Institut National Polytechnique

DEA **I**nformatique **F**ondamentale et **P**arallélisme

Année: 96-97

Laboratoire d'Informatique et de Mathématiques Appliquées
E.N.S.E.E.I.H.T.
Directeur : *G.Padiou*

Conception d'un Langage Fonctionnel d'Acteur et Réalisation de son Compilateur.

Auteur : *Fabien DAGNAT*

Directeur de recherche: *P.Sallé*
Responsable du stage: *M.Pantel*

Equipe Vestale
Projet ML-ACT

Mots clés : Langage Fonctionnel, Programmation Concurrente par Acteur, Typage, Compilation, Sémantique des langages de programmation.

Résumé : Ce rapport présente rapidement la notion de langage fonctionnel et le modèle de programmation par acteur. Puis, il expose un langage construit à partir de ces deux notions, décrit son typage, l'extraction d'un terme d'un calcul de processus afin d'analyser les communications et enfin, sa traduction en ML. Dans la deuxième partie du rapport, on se penche sur une formalisation des sémantiques statiques et dynamiques du langage que l'on a conçu.

Remerciements

Je remercie M. Gérard Padiou, Responsable INP du Laboratoire d'Informatique et de Mathématiques Appliquées, pour m'avoir accueilli dans son laboratoire.

Je remercie M. Patrick Sallé, responsable de l'équipe Vestale pour m'avoir accueilli.

Je remercie M. Marc Pantel pour son encadrement et pour ses conseils, toujours pertinents, durant mon stage.

Je remercie M. Marcel Gandriau et M. Jean-Louis Colaço pour leur aide.

Merci à tous les stagiaires qui contribuent à l'ambiance sympathique qui règne dans le laboratoire.

Table des matières

1 INTRODUCTION **7**
1.1 Travaux de l'équipe d'accueil. 7
1.2 Les acteurs. 8
1.3 CAP :un calcul de processus. 8
1.4 Objectifs du stage. 9
1.5 Syntaxe concrète versus syntaxe abstraite. 10
1.6 La sémantique naturelle. 10
1.7 Plan du rapport. 11

2 UN NOYAU FONCTIONNEL **13**
2.1 Le choix du fonctionnel. 13
2.2 Présentation de ML. 13
2.2.1 Objective CaML. 14
2.2.2 Structures de données. 14
2.2.3 Le filtrage. 15
2.2.4 Les fonctions. 15
2.3 La syntaxe concrète. 16
2.4 La syntaxe abstraite. 18
2.5 Le système de type. 19
2.5.1 Fonctionnement du typage. 19
2.5.2 Des fonctions utiles. 19
2.5.3 Les règles de typage 20
2.6 L'unification. 24
2.7 Les réalisations. 27
2.8 Extensions envisageables. 28

3 LES ACTEURS **29**
3.1 La programmation par acteur. 29
3.2 Un langage d'acteur. 30
3.3 Insertion dans le système de type. 33
3.4 Des exemples. 35
3.4.1 Un système d'impression en réseau. 35
3.4.2 La construction de groupe de diffusion. 36
3.4.3 Une matrice 37
3.5 Réalisation 40
3.6 Conclusion et perspectives. 40

4 L'EXTRACTION D'UN TERME CAP **43**
4.1 Présentation de CAP. 43
4.2 Sa syntaxe. 44
4.3 Un exemple. 45
4.4 Sémantique de la génération de CAP. 45
4.4.1 Contexte de l'extraction. 45
4.4.2 Première étape de traduction. 46
4.4.3 Deuxième étape de traduction. 51
4.5 Réalisations. 54
4.6 Un exemple. 54
4.7 Conclusion 56

5 LA GÉNÉRATION DE CODE CAML **57**
5.1 Les outils 57
5.2 Les structures 57
5.3 L'implantation 58
5.4 La stratégie de traduction 62
5.5 Réalisations 66
5.6 Un exemple 66
5.7 Perspectives de développement 70

6 SÉMANTIQUE FORMELLE DE MINI-ACT. **71**
6.1 La syntaxe de mini-Act. 71
6.2 Le choix d'un formalisme 72
6.3 Le contexte d'évaluation et les valeurs sémantiques. 73
6.3.1 Ensembles de noms. 73
6.3.2 Contextes 73
6.3.3 Fonctions partielles. 73
6.3.4 Substitution. 74
6.3.5 Valeurs sémantiques. 74
6.3.6 Les configurations. 76
6.4 La sémantique dynamique de mini-Act. 76
6.4.1 L'opérateur de filtrage. 76
6.4.2 Les règles de réduction des configurations. 77
6.5 Un exemple de réduction 81
6.6 Conclusion 83

7 LE TYPAGE DE MINI-ACT. **85**
7.1 Les types. 85
7.2 La sémantique statique de mini-Act. 86
7.2.1 Le filtrage. 86
7.2.2 Les expressions. 87
7.3 Commentaires 89
7.4 Conclusion 91

8 CONCLUSION **93**

Chapitre 1

INTRODUCTION

Mon stage s'est déroulé au Laboratoire d'Informatique et de Mathématiques Appliquées de l'École Nationale Supérieure d'Electronique, Electrotechnique, Informatique et Hydraulique de Toulouse, membre de l'Institut de Recherche en Informatique de Toulouse (Unité Mixte de Recherche n° 5055 du CNRS) dans l'équipe *Vestale* du pôle *Programmation, Systèmes et Algorithmes.* Cette équipe est composée à l'ENSEEIHT des membres permanents suivant :

- Marcel Gandriau,
- Christiane Massoutié,
- Marc Pantel,
- Patrick Sallé.

1.1 Travaux de l'équipe d'accueil.

Depuis quelques années, l'équipe s'intéresse aux langages de programmation. Dans l'ordre chronologique, on trouve :

- **Plasma** (P. Sallé) : programmation par acteurs.
- **Plasma++** (P. Sallé) : extension objet de **Plasma**.
- **Alog** (F. Carré) : acteurs et logique.
- **Plasma II** (P. Sallé) : version répartie (multiposte et machine parallèle).
- **Ciel** (M. Gandriau) : Classes et Instances En Logique.
- **FOL** (M. Pantel) : Fonctions et Objets dans un Langage.

ainsi que trois machines virtuelles pour les langages applicatifs : **LILA** (machine séquentielle) et **SMART** (version répartie), **FLAM** (dérivée de la machine de Krivine).

1.2 Les acteurs.

Pour faciliter la tâche des programmeurs, les langages de programmation sont devenus de plus en plus abstraits. Ces langages de haut niveau permettent de se concentrer sur les algorithmes plutôt que sur les détails d'implantation liés à telle ou telle machine. Mais le nombre de langage devenant trop important, la recherche s'est dirigée vers le développement de modèles de programmation suffisamment flexibles pour que l'on puisse les intégrer simplement et efficacement dans les différents langages utilisés couramment.

D'autre part, l'évolution du matériel vers des machines de plus en plus complexes mais puissantes et le monde industriel plus exigeant au niveau économique, ont provoqué une mise en avant de la notion de réutilisabilité par rapport à celle d'efficacité.

Le succès important que rencontre la notion d'objet actuellement, est une preuve de la mise en avant de ces deux aspects.

Le modèle d'acteur a été développé dans cet esprit, mais aussi avec l'objectif de permettre la spécification et l'implantation aisée d'applications distribuées et parallèles. Ce modèle abstrait les problèmes de bas niveau liés à la répartition, tels que les protocoles de communication entre entités ou leurs synchronisations.

Le panorama précédent a montré l'intérêt que l'équipe a pour les acteurs. Parmi les nombreux développements qu'a connu ce modèle, notre référence est celui de Agha présenté dans [Agh86].

L'acteur, abstraction de base du modèle, est un «agent» autonome qui encapsule des données et des procédures comme un objet. Les acteurs communiquent entre eux par envoi asynchrone de messages. A l'image du *mail* d'internet, chaque acteur a une *adresse postale*, son identité, qu'il peut communiquer. Ainsi, la topologie du médium de communication est dynamique. Il est également composé d'un *comportement*, un programme qui indique sa réaction à la réception de tel ou tel message. Ce comportement peut changer de façon radicale. Pour illustrer ce fait, Agha dans [AH92] présente l'exemple d'un acteur gérant un compte bancaire qui, à la suite de la réception d'un message particulier, devient livreur de pizza. Ce changement d'état est généralement spécifié de façon statique, mais il peut aussi l'être de façon dynamique via la réception d'un comportement.

Enfin, il est important de remarquer que contrairement à un processus, un acteur n'est pas une entité qui effectue un calcul puis disparaît. C'est un objet réactif qui n'exécute des calculs que lors de la réception d'un message.

1.3 CAP :un calcul de processus.

La recherche actuelle concentre de nombreux efforts en vu d'obtenir une certaine sûreté de fonctionnement des systèmes informatiques. Or, une des premières choses qui doit être sûre est la conception de ces systèmes, il est donc important de disposer d'outils pour vérifier la validité de tout développement.

Pour cela, dans le cadre de sa thèse, Jean-Louis Colaço, au sein de l'équipe Vestale, a développé différentes méthodes d'analyse statique de programmes basés sur les acteurs. Parmi les outils qu'il a conçu dans ce but, on s'intéressera plus particulièrement à CAP.

CAP est un calcul de processus dans lequel les acteurs et leur système de communication sont des éléments primitifs. Son objectif est de fournir un formalisme élémentaire pour servir à l'étude de systèmes de types pour les langages d'acteurs.

L'inconvénient majeur de CAP est qu'étant de très bas niveau, écrire des programmes significatifs est difficile, voire impossible. Or, pour évaluer pratiquement les outils d'analyse

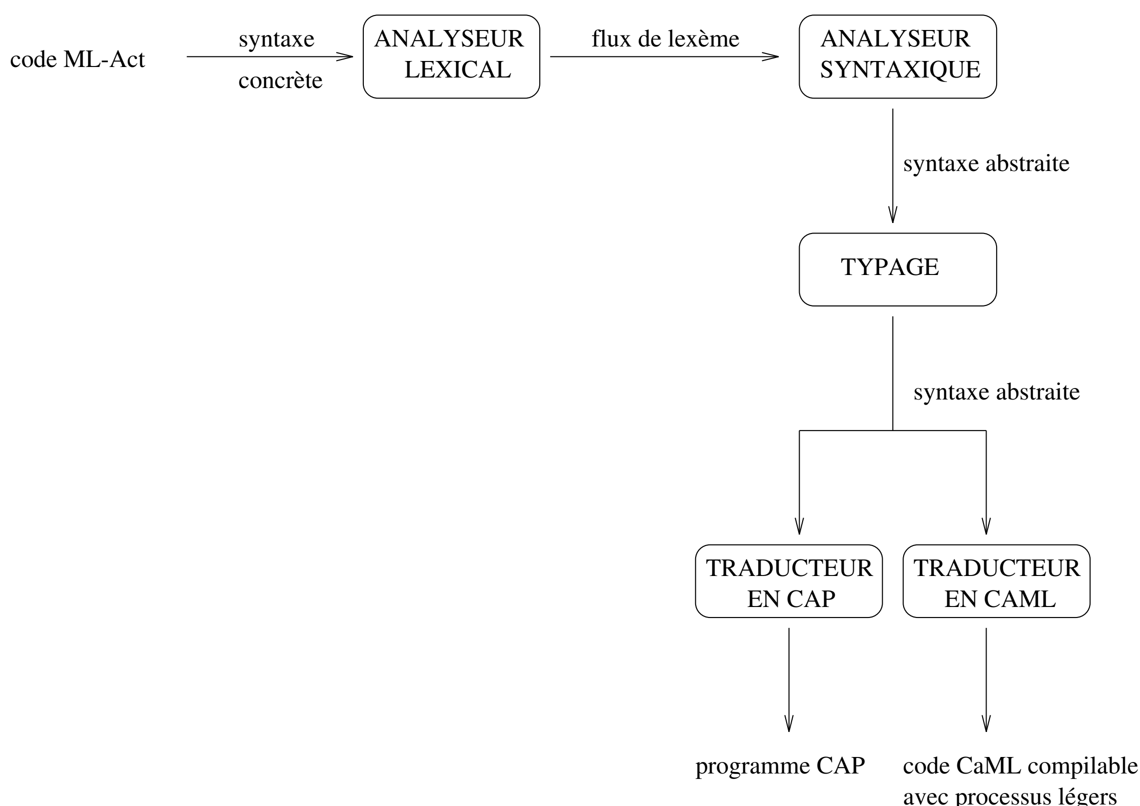
développés par l'équipe, il faudrait disposer de programmes de taille raisonnable en CAP. C'est dans ce but qu'il a été décidé de développer un langage d'acteur de haut niveau, mais qui resterait relativement simple.

1.4 Objectifs du stage.

Dans ce cadre, le travail qui m'a été confié était, dans un premier temps de définir un langage fonctionnel comportant des primitives de programmation acteur. Puis, ensuite j'ai été chargé d'en concevoir et d'en implanter un compilateur. Ce compilateur ayant pour but, à partir d'un programme, d'effectuer une vérification de type fonctionnelle, d'en extraire un terme CAP et de générer un programme exécutable. La figure 1.1 représente schématiquement le travail que j'ai eu à réaliser.

Le langage retenu comme langage de programmation est Objective CaML, présenté dans [Ler96]. C'est un langage fonctionnel fortement typé par inférence, qui autorise néanmoins la programmation d'effets impurs, tels que l'affectation et les références. Il contient également les notions d'objets et de modules permettant une programmation très structurée et modulaire. Il servira également de langage cible du compilateur.

Figure 1.1 Réalisation d'un compilateur de ML-ACT.



1.5 Syntaxe concrète versus syntaxe abstraite.

La syntaxe d'un langage de programmation est souvent très complexe pour faciliter l'analyse syntaxique et éviter au maximum les ambiguïtés. Le deuxième objectif de cette syntaxe, dite concrète, est d'être utilisée par les programmeurs, elle doit donc être relativement naturelle et facile à retenir. Pour cela, on ajoute à la grammaire de nombreux symboles terminaux (ou mots clés) qui n'ont aucune signification intéressante d'un point de vue sémantique. Ils sont souvent désignés sous le nom de sucre syntaxique. Par exemple dans le programme suivant :

```
let a = 1 in a + a
```

Les symboles `let`,`=` et `in` n'ont aucune signification particulière en eux-mêmes. On pourrait simplifier le langage et écrire :

```
bind_in (a,1,a + a)
```

Pour rendre la sémantique plus facile à décrire, on traduit en général le langage de programmation en une structure de donnée ne contenant que les informations importantes. Pour cela, on définit alors ce que l'on appelle la syntaxe abstraite du langage. La deuxième expression ci-dessus pourrait être l'écriture en syntaxe abstraite du `let`. Dans ce rapport, après avoir présenté les syntaxes concrètes, nous expliquerons toutes les manipulations nécessaires à la compilation de ML-ACT en utilisant des syntaxes abstraites.

1.6 La sémantique naturelle.

Il existe de nombreux formalismes différents pour décrire la sémantique d'un langage de programmation. Parmi ces formalismes, nous avons choisi d'utiliser la sémantique naturelle, car elle nécessite moins de prérequis mathématiques que d'autres formalismes, comme par exemple la sémantique dénotationnelle. Celle-ci a été proposée par Kahn (voir [Kah88]) comme une extension de la sémantique opérationnelle structurée de Plotkin (voir [Plo81]). Elle consiste à définir une relation entre les programmes et les résultats : cette relation est appelée «évaluation». En général, on utilise un mécanisme de déduction pour la présenter. Des règles d'inférence données, permettent de calculer la «valeur» d'une expression en fonction des résultats de l'«évaluation» des sous-expressions qui la composent.

Un jugement (ou séquent) est donné sous la forme : $\mathcal{E} \vdash M : N$. Il signifie que l'expression M, dans l'environnement $\mathcal{E}$, produit l'expression N. Remarquons que le symbole «:» utilisé dans le jugement est un séparateur sans signification particulière. Dans le cadre de règle de typage on utilise «:» , pour une évaluation on utilise plutôt «$\Rightarrow$»...

Le mécanisme de déduction est composé d'un ensemble de prémisses, des axiomes, et d'un ensemble de règles d'inférence. Une prémisse est un jugement qui est toujours vrai. Une règle d'inférence permet d'indiquer qu'un jugement J est vrai si un ensemble d'autres jugements $\{J_i\}_{i\in[1,..,n]}$ sont vrais. Elle est représentée par :

$$\text{NOM} : \frac{J_1 \cdots J_n}{J}$$

Le premier terme du séquent en dénominateur est appelé le sujet de la règle. L'utilisation des règles se fait par unification du terme que l'on veut traiter avec les sujets des différentes règles, on utilise alors la première règle dont l'unification n'échoue pas. L'«évaluation» consiste alors, en une régression de règle en règle jusqu'à l'obtention d'axiomes.

Dans les différents systèmes d'inférence des premiers chapitres, la présence d'un terme qui n'est sujet d'aucune règle représente une erreur. Par souci de concision, on a omis toutes les règles de transmission d'erreur et toutes les règles qui causent une erreur. D'après l'hypothèse précédente, si aucune règle ne s'unifie correctement avec le terme, c'est qu'il y a une erreur.

Dans la partie du rapport qui concernera la formalisation plus précise de la sémantique de ML-Act, nous utiliserons un formalisme différent. La présentation de celui-ci, ainsi que les raisons de son choix seront présentées en introduction de cette partie.

1.7 Plan du rapport.

Ce rapport est divisé en six parties principales :

- la construction du noyau fonctionnel et son typage,
- l'ajout à ce noyau de primitives de programmation par acteurs,
- l'extraction automatique d'un terme Cap,
- la génération de code Objective CaML en utilisant des processus légers,
- la sémantique formelle d'un sous-ensemble de ML-Act,
- la formalisation du système de type.

Chapitre 2

UN NOYAU FONCTIONNEL

Le langage que nous avons conçu et qui est décrit dans ce rapport est appelé ML-ACT. Pour présenter toutes ses caractéristiques, nous allons procéder par étape, tout d'abord le noyau fonctionnel qui est l'objet de ce chapitre, puis les constructions acteurs qui seront présentées dans le chapitre suivant.

2.1 Le choix du fonctionnel.

Pour rompre avec la continuelle augmentation du nombre de nouveaux langages de programmation proposés par les laboratoires de recherche, nous avons choisi de concevoir ML-ACT comme un dérivé d'un langage existant. De plus, l'objectif final étant d'obtenir un langage sur lequel l'analyse statique soit la plus fine possible, il a été préféré un langage fonctionnel dont le système de type est beaucoup plus robuste, plus simple et plus puissant que celui de la majorité des langages impératifs.

Ce choix m'a permis de me familiariser avec les langages de la famille ML.

2.2 Présentation de ML.

Nous allons présenter le langage qui nous a servi de modèle, de langage de programmation ainsi que de langage cible pour le compilateur. ML fait partie d'une famille de langage de programmation issu du λ-calcul. En fait, son origine est le métalangage d'un système de preuve appelé LCF dû à D.Scott et R.Milner [GMM+78]. De la famille ML, qui compte de nombreux membres, nous allons utiliser le dialecte CaML qui a été développé à l'INRIA dans le cadre du projet FORMEL (devenu CRISTAL).

CaML est un langage applicatif fortement typé par inférence, un programme est une suite de définitions et d'expressions. Les définitions introduisent des noms et les lient à des valeurs. Schématiquement, l'exécution d'un tel programme consistera à construire un environnement, puis à évaluer ses expressions dans celui-ci. Il s'agit donc initialement d'un langage fonctionnel pur. Pour en faire un langage utilisable pour le développement d'application de taille importante, les effets de bords y sont également permis. Objective CaML permet, de plus, la programmation de modules et d'objets, ainsi que l'utilisation d'exceptions et de références.

Remarquons que l'un des intérêts de l'inférence de type est la possibilité de polymorphisme paramétrique implicite. En effet, lors de la liaison entre une variable et une valeur, on généralise le type de cette valeur afin de rendre la variable polymorphe, c'est-à-dire de

lui donner le type le plus général possible. Pour des raisons de décidabilité de la reconstruction d'un type, on limite le polymorphisme à la liaison lexicale (le «let»).

2.2.1 Objective CaML.

Les outils associés à CaML sont nombreux :

- un interprète (*ocaml*),
- un compilateur de byte-code (*ocamlc*),
- un interprète de ce byte-code (*ocamlrun*),
- un compilateur de code natif (*ocamlopt*),
- un générateur d'analyseurs lexicaux (*ocamllex*),
- un générateur d'analyseurs syntaxiques (*ocamlyacc*),
- un analyseur de dépendances (*ocamldep*),
- un analyseur de performances (*ocamlprof*).

De plus, une librairie relativement complète est disponible, comportant des modules plus ou moins classiques tels que les processus légers, les listes, les tables hash-codées, les accès systèmes, les interfaces graphiques...

Les exemples seront donnés sous la forme de dialogue avec l'interprète afin de donner le type de chaque entité.

2.2.2 Structures de données.

Les données manipulées par CaML sont nombreuses et lui confèrent une expressivité importante. Les principales sont :

- Les entiers, flottants,booléens, caractères et chaînes de caractères.
- Les listes, les tuples et les tableaux.
- Les enregistrements (record) et les unions (type de donnée construit).

En CaML, la mémoire est gérée automatiquement et donc il n'y a pas de notion d'allocation et de libération des structures de données. De même, il n'existe pas de notion de pointeur. En fait, le compilateur les introduit lorsque c'est nécessaire, et ceci de façon totalement transparente pour le programmeur. Cette gestion automatique permet de supprimer les nombreuses erreurs dues à une manipulation erronée de la mémoire par le programmeur.

Les structures de données sont de deux types possibles : *figées* ou *modifiables*, signifiant respectivement que la structure est figée à sa définition ou que l'on peut en modifier certains composants lors des calculs qui suivent. Les listes, les tuples et les unions sont figées alors que les tableaux et les enregistrements sont modifiables.

* `'a' ; "hello word" ; 1 ; 3,14159 ; true` sont des exemples de données de type respectivement char, string, int, float, bool.

* `[] ; [1;2;5] ; "a"::"b"::[]` sont des exemples de listes, leur type sera t list (où t est le type des éléments de la liste). Ces trois exemples montrent également les constructeurs de listes.

* `(1,4) ; ("hello",8,[1 ;2 ;3])` sont des tuples, leur type sera $t_1 * \cdots * t_n$.

* `type rationnel ={numerateur:int ;mutable denominateur:int }` est un enregistrement dont le champ dénominateur est modifiable.

* `type num = Entier of int | Reel of float` est un exemple d'union.

* `[|8 ;4 ;56|]` est un tableau de type int array.

2.2.3 Le filtrage.

En CaML, la manipulation des données se fait principalement via le filtrage. C'est-à-dire que l'on donne un modèle de la structure (le filtre), que l'on vérifie que la structure effective correspond au filtre, et, si c'est le cas, on lie les identificateurs présents dans le filtre avec les valeurs correspondantes de la structure.

Par exemple, l'expression : «`h::t ->` *suite du calcul*» appliquée à une liste, échouera si la liste est vide, et sinon, liera h à la tête de la liste et t à la queue de la liste dans la suite du calcul.

La syntaxe de l'instruction de filtrage d'une expression est :

`match` expression `with`
    filtre$_1$ `->` expression$_1$
  `|` filtre$_2$ `->` expression$_2$
    ...
  `|` filtre$_n$ `->` expression$_n$

2.2.4 Les fonctions.

Les fonctions en CaML sont, comme les valeurs, des liaisons effectuées par l'intermédiaire de l'instruction :

`let` [`rec`] ident [filtre]* = expression [`in` expression]

En ML, les fonctions sont considérées comme des valeurs, on peut donc les utiliser comme paramètre lors de l'appel d'une fonction. Elles peuvent être définies par filtrage ou, de façon classique, par un enchaînement d'instruction. On peut manipuler des fonctions anonymes (équivalent de la fonction du λ-calcul), des fonctions récursives, des fonctions polymorphes, des fonctions d'ordre supérieur et des fonctions partielles. Les fonctions peuvent de plus être écrites sous forme curryfiées ou non.

Présentons quelques exemples :

```
# let rec fib = function
        0 -> 1 |
        1 -> 1 |
        n -> fib (n-1) + fib (n-2);;

val fib : int -> int = <fun>
```

une fonction définie par filtrage qui calcul la suite de Fibonacci

```
# type num = Entier of int | Reel of float;;

type num = Entier of int | Reel of float

# let add_num_1 = function
  (Entier i,Entier j) -> Entier (i + j)
| (Entier i,Reel   j) -> Reel ((float i) +. j)
| (Reel   i,Entier j) -> Reel (i +. (float j))
| (Reel   i,Reel   j) -> Reel (i +. j);;

val add_num_1 : num * num -> num = <fun>
```

une fonction curryfiée

```
# let add_num_2 n1 n2 =
  match (n1,n2) with
    (Entier i,Entier j) -> Entier (i + j)
  | (Entier i,Reel   j) -> Reel ((float i) +. j)
  | (Reel   i,Entier j) -> Reel (i +. (float j))
  | (Reel   i,Reel   j) -> Reel (i +. j);;

val add_num_2 : num -> num -> num = <fun>
```

la même fonction non curryfiée

```
# let rec map f = function
        []   -> [] |
        x::l -> (f x)::(map f l);;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

une fonction d'ordre supérieur, dont le premier paramètre est une fonction

```
# let incrlist = map (function x -> x + 1);;

val incrlist : int list -> int list = <fun>
```

une fonction obtenue par application partielle de la précédente

REMARQUE 1 *On n'a présenté que le noyau de Objective CaML. Pour une description plus complète du langage, on peut consulter [Ler96].*

2.3 La syntaxe concrète.

La grammaire de ML-ACT est décrite dans la figure 2.1 sous la forme EBNF. Les conventions d'écriture des règles sont :

- $NT ::= E_1 \mid E_2$ signifie que NT peut se dériver soit en E_1 soit en E_2,
- $[E]$ signifie que E est facultatif,

- [E]* signifie que E est présent 0 ou n fois,
- [E]+ signifie que E est présent 1 ou n fois,
- *les non-terminaux sont en italique*,
- **les mots et symboles réservés sont en gras**,
- LES TERMINAUX SONT EN MAJUSCULE.

Figure 2.1 La grammaire du noyau fonctionnel de ML-ACT.

