du grade de Maître ès sciences, (M.Sc.)

Faculté des sciences et de génie
UNIVERSITÉ LAVAL
QUÉBEC

2007

Résumé

Habituellement, plus le développement d'un logiciel est avancé plus il est dispendieux de le modifier. Par conséquent, une approche permettant de simplifier l'étape de maintenance permettrait de réduire considérablement le coût lié au développement de programmes. Le langage déclaratif de la méthodologie Lyee permet justement de simplifier la maintenance de programmes. Cependant, les programmes existants écrits dans un langage impératif doivent être traduits pour être utilisés par celle-ci. Ainsi, dans ce travail, nous proposons une fonction de conversion de programmes écrits dans un langage impératif avec des tableaux et entrées/sorties vers un langage déclaratif. Il s'agit d'une extension de langages $\mathcal{L}_1$ et $\mathcal{L}_2$ existants qui supportaient déjà les expressions arithmétiques et booléennes ainsi que les affectations, les boucles et les instructions conditionnelles. Le travail effectué a donc été d'ajouter les tableaux et les entrées/sorties dans ces langages et d'ajuster la fonction de traduction en conséquent. Aussi, une implantation d'une interface de développement permettant de spécifier des programmes dans un langage déclaratif simple à utiliser a été produite.

Remerciements

Avant tout, je tiens à remercier le professeur Béchir Ktari pour m'avoir dirigé dans cette recherche. Ses excellentes idées, son support et ses conseils ont été grandement appréciés. Ceux-ci ont non seulement permis de me guider dans ces travaux, mais aussi d'élargir mon champ de connaissances.

Je remercie aussi les professeurs Mohamed Mejri et Nadia Tawbi pour avoir accepté de prendre le temps nécessaire pour évaluer ce mémoire.

De plus, j'aimerais remercier tous les membres du LSFM et tous les autres étudiants que j'ai côtoyés pour leur agréable compagnie ainsi que leurs connaissances qu'ils ont partagées avec moi. Celles-ci m'ont permis d'apprendre beaucoup sur des sujets variés.

Finalement, je tiens à remercier Josée et ma famille d'avoir toujours été présentes pour moi. Leur soutien et leurs encouragements ont joué un rôle important dans la réalisation de ces travaux.

Table des matières

Liste des tableaux

Table des figures

Introduction

Contexte

L'étape de maintenance du cycle de vie des logiciels est extrêmement coûteuse. Puisqu'elle est nécessaire pour corriger des problèmes au niveau du logiciel ou pour lui ajouter des fonctionnalités, il serait très intéressant de pouvoir réduire le coût qui lui est associé.

Typiquement, l'étape de maintenance implique qu'un ou plusieurs programmeurs doivent relire presque l'intégralité du code pour identifier toutes les parties à modifier, effectuer les changements et s'assurer que le nouveau programme produit fonctionne selon les spécifications. Pour n'importe quel programme le moindrement complexe, cette tâche est relativement difficile. En effet, si les parties du code à modifier ainsi que celles qui en dépendent ne sont pas bien identifiées il est possible que le programme résultant présente certains problèmes. Pour compliquer le tout, l'étape de maintenance est souvent exécutée par des programmeurs qui ne faisaient pas partie de l'équipe de développement initiale et qui ne sont donc pas familiers avec le programme et toutes ses ramifications.

L'utilisation des langages impératifs est très répandue. Ces dernières ajoutent un niveau de complexité supplémentaire à la maintenance du code. En effet, pour un programme écrit dans un langage impératif, la définition d'une variable peut être influencée par différents bouts de code situés dans des endroits différents dans le programme. Comparativement, un langage comme celui présenté par la méthodologie Lyee [25, 26] pourrait simplifier la maintenance. En effet, un programme Lyee est une spécification de définitions de variables. Ainsi, la maintenance est facilitée puisque chaque variable est associée à une définition facile à identifier.

Puisqu'il existe une immense quantité de codes déjà écrits en langage impératif, une fonction de traduction permettant de les traduire vers un langage comme Lyee permettrait d'en simplifier la maintenance. Pour ce faire, les auteurs de [20, 21] ont

proposé une fonction permettant de traduire des programmes écrits dans un langage impératif simple vers un langage déclaratif. Certaines autres approches existent pour traduire un code source vers un autre langage [1, 2, 8, 14], mais leurs langages de destination sont très différents de celui de Lyee.

Objectif

Dans ce mémoire, nous allons étendre les langages impératifs et déclaratifs originalement proposés par [20, 21] de façon à ce qu'ils puissent utiliser des tableaux. La notion d'entrées et de sorties y sera aussi ajoutée. Par la suite, une fonction de traduction permettant de passer d'un langage à l'autre sera présentée.

De plus, une implantation d'une interface de développement (LyeeBuilder) permettant de spécifier simplement des programmes dans un langage déclaratif sera aussi présentée.

Structure du mémoire

Le présent mémoire est structuré de façon à présenter les notions préalables nécessaires avant d'en arriver avec l'approche proposée.

Ainsi, le chapitre 1 présente une approche de décompilation des tableaux et structures de données basée sur les types. Malgré que celle-ci passe d'un langage assembleur vers le langage C, elle est tout de même pertinente puisqu'elle permet de reconstruire des structures de données en passant d'un langage à l'autre.

Par la suite, le chapitre 2 présente l'approche originale permettant de traduire des programmes d'un langage impératif vers un langage déclaratif.

Ensuite, le chapitre 3 présente une approche pour effectuer des coupes de programmes en présence de tableaux. En effet, l'approche originale de traduction s'apparente à la coupe de programme. Ainsi, les problématiques rencontrées lors de la coupe avec les tableaux sont similaires à celles qui ont été rencontrées lors de la traduction avec les tableaux.

Le chapitre 4 présente l'approche proposée pour traduire les programmes d'un langage impératif contenant des tableaux et des entrées/sorties vers un langage déclaratif.

Finalement, le chapitre 5 montre l'implantation de l'interface de développement LyeeBuilder qui a été effectuée.

Chapitre 1

Décompilation basée sur les types

1.1 Introduction

Dans le but de réutiliser des programmes existant dans un contexte favorable à leur modification, il est nécessaire qu'ils soient dans une forme facile à maintenir. Lorsque le code source du programme en question n'est plus disponible, il est extrêmement difficile de maintenir le programme. Pour permettre la décompilation vers le langage source d'origine, certaines approches ont été proposées [3, 5, 23, 24]. La plupart de celles-ci utilisent des techniques différentes pour obtenir le résultat désiré. Par exemple, certaines sont basées sur le flot de contrôle ou d'autres sur la reconstruction des types. Dans ce chapitre, nous nous intéresserons à cette dernière puisqu'elle permet de partir de programmes en assembleur et de reconstruire les structures de données originales dans le langage C.

1.2 Contexte

Pour bien comprendre la technique utilisée pour la décompilation de programmes, certains éléments doivent être définis. Parmi ceux-ci, nous retrouvons les langages utilisés et la forme *Single Static Assignment* (SSA) [6].

1.2.1 Langages utilisés

Deux langages sont utilisés dans les exemples de décompilations, soit C et Register Transfer Language (RTL). RTL est utilisé comme langage source puisqu'il permet de représenter les exemples dans un langage indépendant d'une architecture particulière. Il est supposé que RTL peut effectuer des accès à la mémoire de 8, 16, 32 et 64 bits. Aussi, une pile est utilisée grâce aux instructions *push* et *pop* pour allouer les variables temporaires et locales. Un exemple simple de programme écrit en RTL est présenté dans le tableau 1.1.

```
1  f:
2          ld.w    4[r0],r0        ;Copie les 32 bits suivants
3                                  ;les 4 premiers octets de
4                                  ;l'adresse dans r0
5          xor     r0, r1, r1      ;Effectue un ou exclusif de
6                                  ;r0 avec r1 et mets le
7                                  ;résultat dans r1
8  L1F1:
9          mov     #4 r0           ;Mets la valeur 4 dans r0
10         add     r0,r1,r2        ;Additionne r0 et r1 et mets
11                                 ;le résultat dans r2
12         cmp     #10, r2         ;Compare la valeur 10 et r2
13         bneq    L1F1            ;Saute à l'étiquette L1F1 si
14                                 ;le résultat de la
15                                 ;comparaison est négatif
16 L2F1:
17         ret                     ;Retourne le contrôle à
18                                 ;l'appelant
```

Tab. 1.1 – Exemple de programme écrit en RTL.

La cible de la décompilation est le langage C ANSI [17]. Les types correspondant aux différents accès mémoires précédemment énoncés sont les suivants : *char*, *short*, *int* et *long*. Leur forme *unsigned* est utilisée à moins que la forme signée soit demandée explicitement par l'utilisateur ou par l'utilisation d'instructions non signées. Les pointeurs quant à eux sont différenciés des *int* par leur utilisation.

1.2.2 SSA

Puisque les compilateurs effectuent des optimisations de l'usage des registres [4] qui permettent d'utiliser le même registre pour des variables de types différents si elles ont une durée de vie disjointe, il est nécessaire d'utiliser une méthode permettant de différencier les variables. Pour ce faire, la forme *Single Static Assignment* est utilisée. En effet, elle permet de remplacer du code pour lequel la même variable subit plusieurs affectations par du code pour lequel chaque variable apparaît comme destination qu'une seule fois. Par exemple :

$$\begin{aligned} a &= 1; \\ b &= 2; \\ a &= a + b; \\ b &= 4 \end{aligned}$$

deviendrait :

$$\begin{aligned} a_1 &= 1; \\ b_1 &= 2; \\ a_2 &= a_1 + b_1; \\ b_2 &= 4 \end{aligned}$$

Il est évident qu'une transformation directe comme celle-ci ne pourrait fonctionner lorsque plusieurs chemins d'exécution mènent aux mêmes instructions. Pour contrer ce problème, une affectation logique est effectuée sur une nouvelle variable commune permettant de saisir les effets de chacune des branches. Pour ce faire, une fonction ϕ est utilisée où les chemins d'exécution se rencontrent. Pour $\phi(x, y)$, la fonction prend la valeur de x si l'exécution provient du chemin de gauche et y si elle provient de celui de droite. Par exemple :

```
1 if (a){
2         a = 2;
3         a = a + b;
4 } else {
5         a = b;
6 }
7         b = a + 3;
```

deviendrait :

```
1 if (a){
2         a1 = 2;
3         a2 = a1 + b1;
4 } else {
5         a3 = b1;
6 }
7         a4 = φ(a2, a3);
8         b3 = a4 + 3;
```

Dans le cas de la décompilation, nous avons affaire à des registres. Ainsi, au lieu d'utiliser des nombres pour les différencier nous utiliserons des lettres (ex : $r1a$, $r2c$, ...).

1.3 Du RTL au C

1.3.1 Exemple simple

L'exemple suivant est tiré de l'article [24] pour lequel la procédure pour passer les arguments aux fonctions utilise les registres.

```
1 f:     ld.w    4[r0],r0
2        mul     r0,r0,r0
3        xor     r0,r1,r0
4        ret
```

Puisque f semble avoir 2 arguments, il est facile de déduire qu'elle pourrait être exprimée ainsi :

```
1 int f(int r0, int r1)
2 {
3         r0 = *(int *)(r0+4);
4         r0 = r0 * r0;
5         r0 = r0 ^ r1;
6         return r0;
7 }
```

En utilisant la forme SSA, nous obtenons le programme suivant :

```
1 int f(int r0, int r1)
2 {
3         r0a = *(int *)(r0+4);
4         r0b = r0a * r0a;
5         r0c = r0b ^ r1;
6         return r0c;
7 }
```

La forme SSA fait apparaître que l'argument $r0$ aurait un type ($int*$) plutôt qu'un type (int). Comme $r0$ est maintenant un pointeur sur un entier, l'affectation de $r0a$ pourrait être remplacée par $r0[1]$ en supposant qu'il s'agit d'un tableau (ou $*(r0 + 1)$ sinon).

```
1 int f(int *r0, int r1)
2 {
3         r0a = r0[1];
4         return (r0a * r0a) ^ r1;
5 }
```

La fonction précédente est une candidate qui a de très bonnes chances d'être la fonction originale qui a été compilée. Évidemment, comme la compilation peut traduire plusieurs fonctions équivalentes vers le même code compilé, la fonction aurait bien pu être :

```
1 int f(int *r0, int r1)
2 {
3         return (*(r0+1) * *(r0+1)) ^ r1;
4 }
```

Pour choisir entre les candidats possibles, plusieurs stratégies peuvent être utilisées. Par exemple, il serait possible de prendre la fonction la plus lisible, laisser choisir l'utilisateur grâce à une interface graphique ou choisir une syntaxe qui répond aux normes de programmation mises en place.

1.3.2 Reconstruction des types

L'algèbre de type interne utilisée est la suivante :

$$\begin{aligned} t &::= char \mid short \mid int \mid ptr(t) \mid array(t) \mid mem(s) \\ s &::= n_1 : t_1, \ldots n_k : t_k \\ r &::= int \mid ptr(t) \end{aligned}$$

où t représente les types, s les membres des *struct* et r les types des registres. Aussi, n_i couvre les nombres naturels et $k > 0$. α couvre t et β couvre r qui permettent d'identifier les nouveaux types générés durant l'unification. La notation $mem(s)$ représente un espace mémoire qui contient des types différents à des emplacements différents.

L'algorithme de reconstruction des types est basé sur l'algorithme W pour ML de Milner [22]. Cependant, il utilise un système de types plus complexe et retarde l'unification au moment où toutes les constantes sont disponibles. Lorsque l'unification échoue, une reconstruction des types C permettant de résoudre l'échec de la résolution des contraintes est produite.

Les instructions machines génèrent des contraintes sur les types des opérandes. Par exemple :

instruction	contrainte
mov r4,r6	t4=t6
ld.w n[r3],r5	$t3 = ptr(mem(n : t5))$
xor r2a,r1b,r1c	$t2a = int, t1b = int, t1c = int$
add r2a,r1b,r1c	$t2a = ptr(\alpha), t1b = int, t1c = ptr(\alpha) \vee$
	$t2a = int, t1b = ptr(\alpha'), t1c = ptr(\alpha') \vee$
	$t2a = int, t1b = int, t1c = int$

Où tk représente le type du registre rk. Il est aussi à noter que les opérateurs surchargés du langage C comme le + produisent plusieurs contraintes dépendamment de comment ils sont utilisés.

Exemple

Pour illustrer le fonctionnement de la décompilation, tout comme dans [24],nous utiliserons un algorithme additionnant les éléments contenus dans une liste chaînée. La structure de données utilisée sera la suivante :

```
1 struct Liste {int nombre; struct Liste *suivant}
```

Quant au code C utilisé, il sera basé sur l'algorithme itératif d'addition des éléments d'une liste.

```
1 int f(struct Liste *x)
2 {
3         int r = 0;
4         for(; x!=0; x = x->suivant) r += x->nombre;
5         return r;
6 }
```

Une fois compilé en RTL, il produit le code suivant :

```
1  f:
2          mov     #0,r1
3          cmp     #0,r0
4          beq     L4F2
5  L3F2:
6          ld.w    0[r0],r2
7          add     r2,r1,r1
8          ld.w    4[r0],r0
9          cmp     #0,r0
10         bne     L3F2
11 L4F2:
12         mov     r1,r0
13         ret
```

Pour poursuivre avec la décompilation, la forme SSA doit être produite.

```
1  f:
2              mov      r0,r0a
3              mov      #0,r1a
4              cmp      #0,r0a
5              beq      L4F2
6  L3F2:
7              mov      φ(r0a,r0c) r0b
8              mov      φ(r1a,r1c) r1b
9              ld.w     0[r0b],r2a
10             add      r2a,r1b,r1c
11             ld.w     4[r0b],r0c
12             cmp      #0,r0c
13             bne      L3F2
14 L4F2:
15             mov      φ(r1a,r1c) r1d
16             mov      r1d,r0d
17             ret
```

Une fois la forme SSA générée, les contraintes de types peuvent enfin être produites.

```
1  f:                                          tf = t0 → t99
2              mov      r0,r0a                 t0 = t0a
3              mov      #0,r1a                 t1a = int ∨
4                                               t1a = ptr(α1)
5              cmp      #0,r0a                 t0a = int ∨
6                                               t0a = ptr(α2)
7              beq      L4F2
8  L3F2:
9              mov      φ(r0a,r0c) r0b         t0b = t0a ∧ t0b = t0c
10             mov      φ(r1a,r1c) r1b         t1b = t1a ∧ t1b = t1c
11             ld.w     0[r0b],r2a             t0b = ptr(mem(0 : t2a))
12             add      r2a,r1b,r1c            t2a = ptr(α3) ∧
13                                              t1b = int ∧
14                                              t1c = ptr(α3) ∨
```

```
15                                          t2a = int ∧
16                                          t1b = ptr(α4) ∧
17                                          t1c = ptr(α4) ∨
18                                          t2a = int ∧
19                                          t1b = int ∧
20                                          t1c = int
21          ld.w    4[r0b], r0c             t0b = ptr(mem(4 : t0c))
22          cmp     #0, r0c                 t0c = int ∨
23                                          t0c = ptr(α5)
24          bne     L3F2
25  L4F2:
26          mov     φ(r1a, r1c) r1d         t1d = t1a ∧ t1d = t1c
27          mov     r1d, r0d                t1d = int ∨
28                                          t1d = ptr(α6)
29          ret                             t99 = t0d
```

Avec ces contraintes, il est maintenant possible d'effectuer la reconstruction des types pour la fonction f. En utilisant l'algorithme W pour ML de Milner avec les ajustements mentionnés plus haut, les contraintes suivantes se retrouvent en cas problématique :

$$\begin{aligned} t0b &= t0c \\ t0b &= ptr(mem(0 : t2a)) \\ t0b &= ptr(mem(4 : t0c)) \end{aligned}$$

Pour briser le cycle, nous utilisons :

$$struct\ Liste1\ \{t2a\ m0;\ t0c\ m4; \ldots\}$$

$$t0c = ptr(mem(0 : t2a, 4 : t0c)) = ptr(struct\ Liste1)$$

Il est à noter que l'on ne peut savoir si la structure de données contient d'autres éléments, d'où l'utilisation des points de suspension. Lors de la traduction en C, un type optionnel de remplissage *Tpad* sera utilisé. En effet, il est possible qu'il n'y ait pas d'autres éléments aussi, comme c'est le cas ici, et comme le langage C ne permet pas d'avoir un type de taille nulle, il doit donc être optionnel.

En gardant en mémoire que cette *mem* est représentée par *struct Liste*1, nous pouvons résoudre les contraintes de types de 2 façons différentes :

$$\begin{aligned} t0 = t0a = t0b = t0c &= ptr(struct\ Liste1) \\ t99 = t1a = t1b = t1c = t1d = t2a = t0d &= int \\ tf &= ptr(struct\ Liste1) \rightarrow int \end{aligned}$$

ou

$$\begin{aligned} t0 = t0a = t0b = t0c &= ptr(struct\ Liste1) \\ t2a &= int \\ t99 = t1a = t1b = t1c = t1d = t0d &= ptr(\alpha 4) \\ tf &= ptr(struct\ Liste1) \rightarrow ptr(\alpha 4) \end{aligned}$$

La deuxième solution peut être rapidement écartée puisqu'elle mène à un code qui n'est pas conforme à la norme ANSI. En effet, en utilisant cette solution avec le code RTL, nous obtenons un programme qui initialise un pointeur à 0 et ensuite le déplace dans la mémoire d'une distance égale à la somme des éléments contenus dans la liste.

Maintenant, il est possible de retrouver un code dans le langage C en reconstituant des expressions à partir des variables utilisées une seule fois et grâce au *pattern matching*.

```
1  struct Liste1 {int m0; struct Liste1 *m4; Tpad m8;};
2
3  int f(struct Liste1)
4  {
5          int r = 0;
6          if(x != 0)
7          {
8                  do
9                  {
10                         r += x->m0;
11                         x = x->m4;
12                 }while (x != 0)
13         }
14         return r;
15 }
```

Ce code reconstruit n'est pas identique au code original, mais de simples transformations suffiraient à lui redonner sa forme originale.

Reconstruction des tableaux

L'exemple précédent montre bien comment décompiler les structures de données. Cependant, elles ne sont pas les seuls éléments représentant un espace mémoire. En effet, les tableaux sont aussi utilisés dans ce but lorsque tous les éléments sont du même type. Compte tenu de cette caractéristique, il est possible d'identifier les tableaux dans le code RTL grâce à l'index utilisé pour accéder aux éléments. Par exemple, le code suivant :

```
1  ld.w    (r5)[r0],r3
```

Cette instruction en RTL générerait les contraintes de types suivantes étant donné que β représente les types que peuvent prendre les registres.

```
1  ld.w    (r5)[r0],r3    t0 = ptr(array(β)) ∧
2                          t5 = int ∧t3 = β ∨
3                          t0 = int ∧t5 = ptr(array(β)) ∧ t3 = β
```

1.3.3 Unification des types

L'unification des types est basée sur l'unification d'Herbrand avec les règles additionnelles pour les cas où cette dernière échoue.

- les variables de type α s'unifient avec le type t contenant α menant à $t[struct\ G/\alpha]$ avec la production de la définition de la *struct* G.
- $array(t)$ et $mem(n_1 : t_1, \ldots n_k L t_k)$ s'unifient à $array(t)$ quand $(\forall i) t_i = t$
- $mem(s_1)$ et $mem(s_2)$ s'unifient à $mem(s_1 \cup s_2)$.

1.3.4 Sélection des types C

Une fois l'unification terminée, il est nécessaire de les faire correspondre à des types dans le langage C. Comme les types générés sont plus expressifs que les types de C, il

est nécessaire de choisir parmi plusieurs possibilités. Celles-ci sont sélectionnées selon si elles sont plus ou moins fréquentes, selon le style de programmation. La méthode proposée par l'article [24] est la suivante :

- *char*, *short*, *int*, deviennent **char**, **short**, **int** ;
- $ptr(t)$ devient T * où T représente t ;
- $array(t)$ devient T [] où T représente t ;
- $mem(s)$ devient :
  - **struct G** if **struct G** a été généré à l'unification pour cette *mem* ;
  - **T** si $s = (0 : t)$ et **T** représente t ;
  - **struct G** où **G** est une nouvelle définition de structure qui dispose les membres typés convertis de s aux bonnes positions. Cet élément peut nécessiter l'utilisation des *Unions* dans le cas où deux éléments seraient superposés ou de cases de remplissage dans le cas où il y aurait des trous.

1.4 Conclusion

Malgré les exemples convaincants montrés, cette technique présente quelques limitations. Principalement, il n'y a pas façon de distinguer certains types de structures de données ou de tableaux. Pensons, par exemple, à une structure de données contenant des *int* suivis d'un tableau de *int* ou d'un tableau de *int* à deux dimensions. Ces deux derniers exemples sont indiscernables d'une structure contenant un nombre de variables de type *int* de la même taille ou d'un tableau à une dimension. Aussi, certaines instructions RTL sont impossibles à traduire en C répondant à la norme ANSI ; Par exemple, l'instruction `jmp r0`.

Malgré ces points, cette approche reste très prometteuse pour permettre la traduction de code RTL vers le langage C.

Chapitre 2

Conversion de l'impératif au déclaratif

2.1 Introduction

Certains travaux ont déjà été effectués dans le but permettre le passage d'un paradigme de programmation à un autre. Plus particulièrement, une fonction de conversion permettant de traduire les programmes écrits dans un langage impératif simple appelé $\mathcal{L}_1$ vers un langage déclaratif $\mathcal{L}_2$ a déjà été définie [20, 21]. Ce travail constitue une base solide pour nous permettre de traiter cette traduction au niveau des tableaux. Ainsi, dans ce chapitre, les deux langages originaux, $\mathcal{L}_1$ et $\mathcal{L}_2$ seront présentés ainsi que la fonction de traduction qui leur est rattachée.

2.2 Langage impératif

Le langage impératif $\mathcal{L}_1$ représente un langage simple dans lequel plusieurs éléments que l'on retrouve dans les langages utilisés couramment en informatique ont été omis. Cependant, il contient plusieurs éléments importants telles l'affectation, les opérations arithmétiques, les expressions conditionnelles et les boucles.

2.2.1 Syntaxe de $\mathcal{L}_1$

Telle que mentionnée plus tôt, la syntaxe du langage $\mathcal{L}_1$ est relativement simple. Sa grammaire BNF est présentée dans le tableau 2.1.

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \\
b & ::= & \mathit{true} \mid \neg b \mid b_1 \vee b_2 \mid a_1 < a_2 \mid a_1 = a_2 \\
P & ::= & x = a \\
 & \mid & \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \\
 & \mid & \textbf{while } b \textbf{ do } P \\
 & \mid & P_1; P_2
\end{array}
$$

TAB. 2.1 – Syntaxe du langage $\mathcal{L}_1$.

Les expressions arithmétiques peuvent être soit une constante, une variable, l'addition de deux variables ou la multiplication de deux variables.

Les expressions booléennes peuvent soit être *vrai* (*true*), la négation d'une expression booléenne, le "ou" logique de deux expressions ou la comparaison de deux expressions arithmétiques soit par l'égalité ou leur ordre.

Finalement, un programme est constitué d'affectations, d'expressions conditionnelles, de boucles et de séquences d'instructions.

2.2.2 Sémantique de $\mathcal{L}_1$

Pour définir la sémantique des expressions, nous allons noter l'ensemble des mémoires possibles par Γ. Ces mémoires contiennent des noms de variables associées à leur valeur. Par exemple, $[x \mapsto 3, y \mapsto 4] \in \Gamma$.

$$
\begin{array}{rcl}
[\![n]\!]_{A_1}(s) & = & n \\
[\![x]\!]_{A_1}(s) & = & s(x) \\
[\![a_1 + a_2]\!]_{A_1}(s) & = & [\![a_1]\!]_{A_1}(s) + [\![a_2]\!]_{A_1}(s) \\
[\![a_1 * a_2]\!]_{A_1}(s) & = & [\![a_1]\!]_{A_1}(s) * [\![a_2]\!]_{A_1}(s) \\
[\![true]\!]_{B_1}(s) & = & true \\
[\![\neg b]\!]_{B_1}(s) & = & \begin{cases} true & \text{si } [\![b]\!]_{B_1}(s) = false \\ false & \text{si } [\![b]\!]_{B_1}(s) = true \end{cases} \\
[\![b_1 \vee b_2]\!]_{B_1}(s) & = & \begin{cases} false & \text{si } [\![b_1]\!]_{B_1}(s) = false \text{ et } [\![b_2]\!]_{B_1}(s) = false \\ true & \text{sinon} \end{cases} \\
[\![a_1 < a_2]\!]_{B_1}(s) & = & \begin{cases} true & \text{si } [\![a_1]\!]_{A_1}(s) < [\![a_2]\!]_{A_1}(s) \\ false & \text{si } [\![a_1]\!]_{A_1}(s) \geq [\![a_2]\!]_{A_1}(s) \end{cases} \\
[\![a_1 = a_2]\!]_{B_1}(s) & = & \begin{cases} true & \text{si } [\![a_1]\!]_{A_1}(s) = [\![a_2]\!]_{A_1}(s) \\ false & \text{sinon} \end{cases} \\
[\![x = a]\!]_{\mathcal{L}_1}(s) & = & s \dagger [x \mapsto [\![a]\!]_{A_1}(s)] \\
[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]_{\mathcal{L}_1}(s) & = & cond([\![b]\!]_{\mathcal{B}_1}(s), [\![P_1]\!]_{\mathcal{L}_1}(s), [\![P_2]\!]_{\mathcal{L}_1}(s)) \\
[\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1}(s) & = & \mu f_{b,P}(g)(s) \\
[\![P_1; P_2]\!]_{\mathcal{L}_1}(s) & = & ([\![P_2]\!]_{\mathcal{L}_1} \circ [\![P_1]\!]_{\mathcal{L}_1})(s) \\
\text{avec :} & & \\
s\ x = v & \text{si} & \{x \mapsto v\} \in s \\
f_{b,P}\ (g)(s) & = & cond([\![b]\!]_{B_1}(s), g \circ [\![P]\!]_{\mathcal{L}_1}(s), s)
\end{array}
$$

TAB. 2.2 – Sémantique du langage $\mathcal{L}_1$.

La sémantique des expressions du langage $\mathcal{L}_1$, telle que présentée dans le tableau 2.2, nécessite l'introduction de certaines fonctions.

Premièrement, la fonction $[\![-]\!]_{\mathcal{A}_1}$ permet d'obtenir la sémantique d'une expression arithmétique. Elle prend une expression arithmétique ainsi qu'une mémoire (un élément de Γ) et retourne un entier relatif.

$$[\![-]\!]_{\mathcal{A}_1} \ : \ \mathcal{A}_1 \to (\Gamma \to \mathbb{Z})$$

Ainsi, avec $s = [x \mapsto 2]$, la sémantique de l'expression $x * 5$ serait :

$$\begin{aligned} [\![x * 5]\!]_{A_1}(s) &= [\![x]\!]_{A_1}(s) * [\![5]\!]_{A_1}(s) \\ &= (s(x)) * 5 \\ &= 2 * 5 \\ &= 10 \end{aligned}$$

Parallèlement à $[\![-]\!]_{\mathcal{A}_1}$, la fonction $[\![-]\!]_{\mathcal{B}_1}$ permet d'obtenir la sémantique d'une expression booléenne. Elle prend une expression booléenne ainsi qu'une mémoire et retourne une valeur de vérité.

$$[\![-]\!]_{\mathcal{B}_1} \ : \ \mathcal{B}_1 \to (\Gamma \to \mathit{bool})$$

Ainsi, avec $s = [y \mapsto 3]$, la sémantique de l'expression $5 < y$ serait :

$$\begin{aligned} [\![5 < y]\!]_{B_1}(s) &= \begin{cases} true & \text{si } [\![5]\!]_{A_1}(s) < [\![y]\!]_{A_1}(s) \\ false & \text{si } [\![5]\!]_{A_1}(s) \geq [\![y]\!]_{A_1}(s) \end{cases} \\ &= \begin{cases} true & \text{si } 5 < (s(y)) \\ false & \text{si } 5 \geq (s(y)) \end{cases} \\ &= \begin{cases} true & \text{si } 5 < 3 \\ false & \text{si } 5 \geq 3 \end{cases} \\ &= false \end{aligned}$$

Quant à elle, la fonction $[\![-]\!]_{\mathcal{L}_1}$ permet d'obtenir la sémantique d'une expression du langage $\mathcal{L}_1$.

$$[\![-]\!]_{\mathcal{L}_1} \ : \ \mathcal{L}_1 \to (\Gamma \to \Gamma)$$

Ainsi, la sémantique de l'affectation à x d'une expression arithmétique $(x = a)$ est l'affectation à x du résultat de l'évaluation de a, soit :

$$\begin{aligned} [\![x = a]\!]_{\mathcal{L}_1}(s) &= s \dagger [x \mapsto [\![a]\!]_{A_1}(s)] \\ \text{avec } (s \dagger [x \mapsto t])(y) &= \begin{cases} t & \text{si } x = y \\ s(y) & \text{si } x \neq y \end{cases} \end{aligned}$$

Pour la séquence d'instructions, sa sémantique est le résultat de la composition de la sémantique de la deuxième expression avec la sémantique de la première expression. Ainsi, pour tout environnement s dans Γ :

$$\begin{aligned} [\![P_1; P_2]\!]_{\mathcal{L}_1}(s) &= ([\![P_2]\!]_{\mathcal{L}_1} \circ [\![P_1]\!]_{\mathcal{L}_1})(s) \\ &= [\![P_2]\!]_{\mathcal{L}_1}([\![P_1]\!]_{\mathcal{L}_1}(s)) \end{aligned}$$

Ensuite, la sémantique du "if" et "$while$", qui utilise le point fixe [27, 28] , nécessite la définition de fonctions auxiliaires.

$$\begin{aligned} \mathit{cond} &: ((\Gamma \to \mathit{bool}) \times (\Gamma \to \Gamma) \times (\Gamma \to \Gamma)) \to (\Gamma \to \Gamma) \\ f_{(b,P)} &: (\Gamma \to \Gamma) \to (\Gamma \to \Gamma) \end{aligned}$$

Pour tout environnement s dans Γ, nous avons :

$$\mathit{cond}(b, f, g)(s) = \begin{cases} f(s) & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = \mathit{true} \\ g(s) & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = \mathit{false} \end{cases}$$

Ainsi, avec une mémoire $s = [x \mapsto 2]$ la sémantique de $if\ x = 2\ then\ y = 1\ else\ y = 2$ donne :

$$
\begin{aligned}
& [\![\textbf{if } x = 2 \textbf{ then } y = 1 \textbf{ else } y = 2]\!]_{\mathcal{L}_1}(s) \\
= \;& cond([\![x = 2]\!]_{B_1}(s), [\![y = 1]\!]_{\mathcal{L}_1}(s), [\![y = 2]\!]_{\mathcal{L}_1}(s)) \\
= \;& cond([\![x = 2]\!]_{B_1}(s), s \dagger [y \mapsto 1], s \dagger [y \mapsto 2]) \\
= \;& \begin{cases} s \dagger [y \mapsto 1] & \text{si } [\![x = 2]\!]_{\mathcal{B}_1}(s) = true \\ s \dagger [y \mapsto 2] & \text{si } [\![x = 2]\!]_{\mathcal{B}_1}(s) = false \end{cases} \\
= \;& \begin{cases} s \dagger [y \mapsto 1] & \text{si } (2 = 2) = true \\ s \dagger [y \mapsto 2] & \text{si } (2 = 2) = false \end{cases} \\
= \;& [x \mapsto 2, y \mapsto 1]
\end{aligned}
$$

Intuitivement, la sémantique du *while* devrait satisfaire :

$$
[\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1}(s) \;=\; \begin{cases} s & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = false \\ [\![P;\ \textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1}(s) & \text{si } [\![b]\!]_{B_1}(s) = true \end{cases}
$$

Ce qui revient à :

$$
[\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1}(s) \;=\; \begin{cases} s & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = false \\ [\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1} s \circ [\![P]\!]_{\mathcal{L}_1}(s) & \text{si } [\![b]\!]_{B_1}(s) = true \end{cases}
$$

Ainsi, $[\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1}$ doit être une solution de l'équation suivante :

$$
[\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1} \;=\; f_{(b,P)}([\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1})
$$

Où pour $g : \Gamma \to \Gamma$ et pour tout environnement s dans Γ, nous avons :

$$
(f_{(b,P)}(g))(s) \;=\; \begin{cases} s & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = false \\ g([\![P]\!]_{\mathcal{L}_1}(s)) & \text{si } [\![b]\!]_{B_1}(s) = true \end{cases}
$$

Ainsi, la sémantique de l'expression **while** est le plus petit point fixe de la fonction f. Soit :

$$
[\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1}(s) \;=\; \mu f_{b,P}(g)(s)
$$

2.2.3 Exemple

L'exemple suivant représente un programme P qui calcule $3^{(x+4)} + 1$. Pour calculer sa sémantique, nous utiliserons la mémoire $s = [x \mapsto 1]$.

```
1 x = x + 4;
2 y = 3;
3 while (1 < x) do
4           y = y * y;
5           x = x - 1;
6 y = y + 1;
```

avec $b = (1 < x)$, P_1 représentant les lignes 2 à 6, P_2 les lignes 3 à 6 et P_3 le corps de la boucle, la sémantique du programme P est :

$$\begin{aligned}
[\![P]\!]_{\mathcal{L}_1}(s) &= ([\![P_1]\!]_{\mathcal{L}_1} \circ [\![x = x + 4]\!]_{\mathcal{L}_1})(s) \\
&= [\![P_1]\!]_{\mathcal{L}_1}([\![x = x + 4]\!]_{\mathcal{L}_1}(s)) \\
&= [\![P_1]\!]_{\mathcal{L}_1}(s \dagger [x \mapsto [\![x + 4]\!]_{A_1}(s)])
\end{aligned}$$

avec $s_1 = s \dagger [x \mapsto [\![x + 4]\!]_{A_1}(s)]$:

$$\begin{aligned}
[\![P]\!]_{\mathcal{L}_1}(s) &= ([\![P_2]\!]_{\mathcal{L}_1} \circ [\![y = 3]\!]_{\mathcal{L}_1})(s_1) \\
&= [\![P_2]\!]_{\mathcal{L}_1}([\![y = 3]\!]_{\mathcal{L}_1}(s_1)) \\
&= [\![P_2]\!]_{\mathcal{L}_1}(s_1 \dagger [y \mapsto 3])
\end{aligned}$$

Avec $s_2 = s \dagger [x \mapsto [\![x + 4]\!]_{A_1}(s), y \mapsto 3]$:

$$\begin{aligned}
f_{b,P_3}(g)(s_2) &= \begin{cases} s_2 & \text{si } 1 \geq x \\ g([\![P_3]\!]_{\mathcal{L}_1}(s_2)) & \text{si } 1 < x \end{cases} \\
&= \begin{cases} s_2 & \text{si } 1 \geq x \\ g([\![y \mapsto y * y, x \mapsto x - 1]\!]_{\mathcal{L}_1}(s_2)) & \text{si } 1 < x \end{cases}
\end{aligned}$$

En appliquant la fonction f_{b,P_3} plusieurs fois à l'environnement indéfini, noté $\perp$, il est possible d'obtenir son plus petit point fixe.

$$\begin{aligned} f_{b,P_3}(\perp)(s_2) &= \begin{cases} s_2 & \text{si } 1 \geq x \\ \perp(s_2) & \text{si } 1 < x \end{cases} \\ &= \begin{cases} s_2 & \text{si } 1 \geq x \\ indef & \text{si } 1 < x \end{cases} \end{aligned}$$

$$\begin{aligned} f^2_{b,P_3}(\perp)(s_2) &= f_{b,P_3}(f_{b,P_3}(\perp))(s_2) \\ &= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ f_{b,P_3}(\perp)[\![P_3]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \\ &= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ [\![P_3]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}[\![P_3]\!]_{\mathcal{L}_1}(s_2) = false \wedge [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \\ indef & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}[\![P_3]\!]_{\mathcal{L}_1}(s_2) = true \wedge [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \\ &= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ [\![\begin{array}{l} y = y * y; \\ x = x - 1 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![1 < x - 1]\!]_{\mathcal{B}_1}(s_2) = false \wedge \\ [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \\ indef & \text{si } \begin{cases} [\![1 < x - 1]\!]_{\mathcal{B}_1}(s_2) = true \wedge \\ [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \end{cases} \\ &= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ [\![y = y * y; x = 1]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![2 = x]\!]_{\mathcal{B}_1}(s_2) = true \\ indef & \text{si } [\![2 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \end{aligned}$$

$$
\begin{aligned}
f^3_{b,P_3}(\bot)(s_2) &= f_{b,P_3}(f^2_{b,P_3}(\bot))(s_2) \\
&= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ f_{b,P_3}(f_{b,P_3}(\bot))[\![P_3]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \\
&= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ [\![P_3]\!]_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![1 < x]\!]_{\mathcal{B}_1}[\![P_3]\!]_{\mathcal{L}_1}(s_2) = false \wedge \\ [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \\ f_{b,P_3}(\bot)[\![P_3]\!]^2_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![1 < x]\!]_{\mathcal{B}_1}[\![P_3]\!]_{\mathcal{L}_1}(s_2) = true \wedge \\ [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \end{cases} \\
&= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ [\![P_3]\!]_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![1 < x]\!]_{\mathcal{B}_1}[\![P_3]\!]_{\mathcal{L}_1}(s_2) = false \wedge \\ [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \\ [\![P_3]\!]^2_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![1 < x]\!]_{\mathcal{B}_1}[\![P_3]\!]^2_{\mathcal{L}_1}(s_2) = false \wedge \\ [\![2 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \\ indef & \text{si } \begin{cases} [\![1 < x]\!]_{\mathcal{B}_1}[\![P_3]\!]^2_{\mathcal{L}_1}(s_2) = true \wedge \\ [\![2 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases} \end{cases} \\
&= \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ \left[\!\!\left[\begin{array}{l} y = y * y; \\ x = x - 1 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![2 = x]\!]_{\mathcal{B}_1}(s_2) = true \\ \left[\!\!\left[\begin{array}{l} y = y * y * y; \\ x = x - 2 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![3 = x]\!]_{\mathcal{B}_1}(s_2) = true \\ indef & \text{si si } [\![3 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases}
\end{aligned}
$$

$$
f^4_{b,P_3}(\bot)(s_2) = \begin{cases} s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\ \left[\!\!\left[\begin{array}{l} y = y * y; \\ x = x - 1 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![2 = x]\!]_{\mathcal{B}_1}(s_2) = true \\ \left[\!\!\left[\begin{array}{l} y = y * y * y; \\ x = x - 2 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![3 = x]\!]_{\mathcal{B}_1}(s_2) = true \\ \left[\!\!\left[\begin{array}{l} y = y * y * y * y; \\ x = x - 3 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![4 = x]\!]_{\mathcal{B}_1}(s_2) = true \\ indef & \text{si si } [\![4 < x]\!]_{\mathcal{B}_1}(s_2) = true \end{cases}
$$

Le plus petit point fixe de la fonction f_{b,P_3} est donc :

$$
\begin{aligned}
\mu f_{b,P_3}(s_2) &= \bigsqcup_{j\geq 0}\{f'^{j}_{(b),P_3}(\bot)\}(s_2) \\
&= \begin{cases}
s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\
[\![P_3]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![2 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
[\![P_3]\!]^2_{\mathcal{L}_1}(s_2) & \text{si } [\![3 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
[\![P_3]\!]^3_{\mathcal{L}_1}(s_2) & \text{si } [\![4 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
\vdots & \\
[\![P_3]\!]^n_{\mathcal{L}_1}(s_2) & \text{si } [\![n+1 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
indef & \text{si } [\![n+1 < x]\!]_{\mathcal{B}_1}(s_2) = true
\end{cases} \\
&= \begin{cases}
s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\
[\![\begin{array}{l} y = y * y; \\ x = x - 1 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![2 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
[\![\begin{array}{l} y = y * y * y; \\ x = x - 2 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![3 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
[\![\begin{array}{l} y = y * y * y * y; \\ x = x - 3 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![4 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
\vdots & \\
[\![\begin{array}{l} y = y^{(n+1)}; \\ x = x - n + 1 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![n+1 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
indef & \text{si } [\![n+1 < x]\!]_{\mathcal{B}_1}(s_2) = true
\end{cases} \\
&= \begin{cases}
s_2 & \text{si } [\![1 < x]\!]_{\mathcal{B}_1}(s_2) = false \\
[\![\begin{array}{l} y = y * y; \\ x = x - 1 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![2 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
[\![\begin{array}{l} y = y * y * y; \\ x = x - 2 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![3 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
[\![\begin{array}{l} y = y * y * y * y; \\ x = x - 3 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![4 = x]\!]_{\mathcal{B}_1}(s_2) = true \\
\vdots & \\
[\![\begin{array}{l} y = y^{(n+1)}; \\ x = x - n + 1 \end{array}]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![n+1 = x]\!]_{\mathcal{B}_1}(s_2) = true
\end{cases}
\end{aligned}
$$

Ainsi avec s_3 représentant l'état de l'environnement lorsque le point fixe est calculé :

$$[\![P]\!]_{\mathcal{L}_1} s \quad = \quad [\![y = y + 1]\!]_{\mathcal{L}_1}(s_3)$$

Ainsi, avec la mémoire $s = [x \mapsto 1]$, nous obtenons :

$$\begin{aligned} [\![x = 5]\!]_{\mathcal{B}_1}(s_2) &= (1 + 4 = 5) \\ &= (5 = 5) \\ &= \mathit{true} \end{aligned}$$

Donc :

$$\begin{aligned} \mu f_{b,P_3}(s_2) &= [\![P_3]\!]^4_{\mathcal{L}_1}(s_2) \\ &= \left[\!\!\left[\begin{array}{l} y = y * y * y * y * y; \\ x = x - 4 \end{array} \right]\!\!\right]_{\mathcal{L}_1} (s_2) \\ &= \left[\begin{array}{l} y \mapsto [\![y * y * y * y * y]\!]_{\mathcal{A}_1}(s_2), \\ x \mapsto [\![x - 4]\!]_{\mathcal{A}_1}(s_2) \end{array} \right] \\ &= [y \mapsto 243, x \mapsto 1] \end{aligned}$$

Ainsi :

$$\begin{aligned} [\![P]\!]_{\mathcal{L}_1}(s) &= [\![y \mapsto y + 1]\!]_{\mathcal{L}_1}[y \mapsto 243, x \mapsto 1] \\ &= [y \mapsto 244, x \mapsto 1] \end{aligned}$$

2.3 Langage déclaratif

Le langage déclaratif utilisé est un petit langage simple qui permet de représenter l'ensemble des programmes de $\mathcal{L}_1$ sous une forme déclarative.

2.3.1 Syntaxe de $\mathcal{L}_2$

La syntaxe de $\mathcal{L}_2$ permet de créer des programmes de façon déclarative. Les programmes créés avec ce langage sont en fait un ensemble de variables auxquelles sont associées une définition. Ainsi, parallèlement à $\mathcal{L}_1$, celle de $\mathcal{L}_2$ est relativement simple. Elle définit des expressions arithmétiques, booléennes et des programmes.

Les expressions arithmétiques peuvent être une constante, une variable, l'addition ou la multiplication de deux expressions arithmétiques, une expression conditionnelle ou bien une expression récursive. Cette dernière est représentée par le l'expression $\mu^{x}_{b\ [x\mapsto a_1,\ldots,x_n\mapsto a_n]}[x' \mapsto a'_1, \ldots, x'_m \mapsto a'_m]$. Pour faire un parallèle avec le langage impératif $\mathcal{L}_1$, $[x \mapsto a_1, \ldots, x_n \mapsto a_n]$ représenterait le corps d'une boucle, $[x' \mapsto a'_1, \ldots, x'_m \mapsto a'_m]$ représenterait l'initialisation des variables avant l'exécution de la boucle.

Les expressions booléennes peuvent être *vrai* (*true*), la négation d'une expression booléenne, le "ou", le "plus petit que" ou le "égal" de deux expressions booléennes.

Un programme quant à lui est constitué d'un ensemble de définitions de variables.

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \\
 & \mid & if(b, a_1, a_2) \mid \mu^{x}_{b\ [x\mapsto a_1,\ldots,x_n\mapsto a_n]}[x' \mapsto a'_1, \ldots, x'_m \mapsto a'_m] \\
b & ::= & true \mid \neg b \mid b_1 \vee b_2 \mid a_1 < a_2 \mid a_1 = a_2 \\
P & ::= & x \mapsto a \\
 & \mid & P_1 \cup P_2
\end{array}
$$

TAB. 2.3 – Syntaxe du langage $\mathcal{L}_2$.

Par exemple, le programme suivant est syntaxiquement correct dans $\mathcal{L}_2$:

$$
\left\{
\begin{array}{rcl}
x & \mapsto & 4, \\
y & \mapsto & 6 + 8, \\
z & \mapsto & if(10 < 4 + 5, 2, 1)
\end{array}
\right\}
$$

2.3.2 Sémantique de $\mathcal{L}_2$

Pour accompagner la syntaxe définie, la sémantique est présentée dans le tableau 2.4.

$$
\begin{array}{rcl}
[\![n]\!]_{A_2}(s) & = & n \\
[\![x]\!]_{A_2}(s) & = & s(x) \\
[\![a_1 + a_2]\!]_{A_2}(s) & = & [\![a_1]\!]_{A_2}(s) + [\![a_2]\!]_{A_2}(s) \\
[\![a_1 * a_2]\!]_{A_2}(s) & = & [\![a_1]\!]_{A_2}(s) * [\![a_2]\!]_{A_2}(s) \\
[\![if(b, a_1, a_2)]\!]_{A_2}(s) & = & cond([\![b]\!]_{B_2}(s), [\![a_1]\!]_{A_2}(s), [\![a_2]\!]_{A_2}(s)) \\
[\![\mu^x_{b,\ P} P']\!]_{A_2}(s) & = & (\mu f_{b,P}(x))[\![P']\!]_{\mathcal{L}_2}(s) \\
[\![true]\!]_{B_2}(s) & = & true \\
[\![\neg b]\!]_{B_2}(s) & = & \begin{cases} true & \text{si } [\![b]\!]_{B_2}(s) = false \\ false & \text{si } [\![b]\!]_{B_2}(s) = true \end{cases} \\
[\![b_1 \vee b_2]\!]_{B_2}(s) & = & \begin{cases} false & \text{si } \begin{cases} [\![b_1]\!]_{B_2}(s) = false \wedge \\ [\![b_2]\!]_{B_1}(s) = false \end{cases} \\ true & \text{sinon} \end{cases} \\
[\![a_1 < a_2]\!]_{B_2}(s) & = & \begin{cases} true & \text{si } [\![a_1]\!]_{A_2}(s) < [\![a_2]\!]_{A_1}(s) \\ false & \text{si } [\![a_1]\!]_{A_2}(s) \geq [\![a_2]\!]_{A_1}(s) \end{cases} \\
[\![a_1 = a_2]\!]_{B_2}(s) & = & \begin{cases} true & \text{si } [\![a_1]\!]_{A_2}(s) = [\![a_2]\!]_{A_2}(s) \\ false & \text{sinon} \end{cases} \\
[\![x \mapsto a]\!]_{\mathcal{L}_2}(s) & = & s \dagger \{x \mapsto [\![a]\!]_{A_2}(s)\} \\
[\![P_1 \cup P_2]\!]_{\mathcal{L}_2}(s) & = & [\![P_1]\!]_{\mathcal{L}_2}(s) \cup [\![P_2]\!]_{\mathcal{L}_2}(s) \\
\text{avec :} & & \\
P & = & [x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] \\
P' & = & [x' \mapsto a'_1, \ldots, x'_m \mapsto a'_m] \\
f_{b,P}\ g(s) & = & cond([\![b]\!]_{B_2}(s), g \circ [\![P]\!]_{\mathcal{L}_2}(s), s)
\end{array}
$$

TAB. 2.4 – Sémantique du langage $\mathcal{L}_2$.

La sémantique des expressions arithmétiques est définie comme une fonction qui, pour une expression donnée, prend un environnement et retourne un entier, soit :

$$[\![-]\!]_{\mathcal{A}_2} \ : \ \mathcal{A}_2 \to (\Gamma \to \mathbb{Z})$$

Ainsi, la sémantique de l'expression $x + y$ avec l'environnement $s = [x \mapsto 3, y \mapsto 2]$ donne :

$$\begin{aligned} [\![x + y]\!]_{A_2}(s) &= [\![x]\!]_{A_2}(s) + [\![y]\!]_{A_2}(s) \\ &= 3 + 2 \\ &= 5 \end{aligned}$$

De la même façon, la sémantique des expressions booléennes est définie comme une fonction qui, pour une expression donnée, prend un environnement et retourne une valeur de vérité, soit :

$$[\![-]\!]_{\mathcal{B}_2} \ : \ \mathcal{B}_2 \to (\Gamma \to \mathit{bool})$$

Pour ce qui est de la sémantique d'une des expressions du langage $\mathcal{L}_2$, elle est donnée par :

$$[\![-]\!]_{\mathcal{L}_2} \ : \ \mathcal{L}_2 \to (\Gamma \to \Gamma)$$

La fonction auxiliaire *cond* est définie d'une façon semblable à celle du langage $\mathcal{L}_1$ soit :

$$\mathit{cond}([\![b]\!]_{\mathcal{B}_2}, [\![a_1]\!]_{\mathcal{L}_2}, [\![a_2]\!]_{\mathcal{L}_2}) \circ s = \begin{cases} [\![a_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = \mathit{true} \\ [\![a_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = \mathit{false} \end{cases}$$

La sémantique des expressions récursives ($\mu^x_{b,\ P}[x' \mapsto a'_1, \ldots, x'_m \mapsto a'_m]$) correspond à la projection sur la variable x du plus petit point fixe de la fonction $f_{b,\ P}$ appliquée à l'environnement $[x' \mapsto a'_1, \ldots, x'_m \mapsto a'_m]$.