```
fichier      ::=  [declaration;;|expression;;]*
declaration  ::=  let [rec] definition [and definition]*
definition   ::=  IDENT [filtre]* = expression
filtre       ::=  IDENT |
                  constante |
                  _ |
                  (filtre) |
                  filtre [,filtre]+ |
                  [filtre [;filtre]*] |
                  filtre::filtre |
                  filtre | filtre
expression   ::=  (expression) |
                  begin expression end |
                  [expression]+ |
                  unaire expression |
                  expression binaire expression |
                  [expression [;expression]*] |
                  IDENT |
                  constante |
                  if expression then expression [else expression] |
                  match expression with filtrage [| filtrage]* |
                  function filtrage [| filtrage]* |
                  declaration in expression
filtrage     ::=  filtre -> expression
constante    ::=  ENTIER |
                  FLOTTANT |
                  CARACTERE |
                  BOOLEEN |
                  CHAINE |
                  [] |
                  ()
unaire       ::=  -. | - | fst | snd | hd | tl | not
binaire      ::=  +. | -. | *. | /. | + | - | * | / | mod | :: | @ | & | or | ^ |
                  < | > | <> | = | <= | >= | ;
```

La syntaxe est héritée de CaML, les expressions y ont la même signification. En fait, il s'agit d'une simplification de CaML.

2.4 La syntaxe abstraite.

La syntaxe abstraite retenue est présentée dans la figure 2.2. Elle a été conçue dans un souci d'efficacité, il y a donc certaines redondances. Par exemple, un filtrage («match») aurait pu être exprimé sous la forme d'une application («apply») d'une fonction («function») à une valeur.

La syntaxe utilisée est du pseudo-ML, les paramètres sont décrit via les types de ML, avec en particulier t option qui est le type (None ∪ Some t) et rec_flag qui est le type (Recursive ∪ Nonrecursive).

Ces constructeurs permettent de construire un arbre abstrait qui reflète le contenu du programme.

Figure 2.2 La syntaxe abstraite du noyau fonctionnel de ML-ACT.

prog	=	inst list
inst	= **Eval**	expression
	= **Value**	rec_flag * (string * expression) list
constant	= **Int**	int
	= **Char**	char
	= **String**	string
	= **Float**	string
	= **Nil**	
	= **Nop**	
	= **False**	
	= **True**	
pattern	= **Any**	
	= **Var**	string
	= **Constant**	constant
	= **Tuple**	pattern list
	= **Cons**	pattern * pattern
	= **Or**	pattern * pattern
expression	= **Ident**	string
	= **Constant**	constant
	= **Let**	rec_flag * (string * expression) list * expression
	= **Function**	(pattern * expression) list
	= **Apply**	expression * expression list
	= **Match**	expression * (pattern * expression) list
	= **Tuple**	expression list
	= **Cons**	expression * expression
	= **Ifthenelse**	expression * expression * expression option
	= **Sequence**	expression * expression

Pour obtenir l'arbre de syntaxe abstraite représentant un programme, il faut utiliser des règles de traduction. Cette tâche sera réalisée par l'analyseur lexical et l'analyseur syntaxique. Pour ne pas alourdir l'exposé, on ne donnera pas ces règles. Elles sont simples, directes et la syntaxe abstraite a été conçue de façon à être suffisamment explicite.

2.5 Le système de type.

2.5.1 Fonctionnement du typage.

Objective CaML reconstruit le type de chaque entité d'un programme automatiquement. Pour cela, il associe à celles-ci une variable de type, puis effectue une unification à la volée. Mais, dans le cadre de notre projet, nous souhaitions pouvoir changer le plus simplement possible le système de type. Nous avons choisi d'effectuer un typage en deux étapes :

- Une première consistant en la collecte de contraintes entre types,
- La seconde étant la résolution de ces contraintes.

Il suffira, si on souhaite changer le mécanisme de typage, de modifier la forme des contraintes ainsi que le module chargé de les résoudre. Cette politique confère une grande flexibilité au système de type de ML-ACT.

Le système de type choisi est décrit par les règles présentées ci-dessous.

2.5.2 Des fonctions utiles.

Tout d'abord présentons les fonctions utilisées dans les règles d'inférence décrivant le système de type :

* **newvar() :** fonction qui renvoie une variable de type "fraîche" (ie qui n'a jamais été utilisée précédemment).

* **solve($\mathcal{C},\tau$) :** fonction qui résout l'ensemble des contraintes $\mathcal{C}$ par unification, puis renvoie la valeur obtenue pour τ après généralisation des variables libres.

* **replace($\mathcal{E}$,id,τ) :** fonction qui, si id est dans $\mathcal{E}$ remplace sa liaison dans $\mathcal{E}$ par (id,τ), et qui sinon l'ajoute.

* **search(id,$\mathcal{E}$) :** fonction qui échoue si id n'est pas dans $\mathcal{E}$ et dans le cas contraire, renvoie le type qui lui est associé dans $\mathcal{E}$.

* **join($\mathcal{E}_1$,$\mathcal{E}_2$) :** fonction qui renvoie l'union de $\mathcal{E}_1$ et de $\mathcal{E}_2$ si leur intersection est vide et sinon lève une erreur.

* **joinor($\mathcal{E}_1$,$\mathcal{E}_2$) :** fonction qui renvoie un environnement vide si $\mathcal{E}_1$ et $\mathcal{E}_2$ sont tous deux vides et sinon lève une erreur.

* **predef($\mathcal{E}$,funlist) :** fonction qui parcourt itérativement la liste funlist pour ajouter à $\mathcal{E}$ les fonctions qu'elle contient. On pourrait en CaML l'écrire de la façon suivante :

```
let predef env_init fun_list =
  List.fold_left
    (fun env (id,exp) -> (id,newvar ())::env)
    env_init fun_list
```

∗ **typeof(cst) :** fonction qui type les constantes, et dont le code CaML serait :