Concernant l'affectation, sa sémantique est le résultat de l'évaluation de l'expression arithmétique correspondant à la variable. Pour tout environnement s dans Γ[1] :

$$\begin{aligned} [\![x \mapsto a]\!] \circ s &= [x \mapsto [\![a]\!]] \circ s \\ &= s \dagger \{x \mapsto [\![a]\!](s)\} \end{aligned}$$

La sémantique de l'union de deux programmes est l'union de la sémantique de ces deux programmes. Il est à noter que les deux programmes doivent être disjoints, car

[1] Pour plus de clarté, les accolades sont omises lors du calcul de la sémantique.

sinon il pourrait y avoir des contradictions dans les définitions de variables. Par exemple, le programme $[x \mapsto 1, x \mapsto 3]$ n'est pas valide.

2.3.3 Exemple

L'exemple suivant correspond à un programme P calculant $3^{(x+4)} + 1$.

$$\left\{ y \mapsto 1 + \mu^{y}_{(1<x),[y\mapsto y*y, x\mapsto x-1]}[x \mapsto x+4, y \mapsto 3] \right\}$$

Sa sémantique est donnée ainsi :

$$\begin{aligned} [\![P]\!]_{\mathcal{L}_2}(s) &= s \dagger [y \mapsto [\![1 + \mu^{y}_{(1<x),[y\mapsto y*y, x\mapsto x-1]}[x \mapsto x+4, y \mapsto 3]]\!]_{\mathcal{L}_2}(s)] \\ &= s \dagger [y \mapsto [\![1]\!]_{\mathcal{L}_2}(s) + [\![\mu^{y}_{(1<x),[y\mapsto y*y, x\mapsto x-1]}[x \mapsto x+4, y \mapsto 3]]\!]_{\mathcal{L}_2}(s)] \end{aligned}$$

Avec $P_1 = [y \mapsto y * y, x \mapsto x - 1]$, la sémantique de $\mu^{y}_{(1<x),P_1}$ est la projection du plus petit point fixe de la fonction $f_{(i<x),P_1}$ sur la variable y. Ainsi, en appliquant plusieurs fois $\perp$ sur cette fonction, nous obtenons son plus petit point fixe.

$$\begin{aligned} f_{(1<x),P_1}(\perp)(s) &= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ (\perp)(s) & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \\ &= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ indef & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \end{aligned}$$

$$\begin{aligned} f^2_{(1<x),P_1}(\perp)(s) &= f_{(1<x),P_1}(f_{(1<x),P_1}(\perp))(s) \\ &= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ f_{(1<x),P_1}(\perp)[\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \\ &= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}[\![P_1]\!]_{\mathcal{L}_2}(s) = false \wedge [\![1 < x]\!]_{\mathcal{B}_2}(s) = true \\ indef & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}[\![P_1]\!]_{\mathcal{L}_2}(s) = true \wedge [\![1 < x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \\ &= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![1 < x-1]\!]_{\mathcal{B}_2}(s) = false \wedge [\![1 < x]\!]_{\mathcal{B}_2}(s) = true \\ indef & \text{si } [\![1 < x-1]\!]_{\mathcal{B}_2}(s) = true \wedge [\![1 < x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \\ &= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![x = 2]\!]_{\mathcal{B}_2}(s) = true \\ indef & \text{si } [\![2 < x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \end{aligned}$$

$$
\begin{aligned}
f^3_{(1<x),P_1}(\bot)(s) &= f_{(1<x),P_1}(f^2_{(1<x),P_1}(\bot))(s) \\
&= \begin{cases} s & \text{si } [\![1<x]\!]_{\mathcal{B}_2}(s) = false \\ f_{(1<x),P_1}(f_{(1<x),P_1}(\bot))[\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![1<x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \\
&= \begin{cases} s & \text{si } [\![1<x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } \begin{cases} [\![1<x]\!]_{\mathcal{B}_2}[\![P_1]\!]_{\mathcal{L}_2}(s) = false \wedge \\ [\![1<x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \\ f_{(1<x),P_1}(\bot)[\![P_1]\!]^2_{\mathcal{L}_2}(s) & \text{si } \begin{cases} [\![1<x]\!]_{\mathcal{B}_2}[\![P_1]\!]_{\mathcal{L}_2}(s) = true \wedge \\ [\![1<x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \end{cases} \\
&= \begin{cases} s & \text{si } [\![1<x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![2=x]\!]_{\mathcal{B}_2}(s) = true \\ [\![P_1]\!]^2_{\mathcal{L}_2}(s) & \text{si } \begin{cases} [\![3<x]\!]_{\mathcal{B}_2}[\![P_1]\!]_{\mathcal{L}_2}(s) = false \wedge \\ [\![2<x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \\ indef & \text{si } \begin{cases} [\![3<x]\!]_{\mathcal{B}_2}[\![P_1]\!]_{\mathcal{L}_2}(s) = true \wedge \\ [\![2<x]\!]_{\mathcal{B}_2}(s) = true \end{cases} \end{cases} \\
&= \begin{cases} s & \text{si } [\![1<x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![2=x]\!]_{\mathcal{B}_2}(s) = true \\ [\![P_1]\!]^2_{\mathcal{L}_2}(s) & \text{si } [\![3=x]\!]_{\mathcal{B}_2}(s) = true \\ indef & \text{si } [\![3<x]\!]_{\mathcal{B}_2}(s) = true \end{cases}
\end{aligned}
$$

La forme générale de $f^j_{(1<x),P_1}(\bot)(s)$ est donc :

$$
f^j_{(1<x),P_1}(\bot)(s) = \begin{cases} s & \text{si } [\![1<x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![2=x]\!]_{\mathcal{B}_2}(s) = true \\ [\![P_1]\!]^2_{\mathcal{L}_2}(s) & \text{si } [\![3=x]\!]_{\mathcal{B}_2}(s) = true \\ [\![P_1]\!]^3_{\mathcal{L}_2}(s) & \text{si } [\![4=x]\!]_{\mathcal{B}_2}(s) = true \\ \vdots & \\ [\![P_1]\!]^j_{\mathcal{L}_2}(s) & \text{si } [\![j+1=x]\!]_{\mathcal{B}_2}(s) = true \\ indef & \text{si } [\![j+1<x]\!]_{\mathcal{B}_2}(s) = true \end{cases}
$$

Ainsi, le plus petit point fixe est :

$$\mu f^j_{(1<x),P_1} s = \bigsqcup_{j\geq 0}\{f^j_{(1<x),P_1}(\bot)\}(s)$$

$$= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ [\![P_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![2 = x]\!]_{\mathcal{B}_2}(s) = true \\ [\![P_1]\!]^2_{\mathcal{L}_2}(s) & \text{si } [\![3 = x]\!]_{\mathcal{B}_2}(s) = true \\ [\![P_1]\!]^3_{\mathcal{L}_2}(s) & \text{si } [\![4 = x]\!]_{\mathcal{B}_2}(s) = true \\ \vdots & \\ [\![P_1]\!]^n_{\mathcal{L}_2}(s) & \text{si } [\![n+1 = x]\!]_{\mathcal{B}_2}(s) = true \end{cases}$$

$$= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2}(s) = false \\ [\![y \mapsto y * y, x \mapsto x - 1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![2 = x]\!]_{\mathcal{B}_2}(s) = true \\ [\![y \mapsto y * y * y, x \mapsto x - 2]\!]_{\mathcal{L}_2}(s) & \text{si } [\![3 = x]\!]_{\mathcal{B}_2}(s) = true \\ [\![y \mapsto y * y * y * y, x \mapsto x - 3]\!]_{\mathcal{L}_2}(s) & \text{si } [\![4 = x]\!]_{\mathcal{B}_2}(s) = true \\ \vdots & \\ [\![y \mapsto y^{n+1}, x \mapsto x - n]\!]_{\mathcal{L}_2}(s) & \text{si } [\![n+1 = x]\!]_{\mathcal{B}_2}(s) = true \end{cases}$$

Puisque dans l'exemple, le plus petit point fixe est appliqué à l'environnement $[x \mapsto x + 4, y \mapsto 3]$, nous obtenons :

$$\mu f^j_{(1<x),P_1}[x \mapsto x + 4, y \mapsto 3](s)$$

$$= \begin{cases} s & \text{si } [\![1 < x]\!]_{\mathcal{B}_2} s = false \\ [\![y \mapsto 3 * 3, x \mapsto x + 3]\!]_{\mathcal{L}_2}(s) & \text{si } [\![-2 = x]\!]_{\mathcal{B}_2}(s) = true \\ [\![y \mapsto 3 * 3 * 3, x \mapsto x + 2]\!]_{\mathcal{L}_2}(s) & \text{si } [\![-1 = x]\!]_{\mathcal{B}_2}(s) = true \\ [\![y \mapsto 3 * 3 * 3 * 3, x \mapsto x + 1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![0 = x]\!]_{\mathcal{B}_2}(s) = true \\ \vdots & \\ [\![y \mapsto 3^{n+1}, x \mapsto x - n + 4]\!]_{\mathcal{L}_2}(s) & \text{si } [\![n+5 = x]\!]_{\mathcal{B}_2}(s) = true \end{cases}$$

Par exemple, avec $s = [x \mapsto 1]$, nous obtenons :

$$[\![P]\!]_{\mathcal{L}_2}(s) = s \dagger [y \mapsto [\![1]\!]_{\mathcal{L}_2} s + [\![\mu^y_{(1<x),[y\mapsto y*y,x\mapsto x-1]}[x \mapsto x + 4, y \mapsto 3]]\!]_{\mathcal{L}_2}(s)]$$

Puisque :

$$\begin{aligned} [\![x = 1]\!]_{\mathcal{B}_2}(s) &= (1 = 1) \\ &= true \end{aligned}$$

Nous obtenons :

$$\begin{aligned} [\![P]\!]_{\mathcal{L}_2}(s) &= s \dagger [\![y \mapsto 1 + (3 * 3 * 3 * 3 * 3)]\!]s \\ &= [y \mapsto 244, x \mapsto 1] \end{aligned}$$

2.4 Conversion

Pour traduire des programmes du langage $\mathcal{L}_1$ vers le langage $\mathcal{L}_2$, certaines notions doivent être apportées.

Premièrement, l'ensemble des variables affectées dans un programme P de $\mathcal{L}_1$ est noté $V(P)$. Cette dernière est définie dans le tableau 2.5

$$\begin{aligned} V(x = a) &= \{x\} \\ V(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) &= V(P_1) \cup V(P_2) \\ V(\textbf{while } b \textbf{ do } P) &= V(P) \\ V(P_1; P_2) &= V(P_1) \cup V(P_2) \end{aligned}$$

TAB. 2.5 – Ensemble des variables définies dans P.

Par exemple :

$$\begin{aligned} V(\textbf{if}(i < 3) \textbf{ then } x = 2 \textbf{ else } x = 1;\ y = 4) &= V(x = 2) \cup V(x = 1;\ y = 4) \\ &= V(x = 2) \cup V(x = 1) \cup V(y = 4) \\ &= \{x, y\} \end{aligned}$$

Ensuite, l'application est définie par le tableau 2.6. L'application d'un programme P à une expression e est notée $P(e)$.

$$
\begin{array}{rcl}
P(c) & = & c \\
P(x) & = & \begin{cases} a & \text{si } [x \mapsto a] \in P \\ x & \text{sinon} \end{cases} \\
P(\neg b) & = & \neg(P(b)) \\
P(a_1 \ op \ a_2) & = & P(a_1) \ op \ P(a_2) \\
P(if(b, a_1, a_2)) & = & if(P(b), P(a_1), P(a_2)) \\
P(\mu^x_{b,P_1}[x'_1 \mapsto a'_1, \ldots, x'_m \mapsto a'_m]) & = & \mu^x_{b,P_1} P \dagger [x'_1 \mapsto P(a'_1), \ldots, x'_m \mapsto P(a'_m)] \\
P(x \mapsto a_1) & = & x \mapsto P(a_1) \\
\text{avec } c & \in & \{n, true, false\} \\
op & \in & \{+, *, =, <, \vee\} \\
P_1 & = & [x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]
\end{array}
$$

TAB. 2.6 – Application.

Par exemple, l'application de P à $x + 1$ avec $P = [x \mapsto 3]$:

$$
\begin{array}{rcl}
P(x+1) & = & P(x) + P(1) \\
& = & 3 + 1 \\
& = & 4
\end{array}
$$

Finalement, la fonction $\mathcal{D}$, telle que montrée par le tableau 2.7, permet de traduire un programme de $\mathcal{L}_1$ vers $\mathcal{L}_2$.

Une affectation "$x = a$" est traduite simplement par $\{x \mapsto a\}$.

La traduction de la séquence de deux programmes est la composition de la conversion du second programme avec celle du premier.

$$
\begin{array}{rcl}
\mathcal{D}(x = a) & = & \{x \mapsto a\} \\
\mathcal{D}(P_1; P_2) & = & D(P_2) \circ D(P_1) \\
& = & \bigcup_{x \in V(P_2)} \{(x \mapsto D(P_1)(D(P_2)(x)))\} \cup \\
& & \bigcup_{x \in (V(P_1) - V(P_2))} \{x \mapsto D(P_1)(x)\} \\
\mathcal{D}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) & = & \bigcup_{x \in V(P_1;P_2)} \{x \mapsto if(b, D(P_1)(x), D(P_2)(x))\} \\
\mathcal{D}(\textbf{while } b \textbf{ do } P) & = & \bigcup_{x \in V(P)} \{x \mapsto \mu^x_{b,D(P)}\}
\end{array}
$$

TAB. 2.7 – $\mathcal{D}$ - La fonction de traduction de $\mathcal{L}_1$ à $\mathcal{L}_2$.

La traduction d'une expression conditionnelle donne un ensemble d'affectations d'expressions conditionnelles.

La traduction d'une expression *while* donne un ensemble d'affectations d'expressions récursives.

2.4.1 Exemple

Pour faire suite aux deux exemples donnés lors de la présentation des deux langages, nous allons traduire le programme P suivant :

```
1 x = x + 4;
2 y = 3;
3 while (1 < x) do
4           y = y * y;
5           x = x - 1;
6 y = y + 1;
```

Ainsi, P_1 représentant les lignes 2 à 6, P_2 les lignes 3 à 6 et P_3 le corps de la boucle, la traduction donne :

$$
\begin{aligned}
\mathcal{D}(P) &= \mathcal{D}(P_1) \circ \mathcal{D}(x = x + 4) \\
\mathcal{D}(P_1) &= \mathcal{D}(P_2) \circ \mathcal{D}(y = 3) \\
\mathcal{D}(P_2) &= \mathcal{D}(y = y + 1) \circ \mathcal{D}(\textbf{while } (1 < x) \textbf{ do } P_3 \\
\mathcal{D}(\textbf{while } (1 < x) \textbf{ do } P_3 &= \bigcup_{x \in P_3} \{x \mapsto \mu^x_{(1<x),D(P_3)}\} \\
\mathcal{D}(P_3) &= \mathcal{D}(x = x - 1) \circ \mathcal{D}(y = y * y) \\
\mathcal{D}(P_3) &= \{x \mapsto x - 1, y \mapsto y * y\} \\
\mathcal{D}(\textbf{while } (1 < x) \textbf{ do } P_3) &= \left\{ \begin{array}{l} x \mapsto \mu^x_{(1<x),[x \mapsto x-1, y \mapsto y*y]} \\ y \mapsto \mu^y_{(1<x),[x \mapsto x-1, y \mapsto y*y]} \end{array} \right\} \\
\mathcal{D}(P_2) &= \left\{ \begin{array}{l} x \mapsto \mu^x_{(1<x),[x \mapsto x-1, y \mapsto y*y]} \\ y \mapsto 1 + \mu^y_{(1<x),[x \mapsto x-1, y \mapsto y*y]} \end{array} \right\} \\
\mathcal{D}(P_1) &= \left\{ \begin{array}{l} x \mapsto \mu^x_{(1<x),[x \mapsto x-1, y \mapsto y*y]}[y \mapsto 3] \\ y \mapsto 1 + \mu^y_{(1<x),[x \mapsto x-1, y \mapsto y*y]}[y \mapsto 3] \end{array} \right\} \\
\mathcal{D}(P) &= \left\{ \begin{array}{l} x \mapsto \mu^x_{(1<x),[x \mapsto x-1, y \mapsto y*y]}[y \mapsto 3, x \mapsto x + 4] \\ y \mapsto 1 + \mu^y_{(1<x),[x \mapsto x-1, y \mapsto y*y]}[y \mapsto 3, x \mapsto x + 4] \end{array} \right\}
\end{aligned}
$$

Il est à noter que la traduction du programme P ne donne pas le même résultat que le programme présenté en exemple dans la section du langage $\mathcal{L}_2$. Cette différence provient du fait que la variable x ne nous intéressait pas puisqu'elle n'est qu'une variable de travail (elle ne contient pas de résultat intéressant). Ainsi, lors de la définition du programme en langage déclaratif, elle a été utilisée dans la définition de y comme il se doit, mais elle n'avait pas de définition indépendante rattachée à elle. Si nous avions voulu avoir un programme correspondant réellement au programme P de $\mathcal{L}_1$, nous aurions dû écrire le programme que la traduction ci-haut a produit.

2.5 Conclusion

Dans ce chapitre, les résultats du travail effectué par M. Mbarki [21] ont été présentés. Ainsi, deux langages ont été définis, un langage impératif simple ainsi qu'un langage déclaratif. Ensuite, une fonction de conversion permettant de traduire les programmes écrits dans le premier langage vers le deuxième a été définie. Il est important de noter que même si nous avons omis cette partie, l'auteur prouve que la fonction de conversion $\mathcal{D}$ préserve la sémantique des programmes lors de la traduction. Pour plus de détails sur la preuve, se référer à la section 8.9 du mémoire de maîtrise [21].

Chapitre 3

Coupe de programmes en présence de tableaux et de pointeurs

3.1 Introduction

Telle que définie dans [7], la coupe de programmes est une méthode viable pour focaliser sur un sous-composant spécifique d'un programme. Comme il a été possible de remarquer à la section précédente, l'algorithme utilisé pour traduire les programmes impératifs vers un langage déclaratif porte certaines similarités avec un algorithme de coupe de programmes. En effet, chacune des définitions de variables résultantes de la traduction focalise spécifiquement sur la définition de la variable en question, sans tenir compte du reste du programme d'origine. Ainsi, comme le but de notre travail est de permettre la traduction pour les tableaux, il est intéressant d'identifier les problématiques ainsi que les solutions apportées à la coupe de programmes en présence de tableaux. Premièrement, la notion de coupe de programmes sera introduite. Ensuite, une comparaison entre celle-ci et la fonction de conversion sera présentée. Finalement, les notions relatives à la problématique seront présentées.

3.2 Coupe de programmes

Cette section introduit l'algorithme de coupe (*slicing*) tel que présenté initialement par Weiser [30]. Pour ce faire, certaines notions se doivent d'être définies.

Comme un critère de coupe (*slicing criterion*) permet de définir un cadre pour observer le comportement d'un programme, il est nécessaire que celui-ci soit bien défini. Ainsi, un critère de coupe est défini par la paire $< i, v >$ où i est le numéro de l'instruction à laquelle l'observation sera faite et v est l'ensemble des variables à observer.

La représentation des programmes est faite grâce aux graphes de flots (*flowgraph*). Ceux-ci s'expriment grâce à un tuple $G =< N, E, n_0 >$, où N est l'ensemble des noeuds, E est l'ensemble des arcs et n_0 est le noeud de départ. Si (n, m) est un arc dans E, alors n est un successeur immédiat de m. Un chemin de longueur k de n à m est un ensemble de noeuds $p(0), p(1), \ldots, p(k)$ tel que $p(0) = n$, $p(k) = m$ et $(P(i), p(i+1))$ appartient à E pour tout $0 \leq i \leq k-1$. Un noeud m domine n $(DOM(n) = m)$ si et seulement si m se retrouve sur tous les chemins menant à n. Puisque les noeuds représentent les instructions du programme, l'utilisation de ces termes peut être interchangée.

Ainsi, pour reprendre la notation de l'article [16], nous avons :

$IMS(n)$	représente l'ensemble des successeurs immédiats de n.
$ID(n)$	représente le dominateur immédiat de n.
$RID(n)$	représente le dominateur inverse immédiat de n.
$ND(n)$	représente toutes les instructions qui sont sur un chemin entre n et $RID(n)$ excluant les deux extrémités n et $RID(n)$. $ND(n)$ est vide à moins que n ait plus d'un successeur immédiat.
$USE(n)$	représente toutes les variables dont la valeur peut être utilisée à n.
$MOD(n)$	représente toutes les variables dont la valeur peut être modifiée à n.
$RIN_C(n)$	représente l'ensemble des variables liées à la position de l'instruction n. Chacune des variables dans cet ensemble a un effet potentiel sur la valeur des variables dans V.
$POS(C)$	représente la position de l'instruction du critère de coupe $(POS(C) = i)$.

3.2.1 Algorithme de coupe

Pour faire ressortir les coupes en utilisant l'analyse de flot de données, il est nécessaire de tracer à partir de la fin l'influence possible des variables. Pour ce faire, les

variables dont la valeur peut être utilisée (USE) et les variables dont la valeur peut être modifiée (MOD) seront utilisées. En utilisant les fonctions précédemment définies avec i et n représentant des numéros d'instruction, RIN_C est définie formellement ainsi :

$$\begin{aligned} RIN_C^0(n) &= \{V|n=i\} \cup \{USE(n)|MOD(n) \cap RIN_C^0(IMS(n)) \neq \emptyset\} \cup \\ &\quad \{RIN_C^0(IMS(n)) - MOD(n)\} \\ S_C^0 &= \{n|MOD(n) \cap RIM_C^0(IMS(n)) \neq \emptyset\} \\ B_C^0 &= \{b|ND(b) \cap S_C^0 \neq \emptyset\} \end{aligned}$$

$RIN_C^0(n)$ contient les variables dont la valeur peut influencer directement le comportement des variables contenues dans $VAR(C)$. S_C^0 quant à lui contient les instructions dont l'exécution peut influencer directement la valeur des variables contenues dans $RIN_C^0(n)$. Pour obtenir un ensemble complet de toutes les instructions pouvant avoir un effet direct ou indirect sur la valeur des variables contenues dans $VAR(C)$, les itérations suivantes doivent être utilisées jusqu'à l'obtention du plus petit point fixe de S_c^i.

$$\begin{aligned} RIN_C^{i+1}(n) &= RIN_C^i(n) \underset{b \in B_C^i}{\cup} RIN_{BC(b)}^0(n) \\ S_C^{i+1} &= \{n|MOD(n) \cap RIN_C^{i+1}(IMS(n)) \neq \emptyset \textit{ ou } n \in B_c^i\} \\ B_C^{i+1} &= \{b|ND(b) \cap S_C^i \neq \emptyset\} \end{aligned}$$

où $BC(b)$ est le critère d'instructions de branchement défini par : $< b, USE(b) >$.

L'exemple suivant démontre le fonctionnement de l'algorithme.

```
1  w = read();
2  x = read();
3  y = 7;
4  z = 3;
5  if (w < 10) {
6          w = x + y;
7  }else{
8          z = z+1;
9          w = x * x;
10 }
11 z = z + y + 3;
12 w = w + 1;
13 x = x + 1;
```

Avec la fonction *read*() retournant un entier entré par l'utilisateur.

Avec le critère $< 13, \{w\} >$, nous obtenons les RIN_C^0 suivants :

```
1  w = read();                                  {}
2  x = read();                                  {}
3  y = 7;                                       {x}
4  z = 3;                                       {x,y}
5  if (w < 10) {                                {x,y}
6          w = x + y;                           {x,y}
7  }else{
8          z = z+1;                             {x}
9          w = x * x;                           {x}
10 }
11 z = z + y + 3;                  {w}
12 w = w + 1;                      {w}
13 x = x + 1;                      {w}
```

Ainsi :

$$B_C^0 = \{5\}$$
$$S_C^0 = \{2, 3, 6, 9, 12\}$$

Nous devons maintenant calculer RIN_C^1. Avec $BC(5) = < 5, \{w\} >$

```
1  w = read();                                  {}
2  x = read();                                  {w}
3  y = 7;                                       {w,x}
4  z = 3;                                       {w,x,y}
5  if (w < 10) {                                {w,x,y}
6          w = x + y;                           {x,y}
7  }else{
8          z = z+1;                             {x}
9          w = x * x;                           {x}
10 }
11 z = z + y + 3;                  {w}
12 w = w + 1;                      {w}
13 x = x + 1;                      {w}
```

Ainsi, nous obtenons :

$$S_C^1 = \{1, 2, 3, 5, 6, 9, 12\}$$

En réexécutant l'itération une autre fois, nous obtenons le même résultat, il est donc possible d'affirmer que les instructions faisant partie de la coupe liée au critère $< 13, \{w\} >$ correspondent à :

$$S_C = \{1, 2, 3, 5, 6, 9, 12\}$$

3.3 Comparaison à la conversion

Suite à la présentation de la coupe de programmes et de la fonction de conversion telles que montrées au chapitre 2, il est possible de remarquer une certaine similitude entre les deux. En effet, à partir d'un programme, la coupe permet d'identifier les instructions ayant une influence sur une instruction donnée. De son côté, la traduction permet d'obtenir la définition d'une variable à partir d'un programme. Cette définition est produite à l'aide des instructions ayant une influence sur la variable en question. Il est à noter que même si la syntaxe langage $\mathcal{L}_2$ permet de créer des programmes dont les définitions dépendent d'une ou plusieurs autres variables (ex : $[x \mapsto 3, y \mapsto x + 6]$), la traduction respecte la sémantique qui ne permet pas cette dépendance.

Ainsi, la traduction pourrait être vue comme l'ensemble des coupes dont le critère de coupe correspond à la dernière instruction du programme et une variable du programme. Puisque chacun des éléments d'un tel ensemble représente les instructions ayant un effet sur une variable donnée, l'ensemble permet de retrouver la définition de chacune des variables.

Il est aussi possible de remarquer cette similitude dans la fonction utilisée pour la traduction. Pour ce faire, la traduction de la séquentialité, telle que présentée à la page 35, sera utilisée :

$$\begin{aligned}\mathcal{D}(P_1; P_2) &= D(P_2) \circ D(P_1) \\ &= \bigcup_{x \in V(P_2)} \{(x \mapsto D(P_1)(D(P_2)(x)))\} \cup \\ &\quad \bigcup_{x \in (V(P_1) - V(P_2))} \{x \mapsto D(P_1)(x)\}\end{aligned}$$

Dans cette définition, nous pouvons voir que pour chacune des variables affectées dans P_2, la traduction de P_1 est appliquée à sa définition dans la traduction de P_2. Lors de cette application, si la définition dans $\mathcal{D}(P_2)$ utilise une variable précédemment définie dans la traduction de P_1, sa valeur est utilisée sinon, elle reste inchangée. Puisque la traduction est construite à partir des instructions du programme, chacune des instructions ayant une influence sur la valeur finale d'une variable x sera prise en compte dans la définition de x.

La majeure différence entre la traduction utilisant la coupe décrite au début de la section et la fonction de traduction est que cette dernière applique le résultat des instructions précédentes au fur et à mesure de son exécution.

3.4 Coupe en présence de tableaux et de pointeurs

Tels que mentionnés dans [9, 16], plusieurs problèmes surviennent lorsque l'algorithme précédent se retrouve en face d'un langage comme le C. Un de ces problèmes est en lien avec les pointeurs et les tableaux. En effet, dans l'algorithme présenté plus tôt, chacun des éléments est traité comme un objet unique. Une modification ou l'usage d'un élément sont traités comme si tout l'objet était touché.

Pour reprendre l'exemple de [16] :

```
1 *(p+i) = c1;
2 *(p+j) = c2;
3 if (e)
4           k = i;
5 else k = j;
6 x = *(p+k);
7 printf("%d", x);
```

En utilisant l'algorithme présenté précédemment avec le critère de coupe $< 7, x >$, l'instruction 1 ne serait pas incluse dans la coupe. Cependant, lorsqu'on regarde ce que le programme effectue, il est important que cette instruction se retrouve dans la coupe.

Pour résoudre ce problème, les auteurs de [16] proposent l'utilisation de variables bidons ($DUMMY$) pour chaque pointeur. Ces variables sont annotées du niveau d'accès indirect du pointeur. Par exemple, pour un pointeur $**ptr$ les variables $(1)p$ et $(2)p$ seront introduites. L'utilisation de l'annotation (-1) représente l'adresse d'une variable, ce qui permet de représenter une variable dont l'adresse est copiée.

Ainsi, la définition de $RIN(n)_C$ de l'algorithme présenté précédemment doit être modifiée pour tenir compte de ces nouveaux types de variables.

$$\begin{aligned} RIN_C(n) \;=\; & \{V|n=i\} \cup \{USE(n)|MOD(n) \cap RIN_C(IMS(n)) \neq \emptyset\} \cup \\ & \{MOD(n) \cap RIN_C(IMS(n))|DUMMY(v) ou ARRAY(v)\} \\ & \cup \{RIN_C(IMS(n)) - MOD(n)\} \end{aligned}$$

avec

$$DUMMY(v) \;=\; \begin{cases} vrai & \text{si v est une variable bidon} \\ faux & \text{sinon} \end{cases}$$

$$ARRAY(v) \;=\; \begin{cases} vrai & \text{si v est un tableau} \\ faux & \text{sinon} \end{cases}$$

Ainsi pour l'exemple précédent, nous obtenons les RIN_C^0 suivants :

```
1  *(p+i) = c1;                {p,(1)p,i,j}
2  *(p+j) = c2;                {p,(1)p,i,j}
3  if (e)                      {p,(1)p,i,j}
4          k = i;              {p,(1)p,i}
5  else k = j;                 {p,(1)p,j}
6  x = *(p+k);                 {p,(1)p,k}
7  printf("%d", x);            {x}
```

ce qui donne :

$$B_C^0 = \{3\}$$

$$S_C^0 = \{1, 2, 4, 5, 6\}$$

Maintenant, le RIN_C^1, avec $BC(3) =< 3, \{e\} >$ donne :

```
1  *(p+i) = c1;                {e,p,(1)p,i,j}
2  *(p+j) = c2;                {e,p,(1)p,i,j}
3  if (e)                      {e,p,(1)p,i,j}
4          k = i;              {p,(1)p,i}
5  else k = j;                 {p,(1)p,j}
6  x = *(p+k);                 {p,(1)p,k}
7  printf("%d", x);            {x}
```

Ce qui donne la tranche suivante :

$$S_C^1 = \{1, 2, 3, 4, 5, 6\}$$

En réexécutant l'algorithme une autre fois, nous nous apercevons que nous avons atteint le plus petit point fixe, la tranche du programme pour le critère $< 7, x >$ est donc :

$$S_C = \{1, 2, 3, 4, 5, 6\}$$

3.5 Conclusion

Ainsi, dans ce chapitre, nous avons montré que les auteurs de [16] ont présenté une adaptation de l'algorithme original de coupe de Weiser [30] permettant de prendre en compte les tableaux. La méthode que nous proposons dans le prochain chapitre, sans en être directement inspirée, présente certaines similarités avec celle-ci.

Chapitre 4

Conversion de programmes contenant des tableaux et des entrées/sorties

4.1 Introduction

L'approche de traduction de l'impératif au déclaratif présenté précédemment constitue un excellent fondement pour simplifier la maintenance de programmes. Dans cette optique, le travail présenté dans le présent chapitre constitue une extension de cette approche. Premièrement, certains ajustements lui seront apportés dans le but de simplifier les programmes écrits dans le langage $\mathcal{L}_2$. Ensuite, des notions de tableaux, d'entrées et de sorties lui seront ajoutées. En effet, puisque ces derniers sont présents dans les langages impératifs actuels, il est important de les traiter pour enrichir l'ensemble des programmes pouvant être traduits. Ainsi, tel que nous l'avons présenté [13], un concept permettant de les représenter et de les utiliser doit exister dans le langage déclaratif en plus d'une fonction de traduction pour permettre de passer de l'impératif au déclaratif.

4.2 Adaptations au langage déclaratif $\mathcal{L}_2$

Tel que mentionné plus tôt, le langage déclaratif original présenté à la section 2.3 (page 26) simplifie la maintenance de programmes comparativement à un langage impératif. Un de ses avantages, est que les définitions de variables sont totalement

indépendantes les unes des autres. Cependant, cette notion peut aussi s'avérer un inconvénient. En effet, il est possible qu'une partie de différentes définitions soit identique. Dans un tel cas, il serait intéressant de pouvoir l'isoler dans une autre définition et de la référencer. Par exemple, le programme suivant :

$$\left\{ \begin{array}{l} x \mapsto \mu^{x}_{(i<4),[i\mapsto i+1,x\mapsto x*x]}[i \mapsto 0, x \mapsto 5], \\ y \mapsto \mu^{y}_{(i<5),[i\mapsto i+1,y\mapsto y+x]}[i \mapsto 0, y \mapsto 3, x \mapsto \mu^{x}_{(i<4),[i\mapsto i+1,x\mapsto x*x]}[i \mapsto 0, x \mapsto 5]] \end{array} \right\}$$

serait plus simple à écrire si une notion de dépendance entre les variables existait. Ainsi sa définition pourrait être :

$$\left\{ \begin{array}{l} x \mapsto \mu^{x}_{(i<4),[i\mapsto i+1,x\mapsto x*x]}[i \mapsto 0, x \mapsto 5], \\ y \mapsto \mu^{y}_{(i<5),[i\mapsto i+1,y\mapsto y+x]}[i \mapsto 0, y \mapsto 3] \end{array} \right\}$$

Cette nouvelle version offre plusieurs avantages parmi lesquels : meilleures comprehension et maintenance de programmes ; une mise à jour de la définition de x ne requiert pas la mise à jour de plusieurs portions (identiques) d'un programme donné appartenant à $\mathcal{L}_2$; etc.

Ainsi, cette section présente une adaptation à la sémantique du langage $\mathcal{L}_2$ d'origine ainsi qu'à la fonction de traduction pour permettre la dépendance entre les définitions.

4.2.1 Adaptation à la Sémantique

La notion d'indépendance des variables définies dans la langage $\mathcal{L}_2$ (présenté dans le chapitre 2) provient de la sémantique de l'union de programmes ($P_1 \cup P_2$). En effet, celle-ci est définie comme suit :

$$[\![P_1 \cup P_2]\!]_{\mathcal{L}_2}(s) \;=\; [\![P_1]\!]_{\mathcal{L}_2}(s) \cup [\![P_2]\!]_{\mathcal{L}_2}(s)$$

Dans cette définition, il est possible de remarquer que la sémantique associée aux sous-programmes P_1 et P_2 est évaluée en considérant la même mémoire s. Pour permettre à une définition de référencer une autre variable, la sémantique de l'union de programmes doit être définie comme suit :

$$[\![P_1 \cup P_2]\!]_{\mathcal{L}_2}(s) \;=\; [\![P_1]\!]_{\mathcal{L}_2}([\![P_2]\!]_{\mathcal{L}_2}(s)) \text{ si } P_2 \sqsubseteq P_1$$

Cette définition utilise une relation d'ordre[1] $\sqsubseteq$ sur les programmes $P \in \mathcal{L}_2$. Cette relation est définie comme suit :

$$P_2 \sqsubseteq P_1 \;\text{ ssi }\; Use(P_2) \cap Def(P_1) \;=\; \emptyset$$

[1] Réflexive, transitive et antisymétrique.

Les fonctions Def, calculant l'ensemble des variables définies dans un programme, et Use, calculant l'ensemble des variables utilisées dans un programme, sont définies dans les tableaux 4.1 et 4.2.

$$\begin{array}{rcl} Def(x = a) & = & \{x\} \\ Def(P_1; P_2) & = & Def(P_1) \cup Def(P_2) \end{array}$$

TAB. 4.1 – Ensemble des variables définies P.

$$\begin{array}{rcl} Use(x \mapsto a) & = & Use(a) \\ Use(P_1; P_2) & = & Use(P_1) \cup Use(P_2) \\ \\ Use(n) & = & \emptyset \\ Use(x) & = & \{x\} \\ Use(a_1 \ op \ a_2) & = & Use(a_1) \cup Use(a_2) \\ Use(if(b, a_1, a_2)) & = & Use(b) \cup Use(a_1) \cup Use(a_2) \\ Use(\mu^x_{b,P} \, P') & = & Use(b) \cup Use(P') \cup (Use(P) - Def(P')) \\ \\ Use(true) & = & \emptyset \\ Use(\neg b) & = & Use(b) \\ Use(b_1 \vee b_2) & = & Use(b_1) \cup Use(b_2) \\ Use(a_1 = a_2) & = & Use(a_1) \cup Use(a_2) \\ Use(P_1; P_2) & = & Use(P_1) \cup Use(P_2) \\ \\ \text{avec} & & op \in \{+, *, <, =\} \end{array}$$

TAB. 4.2 – Ensemble des variables utilisées dans P.

Cependant, comme la relation d'ordre implique qu'il ne doit pas y avoir de dépendances circulaires entre les variables, la sémantique de l'union de deux programmes telle que vue à la page 28 doit être utilisée pour les boucles. Par exemple, le programme suivant introduit des dépendances circulaires :

$$\{x \mapsto \mu^x_{(x<30),[x \mapsto y+3, y \mapsto x+4]}[x \mapsto 5, y \mapsto 6]\}$$

4.2.2 Fonction de traduction

Suite à cet ajustement de la sémantique, la fonction de traduction doit aussi être adaptée. Sa nouvelle définition est présentée dans les tableaux 4.3 et 4.4.

$$
\begin{array}{rcl}
\mathcal{D}(x = a) & = & \{x \mapsto a\} \\
\mathcal{D}(P_1; P_2) & = & \bigcup_{x \in (V(P_1) - V(P_2))} \{x \mapsto D(P_1)(x)\} \cup \\
 & & \bigcup_{x \in (V(P_1) \cap V(P_2))} \{x \mapsto (D(P_1)/x)(D(P_2)(x))\} \cup \\
 & & \bigcup_{x \in (V(P_2) - V(P_1))} \{x \mapsto M(((D(P_1))/y)(D(P_2)(x)), z)\} \\
 & & \text{avec } y \in (V(P_1) \cap V(P_2)) \\
 & & \text{et } z \in (V(P_1) - V(P_2)) \\
\mathcal{D}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) & = & \bigcup_{x \in V(P_1;P_2)} \{x \mapsto if(b, \mathcal{D}'(P_1)(x), \mathcal{D}'(P_2)(x))\} \\
\mathcal{D}(\textbf{while } b \textbf{ do } P) & = & \bigcup_{x \in V(P)} \{x \mapsto \mu^x_{b, \mathcal{D}'(P)}\}
\end{array}
$$

TAB. 4.3 – $\mathcal{D}$ - La fonction de traduction de $\mathcal{L}_1$ à $\mathcal{L}_2$.

$$
\begin{array}{rcl}
\mathcal{D}'(P_1; P_2) & = & \mathcal{D}'(P_2) \circ \mathcal{D}'(P_1) \\
 & = & \bigcup_{x \in V(P_2)} \{(x \mapsto (\mathcal{D}'(P_2)(x)))D'(P_1)\} \cup \\
 & & \bigcup_{x \in (V(P_1) - V(P_2))} \{x \mapsto D'(P_1)(x)\} \\
\mathcal{D}'(P) & = & \mathcal{D}(P)
\end{array}
$$

TAB. 4.4 – La fonction de traduction $\mathcal{D}'$.

La fonction $M(e, z)$, où e est une expression du langage $\mathcal{L}_2$ et z un ensemble de noms de variable a été défini tel que le montre le tableau 4.5. L'idée derière cette fonction est

de marquer les variables pour lesquelles il n'est pas nécessaire d'effectuer l'application. L'idée intuitive derrière la traduction de la séquence est que la fonction de traduction ne fait la substitution que lorsque nécessaire, d'où la nécessité de $y \in (V(P_1) \cap V(P_2))$. Cependant, pour s'assurer que les variables qui n'ont pas été substituées à cette étape ne le soient à une autre, il est nécessaire de les identifier. Par conséquent,l'application, tel que le montre le tableau 4.6 a été ajustée pour tenir compte de cet élément. Il est à noter que l'identification des variables à ne pas substituer n'est utilisée que pour la traduction. Par conséquent, les exposants m et z peuvent être retirés du résultat de la traduction.

$$
\begin{array}{rcl}
M(c,z) & = & c \\
M(x,z) & = & \begin{cases} x^m & \text{si } x \in z \\ x & \text{sinon} \end{cases} \\
M(\neg b,z) & = & \neg(M(b,z)) \\
M(a_1\ op\ a_2,z) & = & M(a_1,z)\ op\ M(a_2,z) \\
M(if(b,a_1,a_2),z) & = & if(M(b,z),M(a_1,z),M(a_2,z)) \\
M(\mu^x_{b,P_1}[x'_1 \mapsto a'_1,\ldots,x'_m \mapsto a'_m]^{z'},z) & = & \mu^x_{b,P_1} \dagger [x'_1 \mapsto M(a'_1),\ldots,x'_m \mapsto M(a'_m)]^{z' \cup z} \\
\text{avec } c & \in & \{n, true, false\} \\
op & \in & \{+,*,=,<,\vee\} \\
P_1 & = & \{x_1 \mapsto a_1,\ldots,x_n \mapsto a_n\}
\end{array}
$$

Tab. 4.5 – La fonction d'identification des variables à ne pas substituer.

L'exemple de la traduction du programme P suivant montre bien la différence entre la fonction d'origine et celle-ci.

```
1 x = 5;
2 y = 6;
3 z = x + y;
4 y = 0;
```

$$
\begin{array}{rcl}
P(c) & = & c \\
P(x) & = & \begin{cases} a & \text{si } [x \mapsto a] \in P \\ x & \text{sinon} \end{cases} \\
P(x^m) & = & x^m \\
P(\neg b) & = & \neg(P(b)) \\
P(a_1 \ op \ a_2) & = & P(a_1) \ op \ P(a_2) \\
P(if(b, a_1, a_2)) & = & if(P(b), P(a_1), P(a_2)) \\
P(\mu^x_{b,P_1}[x'_1 \mapsto a'_1, \ldots, x'_m \mapsto a'_m]^z) & = & \mu^x_{b,P_1} P/z \dagger [x'_1 \mapsto P(a'_1), \ldots, x'_m \mapsto P(a'_m)] \\
P(x \mapsto a_1) & = & x \mapsto P(a_1) \\
\text{avec } c & \in & \{n, true, false\} \\
op & \in & \{+, *, =, <, \vee\} \\
P_1 & = & [x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]
\end{array}
$$

TAB. 4.6 – Application.

Avec P_1 représentant les lignes 2 à 4 du programme P :

$$
\begin{array}{rcl}
\mathcal{D}(P) & = & \{x \mapsto \mathcal{D}(x = 5)(x)\} \cup \\
 & & \bigcup_{x \in \{y,z\}} \{x \mapsto (\mathcal{D}(P_1)(x))\}
\end{array}
$$

Avec P_2 représentant les lignes 3 et 4 du programme P :

$$
\begin{array}{rcl}
\mathcal{D}(P_1) & = & \{y \mapsto M(((D(y = 6))/\{y\})(D(P_2)(y)), \{\})\} \cup \\
 & & \{z \mapsto M(((D(y = 6))/\{y\})(D(P_2)(z)), \{\})\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}(P_2) & = & \{z \mapsto \mathcal{D}(z = x + y)(z)\} \cup \\
 & & \{y \mapsto \mathcal{D}(y = 0)(y)\} \\
 & = & \{y \mapsto 0, z \mapsto x + y\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}(P_1) & = & \{y \mapsto ([y \mapsto 6])(0)\} \cup \\
 & & \{z \mapsto (\{y \mapsto 6\})(x + y)\} \\
 & = & \{y \mapsto 0, z \mapsto x + 6\}
\end{array}
$$

$$\begin{aligned}\mathcal{D}(P) &= \{x \mapsto 5\} \cup \\ &\quad \{y \mapsto 0, z \mapsto x + 6\} \\ &= \left\{ \begin{array}{l} x \mapsto 5, \\ y \mapsto 0, \\ z \mapsto x + 6 \end{array} \right\}\end{aligned}$$

Comparativement, si on avait traduit le même programme avec l'ancienne fonction de traduction, le résultat obtenu serait :

$$\left\{ \begin{array}{l} x \mapsto 5, \\ y \mapsto 0, \\ z \mapsto 5 + 6 \end{array} \right\}$$

4.3 Langage impératif $\mathcal{L}_1$

Le langage source utilisé est une extension du langage décrit précédemment, sans toutefois inclure tous les éléments manquants pour en faire un langage complet. En effet, plusieurs éléments comme les appels de fonctions ne sont pas présents dans cette nouvelle version du langage $\mathcal{L}_1$. La syntaxe des nouveaux éléments est basée sur celle que l'on retrouve dans l'article "Points-to Analysis by Type Inference of Programs with Structures and Unions" [29].

4.3.1 Syntaxe

En plus des opérations arithmétiques, des expressions booléennes, de l'affectation, de la séquence d'instructions et de quelques structures de contrôle, le langage $\mathcal{L}_1$ permet aussi de traiter les espaces mémoire. En effet, sa syntaxe comprend des expressions permettant l'allocation et la libération d'espaces mémoire en plus de la mise à jour de ses éléments. L'expression $x = allocate(a)$ assigne à x un bloc de mémoire de v éléments (v étant égal à l'évaluation de l'expression a). Pour accéder à ces items, les crochets sont utilisés ; par exemple, $x[2]$ permet d'accéder au deuxième élément du bloc x. Évidemment, la mémoire doit aussi pouvoir être libérée, c'est le rôle que remplit l'instruction $free(x)$ qui libère le bloc x.

Les entrées et sorties sont deux instructions du langage $\mathcal{L}_1$. La première $input(x)$ permet de recevoir une entrée de l'utilisateur et de la stocker dans une variable x. En ce qui a trait aux sorties ($output(x)$), elles permettent d'afficher le contenu de la

variable x. Il est à noter qu'une variable utilisée en entrée ne peut être utilisée pour stocker d'autres types de données ; seulement des entrées. Par exemple, si la variable x est utilisée en entrée, elle ne pourra être par la suite utilisée pour contenir la valeur de $y+z$. Cependant, elle pourrait être utilisée à nouveau pour stocker une nouvelle entrée.

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid x[a_1]...[a_n] \mid a_1 + a_2 \mid a_1 * a_2 \\
b & ::= & true \mid \neg b \mid b_1 \vee b_2 \mid a_1 < a_2 \mid a_1 = a_2 \\
P & ::= & x = a \\
 & \mid & x[a_1]...[a_n] = a \\
 & \mid & x = allocate(a) \\
 & \mid & free(x) \\
 & \mid & output(x) \\
 & \mid & input(x) \\
 & \mid & \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \\
 & \mid & \textbf{while } b \textbf{ do } P \\
 & \mid & P_1; P_2
\end{array}
$$

TAB. 4.7 – Syntaxe du langage $\mathcal{L}_1$.

Le programme suivant montre l'utilisation de la condition (if) avec un tableau. Il commence par créer un tableau de trois éléments puis utilise le premier élément dans la condition et met à jour un des deux autres éléments selon le cas.

```
1 x = allocate(3);
2 x[1] = 4;
3 x[2] = 5;
4 x[3] = 6;
5 if x[1] < 5 then
6       x[2] = 8;
7 else
8       x[3] = 9;
```

4.3.2 Sémantique

La sémantique des expressions du langage $\mathcal{L}_1$, telle que présentée plus loin dans les tableaux 4.8, 4.9 et 4.10, nécessite certains ajustements à l'environnement Γ concernant les entrées et sorties. Premièrement, pour calculer la sémantique d'un programme contenant des entrées, celles-ci doivent être connues. Ainsi, pour chaque variable utilisée en entrée, une valeur ou un tableau de valeurs (si la même variable est utilisée pour plusieurs entrées) doit être présent dans l'environnement Γ de départ. Aussi, un compteur doit exister pour identifier quelle entrée le programme est rendu à recevoir pour chaque variable. Un comportement similaire est aussi nécessaire pour les variables de sorties, à l'exception que la variable supplémentaire contenue dans l'environnement Γ est vide initialement et qu'après le calcul de la sémantique, elle contient les valeurs affichées pour chaque variable lors de l'exécution du programme.