```
let typeof = function
  Int _    -> int
| Char _   -> char
| String _ -> string
| Float _  -> float
| Nil      -> (newvar ()) list
| Nop      -> unit
| True     -> bool
| False    -> bool
```

2.5.3 Les règles de typage

Les symboles utilisés seront :

- $\mathcal{E}$ pour les environnements,
- $\mathcal{C}$ pour les ensembles de contraintes,
- τ pour les types,
- $\{\}$ pour les environnements vides ou les ensembles de contraintes vides.

Les différentes règles sont regroupées en 5 familles différentes :

- Programmes,
- Instructions,
- Définitions,
- Expressions,
- Filtres.

Les règles vont utiliser des jugements de la forme :

«les paramètres ⊢ l'objet traité : le résultat»

On peut donc voir un jugement comme un appel de fonction qui prend en paramètre les données à gauche du ⊢, et renvoie comme résultat les données à droite du :. Cette vision algorithmique des règles d'inférence permet d'utiliser ce type de présentation que je trouve plus clair qu'un algorithme et ses nombreuses conditionnelles imbriquées.

Un Programme :

Un programme est une liste non vide d'instructions, il y a deux cas :

$$\text{PROG_1}: \frac{\mathcal{E} \vdash \text{inst} : \mathcal{E}_1}{\mathcal{E} \vdash [\text{inst}] : \mathcal{E}_1} \qquad \text{PROG_2}: \frac{\mathcal{E} \vdash \text{inst} : \mathcal{E}_1 \quad \mathcal{E}_1 \vdash \text{instl} : \mathcal{E}_2}{\mathcal{E} \vdash \text{inst::instl} : \mathcal{E}_2}$$

La règle (*prog_1*) est celle de l'analyse d'un programme contenant une seule instruction, et la règle (*prog_2*), celle d'une liste de plus d'une instruction. Elles prennent en paramètre d'entrée un environnement et renvoient cet environnement modifié. Elles signifient que l'on construit l'environnement au fur et à mesure du typage des différentes instructions en utilisant les règles suivantes concernant les instructions.

Une instruction :

Les trois règles représentent les trois types d'instructions possibles : une expression, un bloc de définitions ou un bloc de définitions récursives. Ces règles construisent un environnement à partir d'un environnement en entrée.

$$\text{IEVAL} : \frac{\mathcal{E} \vdash \text{expr} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash \text{Eval expr} : \mathcal{E}} \qquad \text{IVALUE} : \frac{\mathcal{E} \vdash \text{funlist} : \mathcal{E}_1}{\mathcal{E} \vdash \text{Value(Nonrecursive,funlist)} : \mathcal{E}_1}$$

$$\text{IVALUE_REC} : \frac{\text{predef}(\mathcal{E}\text{,funlist}) \vdash \text{funlist} : \mathcal{E}_1}{\mathcal{E} \vdash \text{Value(Recursive,funlist)} : \mathcal{E}_1}$$

La règle (*ieval*) utilise les règles concernant les expressions et indique que pour une expression, on ne fait que vérifier qu'elle est bien typée. Pour les blocs de définitions, on construit les environnements de ces nouvelles définitions via les règles concernant les définitions. La seule différence entre les règles (*ivalue_rec*) et (*ivalue*) est l'utilisation de la fonction **predef** présenté dans la section 2.5.2, qui permet de rajouter à l'environnement tous les nouveaux identificateurs du bloc afin de permettre la récursivité.

Les définitions :

Une définition est en fait une liste de couple (id,exp) où id est un identificateur et exp l'expression qui lui est associée. Il y a donc deux cas, selon que cette liste contienne un élément ou plus d'un. Chaque règle prend en entrée un environnement et retourne un environnement.

$$\text{DEF_1} : \frac{\mathcal{E} \vdash \text{exp} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash [(\text{id,exp})] : \text{replace}(\mathcal{E}, \text{id}, \text{solve}(\mathcal{C} \cup \{\tau = \text{search}(\text{id},\mathcal{E})\}, \tau))}$$

$$\text{DEF_2} : \frac{\mathcal{E} \vdash \text{exp} : \tau,\, \mathcal{C} \quad \text{replace}(\mathcal{E}, \text{id,solve}(\mathcal{C} \cup \{\tau = \text{search}(\text{id},\mathcal{E})\}, \tau)) \vdash \text{ld} : \mathcal{E}_1}{\mathcal{E} \vdash (\text{id,exp})\text{::ld} : \mathcal{E}_1}$$

Les règles (*def_1*) et (*def_2*) utilisent les règles des expressions qui suivent et signifient que pour chaque couple, on type l'expression. Puis, on résout les contraintes et on généralise pour obtenir le type qu'aura l'identificateur. Cette tâche est réalisée par la fonction **solve**. Pour simplifier l'écriture des règles, on suppose que si **search** échoue, alors la contrainte $\{\tau = \text{search}(\text{id},\mathcal{E})\}$ n'est pas ajoutée à $\mathcal{C}$.

Les expressions :

Il y a onze types d'expression qui donnent quinze règles de typage, celle-ci prennent toutes en entrée un environnement et retournent le type de l'expression ainsi qu'un ensemble de contraintes.

$$\text{EID} : \frac{}{\mathcal{E} \vdash \text{Ident id} : \text{search}(\text{id},\mathcal{E}),\ \{\}} \qquad \text{ECST} : \frac{}{\mathcal{E} \vdash \text{Constant cst} : \text{typeof(cst)},\ \{\}}$$

(*eid*) et (*ecst*) sont les seuls axiomes du système d'inférence concernant les expressions. Leur signification est claire : pour un identificateur, on récupère son type dans l'environnement, et pour une constante, on utilise la fonction **typeof**.

$$\text{ELETREC}: \frac{\text{predef}(\mathcal{E},\text{funlist}) \vdash \text{funlist} : \mathcal{E}_1 \quad \mathcal{E}_1 \vdash \text{expr} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash \text{Let(Recursive,funlist,expr)} : \tau,\, \mathcal{C}}$$

$$\text{ELET}: \frac{\mathcal{E} \vdash \text{funlist} : \mathcal{E}_1 \quad \mathcal{E}_1 \vdash \text{expr} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash \text{Let(Nonrecursive,funlist,expr)} : \tau,\, \mathcal{C}}$$

Le typage des «let» est similaire a celui des blocs de définitions, et donc la forme de ces règles s'apparente à celle des règles vues précédemment. La différence est que l'environnement construit n'est pas ajouté à l'environnement global. En effet, sa portée se limite à l'expression du «let».

$$\text{EFUNC_1}: \frac{\tau_3\text{=newvar()}\ \tau_4\text{=newvar()} \quad \vdash \text{pat} : \tau_1,\, \mathcal{E}_1,\, \mathcal{C}_1 \quad \mathcal{E} \cup \mathcal{E}_1 \vdash \text{expr} : \tau_2,\, \mathcal{C}_2}{\mathcal{E} \vdash \text{Function[(pat,expr)]} : \tau_3 \to \tau_4,\, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \tau_3\} \cup \{\tau_2 = \tau_4\}}$$

$$\text{EFUNC_2}: \frac{\mathcal{E} \vdash \text{Function l} : \tau_3 \to \tau_4,\, \mathcal{C}_3 \quad \vdash \text{pat} : \tau_1,\, \mathcal{E}_1,\, \mathcal{C}_1 \quad \mathcal{E} \cup \mathcal{E}_1 \vdash \text{expr} : \tau_2,\, \mathcal{C}_2}{\mathcal{E} \vdash \text{Function(pat,expr)::l} : \tau_3 \to \tau_4,\, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 = \tau_3\} \cup \{\tau_2 = \tau_4\}}$$

Le typage d'une fonction utilise les règles sur les filtres qui sont présentées un peu plus loin. Les règles (*efunc_1*) et (*efunc_2*) signifient que tous les filtres d'une part, et toutes les expressions d'autre part, doivent avoir le même type respectivement τ_{dep} et τ_{arr}. Alors, le type du filtrage sera un type fonction: $\tau_{dep} \to \tau_{arr}$.

$$\text{EAPP_1}: \frac{\tau_3\text{=newvar()}\ \tau_4\text{=newvar()} \quad \mathcal{E} \vdash \text{exprf} : \tau_1,\, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{expra} : \tau_2,\, \mathcal{C}_2}{\mathcal{E} \vdash \text{Apply(exprf,[expra])} : \tau_4,\, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \tau_3 \to \tau_4\} \cup \{\tau_2 = \tau_3\}}$$

$$\text{EAPP_2}: \frac{\mathcal{E} \vdash \text{Apply(Apply(exprf,[expra]),l)} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash \text{Apply(exprf,expra::l)} : \tau,\, \mathcal{C}}$$

Le typage de l'application signifie que l'on type d'abord la fonction puis ses arguments. Supposons que le type obtenu pour la fonction est $\tau_{dep} \to \tau_{arr}$. Le type de l'application est alors le type de retour du type de la fonction (ie le type à droite de la flèche :τ_{arr}), et ceci si le type des arguments est compatible avec celui du type de départ du type de la fonction (ie le type à gauche de la flèche τ_{dep}).

$$\text{EMATCH}: \frac{\tau\text{=newvar()} \quad \mathcal{E} \vdash \text{Function(fun)} : \tau_1,\, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{expr} : \tau_2,\, \mathcal{C}_2}{\mathcal{E} \vdash \text{Match(expr,fun)} : \tau,\, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \tau_2 \to \tau\}}$$

En fait, le filtrage a été gardé dans la syntaxe abstraite pour optimiser son traitement. Comme il ne s'agit que de l'application d'un filtrage, son analyse consiste donc en une combinaison des quatre règles précédentes.

$$\text{ETUPLE_1}: \frac{\mathcal{E} \vdash \text{expr} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash \text{Tuple [expr]} : \tau,\, \mathcal{C}} \qquad \text{ETUPLE_2}: \frac{\mathcal{E} \vdash \text{expr} : \tau_1,\, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{Tuple l} : \tau_2,\, \mathcal{C}_2}{\mathcal{E} \vdash \text{Tuple expr::l} : \tau_1 * \tau_2,\, \mathcal{C}_1 \cup \mathcal{C}_2}$$

Le typage d'un tuple est simple. Il s'agit en effet, uniquement du typage de toutes ces composantes l'une après l'autre.

$$\text{ECONS}: \frac{\mathcal{E} \vdash \text{exprh} : \tau_1, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{exprt} : \tau_2, \mathcal{C}_2}{\mathcal{E} \vdash \text{Cons(exprh,exprt)} : \tau_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \textit{list} = \tau_2\}}$$

Pour typer la construction de liste, il faut tout d'abord typer la tête puis la queue. De plus, les types obtenus doivent être compatibles, car on ne tolère que les listes d'éléments d'un type bien précis.

$$\text{EIF}: \frac{\mathcal{E} \vdash \text{e1} : \tau_1, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{e2} : \tau_2, \mathcal{C}_2}{\mathcal{E} \vdash \text{Ifthenelse(e1,e2,None)} : \mathcal{U}nit, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \mathcal{B}ool\} \cup \{\tau_2 = \mathcal{U}nit\}}$$

$$\text{EIFELSE}: \frac{\mathcal{E} \vdash \text{e1} : \tau_1, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{e2} : \tau_2, \mathcal{C}_2 \quad \mathcal{E} \vdash \text{e3} : \tau_3, \mathcal{C}_3}{\mathcal{E} \vdash \text{Ifthenelse(e1,e2,Some e3)} : \tau_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 = \mathcal{B}ool\} \cup \{\tau_3 = \tau_2\}}$$

De façon similaire au filtrage, la conditionnelle est un ajout non indispensable à la syntaxe abstraite. Ses règles sont donc à rapprocher de celle de (*ematch*). Il convient de remarquer tout de même que si le «else» est absent, on impose au «then» de ne rien renvoyer.

$$\text{ESEQ}: \frac{\mathcal{E} \vdash \text{expr1} : \tau_1, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{expr2} : \tau_2, \mathcal{C}_2}{\mathcal{E} \vdash \text{Sequence(expr1,expr2)} : \tau_2, \mathcal{C}_1 \cup \mathcal{C}_2}$$

Enfin, le type d'une séquence est celui de la dernière expression et nécessite le typage des expressions qui la compose.

Les filtres :

Les sept règles de typage des filtres ne prennent rien en entrée, mais produisent en sortie : le type du filtre, un environnement contenant les variables du filtre et un ensemble de contraintes.

$$\text{PANY}: \frac{\tau\text{=newvar()}}{\vdash \text{Any} : \tau, \{\}, \{\}} \qquad \text{PVAR}: \frac{\tau\text{=newvar()}}{\vdash \text{Var id} : \tau, \{\text{id} : \tau\}, \{\}}$$

$$\text{PCST}: \frac{}{\vdash \text{Constant cst} : \text{typeof(cst)}, \{\}, \{\}}$$

Malgré leur apparence, les règles (*pany*) et (*pvar*) sont des axiomes comme (*pcst*). Elles signifient que le type de «_» ainsi que celui d'une variable sont quelconques. De plus, on ajoute la liaison (variable,type) à l'environnement. Enfin, la troisième règle est identique à celle concernant les expressions constantes.

$$\text{POR}: \frac{\vdash \text{pat1} : \tau_1, \mathcal{E}_1, \mathcal{C}_1 \quad \vdash \text{pat2} : \tau_2, \mathcal{E}_2, \mathcal{C}_2}{\vdash \text{Or(pat1,pat2)} : \tau, \text{joinor}(\mathcal{E}_1,\mathcal{E}_2), \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \tau_2\}}$$

La règle (*por*) est imposée par CaML qui ne tolère aucune liaison de variable dans un filtre «or». On interdit donc les variables dans ce type de filtre. Mais une extension du

système de type pourrait consister à remplacer cette contrainte par le fait que les deux branches de l'alternative lient les mêmes variables.

$$\text{PTUPLE_1} : \frac{\vdash \text{pat} : \tau,\ \mathcal{E},\ \mathcal{C}}{\vdash \text{Tuple [pat]} : \tau,\ \mathcal{E},\ \mathcal{C}}$$

$$\text{PTUPLE_2} : \frac{\vdash \text{pat} : \tau_1,\ \mathcal{E}_1,\ \mathcal{C}_1 \quad \vdash \text{Tuple l} : \tau_2,\ \mathcal{E}_2,\ \mathcal{C}_2}{\vdash \text{Tuple pat::l} : \tau_1 * \tau_2,\ \text{join}(\mathcal{E}_1,\mathcal{E}_2),\ \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\text{PCONS} : \frac{\vdash \text{path} : \tau_1,\ \mathcal{E}_1,\ \mathcal{C}_1 \quad \vdash \text{patt} : \tau_2,\ \mathcal{E}_2,\ \mathcal{C}_2}{\vdash \text{Cons(path,patt)} : \tau_2,\ \text{join}(\mathcal{E}_1,\mathcal{E}_2),\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \mathit{list} = \tau_2\}}$$

Enfin, les trois règles ci-dessus sont identiques à leur équivalent «expression» avec la particularité qu'une variable ne peut apparaître qu'une seule fois dans un filtre.

REMARQUE 2 *La politique de généralisation des variable de type libre peut être systématique car pour le moment* ML-ACT *est un langage fonctionnel pur. Donc, on généralisera toujours les variables de type libres en sortie d'un «let».*

2.6 L'unification.

La collecte des contraintes de type mène à un ensemble d'équations sur ces types. La recherche d'un type correspond alors à la résolution de cet ensemble. La recherche d'une solution se ramène en fait à la recherche d'un système «résolu», équivalent au premier. L'algorithme d'unification consiste donc en une suite de transformation, de système de contraintes en système de contraintes équivalents.

L'algorithme que nous avons conçu et implanté, est inspiré de celui de Robinson présenté dans le livre de R.Lalement [Lal90].

Dans notre contexte, on dispose de variable de type ainsi que des types constants et des constructeurs de types suivant :

- int, char, float, bool, string et unit d'arité nulle,
- list(t) d'arité un,
- arrow(t_1,t_2) d'arité deux,
- tuple(lt) où lt est une liste de types, d'arité variable.

On utilisera, de plus, une structure de donnée qui contiendra les contraintes. Dans notre programme, nous avons utilisé une file mais on aurait pu aussi bien utiliser une liste, une pile ... On la notera **filec**. La boucle principale de l'algorithme sera :

- remplir filec avec les contraintes
- tant que filec $\neq \emptyset$ faire :
  - récupérer la tête de filec
  - unification de la contrainte ainsi obtenue
- fin tant que

La fonction d'unification proprement dite, utilise un environnement **env** qui contiendra la valeur des variables de type. Celui-ci est construit au fur et à mesure de l'unification

des contraintes. Pour représenter cet environnement, on a utilisé une table hash-codée de couple (numéro de variable,référence à un type), mais les seules primitives nécessaires sont :

- **find** (int → type) qui renvoie la valeur actuelle de la variable de type de numéro passé en paramètre.
- **add** (int → type → unit) qui ajoute le couple (num,t) dans l'environnement de typage.
- **replace** (int → type → unit) qui remplace la liaison actuelle de num par une liaison (num,t).

L'algorithme est :

- selon que les deux types soient :
- deux variables de même numéro : rien
- deux variables de numéro différents, v_1 et v_2 :
*- si v_1 et v_2 sont dans **env** :*
- unification de leur deux valeurs.
*- si v_1 ou v_2 est dans **env** :*
*- on ajoute (add) la variable absente de **env** avec pour valeur la valeur de celle qui y est.*
*- si ni v_1 ni v_2 ne sont dans **env** :*
- on les ajoute (add) toutes les deux avec pour valeur la première variable.
- une variable v et un type quelconque t :
*- si v est dans **env** :*
- si elle est elle-même sa propre valeur :
- on remplace (replace) cette valeur par t.
- sinon :
- unification de cette valeur et de t.
- sinon :
- on ajoute (add) la liaison (v,t).
- deux constantes de même nom : rien
- deux listes, list(t_1) et list(t_2) :
- unification de t_1 et t_2.
- deux fonctions, $t_1 \to t_2$ et $t'_1 \to t'_2$:
- unification de t_1 et t'_1.
- unification de t_2 et t'_2.
- deux tuples, $t_1 * ... * t_n$ et $t'_1 * ... * t'_n$ de même taille :
- unification de t_1 et t'_1.
$\vdots$
- unification de t_n et t'_n.
- sinon : erreur, l'unification entre les deux types est impossible.

Une fois l'environnement obtenu, on va utiliser une fonction qui va chercher les valeurs effectives des différentes variables de type. Pour cela, elle va effectuer un calcul récursif en profondeur, au cours duquel elle va construire une liste des variables déjà rencontrées pour détecter les circularités.

Présentons le fonctionnement de cet algorithme sur un exemple simple. Soit l'ensemble de contraintes suivant :

- $v_0 = v_1$
- $v_1 = v_2 * int$
- $v_3 = v_2\ list$
- $v_4 = v_2\ list$
- $v_5 = v_3 * v_4$
- $v_3 = float\ list$

On va décrire l'évolution de l'environnement au cours de l'application de l'algorithme présenté ci-dessus. Au cours de chaque étape, la contrainte traitée est soulignée. De plus, l'environnement résultat est fournit dans la troisième colonne.

	ensemble de contrainte	environnement
1	$\underline{v_0 = v_1}; v_1 = v_2 * int; v_3 = v_2\ list;$ $v_4 = v_2\ list; v_5 = v_3 * v_4; v_3 = float\ list$	$(0, v_0); (1, v_0)$
2	$\underline{v_1 = v_2 * int}; v_3 = v_2\ list; v_4 = v_2\ list;$ $v_5 = v_3 * v_4; v_3 = float\ list$	$(0, v_0); (1, v_0)$
3	$\underline{v_0 = v_2 * int}; v_3 = v_2\ list; v_4 = v_2\ list;$ $v_5 = v_3 * v_4; v_3 = float\ list$	$(0, v_2 * int); (1, v_0)$
4	$\underline{v_3 = v_2\ list}; v_4 = v_2\ list; v_5 = v_3 * v_4;$ $v_3 = float\ list$	$(0, v_2 * int); (1, v_0); (3, v_2\ list)$
5	$\underline{v_4 = v_2\ list}; v_5 = v_3 * v_4; v_3 = float\ list$	$(0, v_2 * int); (1, v_0); (3, v_2\ list);$ $(4, v_2\ list)$
6	$\underline{v_5 = v_3 * v_4}; v_3 = float\ list$	$(0, v_2 * int); (1, v_0); (3, v_2\ list);$ $(4, v_2\ list); (5, v_3 * v_4)$
7	$\underline{v_3 = float\ list}$	$(0, v_2 * int); (1, v_0); (3, v_2\ list);$ $(4, v_2\ list); (5, v_3 * v_4)$
8	$\underline{v_2\ list = float\ list}$	$(0, v_2 * int); (1, v_0); (3, v_2\ list);$ $(4, v_2\ list); (5, v_3 * v_4)$
9	$\underline{v_2 = float}$	$(0, v_2 * int); (1, v_0); (2, float); (3, v_2\ list);$ $(4, v_2\ list); (5, v_3 * v_4)$

L'environnement résultat de la première phase est donc :

$$\{(0, v_2 * int); (1, v_0); (2, float); (3, v_2\ list); (4, v_2\ list); (5, v_3 * v_4)\}$$

Maintenant décrivons sur cet exemple la deuxième partie de l'algorithme qui n'a pas été décrite en détail. Les différents accès à l'environnement sont décrits ci-dessous. On ne donne que le résultat calculé de proche en proche avec l'aide de cet environnement que l'on vient de construire.

1. $v_0 \Rightarrow v_2 * int \Rightarrow float * int$
2. $v_1 \Rightarrow v_0 \Rightarrow float * int$
3. $v_2 \Rightarrow float$

4. $v_3 \Rightarrow v_2\ list \Rightarrow float\ list$

5. $v_4 \Rightarrow v_2\ list \Rightarrow float\ list$

6. $v_5 \Rightarrow v_3 * v_4 \Rightarrow float\ list * v_4 \Rightarrow float\ list * float\ list$

Cet algorithme n'est pas exactement celui implanté car il y manque le typage de la partie acteur. Il sera complété dans le chapitre suivant.

2.7 Les réalisations.

J'ai réalisé une famille de modules qui implantent l'analyseur lexical et l'analyseur syntaxique ainsi que le typage de ML-ACT. Dans un premier temps et pour me familiariser avec CaML, les analyseurs avaient été effectivement programmés. Mais, dans la version finale ils sont composés d'un fichier Ocamllex et d'un fichier Ocamlyacc, ces outils étant les pendants de lex et yacc (du monde C/UNIX) pour le monde CaML.

La liste des modules composant le compilateur est :

asttype.mli : une partie de la syntaxe abstraite (constantes),

parsedtree.mli : la syntaxe abstraite avec la localisation dans le programme source,

typedtree.mli : la syntaxe abstraite avec la localisation dans le programme source et en plus les types,

lexer.mll : l'analyseur lexical,

parser.mly : l'analyseur syntaxique,

types.ml : la définition des types ainsi que leur manipulation,

type_checker.ml : le typage d'un programme,

unify.ml : les fonctions chargées de la résolution des contraintes par unification et de la généralisation des variables de type,

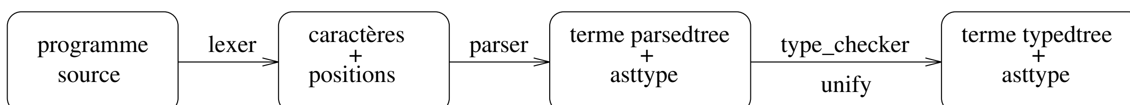
env.ml : fournit l'objet environnement ainsi que les fonctions pour le manipuler,

location.ml : gestion de la localisation des expressions dans le texte source,

compiler.ml : le programme principal qui utilise tous les modules précédents pour compiler un programme source.

On peut résumer la façon dont sont utilisés ces modules par le schéma 2.3.

Figure 2.3 Compilation schématique.



Le résultat de ces différentes étapes est un arbre syntaxique décoré. Celui-ci contient pour chaque nœud sa position dans le programme source (pour signaler la position d'éventuelles erreurs) et son type. Les informations de typage sont conservées, car elles seront utilisées lors de l'extraction du terme CAP et lors de la génération de CaML.

2.8 Extensions envisageables.

Du point de vue de la syntaxe, on pourrait étendre ML-ACT en complétant sa syntaxe pour rejoindre celle de CaML, comme par exemple le rajout de garde dans le filtrage. Puis, il serait intéressant d'intégrer les notions d'exceptions (ce qui serait relativement simple) et celles de types construits et de modules (qui imposent une refonte du système de type et de l'unification).

D'autre part, l'objectif étant de disposer d'outils d'analyse les plus fins possibles, il serait souhaitable de développer un système de type plus puissant. En effet, l'approximation effectuée par le système de type actuel est très grossière. Elle ne permet donc pas de fournir des types très précis.

C'est dans ce but que le typage actuel fonctionne, par collecte de contraintes puis résolution de ces contraintes. On pourrait, par exemple, utiliser un système avec une notion de sous-typage, d'union de type, d'intersection de type... Le travail consisterait alors en la connexion du compilateur avec un programme de résolution de contraintes ensemblistes à la Aiken et Wimmers [AW92] et [AW93], qui a été réalisé par Xavier Thirioux durant son stage de troisième année en 1994 [Thi94], dans le cadre du langage FOL développé par M.Pantel [Pan94].

Chapitre 3

LES ACTEURS

Dans ce chapitre, après une présentation du modèle d'acteur, nous étendrons ML-ACT et son système de type.

3.1 La programmation par acteur.

Le modèle d'acteur fournit des primitives et des constructions de haut niveau d'abstraction pour la programmation d'applications concurrentes.

Tout d'abord l'objet central du modèle est l'acteur. C'est une entité réactive qui peut être décrite par un couple composé :

- d'une *adresse postale*,
- d'un *comportement*.

L'adresse représente l'identité de l'acteur. Elle est unique et restera la même tout au long de la vie de l'acteur. Le comportement indique la réaction de l'acteur à l'arrivée d'un message. Il n'est pas figé et peut changer au cours de l'existence de l'acteur. C'est un couple composé :

- d'une *interface*,
- de *champs privés*.

L'interface est la partie réaction proprement dite. Il s'agit de la liste des messages acceptés par l'acteur ainsi que du programme ou script associé à l'événement de réception de ceux-ci. Ce script est composé, outre de calculs classiques, de :

- l'*envoi* d'un nombre fini de messages,
- la *création* d'un nombre fini d'acteurs,
- le *changement* de son comportement.

Les champs privés forment un enregistrement privé qui contient la mémoire locale de l'acteur.

La seconde abstraction du modèle d'acteur est le message, c'est un couple constitué par :

- une *étiquette*,
- des *paramètres*.

L'étiquette est la clé qui permet à l'acteur de choisir une réaction parmi celles de son comportement. Les paramètres sont les données qui sont transportées par le message. Ces données peuvent être de type classique comme des entiers, des chaînes de caractères ou des listes, mais elles peuvent également être des adresses ou des comportements d'acteurs.

La communication entre les acteurs est de type point à point, c'est-à-dire qu'il faut connaître l'adresse d'un acteur pour lui envoyer un message. Ces acteurs, dont on connaît l'adresse, forment l'ensemble des *connaissances.* On remarque que cet ensemble est dynamique car, au cours de son exécution, un acteur peut recevoir l'adresse d'un autre acteur qu'il ne connaissait pas encore, ou oublier l'adresse d'un acteur qu'il connaissait. La topologie de communication est donc dynamique. De plus, l'envoi de message est une opération non-bloquante car asynchrone. Ainsi, on évite le difficile écueil de la programmation concurrente qu'est l'interblocage. En effet, les acteurs peuvent devenir passifs, mais sont toujours accessibles aux autres acteurs. Pour supporter un tel protocole de communication les messages sont généralement mémorisés dans des files ;une file étant associée à chaque adresse d'acteur. La seule hypothèse faite sur le médium de communication est qu'il est sûr, c'est-à-dire, qu'un message envoyé sera toujours reçu par son destinataire en un temps fini. En revanche, aucune hypothèse n'est faite sur l'ordre de réception des messages. Remarquons que ce protocole de communication impose de nouvelles méthodes de programmation car on ne peut pas recevoir directement une réponse à un message envoyé. Si on a besoin lors d'un calcul du résultat d'un calcul effectué par un autre acteur, on doit passer par un état intermédiaire où on attend uniquement le résultat de ce calcul. Notons que le modèle de communication choisi permet d'éviter les problèmes d'accès concurrent à des données. En effet, la mise en file des messages provoque une sérialisation automatique des accès.

Le modèle ainsi décrit permet la programmation d'applications concurrentes avec une certaine transparence au niveau des communications. De plus, par nature, les acteurs vérifient les «bonnes» propriétés d'extensibilité et de réutilisabilité.

L'équipe Plasma de l'IRIT (Arcangeli, Marcoux, Maurel et Sallé) a déjà conçu un langage d'acteurs : *Plasma II* ([MMS88]) dont la syntaxe est proche de LISP. Dans le but d'appliquer des mécanismes de typage, nous avons choisi de proposer un langage dérivé de ML (qui est l'objet de nombreuses analyses statiques), plutôt que de LISP qui n'est pas typé.

3.2 Un langage d'acteur.

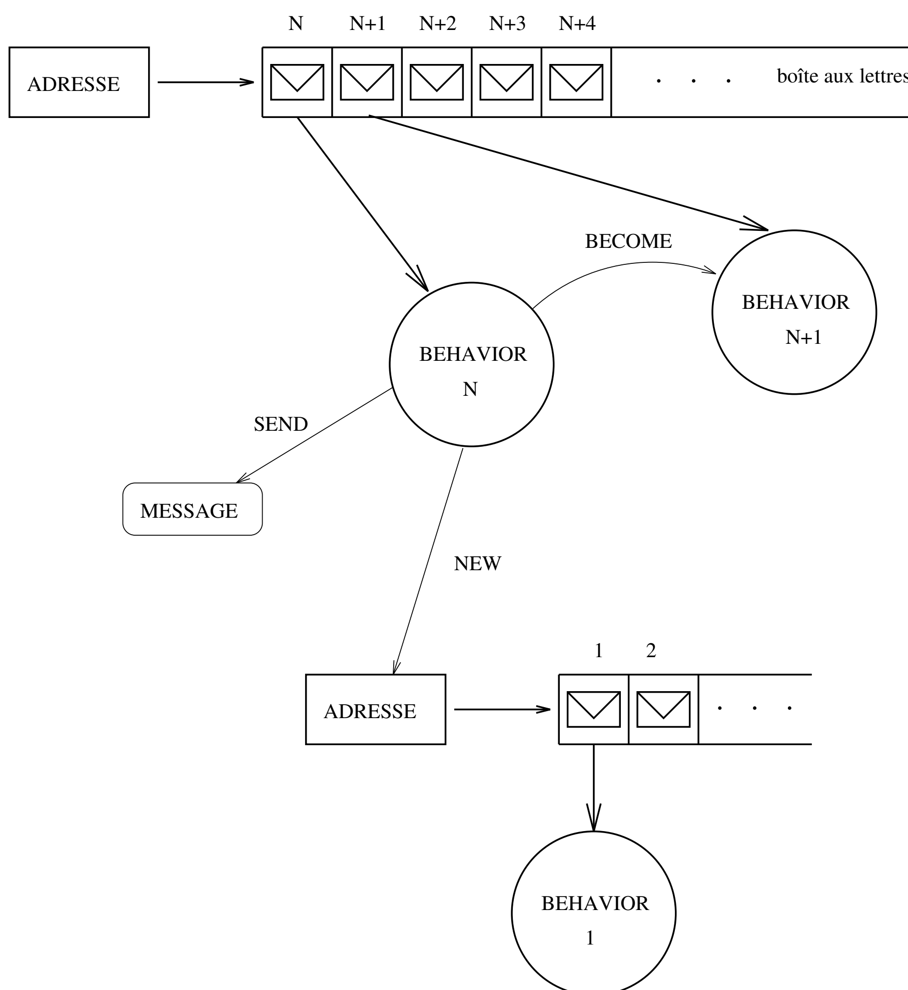
Pour pouvoir utiliser le modèle d'acteur les primitives que nous avons ajoutées à ML-Act, sont :

- l'envoi de message : `send to`,
- la création d'un acteur : `new`,
- le changement de comportement : `become`,
- un changement de comportement particulier, la mort d'un acteur : `suicide`,
- l'accès d'un acteur à sa propre adresse : `ego`,
- la possibilité de construire des comportements : `behavior message val mutable <-`,

– enfin, la construction de messages qui sera similaire à l'appel de fonction classique et à la construction de termes.

La figure 3.1 décrit succinctement le comportement d'un acteur qui traiterait son n$^{\text{ième}}$ message. Elle montre les trois opérations qui spécifient le comportement de tout acteur : send, new et become.

Figure 3.1 Un acteur traitant son n$^{\text{ième}}$ message.



La figure 3.2 décrit les ajouts à la grammaire de ML-Act. Les règles présentées remplacent celles de même nom de la figure 2.1, à l'exception de la nouvelle règle pour «*expression*» qui ne fait que compléter la première.

Le complément de la syntaxe abstraite nécessaire à la description complète de ML-Act est présenté dans la figure 3.3. Les remarques faites pour la grammaire s'appliquent aussi à cette description.

Remarque 3 *Dans la notion d'acteur que va implanter* ML-Act, *le* `self` *n'est pas nécessaire. En effet, on considère que par défaut un acteur boucle sur son comportement et donc, fait automatiquement un* `become self` *si il ne se suicide pas ou ne change pas d'état. Ce choix a été fait car généralement un acteur ne change pas son interface, mais modifie uniquement sa mémoire locale. Les opérations de suicide et de changement de*

Figure 3.2 La grammaire de ML-ACT.

fich	::=	[*declaration*;; | *expression*;; | *comportements*;;]*
comportements	::=	**behavior** *comportement* [**and** *comportement*]* **end**
comportement	::=	IDENT [*filtre*]* = [*champ* | *message*]*
definition	::=	IDENT [*filtre*]* = *corps*
corps	::=	*expression* |
		new *expression*
champ	::=	**val** [**mutable**] IDENT = *expression*
message	::=	**message** IDENT [*filtre*]* = *expression*
expression	::=	**send** *expression* **to** *expression* |
		become *expression* |
		ego |
		suicide |
		IDENT <- *expression*

Figure 3.3 La syntaxe abstraite de ML-ACT.

inst	= **Beh**	(string * pattern list * value list * mess list) list
expression	= **Send**	expression * expression
	= **Set**	string * expression
	= **Become**	expression
	= **Ego**	
	= **Suicide**	
	= **New**	expression
value	=	bool * string * expression
mess	=	string * pattern list * expression

comportement sont moins courantes et donc doivent être explicitement spécifiées par le programmeur.

3.3 Insertion dans le système de type.

Les conventions d'écriture sont les mêmes que celles de 2.5.3. On va compléter les règles concernant les expressions et ajouter les règles de définition des comportements, des champs et des messages. De plus, aux types classiques, nous ajoutons les types constants : $\mathcal{M}ess$, $\mathcal{A}ddr$ et $\mathcal{B}eh$. Cette approximation permet d'éliminer les erreurs grossières. Des vérifications plus précises sur les comportements et messages seront effectuées sur le terme CAP extrait (voir chapitre suivant).

Afin de simplifier la présentation des règles de typage, on a supprimé tout ce qui concerne la vérification de l'endroit où sont placées les expressions, les «become» ou les «suicide» devant être dans le corps d'un comportement. Dans la pratique, il suffit de véhiculer un booléen dans les règles indiquant si on est dans le corps de définition d'un comportement.

Une instruction :

La règle suivante complète les règles de typage des instructions du chapitre précédent.

$$\text{IBEH}: \frac{\text{predef2}(\mathcal{E},\text{behlist}) \vdash \text{behlist} : \mathcal{E}_1}{\mathcal{E} \vdash \text{Beh behlist} : \mathcal{E}_1}$$

La seule particularité de cette règle est l'utilisation de la fonction **predef2** qui est similaire à la fonction **predef** du chapitre précédent mais qui exploite des comportements au lieu de liaisons. Elle fournit donc un environnement dans lequel ont été prédéfini tous les identificateurs noms de comportement avec le type $\tau \rightarrow \mathcal{B}eh$, ainsi que la définition des identificateurs de messages avec le type $\mathcal{M}ess$. Elle autorise ainsi une récursivité mutuelle de ceux-ci.

Une expression :

$$\text{ESEND}: \frac{\mathcal{E} \vdash \text{expm} : \tau_1, \mathcal{C}_1 \quad \mathcal{E} \vdash \text{expd} : \tau_2, \mathcal{C}_2}{\mathcal{E} \vdash \text{Send(expm,expd)} : \mathcal{U}nit,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \mathcal{M}ess\} \cup \{\tau_2 = \mathcal{A}ddr\}}$$

La seule particularité de la règle (*esend*) est qu'elle impose au message d'être de type $\mathcal{M}ess$ et que le destinataire soit de type $\mathcal{A}ddr$.

$$\text{EEGO}: \frac{}{\mathcal{E} \vdash \text{Ego} : \mathcal{A}ddr,\ \{\}} \qquad \text{ESUICIDE}: \frac{}{\mathcal{E} \vdash \text{Suicide} : \mathcal{U}nit,\ \{\}}$$

On ajoute deux axiomes pour typer Ego et Suicide.

$$\text{ESET}: \frac{\mathcal{E} \vdash \text{exp} : \tau, \mathcal{C}}{\mathcal{E} \vdash \text{Set(id,exp)} : \mathcal{U}nit,\ \mathcal{C} \cup \{\tau = \text{search}(\text{id},\mathcal{E})\}}$$

Afin de simplifier, on ne vérifie pas dans (*eset*) que le champ que l'on affecte est modi-

fiable. En fait, pour pouvoir le faire il suffirait d'ajouter aux couples (identificateur,type) de l'environnement une troisième information qui indiquerait s'il le champ est modifiable ou non.

$$\text{EBECOME} : \frac{\mathcal{E} \vdash \text{exp} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash \text{Become exp} : \mathcal{U}nit,\, \mathcal{C} \cup \{\tau = \mathcal{B}eh\}}$$

$$\text{ENEW} : \frac{\mathcal{E} \vdash \text{exp} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash \text{New exp} : \mathcal{A}ddr,\, \mathcal{C} \cup \{\tau = \mathcal{B}eh\}}$$

Enfin, les deux règles précédentes typent l'expression, imposent que son type soit $\mathcal{B}eh$ et renvoient respectivement un type $\mathcal{U}nit$ et un type $\mathcal{A}ddr$.

Les définitions de comportement :

Pour alléger la présentation des règles, nous avons scindé ce typage en deux étapes : le parcours de la liste (règles (*behlist_1*) et (*behlist_2*)) et le typage d'un comportement proprement dit (*beh*).

$$\text{BEHLIST_1} : \frac{\mathcal{E} \vdash \text{b} : \mathcal{E}_1}{\mathcal{E} \vdash [\text{b}] : \mathcal{E}_1} \qquad \text{BEHLIST_2} : \frac{\mathcal{E} \vdash \text{b} : \mathcal{E}_1 \quad \mathcal{E}_1 \vdash \text{bl} : \mathcal{E}_2}{\vdash \text{b::bl} : \mathcal{E}_2}$$

$$\text{BEH} : \frac{\vdash \text{pl} : \tau l,\, \mathcal{E}_1,\, \mathcal{C} \quad \mathcal{E} \cup \mathcal{E}_1 \vdash \text{vl} : \mathcal{E}_2 \quad \mathcal{E} \cup \mathcal{E}_1 \cup \mathcal{E}_2 \vdash \text{ml} :}{\mathcal{E} \vdash (\text{id,pl,vl,ml}) : \text{replace}(\mathcal{E},\, \text{id},\, \text{behtype}(\mathcal{C},\, \tau l))}$$

Les variables pl, vl et ml représentent respectivement une liste de filtres, une liste de champs et une autre de messages. Les règles les traitant sont présentées ci-après. La règle de typage d'un comportement utilise la fonction **replace** du chapitre précédent ainsi que la fonction **behtype**. Celle-ci, ayant pour paramètres $\mathcal{C}$ et $\tau l (=[\tau_1;...;\tau_n])$, va dans un premier temps résoudre les contraintes $\mathcal{C}$. De cette résolution, on obtient les types réels des paramètres du comportement et on construit un type de la forme : $\tau_1 \rightarrow ... \rightarrow \tau_n \rightarrow \tau$ que l'on va transmettre.

Les champs :

Les règles de typage des champs sont issues directement des règles de définition du chapitre précédent. Elles ajoutent à l'environnement passé en paramètre la définition des champs.

$$\text{VAL_1} : \frac{\mathcal{E} \vdash \text{exp} : \tau,\, \mathcal{C}}{\mathcal{E} \vdash [(\text{mut,id,exp})] : \{\text{id:solve}(\mathcal{C},\, \tau)\}}$$

$$\text{VAL_2} : \frac{\mathcal{E} \vdash \text{exp} : \tau,\, \mathcal{C} \quad \mathcal{E} \vdash \text{vl} : \mathcal{E}_1}{\mathcal{E} \vdash (\text{mut,id,exp})\text{::vl} : \mathcal{E}_1 \cup \{\text{id:solve}(\mathcal{C},\, \tau)\}}$$

Dans le cas où l'on voudrait vérifier l'application de la mise à jour, ces règles seraient, de plus, chargées de sauvegarder dans l'environnement le caractère modifiable ou figé de la variable.

Les messages :

Les règles de typage des messages sont similaires aux règles concernant les champs, la seule différence est l'utilisation des paramètres.

$$\text{MESS_1} : \frac{\vdash \text{pl} : \tau l,\, \mathcal{E}_1,\, \mathcal{C} \quad \mathcal{E} \cup \mathcal{E}_1 \vdash \text{exp} : \tau,\, \mathcal{C}_1 \quad \text{solve2}(\mathcal{C} \cup \mathcal{C}_1)}{\mathcal{E} \vdash [(\text{id,pl,exp})] :}$$

$$\text{MESS_2} : \frac{\vdash \text{pl} : \tau l,\, \mathcal{E}_1,\, \mathcal{C} \quad \mathcal{E} \cup \mathcal{E}_1 \vdash \text{exp} : \tau,\, \mathcal{C}_1 \quad \text{solve2}(\mathcal{C} \cup \mathcal{C}_1) \quad \mathcal{E} \vdash \text{ml} :}{\mathcal{E} \vdash (\text{id,pl,exp})::\text{ml} :}$$

De même que pour les comportements, pl représente une liste de filtre. On a utilisé la fonction **solve2** qui résout les contraintes pour vérifier qu'il n'y a pas d'erreur. Le typage de message ne renvoie rien. En effet, on vérifie uniquement qu'ils ne contiennent pas d'erreur grossière.

Les paramètres :

Ces règles vont utiliser les règles sur les filtres du chapitre précédent, ainsi que la fonction **join**. Il s'agit simplement de construire la liste des types des filtres.

$$\text{PARAM_1} : \frac{\vdash \text{pat} : \tau,\, \mathcal{E},\, \mathcal{C}}{\vdash [\text{pat}] : [\tau],\, \mathcal{E},\, \mathcal{C}} \qquad \text{PARAM_2} : \frac{\vdash \text{pat} : \tau,\, \mathcal{E},\, \mathcal{C} \quad \vdash \text{pl} : \tau l,\, \mathcal{E}_1,\, \mathcal{C}_1}{\vdash \text{pat::pl} : \tau{::}\tau l,\, \text{join}(\mathcal{E},\, \mathcal{E}_1),\, \mathcal{C} \cup \mathcal{C}_1}$$

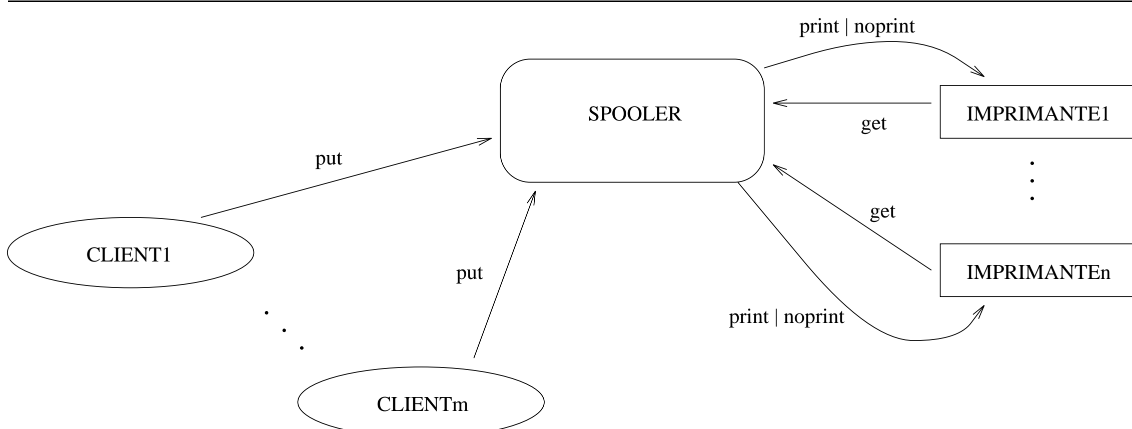
3.4 Des exemples.

Dans cette section, nous allons présenter trois exemples qui permettent de se faire une idée de ce qu'apporte le modèle d'acteur.

3.4.1 Un système d'impression en réseau.

La figure 3.4 représente schématiquement le système que l'on va décrire.

Figure 3.4 Impression en réseau.



Pour les clients, la commande d'impression sera «`send (put job) to spooler`». Elle suppose uniquement que tous les clients connaissent l'adresse `spooler`.

Le comportement d'un spooler sera :

```
behavior empty_spooler () =
  message put job =
    become jobed_spooler job
  message get iadd =
    send noprint to iadd
and jobed_spooler jobinit =
  val mutable jobs = [jobinit]
  message put job =
    jobs <- put_last job jobs
  message get iadd =
    send (print (hd jobs)) to iadd;
    jobs <- tl jobs;
    if jobs = [] then become empty_spooler ()
end
```

On a supposé l'existence d'une fonction `put_last`. Celle-ci met l'élément passé en premier paramètre à la fin de la liste passée en second paramètre.

Le comportement d'une imprimante sera alors :

```
behavior active_printer spooler timer =
  message print job =
    (* IMPRESSION DE job *)
    send (get ego) to spooler
  message noprint =
    send (start ego) to timer;
    become passive_printer spooler
and passive_printer spooler =
  message timeout timer =
    send (get ego) to spooler;
    become active_printer spooler timer
end
```

Ce comportement utilise un acteur de comportement *Timer* qui compte un certain temps à partir de la réception d'un message `start`;puis, renvoie un message `timeout` indiquant la fin de l'attente.

Le comportement de l'imprimante est composé de deux états : un état «active», où elle imprime des requêtes reçues ; un état «passive», qui est une mise en veille.

Cet exemple montre la puissance du modèle acteur. En effet, quelque soit le nombre de clients et le nombre d'imprimantes, le code du spooler est le même.

3.4.2 La construction de groupe de diffusion.

Pour créer un groupe de diffusion simple, nous allons utiliser un acteur qui sera chef du groupe. Lorsqu'un membre du groupe ou un acteur extérieur au groupe, voudra diffuser

quelque chose vers le groupe, il s'adressera à ce coordonnateur. Le comportement de celui-ci pourrait être le suivant :

```
behavior group_leader initmember =
  val mutable members = [initmember]
  message join add =
    members <- add::members
  message quit add =
    let rec rec_quit = function
      []   -> []
    | h::t -> if h = add then t
              else h::(rec_quit t)
    in members <- rec_quit members;
       if members = [] then become empty_group_leader ()
  message broadcast m =
    let rec rec_send = function
      []   -> ()
    | h::t -> send m to h;
              rec_send t
    in rec_send members
and empty_group_leader () =
  message join add =
    become group_leader add
end
```

3.4.3 Une matrice

Nous allons présenter une structure de donnée codée par des acteurs : une matrice. Cette structure de matrice est construite dans le but de servir à effectuer des produits matrice-vecteur. Chaque élément de la matrice sera un acteur. La construction de quatre types de comportement est nécessaire :

- L'élément (1,1) de comportement **matrix** initialise la matrice et lance les calculs.
- Les éléments de la première ligne (**line1**) sont chargés de décomposer le vecteur paramètre.
- Les éléments de la première colonne (**row1**) lancent le calcul des sommes sur les lignes.
- Les autres éléments (**elt**) effectuent leur produit local et participent à chaque somme de ligne.

De plus, chacun de ces éléments contiendra une valeur et pointera sur son voisin de droite et sur celui du dessous.

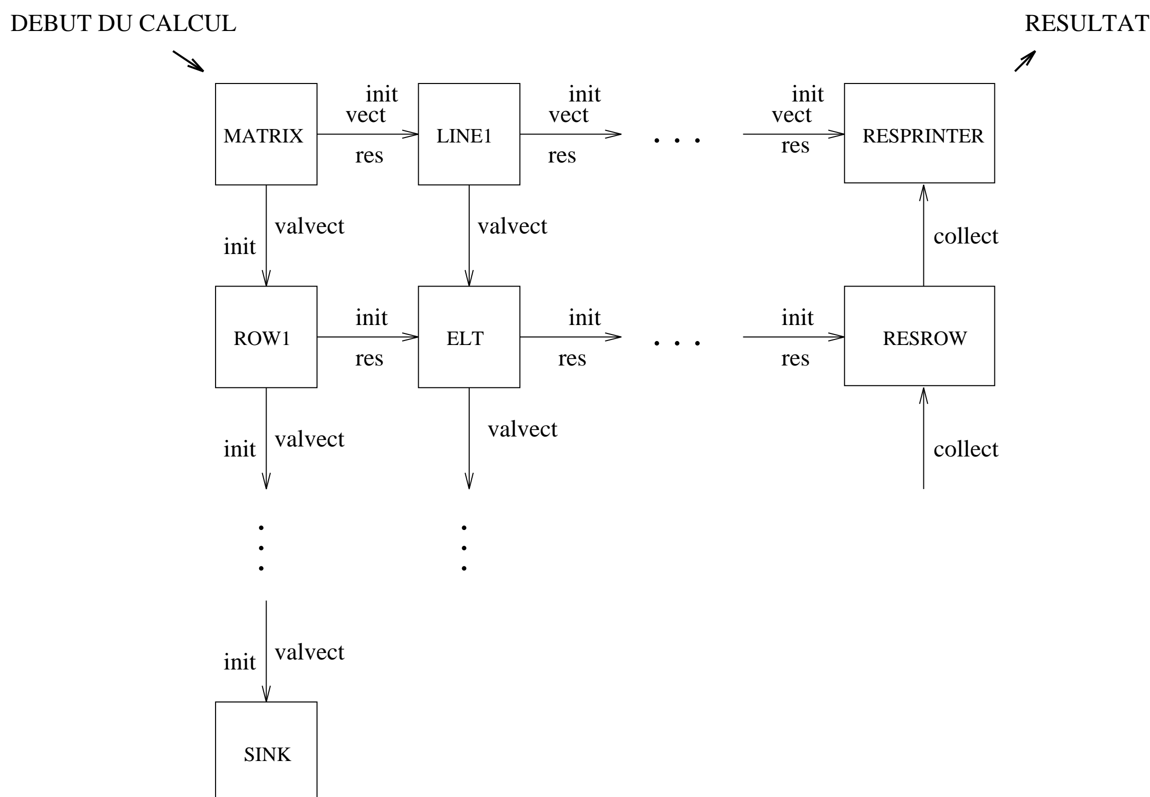
Pour gérer l'affichage et afin de ne pas séparer le cas de la dernière colonne et celui de la dernière ligne, on va, de plus, utiliser trois autres types d'acteur : poubelle (**sink**) pour absorber les messages issus de la dernière ligne et **resrow** et **resprinter** qui se chargeront, grâce un chaînage de bas en haut, de reconstruire le vecteur résultat et de l'afficher.

Tous ces comportements sont composés de trois états : actif ,en attente ou passif. Selon que l'acteur est prêt à effectuer des calculs, en attente d'un résultat intermédiaire, ou qu'il ait déjà fini ses calculs.

Afin d'éviter les mélanges de calcul, il est nécessaire d'attendre le résultat du calcul en cours avant de pouvoir en relancer un autre.

Remarquons que le code des différents acteurs ne dépend pas de la taille de la matrice. Par contre, il suppose que la matrice est bien construite et que le vecteur passé en paramètre a la bonne taille. La construction de la matrice est présenté dans le schéma 3.5.

Figure 3.5 Une matrice.




```
(* r = right ; d = down *)
behavior matrix r d v =
  message calc values =
    match values with
      h::t ->
        send init to d; (* initialisation des elements *)
        send init to r;
        send vect t to r; (* transmet les valeurs du vecteur *)
        send res (v*h) to r; (* lancement calcul resultat *)
        send valvect h to d (* valeur en dessous *)
    | _ ->
        print_string "Erreur : le vecteur n'a pas la bonne dimension"
end;;
behavior a_row1_elt r d v =
  message valvect value =
    become p_row1_elt r d v; (* on s'arrete *)
    send res (v*h) to r; (* lancement calcul resultat *)
```

```
    send valvect value to d (* transmet valeur du vecteur *)
and p_row1_elt r d v =
  message init =
    become a_row1_elt r d v; (* devient actif *)
    send init to r; (* relance sa ligne *)
    send init to d (* transmet initialisation *)
end;;
behavior a_line1_elt r d v =
  message vect values =
    match values with
      h::t ->
        become w_line1_elt r d v (v*h); (* on attend somme *)
        send vect t to r; (* transmet les valeurs du vecteur *)
        send valvect h to d (* valeur en dessous *)
    | _ ->
        print_string "Erreur : le vecteur n'a pas la bonne dimension";
        become p_line1_elt r d v
and w_line1_elt r d v1 v2 =
  message res v =
    become p_line1_elt r d v1; (* on s'arrete *)
    send res (v+v2) to r (* calcul somme+suite calcul resultat *)
and p_line1_elt r d v =
  message init =
    become a_line1_elt r d v; (* devient actif *)
    send init to r (* relance sa ligne *)
end;;
behavior a_resrow_elt u =
  message init = ()
  message res value =
    become w_resrow_elt u value
and w_resrow_elt u v =
  message collect l =
    become a_resrow_elt u; (* devient actif *)
    send collect (v::l) to u (* transmet liste resultat *)
end;;
behavior a_resprinter =
  message init = ()
  message vect v = ()
  message res value =
    become w_resprinter value
and w_resprinter v =
  message collect l =
    become a_resprinter; (* devient actif *)
    print_list (v::l) (* affichage resultat *)
end;;
behavior a_elt r d v =
  message valvect value =
    become w_elt r d v (v*value); (* on attend somme *)
```