Ainsi, malgré ces ajustements, les signatures des fonctions sémantiques $[\![\_]\!]_{\mathcal{A}_1}$, $[\![\_]\!]_{\mathcal{B}_1}$ et $[\![\_]\!]_{\mathcal{L}_1}$ restent les mêmes :

$$
\begin{aligned}
[\![\_]\!]_{\mathcal{A}_1} &: \mathcal{A}_1 \to \Gamma \to \mathbb{Z} \\
[\![\_]\!]_{\mathcal{B}_1} &: \mathcal{B}_1 \to \Gamma \to \mathit{bool} \\
[\![\_]\!]_{\mathcal{L}_1} &: \mathcal{L}_1 \to \Gamma \to \Gamma
\end{aligned}
$$

Il est à noter que la façon utilisée pour représenter les tableaux donne à chaque indice une valeur ; par exemple :

$$\langle \{1\} \mapsto 5, \{2\} \mapsto 7 \rangle$$

représente un tableau de deux éléments dont le premier a la valeur 5 et le deuxième la valeur 7. Aussi, pour simplifier la notation, les indices peuvent être regroupés lorsqu'ils correspondent à la même expression. Par exemple :

$$\langle \{1, 2\} \mapsto 8, \{3, 4, 5\} \mapsto 9 \rangle$$

représente le tableau suivant :

1	2	3	4	5
8	8	9	9	9

Aussi, voici la sémantique de l'expression $x[1] = 4$ avec une mémoire $s = [x \mapsto \left\langle \ \{1\} \mapsto 1, \{2\} \mapsto 2, \{3\} \mapsto 3 \ \right\rangle]$:

$$\begin{aligned}
[\![x[1] = 4]\!]_{\mathcal{L}_1}(s) &= s \dagger [x \mapsto \left\langle \ \{[\![1]\!]_{\mathcal{A}_1} s\} \mapsto \{[\![4]\!]_{\mathcal{A}_1}(s)\} \ \right\rangle] \\
&= s \dagger [x \mapsto \left\langle \ \{1\} \mapsto \{4\} \ \right\rangle] \\
&= [x \mapsto \left\langle \ \{1\} \mapsto 4, \{2\} \mapsto 2, \{3\} \mapsto 3 \ \right\rangle]
\end{aligned}$$

De plus, les abréviations et opérateurs suivants seront utilisés :

$$\begin{aligned}
x_{i\ldots j}\ s = v_{i\ldots j} \quad &\text{si} \quad (x \mapsto \langle \ldots \langle \ldots v_j \ldots \rangle_i \ldots \rangle) \in s \\
x\ s = v \quad &\text{si} \quad (x \mapsto v) \in s \\
d_1 \quad &= \quad \langle \{[\![a_1]\!]_{A_1} s\} \mapsto \langle \ldots \{\langle [\![a_n]\!]_{A_1} s\} \mapsto \langle [\![a']\!]_{A_1} s \ldots \rangle \rangle \\
x{+}{+} \quad &= \quad x \mapsto x + 1 \\
f_{b,P} g(s) \quad &= \quad cond([\![b]\!]_{B_1} s, g \circ [\![P]\!]_{\mathcal{L}_1} s, id)
\end{aligned}$$

$$\begin{aligned}
&[\ldots,\ x \mapsto \langle \{w\} \mapsto t \rangle] \dagger [x \mapsto \langle \{y\} \mapsto z \rangle] \\
= \quad &\begin{cases} [\ldots,\ x \mapsto \langle \{y\} \mapsto z \rangle] & \text{si } w = y \\ \left[\ldots,\ x \mapsto \left\langle \begin{array}{l} \{w\} \mapsto t, \\ \{y\} \mapsto z \end{array} \right\rangle \right] & \text{si } w \neq y \end{cases}
\end{aligned}$$

$$\begin{aligned}
[\![n]\!]_{A_1}(s) &= n \\
[\![x]\!]_{A_1}(s) &= s(x) \\
[\![x[a_1]...[a_n]]\!]_{A_1}(s) &= x_{[\![a_1]\!]_{A_1}(s)\ldots[\![a_n]\!]_{A_1}(s)}\ (s) \\
[\![a_1 + a_2]\!]_{A_1}(s) &= [\![a_1]\!]_{A_1}(s) + [\![a_2]\!]_{A_1}(s) \\
[\![a_1 * a_2]\!]_{A_1}(s) &= [\![a_1]\!]_{A_1}(s) * [\![a_2]\!]_{A_1}(s)
\end{aligned}$$

TAB. 4.8 – Sémantique des expressions arithmétiques du langage $\mathcal{L}_1$.

$$
\begin{array}{rcl}
[\![true]\!]_{\mathcal{B}_1}(s) & = & true \\
[\![\neg b]\!]_{\mathcal{B}_1}(s) & = & \neg[\![b]\!]_{\mathcal{B}_1}(s) \\
[\![b_1 \vee b_2]\!]_{\mathcal{B}_1}(s) & = & [\![b_1]\!]_{\mathcal{B}_1}(s) \vee [\![b_2]\!]_{\mathcal{B}_1}(s) \\
[\![a_1 < a_2]\!]_{\mathcal{B}_1}(s) & = & [\![a_1]\!]_{\mathcal{A}_1}(s) < [\![a_2]\!]_{\mathcal{A}_1}(s) \\
[\![a_1 = a_2]\!]_{\mathcal{B}_1}(s) & = & [\![a_1]\!]_{\mathcal{A}_1}(s) = [\![a_2]\!]_{\mathcal{A}_1}(s)
\end{array}
$$

TAB. 4.9 – Sémantique des expressions booléennes du langage $\mathcal{L}_1$.

$$
\begin{array}{rcl}
[\![x = a]\!]_{\mathcal{L}_1}(s) & = & s \dagger [x \mapsto [\![a]\!]_{A_1}(s)] \\
[\![x[a_1]...[a_n] = a']\!]_{\mathcal{L}_1}(s) & = & s \dagger [x \mapsto d_1] \\
[\![x = allocate(y)]\!]_{\mathcal{L}_1}(s) & = & s \dagger [x \mapsto \langle \{1, \ldots, [\![y]\!]_{A_1}(s)\} \mapsto null \rangle] \\
[\![free(x)]\!]_{\mathcal{L}_1}(s) & = & s_1 \text{ si } s = s_1 \dagger [x \mapsto \langle \ldots \rangle] \\
[\![output(x)]\!]_{\mathcal{L}_1}(s) & = & s \dagger [x_O[x_{Oc}] \mapsto [\![x]\!]_{A_1}(s),\ x_{Oc}{+}{+}] \\
[\![input(x)]\!]_{\mathcal{L}_1}(s) & = & s \dagger [x \mapsto x_I[x_{Ic}],\ x_{Ic}{+}{+}] \\
[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]_{\mathcal{L}_1}(s) & = & cond([\![b]\!]_{\mathcal{B}_1} s, [\![P_1]\!]_{\mathcal{L}_1} s, [\![P_2]\!]_{\mathcal{L}_1}(s)) \\
[\![\textbf{while } b \textbf{ do } P]\!]_{\mathcal{L}_1}(s) & = & \mu f_{b,P} g(s) \\
[\![P_1; P_2]\!]_{\mathcal{L}_1}(s) & = & ([\![P_2]\!]_{\mathcal{L}_1} \circ [\![P_1]\!]_{\mathcal{L}_1})(s)
\end{array}
$$

TAB. 4.10 – Sémantique du langage $\mathcal{L}_1$.

Pour ce qui est de la sémantique des entrées ($input(x)$), elle définit la valeur de x comme étant la valeur courante entrée par l'utilisateur. Pour déterminer quelle est la valeur courante, le compteur x_{Ic} est utilisé. Ce dernier est incrémenté à chaque fois qu'une instruction d'entrée de la variable x est rencontrée.

Le même raisonnement s'applique pour les sorties ($output(x)$) à l'exception que la valeur de la variable x est stockée dans la variable de sortie x_O à l'index correspondant au compteur x_{Oc}.

Dans le cas de l'affectation d'une expression arithmétique à un élément d'un tableau ($x[a_1] \ldots [a_n] = a'$) sa sémantique est l'affectation du résultat de l'évaluation de l'expression arithmétique de droite (a') à la case du tableau x dont l'indice correspond à la valeur des expressions arithmétiques entre crochets ($a_1 \ldots a_n$). Étant donné que ces expressions arithmétiques peuvent être des entrées, la sémantique avec un tableau à une dimension[2] est :

[2]Pour un tableau à n dimensions, la sémantique se déduit naturellement.

$$[\![x[a_1] = a_2]\!]_{\mathcal{L}_1}(s) \quad = \quad s \dagger [x \mapsto \langle \{[\![a_1]\!]_{\mathcal{A}_1}(s)\} \mapsto [\![a_2]\!]_{\mathcal{A}_1}(s)\rangle]$$

La sémantique de l'allocation de mémoire est relativement simple ; elle assigne à une variable un tableau d'un certain nombre d'éléments (a) ayant tous la valeur initiale *null*.

$$[\![x = allocate(a)]\!]_{\mathcal{L}_1}(s) \quad = \quad s \dagger [x \mapsto \langle \{1 \ldots [\![a]\!]_{\mathcal{A}_1}(s)\} \mapsto null\rangle]$$

La sémantique de *free*, quant à elle, libère la mémoire du tableau désigné. Par exemple, avec $s = [x \mapsto \left\langle \ \{1,2\} \mapsto 5, \{3\} \mapsto 7 \ \right\rangle]$, la sémantique de $free(x)$ donne :

$$[\![free(x)]\!]_{\mathcal{L}_1}(s) \quad = \quad []$$

Ensuite, la sémantique des instructions "if" et "$while$" est similaire à celle précédemment définie.

$$\begin{array}{rcl} cond & : & (bool \times (\mathcal{L}_1 \to \Gamma \to \Gamma) \times \\ & & (\mathcal{L}_1 \to \Gamma \to \Gamma)) \to \\ & & (\mathcal{L}_1 \to \Gamma \to \Gamma) \\ f_{(b,P)} & : & (\mathcal{L}_1 \to \Gamma \to \Gamma) \to (\mathcal{L}_1 \to \Gamma \to \Gamma) \end{array}$$

Pour tout environnement s dans Γ, nous avons :

$$cond(b, f, g)(s) = \begin{cases} f(s) & \text{si } b = true \\ g(s) & \text{si } b = false \end{cases}$$

Pour $g : \Gamma \to \Gamma$ et pour tout environnement s dans Γ, nous avons :

$$(f_{(b,P)}(g))(s) \quad = \quad \begin{cases} s & \text{si } b = false \\ g([\![P]\!]_{\mathcal{L}_1}(s)) & \text{si } b = true \end{cases}$$

4.3.3 Exemples

Cette section illustre des exemples de programmes écrits dans le langage $\mathcal{L}_1$. Ceux-ci visent à montrer l'utilisation du langage avec les tableaux.

Exemple 1

Cet exemple utilise le programme présenté dans la section 4.3.1, soit :

```
1 x = allocate(3);
2 x[1] = 4;
3 x[2] = 5;
4 x[3] = 6;
5 if x[1] < 5 then
6       x[2] = 8;
7 else
8       x[3] = 9;
```

La sémantique de cet exemple, avec la mémoire $s = []$, est :

Avec P_1 correspondant aux lignes 2 à 8 du programme P.

$$\begin{aligned}
[\![P]\!]_{\mathcal{L}_1}(s) &= ([\![P_1]\!]_{\mathcal{L}_1} \circ [\![x = allocate(3)]\!]_{\mathcal{L}_1})(s) \\
&= [\![P_1]\!]_{\mathcal{L}_1}([\![x = allocate(3)]\!]_{\mathcal{L}_1}(s)) \\
&= [\![P_1]\!]_{\mathcal{L}_1}[x \mapsto \langle \{1,2,3\} \mapsto null \rangle]
\end{aligned}$$

Avec P_2 correspondant aux lignes 3 à 8 du programme P et $s_1 = [x \mapsto \langle \{1,2,3\} \mapsto null \rangle]$.

$$\begin{aligned}
[\![P_1]\!]_{\mathcal{L}_1}(s_1) &= ([\![P_2]\!]_{\mathcal{L}_1} \circ [\![x[1] = 4]\!]_{\mathcal{L}_1})(s_1) \\
&= [\![P_2]\!]_{\mathcal{L}_1}([\![x[1] = 4]\!]_{\mathcal{L}_1}(s_1)) \\
&= [\![P_2]\!]_{\mathcal{L}_1}[x \mapsto \langle \{1\} \mapsto 4, \{2,3\} \mapsto null \rangle]
\end{aligned}$$

Avec P_3 correspondant aux lignes 4 à 8 du programme P et $s_2 = [x \mapsto \langle \{1\} \mapsto 4, \{2,3\} \mapsto null \rangle]$.

$$\begin{aligned}
[\![P_2]\!]_{\mathcal{L}_1}(s_2) &= ([\![P_3]\!]_{\mathcal{L}_1} \circ [\![x[2] = 5]\!]_{\mathcal{L}_1})(s_2) \\
&= [\![P_3]\!]_{\mathcal{L}_1}([\![x[2] = 5]\!]_{\mathcal{L}_1}(s_2)) \\
&= [\![P_3]\!]_{\mathcal{L}_1} \left[x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 4, \\ \{2\} \mapsto 5, \\ \{3\} \mapsto null \end{array} \right\rangle \right]
\end{aligned}$$

Avec P_4 correspondant aux lignes 5 à 8 du programme P et

$$s_3 = \left[x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 4, \\ \{2\} \mapsto 5, \\ \{3\} \mapsto null \end{array} \right\rangle \right]$$

.

$$\begin{aligned} [\![P_3]\!]_{\mathcal{L}_1}(s_3) &= ([\![P_3]\!]_{\mathcal{L}_1} \circ [\![x[3] = 6]\!]_{\mathcal{L}_1})(s_3) \\ &= [\![P_3]\!]_{\mathcal{L}_1}([\![x[3] = 6]\!]_{\mathcal{L}_1}(s_3)) \\ &= [\![P_3]\!]_{\mathcal{L}_1} \left[x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 4, \\ \{2\} \mapsto 5, \\ \{3\} \mapsto 6 \end{array} \right\rangle \right] \end{aligned}$$

Avec $s_4 = \left[x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 4, \\ \{2\} \mapsto 5, \\ \{3\} \mapsto 6 \end{array} \right\rangle \right]$

$$\begin{aligned} [\![P_4]\!]_{\mathcal{L}_1}(s_4) &= cond([\![x[1] < 5]\!]_{B_1}(s_4), [\![x[2] = 8]\!]_{\mathcal{L}_1}(s_4), [\![x[3] = 9]\!]_{\mathcal{L}_1}(s_4)) \\ &= cond([\![x[1] < 5]\!]_{B_1}(s_4), s_4 \dagger [x \mapsto \langle \{2\} \mapsto 8 \rangle], s_4 \dagger [x \mapsto \langle \{3\} \mapsto 9 \rangle]) \\ &= \begin{cases} s_4 \dagger [x \mapsto \langle \{2\} \mapsto 8 \rangle] & \text{si } [\![x[1] < 5]\!]_{B_1}(s_4) = true \\ s_4 \dagger [x \mapsto \langle \{3\} \mapsto 9 \rangle] & \text{si } [\![x[1] < 5]\!]_{B_1}(s_4) = false \end{cases} \\ &= \begin{cases} s_4 \dagger [x \mapsto \langle \{2\} \mapsto 8 \rangle] & \text{si } (4 < 5) = true \\ s_4 \dagger [x \mapsto \langle \{3\} \mapsto 9 \rangle] & \text{si } (4 < 5) = false \end{cases} \\ &= s_4 \dagger [x \mapsto \langle \{2\} \mapsto 8 \rangle] \\ &= \left[x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 4, \\ \{2\} \mapsto 8, \\ \{3\} \mapsto 6 \end{array} \right\rangle \right] \end{aligned}$$

Donc :

$$[\![P]\!]_{\mathcal{L}_1}(s) = \left[x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 4, \\ \{2\} \mapsto 8, \\ \{3\} \mapsto 6 \end{array} \right\rangle \right]$$

Exemple 2

Voici un exemple de programme utilisant les tableaux et les boucles. Il crée un tableau de dix éléments, les initialise à la valeur de leur position puis met la valeur 0 dans le premier élément. Tout comme pour l'exemple précédent, une mémoire s vide sera utilisée pour calculer la sémantique.

```
1 x = allocate(10);
2 i=1;
3 while i < 11 do
4       x[i] = i;
5       i = i+1;
6 x[1] = 0;
```

Avec P_1 correspondant aux lignes 2 à 6 du programme P.

$$\begin{aligned}
[\![P]\!]_{\mathcal{L}_1}(s) &= ([\![P_1]\!]_{\mathcal{L}_1} \circ [\![x = allocate(10)]\!]_{\mathcal{L}_1})(s) \\
&= [\![P_1]\!]_{\mathcal{L}_1}([\![x = allocate(10)]\!]_{\mathcal{L}_1}(s)) \\
&= [\![P_1]\!]_{\mathcal{L}_1} \left[x \mapsto \left\langle \{1, \ldots, 10\} \mapsto null \right\rangle \right]
\end{aligned}$$

Avec P_2 correspondant aux lignes 3 à 6 du programme P et

$$s_1 = \left[x \mapsto \left\langle \{1, \ldots, 10\} \mapsto null \right\rangle \right]$$

.

$$\begin{aligned}
[\![P_1]\!]_{\mathcal{L}_1}(s_1) &= ([\![P_2]\!]_{\mathcal{L}_1} \circ [\![i = 1]\!]_{\mathcal{L}_1})(s_1) \\
&= [\![P_2]\!]_{\mathcal{L}_1}([\![i = 1]\!]_{\mathcal{L}_1}(s_1)) \\
&= [\![P_2]\!]_{\mathcal{L}_1} \left[x \mapsto \left\langle \{1, \ldots, 10\} \mapsto null \right\rangle, i \mapsto 1 \right]
\end{aligned}$$

Avec P_3 correspondant aux lignes 3 à 5 du programme P et

$$s_2 = \left[x \mapsto \left\langle \{1, \ldots, 10\} \mapsto null \right\rangle, i \mapsto 1 \right]$$

.

$$\begin{aligned}
[\![P_2]\!]_{\mathcal{L}_1}(s_2) &= ([\![x[1] = 0]\!]_{\mathcal{L}_1} \circ [\![P_3]\!]_{\mathcal{L}_1})(s_2) \\
&= [\![x[1] = 0]\!]_{\mathcal{L}_1}([\![P_3]\!]_{\mathcal{L}_1}(s_2))
\end{aligned}$$

Avec P_4 correspondant les lignes 4 et 5 du programme P, on obtient :

$$[\![P_3]\!]_{\mathcal{L}_1}(s_2) = \mu f_{(i<11),P_4} g(s_2)$$

$$f_{(i<11),P_4}g(s_2) = cond([\![i < 11]\!]_{B_1}s_2, g \circ [\![P_4]\!]_{\mathcal{L}_1}(s_2), id)$$

Le plus petit point fixe de la fonction est :

$$\mu f_{(i<11),P_4} = \bigsqcup_{y \geq 0} f^y_{(i<11),P_4}(\bot)$$

Il est possible d'obtenir le plus petit point fixe en appliquant la fonction f plusieurs fois à $(\bot)$.

$$\begin{aligned} f_{(i<11),P_4}(\bot)(s_2) &= \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ \bot(s_2) & \text{si } [\![i < 11]\!]_{B_1}(s_2) = true \end{cases} \\ &= \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ indef & \text{si } [\![i < 11]\!]_{B_1}(s_2) = true \end{cases} \end{aligned}$$

$$\begin{aligned} & f^2_{(i<11),P_4}(\bot)(s_2) \\ = \; & f_{(i<11),P_4}(f_{(i<11),P_4}(\bot))(s_2) \\ = \; & \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ f_{(i<11),P_4}(\bot)[\![P_4]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![i < 11]\!]_{B_1}(s_2) = true \end{cases} \\ = \; & \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ [\![P_4]\!]_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]_{\mathcal{L}_1}(s_2) = false \end{cases} \\ indef & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]_{\mathcal{L}_1}(s_2) = true \end{cases} \end{cases} \\ = \; & \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ \left[\!\!\left[\begin{array}{l} x[i] = i; \\ i = i + 1 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i - 1 < 11]\!]_{B_1}(s_2) = false \end{cases} \\ indef & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i - 1 < 11]\!]_{B_1}(s_2) = true \end{cases} \end{cases} \end{aligned}$$

$$= \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ \left[\!\!\left[\begin{array}{l} x[10] = 10; \\ i = 11 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![i = 10]\!]_{B_1}(s_2) = true \\ indef & \text{si } [\![i < 10]\!]_{B_1}(s_2) = true \end{cases}$$

$$\begin{array}{ll} & f'^{3}_{(i<11),P_4}(\bot)(s_2) \\ = & f'_{(i<11),P_4}(f'^{2}_{(i<11),P_4}(\bot))(s_2) \\ = & \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ f_{(i<11),P_4}(f_{(i<11),P_4}(\bot))[\![P_4]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![i < 11]\!]_{B_1}(s_2) = true \end{cases} \\ = & \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ [\![P_4]\!]_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]_{\mathcal{L}_1}(s_2) = false \end{cases} \\ f_{(i<11),P_4}(\bot)[\![P_4]\!]^2_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]_{\mathcal{L}_1}(s_2) = true \end{cases} \end{cases} \\ = & \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ [\![P_4]\!]_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]_{\mathcal{L}_1}(s_2) = false \end{cases} \\ [\![P_4]\!]^2_{\mathcal{L}_1}(s_2) & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]_{\mathcal{L}_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]^2_{\mathcal{L}_1}(s_2) = false \end{cases} \\ indef & \text{si } \begin{cases} [\![i < 11]\!]_{B_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]_{\mathcal{L}_1}(s_2) = true \wedge \\ [\![i < 11]\!]_{B_1}[\![P_4]\!]^2_{\mathcal{L}_1}(s_2) = true \end{cases} \end{cases} \\ = & \begin{cases} s_2 & \text{si } [\![i < 11]\!]_{B_1}(s_2) = false \\ \left[\!\!\left[\begin{array}{l} x[10] = 10; \\ i = 11 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![i = 10]\!]_{B_1}(s_2) = true \\ \left[\!\!\left[\begin{array}{l} x[9] = 9; \\ x[10] = 10, \\ i = 11 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![i = 9]\!]_{B_1}(s_2) = true \\ indef & \text{si } [\![i < 9]\!]_{B_1}(s_2) = true \end{cases} \end{array}$$

Le plus petit point fixe de la fonction $f_{(i<n),P_4}$ est donc :

$$
\begin{array}{ll}
 & \mu f_{(i<11),P_4}(s_2) \\
= & \bigsqcup_{y\geq 0}\{f^y_{(i<11),P_4}(\bot)\}(s_2) \\
= & \begin{cases}
s_2 & \text{si } [\![i<11]\!]_{B_1}(s_2)=false \\
[\![P_4]\!]_{\mathcal{L}_1}(s_2) & \text{si } [\![i=10]\!]_{B_1}(s_2)=true \\
[\![P_4]\!]^2_{\mathcal{L}_1}(s_2) & \text{si } [\![i=9]\!]_{B_1}(s_2)=true \\
\vdots & \\
[\![P_4]\!]^n_{\mathcal{L}_1}(s_2) & \text{si } [\![i=11-n]\!]_{B_1}(s_2)=true \\
indef & \text{si } [\![i<11-n]\!]_{B_1}(s_2)=true
\end{cases} \\
= & \begin{cases}
s_2 & \text{si } [\![i<11]\!]_{B_1}(s_2)=false \\
\left[\!\!\left[\begin{array}{l} x[10]=10; \\ i=11 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![i=10]\!]_{B_1}(s_2)=true \\
\left[\!\!\left[\begin{array}{l} x[9]=9; \\ x[10]=10; \\ i=11 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![i=9]\!]_{B_1}(s_2)=true \\
\vdots & \\
\left[\!\!\left[\begin{array}{l} x[11-n]=[11-n]; \\ \vdots \\ x[9]=9; \\ x[10]=10 \\ i=11 \end{array} \right]\!\!\right]_{\mathcal{L}_1}(s_2) & \text{si } [\![i=11-n]\!]_{B_1}(s_2)=true \\
indef & \text{si } [\![i<11-n]\!]_{B_1}(s_2)=true
\end{cases}
\end{array}
$$

Avec $s_2 = \left[\; x \mapsto \left\langle \; \{1, \ldots, 10\} \mapsto null \; \right\rangle, i \mapsto 1 \; \right]$, on obtient donc :

$$[\![P_3]\!]_{\mathcal{L}_1}(s_2) = \left[\begin{array}{l} \\ \\ \\ \\ x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 1, \\ \{2\} \mapsto 2, \\ \{3\} \mapsto 3, \\ \{4\} \mapsto 4, \\ \{5\} \mapsto 5, \\ \{6\} \mapsto 6, \\ \{7\} \mapsto 7, \\ \{8\} \mapsto 8, \\ \{9\} \mapsto 9, \\ \{10\} \mapsto 10 \end{array} \right\rangle, \\ \\ \\ \\ i \mapsto 11 \end{array} \right]$$

Ainsi :

$$\begin{aligned} [\![P_2]\!]_{\mathcal{L}_1}(s_2) &= [\![x[1] = 0]\!]_{\mathcal{L}_1}([\![P_3]\!]_{\mathcal{L}_1}(s_2)) \\ &= \left[\begin{array}{l} \\ \\ \\ \\ x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 0, \\ \{2\} \mapsto 2, \\ \{3\} \mapsto 3, \\ \{4\} \mapsto 4, \\ \{5\} \mapsto 5, \\ \{6\} \mapsto 6, \\ \{7\} \mapsto 7, \\ \{8\} \mapsto 8, \\ \{9\} \mapsto 9, \\ \{10\} \mapsto 10 \end{array} \right\rangle, \\ \\ \\ \\ i \mapsto 11 \end{array} \right] \end{aligned}$$

Donc la sémantique du programme P est :

$$[\![P]\!]_{\mathcal{L}_1}(s) \quad = \quad \left[\begin{array}{l} x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 0, \\ \{2\} \mapsto 2, \\ \{3\} \mapsto 3, \\ \{4\} \mapsto 4, \\ \{5\} \mapsto 5, \\ \{6\} \mapsto 6, \\ \{7\} \mapsto 7, \\ \{8\} \mapsto 8, \\ \{9\} \mapsto 9, \\ \{10\} \mapsto 10 \end{array} \right\rangle, \\ i \mapsto 11 \end{array}\right]$$

Exemple 3

L'exemple suivant montre un programme plus concret qui constitue en fait une implantation du tri par insertion qui utilise un tableau $tab1$ de taille n comme source et un tableau $tab2$ comme destination.

```
1  i = 2;
2  tab2 = allocate(n);
3  tab2[1] = tab1[1];
4  while i ≤ n do
5      j = i - 1;
6      while j ≥ 1 ∧ tab2[j] > tab1[i] do
7          tab2[j + 1] = tab2[j];
8          j = j - 1;
9      tab2[j + 1] = tab1[i];
10     i = i + 1;
```

Avec une mémoire

$$s = \left[tab1 \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 3, \\ \{2\} \mapsto 1, \\ \{3\} \mapsto 2 \end{array} \right\rangle, n \mapsto 3 \right]$$

la sémantique du programme P précédent donne :

$$[\![P]\!]_{\mathcal{L}_1}(s) = \left[\begin{array}{l} tab1 \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 3, \\ \{2\} \mapsto 1, \\ \{3\} \mapsto 2 \end{array} \right\rangle, \\ n \mapsto 3, \\ i \mapsto 4, \\ j \mapsto 1, \\ tab2 \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 1, \\ \{2\} \mapsto 2, \\ \{3\} \mapsto 3 \end{array} \right\rangle \end{array}\right]$$

4.4 Langage déclaratif $\mathcal{L}_2$

Le langage déclaratif $\mathcal{L}_2$ sera utilisé comme cible de la traduction. Ainsi, parallèlement au langage $\mathcal{L}_1$, le langage $\mathcal{L}_2$ doit permettre les mêmes types d'opérations.

4.4.1 Syntaxe

La syntaxe du langage $\mathcal{L}_2$, comme présentée dans le tableau 4.11, permet entre autres de représenter les expressions arithmétiques et booléennes tout comme dans le langage original. Elle comprend aussi une façon de représenter les entrées et sorties ainsi que certaines structures de données (d) comme les tableaux.