```
    send valvect value to d (* transmet valeur du vecteur *)
and w_elt r d v1 v2 =
  message res v =
    become p_elt r d v1; (* on s'arrete *)
    send res (v+v2) to r (* calcul somme+suite calcul resultat *)
and p_elt r d v =
  message init =
    become a_elt r d v; (* devient actif *)
    send init to r (* relance sa ligne *)
end;;
behavior sink () =
  message init = ()
  message valvect v = ()
end;;
```

3.5 Réalisation

En fait, cette partie est complètement intégrée à ce qui a été présenté au chapitre précédent. L'unification, telle qu'elle avait été définie, doit être modifiée de la façon suivante :

- Ajout des types constants *Addr* et *Beh* aux types fonctionnels.
- Ajout du type $Mess$ qui est un peu particulier. En effet, ne pouvant simplement typer statiquement les messages, on a pris l'option de dire que toute application d'une entité de type $Mess$ aurait le type message. Ainsi, on ne vérifie pas le passage de paramètre à un message. Cela sera effectué par les outils d'analyse du terme CAP engendré.

Remarquons que seul le deuxième ajout demande une modification de l'algorithme. On va ajouter à l'algorithme le code suivant :

- deux messages : rien
- un message et une fonction de type $\tau_1 \rightarrow \tau_2$:
- unification de $Mess$ et τ_2.

REMARQUE 4 *La difficulté de typage des messages provient du fait que les messages peuvent contenir des paramètres de types différents selon le comportement qui va les traiter. Ainsi, par exemple, un acteur peut dans un état A traiter un message d'étiquette m prenant un entier en paramètre, et dans un état B le même message m avec une liste de caractères. Donc, la détection d'une erreur de construction de message correspond au fait que, dans tous les futurs comportements de l'acteur, il ne pourra jamais traiter ce message avec ces paramètres. Cette analyse relativement compliquée est actuellement réalisée sur le terme* CAP *(voir chapitre 4).*

3.6 Conclusion et perspectives.

Les acteurs présentés dans ce chapitre ne sont pas complètement spécifiés dans le sens où l'on n'a pas encore défini une sémantique précise du langage. De plus, dans le cadre d'une implantation plus intéressante du concept, il serait peut être souhaitable d'ajouter à

ML-ACT des primitives et notions de communication un peu plus riches. Par exemple, il serait intéressant d'ajouter au modèle une notion de groupe d'acteur et donc de *broadcast*. D'autre part, on pourrait intégrer également un *synchro_send* qui serait un envoi de message synchrone. Cet ajout permettrait de simplifier certains échanges d'information, mais il faciliterait l'apparition d'interblocages. Les diverses idées de ML-ACT et d'extensions possibles sont basées sur l'étude de [AH92] et de [GAP92].

Concernant plus précisément le typage, il conviendrait de revenir sur le typage des messages. On pourrait imaginer faire une première passe durant laquelle on relèverait tous les messages et on inférerait leur type. Puis, lors du typage à la rencontre d'un identificateur de message, on chercherait si un des types calculés convient.

Chapitre 4

L'EXTRACTION D'UN TERME CAP

Le Calcul d'Acteurs Primitifs (CAP) est un calcul de processus, proposé par l'équipe dans [CPSS95] et [CPS96] pour exprimer la sémantique des langages d'acteurs. Il est dit *primitif* car il ne contient aucune structure de contrôle séquentielle et aucune structure de donnée. En effet, la communication est une structure de contrôle suffisante pour effectuer des calculs et les acteurs suffisent pour coder des structures de données. Nous allons tout d'abord présenter ce calcul avant de présenter l'extraction d'un terme CAP à partir d'un programme ML-ACT.

4.1 Présentation de CAP.

Dans ce calcul, l'adresse d'un acteur est représentée par un nom qui est différent des autres identificateurs. Un comportement est composé d'une interface qui contient les messages qu'il sait traiter et un enregistrement privé qui liste les données que renferme l'acteur. L'interface est notée m_i dans la syntaxe et les champs privés sont notés p_j.

Un exemple d'expression du calcul qui présente la plupart des constructions de CAP, est :

$$\begin{aligned} \nu a,b,d(a \rhd [&read(c) = \zeta(e,s)(c \lhd rep(s.val) \parallel e \rhd s) \\ &write(v) = \zeta(e,s) e \rhd s.val \leftarrow v \quad , \\ &val = b] \parallel \\ a\lhd\ &write(d) \parallel \cdots) \parallel \cdots \end{aligned}$$

L'opérateur ν crée trois noms (a, b, d) dont la portée est comprise entre les deux parenthèses englobantes. La partie de cette expression que nous avons détaillée contient, en parallèle, un acteur d'adresse a et un message $write(d)$ qui lui est destiné. Le comportement de cet acteur a deux méthodes $read(c)$ et $write(v)$ ainsi qu'un champ privé val. Les points de suspension représentent le reste de l'expression sous la portée des noms a, b, c et à l'extérieur.

Une particularité de ce calcul est l'opérateur $\zeta(e, s)$ (zeta) qui lie, lors de la prise en compte d'un message, l'adresse de l'acteur à la variable e (que nous appellerons aussi **ego**) et son comportement courant à la variable s (aussi appelé **self**). Il permet d'exprimer le mécanisme de changement de comportement.

Nous allons uniquement présenter la syntaxe de CAP, mais avant tout, évoquons quelques idées sur sa sémantique.

Tout d'abord, il est clair que ne nous pouvons pas installer plusieurs comportements simultanément sur la même adresse. De plus, on considère que les noms libres ont été créés au niveau le plus englobant. Enfin, remarquons qu'il est possible d'avoir des messages *orphelins*, c'est à dire, qui ne seront jamais traités. Si un comportement ne sait pas traiter un message, plusieurs solutions sont envisageables :

- soit un mécanisme de délégation qui envoie ce message à un délégué qui le traitera,
- soit disposer d'un acteur particulier, généralement appelé *complaint*, qui récupère les messages que chaque acteur n'a pas su traiter,
- soit l'acteur peut mémoriser le message et se le renvoyer (avec le risque de saturer sa mémoire).

Le but de CAP est de tester des systèmes de type qui permettent de limiter au maximum l'existence de ces messages orphelins lors de l'exécution d'un programme d'acteur, ainsi que d'éviter la création erronée d'acteurs.

4.2 Sa syntaxe.

Le calcul repose sur la notion de *configuration* ($Config$) qui représente un ensemble d'acteurs s'exécutant en parallèle et de messages transitant dans le médium de communication et comprend :

- *la création de noms*,
- *le constructeur parallèle*,
- *la formation de messages*,
- *la création d'acteurs*,
- *la formation de comportements*,
- *la sélection de champs*,
- *la mise à jour de champs.*

Un acteur est représenté par un couple de termes ($T_1 \rhd T_2$) dénotant respectivement son *adresse* et son *comportement*. Un message est un triplet *destinataire*, *étiquette* (m) et *contenu* ($T_1 \lhd m(\widetilde{T_2})$). Les termes ($Terme$) dans la syntaxe représentent des comportements, des mises à jour, des sélections, des noms ou des variables. On dispose des ensembles et de leurs membres suivant :

- N un ensemble infini de noms - $a \in N$,
- V un ensemble infini de variables - $x, e_i, s_i \in V$,
- L un ensemble infini d'étiquettes - $m_i, p_j \in L$.

Un élément de N^* et V^* sera noté avec un tilde comme $\tilde{x_i} \in V^*$. De plus, on représentera une configuration ($Config$) par C_i, et un terme ($Terme$) par T_j.

L'ensemble des expressions du calcul est donné par la grammaire de la figure 4.1.

Figure 4.1 La grammaire de CAP.

$Config$	$::=$	$\emptyset$	configuration vide
	$\mid$	$(Config)$	parenthésage
	$\mid$	$Config \parallel Config$	composition parallèle
	$\mid$	$\nu a\ Config$	restriction
	$\mid$	$Terme \triangleright Terme$	construction d'acteur
	$\mid$	$Terme \triangleleft m(\widetilde{Terme})$	construction de message
$Terme$	$::=$	$[m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i\,, p_j = T_j]_{i\in 1..n, j\in 1..l}$	comportement
	$\mid$	$Terme.p_j \leftarrow Terme$	mise à jour
	$\mid$	$Terme.p_j$	sélection
	$\mid$	a	nom
	$\mid$	x	variable
	$\mid$	$(Terme)$	parenthésage

4.3 Un exemple.

Le comportement d'une cellule a deux méthodes : *read*, qui prend en paramètre le nom de l'acteur à qui devra être envoyé son contenu, et *write*, dont l'argument est la valeur à conserver. Un tel exemple se code aisément grâce au champ privé *val*, qui contient l'état courant de la cellule.

$$\begin{aligned} Cellule(val_init) \stackrel{\text{def}}{=} [&read(c) = \zeta(e,s)(c \triangleleft rep(s.val) \parallel e \triangleright s) \\ &write(v) = \zeta(e,s)e \triangleright s.val \leftarrow v\,, \\ &val = val_init\,] \end{aligned}$$

Nous voyons, sur cet exemple, comment se spécifie un changement de comportement, la variable e qui capture l'**ego** est réinstanciée avec le nouveau comportement. Dans le cas du *read*, il n'y a pas de changement de comportement, e est donc réassocié à s (**self**). Dans le cas du *write*, le nouveau comportement est le même que l'ancien, on modifie uniquement la valeur du champ *val*.

4.4 Sémantique de la génération de CAP.

4.4.1 Contexte de l'extraction.

La traduction de ML-ACT en CAP va être effectuée en deux étapes, une étape de construction d'un arbre abstrait représentant le programme et une seconde étape de génération de CAP proprement dite. En fait, on va utiliser une syntaxe abstraite particulière qui va servir de pivot entre la syntaxe abstraite de ML-ACT et la syntaxe concrète de CAP. Ce mode de conception a été choisi pour deux raisons principales. D'une part, afin d'alléger les règles de sémantique guidant la traduction. D'autre part, dans le but de simplifier et donc de rendre plus souple et plus compréhensible le programme lui-même.

Pour générer le terme CAP correspondant à un programme, on effectue donc une passe sur l'arbre syntaxique décoré avec les types. Au cours de celle-ci, on va construire d'une part un arbre abstrait CAP et d'autre part on va légèrement modifier l'arbre abstrait du programme pour faciliter la génération de code que l'on abordera dans le chapitre suivant. L'arbre abstrait obtenu, dont la syntaxe est décrite dans la figure 4.2, est une sorte de

terme CAP de plus haut niveau. En effet, on n'a pas supprimé la récursivité mutuelle des comportements qui n'existe pas en CAP. D'autre part le calcul des **self** et **ego** n'a pas été effectué.

Le type «data» correspond aux données qu'elles soient des variables, des accès à des champs, des **ego** ou des noms (adresses,messages et comportements). Le type «bind» va servir à mémoriser si une variable de filtrage doit être liée ou non avec le paramètre effectif, lors de l'application de celui-ci. Le cas «Maybe» est utilisé lorsque le filtre est un identificateur de type variable. En effet, on ne saura s'il faut le conserver, qu'au moment de l'application selon le type de donnée qui lui est transmise. Les noms des configurations sont relativement explicites et représentent à la fois des entités CAP (**Com** pour les messages, **Actor** pour les acteurs...) et des objets qui seront utiles à la traduction mais ne seront pas traduits (**Closure**,**New**,**Appbeh** et **Appmess**).

Figure 4.2 La syntaxe abstraite pivot de la traduction en CAP.

data	= **Var**	string	
	= **Val**	string	
	= **Adr**	string	
	= **Mess**	string	
	= **Beh**	string	
	= **Ego**	string	
bind	= **Yes**	string	
	= **Maybe**	string	
	= **No**		
conf	= **Empty**		rien
	= **Data**	data	donnée
	= **Com**	data*string*conf list	message
	= **Actor**	data*string*conf list	acteur
	= **Suicide**		suicide
	= **Parallel**	conf list	parallèle
	= **Create**	string list*conf	création de nom
	= **Set**	string*conf	mise a jour
	= **Beh**	(string*bind list*val list*mess list) list	définition de comportement
	= **Closure**	conf*(string*conf) list*bind list	fermeture fonctionnelle
	= **New**	string*conf list	création d'acteur
	= **Appmess**	string*conf list	construction de message
	= **Appbeh**	string*conf list	application de comportement
val	=	string*conf	
mess	=	string*string list*conf	

4.4.2 Première étape de traduction.

Pour présenter les règles de sémantique concernant cette première étape de la traduction, nous allons utiliser des fonctions de traduction plutôt que des règles d'inférence. Ces fonctions seront définies par filtrage, c'est-à-dire que l'on appliquera à chaque fois la première règle qui s'unifiera au niveau des paramètres et des résultats. On suppose que lorsque aucunes des formules ne peut être utilisée, c'est qu'il y a une erreur. On va construire une fonction $\mathbb{T}$ qui va traduire un arbre abstrait de programme ML-ACT en un arbre abstrait de la syntaxe que l'on vient de présenter. Pour cela, on construira plusieurs fonctions selon l'entité à traduire (**TF**,**TE**,...).

Les filtres :

La fonction qui va traduire les filtres sera notée **TF**, elle aura deux paramètres : le filtre à traduire et un booléen indiquant si on est un filtre englobant ou englobé. Son rôle va être de dire si le filtre peut contenir de la communication. Elle permet également de construire l'environnement de toutes les variables du filtre. Le résultat de **TF** est un couple composé du type de liaison du filtre et d'un environnement associant les variables du filtre et leur valeur actuelle. En fait, la seule valeur produite sera «*Empty*» pour les filtres de type fonctionnel. Cela permet d'éliminer directement toutes les opérations utilisant des identificateurs dont on sait que le type de liaison est «No», et qui sont inutiles dans l'analyse de communication. Le booléen utilisé comme argument permet de savoir si le filtre est un identificateur ou si c'est un autre type de filtre. En effet, seuls les identificateurs peuvent «contenir» de la communication (ie soit représenter une adresse ou soit contenir un comportement). Cette restriction est due au bas niveau de CAP et impose que les entités communicables ne soit pas conservées dans des structures de donnée (Tuple ou Liste).

Afin de simplifier l'apparence la fonction de traduction, on a supprimé les types dans les constructeurs représentant l'arbre. Pour y accéder, on utilisera **Typepart** qui capturera le type de l'entité en cours de traitement.

$$
\begin{array}{lcl}
\textbf{TF}(\text{true,Var id}) & = & \textit{si}\ \text{Typepart}\ \textit{est} \\
 & & \quad \text{Adr s}\ \textit{alors}\ \text{Yes s},\{\} \\
 & & \quad \text{Beh s}\ \textit{alors}\ \text{Yes s},\{\} \\
 & & \quad \text{Var s}\ \textit{alors}\ \text{Maybe s},\{\} \\
 & & \textit{sinon}\ \text{No},\{\text{s=Empty}\} \\
\textbf{TF}(\text{false,Var id}) & = & \text{No},\{\text{s=Empty}\}
\end{array}
$$

Le traitement d'une variable comporte donc deux cas : soit le filtre est une variable (booléen à true), soit la variable est interne à un filtre complexe.

$$\textbf{TF}(_,\text{Any}) = \textbf{TF}(_,\text{Constant}\ _) = \textbf{TF}(_,\text{Or}(_,_)) = \text{No},\{\}$$

Le fait que l'on puisse traduire directement l'alternative, provient de l'hypothèse imposée par le typage, à savoir qu'un filtre de type «or» ne contient pas d'identificateur.

$$
\begin{array}{lcllcl}
\textbf{TF}(_,\text{Cons(h,t)}) & = & \text{No},\mathcal{E}_1 \cup \mathcal{E}_2 & \text{avec} & \textbf{TF}(\text{false,h}) = _,\mathcal{E}_1 \\
 & & & & \textbf{TF}(\text{false,t}) = _,\mathcal{E}_2 \\
\textbf{TF}(_,\text{Tuple}[\text{f}_1;...]) & = & \text{No},\bigcup_i \mathcal{E}_i & \text{avec} & \textbf{TF}(\text{false,f}_i) = _,\mathcal{E}_i
\end{array}
$$

Ces règles montrent que lors du parcours en profondeur d'un filtre, on transmet «false» comme argument aux appels de la fonction de traduction. Remarquons que le typage étant effectuer, on sait que les $\mathcal{E}_i$ sont tous totalement séparés.

Les expressions :

La fonction d'analyse des expressions **TE** utilisent un environnement qui contient les variables de filtre et leur valeur. Cet environnement est construit par la fonction vue ci-dessus. Elle produit un arbre abstrait de la syntaxe de la figure 4.2.

Les règles qui suivent, indiquent que la partie fonctionnelle est oubliée. Elles imposent, notamment, qu'il n'y ait pas de données communicables dans les listes et les tuples, et

qu'il n'y ait pas de communication (ie de send, become...) dans les filtrages («match») et les conditionnelles («if»). Dans le traducteur réalisé, on vérifie que ces restrictions sont appliquées.

$$\mathbf{TE}(\text{Constant }_) = \mathbf{TE}(\text{Tuple }_) = \mathbf{TE}(\text{Cons}(_,_)) = \mathbf{TE}(\text{Match}(_,_)) = \mathbf{TE}(\text{Ifthenelse}(_,_,_)) = \text{Empty}$$

$$\begin{array}{lcl}
\mathbf{TE}(\text{Ident(s,Actor)},_) & = & \text{Data(Adr s)} \\
\mathbf{TE}(\text{Ident(s,Behavior)},_) & = & \text{Data(Beh s)} \\
\mathbf{TE}(\text{Ident(s,Message)},_) & = & \text{Data(Mess s)} \\
\mathbf{TE}(\text{Ident(s,Val)},_) & = & \text{Data(Val s)} \\
\mathbf{TE}(\text{Ident(s,Value)},\mathcal{E}) & = & \textit{si } \text{search(s},\mathcal{E})\text{=c } \textit{alors } \text{c} \\
 & & \textit{sinon } \text{Data(Var s)} \\
\mathbf{TE}(\text{Ident(s,Patvar)},\mathcal{E}) & = & \textit{si } \text{search(s},\mathcal{E})\text{=c } \textit{alors } \text{c}
\end{array}$$

Dans les six règles ci-dessus, on a utilisé une version de «Ident» légèrement différente de la syntaxe abstraite présenté au chapitre 2. En effet, au cours du typage, on modifie l'arbre afin de lui intégrer la famille de chaque identificateur. Dans les deux dernières règles, on utilise la fonction **search** déjà définie. L'utilisation qui en est faite pour traduire les variables de filtre (cas Patvar) spécifie que si s ne figure pas dans l'environnement, il y a une erreur.

Le résultat de l'analyse d'une séquence nécessite l'analyse des expressions qui la compose et est transformée en un «parallel».

$$\mathbf{TE}(\text{Sequence}(e_1,e_2),\mathcal{E}) = \text{Parallel } [\mathbf{TE}(e_1,\mathcal{E});\mathbf{TE}(e_2,\mathcal{E})]$$

En CAP, on ne suppose pas d'ordre dans l'exécution des instructions. Il est possible d'effectuer directement cette traduction parce que la séquence de ML-ACT n'impose un ordre que sur les opérations fonctionnelles, qui ici disparaissent, ainsi que sur les accès aux champs privés et leurs mises à jour. Le problème des accès aux champs privés sera résolu lors de la génération de CAP. Pour cela, on effectuera un parcours dans l'ordre de toutes les instructions, en modifiant la valeur des champs : c'est ce que j'ai appelé le calcul du **self**.

$$\mathbf{TE}(\text{Send}(e_1,e_2),\mathcal{E}) = \text{Com(d,lbl,arg) avec } \begin{array}{l} \mathbf{TE}(e_1,\mathcal{E}) = \text{Appmess(lbl,arg)} \\ \mathbf{TE}(e_2,\mathcal{E}) = \text{Data d} \end{array}$$

Lors de la traduction de l'envoi de message, on retrouve les limitations imposées par le typage. On impose de plus, que la première expression du send soit effectivement un message (ie une application d'un identificateur de message à plusieurs arguments). Il est, par exemple, interdit d'utiliser une conditionnelle («if») dont les branches contiendraient deux messages différents.

$$\begin{array}{lcl}
\mathbf{TE}(\text{Set(id,e)},\mathcal{E}) & = & \text{Set(id},\mathbf{TE}(\text{e},\mathcal{E})) \\
\mathbf{TE}(\text{Ego},_) & = & \text{Data Ego} \\
\mathbf{TE}(\text{Suicide},_) & = & \text{Suicide}
\end{array}$$

Les trois cas précédents consistent simplement en une traduction directe d'une syntaxe à l'autre.

$$\mathbf{TE}(\text{Become e},\mathcal{E}) = \text{Actor(Ego,lbl,arg)}$$
$$\mathbf{TE}(\text{New e},\mathcal{E}) = \text{New(lbl,arg)}$$
$$\text{avec } \mathbf{TE}(\text{e},\mathcal{E}) = \text{Appbeh(lbl,arg)}$$

Le changement de comportement et la création d'acteurs imposent tous deux que l'expression qui les compose soit un comportement appliqué à qui on a fournit tous ses paramètres.

$$\mathbf{TE}(\text{Let(Nonrecursive,f,e)},\mathcal{E}) = \text{Create}(l_1,\text{Parallel } (\mathbf{TE}(\text{e},\mathcal{E} \cup \mathcal{E}_1)::l_2)) \text{ avec } \mathbf{TL}(\text{f},\mathcal{E}) = l_1,l_2,\mathcal{E}_1$$
$$\mathbf{TE}(\text{Let(Recursive,f,e)},\mathcal{E}) = \text{Create}(l_1,\text{Parallel } (\mathbf{TE}(\text{e},\mathcal{E} \cup \mathcal{E}_1)::l_2)) \text{ avec } \mathbf{TL}(\text{f,predef(f},\mathcal{E})) = l_1,l_2,\mathcal{E}_1$$

L'analyse des blocs de définitions locales est similaire, que le bloc soit récursif ou non. La seule différence entre les deux est le fait que l'on utilise la fonction **predef** qui prédéfinie les fonctions récursives en leur attribuant «*Empty*» (aucune communication) comme valeur. Cette limite est imposée par le dépliage des fonctions qui n'est pas possible pour les fonctions récursives.

$$\mathbf{TE}(\text{Function } [(p_1,e_1);...],\mathcal{E}) = \text{Closure}(\mathbf{TE}(e_1,\mathcal{E} \cup \mathcal{E}_1),[],[b]) \text{ avec } \mathbf{TF}(p_1,\text{true}) = b,\mathcal{E}_1$$

La traduction de fonction est un peu particulière, elle ne prend en compte que la première branche du filtre. En effet, la traduction en CAP impose au programme d'être déterministe (qu'il n'y ait pas de choix) mais on veut pouvoir utiliser le «function» pour les filtrages à une seule «possibilité» (qui correspondent à l'abstraction classique du λ-calcul).

$$\mathbf{TE}(\text{Apply(e},[e_1;...]),\mathcal{E}) = \text{capofapply}(\mathbf{TE}(\text{e},\mathcal{E}),[\mathbf{TE}(e_1,\mathcal{E});...])$$

Enfin, la traduction de l'application est l'opération la plus complexe. On utilise la fonction **capofapply** pour masquer la complexité de cette règle. On suppose que l'application a été dépliée pour l'application de cette règle, c'est-à-dire que la fonction (e) n'est pas une application. L'algorithme de la fonction **capofapply**(c, cl) (cl=[c_1;...]) est le suivant :

- *selon que c est :*
  - *une fermeture :*
    - *on construit l'environnement résultat du passage des paramètres (sachant que les paramètres associés à un No ne sont pas pris en compte, ainsi que ceux qui sont associés à des $Maybe$, et qui correspondent à une valeur Empty) et la liste l des paramètres restant à appliquer*
    - *si l est vide :*
      - *la fonction dispose de tous ces paramètres, on remplace donc l' application par le corps de la fonction dans lequel on a substitué les noms des paramètres par leurs valeurs*
    - *sinon :*
      - *on renvoie une fermeture avec le nouvel environnement*
  - *un message $Mess$(nom) :*

- on renvoie un Appmess(nom,cl)
- un message Beh(nom) :
- on renvoie un Appbeh(nom,cl)
- sinon : erreur

Les définitions :

Une définition est, en fait, une liste de couple (id,exp) où id est un identificateur et exp l'expression qui lui est associée. Sa fonction de traduction **TL** construit trois objets :

- La liste des noms d'acteurs créés,
- La liste du code de ces créations,
- Un environnement qui contient la valeur CAP des fonctions.

Pour simplifier l'apparence de la fonction de traduction, on ne va traiter que la définition d'une fonction. Le cas d'une liste de fonction se traite par concaténation des listes produites par chacune des définitions et union de tous les environnements obtenus. Cette fonction portant sur une seule définition est notée **TL'**.

TL'((id,e),$\mathcal{E}$) =	*si* **TE**(e,$\mathcal{E}$) = New(lbl,arg) *alors*
	[id],[Actor(Adr(id),lbl,arg)],{}
	sinon [],[],{id=**TE**(e,$\mathcal{E}$)}

Une instruction :

Parmi les quatre types d'instructions, la traduction de la première est simplement une traduction d'expression et les deux suivantes sont issues directement des règles de traduction des liaisons lexicales.

TI(Eval e,$\mathcal{E}$)	=	**TE**(e,$\mathcal{E}$),$\mathcal{E}$
TI(Value(Nonrecursive,f),$\mathcal{E}$)	=	Create(l_1,Parallel l_2),$\mathcal{E} \cup \mathcal{E}_1$
		avec **TL**(f,$\mathcal{E}$) = l_1,l_2,$\mathcal{E}_1$
TI(Value(Recursive,f),$\mathcal{E}$)	=	Create(l_1,Parallel l_2),$\mathcal{E} \cup \mathcal{E}_1$
		avec **TL**(f,predef($\mathcal{E}$,f)) = l_1,l_2,$\mathcal{E}_1$
TI(Beh b,$\mathcal{E}$)	=	Beh(**TB**(b,$\mathcal{E}$)),$\mathcal{E}$

La traduction de la formation des comportements va utiliser la fonction **TB** décrite ci-dessous.

Les comportements, champs et messages :

A nouveau pour simplifier la présentation, on va se consacrer à la traduction d'un comportement. Le cas général se déduit immédiatement par un simple parcours de la liste des comportements et la construction de la liste des résultats.

Présentons d'abord une petite fonction auxiliaire **TP**, qui va extraire la liste des paramètres de chaque message.

$$\begin{array}{lcl} \mathbf{TP}(\text{Yes s::pl}) & = & \text{s::}\mathbf{TP}(\text{pl}) \\ \mathbf{TP}(\text{_::pl}) & = & \mathbf{TP}(\text{pl}) \end{array}$$

La traduction d'un comportement sera effectué par la fonction **TB'**:

$$\begin{array}{l} \mathbf{TB'}((\text{id},[\text{p}_1;...],[(\text{mut}_1,\text{id}_1,\text{e}_1);...],\ [(\text{id}'_1,\text{pl}_1,\text{e}'_1);...]),\mathcal{E}) = \\ \quad (\text{id},[\text{p}'_1;...],[(\text{id}_1,\mathbf{TE}(\text{e}_1,\mathcal{E}'));...],\ [(\text{id}'_1,\mathbf{TP}(\text{pl}_1),\mathbf{TE}(\text{e}'_1,\mathcal{E}');...]) \\ \quad\quad \text{avec } \mathbf{TF}(\text{p}_i,\text{true}) = \text{p}'_i,\mathcal{E}_i \text{ et } \mathcal{E}'=\mathcal{E} \cup \bigcup_i \mathcal{E}_i \end{array}$$

Un Programme :

Finalement, la traduction $\mathbb{T}$ du programme consistera en un simple parcours de la liste en analysant chaque instruction.

$$\mathbb{T}([\text{i}_1;...]) = [\text{i}'_1;...] \text{ avec } \mathbf{TI}(\text{i}_j,\mathcal{E}_{j-1}) = \text{i}'_j,\mathcal{E}_j \text{ et } \mathcal{E}_0 = \{\}$$

4.4.3 Deuxième étape de traduction.

Ensuite, le parcours de l'arbre abstrait construit va permettre le dépliage des comportements mutuellements récursifs pour les transformer en comportement simple de CAP. Simultanément, on calcule les **self**. En fait, ceux-ci sont représentés par une chaîne de caractères qui initialement vaut "s" concaténé à la valeur d'un compteur. Puis, pour chaque mise à jour (*Set*), on la transforme en (*self*.*champ*=*valeur*).

La première étape de cette deuxième phase est une passe sur l'arbre issu de l'analyse ci-dessus afin de supprimer tous les termes inutiles. Par exemple, on supprime tous les «*Empty*» dans un «*Parallel*», on supprime également les champs privés de contenu vide, on aplatit les «*Parallel*»...

Ensuite, on va utiliser les fonctions présentées ci-après. Celles-ci vont utiliser les données suivantes :

nom : (string) Le nom du comportement courant,

num : (int) Un entier pour numéroter les **ego** et **self** lors de la capture par le ζ,

self : (string) La chaîne de caractère représentant le **self**, celle-ci suit la loi déjà décrite,

futur : (End | Same | Change) Il permet de mémoriser la rencontre avec des commandes de changement de comportement,

passé : ((string * string) list) Environnement qui va contenir les comportements qui englobe le code actuellement analysé ainsi que leur **self**.

Pour diminuer la quantité de paramètres à écrire, on notera l'ensemble de ces données $\mathcal{E}$ sauf le futur (sur lequel des filtrages auront lieu). De plus, on référencera une de ses

composantes par une notation pointée ($\mathcal{E}$.nom). Pour indiquer la modification d'une de ces composantes, on utilisera la notation : $\mathcal{E}$.passé $\leftarrow$ *valeur*.

La fonction de traduction principale **T** va être de la forme **T**(expression,futur,environnement) et retournera trois objets :le code, qui sera une chaîne de caractère contenant le terme CAP, et les valeurs de «futur» et «self» après le traitement. Pour décrire le code, on utilisera des guillemets (") pour indiquer les caractères constants que l'on ajoute au code, et on supposera que la concaténation est représentée par la simple juxtaposition (ie «se» représentera la chaîne résultat de la concaténation du contenu de «s» et de celui de «e»). La chaîne vide sera représentée par $\emptyset$. La fonction **up** sera utilisée durant la traduction, elle transforme une chaîne de caractère en mettant en majuscule sa première lettre.

Tout d'abord, présentons une fonction auxiliaire qui va traduire les données :

$$\begin{array}{lcl}
\mathbf{T}_1(\text{Var } s,_,_) & = & s \\
\mathbf{T}_1(\text{Val } s,_,self) & = & self\texttt{"."}s\texttt{"_v"} \\
\mathbf{T}_1(\text{Adr } s,_,_) & = & \text{up}(s) \\
\mathbf{T}_1(\text{Mess } s,_,_) & = & s \\
\mathbf{T}_1(\text{Beh } s,_,_) & = & s \\
\mathbf{T}_1(\text{Ego},n,_) & = & \texttt{"e"}n
\end{array}$$

où n est un entier qui représente l'indice du comportement courant et $self$ est une chaîne de caractères contenant la valeur courante du self.

Remarquons que dans le cas d'un champ privé (Val), on ajoute un "_v" au nom de celui-ci. En effet, pour que le dépliage des comportements se termine, ceux-ci ne doivent pas avoir de paramètres. On transforme donc chaque paramètre en un champ privé. L'ajout du suffixe prévient, alors, d'un possible conflit de nom entre les paramètres ("_p") et les champs privés ("_v").

Afin d'alléger la présentation de la traduction, on suppose disposer d'un environnement global $\mathcal{E}_{beh}$. Celui-ci va contenir, au fur et à mesure de la traduction, la valeur de tous les comportements actuellement définis. Donc, à la rencontre d'un nœud Beh(s,pl,vl,ml), on ajoutera à $\mathcal{E}_{beh}$ un couple (s,(pl,vl,ml)). Comme dans les chapitres précédents, on accédera à la valeur d'un comportement par la fonction **search**.

Pour générer le code d'un comportement, on utilisera la fonction $\mathbf{T}_2$ dont les paramètres sont le nom du comportement courant (s), l'environnement courant ($\mathcal{E}$) et la liste des arguments (s_i).

$$\begin{array}{ll}
\mathbf{T}_2(s,\mathcal{E},[s_1;...]) = & \texttt{"["}sm_1\sqcup\ a_1^1\sqcup...\texttt{"=\$(e"}n\texttt{,"s"}n\texttt{")("}scm_1\texttt{")}\sqcup\texttt{"} \\
& ... \\
& \texttt{","}sv_1\texttt{"_v="}scv_1\sqcup\ ... \\
& p_1\texttt{"_p="}s_1\sqcup\ ...\ \texttt{"]"} \\
\text{où}\quad n=\mathcal{E}.\text{num}+1 & \mathcal{E}'=(\mathcal{E}.\text{num}\leftarrow n).\text{passé}\leftarrow(\mathcal{E}.\text{nom},\mathcal{E}.\text{self}) \\
\quad \text{search}(s,\mathcal{E}_{beh}) = ([p_1;...],[(sv_1,cv_1);...],[(sm_1,[a_1^1;...],cm_1);...]) & \\
\quad \mathbf{T}(cv_i,\text{Same},\mathcal{E}') = scv_i,_,_ & \\
\quad \mathbf{T}(cm_i,\text{Same},\mathcal{E}') = temp_i,\text{futur},_ & \\
\quad \textit{si}\ \text{futur} = \text{Same}\ \textit{alors} & \\
\qquad scm_i = temp_i\texttt{"|e"}n\texttt{">s"}n & \\
\quad \textit{sinon}\ scm_i = temp_i &
\end{array}$$

Remarquons que lors de la traduction d'un message, on ajoute le comportement courant

au passé et on incrémente le compteur. Lorsque cette partie de traduction est terminée et que le futur vaut «Same» alors cela signifie que l'on boucle sur son propre comportement.

De plus, on suppose que si un des s_i est vide, alors on n'écrit pas la définition de champ privé correspondante.

Donnons maintenant la fonction de traduction **T** d'un terme de la syntaxe pivot en un terme CAP. Tout d'abord, la sémantique du suicide, qui correspond à un «exit» et impose qu'après la rencontre d'un suicide tout le code est inutile.

$$\mathbf{T}(_,\text{End},\mathcal{E}) = \emptyset,\text{End},\mathcal{E}$$

Cette règle est donnée en premier car c'est elle qui est prioritaire.

$$\begin{array}{lcl}
\mathbf{T}(\text{Empty},\text{futur},\mathcal{E}) & = & \emptyset,\text{futur},\mathcal{E}.\text{self} \\
\mathbf{T}(\text{Suicide},\text{Same},\mathcal{E}) & = & \emptyset,\text{End},\mathcal{E}.\text{self} \\
\mathbf{T}(\text{Set}(s,c),\text{futur},\mathcal{E}) & = & \emptyset,\text{f},\text{self}\texttt{"."}s\texttt{"="}sc \\
 & & \text{avec } \mathbf{T}(c,\text{futur},\mathcal{E}) = sc,\text{f},\text{self} \\
\mathbf{T}(\text{Data } d,\text{futur},\mathcal{E}) & = & \mathbf{T}_1(d,\mathcal{E}.\text{num},\mathcal{E}.\text{self}),\text{futur},\mathcal{E}.\text{self}
\end{array}$$

Le fait d'imposer un futur de valeur «Same» dans la traduction du suicide, signifie que l'on interdit le suicide si il y a déjà eu un changement de comportement.

$$\begin{array}{c}
\mathbf{T}(\text{Com}(d,s,[c_1;...]),\text{futur},\mathcal{E}) = \mathbf{T}_1(d,\mathcal{E}.\text{num},\mathcal{E}.\text{self})\texttt{"<"}s_\sqcup s_{1\sqcup}...,\text{f}_n,\text{self}_n,\text{f},\text{self}\texttt{"."}s\texttt{"="}sc \\
\text{avec } \mathbf{T}(c_i,\text{f}_{i-1},\mathcal{E}_{i-1}) = s_i,\text{f}_i,\text{self}_i \\
\text{f}_0\text{=futur} \text{ , } \mathcal{E}_0\text{=}\mathcal{E} \text{ et } \mathcal{E}_i\text{=}\mathcal{E}_{i-1}.\text{self}\leftarrow\text{self}_i
\end{array}$$

L'envoi de message se traduit directement. On fait passer d'argument en argument les valeurs du futur et du self. On traduit ainsi l'ordre d'évaluation des paramètres de ML-ACT qui se fait de gauche à droite.

$$\begin{array}{ll}
 & \mathbf{T}(\text{Parallel}[c_1;...],\text{futur},\mathcal{E}) = s_1\texttt{"|"}...,\text{f}_n,\text{self}_n \\
\text{avec} & \mathbf{T}(c_i,\text{f}_{i-1},\mathcal{E}_{i-1}) = s_i,\text{f}_i,\text{self}_i \\
 & \text{f}_0 = \text{futur} \text{ , } \mathcal{E}_0\text{=}\mathcal{E} \text{ et } \mathcal{E}_i\text{=}\mathcal{E}_{i-1}.\text{self}\leftarrow\text{self}_i
\end{array}$$

La mise en parallèle de configuration consiste uniquement en l'ajout entre chacune de leur valeur d'un |, ainsi que la transmission de gauche à droite du futur et du self.

$$\begin{array}{c}
\mathbf{T}(\text{Create}([s_1;...],c),\text{futur},\mathcal{E}) = \texttt{"\~{}"}\text{up}(s_1)...\texttt{"("}s\texttt{")"},\text{f},\text{self} \\
\text{avec } \mathbf{T}(c,\text{futur},\mathcal{E}) = s,\text{f},\text{self}
\end{array}$$

La création de nom ajoute des tildes (˜) et met la première lettre en majuscule de tous les nouveaux noms.

$$\begin{array}{ll}
 & \mathbf{T}(\text{Actor}(\text{Ego},s,[c_1;...]),\text{Same},\mathcal{E}) = \texttt{"e"}\mathcal{E}.\text{num}\texttt{">"}\textit{temp},\text{Change},\text{self}_n \\
\text{avec} & \textit{si } \text{search}(s,\mathcal{E}.\text{passé}) = c \textit{ alors} \\
 & \quad \textit{temp} = \texttt{"(...("}c\texttt{"."}p_1\texttt{"="}s_1\texttt{")...)"} \\
 & \textit{sinon temp} = \mathbf{T}_2(s,\mathcal{E},[s_1;...]) \\
 & \text{search}(c,\mathcal{E}_{beh}) = ([p_1;...],_,_) \\
 & \mathbf{T}(c_i,\text{Change},\mathcal{E}_{i-1}) = s_i,_,\text{self}_i \\
 & \mathcal{E}_0\text{=}\mathcal{E} \text{ et } \mathcal{E}_i\text{=}\mathcal{E}_{i-1}.\text{self}\leftarrow\text{self}_i
\end{array}$$

Le changement de comportement doit être unique, on impose donc un futur de valeur «Same». Sa traduction se scinde en deux cas selon que le nouveau comportement soit un comportement englobant (il appartient alors au passé) ou non. Dans le premier cas, il s'agit juste de mettre à jour les paramètres, alors que dans le second, il faut débuter un nouveau dépliage de comportement.

$$\mathbf{T}(\text{Actor}(d,s,[c_1;...]),\text{futur},\mathcal{E}) = \mathbf{T}_1(d,\mathcal{E})\texttt{">"}\mathit{temp},\text{f}_n,\text{self}_n$$

$$\begin{array}{ll} \text{avec} & \mathit{si}\ \text{search}(s,\mathcal{E}.\text{passé}) = c\ \mathit{alors} \\ & \quad \mathit{temp} = \texttt{"(...("}c\texttt{"."}p_1\texttt{"="}s_1\texttt{")...)"} \\ & \mathit{sinon}\ \mathit{temp} = \mathbf{T}_2(s,\mathcal{E},[s_1;...]) \\ & \text{search}(c,\mathcal{E}_{beh}) = ([p_1;...],_,_) \\ & \mathbf{T}(c_i,\text{f}_{i-1},\mathcal{E}_{i-1}) = s_i,\text{f}_i,\text{self}_i \\ & \text{f}_0 = \text{futur}\ ,\ \mathcal{E}_0{=}\mathcal{E}\ \text{et}\ \mathcal{E}_i{=}\mathcal{E}_{i-1}.\text{self}{\leftarrow}\text{self}_i \end{array}$$

Enfin, la création d'un acteur est similaire au changement de comportement exception faite de l'hypothèse sur le futur.

4.5 Réalisations.

Pour cette partie du compilateur j'ai écrit les 4 modules suivants :

comsynt.ml : La syntaxe abstraite pivot de la traduction.

comedtree.ml : La syntaxe abstraite typée légèrement modifiée.

com_checker.ml : L'analyseur générant les arbres correspondant aux programmes dans les syntaxes ci-dessus.

cap_generator.ml : La génération effective du terme CAP.

Cette partie m'a demandé beaucoup de travail car il fallu essayer de maîtriser le formalisme ardu qu'est CAP, et passer d'un langage de haut niveau relativement permissif au niveau des constructions, vers un langage plus contraignant. Pour exposer le travail réalisé, nous allons présenter le résultat obtenu sur un programme exemple.

4.6 Un exemple.

L'exemple classique que nous allons mettre en œuvre est le cas des philosophes. Le problème est le suivant : n philosophes se partagent n fourchettes. Pour manger, ils ont besoin d'avoir deux fourchettes. Leur comportement alterne entre la faim et la réflexion. Les fourchettes sont réparties de façon à ce qu'un philosophe ait une fourchette à sa droite et une autre à sa gauche.

Afin d'avoir un terme CAP de taille raisonnable, on s'est limité à deux philosophes et deux fourchettes.

```
behavior free_fork () =
  message getL a = send okL to a;become lefted_fork ()
  message getR a = send okR to a;become righted_fork ()

and lefted_fork () =
```

```
  message getR a = send nopeR to a
  message leaveL = become free_fork ()

and righted_fork () =
  message getL a = send nopeL to a
  message leaveR = become free_fork ()
end;;
```

Le comportement d'une fourchette comporte donc trois état selon qu'elle soit libre (free_fork), prise par le philosophe à sa droite (righted_fork) ou celui à sa gauche (lefted_fork).

```
behavior thinking_philo g d =
  message hungry = send (getL ego) to g;send (getR ego) to d;
                   become hungry_philo g d
  message nopeL = send hungry to ego
  message nopeR = send hungry to ego

and hungry_philo g d =
  message okL = become lefted_forked_philo g d
  message okR = become righted_forked_philo g d
  message nopeL = become thinking_philo g d;send hungry to ego
  message nopeR = become thinking_philo g d;send hungry to ego
  message hungry = ()

and lefted_forked_philo g d =
  message okR = send leaveR to d; send leaveL to g;
                become thinking_philo g d; send hungry to ego
  message nopeR = send leaveL to g; become thinking_philo g d;
                  send hungry to ego
  message hungry = ()

and righted_forked_philo g d =
  message okL = send leaveR to d; send leaveL to g;
                become thinking_philo g d;send hungry to ego
  message nopeL = send leaveR to d;become thinking_philo g d;
                  send hungry to ego
  message hungry = ()
end;;
```

Le comportement d'un philosophe comporte quatre états, selon qu'il pense (thinking_philo), qu'il ait faim (hungry_philo), qu'il ait sa fourchette droite (righted_forked_philo) ou sa fourchette gauche (lefted_forked_philo).

```
let rec f1 = new free_fork ()
    and f2 = new free_fork ()
    and p1 = new thinking_philo f1 f2
    and p2 = new thinking_philo f2 f1
in send hungry to p1;
   send hungry to p2;;
```

Enfin, le programme consiste donc, en la création de deux fourchettes et deux philosophes, ainsi qu'a l'envoi de messages donnant faim aux philosophes.

Le terme CAP généré par le programme est retranscrit dans la figure 4.3 ; on a mis le résultat sous une forme «lisible». On y a changé légèrement les identificateurs générés par le compilateur.

Figure 4.3 Deux philosophes et deux fourchettes en CAP.

$$
\begin{array}{l}
\nu F_1\ \nu F_2\ \nu P_1\ \nu P_2(\\
P_1 \triangleleft \mathit{hungry}() \parallel \\
P_2 \triangleleft \mathit{hungry}() \parallel \\
F_1 \triangleright [\mathit{getL}(a) = \zeta(e_1, s_1)(a \triangleleft \mathit{okL}() \parallel e_1 \triangleright [\mathit{getR}(a) = \zeta(e_2, s_2)(a \triangleleft \mathit{nopeR}() \parallel e_2 \triangleright s_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{leaveL}() = \zeta(e_2, s_2)(e_2 \triangleright s_1)]) \\
\qquad \mathit{getR}(a) = \zeta(e_1, s_1)(a \triangleleft \mathit{okR}() \parallel e_1 \triangleright [\mathit{getL}(a) = \zeta(e_2, s_2)(a \triangleleft \mathit{nopeL}() \parallel e_2 \triangleright s_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{leaveR}() = \zeta(e_2, s_2)(e_2 \triangleright s_1)])] \parallel \\
F_2 \triangleright [\mathit{getL}(a) = \zeta(e_1, s_1)(a \triangleleft \mathit{okL}() \parallel e_1 \triangleright [\mathit{getR}(a) = \zeta(e_2, s_2)(a \triangleleft \mathit{nopeR}() \parallel e_2 \triangleright s_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{leaveL}() = \zeta(e_2, s_2)(e_2 \triangleright s_1)]) \\
\qquad \mathit{getR}(a) = \zeta(e_1, s_1)(a \triangleleft \mathit{okR}() \parallel e_1 \triangleright [\mathit{getL}(a) = \zeta(e_2, s_2)(a \triangleleft \mathit{nopeL}() \parallel e_2 \triangleright s_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{leaveR}() = \zeta(e_2, s_2)(e_2 \triangleright s_1)])] \parallel \\
P_1 \triangleright [\mathit{hungry}() = \zeta(e_1, s_1)(s_1.g \triangleleft \mathit{getL}(e_1) \parallel s_1.d \triangleleft \mathit{getR}(e_1) \parallel \\
\qquad\qquad\qquad\qquad\qquad e_1 \triangleright [\mathit{okL}() = \zeta(e_2, s_2)(e_2 \triangleright [\mathit{okR}() = \zeta(e_3, s_3)(s_3.d \triangleleft \mathit{leaveR}() \parallel s_3.g \triangleleft \mathit{leaveL}() \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}()) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{nopeR}() = \zeta(e_3, s_3)(s_3.g \triangleleft \mathit{leaveL}() \parallel e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}())\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g = s_2.g \quad d = s_2.d]) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{okR}() = \zeta(e_2, s_2)(e_2 \triangleright [\mathit{okL}() = \zeta(e_3, s_3)(s_3.d \triangleleft \mathit{leaveR}() \parallel s_3.g \triangleleft \mathit{leaveL}() \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}()) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{nopeL}() = \zeta(e_3, s_3)(s_3.d \triangleleft \mathit{leaveR}() \parallel e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}())\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g = s_2.g \quad d = s_2.d])\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad g = s_1.g \quad d = s_1.d])\ , \\
\qquad g = F_1 \quad d = F_2] \parallel \\
P_2 \triangleright [\mathit{hungry}() = \zeta(e_1, s_1)(s_1.g \triangleleft \mathit{getL}(e_1) \parallel s_1.d \triangleleft \mathit{getR}(e_1) \parallel \\
\qquad\qquad\qquad\qquad\qquad e_1 \triangleright [\mathit{okL}() = \zeta(e_2, s_2)(e_2 \triangleright [\mathit{okR}() = \zeta(e_3, s_3)(s_3.d \triangleleft \mathit{leaveR}() \parallel s_3.g \triangleleft \mathit{leaveL}() \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}()) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{nopeR}() = \zeta(e_3, s_3)(s_3.g \triangleleft \mathit{leaveL}() \parallel e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}())\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g = s_2.g\ d = s_2.d]) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{okR}() = \zeta(e_2, s_2)(e_2 \triangleright [\mathit{okL}() = \zeta(e_3, s_3)(s_3.d \triangleleft \mathit{leaveR}() \parallel s_3.g \triangleleft \mathit{leaveL}() \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}()) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{nopeL}() = \zeta(e_3, s_3)(s_3.d \triangleleft \mathit{leaveR}() \parallel e_3 \triangleright (s_1.g \Leftarrow s_3.g).d \Leftarrow s_3.d \parallel e_3 \triangleleft \mathit{hungry}())\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g = s_2.g \quad d = s_2.d])\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad g = s_1.g \quad d = s_1.d])\ , \\
\qquad g = F_2 \quad d = F_1]))))
\end{array}
$$

4.7 Conclusion

La traduction en CAP nécessite un certain de degré de déterminisme, il faut, par exemple, qu'il n'y ait pas de choix dans les communications, ce qui restreint beaucoup le nombre des acteurs traduisibles. On ne peut pas, par exemple, traduire le programme d'impression du chapitre précédent. De plus, le fait qu'il n'y ait ni type de base, ni constructeur de type dans ce calcul de processus, interdit également la traduction du programme de groupe. En effet, celui-ci utilise des listes pour conserver des adresses et effectue des envois dans des parties récursives. Deux pistes sont envisageables afin de réduire ces faiblesses :

- Étendre CAP en lui ajoutant un opérateur de choix (+) et en lui ajoutant la manipulation de donnée par des ajouts de type constants ainsi que des constructeurs.
- La deuxième possibilité serait de passer outre CAP et de générer directement des contraintes de communication. En effet, CAP n'est qu'un support minimaliste utilisé comme source de contraintes, celle-ci étant analysées par la suite. On pourrait donc à partir du programme ML-ACT générer directement ces contraintes.

Chapitre 5

LA GÉNÉRATION DE CODE CAML

Ce chapitre va présenter la structure choisie pour l'implantation des acteurs en ML, ainsi que la stratégie de génération de code réalisée. La présentation sera faite à travers le commentaire du code engendré pour un exemple.

5.1 Les outils

Remarquons, avant tout, que la génération de code actuelle est simpliste. En effet, nous simulons le parallélisme implicite entre les acteurs par leur encapsulation dans des processus légers. Il est clair que si l'on souhaitait réaliser un véritable langage concurrent, il serait nécessaire de l'implanter dans le cadre de «vraies» machines parallèles ou distribuées. Mais un tel développement dépasse très largement le cadre de notre étude.

Les processus légers que nous utilisons sont des «objets» qui exécutent un programme indépendamment les uns des autres. En fait, ces processus vont se partager le temps de calcul affecté au processus UNIX qui les contient. Les principales différences avec les processus (lourds) UNIX sont : ils possèdent des primitives de synchronisations et de communications de haut niveau, on peut en créer un nombre très important et ils n'ont pas de structure arborescente «père-fils» qui les relient.

Ceux-ci sont implantés dans le monde CaML sous la forme d'une librairie (*threads*). Parmi les modules fournissant des primitives de synchronisation, nous utiliserons **Mutex** et **Condition** qui mettent à disposition respectivement des sémaphores d'exclusion mutuelle et des variables conditions.

5.2 Les structures

Chaque *acteur* sera traduit en un processus léger dont le code correspondant à l'exécution d'une fonction associée à son comportement. Dans une boucle infinie, il scrutera sa boite aux lettres et à chaque fois qu'un message y sera, il exécutera la réaction qui lui est associée.

Les *boites aux lettres* des différents acteurs ne sont pas incluses dans leur processus, mais, pour simplifier l'envoi de messages, elles sont situées dans la mémoire commune à tous les acteurs. Ainsi, l'envoi d'un message consistera uniquement en son dépôt dans la «bonne» boite aux lettres. La structure de ces boites aux lettres est donc celle d'un quadruplet contenant : un sémaphore d'exclusion mutuelle, une variable condition et deux

queues. Le mutex protège l'accès à la structure en empêchant tout accès concurrent. La variable condition permettra de réactiver un acteur devenu passif en attendant des messages. Enfin, les deux queues n'ont pas de spécification précise. En fait, on peut imaginer tester différents types de stockage (liste, file, tableau,...). Il s'agit des structures qui conserveront les messages en attente de traitement. Une seule hypothèse est faite sur leur fonctionnement : la première des deux queues contient les messages qui n'ont jamais été consultés, la deuxième contient ceux qui ont déjà été consultés et qui n'ont pu être traités. De plus, lors de l'exécution d'un acteur, on ne consultera la deuxième queue que lors des changements de comportement, afin d'essayer de les traiter.

Dans le but de conserver tous les *messages* dans une même structure de données, ceux-ci devront tous avoir le même aspect, celui d'un triplet. Celui-ci sera composé d'une étiquette, un argument et une clef. A cette fin, on suppose que tous les paramètres d'un message forment un n-uplet qui sera son argument. Le filtrage afin de déterminer la réaction de l'acteur s'effectue sur l'étiquette. La clef, ajoutée artificiellement, permet de simuler le déséquencement possible des messages. En effet, lors de la construction d'un message, on tirera un nombre aléatoire qui sera sa clef ; et lors de la mise du message dans la boite aux lettres, on le classera selon sa clef.

Enfin, la dernière entité du modèle d'acteurs à décrire est le *comportement*. Celui-ci sera traduit en une fonction qui aura les différents champs privés comme données. Celles-ci seront définies au moyen de liaisons lexicale, et si elles sont modifiables, seront implantées sous forme de référence. La fonction ainsi construite effectuera une boucle infinie de test sur l'état de la boite aux lettres et filtrera les messages sur leur étiquette. Le changement de comportement d'un acteur consistera en l'appel de la fonction correspondant au nouveau comportement. Ainsi, le graphe d'appel de ces fonctions mutuellement récursives simulera l'automate des différents états possibles d'un acteur.

On peut résumer la fonction de traduction par le tableau suivant :

comportement	fonction
adresse d'acteur	entier unique
acteur	processus léger
messages	(étiquette,argument,clef)
boite aux lettres	(mutex,condition,queue1,queue2)
envoi de message	mise dans la «bonne» queue
création d'acteur	création :processus léger,boite aux lettres et adresse
changement de comportement	appel de la «bonne» fonction

5.3 L'implantation

A chaque compilation d'un fichier contenant un programme en ML-ACT, nous produisons :

- un fichier **address.ml** qui fournit le type qui abstrait les adresses. Son code est :

```
type t = Add of int
```

- un fichier **message.ml** qui fournit les types *étiquette*, *argument* et *message*. Les noms d'étiquettes, ainsi que les différents types d'arguments possibles sont collectés par analyse du programme lors de la compilation. Son code est :

```
type label =
  L_label1
| ...
| L_labeln

type argument =
  Parg1 of type1
| ...
| Pargm of typem

type t =
  { lbl : label;
    arg : argument;
    key : int }
```

- un fichier **fifo.ml** qui abstrait les queues utilisées dans les boite aux lettres. Le fait de les implanter dans un module séparé permet de changer leur structure aisément. Moyennant la conservation de la même interface, on peut envisager générer plusieurs type de structure laissant ainsi le choix de la structure au programmeur. Son interface doit être :

```
exception Empty  (* queue vide   *)
exception Full   (* queue pleine *)

type 'a t  (* queue *)
(* doit etre une structure modifiable *)

val create : unit -> 'a t
(* creation d'une queue *)

val take : 'a t -> 'a
(* prend un element dans la queue *)
(* si la queue est vide : Empty   *)

val add : Message.t -> Message.t t -> unit
(* met un message dans la queue       *)
(* non polymorphe car utilise la clef *)
(* pour mettre message a sa place     *)
(* si queue est pleine : Full         *)

val empty 'a t -> bool
(* teste si la queue est vide *)
```

- un fichier **mailbox.ml** qui implante les boites aux lettres. Au quadruplet de la structure a été ajouté un indicateur afin d'indiquer dans quelle queue on va chercher le prochain message à traiter. Son code est :

```
type situation =
  Queue1
| Queue2

type t = { mutex  : Mutex.t;
           cond   : Condition.t;
           queue1 : Message.t Fifo.t; (* queue premiere fois *)
           queue2 : Message.t Fifo.t; (* queue deja testes   *)
           mutable state : situation }

let create ()  =
  { mutex  = Mutex.create ();
    cond   = Condition.create ();
    queue1 = Fifo.create ();
    queue2 = Fifo.create ();
    state  = Queue1 }

let take bal =
  Mutex.lock bal.mutex;
  let mess =
    (match bal.state with
       Queue1 ->
       (* on prend dans la queue premiere fois*)
       while Fifo.empty bal.queue1 do
         Condition.wait bal.cond bal.mutex
       done;
       (* queue1 n'est pas vide *)
       Fifo.take bal.queue1
     | Queue2 ->
       (* on peut prendre dans la queue testes *)
         while (Fifo.empty bal.queue1 & Fifo.empty bal.queue2) do
           Condition.wait bal.cond bal.mutex
         done;
     (* on va essayer d'abord queue2 *)
         if (not (Fifo.empty bal.queue2)) then
         (* queue2 n'est pas vide *)
           Fifo.take bal.queue2
         else
        (* queue1 n'est pas vide *)
          Fifo.take bal.queue1) in
  Mutex.unlock bal.mutex;
  mess

let put bal mess =
(* on met un message - ENVOI - *)
  Mutex.lock bal.mutex;
  Fifo.add mess bal.queue1;
  Condition.signal bal.cond;
```

```
  Mutex.unlock bal.mutex

let put_back bal mess =
(* on remet un message - ECHEC TRAITEMENT- *)
  Mutex.lock bal.mutex;
  Fifo.add mess bal.queue2;
  Condition.signal bal.cond;
  Mutex.unlock bal.mutex

let set bal =
(* on a change d'etat, on peut donc regarder dans queue2 *)
  bal.state <- Queue2
```

– un fichier **medium.ml** qui fournit la structure de tableau extensible qui va contenir toutes les boites aux lettres, ainsi que la fonction d'envoi de message. Son code est :

```
open Message
exception Fatal_error
type t = (Mailbox.t array option) ref (* les boites aux lettres *)

let create () = ref None (* creation du medium *)

let add medium =
(* ajout d'une boite, en fait creation d'un acteur *)
  let lb = Mailbox.create () in
  let arb = Array.create 1 lb in
  let new_ar = (match !medium with
                  None     -> arb
                | Some arr -> Array.append arr arb) in
  medium := Some new_ar;
  Address.Add (Array.length new_ar -1)

let send label args dest medium =
(* envoi d'un message a un acteur *)
  match !medium with
    None -> raise Fatal_error
  | Some m ->
      let random = Random.int 100 in (* 100 nombre max genere *)
      let mess = { lbl = label;
                   arg = args;
                   key = random } in
      let id = (match dest with Address.Add i -> i) in
      let lb = Array.get m id in
      Mailbox.put lb mess

let my_mailbox add medium =
(* recuperation d'une boite aux lettres *)
  match !medium with
    None -> raise Fatal_error
```

```
  | Some m ->
      (match add with
         Address.Add i -> Array.get m i)
```

- un fichier **Makefile** qui permet la compilation et l'édition de lien afin de produire l'exécutable correspondant au programme.
- enfin, le fichier ML résultat de la traduction du programme ML-ACT. Pour initialiser l'environnement où s'exécuteront les acteurs, il aura l'entête suivante :

```
open Message
exception Dead

let medium = Medium.create ()  (* le medium *)

let create_add () = Medium.add medium
(* creation d'un acteur *)

let send label args dest =
(* envoi de message *)
  let send_proc () =
    Medium.send label args dest medium;
    Thread.exit
  in Thread.create send_proc ();()

let next_message id =
(* prochain message *)
  let lb = Medium.my_mailbox id medium in
  Letter_box.take lbn_cut 1;

let put_back_message id mess =
(* remet le message *)
  let lb = Medium.my_mailbox id medium in
  Letter_box.put_back lb mess

let set_chgt id =
(* signale le changement de comportement *)
  let lb = Medium.my_mailbox id medium in
  Letter_box.set lb
```

5.4 La stratégie de traduction

Au cours de la traduction, on va construire deux listes qui contiendront respectivement toutes les étiquettes de message du programme et tous les arguments possibles pour les messages. Elles permettrons de générer le fichier message.ml déjà décrit.

Afin d'éviter les conflits possibles entre les variables du programme et celles auxiliaires générées automatiquement, on ajoute systématiquement un «_» à la fin de chaque variable du programme.

Nous allons commencer par présenter la traduction d'un comportement. Pour cela donnons la forme générale d'un comportement en ML-ACT :

```
behavior nom =
 val v1= val1
  ...
 val mutable vm1= valm1
  ...
 message m1 =
  code
  ...
end;;
```

Il sera traduit en une fonction CaML dont l'allure est :

```
let rec nom_ id =
 let v1_ = T(val1) in
  ...
 let vm1_ = ref(T(valm1)) in
  ...
 set_chgt id;
 try
  while true do
   let mess = next_message id in
   match (mess.lbl,mess.arg) with
    (L_m1,Parg?) ->
     begin
     T(code)
     end
   | ...
   | _ ->put_back_message id mess
  done
 with Dead ->Thread.exit ()
```

La fonction T utilisée est la fonction de traduction des expressions ML-ACT en expression ML. Nous allons la présenter sous la forme d'une définition par cas, en les séparant selon le type de l'entité traduite.

Les constantes :

T(Int i)	=	i	T(Nop)	=	`()`
T(Char c)	=	c	T(Nil)	=	`[]`
T(String s)	=	s	T(False)	=	`false`
T(Float s)	=	*float_of_string* s	T(True)	=	`true`

La traduction des constantes est immédiate, il s'agit simplement d'une réécriture.

Les filtres :

T(Any)	=	_	T(Cons(h,t))	=	T(h)::T(t)
T(Var s)	=	s_	T(Or(p_1,p_2)	=	T(p_1)|T(p_2)
T(Constant c)	=	T(c)	T(Tuple [p_1;...;p_n])	=	(T(p_1),...,T(p_n))

La seule particularité de la traduction des filtres est l'ajout du «_» commenté précédemment.

Les expressions :

T(Constant c)	=	T(c)	T(Ident(s,Operator))	=	s
T(Ident(s,Message))	=	L_s	T(Ident(s,_))	=	s_
T(Ident(s,Mutval))	=	!s_			

La traduction d'un identificateur varie selon le type de celui-ci. En effet, les étiquettes de message se sont vues ajouter un «L_» pour les transformer en constructeur. L'accès à une valeur modifiable doit comporter une opération de déréférencement, car on code celles-ci par une référence. Enfin, les opérateur de ML-ACT et CaML étant identique, il ne faut pas leur ajouter le «_». Dans tous les autres cas, on ajoute, comme on l'a déjà vu, ce caractère.

Supposons que lc soit la liste [(p_1,e_1);...] avec p_i qui est le filtre et e_i l'expression associée.

```
T(Function lc)  =  (function T(p1) -> T(e1)
                   ...)
T(Match(e,lc))  =  (match T(e) with
                   T(p1) -> T(e1)
                   ...)
```

La traduction des filtrages et fonctions définies par filtrage est directe. Il en est de même pour les conditionnelles, tuple et construction de liste :

```
T(Ifthenelse(e1,e2,None))  =  (if T(e1) then T(e2))
T(Ifthenelse(e1,e2,e3))    =  (if T(e1) then T(e2) else T(e3))
T(Cons(h,t))               =  T(h)::T(t)
T(Tuple [p1;...;pn])       =  (T(p1),...,T(pn))
```

La traduction de la séquence soulève le problème de l'exécution des instructions situées après un changement d'état. Pour cela, on suppose disposer d'une fonction qui vérifie si la séquence contient un changement d'état et d'une autre qui transforme e_1;e_2 en e'_1;Become(s,[a_1;...]);e'_2 où e'_1 est la partie de la séquence avant le changement d'état et e'_2 celle après.

Donc si il n'y pas de changement d'état :

```
T(Sequence(e1,e2))  =  T(e1);T(e2)
```

Et dans le cas contraire :

```
T(Sequence(e1,e2))  =  T(e'1);
                       Thread.create(fun id -> s_ id  T(a1)...T(an)) id ;
                       T(e'2);raise Dead
```

La politique choisie est donc de recréer un nouveau processus léger qui effectue le changement d'état, et de faire exécuter les calculs qui restent par le processus courant qui s'arrêtera après.

```
T(Apply(Ident(s,Operator),[e1;e2]))  =  T(e1) s T(e2)
T(Apply(e,[e1;...]))                 =  T(e) T(e1)...
```

L'application doit être séparé selon que la fonction soit infixe ou non.

```
T(Send((s,[a1;...]),e))  =  send L_s (Parg? T(a1)...) T(e)
T(Become(s,[a1;...]))    =  (s_ id  T(a1)...T(an))
```

La syntaxe de l'envoi de message et celle du changement d'état sont légèrement différentes de la version donnée au chapitre 2. Le but est de simplifier ces deux règles en conservant directement pour l'envoi : l'étiquette du message et ses arguments ; pour le changement de comportement : le nom du comportement et ses arguments d'initialisation. L'envoi d'un message consiste en un appel de la fonction «send» et le changement d'état correspond à un appel de la fonction associée au nouveau comportement.

```
T(Suicide)  =  raise Dead
T(Ego)      =  id
T(Set(s,e)) =  s_:=  T(e)
```

Le suicide consiste en la levée de l'exception de fin de vie d'un acteur. Ego n'est qu'une simple référence à l'identité de l'acteur qui s'exécute, soit `id`. La mise à jour d'un champ privé modifiable se traduit en une affectation de la référence lui correspondant.

```
T(Letrec((ln,lf),e))  =  let n1_ = create_add ()
                         ...
                         in
                         let rec nf1_ = T(e1)
                         ...
                         in
                         Thread.create(fun id -> n'1_ id T(a^1_1) ... T(a^1_p1)) n1_;
                         ...
                         T(e)
T(Let((ln,lf),e))     =  let n1_ = create_add ()
                         ...
                         and nf1_ = T(e1)
                         ...
                         in
                         Thread.create(fun id -> n'1_ id T(a^1_1) ... T(a^1_p1)) n1_;
                         ...
                         T(e)
```

Pour présenter la traduction des liaisons lexicales, nous utiliserons une syntaxe quelque peu différente de celle qui était utilisée précédemment. On suppose que lors du typage, on modifie en conséquence l'arbre abstrait représentant le programme. Leur forme sera Letrec((ln,lf),e) et Let((ln,lf),e) où ln est la liste des création d'acteurs et lf celle des définitions de fonction. La liste ln vaut $[(n_1,n'_1,[a_1^1;...;a_{p_1}^1]);...]$ avec n_i qui est le nom de l'acteur créé, n'_i le nom de son comportement initial et la liste est celle des arguments initialisant ce comportement. De même, lf vaut $[(nf_1,e_1);...]$ avec nf_i qui est le nom de la fonction et e_i sa valeur.

Pour des raisons de compatibilité avec le système de type de CaML, les créations d'adresse, dans le cas récursif, ne peuvent pas être intégrées dans le `let rec`. Notons que la création d'un acteur est séparée en deux phases distinctes : la création de l'adresse et de la boite aux lettres et la création du processus léger qui va exécuter le comportement de l'acteur.

REMARQUE 5 *On ajoute en fin de programme une boucle sans fin afin de laisser s'exécuter les processus créés par le programme. Donc pour arrêter l'exécution du programme traduit, il faudra généralement une interruption système.*

5.5 Réalisations

Pour cette partie du compilateur j'ai écrit les 3 modules suivants :

make_auxfile.ml : La génération automatique des fichiers présentés précédemment : address.ml, fifo.ml, mailbox.ml et medium.ml.

make_makefile.ml : Génère le fichier Makefile.

ml_generator.ml : Produit les fichiers : message.ml déjà vu et le fichier qui contient le code CaML correspondant à la traduction. Ce dernier est sauvegarder sous le nom : nom_.ml où nom est celui du programme source sans l'extension mla.

Pour exécuter les programme ainsi traduit, il faut utiliser la commande make associée au fichier Makefile généré, afin d'effectuer la compilation CaML nécessaire.

La présentation du résultat produit par le compilateur sur un exemple permet de se rendre compte de la traduction effectuée.

5.6 Un exemple

Afin de diversifier les exemple présenté dans ce rapport, nous ne présenterons pas la traduction des philosophes. A la place, nous allons utiliser un programme qui implante un crible d'Eratosthène.

```
behavior mult value master =
  message start =
    if value > 100 then suicide
    else (become (generator value master);
          send next to ego;
          let a = new mult (value+1) master in
          send start to a)
and generator value master =
  val mutable num = 2
```

```
  message next =
    let res = value*num in
    if res > 100 then suicide
    else (send (reply res) to master;
          num <- num+1;
          send next to ego)
end;;

behavior master =
  val mutable l = [0]
  message start =
    let a = new mult 2 ego in
    send start to a
  message reply num = l <- num::l
  message get c = send (reply l) to c
end;;

behavior viewer master =
  message start = send (get ego) to master
  message reply l =
    let rec print_liste = function
      [] -> ()
    | h::t -> print_int h;print_liste t
    in
      print_liste l;
      become waiter 15 master;
      send top to ego
and waiter value master =
  val mutable time = value
  message top =
    if time = 0 then (become viewer master;
                      send start to ego)
    else (time <- time -1;
          send top to ego)
end;;

let rec maitre = new master
    and printer = new viewer maitre
in send start to printer;
   send start to maitre;;
```

Ce programme utilise trois entités : un maître, un afficheur et un calculateur (esclave). L'afficheur se contente de demander une liste d'entiers à intervalle fixe et de l'afficher à l'écran quand il la reçoit. Le maître lance le calcul par la création d'un premier esclave puis ajoutera les entiers reçus à sa liste interne. Les esclaves se reproduisent, à chaque début de calcul, ils créent un autre esclave qui s'occupera du nombre suivant. Ils envoient tous les multiples de leur valeur initiale au maître (indiquant que le nombre n'est alors pas premier) et se suicident lorsqu'ils ont atteint 100.