Tout comme pour le langage $\mathcal{L}_1$, le langage $\mathcal{L}_2$ supporte maintenant les entrées et sorties. Comme il s'agit d'un langage déclaratif, les sorties sont identifiées comme une définition de variable avec un indice O puisque c'est la sortie (Output) qui est définie et non une variable de l'environnement. La variable prend la valeur de l'expression qui est émise en sortie. Il peut s'agir d'un tableau puisque plusieurs sorties peuvent être faites pour la même variable ; chaque élément du tableau représente une sortie. En ce qui a trait aux entrées (*input*), elles indiquent que la variable est utilisée comme une entrée. De cette façon, la même entrée peut être utilisée dans différentes définitions de variables.

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid x[a_1] \ldots [a_n] \mid if(b, a_1, a_2) \mid rec \\
b & ::= & true \mid \neg b \mid b_1 \vee b_2 \mid a_1 < a_2 \mid a_1 = a_2 \\
elem & ::= & \{a_1, \ldots, a_n\} \mapsto e \mid rec \mid elem, elem \\
d & ::= & \langle elem \rangle \\
e & ::= & a \mid d \\
rec & ::= & \mu^x_{b\ [x \mapsto e_1, \ldots, x_n \mapsto e_n]}[x' \mapsto e'_1, \ldots, x'_m \mapsto e'_m] \\
P & ::= & x \mapsto e \\
 & \mid & x \mapsto input \\
 & \mid & x_O \mapsto e \\
 & \mid & P_1 \cup P_2
\end{array}
$$

TAB. 4.11 – Syntaxe du langage $\mathcal{L}_2$.

Par exemple :

$$
\left\{
\begin{array}{l}
w \mapsto input, \\
x \mapsto 3 + w, \\
y \mapsto 5 * w
\end{array}
\right\}
$$

Deux notations sont utilisées pour les tableaux pour permettre de bien représenter tous les cas possibles facilement. La première est utilisée pour représenter un tableau dont les éléments sont déterminés sans dépendance aux autres. Par exemple, le tableau suivant serait représenté par l'expression $\langle \{1, \ldots, 4\} \mapsto 0, \{5, 6\} \mapsto 1 \rangle$.

1	2	3	4	5	6
0	0	0	0	1	1

La notation utilisant le point fixe (rec) quant à elle est utilisée pour représenter les tableaux dont la valeur des éléments dépend récursivement de celle des autres éléments. Un exemple de celle-ci sera donné dans la section sur la sémantique de $\mathcal{L}_2$.

4.4.2 Sémantique

Les fonctions sémantiques $[\![\_]\!]_{\mathcal{A}_2}$, $[\![\_]\!]_{\mathcal{B}_2}$ et $[\![\_]\!]_{\mathcal{L}_2}$ ont les signatures suivantes dans $\mathcal{L}_2$:

$$\begin{aligned}
[\![_]\!]_{\mathcal{A}_2} &: \mathcal{A}_2 \to \Gamma \to \mathbb{Z} \\
[\![_]\!]_{\mathcal{B}_2} &: \mathcal{B}_2 \to \Gamma \to \mathit{bool} \\
[\![_]\!]_{\mathcal{L}_2} &: \mathcal{L}_2 \to \Gamma \to \Gamma
\end{aligned}$$

Comme la définition d'une structure de données est un ensemble de définitions associées à des étiquettes, la sémantique d'une structure de données est définie par :

$$[\![\_]\!]_{\mathcal{D}_2} : \mathcal{D}_2 \to \Gamma \to \Gamma$$

Dans le langage $\mathcal{L}_2$, l'environnement Γ contient, tout comme dans $\mathcal{L}_1$, les valeurs des entrées du programme. Si la même variable prend plusieurs valeurs en entrée dans le temps, ces différentes valeurs peuvent être accédées de la même façon que les éléments d'un tableau. Par exemple, avec une variable $x \mapsto \mathit{input}$ et un environnement de départ $[x_I \mapsto \langle\{1\} \mapsto 4\{2\} \mapsto 6\rangle]$, la sémantique de la variable $w \mapsto x[2]$ sera $w \mapsto 6$. En ce qui a trait aux sorties , elles sont identifiées directement dans la définition du programme.

Malgré que certains exemples utilisent un environnement initial non vide, les programmes de $\mathcal{L}_2$ ont toujours un environnement vide ($s = []$) initialement (à l'exception des variables d'entrée). Les exemples ont été faits de cette façon seulement pour réduire leur taille.

Comme le montre le tableau 4.12, le mot clé pos_L permet de représenter la position de l'élément courante dans l'expression permettant de calculer sa valeur. L'indice L représente le niveau de profondeur dans le tableau. Par exemple :

$$\left[\, x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 4, \\ \{2\} \mapsto \left\langle\, \{1\} \mapsto pos_1 * pos_2 \,\right\rangle \end{array} \right\rangle \,\right]$$

Ici, pos_1 prendrait la valeur de 2 et pos_2 1.

La fonction auxiliaire *cond* est définie d'une façon semblable à celle du langage $\mathcal{L}_2$ d'origine à l'exception qu'elle prend en compte les entrées, soit :

$$\mathit{cond}([\![b]\!]_{\mathcal{B}_2}(s), [\![a_1]\!]_{\mathcal{L}_2}(s), [\![a_2]\!]_{\mathcal{L}_2}(s)) = \begin{cases} [\![a_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = \mathit{true} \\ [\![a_1]\!]_{\mathcal{L}_2}(s) & \text{si } [\![b]\!]_{\mathcal{B}_1}(s) = \mathit{false} \end{cases}$$

$$
\begin{array}{rcl}
[\![n]\!]_{A_2}(s) & = & n \\
[\![x]\!]_{A_2}(s) & = & s(x) \\
[\![a_1 \ op \ a_2]\!]_{A_2}(s) & = & [\![a_1]\!]_{A_2}(s) \ op \ [\![a_2]\!]_{A_2}(s) \\
[\![if(b, a_1, a_2)]\!]_{A_2}(s) & = & cond([\![b]\!]_{B_2}(s), [\![a_1]\!]_{A_2}(s), [\![a_2]\!]_{A_2}(s)) \\
[\![x[a_1] \ldots [a_n]P]\!]_{A_2}(s) & = & [\![P]\!]_{\mathcal{L}_2}(s) \ (x_{[\![a_1]\!]_{A_2}(s) \ldots [\![a_n]\!]_{A_2}(s)}) \\
[\![\mu^x_{b,\ P} P']\!]_{A_2}(s) & = & (\mu f_{b,P}(x))[\![P']\!]'_{\mathcal{L}_2}(s) \\
[\![true]\!]_{B_2}(s) & = & true \\
[\![\neg b]\!]_{B_2}(s) & = & \neg [\![b]\!]_{B_2}(s) \\
[\![b_1 \vee b_2]\!]_{B_2}(s) & = & [\![b_1]\!]_{B_2}(s) \ \vee [\![b_2]\!]_{\mathcal{B}_1} s \\
[\![a_1 < a_2]\!]_{B_2}(s) & = & [\![a_1]\!]_{A_2}(s) < [\![a_2]\!]_{\mathcal{A}_1}(s) \\
[\![a_1 = a_2]\!]_{B_2}(s) & = & [\![a_1]\!]_{A_2}(s) = [\![a_2]\!]_{A_2}(s) \\
[\![\{a_1, \ldots, a_n\} \mapsto a]\!]_{D_2}(s) & = & \bigcup_{y \in \{[\![a_1]\!]_{A_2}(s), \ldots, [\![a_n]\!]_{A_2}(s)\}} \{y\} \mapsto [\![a[y/pos_L]]\!]_{A_2}(s) \\
[\![\mu^x_{b,\ P} P']\!]_{D_2}(s) & = & (\mu f_{b,P}(x))[\![P']\!]'_{\mathcal{L}_2}(s) \\
[\![\langle elem_1, elem_2 \rangle]\!]_{D_2}(s) & = & [\![elem_1]\!]_{D_2}([\![elem_2]\!]_{D_2}(s)) \\
[\![\langle elem \rangle]\!]_{D_2}(s) & = & \langle [\![elem]\!]_{D_2}(s) \rangle \\
[\![x \mapsto a]\!]_{\mathcal{L}_2}(s) & = & s \dagger [x \mapsto [\![a]\!]_{A_2}(s)] \\
[\![x \mapsto input]\!]_{\mathcal{L}_2}(s) & = & s(x_I) \\
[\![x \mapsto d]\!]_{\mathcal{L}_2}(s) & = & s \dagger [x \mapsto [\![d]\!]_{D_2}(s)] \\
[\![x_O \mapsto a]\!]_{\mathcal{L}_2}(s) & = & s \dagger [x_O \mapsto [\![a]\!]_{A_2}(s)] \\
[\![x_O \mapsto d]\!]_{\mathcal{L}_2}(s) & = & s \dagger [x_O \mapsto [\![d]\!]_{D_2}(s)] \\
[\![P_1 \cup P_2]\!]_{\mathcal{L}_2}(s) & = & [\![P_1]\!]_{\mathcal{L}_2}([\![P_2]\!]_{\mathcal{L}_2}(s)) \text{ si } P_2 \sqsubseteq P_1 \\
[\![P_1 \cup P_2]\!]'_{\mathcal{L}_2}(s) & = & [\![P_1]\!]_{\mathcal{L}_2} s \cup [\![P_2]\!]_{\mathcal{L}_2}(s) \\
\text{avec :} & & \\
op & \in & \{+, *\} \\
x_{i \ldots j} s & = & v \text{ si } x \mapsto \langle \{i\} \mapsto \langle \ldots v \ldots \rangle \rangle \in s \\
P & = & [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] \\
P' & = & [x'_1 \mapsto e'_1, \ldots, x'_n \mapsto e'_n] \\
f_{b,P} g & = & cond([\![b]\!]_{B_2} s, g \circ [\![P]\!]_{\mathcal{L}_2} s, id) \\
P_2 \sqsubseteq P_1 & = & Use(P_2) \cap Def(P_1) \ = \ \emptyset
\end{array}
$$

TAB. 4.12 – Sémantique du langage $\mathcal{L}_2$.

Avec les fonctions $Def(P)$ et $Use(P)$ telles que définies dans les tableaux 4.13 et 4.14.

$$\begin{array}{rcl} Def(x = e) & = & \{x\} \\ Def(P_1; P_2) & = & Def(P_1) \cup Def(P_2) \end{array}$$

TAB. 4.13 – Ensemble des variables définies dans P.

$$\begin{array}{rcl} Use(x \mapsto e) & = & Use(e) \\ Use(P_1; P_2) & = & Use(P_1) \cup Use(P_2) \\ \\ Use(n) & = & \emptyset \\ Use(x) & = & \{x\} \\ Use(a_1\ op\ a_2) & = & Use(a_1) \cup Use(a_2) \\ Use(x[a_1]\ldots[a_n]) & = & Use(a_1) \cup \ldots \cup Use(a_n) \\ Use(if(b, a_1,\ a_2)) & = & Use(b) \cup Use(a_1) \cup Use(a_2) \\ Use(\mu^x_{b,P} P') & = & Use(b) \cup Use(P') \cup (Use(P) - Def(P')) \\ \\ Use(true) & = & \emptyset \\ Use(\neg b) & = & Use(b) \\ Use(b_1 \vee b_2) & = & Use(b_1) \cup Use(b_2) \\ Use(a_1 = a_2) & = & Use(a_1) \cup Use(a_2) \\ \\ \text{Avec} & & op \in \{+, *, <, =\} \end{array}$$

TAB. 4.14 – Ensemble des variables utilisées dans P.

La sémantique des expressions récursives ($\mu^x_{b,\ P}[x' \mapsto e'_1, \ldots, x'_m \mapsto e'_m]$) correspond à la projection sur la variable x du plus petit point fixe de la fonction $f_{b,\ P}$ appliquée à l'environnement $[x' \mapsto e'_1, \ldots, x'_m \mapsto e'_m]$.

La sémantique de l'affectation comporte une subtilité supplémentaire par rapport à celle de la version précédente de $\mathcal{L}_2$. En effet, elle permet d'affecter le résultat de

l'évaluation d'une structure de données à une variable. Plus précisément, pour tout environnement s dans Γ :

$$\begin{aligned}[\![x \mapsto d]\!]_{\mathcal{L}_2}(s) &= [x \mapsto [\![d]\!]_{\mathcal{D}_2}] \circ s \\ &= s \dagger [x \mapsto [\![d]\!]_{\mathcal{D}_2}(s)]\end{aligned}$$

L'environnement est mis à jour grâce à † qui permet d'y ajouter des éléments ou de les redéfinir. Par exemple, avec une expression arithmétique y, $s \dagger [x \mapsto y]$ permet d'ajouter à l'environnement s le fait que x prend la valeur de l'évaluation de y si x ne fait pas déjà partie de s et sinon, x va être redéfini. Cependant, la notion de cases mémoires doit être introduite pour les structures de données. En effet, lorsqu'un environnement ne définit que certaines parties d'une structure de données, et qu'une nouvelle définition de cette structure de données est ajoutée à s, les éléments contenus dans la nouvelle définition écrasent celles de l'ancienne, mais sans effacer les autres. Ainsi :

$$\{x \mapsto \langle\{1,2,3\} \mapsto 0\rangle\} \dagger \{x \mapsto \langle\{3,4,5\} \mapsto 1\rangle\} = \{\{1,2\} \mapsto 0, x \mapsto \langle\{3,4,5\} \mapsto 1\rangle\}$$

Malgré que la notation utilisée pour le point fixe ressemble à celle du langage $\mathcal{L}_1$, celle du langage $\mathcal{L}_2$ n'a pas de séquences d'instructions. Elle utilise plutôt des sous-programmes de $\mathcal{L}_2$. Par exemple, la sémantique du programme P suivant serait :

$$P = \left\{ \begin{array}{l} x \mapsto 1, \\ y \mapsto \mu^{y}_{x<3,[y\mapsto y+1]}[x \mapsto 2, y \mapsto 5] \end{array} \right\}$$

$$[\![P]\!]_{\mathcal{L}_2}(s) = \left[\begin{array}{l} x \mapsto 1, \\ y \mapsto 6 \end{array} \right]$$

Dans cet exemple, $[y \mapsto y+1]$ et $[x \mapsto 2, y \mapsto 5]$ sont des définitions de variables, celles-ci ne sont pas dépendantes de l'ordre dans lequel elles sont déclarées.

De plus, dans l'expression $\mu^{x}_{b,P_1} P_2$, le P_2 est utilisé pour permettre d'initialiser des variables avant le calcul du point fixe. De cette façon, il est possible d'initialiser un compteur pour itérer un certain nombre de fois. Sans l'utilisation du programme P_2, le compteur devrait être une déclaration de variable indépendante dans le programme. Ceci introduirait des variables temporaires dans la définition d'un programme. Par exemple sans P_2, nous devrions écrire le programme suivant :

$$P \quad = \quad \left\{ \begin{array}{l} y \mapsto \mu^{y}_{i<10,[y \mapsto i*i]}[i \mapsto 5], \\ x \mapsto \mu^{x}_{i<10,[x \mapsto i]}[i \mapsto 1], \end{array} \right\}$$

De la façon suivante :

$$P' \quad = \quad \left\{ \begin{array}{l} x \mapsto \mu^{x}_{i<10,[x \mapsto i]}, \\ y \mapsto \mu^{y}_{j<10,[y \mapsto j*j]}, \\ i \mapsto 1, \\ j \mapsto 5, \end{array} \right\}$$

Nous pouvons remarquer que la nouvelle version nécessite l'introduction de deux nouvelles définitions.

4.4.3 Exemples

Exemple 1

L'exemple suivant montre un programme simple qui initialise tous les éléments d'un tableau de dix éléments à 0 :

$$P = \{x \mapsto \langle \{1, \ldots, 10\} \mapsto 0 \rangle\}$$

Exemple 2

Celui-ci montre l'utilisation du mot clé pos_L en prenant un tableau y de n éléments et en incrémentant la valeur de tous ses éléments de 1 :

$$P = \{x \mapsto \langle \{1, \ldots, n\} \mapsto y[pos_1] + 1 \rangle\}$$

Par exemple, avec l'environnement $s = [n \mapsto 3, y \mapsto \langle \{1, 2, 3\} \mapsto 8 \rangle]$:

$$\begin{aligned}
[\![P]\!]_{\mathcal{L}_2} s &= s \dagger [x \mapsto \langle \bigcup_{z \in \{[\![1]\!]_{A_2} s, [\![2]\!]_{A_2} s, [\![3]\!]_{A_2}(s)\}} z \mapsto [\![y[z]+1]\!]_{A_2} s \rangle] \\
&= s \dagger [x \mapsto \langle \bigcup_{z \in \{1,2,3\}} z \mapsto [\![y[z]+1]\!]_{A_2}(s) \rangle] \\
&= s \dagger [x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto [\![y[1]+1]\!]_{A_2}(s), \\ \{2\} \mapsto [\![y[2]+1]\!]_{A_2}(s), \\ \{3\} \mapsto [\![y[3]+1]\!]_{A_2}(s) \end{array} \right\rangle] \\
&= s \dagger [x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto [\![y[1]]\!]_{A_2}(s) + [\![1]\!]_{A_2}(s), \\ \{2\} \mapsto [\![y[2]]\!]_{A_2}(s) + [\![1]\!]_{A_2}(s), \\ \{3\} \mapsto [\![y[3]]\!]_{A_2}(s) + [\![1]\!]_{A_2}(s) \end{array} \right\rangle] \\
&= s \dagger [x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 8+1, \\ \{2\} \mapsto 8+1, \\ \{3\} \mapsto 8+1 \end{array} \right\rangle] \\
&= s \dagger [x \mapsto \langle \{1,2,3\} \mapsto 9 \rangle]
\end{aligned}$$

Exemple 3

L'exemple suivant montre un programme utilisant le point fixe :

$$P_3 = \{x \mapsto \langle \mu^x_{(i \leq n), [x \mapsto \langle \{i\} \mapsto x[i-1] * x[i] \rangle, i \mapsto i+1]} [x \mapsto \langle \{1,2,3\} \mapsto 8 \rangle] \rangle\}$$

Il met dans la case i la valeur de la case $i-1$ multipliée par la valeur courante de la case i pour toutes les cases dont l'indice est compris entre i et n.

Pour le montrer, nous allons utiliser l'environnements s suivant :

$$s = [i \mapsto 2, n \mapsto 3]$$

Ainsi, avec la sémantique du langage $\mathcal{L}_2$, nous avons :

$$[\![x \mapsto \langle elem_1 \rangle]\!]_{\mathcal{L}_2}(s) = s \dagger [x \mapsto \langle [\![elem_1]\!]_{A_2}(s) \rangle]$$

Avec :

$$\begin{aligned}
elem_1 &= \mu^x_{(i \leq n), P_1} P_2 \\
P_1 &= \{x \mapsto \langle \{i\} \mapsto x[i-1] * x[i] \rangle, i \mapsto i+1\} \\
P_2 &= \{x \mapsto \langle \{1,2,3\} \mapsto 8 \rangle\}
\end{aligned}$$

Nous avons :

$$[\![\mu^{x}_{(i\leq n),\ P_1} P_2]\!]_{D_2}(s) = (\mu f_{(i\leq n),P_1}(x))[\![P_2]\!]_{\mathcal{L}_2}(s)$$

En déterminant le plus petit point fixe, nous obtenons :

$$(\mu f_{(i\leq n),P_1}(x))[\![P_2]\!]_{\mathcal{L}_2}(s) = [x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 8, \\ \{2\} \mapsto 64, \\ \{3\} \mapsto 512 \end{array} \right\rangle, i \mapsto 4]$$

Donc, en effectuant la projection :

$$\begin{array}{rcl} [\![P]\!]_{\mathcal{L}_2}(s) & = & s \dagger [x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 8, \\ \{2\} \mapsto 64, \\ \{3\} \mapsto 512 \end{array} \right\rangle] \\ & = & [i \mapsto 2, n \mapsto 3, x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 8, \\ \{2\} \mapsto 64, \\ \{3\} \mapsto 512 \end{array} \right\rangle] \end{array}$$

Exemple 4

Le programme suivant représente un algorithme de tri par sélection.

$$P = \{tab2 \mapsto \langle \mu^{tab2}_{(i\leq n),P_1}[tab2 \mapsto \langle \{1\} \mapsto tab1[1]\rangle, i \mapsto 2]\rangle\}$$

Avec

$$P_1 = \left\{ \begin{array}{l} i \mapsto i+1, \\ tab2 \mapsto \left\langle \begin{array}{l} \{(\mu^{j}_{(b_1),P_2}[j \mapsto i-1]) + 1\} \mapsto tab1[i] \\ \mu^{tab2}_{(b_1),P_2}[j \mapsto i-1] \end{array} \right\rangle \end{array} \right\}$$

$$b_1 = j \geq 1 \wedge tab2[j] > tab1[i]$$

et

$$P_2 = \{tab2 \mapsto \langle \{j+1\} \mapsto tab2[j]\rangle, j \mapsto j-1\}$$

En utilisant un $s = [tab1 \mapsto \langle 3,1,2\rangle, n \mapsto 3]$ la sémantique du programme P donne :

$$[\![P]\!]_{\mathcal{L}_2}(s) = \left[tab1 \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 3, \\ \{2\} \mapsto 1, \\ \{3\} \mapsto 2 \end{array} \right\rangle, n \mapsto 3, tab2 \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 1, \\ \{2\} \mapsto 2, \\ \{3\} \mapsto 3 \end{array} \right\rangle \right]$$

4.5 Traduction de $\mathcal{L}_1$ vers $\mathcal{L}_2$

La méthode utilisée pour traduire un programme est semblable à celle utilisée avec les langages $\mathcal{L}_1$ et $\mathcal{L}_2$ d'origine. Cependant, quelques points ont été ajoutés pour tenir compte des modifications apportées aux deux langages. Ainsi, nous obtenons maintenant l'ensemble des variables définies dans P soit, $V(P)$, comme le montre le tableau 4.15, l'application comme le montre le tableau 4.16, l'identification des variables à ne pas substituer (tableau 4.17), la pré et porttraduction et finalement, la fonction de traduction est représentée dans les tableaux 4.18 et 4.19.

4.5.1 Ensemble des variables définies

L'ensemble des variables définies dans P est déterminé selon les règles telles que montrées dans le tableau 4.15.

$$\begin{array}{rcl} V(x = a) & = & \{x\} \\ V(x[a_1]...[a_n] = e) & = & \{x\} \\ V(x = allocate(y)) & = & \{x\} \\ V(free(x)) & = & \{x\} \\ V(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) & = & V(P_1) \cup V(P_2) \\ V(\textbf{while } b \textbf{ do } P) & = & V(P) \\ V(P_1; P_2) & = & V(P_1) \cup V(P_2) \end{array}$$

TAB. 4.15 – Ensemble des variables définies dans P.

Ainsi comme nous pourrions nous y attendre intuitivement pour le programme P suivant :

```
1 y = 3
2 x = allocate(2);
3 x[1] = 6;
```

$$\begin{aligned} V(P) &= V(y=3) \cup V(x = allocate(2);) \cup V(x[1] = 6;) \\ &= \{y\} \cup \{x\} \cup \{x\} \\ &= \{y, x\} \end{aligned}$$

4.5.2 Application

Soit une expression de $\mathcal{A}_2 \cup \mathcal{B}_2 \cup \mathcal{D}_2$ et P un programme de $\mathcal{L}_2$. L'application de P à a notée $P(a)$ se définit comme le montre la tableau 4.16.

L'application au niveau des structures de données est particulière, l'application d'un programme P à $\langle\{a_1\} \mapsto a'_1\rangle$ donne $\langle\{P(a_1)\} \mapsto P(a'_1)\rangle$, soit un tableau dont P est appliqué à l'élément a_1 qui a la valeur de l'application de P à a'_1, si le tableau n'est pas déjà déclaré dans P. S'il l'est déjà, nous devons y ajouter le nouvel élément.

Exemple 1

Voici un exemple avec un programme P qui ne contient pas le tableau z en question soit $\langle\{x\} \mapsto x + 1\rangle$ avec $P = \{x \mapsto 3\}$:

$$\begin{aligned} (\langle\{x\} \mapsto x + 1\rangle)^z P &= \langle\{xP\} \mapsto (x+1)P\rangle \\ &= \langle\{3\} \mapsto xP + 1P\rangle \\ &= \langle\{3\} \mapsto 3 + 1\rangle \\ &= \langle\{3\} \mapsto 4\rangle \end{aligned}$$

$$
\begin{array}{rcl}
P(c) & = & c \\
P(x) & = & \begin{cases} a & \text{si } [x \mapsto a] \in P \\ x & \text{sinon} \end{cases} \\
P(x^m) & = & x^m \\
P(x[a_1]...[a_n]P_1) & = & x[P(a_1)]...[P(a_n)]P \dagger P_1 \\
P((x[a_1]...[a_n]P_1)^m) & = & x[a_1]...[a_n]P_1)^m \\
P(\langle elem_1 \rangle^x) & = & \begin{cases} \langle P(elem_1), elem \rangle & si\ [x \mapsto \langle elem \rangle] \in P \\ \langle P(elem_1) \rangle & sinon \end{cases} \\
P(\{a_1, a_n\} \mapsto e) & = & \{P(a_1), P(a_n)\} \mapsto P(e) \\
P(null^x) & = & P' \text{avec } P = P' \dagger [x \mapsto d] \\
P(\neg b) & = & \neg(P(b)) \\
P(a_1\ op\ a_2) & = & P(a_1)\ op\ P(a_2) \\
P(if(b, a_1, a_2)) & = & if(P(b), P(a_1), P(a_2)) \\
P(\mu^x_{b,P_1}[x'_1 \mapsto e'_1, \ldots, x'_m \mapsto e'_m]^z) & = & \mu^x_{b,P_1} P/z \dagger [x'_1 \mapsto P(e'_1), \ldots, x'_m \mapsto P(e'_m)] \\
P(x \mapsto a_1) & = & x \mapsto P(a_1) \\
P(x \mapsto d) & = & x \mapsto P(d^x) \\
\text{avec } c & \in & \{n, true, false\} \\
op & \in & \{+, *, =, <, \vee\} \\
e & \in & \{a, d\} \\
P_1 & = & [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]
\end{array}
$$

TAB. 4.16 – Application.

Exemple 2

L'exemple suivant montre l'application de P à $(\langle \{y\} \mapsto x \rangle)^{tab}$ dans le cas où le tableau existe dans P :

$$
P = \left\{ \begin{array}{l} x \mapsto 3, \\ y \mapsto 1, \\ tab \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 0, \\ \{3\} \mapsto 0 \end{array} \right\rangle \end{array} \right\}
$$

$$
\begin{aligned}
P(\langle\{y\} \mapsto x\rangle^{tab}) &= \left\langle \begin{array}{l} \{P(y)\} \mapsto P(x), \\ \{1\} \mapsto 0, \\ \{3\} \mapsto 0 \end{array} \right\rangle \\
&= \left\langle \begin{array}{l} \{1\} \mapsto 3, \\ \{1\} \mapsto 0, \\ \{3\} \mapsto 0 \end{array} \right\rangle
\end{aligned}
$$

4.5.3 Identification des variables à ne pas substituer

Tout comme présenté au début de ce chapitre, une identification des variables à ne pas substituer est nécessaire lors de la traduction avec les tableaux. Cette fonction $M(e, z)$ est définie comme le montre le tableau 4.17

$$
\begin{aligned}
M(c, z) &= c \\
M(x, z) &= \begin{cases} x^m & \text{si } x \in z \\ x & \text{sinon} \end{cases} \\
M(x[a_1]...[a_n]P_1, z) &= \begin{cases} (x[a_1]...[a_n]P_1)^m & \text{si } x \in z \\ x[a_1]...[a_n]P_1 & \text{sinon} \end{cases} \\
M(\langle elem_1\rangle^x, z) &= \langle M(elem_1, z)\rangle^x \\
M(\{a_1, a_n\} \mapsto e, z) &= \{M(a_1, z), M(a_n, z)\} \mapsto M(e, z) \\
P(null^x) &= P' \text{avec } P = P' \dagger [x \mapsto d] \\
M(\neg b, z) &= \neg(M(b, z)) \\
M(a_1\ op\ a_2, z) &= M(a_1, z)\ op\ M(a_2, z) \\
M(if(b, a_1, a_2), z) &= if(M(b, z), M(a_1, z), M(a_2, z)) \\
M(\mu^x_{b,P_1}[x'_1 \mapsto a'_1, \ldots, x'_m \mapsto a'_m]^{z'}, z) &= \mu^x_{b,P_1} \dagger [x'_1 \mapsto M(a'_1), \ldots, x'_m \mapsto M(a'_m)]^{z' \cup z} \\
\text{avec } c &\in \{n, true, false\} \\
op &\in \{+, *, =, <, \vee\} \\
e &\in \{a, d\} \\
P_1 &= [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]
\end{aligned}
$$

TAB. 4.17 – La fonction d'identification des variables à ne pas substituer.

4.5.4 Pré et Posttraduction

À cause des variables d'entrées (*input*) et de sorties (*output*), les programmes écrits dans le langage $\mathcal{L}_1$ doivent subir une transformation avant de pouvoir effectuer la traduction. Parallèlement, une autre étape doit être effectuée après l'étape de traduction pour que le programme corresponde à l'original dans le langage $\mathcal{L}_2$. Ces étapes sont dues à la présence des compteurs dans la langage $\mathcal{L}_1$ et de leur absence dans le langage $\mathcal{L}_2$.

La première chose effectuée par la prétraduction est de créer deux nouvelles variables pour chacune des variables utilisées comme sortie. La première variable représente un tableau correspondant à la variable de sortie. La deuxième quant à elle, est un compteur permettant d'identifier combien de fois la variable a été utilisée en sortie. Ainsi, chaque expression $output(x)$ est remplacée par $n = n + 1;\ x_O[n] = x$ de façon à enregistrer la valeur de chaque sortie. Pour la traduction les variables x et x_O sont traitées comme deux variables distinctes. Pour ce qui est des entrées, le fonctionnement est relativement similaire. En effet, une nouvelle variable est créée au début du programme pour chacune des variables utilisée en entrée. Chaque affectation à une variable utilisée en entrée (x dans cet exemple) est remplacé par $n = n+1$ et chaque fois qu'une variable d'entrée est utilisée dans une définition, elle est remplacée par $x[n]$. Il est à noter que l'instruction $input(x)$ n'est remplacée que par une instruction comparativement à deux dans le cas d'une sortie. Par exemple, la prétraduction du programme suivant :

```
1 x = 1;
2 output(x);
3 while x < 10 do
4       output(x);
5       x = x + 2;
```

donnerait :

```
1 n = 0;
2 x = 1;
3 n = n + 1;
4 x_o[n] = x;
5 while x < 10 do
6       n = n + 1;
7       x_o[n] = x;
8       x = x + 2;
```

L'étape de posttraduction quant à elle est utilisée pour éliminer les variables introduites lors de la prétraduction et réintroduire les entrées retirées. Ainsi, chaque instance de compteurs est remplacée par sa valeur puis chacune des variables utilisées comme compteur est retirée du programme résultant. Ensuite, toutes les variables utilisées en entrée avant la prétraduction sont réintroduites ainsi : $x \mapsto input$ (dans le cas d'une variable x qui aurait été utilisée en entrée).

4.5.5 Fonction de traduction

Les fonctions $\mathcal{D}$ et $\mathcal{D}'$ permettant de traduire un programme du langage $\mathcal{L}_1$ vers le langage $\mathcal{L}_2$ sont données dans les tableaux 4.18 et 4.19.

$$
\begin{array}{rcl}
\mathcal{D}(x = a) & = & \{x \mapsto a\} \\
\mathcal{D}(x = allocate(n)) & = & \{x \mapsto \langle x_{\{1,\ldots,n\}} \mapsto null\rangle\} \\
\mathcal{D}(x[a_1]...[a_n] = a) & = & \{x \mapsto \langle \ldots \{a_1\} \mapsto \langle \ldots \{a_n\} \mapsto a \ldots\rangle \ldots\rangle\} \\
\mathcal{D}(free(x)) & = & \{x \mapsto null\} \\
\mathcal{D}(P_1; P_2) & = & \bigcup_{x \in (V(P_1) - V(P_2))} \{x \mapsto D(P_1)(x)\} \cup \\
 & & \bigcup_{x \in (V(P_1) \cap V(P_2))} \{x \mapsto (D(P_1)/x)(D(P_2)(x))\} \cup \\
 & & \bigcup_{x \in (V(P_2) - V(P_1))} \{x \mapsto M(((D(P_1))/y)(D(P_2)(x)), z)\} \\
 & & \text{avec } y \in (V(P_1) \cap V(P_2)) \\
 & & \text{et } z \in (V(P_1) - V(P_2)) \\
\mathcal{D}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) & = & \bigcup_{x \in V(P_1;P_2)} \{x \mapsto if(b, D'(P_1)(x), D'(P_2)(x))\} \\
\mathcal{D}(\textbf{while } b \textbf{ do } P) & = & \bigcup_{x \in V(P)} \{x \mapsto \mu^x_{b,D'(P)}\}
\end{array}
$$

TAB. 4.18 – $\mathcal{D}$ - La fonction de traduction de $\mathcal{L}_1$ à $\mathcal{L}_2$.

$$
\begin{array}{rcl}
\mathcal{D}'(P_1; P_2) & = & D'(P_2) \circ D'(P_1) \\
& = & \bigcup_{x \in V(P_2)} \{(x \mapsto (D'(P_2)(x)))D'(P_1)\} \cup \\
& & \bigcup_{x \in (V(P_1) - V(P_2))} \{x \mapsto D'(P_1)(x)\} \\
\mathcal{D}'(P) & = & \mathcal{D}(P)
\end{array}
$$

TAB. 4.19 – La fonction de traduction $\mathcal{D}'$.

Elles sont sensiblement identiques aux fonctions de traduction présentées à la section 4.2.2 à l'exception de quelques règles qui y ont été ajoutées. En effet, les instructions relatives à la mémoire, telles l'allocation, la libération et la mise à jour d'un élément peuvent maintenant être traduites.

Par exemple, pour traduire $(x[2] = 3)$ avec x un espace mémoire de deux éléments, nous obtenons $\{x \mapsto \langle\{2\} \mapsto 3\rangle\}$.

Les exemples suivants permettent de bien visualiser comment un programme du langage $\mathcal{L}_1$ avec des structures de données peut être traduit vers le langage $\mathcal{L}_2$.

4.5.6 Exemples

Exemple 1

Voici un exemple d'un programme P dans le langage $\mathcal{L}_1$. Cet exemple permet de voir comment la traduction se comporte avec les entrées et sorties.

```
1 x = allocate(3);
2 input(z);
3 x[1] = z;
4 x[2] = 1;
5 x[3] = x[1]+x[2];
6 y = x[3];
7 output(y);
```

Sa prétraduction donne le programme P' suivant :

```
1  n = 0;
2  m = 0;
3  x = allocate(3);
4  n = n + 1;
5  x[1] = z[n];
6  x[2] = 1;
7  x[3] = x[1]+x[2];
8  y = x[3];
9  m = m + 1;
10 y_o[m] = y;
```

Suite à la prétraduction, la traduction vers le langage $\mathcal{L}_2$ donne :

$$\mathcal{D}(P') = \left\{ \begin{array}{l} y_O \mapsto \langle \{m\} \mapsto y \rangle, \\ m \mapsto 1, \\ y \mapsto x[3], \\ x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto z[1], \\ \{2\} \mapsto 1, \\ \{3\} \mapsto z[1]+1 \end{array} \right\rangle, \\ n \mapsto 1, \end{array} \right\}$$

En appliquant la posttraduction, nous obtenons le programme suivant :

$$\mathcal{D}(P) = \left\{ \begin{array}{l} y_O \mapsto \langle \{1\} \mapsto x[3] \rangle, \\ y \mapsto x[3], \\ x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto z[1], \\ \{2\} \mapsto 1, \\ \{3\} \mapsto z[1]+1 \end{array} \right\rangle, \\ z \mapsto input \end{array} \right\}$$

Exemple 2

L'exemple suivant montre la traduction d'un programme qui nécessite l'utilisation du point fixe. Le programme à traduire est :

```
1 x = allocate(10);
2 i = 1;
3 while i < 11 do
4       x[i] = i;
5       i = i+1;
6 x[1] = 0;
```

Sa traduction, avec P_1 correspondant à $[\mu^x_{i<11,P_4}[i \mapsto 1, x \mapsto \langle\{1,\ldots,10\} \mapsto null\rangle]]$, est :

$$\mathcal{D}(P) \;=\; \left\{ \begin{array}{l} x \mapsto \left\langle \begin{array}{l} \{1\} \mapsto 0, \\ P_1, \\ \{1,\ldots,10\} \mapsto null \end{array} \right\rangle, \\ i \mapsto \mu^i_{i<11,P_4}[i \mapsto 1, x \mapsto \langle\{1,\ldots,10\} \mapsto null\rangle] \end{array} \right\}$$

Exemple 3

L'exemple suivant présente un programme permettant d'afficher le carré d'un nombre entré par l'utilisateur.

Le programme P à traduire est le suivant :

```
1 input(x);
2 while x ≠ 0 do
3       carre = x*x;
4       output(carre);
5       x = input;
```

Une fois l'étape de prétraduction effectuée, nous obtenons le programme P' suivant :

```
1 n = 0;
2 m = 0;
3 n = n+1;
4 while x ≠ 0 do
5       carre = x[n]*x[n];
6       m = m+1;
7       carre_o[m] = carre;
8       n = n+1;
```

En utilisant la fonction de traduction sur ce programme, nous obtenons :

$$D(P') = \left\{ \begin{array}{l} carre \mapsto \mu^{carre}_{\neq 0,P_1}[n \mapsto 1, m \mapsto 0], \\ m \mapsto \mu^{m}_{x\neq 0,P_1}[n \mapsto 1, m \mapsto 0], \\ carre_O \mapsto \langle \mu^{carre_O}_{x\neq 0,P_1}[n \mapsto 1, m \mapsto 0]\rangle, \\ n \mapsto \mu^{n}_{x\neq 0,P_1}[n \mapsto 1, m \mapsto 0], \\ x \mapsto \mu^{x}_{x\neq 0,P_1}[n \mapsto 1, m \mapsto 0] \end{array} \right\}$$

Avec :

$$P_1 = \left\{ \begin{array}{l} carre \mapsto x[n] * x[n], \\ m \mapsto m + 1, \\ carre_O \mapsto \langle \{m + 1\} \mapsto x * x\rangle, \\ n \mapsto n + 1 \end{array} \right\}$$

Une fois la posttraduction terminée, le programme résultant est :

$$D(P) = \left\{ \begin{array}{l} carre \mapsto \mu^{carre}_{\neq 0,P_1}[n \mapsto 1, m \mapsto 0], \\ carre_O \mapsto \langle \mu^{carre_O}_{x\neq 0,P_1}[n \mapsto 1, m \mapsto 0]\rangle, \\ x \mapsto input \end{array} \right\}$$

Avec :

$$P_1 \quad = \quad \left\{ \begin{array}{l} carre \mapsto x[n] * x[n], \\ m \mapsto m + 1, \\ carre_O \mapsto \langle \{m + 1\} \mapsto x * x \rangle, \\ n \mapsto n + 1 \end{array} \right\}$$

Comme il est possible de le voir, le programme résultant représente bien ce que l'on attendait du programme original. La sortie ($carre_O$) correspond toujours au carré de la variable précédemment entrée par l'utilisateur. En effet, lors de la sortie, l'index de sortie m correspondra toujours à l'index précédemment n utilisé pour l'entrée.

4.6 Conclusion et travaux futurs

Finalement, ce chapitre a présenté l'extension des langages originaux ainsi que de la fonction de traduction pour permettre de traiter les tableaux et les entrées et sorties. Pour ce faire, les langages $\mathcal{L}_1$ et $\mathcal{L}_2$ ont été étendus pour permettre de prendre en compte ces nouveaux éléments. En ce qui a trait à la fonction de traduction, elle a aussi été modifiée et une étape de traduction supplémentaire a été ajoutée. La preuve que la conversion préserve la sémantique des programmes lors de la traduction n'a pas été faite. Cependant, à partir de la preuve originale, elle est relativement directe puisqu'elle utilise les éléments des langages originaux auxquels ont été ajoutés quelques subtilités sans impacts majeurs sur la preuve. Il n'en reste pas moins que la preuve devrait être effectuée pour s'assurer que la sémantique est bien respectée lors de la traduction.

En ce qui a trait aux travaux futurs, il serait intéressant d'étendre les langages et la conversion pour permettre encore plus de fonctionnalités. Par exemple, des structures de données telles que les tables de hachage ou les arbres seraient probablement utiles pour construire des programmes plus complexes. Il serait aussi intéressant que les fonctions soient au moins supportées par le langage impératif utilisé comme source de la conversion. De plus, une simplification du programme résultant de la traduction serait probablement intéressante. Par exemple, des simplifications utilisées pour la coupe de programmes [15] pourraient être introduites. De plus, une évaluation partielle pourrait être intéressante. Par exemple, lors de la traduction d'un programme, il est fréquent de se retrouver avec des définitions comme : $x \mapsto y + 3 + 5$. Dans ces cas, la définition de x pourrait être facilement simplifiée. Aussi, si nous ne nous intéressons qu'aux variables utilisées en sortie, toutes les autres pourraient être supprimées puisque la traduction produit des définitions indépendantes les unes des autres. De cette façon, les variables temporaires du programme dans le langage $\mathcal{L}_1$ ne se retrouveraient, dans $\mathcal{L}_2$, que pour

les définitions de variables où elles sont nécessaires au lieu d'avoir une définition à elle en plus.

Un autre point à considérer comme travail futur est la simplification du résultat de la fonction de traduction notamment en ce qui concerne les définitions utilisant l'expression $\mu^x_{b,P}\, P'$. À cette fin, nous envisageons d'introduire une notation qui ferait plutôt appel à la récursion. En effet, il est intéressant de constater qu'étant donnée une mémoire s, l'expression $\mu^x_{b,P}\, P'$ peut être calculée en appelant une fonction *Frec* comme suit : $\mathit{Frec}^x(b, P, s \dagger P')$. Cette fonction est définie comme suit :

$$\mathit{Frec}^x(b, P, s) = \textbf{if } \neg[\![b]\!]_{B_2} s \textbf{ then } s\ x \textbf{ else } \mathit{Frec}^x(b, P, [\![P]\!]_{\mathcal{L}_2} s)$$

En lien avec ce travail, une interface graphique de développement a été produite. Celle-ci permet de produire des programmes graphiques grâce à des spécifications de variables similaires à celles retrouvées dans le langage $\mathcal{L}_2$.

Chapitre 5

LyeeBuilder

5.1 Introduction

En parallèle à la traduction de programmes d'un langage impératif contenant des tableaux vers un langage déclaratif, une implantation permettant de créer des programmes à partir de spécifications Lyee a aussi été produite. Telle que présentée dans [18, 19], celle-ci permet de créer une application à l'aide d'une interface graphique et de définir la spécification de chaque composant. À partir de la disposition des éléments graphiques et des spécifications, le code JAVA permettant de faire fonctionner le programme sous n'importe quelle plateforme est généré automatiquement. L'environnement de développement Eclipse a été utilisé comme base pour développer LyeeBuilder.

La présentation de LyeeBuilder sera faite ainsi. Premièrement, les propriétés Lyee seront présentées. Celles-ci permettent de spécifier le comportement des composants. Ensuite, la syntaxe utilisée pour faire les spécifications sera montrée. Par la suite, les composants disponibles pour la création d'applications seront énumérés ainsi qu'une explication de ceux-ci. Finalement, un exemple de la création d'une application simple sera fait pour permettre de voir de façon pratique LyeeBuilder.

5.2 Propriétés Lyee

Les propriétés suivantes peuvent être définies pour un objet d'un programme créé avec LyeeBuilder :

- *Condition* : expression booléenne qui spécifie quand la définition d'un mot est calculée.
- *Définition* : expression booléenne, arithmétique ou d'action d'écran qui définit un mot.
- *I/O* : spécifie si un mot est une entrée, une sortie ou les deux.

Comme la section 5.6.2 le montre, il existe un lien très étroit entre ces propriétés et le langage $\mathcal{L}_2$.

En plus de celles-ci, les propriétés suivantes sont aussi présentes dans LyeeBuilder :

- *Security* : spécifie le niveau de sécurité lié au mot.
- *Type* : spécifie le type du mot (int, float, string, boolean, button).
- *whenOutputted* : expression booléenne spécifiant quand un mot peut être transmis en sortie.
- *whenInputted* : expression booléenne spécifiant quand un mot peut accepter une entrée.
- *whenEnabled* : expression booléenne spécifiant quand un bouton ou un champ est disponible.

5.3 Syntaxe

La syntaxe utilisée pour définir les programmes est présentée dans le tableau 5.1 où a représente une expression arithmétique, s une chaîne de caractères, b une expression booléenne, *ScreenAction* une action pouvant être effectuée sur une fenêtre et " ^ " l'opérateur utilisé pour concaténer deux chaînes de caractère.

Comme il est possible de remarquer, cette syntaxe est très proche de celle utilisée pour le langage $\mathcal{L}_2$ présenté précédemment.

5.4 Composants

Tels que présentés ci-dessous, plusieurs composants graphiques sont disponibles pour créer un programme dans LyeeBuilder. La liste suivante représente ceux qui sont actuellement disponibles.

- *LyeeTextField* : Ce composant représente un champ de texte qui est utilisé pour

Syntaxe des expressions arithmétiques :

$a ::= v \mid Id \mid (a) \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$

Id= L'identifiant d'un
v= Un entier

Syntaxe des expressions de chaînes de caractères :

$s ::= Constant\ String \mid s_1 \hat{} s_2$

L'opérateur "^" permet la concaténation de deux chaînes de caractères

Syntaxe des expressions booléennes :

$$\begin{array}{lllll} b ::= & true & \mid false & \mid (b) & \\ \mid & b_1 == b_2 & \mid b_1 != b_2 & \mid b_1 \,\&\&\, b_2 & \mid b_1 \,||\, b_2 \\ \mid & s_1 == s_2 & \mid s_1 != s_2 & & \\ \mid & a_1 == a_2 & \mid a_1 != a_2 & & \\ \mid & a_1 < a_2 & \mid a_1 \leq a_2 & \mid a_1 > a_2 & \mid a_1 \geq a_2 \end{array}$$

Syntaxe des actions d'écran :

$$\begin{array}{lll} ScreenAction & ::= & ScreenName.open \mid ScreenName.close \\ & \mid & ScreenAction \,\&\&\, ScreenAction \end{array}$$

TAB. 5.1 – Syntaxe.

interagir avec l'utilisateur en entrée ou en sortie. Son type est défini par la propriété *Type*. Lorsqu'il s'agit d'une entrée (le champ *I/O* défini à *I*), la propriété *whenInputted* peut être utilisée pour définir une condition pour valider l'entrée. Lorsqu'il s'agit d'une sortie, les propriétés *Condition*, *Definition* et *whenOutputted* sont utilisées par le programmeur pour définir le comportement du composant.

- *LyeePasswordField* : Cet élément est très semblable au précédent à l'exception qu'il est utilisé pour les mots de passe. Ainsi, il est plutôt utilisé en tant qu'entrée.
- *Label* : Ce composant est principalement utilisé pour identifier le nom d'un autre composant à un utilisateur. Par exemple, il peut être défini à "Nom : " à gauche d'un *LyeeTextField* pour signifier à l'utilisateur que le programme s'attend à avoir un nom dans ce dernier. Il peut aussi être utilisé comme sortie de la même façon qu'un *LyeeTextField*.
- *Button* : Ce composant, qui doit être de type "Button", est utilisé pour l'interac-

tion avec l'utilisateur. Par exemple, il peut être utilisé pour déclencher des calculs ou lancer une action associée à une fenêtre. Ainsi, pour ouvrir une fenêtre appelée "s1", sa définition serait : "s1.open". Plusieurs actions peuvent être associées à un même bouton en utilisant l'opérateur && pour les séparer. De plus, il est à noter que le nom "$s0$" est utilisé pour représenter le programme.

- *ListBox* : Ce composant est habituellement utilisé en entrée et en sortie. En entrée, sa valeur correspond à l'item sélectionné par l'utilisateur. En sortie, la valeur reçue est ajoutée à la liste.
- *CheckBox* : Ce composant est habituellement utilisé comme entrée. Sa valeur correspond à l'état de la case (cochée ou non).
- *Composants Lyee DB* : Ces composants permettent l'interaction avec une base de données. Il est ainsi possible de stocker des données dans une base de données et de les afficher dans des champs texte ou des grilles.

À ceux-ci sont associées les propriétés précédemment définies. En utilisant la syntaxe de la section précédente, un programmeur peut définir les propriétés et facilement obtenir un programme qui fonctionne selon ce qu'il a spécifié. Ici le terme programmeur est utilisé pour différencier la personne qui crée le programme par rapport à un utilisateur, celui qui l'utilise. Cependant, pour créer un programme dans LyeeBuilder, aucune notion de programmation n'est nécessaire puisqu'il ne s'agit que de définitions.

Pour faire le parallèle avec le langage $\mathcal{L}_2$, les composants sont semblables aux variables de celui-ci. En effet, à ceux-ci sont associées des définitions et l'ensemble des composants forme le programme complet.

5.5 Implantation

L'implantation de LyeeBuilder repose sur la plateforme Eclipse [11]. Ce choix a été fait à cause de l'extensibilité de cet IDE ainsi que des plug-ins qui le composent. L'implantation que nous avons faite peut être divisée en deux parties : l'éditeur visuel et le générateur de code.

L'éditeur visuel est construit comme un plug-in par-dessus le Visual Editor Project (VE) [12]. Ce dernier permet de faciliter la conception visuelle pour les applications JAVA. Puisque le code produit par LyeeBuilder est lui aussi du JAVA, il était intéressant de partir d'un code de base déjà existant. La palette qui présente les composants dans VE a été adaptée pour afficher les composants Lyee identifiés à la section 5.4 au lieu des composants JAVA habituels. En réalité les composants Lyee sont basés sur des compo-

sants JAVA auxquels ont été ajoutés des canaux de communication et des propriétés additionnelles pour la spécification du comportement par le programmeur. De plus, VE est responsable de la génération du code JAVA utilisé pour afficher le comportement graphique des composants, comme leur disposition, leur taille, etc.

Une fois les composants disposés et leurs définitions faites, le code nécessaire pour le fonctionnement de l'application doit être généré. Pour ce faire, le Java Emitter Templates d'Eclipse Modeling Framework (EMF) [10] a été utilisé. Lorsqu'activé, il génère le code additionnel dans les classes JAVA qui permet aux objets Lyee d'utiliser le package Lyee Calculus. Pour ce faire, il lit le code source pour identifier tous les composants Lyee attachés à une fenêtre. Puis il génère le code pour créer les canaux et pour lancer les processus nécessaires. Finalement, il ajoute des appels de fonctions à la méthode principale du programme pour initialiser toutes les fenêtres Lyee.

5.6 Exemples

Cette section présente deux exemples relatifs à LyeeBuilder. Le premier est un exemple des étapes de création d'un programme simple. Le deuxième montre un programme plus complet et illustre les définitions de ses composants.

5.6.1 Exemple 1

Premièrement, le programmeur doit créer un nouveau projet de type "Lyee Application Project" tel que montré à la figure 5.1.