```
(* entete *)

(*********************)
(* CODE DU PROGRAMME *)
(*********************)
```

```
let rec mult_ id value_ master_=
  set_normal id;
  try
    while true do
      let mess = next_message id in
      match (mess.lbl,mess.arg) with
        (L_start,Parg1) ->
        begin
        (if (value_ > 100) then (raise Dead)
         else (Thread.create (fun id ->
                 generator_ id value_ master_) id;
               (send L_next Parg1 id);
               (let a_ = create_add ()
               in
                 Thread.create (fun id -> mult_ id (value_ + 1) master_) a_;
                 (send L_start Parg1 a_));
           raise Dead))
        end
      | _ -> put_back_message id mess
    done
  with Dead -> Thread.exit ()
and generator_ id value_ master_=
  let num_ = ref(2) in
  set_normal id;
  try
    while true do
      let mess = next_message id in
      match (mess.lbl,mess.arg) with
        (L_next,Parg1) ->
        begin
        let res_ = (value_ * !num_)
        in
          (if (res_ > 100) then (raise Dead)
          else ((send L_reply (Parg2(res_)) master_);
                (num_:=(!num_ + 1));
                (send L_next Parg1 id)))
        end
      | _ -> put_back_message id mess
    done
  with Dead -> Thread.exit ()

let rec master_ id =
  let l_ = ref((0::[])) in
  set_normal id;
  try
    while true do
      let mess = next_message id in
      match (mess.lbl,mess.arg) with
        (L_start,Parg1) ->
        begin
        let a_ = create_add ()
        in
          Thread.create (fun id -> mult_ id 2 id) a_;
          (send L_start Parg1 a_)
```

```
        end
      | (L_reply,Parg2(num_)) ->
        begin
        (l_:=(num_::!l_))
        end
      | (L_get,Parg3(c_)) ->
        begin
        (send L_reply (Parg4(!l_)) c_)
        end
      | _ -> put_back_message id mess
    done
  with Dead -> Thread.exit ()

let rec viewer_ id master_=
  set_normal id;
  try
    while true do
      let mess = next_message id in
      match (mess.lbl,mess.arg) with
        (L_start,Parg1) ->
        begin
        (send L_get (Parg3(id)) master_)
        end
      | (L_reply,Parg4(l_)) ->
        begin
        let rec print_liste_ = (function
          [] -> ()
        | h_::t_ -> (print_int h_);
                    (print_liste_ t_))
        in
          (print_liste_ l_);
          Thread.create (fun id ->
            waiter_ id 15 master_) id;
          (send L_top Parg1 id);
          raise Dead
        end
      | _ -> put_back_message id mess
    done
  with Dead -> Thread.exit ()
and waiter_ id value_ master_=
  let time_ = ref(value_) in
  set_normal id;
  try
    while true do
      let mess = next_message id in
      match (mess.lbl,mess.arg) with
        (L_top,Parg1) ->
        begin
        (if (!time_ = 0) then (Thread.create (fun id ->
                                 viewer_ id master_) id;
                               (send L_start Parg1 id);
                               raise Dead)
        else ((time_:=(!time_ - 1));
              (send L_top Parg1 id)))
        end
```

```
      | _ -> put_back_message id mess
    done
  with Dead -> Thread.exit ()

let _ =
  let printer_ = create_add ()
  and maitre_ = create_add ()
  in
    Thread.create (fun id -> viewer_ id maitre_) printer_;
    Thread.create (fun id -> master_ id ) maitre_;
    (send L_start Parg1 printer_);
    (send L_start Parg1 maitre_);