Une fois qu'Eclipse a terminé de créer le nouveau projet vierge, l'environnement présenté au programmeur est tel que présenté à la figure 5.2. L'utilisation de l'environnement est très intuitive. Au centre se retrouve la première fenêtre du projet. À droite de celle-ci se trouve la palette qui présente les éléments disponibles pour y ajouter de nouveaux composants. En dessous de la fenêtre se trouve son code JAVA. Il a été laissé sur cette figure pour montrer que l'application résultante était bien en JAVA, mais le programmeur n'a jamais à y toucher, il travaille plutôt avec cet élément fermé comme le montre la figure 5.3.

Ainsi, tel qu'expliqué précédemment, le programmeur n'a qu'à glisser les composants désirés et définir les propriétés nécessaires telles qu'illustrées à la figure 5.3. Le

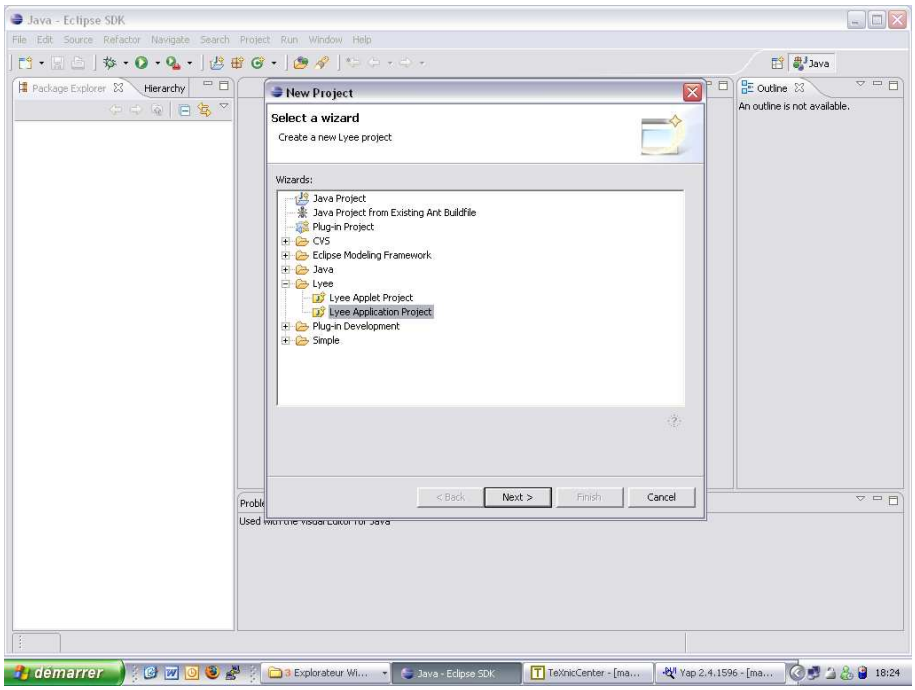

FIG. 5.1 – Création d'un nouveau projet.

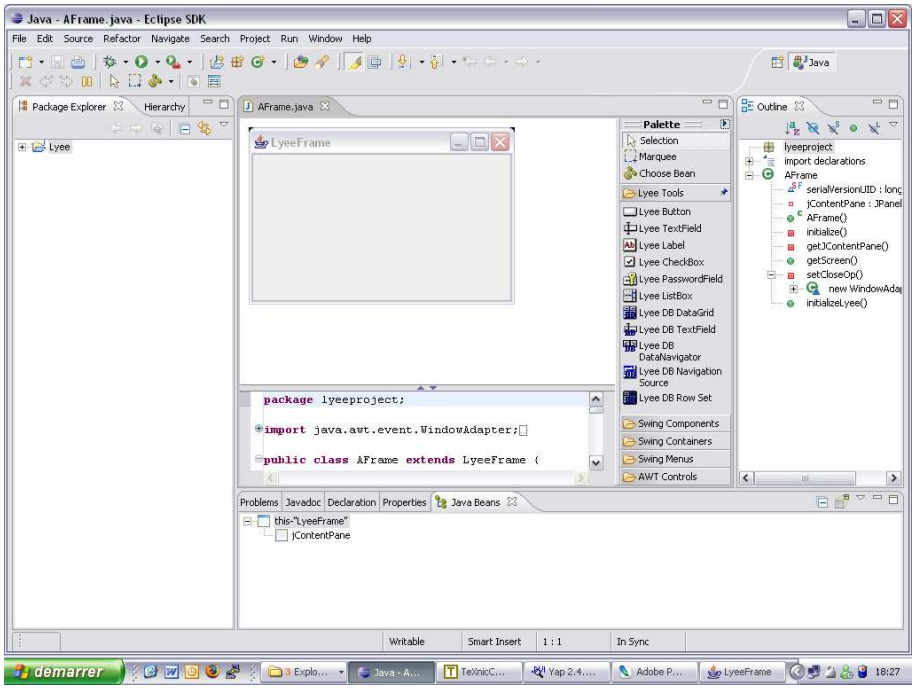

FIG. 5.2 – Nouveau projet.

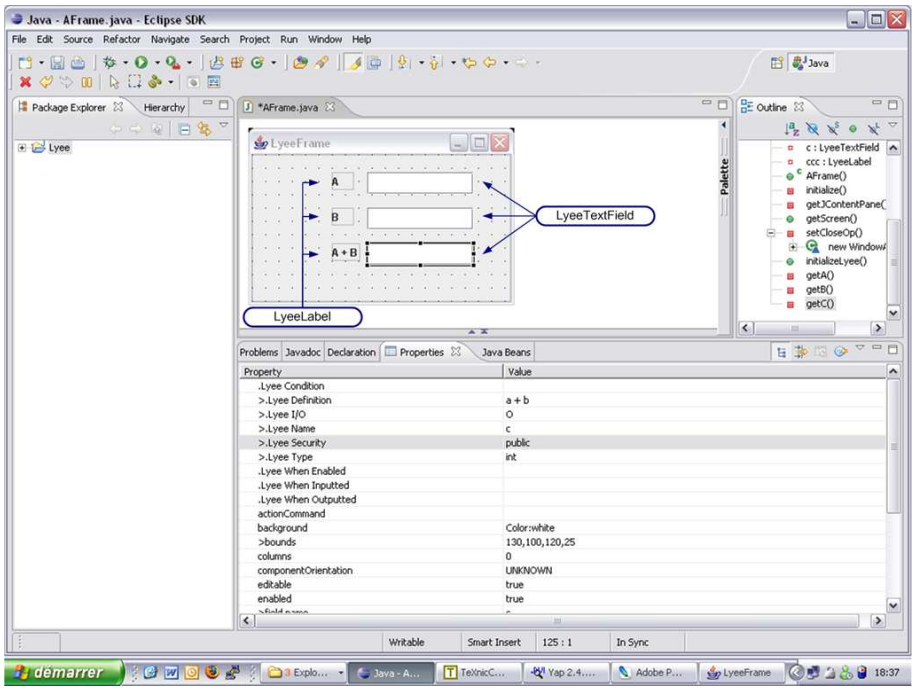

FIG. 5.3 – Création de l'application.

programmeur utilise la section sous la fenêtre servant à la création visuelle. L'application donnée en exemple est très simple, elle ne fait qu'additionner les nombres entrés dans les boîtes "A" et "B" dans celle du bas. Cependant, il est possible de créer des applications plus complexes avec plusieurs fenêtres.

Une fois l'application terminée, le programmeur n'a qu'à demander la génération de code comme à la figure 5.4 et l'application est prête à être exécutée. La figure 5.5 montre le résultat final de l'exemple.

5.6.2 Exemple 2

L'exemple suivant présente un programme simple et les définitions de ses composants. Le programme en question est représenté à la figure 5.6 et sa définition dans le tableau 5.2.

Sans faire chacune des définitions du programme, quelques-unes seront expliquées pour permettre de bien comprendre comment la spécification fonctionne :

- b : La définition de la variable b est égale au résultat de l'équation qui a été spécifiée par le programmeur soit : $(a * e) - d$. Ainsi, lorsque les valeurs des variables a, e

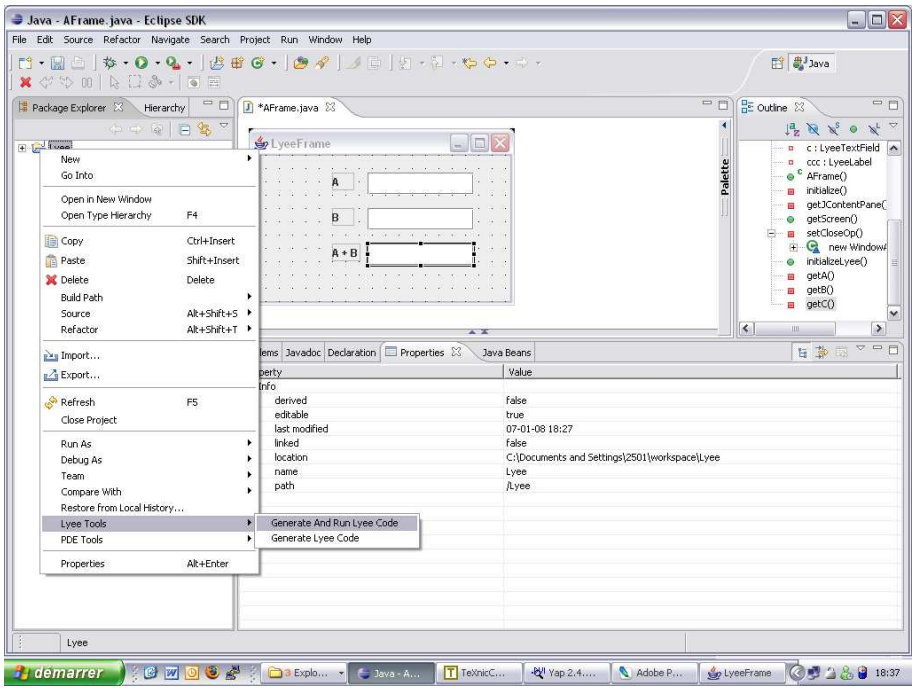

FIG. 5.4 – Génération du code.

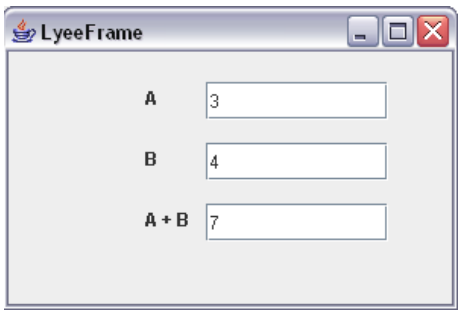

FIG. 5.5 – Application résultante.

et d sont disponibles, et que la condition est vraie, le résultat est affiché.

- c : En raison de la définition de *whenOutputted*, la définition de la variable c ne sera affichée que lorsque le bouton *bCompute* sera appuyé.
- *bNext* : Lorsqu'il est appuyé, ce bouton permet de faire apparaitre la deuxième fenêtre de l'application ($frame2$). Ce bouton n'est activé que si la variable a est différente de 0.
- *bExit* : Tout comme $bNext$ ce bouton permet aussi d'interagir avec l'application. Cependant, celui-ce permet de fermer toutes les fenêtres et de quitter le programme.

À la lumière de ces définitions, un lien évident existe entre ces dernières et le langage

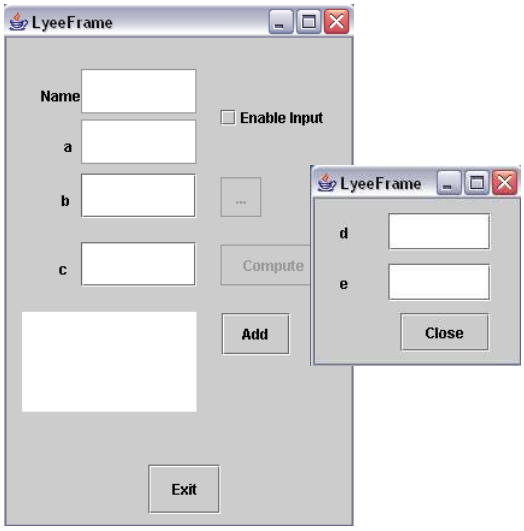

FIG. 5.6 – Programme de l'exemple 2.

$\mathcal{L}_2$. En effet, les définitions utilisant des expressions arithmétiques comme celle de c correspondent à des expressions de $\mathcal{L}_2$.

$$[c \mapsto b * b]$$

En ce qui a trait aux conditions, elles correspondent à des expressions conditionnelles du langage $\mathcal{L}_2$. Par exemple, la définition de b en regard à sa condition serait :

$$[b \mapsto if(e \leq 10, (a * e) - d, b)]$$

Ainsi si la condition est vraie, la valeur de b vaut $(a*e)-d$ et reste inchangée sinon.

5.7 Conclusion et travaux futurs

Ainsi, l'implantation de LyeeBuilder effectuée montre bien que la création de logiciels est grandement simplifiée en utilisant un langage semblable au langage $\mathcal{L}_2$ présenté précédemment. Il est aussi intéressant de noter que bien que le programme résultant

	Condition	Definition	Type	I/O	Security	whenInputted	whenOutputted	whenEnabled
name			string	I				cInput
a				I		a != 0		
b	e $\leq$ 10	(a*e)-d		O				
c		b*b					bCompute	
list		name ^ " " ^ c	string	IO			bAdd	
selected		list		O			bAdd	list != ""
cInput				I				
bNext		frame2.open	Button					a != 0
bCompute			Button					a !=0 && b !=0
bAdd			Button					
bExit		s0.close	Button					
d				I				
e				I		d != 0		
bExit		frame2.close	Button					

TAB. 5.2 – Définitions.

est en JAVA,un autre langage de programmation aurait aussi pu être utilisé comme langage de destination.

En plus de composants supplémentaires, une extension de LyeeBuilder permettant d'importer des programmes écrits dans un langage impératif serait très intéressante. L'intégration de cette nouvelle fonctionnalité pourrait être complètement intégrée à l'environnement de développement. Pour ce faire, il faudra cependant préalablement ajouter le point fixe au langage utilisé par LyeeBuilder. En effet, ce dernier n'offre pas cette possibilité présentement. En considérant la simplication des expressions $\mu^x_{b,P}\, P'$ du langage $\mathcal{L}_2$ présentée à la fin du chapitre précédent, simplification faisant appel à une fonction récursive *Frec* définie comme suit :

$$\mathit{Frec}^x(b, P, s) = \textbf{if } \neg[\![b]\!]_{B_2} s \textbf{ then } s\ x \textbf{ else } \mathit{Frec}^x(b, P, [\![P]\!]_{\mathcal{L}_2} s)$$

il serait intéressant d'étendre LyeeBuilder afin de supporter les définitions récursives. Ainsi, si une variable y est définie, en utilisant la syntaxe de $\mathcal{L}_2$, comme suit :

$$\{\, y \mapsto \mu^y_{b,P}\, P' \,\}$$

sa définition dans LyeeBuilder aura la forme suivante :

$$[\, y \mapsto \left\{ \begin{array}{lcl} [\![b]\!] & \rightarrow & [\![P]\!](y) \\ \neg[\![b]\!] & \rightarrow & y \end{array} \right\}^{P'} \,]$$

Dans cette notation, la mémoire s n'apparaît pas car elle est implicite dans Lyee-

Builder, P' joue le rôle d'une mémoire locale initiale pour le calcul de la variable y et finalement la notation

$$\left\{ \begin{array}{ccc} [\![c_1]\!] & \rightarrow & [\![def_1]\!] \\ & \vdots & \\ [\![c_n]\!] & \rightarrow & [\![def_n]\!] \end{array} \right\}$$

précise que la définition def_i[1] sera évaluée dès que la condition c_i est évaluée à *true*[2]. La différence avec la sémantique du langage $\mathcal{L}_2$ est que cette vérification/évaluation se fait de manière continue ; autrement dit, quand def_i est évaluée, on répète le processus en vérifiant à nouveau si une nouvelle condition c_j est évaluée à *true*, auquel cas on évaluerait la définition def_j et on répéterait de nouveau le processus.

Par exemple, le programme suivant :

$$\{\, y \mapsto \mu^{y}_{i<5,[y\mapsto y+i,i\mapsto i+1]}\, [i \mapsto 0, y \mapsto 1] \,\}$$

serait défini de la manière suivante en LyeeBuilder :

$$[\, y \mapsto \left\{ \begin{array}{ccc} i < 5 & \rightarrow & [\![y \mapsto y+i, i \mapsto i+1]\!](y) \\ i \not< 5 & \rightarrow & y \end{array} \right\}^{[i\mapsto 0, y\mapsto 1]} \,]$$

Il est aussi à noter qu'au moment du dépôt de ce mémoire, une implantation de la traduction de l'impératif au déclaratif permettant de passer de Cobol à $\mathcal{L}_2$ est en cours de développement.

```
1  PROCEDURE DIVISION.
2              ACCEPT x.
3              IF x < 5 THEN
4                      MOVE 4 to y
5              ELSE
6                      MOVE 8 to y
7              END-IF
8              PERFORM UNTIL x < 100
9              ADD x, x GIVING z
10         END-PERFORM
11 STOP RUN.
```

[1] Chaque def_i correspond en fait à un sous-programme nécessaire pour la définition de la variable en question, i.e y.

[2] La fonction d'évaluation $[\![\_]\!]$ reste à définir (elle ne correspond pas forcément à $[\![\_]\!]_{\mathcal{L}_2}$ et $[\![\_]\!]_{B_2}$).

En utilisant le programme Cobol précédent avec l'implantation de la traduction, le résultat suivant est produit :

$$\left\{ \begin{array}{l} x \mapsto input, \\ z \mapsto \mu^{z}_{(!(x[1]<100))[z->x[1]+x[1]]}, \\ y \mapsto if(x[1] < 5, 4, 8) \end{array} \right\}$$

Des travaux concernant l'intégration de la traduction de Cobol avec LyeeBuilder sont en cours. Ceux-ci visent à permettre de traduire des programmes Cobol dans le but d'en faciliter la maintenance ou tout simplement pour les porter à JAVA.

Chapitre 6

Conclusion

La spécification de définitions pour créer des programmes permet de simplifier la tâche du programmeur. En effet, ce dernier n'a plus à se soucier de l'ordre d'exécution des instructions et peut retrouver très facilement la définition d'une variable puisqu'elle n'est pas éparpillée dans le code. Dans cette optique, une fonction de conversion permettant de traduire des programmes écrits dans un langage impératif vers cette forme déclarative permettrait de faciliter la maintenance des programmes existants.

Une telle approche existe déjà pour un langage impératif simple. Ce langage permet d'utiliser des expressions arithmétiques, booléennes ainsi que les affectations, les boucles et les expressions conditionnelles. Parallèlement, un langage déclaratif, utilisé comme destination de la traduction, est aussi défini. La conversion de cette approche permet donc de passer de ce langage impératif vers un langage déclaratif. Son fonctionnement est très similaire à la coupe de programmes. En effet, à partir d'un code impératif, elle permet d'isoler la définition de chacune des variables.

Dans le but d'agrandir l'ensemble des programmes pouvant être traduits vers le langage déclaratif, nous avons étendu l'approche de conversion. À celle-ci, nous avons ajouté des tableaux ainsi que la notion d'entrées et de sorties. Pour ce faire, certaines notions présentes dans le langage original ont dû être adaptées et d'autres ont dû être ajoutées. Par exemple, des étapes de pré et posttraduction sont maintenant nécessaires pour traduire les programmes tout en conservant leur sémantique. Ainsi, suite à ce travail, il est maintenant possible de traduire des programmes écrits dans un langage impératif avec ces nouvelles notions vers un langage déclaratif plus facile à maintenir.

Toujours dans la même optique, nous avons procédé à l'implantation d'une interface de développement permettant de produire des programmes en disposant des composants

graphiques et en leur associant des définitions. Ainsi, une personne désirant créer un programme n'a qu'à glisser les composants graphiques désirés dans la fenêtre et spécifier leurs définitions. Pour ce faire plusieurs composants graphiques tels que les *textfields*, *labels*, *buttons* et plusieurs autres sont disponibles. En ce qui a trait aux définitions, elles sont très similaires au langage $\mathcal{L}_2$ utilisé comme destination de la traduction. Ainsi, il est maintenant possible pour une personne sans connaissance réelle en programmation de créer des programmes à partir de simples spécifications.

Finalement, il est à noter que beaucoup de travail reste à faire avant que toutes les fonctionnalités intéressantes fournies par les langages de programmation actuellement utilisés soient introduites dans les langages et la conversion présentés. Par exemple, des structures de données plus complexes seraient probablement utiles. Celles-ci nécessiteraient sûrement l'ajout de la notion de pointeurs avec tous les problèmes qui en découlent. De plus, les procédures (fonctions) devraient être au moins supportées par le langage impératif utilisé comme source de la conversion. Pour ce faire, une analyse interprocédurale devrait probablement être effectuée. Une simplification des programmes résultant de la traduction serait aussi la bienvenue. En effet, lors de la traduction, une définition est créée pour toutes les variables temporaires du programme. Celles-ci sont superflues dans $\mathcal{L}_2$ et pourraient être éliminées sans changer le comportement du programme. En ce qui a trait à l'interface de développement LyeeBuilder, des composants supplémentaires seraient un atout pour augmenter la diversité des programmes pouvant être produits. De plus, il serait intéressant de porter le travail fait au niveau de la conversion de programmes à LyeeBuilder. L'intégration de cette nouvelle fonctionnalité pourrait être complètement intégrée à l'environnement de développement. Pour ce faire, il faudra cependant préalablement ajouter le point fixe au langage utilisé par LyeeBuilder. En effet, ce dernier n'offre pas cette possibilité présentement. Ainsi, il pourrait éventuellement être possible de récupérer des programmes écrits dans des langages impératifs tels que COBOL pour les rendre plus facile à maintenir.

Bibliographie

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] P. T. Breuer and K. Lano. Creating specifications from code : Reverse-engineering techniques. *Journal of Software Maintenance : Research and Practice*, 3 :145–162, 1991.

[3] Peter T. Breuer and Jonathan P. Bowen. Decompilation : the enumeration of types and grammars. *ACM Trans. Program. Lang. Syst.*, 16(5) :1613–1647, 1994.

[4] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4) :66–74, 2004.

[5] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, Brisbane, Australia, Juillet 1994.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4) :451–490, October 1991.

[7] A. De Lucia. Program slicing : methods and applications. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 142–149, Florence, Italy, 2001.

[8] Helen M. Edwards and Malcolm Munro. Recast : reverse engineering from cobol to ssadm specification. In *ICSE '93 : Proceedings of the 15th international conference on Software Engineering*, pages 499–508, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[9] Michael D. Ernst. Slicing pointers and procedures (abstract). Technical Report MSR-TR-95-23, Microsoft Research, Redmond, WA, January 13, 1995.

[10] Eclipse Foundation. Eclipse Modeling - Eclipse Modeling Framework Project (EMF) - Home. http://www.eclipse.org/emf/, 2007.

[11] Eclipse Foundation. eclipse.org. http://www.eclipse.org, 2007.

[12] Eclipse Foundation. Visual Editor Project. http://www.eclipse.org/vep/, 2007.

[13] D. Godbout, B. Ktari, and M. Mejri. A formal translation from an imperative language with array to a declarative language. In H. Fujita and P. Johannesson, editors, *New Trends in Software Methodologies, Tools and Techniques*, pages 83–99. IOS Press, October 2006. Proceedings of the 5th international workshop on Lyee methodology (SoMeT'06).

[14] William E. Howden and Suehee Pak. The derivation of functional specifications from source code. In *APSEC '96 : Proceedings of the Third Asia-Pacific Software Engineering Conference*, page 166, Washington, DC, USA, 1996. IEEE Computer Society.

[15] Lin Hu, Mark Harman, Robert M. Hierons, and David Binkley. Loop squashing transformations for amorphous slicing. In *WCRE '04 : Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 152–160, Washington, DC, USA, 2004. IEEE Computer Society.

[16] J. Jiang, X. Zhou, and D.J. Robson. Program slicing for c-the problems in implementation. In *Software Maintenance, 1991., Proceedings. Conference on*, pages 182–190, Sorrento, 1991.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988. P C 88 :2.

[18] B. Ktari, H. Fujita, M. Mejri, and D. Godbout. Toward a new software development environment. *Journal of Knowledge-Based Systems*, 2007. sous presse, 23 pages.

[19] B. Ktari, M. Mejri, D. Godbout, and H. Fujita. Lyeebuilder. In Hamido Fujita and Mohamed Mejri, editors, *New Trends in Software Methodologies, Tools and Techniques*, pages 83–99. IOS Press, September 2005. Proceedings of the 4th SoMeT_W05.

[20] M. Mbarki, M. Mejri, and B. Ktari. From imperative to declarative languages : A formal translation. In Hamido Fujita and Paul Johannesson, editors, *New Trends in Software Methodologies, Tools and Techniques*, pages 153–177. IOS Press, September 2004. Proceedings of the 3nd International Workshop on Lyee Methodology, Leipzig, Germany.

[21] Mohamed Mbarki. Paradigmes des langages de programmation : De l'imperatif vers le declaratif. Master's thesis, Universite Laval, April 2004.

[22] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 :348–375, August 1978.

[23] A. Mycroft, A. Ohori, and S. Katsumata. Comparing type-based and proof-directed decompilation. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 362–367, Stuttgart, Germany, 2001.

[24] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *ESOP '99 : Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 208–223, London, UK, 1999. Springer-Verlag.

[25] F. Negoro. Principle of Lyee software. In *the 2000 International Conference on Information Society in 21st Century (IS2000)*, pages 121–189, Aizu, Fukushima, Japan, November 2000.

[26] F. Negoro and I. Hamid. A proposal for intention engineering. In *the 5th East-European Conference Advances in Databases and Information System (ADBIS'2001)*, Vilnius, Lithuania, September 2000.

[27] G. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5 :223–255, 1977.

[28] D. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1–2) :411–440, 6 December 1993.

[29] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *CC '96 : Proceedings of the 6th International Conference on Compiler Construction*, pages 136–150, London, UK, 1996. Springer-Verlag.

[30] Mark Weiser. Program slicing. In *ICSE '81 : Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.