let _ = while true do () done
```

5.7 Perspectives de développement

Dans le cadre de l'implantation d'un véritable compilateur, il serait sans doute souhaitable de changer les structures présentées dans ce chapitre et ne plus utiliser ni de mémoire commune, ni de processus léger. Cette étude, plus complexe, dépasse le cadre de ce projet de DEA et sera traitée ultérieurement. Dans ce but, il faut au préalable définir une sémantique formelle de ML-Act. Cette étude est l'objet de la suite du rapport.

Chapitre 6

SÉMANTIQUE FORMELLE DE MINI-ACT.

Afin de simplifier la présentation de la sémantique de ML-ACT et de s'abstraire des détails d'implantation du langage, nous utilisons un langage intermédiaire : mini-Act. Sa syntaxe est relativement proche de ML-ACT pour rester représentatif de toutes les difficultés inhérentes à celui-ci. L'exposé de la sémantique du langage se fera par le biais de mini-Act, nous allons donc commencer par présenter ce langage.

6.1 La syntaxe de mini-Act.

La syntaxe est composée de deux entités : les expressions et les filtres. Les termes construits en utilisant la grammaire de la figure 6.1 produisent les deux ensembles : $\mathcal{E}xp$ et $\mathcal{F}il$. Dans les règles de la grammaire de mini-Act, nous avons utilisé les conventions suivantes :

- c représente une constante. L'ensemble des constantes sera noté $\mathcal{C}$ et il contiendra : les entiers, les flottants,..., () et []. Pour simplifier l'exposé, on va assimiler les constantes de mini-Act et leurs valeurs après évaluation.
- op représente une primitive. L'ensemble de celles-ci sera noté $\mathbb{P}$ et il contiendra les opérateurs classiques (+, fst...). Le calcul de l'appel d'une primitive se fera via l'opérateur «*compute*» qui encapsulera l'appel de la bonne fonction en librairie et renverra soit son résultat, soit une valeur particulière qui représentera une erreur lors de l'exécution. Parmi les erreurs possibles, on peut signaler que l'on impose l'application totale des primitives.
- $\mathcal{C}$ est un constructeur. Nous supposons disposer d'un ensemble de constructeurs, noté $\mathbb{C}$, et nous ne nous intéresserons pas à la construction de cet ensemble. Pour décrire la sémantique, nous nous servirons de «*ar*» la fonction de $\mathbb{C}$ dans $\mathbb{N}$ qui associe un constructeur et son arité.
- m est une étiquette de message. L'ensemble des étiquettes de messages, noté $\mathbb{M}$, a été séparé de celui des variables, afin de faciliter l'écriture de la sémantique. On ne s'intéressera pas à la façon dont est construit cet ensemble.

Enfin, nous imposerons qu'une expression soit bien parenthésée, mais nous ne nous intéresserons pas à ce parenthésage. Nous ferons donc l'hypothèse qu'il n'y a pas de conflit de reconnaissance lors de l'évaluation d'une expression ou d'un filtre.

Figure 6.1 La syntaxe de mini-Act.

e::=x	variable
| c	constante
| $\mathcal{C}$(e,...,e)	application d'un constructeur
| [λp.e,...,λp.e]	abstraction sur une famille de filtres
| op(e,...,e)	application d'une primitive
| e e	application
| e;e	séquence
| let x = e in e	liaison lexicale
| letrec [x=e,...,x=e] in e	groupe de liaisons lexicales récursives
| send e to e	envoi de message
| m(e,...,e)	construction de message
| letactor [x=e,...,x=e] in e	création d'une famille d'acteurs
| become e	changement de comportement
| suicide	suicide d'un acteur
| x ← e	mise à jour d'un champs privé
| {val[x=e,...],valmut[x=e,...],mess[m=e,...]}	construction de comportement
p::=_	joker
| x	variable
| c	constante
| $\mathcal{C}$(p,...,p)	construction de filtres complexes

REMARQUE 6 *Le filtrage ne contient pas l'alternative car elle n'ajoute rien à l'expressivité du langage mais complique énormément les règles de filtrage.*

6.2 Le choix d'un formalisme

Dans les précédents chapitres, pour la présentation des étapes de traduction, ainsi que le typage de ML-ACT, mon choix s'était porté sur la sémantique naturelle. J'ai donc écrit des règles de sémantique naturelle dans une première étape de mon travail de formalisation de la sémantique de mini-Act. Ce type de sémantique donne une forme relativement intuitive et élégante de la sémantique d'un langage dans le cas où l'on ne traite pas les erreurs. En revanche, dans le cas d'une description complète, le nombre de règles croît de manière impressionnante. En effet, la version que j'ai réalisé comporte environ soixante dix règles d'inférence. Cette description exhaustive a donc le défaut de submerger le lecteur de règles.

Au-delà de ce défaut, la sémantique naturelle est une sémantique opérationnelle «à grand pas». Cette notion de «grand pas» signifie que l'on ne suit l'évaluation des expressions que d'étapes en étapes et que la transition entre ces différentes étapes est importante. Elle a donc la fâcheuse tendance à masquer certaines notions. En outre, elle ne met pas bien en évidence le parallélisme de l'évaluation des programmes écrits dans des langages concurrents ou le non-determinisme présent dans certains opérateurs de choix.

Pour remédier à ces deux points faibles, il a été décidé de changer de formalisme et d'utiliser une sémantique à base de réduction. En effet, celle-ci fournit une meilleure abstraction de la concurrence du fait des «petits pas» effectués lors de l'évaluation. L'évaluation d'un programme consistera donc, en une réduction (réécriture) de proche en proche. L'exécution se termine lorsqu'aucune réduction n'est possible. L'étude qu'a faite R.C.McDowell de la comparaison des différentes sémantiques opérationnelles dans [McD93], nous a conforté

dans notre choix.

La présentation des règles conserve l'apparence de dérivations logiques via des règles d'inférence. La principale différence réside dans la notion de contexte d'évaluation qui va guider l'exécution.

Afin d'alléger les règles, j'ai également décidé de supprimer la notion d'environnement et d'utiliser à sa place la notion de substitution. Dans cette optique, dès que l'on connaît la valeur d'une variable, on remplace toute occurrence de celle-ci par sa valeur, dans la portée de la définition.

6.3 Le contexte d'évaluation et les valeurs sémantiques.

6.3.1 Ensembles de noms.

Afin de simplifier l'exposé, nous allons utiliser des espaces de noms totalement séparés pour les objets des différentes familles. Ainsi, nous supposerons disposer de l'ensemble $\mathbb{A}$ des noms d'acteur. Les différents ensembles de noms vérifient la propriété :

$$\mathbb{V} \cap \mathbb{M} = \mathbb{V} \cap \mathbb{A} = \mathbb{M} \cap \mathbb{A} = \emptyset.$$

Et nous disposons de la fonction :«*newname*» qui renvoie un nouvel objet de $\mathbb{A}$ qui n'a jamais été fourni. En fait, le programmeur ne manipule pas directement les adresses, elles sont gérées de manière automatique lors de la réduction.

6.3.2 Contextes

La réduction de la partie fonctionnelle de mini-Act exige d'être effectuée dans le «bon» ordre. Pour cela, on utilise la notion de contexte d'évaluation qui va guider ces réductions selon la règle du «plus à gauche plus à l'extérieur» (ou appel par valeur). De plus, la sémantique choisie est basée sur l'appel par valeur.

Un contexte sera noté R ; il consiste en un programme contenant un et un seul «trou». Pour remplir ce «trou», on utilisera la notation $\mathrm{R}[e]$ pour signifier que celui-ci est remplacé par e. Cette notion de «trou» permet de connaître la prochaine expression à réduire. Les contextes seront les termes obtenus par la grammaire présentée dans la figure 6.2. Pour simplifier cette grammaire, la notation $\tilde{x}$ sera utilisée pour représenter une suite quelconque de x éventuellement vide.

6.3.3 Fonctions partielles.

Afin de présenter la sémantique de mini-Act, nous allons utiliser les notions classiques de fonctions partielles et de domaines. On assimilera une fonction f et l'ensemble des couples (x,f(x)) pour x$\in$ dom(f), on utilisera donc aussi bien la terminologie des fonctions que celle des ensembles. Par exemple, nous noterons les fonctions partielles en extension, par la description de chaque couple de valeurs possibles ($f = \{x_1 \mapsto f(x_1),..., x_n \mapsto f(x_n)\}$ si $\mathrm{dom}(f) = \{x_1,..., x_n\}$).

Nous allons également utiliser l'opérateur «::» qui est défini par :
- $\{x_1 \mapsto y_1,..., x_n \mapsto y_n\} :: \{x \mapsto y\} = \{x_1 \mapsto y_1,..., x_n \mapsto y_n, x \mapsto y\}$ si $x \notin \{x_1,..., x_n\}$
- $\{x_1 \mapsto y_1,..., x_n \mapsto y_n\} :: \{x_k \mapsto y\} = \{x_1 \mapsto y_1,..., x_k \mapsto y,..., x_n \mapsto y_n\}$

Ce qui signifie que lors de l'ajout d'un couple dont le premier élément est déjà associé à une valeur, on remplace cette liaison.

Figure 6.2 La grammaire des contextes.

<table>
<tr><td>r::=[]</td><td>«trou»</td></tr>
<tr><td>| $\mathcal{C}(\widetilde{v}$,r,$\widetilde{e})$</td><td>calcul des paramètres d'un constructeur</td></tr>
<tr><td>| op($\widetilde{v}$,r,$\widetilde{e}$)</td><td>calcul des paramètres d'une primitive</td></tr>
<tr><td>| r e</td><td>calcul de la fonction</td></tr>
<tr><td>| v r</td><td>calcul du paramètre</td></tr>
<tr><td>| r;e</td><td>calcul en séquence</td></tr>
<tr><td>| let x=r in e</td><td>calcul de la valeur associée au nom</td></tr>
<tr><td>| send r to e</td><td>construction du message</td></tr>
<tr><td>| send v to r</td><td>calcul du destinataire</td></tr>
<tr><td>| m($\widetilde{v}$,r,$\widetilde{e}$)</td><td>calcul des paramètres d'un message</td></tr>
<tr><td>| new(a,r)</td><td>initialisation d'un acteur</td></tr>
<tr><td>| become r</td><td>calcul du nouveau comportement</td></tr>
<tr><td>| $x \leftarrow$r</td><td>calcul de la nouvelle valeur d'un champs privé</td></tr>
<tr><td>| {val[$\widetilde{x=v}$,x=r,$\widetilde{x=e}$], valmut[$\widetilde{x=e}$],mess[$\widetilde{m=e}$]}</td><td>calcul des valeurs de champs privés non-modifiables</td></tr>
<tr><td>| {val[$\widetilde{x=v}$],valmut[$\widetilde{x=v}$,x= r,$\widetilde{x=e}$],mess[$\widetilde{m=e}$]}</td><td>calcul des valeurs de champs privés modifiables</td></tr>
</table>

6.3.4 Substitution.

Une substitution sera une fonction partielle de l'ensemble des variables dans celui composé des valeurs sémantiques et des expressions. En général, elle sera noté ψ ($\psi : \mathbb{V} \rightarrow \mathcal{V}_s \cup \mathcal{E}xp$ avec $\mathcal{V}_s$ ensemble des valeurs sémantiques). L'écriture également employée sera : $[{}^{\mathrm{e}}/_{\mathrm{x}}]$ pour signifier que la substitution remplacera toutes les occurrences de x, qui sont dans sa portée, par e. Nous utiliserons la notation *id* pour désigner la substitution qui ne change rien (identité). On peut formaliser l'action d'une substitution par les règles de la figure 6.3. Dans le cas d'une abstraction, la fonction BV présentée ci-dessous est utilisée pour calculer l'ensemble des variables figurant dans un filtre (ie l'ensemble des variables liées par le filtre).

$$\mathrm{BV}(_) = \emptyset \qquad \mathrm{BV}(x) = \{x\} \qquad \mathrm{BV}(c) = \emptyset \qquad \mathrm{BV}(\mathcal{C}(p_1,...)) = \bigcup_i \mathrm{BV}(p_i)$$

6.3.5 Valeurs sémantiques.

En supplément des ensembles et fonctions décrits précédemment, nous définissons également les fermetures de comportement qui contiennent la mémoire initiale associée à ce comportement, ainsi que les réactions de celui-ci. Il sera noté $\mathbb{B}$, et ses éléments $\langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle$, où $\mathcal{M}$ et $\mathcal{R}$ sont les fonctions partielles suivantes :

- $\mathcal{M} : \mathbb{V} \rightarrow \mathcal{V}_s$, une mémoire pour les acteurs. Celle-ci contiendra les objets modifiables en place définis dans le comportement courant de l'acteur.

- $\mathcal{R} : \mathbb{M} \rightarrow \mathcal{E}xp$, un environnement pour conserver la réaction associée à une étiquette de message dans un comportement.

L'abstraction fonctionnelle est considérée comme une valeur. L'ensemble des valeurs sémantiques est donc l'ensemble des termes construits par la grammaire décrite par la figure 6.4.

Figure 6.3 L'application d'une substitution ψ à une expression.

$[y/x](x) = y \qquad [y/x](z) = z \text{ si } z \neq x \qquad [y/x](c) = c \text{ si } c \in \mathfrak{C}$

$[y/x](\mathcal{C}(e,...)) = \mathcal{C}([y/x](e),...) \qquad [y/x](op(e,...)) = op([y/x](e),...)$

$[y/x](\mathrm{m}(e,...)) = \mathrm{m}([y/x](e),...) \qquad [y/x](e_1\ e_2) = [y/x](e_1)\ [y/x](e_2)$

$[y/x](e_1; e_2) = [y/x](e_1); [y/x](e_2) \qquad [y/x](\text{send } e_1 \text{ to } e_2) = \text{send } [y/x](e_1) \text{ to } [y/x](e_2)$

$[y/x](\text{become } e) = \text{become } [y/x](e) \qquad [y/x](\text{suicide}) = \text{suicide} \qquad [y/x](z \leftarrow e) = z \leftarrow [y/x](e)$

$$[y/x]([\lambda p.e,...]) = \begin{cases} [\lambda p.e,...] & \text{si } x \in \mathrm{BV}(p) \\ [\lambda p.\ [y/x](e),...] & \text{si } x \notin \mathrm{BV}(p) \end{cases}$$

$$[y/x](\text{let } z = e_1 \text{ in } e_2) = \begin{cases} \text{let } z = [y/x](e_1) \text{ in } e_2 & \text{si } x = z \\ \text{let } z = [y/x](e_1) \text{ in } [y/x](e_2) & \text{si } x \neq z \end{cases}$$

$$[y/x](\text{letrec}[x_1 = e_1,...] \text{ in } e) = \begin{cases} \text{letrec}[x_1 = e_1,...] \text{ in } e & \text{si } \exists i\ x_i = x \\ \text{letrec}[x_1 = [y/x](e_1),...] \text{ in } [y/x](e) & \text{si } \forall i\ x_i \neq x \end{cases}$$

$$[y/x](\text{letactor}[x_1 = e_1,...] \text{ in } e) = \begin{cases} \text{letactor}[x_1 = e_1,...] \text{ in } e & \text{si } \exists i\ x_i = x \\ \text{letactor}[x_1 = [y/x](e_1),...] \text{ in } [y/x](e) & \text{si } \forall i\ x_i \neq x \end{cases}$$

$$[y/x](\{\text{val}[x_1 = e_1,...], \text{valmut}[x_p = e_p,...], \text{mess}[m_1 = e'_1,...]\}) =$$
$$\begin{cases} \{\text{val}[x_1 = [y/x](e_1),...], \text{valmut}[x_p = [y/x](e_p),...], \text{mess}[m_1 = e'_1,...]\} & \text{si } \exists i\ x_i = x \\ \{\text{val}[x_1 = [y/x](e_1),...], \text{valmut}[x_p = [y/x](e_p),...], \text{mess}[m_1 = [y/x](e'_1),...]\} & \text{si } \forall i\ x_i \neq x \end{cases}$$

Figure 6.4 Les valeurs sémantiques.

$\mathrm{v}_s ::= \mathrm{c}$	constante
$\mid [\lambda p_1, e_1, ..., \lambda p_n, e_n]$	abstraction fonctionnelle
$\mid \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle$	fermeture de comportement
$\mid \mathrm{a}$	adresse d'un acteur
$\mid \mathcal{C}(\mathrm{v}_s, ..., \mathrm{v}_s)$	terme construit
$\mid \mathrm{m}(\mathrm{v}_s, ..., \mathrm{v}_s)$	message construit

6.3.6 Les configurations.

Afin de modéliser le fait qu'un programme d'acteur est constitué de nombreuses entités s'exécutant en parallèle, nous allons utiliser un objet : la *configuration*. Celle-ci sera noté w, et sera construite par la grammaire de la figure 6.5.

L'écriture de l'évaluation fonctionnelle a la signification suivante :

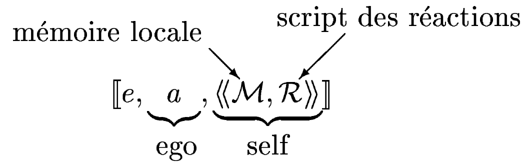


Figure 6.5 Les acteurs et leurs communications.

$w ::= \epsilon$	configuration vide
$\mid w \parallel w$	mise en parallèle de configuration
$\mid a \triangleleft \mathrm{m}(v_1,..., v_n)$	envoi de message
$\mid a \triangleright \langle\langle \mathcal{M}, \mathcal{R} \rangle\rangle$	comportement courant d'un acteur
$\mid [\![e, a, \langle\langle \mathcal{M}, \mathcal{R} \rangle\rangle]\!]$	évaluation fonctionnelle dans un acteur
$\mid$ **error**	erreur lors de l'évaluation

REMARQUE 7 *On va considérer qu'un acteur est initialement créé pour exécuter le code du programme. C'est l'exécution de celui-ci qui va provoquer des créations d'acteurs et des envois de messages. La configuration deviendra, alors, une exécution en parallèle de divers acteurs.*

6.4 La sémantique dynamique de mini-Act.

Nous allons débuter par l'exposé des règles de filtrage des termes. Puis, nous présenterons la réduction des configurations.

6.4.1 L'opérateur de filtrage.

Nous allons nous intéresser aux règles concernant le filtrage des expressions. Dans le cadre de notre langage l'évaluation étant de type appel par valeur, nous utilisons donc un filtrage strict (par opposition à paresseux). Celui-ci s'applique donc uniquement sur les valeurs sémantiques. Pour cela, nous allons définir l'opérateur «/» qui filtrera la valeur passée en premier paramètre par le filtre fournit en second paramètre. Le résultat de cette application sera soit un échec, soit une erreur, soit un environnement contenant toutes les liaisons lexicales des variables du filtre. Donc, le profil de cet opérateur est :

$_/_ : \mathcal{V}_s \times \mathcal{F}il \rightarrow \mathcal{P}(\mathcal{S}) \cup \{\mathbf{failed}, \mathbf{error}\}$

Pour l'ensemble des règles présentées dans la suite de cette section, nous noterons M,M_1,... les éléments de $\mathcal{V}_s$ et p,p_1,... les filtres. De plus, nous représenterons le résultat possible de l'application de «/» par ρ,ρ_1,...(il s'agira donc soit d'une substitution, soit d'un échec du filtrage, soit d'une erreur). L'opérateur d'union disjointe ($\bigsqcup$) sera utilisé ; son résultat est la substitution obtenue par composition si les substitutions, sur lesquelles il est appliqué, ont des domaines deux à deux disjoints. Dans le cas contraire, on suppose que son résultat est **error**.

Le filtrage d'un joker réussit toujours et alors aucune variable n'est substituée (*fil_1*). Celui d'une variable est également toujours un succès, mais fourni une substitution remplaçant celle-ci par la valeur filtrée (*fil_2*). Le filtrage d'une constante impose que la valeur soit égale à la constante, sinon, il y a échec de celui-ci (*fil_4*). En cas de succès, aucun remplacement n'est effectué (*fil_3*).

$$\text{FIL_1}: \frac{}{M/_ \Rightarrow id} \qquad \text{FIL_2}: \frac{}{M/x \Rightarrow [\mathrm{M}/x]} \qquad \text{FIL_3}: \frac{}{c/c \Rightarrow id} \qquad \text{FIL_4}: \frac{M \neq c}{M/c \Rightarrow \mathbf{failed}}$$

Le filtrage d'un filtre complexe - ie :contenant un constructeur d'arité non nulle- échoue si la valeur n'est pas un terme construit avec le même constructeur (*fil_5*) ou si, lors du filtrage des sous-valeurs (aucune erreur ne se produisant), un échec se produit (*fil_7*). Les cas d'erreur sont : soit une erreur s'est produite pendant le filtrage des sous-valeurs (*fil_6*), soit les filtres utilisés ne sont pas linéaires - ie :une même variable est utilisée dans au moins deux sous-filtres différents- (*fil_8*). Enfin, en cas de succès, on construit la substitution résultat par composition disjointe de toutes les substitutions issues des sous-filtrages (*fil_9*).

REMARQUE 8 *Dans les règles* (fil_8) *et* (fil_9), *le résultat de filtrage est noté* ψ *afin d'insister sur le fait que ce sont des substitutions et non une erreur ou un échec.*

$$\text{FIL_5}: \frac{\forall M_i\ M \neq \mathcal{C}(M_1,...,M_n)}{M/\mathcal{C}(p_1,...,p_n) \Rightarrow \mathbf{failed}} \qquad \text{FIL_6}: \frac{M_i/p_i \Rightarrow \rho_i \quad \exists i\ \rho_i = \mathbf{error}}{\mathcal{C}(M_1,...,M_n)/\mathcal{C}(p_1,...,p_n) \Rightarrow \mathbf{error}}$$

$$\text{FIL_7}: \frac{M_i/p_i \Rightarrow \rho_i \quad \exists i\ \rho_i = \mathbf{failed} \quad \forall i\ \rho_i \neq \mathbf{error}}{\mathcal{C}(M_1,...,M_n)/\mathcal{C}(p_1,...,p_n) \Rightarrow \mathbf{failed}}$$

$$\text{FIL_8}: \frac{M_i/p_i \Rightarrow \psi_i \quad \bigsqcup_{i=1}^{n} \psi_i = \mathbf{error}}{\mathcal{C}(M_1,...,M_n)/\mathcal{C}(p_1,...,p_n) \Rightarrow \mathbf{error}} \qquad \text{FIL_9}: \frac{M_i/p_i \Rightarrow \psi_i \quad \bigsqcup_{i=1}^{n} \psi_i = \psi}{\mathcal{C}(M_1,...,M_n)/\mathcal{C}(p_1,...,p_n) \Rightarrow \psi}$$

6.4.2 Les règles de réduction des configurations.

La signification des règles de réduction de la figure est la suivante : $w_1 \to w_2$, pour w_1 se réduit en w_2 en une étape de calcul. On utilisera dans la plupart des règles la notion de contexte présentée dans la section 6.3.2.

Dans toutes les règles de réduction qui suivent, on a adopté la sémantique suivante : lorsqu'une erreur se produit dans une évaluation fonctionnelle, on se contente de faire disparaître l'acteur concerné.

$$\text{NEUT}: \frac{}{w \,\|\, \epsilon \to w} \qquad \text{COMM}: \frac{w_2 \,\|\, w_1 \to w}{w_1 \,\|\, w_2 \to w} \qquad \text{PART}: \frac{w_1 \to w_2}{w \,\|\, w_1 \to w \,\|\, w_2}$$

L'opération de composition parallèle -$\|$- est commutative (*Comm*) et possède un élément neutre -ϵ- (*Neut*). De plus, on peut réduire les différentes composantes en parallèles séparément.

$$\text{REA}: \frac{m \in \text{dom}(\mathcal{R})}{a \triangleright \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle \,||\, a \triangleleft \text{m}(v_1,...,v_n) \rightarrow [\![((\mathcal{R}(m)\, v_1)...v_n), a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

La seule réduction non fonctionnelle de configuration est l'acceptation d'un message par un acteur (*Rea*). Nous n'imposons rien sur les arguments du message à ce stade. En effet, si ils sont trop nombreux ou bien de mauvais type une erreur fonctionnelle se produira, et si ils ne sont pas assez nombreux le résultat de l'évaluation fonctionnelle se terminera par une abstraction fonctionnelle. Or, nous verrons que l'on va interdire cette possibilité.

L'acceptation d'un message par un acteur provoque la disparition de celui-ci et du message de la configuration. Il se transforme en une évaluation fonctionnelle qui va exécuter la réaction de l'acteur à ce message. Notons qu'avec les règles décrites ci-dessus, il est possible de traiter les messages dans tous les ordres possibles.

$$\text{END}: \frac{v \in \mathcal{V}_s}{[\![(), a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow a \triangleright \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle}$$

Lorsque l'exécution fonctionnelle d'un acteur se termine, on réengendre cet acteur en position de lecture d'un message dans sa boite aux lettres. On va considérer que l'exécution d'un acteur doit se terminer par un «unit». Cela permet par exemple de détecter l'erreur signaler ci-dessus, de mauvaise réception d'un message.

$$\text{VAR-ERROR}: \frac{x \notin \text{dom}(\mathcal{M})}{[\![\text{R}[x], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon} \qquad \text{VAR}: \frac{x \in \text{dom}(\mathcal{M})}{[\![\text{R}[x], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\text{R}[\mathcal{M}(x)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

Lorsqu'on rencontre un identificateur qui n'est pas en mémoire (*Var-error*), c'est une erreur. En effet, les seules variables qui ne sont pas substituées sont les champs privés modifiables qui sont gérés par la mémoire. Dans le cas contraire, on le remplace par sa valeur actuelle en mémoire (*Var*).

$$\text{CONST-ERROR}: \frac{ar(\mathcal{C}) \neq n}{[\![\text{R}[\mathcal{C}(e_1,...,e_n)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon}$$

$$\text{PRIM-ERROR}: \frac{\forall i\ v_i \in \mathcal{V}_s \quad compute(op, (v_1,...,v_n)) = \textbf{error}}{[\![\text{R}[\text{op}(v_1,...,v_n)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon}$$

$$\text{PRIM}: \frac{\forall i\ v_i \in \mathcal{V}_s \quad compute(op, (v_1,...,v_n)) = v}{[\![\text{R}[\text{op}(v_1,...,v_n)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\text{R}[v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

L'application d'un constructeur avec un mauvais nombre de paramètres provoque une erreur (*Const-error*). C'est aussi le cas pour un calcul de primitive qui se termine mal (*Prim-error*). Sinon, on ne fait que remplacer l'application de la primitive par le résultat du calcul (*Prim*).

$$\text{SEQ}: \frac{v \in \mathcal{V}_s}{[\![\text{R}[v; e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\text{R}[e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

Lorsqu'on a évalué la première partie d'une séquence, on oublie son résultat puis on évalue sa seconde partie.

$$\textsc{Apply-error}: \frac{v \neq [\lambda p_1.e_1,..., \lambda p_n.e_n]}{[\![\mathrm{R}[v\ e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon} \qquad \textsc{Pm-failed}: \frac{v \in \mathcal{V}_s \quad v/p \Rightarrow \textbf{failed}}{[\![\mathrm{R}[[\lambda p.e]\ v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon}$$

$$\textsc{Pm-error}: \frac{v \in \mathcal{V}_s \quad v/p_1 \Rightarrow \textbf{error}}{[\![\mathrm{R}[[\lambda p_1.e_1,..., \lambda p_n.e_n]\ v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon}$$

$$\textsc{Pm-next}: \frac{v \in \mathcal{V}_s \quad n > 2 \quad v/p_1 \Rightarrow \textbf{failed}}{[\![\mathrm{R}[[\lambda p_1.e_1,..., \lambda p_n.e_n]\ v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\mathrm{R}[[\lambda p_2.e_2,..., \lambda p_n.e_n]\ v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

$$\textsc{Pm}: \frac{v \in \mathcal{V}_s \quad v/p_1 \Rightarrow \psi}{[\![\mathrm{R}[[\lambda p_1.e_1,..., \lambda p_n.e_n]\ v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\mathrm{R}[\psi(e_1)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

L'application d'une valeur qui n'est pas une fonction provoque une erreur (*Apply-error*). Lorsque c'est une fonction qui ne comporte qu'un cas de filtrage, et que le filtrage de la valeur sur laquelle s'applique la fonction échoue, on obtient également une erreur (*Pm-failed*). S'il y a plusieurs possibilité de filtrage, on passe à la suivante (*Pm-next*). Si, quelque soit le nombre de cas de filtrage, une erreur de filtrage se produit, le résultat de l'application est considéré comme une erreur (*Pm-error*). Enfin, si le filtrage réussit, on substitue les variables du filtre par leurs valeurs dans le corps de la branche choisie (*Pm*).

$$\textsc{Let}: \frac{v \in \mathcal{V}_s \quad \psi = [v/x]}{[\![\mathrm{R}[\mathrm{let}\ x = v\ \mathrm{in}\ e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\mathrm{R}[\psi(e)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

$$\textsc{Letrec}: \frac{\forall i \neq j\ x_i \neq x_j \quad \psi = [\psi(e_1)/x_1] ... [\psi(e_n)/x_n]}{[\![\mathrm{R}[\mathrm{letrec}[x_1 = e_1,..., x_n = e_n]\ \mathrm{in}\ e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\mathrm{R}[\psi(e)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

$$\textsc{Letrec}: \frac{\exists i \neq j\ x_i = x_j}{[\![\mathrm{R}[\mathrm{letrec}[x_1 = e_1,..., x_n = e_n]\ \mathrm{in}\ e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon}$$

L'action des liaisons lexicales est simple, il suffit de remplacer, dans leur corps, les noms qu'elles définissent par leurs valeurs (*Let*). Dans le cas d'une liaison récursive, le calcul de la valeur de chaque nom impose le calcul d'un point fixe. Dans la règle (*Letrec*), ce calcul de point fixe apparaît sous la forme d'une définition de substitution qui est elle-même récursive. Deplus, on impose aux noms d'être tous différents.

$$\textsc{Send-error-1}: \frac{\forall m\ v \neq \mathrm{m}(v_1,..., v_n)}{[\![\mathrm{R}[\mathrm{send}\ v\ \mathrm{to}\ e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon}$$

$$\textsc{Send-error-2}: \frac{\forall i\ v_i \in \mathcal{V}_s \quad v \notin \mathbb{A}}{[\![\mathrm{R}[\mathrm{send}\ \mathrm{m}(v_1,..., v_n)\ \mathrm{to}\ v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow \epsilon}$$

$$\textsc{Send}: \frac{\forall i\ v_i \in \mathcal{V}_s \quad a \in \mathbb{A}}{[\![\mathrm{R}[\mathrm{send}\ \mathrm{m}(v_1,..., v_n)\ \mathrm{to}\ a], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \rightarrow [\![\mathrm{R}[()], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \parallel a \triangleleft \mathrm{m}(v_1,..., v_n)}$$

Pour qu'un envoi de message soit correct il faut :

- que la première valeur soit un message (cas d'erreur (*Send-error-1*)).
- que la seconde soit une adresse d'acteur (cas d'erreur (*Send-error-2*)).

Lorsque ces deux conditions sont remplies, on ajoute le message dans la configuration, et on poursuit le calcul avec pour résultat de l'envoi, «unit».

$$\textsc{Become-error}: \frac{\forall \mathcal{M}\, \forall \mathcal{R}\ v \neq \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle}{[\![\mathrm{R}[\text{become } v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \epsilon}$$

$$\textsc{Become}: \frac{a' = newname}{[\![\mathrm{R}[\text{become } \langle\!\langle \mathcal{M}_1, \mathcal{R}_1 \rangle\!\rangle], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to a \triangleright \langle\!\langle \mathcal{M}_1, \mathcal{R}_1 \rangle\!\rangle \,||\, [\![\mathrm{R}[()], a', \langle\!\langle \mathcal{M}, \{\} \rangle\!\rangle]\!]}$$

$$\textsc{Suicide}: \frac{a' = newname}{[\![\mathrm{R}[\text{suicide}], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \epsilon}$$

Le changement d'état appliqué sur autre chose qu'une fermeture de comportement provoque une erreur (*Become-error*). S'il n'y a pas erreur (*Become*), on crée un acteur avec notre adresse courante et ce nouveau comportement, l'évaluation courante changeant d'adresse pour prendre une adresse anonyme. Il change également d'interface, celle-ci devient vide. Remarquons qu'ainsi, il n'y a pas besoin d'imposer un seul changement d'état car personne ne connaissant cet acteur, même s'il change d'état, il ne pourra pas devenir actif. Si un acteur se suicide, son calcul s'arrête (*Suicide*).

REMARQUE 9 *Nous nous sommes placés dans le cas de l'existence d'un ramasse miettes capable de déterminer si un acteur peut encore recevoir des messages et qui, dans le cas contraire, le détruit. On pourrait également, dans la règle* (Become), *remplacer le résultat du calcul par «R[()];suicide» afin de forcer la mort de l'acteur anonyme qui termine l'évaluation.*

$$\textsc{Set-error}: \frac{x \notin \mathrm{dom}(\mathcal{M})}{[\![\mathrm{R}[x \leftarrow e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \epsilon}$$

$$\textsc{Set}: \frac{x \in \mathrm{dom}(\mathcal{M}) \quad v \in \mathcal{V}_s}{[\![\mathrm{R}[x \leftarrow v], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to [\![\mathrm{R}[()], a, \langle\!\langle \mathcal{M} :: \{x \mapsto v\}, \mathcal{R} \rangle\!\rangle]\!]}$$

La mise à jour d'un champ privé, consiste en une erreur si la variable ne figure pas en mémoire (les autres ont été substituées) (*Set-error*), et en un changement de la valeur qui lui est associé dans le cas contraire (*Set*).

$$\textsc{Beh-error}: \frac{(\exists i \neq j\ x_i = x_j) \ \vee\ (\exists i \neq j\ m_i = m_j)}{[\![\mathrm{R}[\{\mathrm{val}[x_1 = e_1, \ldots]\ , \mathrm{valmut}[x_p = e_p, \ldots]\ , \mathrm{mess}[m_1 = e'_1, \ldots]\ \}], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \epsilon}$$

$$\textsc{Beh}: \frac{\forall i \neq j\ ,\ x_i \neq x_j\ m_i \neq m_j \quad \forall i\ v_i \in \mathcal{V}_s \quad \psi = [v_1/x_1] \ldots [v_{p-1}/x_{p-1}]}{\begin{array}{c}[\![\mathrm{R}[\{\mathrm{val}[x_1 = v_1, \ldots]\ , \mathrm{valmut}[x_p = v_p, \ldots]\ , \mathrm{mess}[m_1 = e'_1, \ldots]\ \}], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \\ [\![\mathrm{R}[\langle\!\langle \{x_p \mapsto v_p\} :: \ldots, \{m_1 \mapsto \psi(e'_1)\} :: \ldots \rangle\!\rangle], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]\end{array}}$$

Un comportement doit avoir une interface «linéaire», c'est-à-dire que celui-ci ne doit pas définir deux champs de même nom, et ne doit pas non plus spécifier deux réactions à un même message (*Beh-error*). Si cette condition est remplie, on construit la fermeture comportementale correspondant à cette construction. Pour cela, on substitue les valeurs figées dans les corps des réactions et on met la valeur de celles qui sont modifiables en mémoire.

$$\text{Actor-alloc-error}: \frac{\exists i \neq j \; x_i = x_j}{[\![\mathrm{R}[\mathrm{letactor}[x_1 = e_1, \ldots, x_n = e_n] \text{ in } e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \epsilon}$$

$$\text{Actor-alloc}: \frac{\forall i \neq j \; x_i \neq x_j \quad \forall i \; a_i = newname \quad \psi = [a_1/x_1] \ldots [a_n/x_n]}{\begin{array}{l}[\![\mathrm{R}[\mathrm{letactor}[x_1 = e_1, \ldots, x_n = e_n] \text{ in } e], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \\ \quad [\![\mathrm{R}[\mathrm{new}(a_1, \psi(e_1)); \ldots; \mathrm{new}(a_n, \psi(e_n)); \psi(e)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]\end{array}}$$

$$\text{Actor-init-error}: \frac{\forall \mathcal{M} \, \forall \mathcal{R} \; v \neq \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle}{[\![\mathrm{R}[\mathrm{new}(a', v)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to \epsilon}$$

$$\text{Actor-init}: \frac{}{[\![\mathrm{R}[\mathrm{new}(a', \langle\!\langle \mathcal{M}_1, \mathcal{R}_1 \rangle\!\rangle)], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!] \to a' \triangleright \langle\!\langle \mathcal{M}_1, \mathcal{R}_1 \rangle\!\rangle \parallel [\![\mathrm{R}[()], a, \langle\!\langle \mathcal{M}, \mathcal{R} \rangle\!\rangle]\!]}$$

La création d'acteur est séparée en deux étapes : l'allocation et l'initialisation. L'allocation consiste en la création de nouvelles adresses et ensuite se comporte comme une liaison lexicale en substituant ces adresses au noms auxquels elles sont liées. Le résultat de cette phase est la séquence de toutes les initialisations d'acteurs suivi de l'exécution du corps (*Actor-alloc*). La deuxième phase consiste en l'initialisation d'un acteur, celle-ci impose que l'on fournisse une fermeture comportementale (dans le cas contraire (*Actor-init-error*)), alors, on ajoute en parallèle cet acteur nouvellement créé.

6.5 Un exemple de réduction

Pour préciser l'évaluation pratique d'un programme mini-Act en utilisant la sémantique de réduction que l'on vient de présenter, nous allons traiter un petit programme exemple.

Soit deux cellules (case mémoire) dont le code est le suivant :

let cell = [λv.{valmut[mem = v],
                mess[*get* = [λc.send *set*(mem) to c] ;
                    set = [λv.mem← v]]}]
in letactor[a = cell 1;b = cell 2] in send *get*(b) to a;send *set*(3) to b

où l'on suppose que $\mathbb{M}$ contient au moins les deux valeurs *get* et *set*.

L'évaluation de ce programme consiste, en fait, en la réduction d'une configuration contenant uniquement une évaluation fonctionnelle dans un acteur anonyme ayant une mémoire et une réaction toutes deux vides.

$[\![code, _, \langle\!\langle \emptyset, \emptyset \rangle\!\rangle]\!]$

↓Let↓

$[\![letactor[a = [\,]\, 1; b = [\,]\, 2] in \ldots, _, \ldots]\!]$

Les deux «entre crochet» signifient que l'on a remplacé cell par sa valeur.

Supposons que les appel de newname nous renvoie A_0 et A_1, deux éléments de $\mathbb{A}$.

↓Actor-alloc↓

$[\![new(A_0, [\,]\, 1); new(A_1, [\,]\, 2); send\, get(A_1)\, to\, A_0; send\, set(3)\, to\, A_1, _, \ldots]\!]$

↓Pm↓

$[\![new(A_0, \{1\}); \ldots, _, \ldots]\!]$

La notation {1} signifie que l'on a remplacé v par 1 dans le corps du comportement cell.

↓Beh↓

$[\![new(A_0, \langle\!\langle \{mem \mapsto 1\}, \{get \mapsto ...; set \mapsto ...\}\rangle\!\rangle); ..., _, ...]\!]$

$\downarrow$ACTOR-INIT$\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel [\![new(A_1, \{2\}); ..., _, ...]\!]$

La notation ⟪ x ⟫ est employée pour modéliser la fermeture de comportement de cell dont la valeur associée à mem en mémoire vaut x.

Par un même enchaînement de réduction, on alloue aussi A_1.

$\downarrow$PM$\downarrow$

$\downarrow$BEH$\downarrow$

$\downarrow$ACTOR-INIT$\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \rhd \langle\!\langle 2 \rangle\!\rangle \parallel [\![send\, get(A_1)\, to\, A_0; ..., _, ...]\!]$

$\downarrow$SEND$\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \rhd \langle\!\langle 2 \rangle\!\rangle \parallel [\![(); ... \parallel A_0 \lhd get(A_1), _, ...]\!]$

A ce point de la réduction apparaît un certain parallélisme. En effet, l'évaluation fonctionnelle va continuer à se réduire en même temps que l'acteur A_0 *va traiter son message. Remarquons que depuis que plusieurs objets sont en parallèle, la règle de commutativité de* $\parallel$ *aurait pu être appliquée. Dans la suite, on n'indiquera pas les règles appliquées pour commuter ou pour faire disparaître un élément neutre.*

Pour rendre compte du fait que l'ordre de réduction que l'on choisi n'est plus qu'un des ordres possibles, nous changeons de notation pour les transitions.

$\downarrow\downarrow$REA$\downarrow\downarrow$ $\downarrow\downarrow$SEQ$\downarrow\downarrow$

$A_1 \rhd \langle\!\langle 2 \rangle\!\rangle \parallel [\![[\lambda c.send\, set(mem)\, to\, c] A_1, A_0, \langle\!\langle 1 \rangle\!\rangle]\!] \parallel [\![send\, set(3)\, to\, A_1, _, ...]\!]$

$\downarrow\downarrow$PM$\downarrow\downarrow$ $\downarrow\downarrow$SEND$\downarrow\downarrow$

$A_1 \rhd \langle\!\langle 2 \rangle\!\rangle \parallel [\![send\, set(mem)\, to\, A_1, A_0, \langle\!\langle 1 \rangle\!\rangle]\!] \parallel [\![(), _, ...]\!] \parallel A_1 \lhd set(3)$

$\downarrow\downarrow$VAR$\downarrow\downarrow$ $\downarrow\downarrow$END$\downarrow\downarrow$

$A_1 \rhd \langle\!\langle 2 \rangle\!\rangle \parallel A_1 \lhd set(3) \parallel [\![send\, set(1)\, to\, A_1, A_0, \langle\!\langle 1 \rangle\!\rangle]\!] \parallel _ \rhd \langle\!\langle \emptyset, \emptyset \rangle\!\rangle$

L'acteur anonyme qui est dans la configuration est supposé être détruit par un ramasse miette.

$\downarrow\downarrow$SEND$\downarrow\downarrow$

$A_1 \rhd \langle\!\langle 2 \rangle\!\rangle \parallel A_1 \lhd set(3) \parallel [\![(), A_0, \langle\!\langle 1 \rangle\!\rangle]\!] \parallel A_1 \lhd set(1)$

$\downarrow\downarrow$END$\downarrow\downarrow$

$A_1 \rhd \langle\!\langle 2 \rangle\!\rangle \parallel A_1 \lhd set(3) \parallel A_1 \lhd set(1) \parallel A_0 \rhd \langle\!\langle 1 \rangle\!\rangle$

On va supposer que l'acteur A_1 *reçoit d'abord le message* $set(1)$.

$\downarrow\downarrow$REA$\downarrow\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \lhd set(3) \parallel [\![[\lambda v.mem \leftarrow v] 1, A_1, \langle\!\langle 2 \rangle\!\rangle]\!]$

$\downarrow\downarrow$PM$\downarrow\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \lhd set(3) \parallel [\![mem \leftarrow 1, A_1, \langle\!\langle 2 \rangle\!\rangle]\!]$

$\downarrow\downarrow$SET$\downarrow\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \lhd set(3) \parallel [\![(), A_1, \langle\!\langle 1 \rangle\!\rangle]\!]$

$\downarrow\downarrow$END$\downarrow\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \lhd set(3) \parallel A_1 \rhd \langle\!\langle 1 \rangle\!\rangle$

Par un enchaînement identique, on traite le message $set(3)$.

$\downarrow\downarrow$REA$\downarrow\downarrow$

$\downarrow\downarrow$PM$\downarrow\downarrow$

$\downarrow\downarrow$SET$\downarrow\downarrow$

$\downarrow\downarrow$END$\downarrow\downarrow$

$A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \rhd \langle\!\langle 3 \rangle\!\rangle$

Remarquons que c'est le choix que l'on a fait sur l'ordre de réception des messages qui nous mène à ce résultat. Si l'ordre de réception des deux messages est inversé, le résultat devient : $A_0 \rhd \langle\!\langle 1 \rangle\!\rangle \parallel A_1 \rhd \langle\!\langle 1 \rangle\!\rangle$

Cet exemple ne montre pas l'utilisation de toutes les règles, mais il éclaire l'explication de la sémantique opérationnelle de mini-Act.

6.6 Conclusion

La sémantique formalisée de ce chapitre n'est pas tout a fait celle implantée actuellement. Mais, dans le cas d'une implantation plus sérieuse, elle pourrait servir pour construire une machine abstraite qui exécuterait un programme ML-ACT.

Cette sémantique devrait être retouchée dans le cadre des différentes extensions que l'on souhaite faire de ML-ACT. Il faudra y intégrer les notions de synchronisations, groupes, modules, exceptions... Sa forme actuelle et sa taille raisonnable me laisse penser que ces extensions ne seront pas trop compliquées (sans être immédiates).

Chapitre 7

LE TYPAGE DE MINI-ACT.

Dans le chapitre précédent nous avons présenté un mini-langage et sa sémantique dynamique -ie son évaluation. Afin d'éliminer certaines erreurs pouvant intervenir durant l'exécution d'un programme écrit en mini-Act, nous allons lui donner une sémantique statique -ie un système de type. En effet, pour éliminer autant d'erreurs que possible, il s'agit de simuler l'exécution du programme. Nous allons donc approximer chaque entités par son type. Ce système de type formel est basé sur les travaux présentés dans les chapitres deux et trois, et concernant ML-ACT. Nous allons construire un système de type à la ML, c'est-à-dire un système qui permet d'inférer des types pour tous les objets manipulés. La différence par rapport aux principales implantations de ML est le fait que pour la reconstruction des types nous n'utiliserons pas une unification «à la volée». En effet, dans le but de garder un système plus flexible, nous utiliserons la technique déjà présentée de collecte de contraintes puis de résolution de celles-ci.

Lors de l'écriture de règles de typage, on ne donne pas les cas d'erreur. En effet, un système de type peut être vu comme une condition qu'un programme est «bien formé», et donc, que tout programme ne respectant pas les règles est «mal formé».

7.1 Les types.

Les types que nous utilisons sont les termes construits par la grammaire de la figure 7.1. Dans cette description, nous supposons disposer :

- D'un ensemble ($\mathbb{V}_t$) de variables de type, nous les noterons τ. Une fonction *newvar*, nous fournit une variable inutilisée.
- D'un ensemble de constructeurs de types isomorphes à celui des constructeurs de mini-Act. En fait, un constructeur de type est associé à chaque constructeur du langage. Par exemple, on dispose d'un constructeur pour les tuples, un constructeur pour les listes... Nous ne nous intéresserons pas à la façon dont cet ensemble est construit.
- Enfin, nous disposons d'un ensemble de constantes de types. On peut citer $\mathcal{I}nt$, $\mathcal{F}loat$, $\mathcal{C}har$, $\mathcal{B}ool$, ..., $\mathcal{U}nit$ auquel nous ajouterons pour approximer les phénomènes dus à la concurrence $\mathcal{B}eh$, $\mathcal{A}ddr$, $\mathcal{M}ess$ (qui représentent respectivement les comportements, les adresses et les messages).

Pour rendre compte du polymorphisme dans l'environnement qui conservera le type des variables, nous utiliserons, non pas des types, mais des schémas de type, c'est-à-dire,

Figure 7.1 Les types de mini-Act.

t::=τ	variable de type
$\mid$ $\mathcal{C}_t$(t,...,t)	constructeur de type
$\mid$ t $\rightarrow$ t	type fonctionnel
$\mid$ t $\times$ t	produit cartésien de types
$\mid$ c	constante de type

un couple composé d'un élément $\tilde{\alpha}$ de $\mathbb{V}_t^*$ et d'un type τ (on les notera : $\forall\tilde{\alpha}.\tau$). Ces couples signifient que dans le type τ, les variables de type présente dans le «vecteur» $\tilde{\alpha}$ sont généralisées -ie universellement quantifiées. Nous utiliserons alors les fonctions *Gen* et *Spe*, pour respectivement, passer d'un type au schéma de type lui correspondant (le «vecteur» sera composé des variables de type libres), et de passer d'un schéma de type à un des types lui correspondant (on remplace les variables présentes dans le «vecteur» par de nouvelles variables de type). Il est important de souligner que la généralisation des variables de type sera limitée aux liaisons lexicales («let» et «letrec»).

De plus, on utilisera la fonction «*typeof*» qui nous donnera le type d'une constante de mini-Act. Elle permettra également d'associer un constructeur ou une primitive et son type. En fait, cette fonction va utiliser la fonction de spécialisation, car les types des constantes, primitives et constructeurs sont décrits par des schémas de type (afin de les rendre polymorphes). Par exemple, si *Cons* et *Tuple* sont des constructeurs de mini-Act et *List* et $*$ leur constructeur de type, on a :

$$typeof(Cons) = \tau \times List(\tau) \rightarrow List(\tau)$$
$$typeof(Tuple) = \tau_1 \times ... \times \tau_n \rightarrow *(\tau_1,...,\tau_n)$$

où τ,τ_1,... sont des variables de type «fraîches».

Nous ne détaillerons pas la fonction *typeof* car cela ne présente que peu d'intérêt.

7.2 La sémantique statique de mini-Act.

Nous allons commencer l'exposé du typage par les règles concernant le filtrage. Puis, nous présenterons le typage des expressions. Pour typer les expressions, nous allons utiliser un environnement, qui associera variables de type et schémas de type. En plus des types déjà décrits, nous utiliserons, pour construire ces schémas, le constructeur particulier $mut(t)$ pour indiquer que la variable est modifiable (cette variable est donc un champ privé). Le fonctionnement des environnements sera identique à celui des mémoires du chapitre précédent.

7.2.1 Le filtrage.

La première partie du système de type va concerner les filtres et leur typage. Pour cela, analysons les erreurs dynamiques qui peuvent se produire lors de la construction de filtre. En fait, la seule erreur possible, avec notre sémantique, est la non-linéarité d'un filtre -ie une variable apparaît plusieurs fois dans un filtre. Le but du typage des filtres va donc être de supprimer ce type d'erreur. De plus, on construira un environnement qui permettra de conserver l'approximation de chaque variable du filtre (ie leur type). On étend également l'opérateur d'union disjointe défini sur les substitutions.

Un ensemble de contraintes sera un ensemble de couple de type, on notera chaque contrainte sous la forme : «t_1=t_2».

La forme des séquents utilisés comme jugements des règles de typage est donc :

$$\vdash \textit{le filtre à typer} : \textit{son type}, \textit{l'environnement}, \textit{l'ensemble des contraintes}$$

$$\text{ANY} : \frac{\tau = newvar}{\vdash _ : \tau, \{\}, \{\}} \qquad \text{VAR} : \frac{\tau = newvar}{\vdash x : \tau, \{x : \tau\}, \{\}} \qquad \text{CST} : \frac{\tau = typeof(c)}{\vdash c : \tau, \{\}, \{\}}$$

$$\text{CONST} : \frac{\vdash p_i : \tau_i, \mathcal{E}_i, C_i \quad typeof(\mathcal{C}) = t \quad \tau = newvar \quad \bigsqcup_{i=1}^{n} \mathcal{E}_i = \mathcal{E}}{\vdash \mathcal{C}(p_1,..., p_n) : \tau, \mathcal{E}, \bigcup_i C_i \cup \{\tau_1 \times ... \times \tau_n \to \tau = t\}}$$

Dans la règle de typage d'une variable, nous avons associé à x le type τ dans l'environnement résultat en identifiant τ et le schéma de type de «vecteur» vide et de type τ.

Notons que la seule règle contraignant les variables de type introduites dans les filtres est la règle (*Const*). Elle vérifie ainsi que le filtre respecte la construction de terme. La construction de l'environnement résultat utilise une union disjointe, et donc, impose que les environnements intermédiaires soient disjoints.

7.2.2 Les expressions.

Par rapport aux règles de typage des filtres, l'unique changement est la forme des séquents, elle devient :

$$\textit{environnement de typage} \vdash \textit{expression à typer} : \textit{son type}, \textit{l'ensemble des contraintes}$$

$$\text{VAR-MUT} : \frac{x \in \text{dom}(\mathcal{E}) \quad Spe(\mathcal{E}(x)) = mut(t)}{\mathcal{E} \vdash x : t, \{\}} \qquad \text{VAR} : \frac{x \in \text{dom}(\mathcal{E}) \quad \forall t\ Spe(\mathcal{E}(x)) \neq mut(t)}{\mathcal{E} \vdash x : Spe(\mathcal{E}(x)), \{\}}$$

$$\text{CST} : \frac{\tau = typeof(c)}{\mathcal{E} \vdash c : \tau, \{\}}$$

On distingue le traitement des variables selon le fait qu'elles soient modifiables ou non. Mais le résultat est similaire : pas de contraintes et le type qui figure dans l'environnement. Le typage d'une constante est simplement un appel à *typeof*.

$$\text{CONST} : \frac{\mathcal{E} \vdash e_i : \tau_i, C_i \quad typeof(\mathcal{C}) = t \quad \tau = newvar}{\mathcal{E} \vdash \mathcal{C}(e_1,..., e_n) : \tau, \bigcup_i C_i \cup \{\tau_1 \times ... \times \tau_n \to \tau = t\}}$$

$$\text{PRIM} : \frac{\mathcal{E} \vdash e_i : \tau_i, C_i \quad typeof(op) = t \quad \tau = newvar}{\mathcal{E} \vdash \text{op}(e_1,..., e_n) : \tau, \bigcup_i C_i \cup \{\tau_1 \times ... \times \tau_n \to \tau = t\}}$$

$$\text{MESS} : \frac{\mathcal{E} \vdash e_i : \tau_i, C_i}{\mathcal{E} \vdash \text{m}(e_1,..., e_n) : \mathcal{M}ess, \bigcup_i C_i}$$

Les typages d'une construction de terme et d'une application d'une primitive sont similaires et consistent en la production d'une contrainte entre le type obtenu par *typeof*

et celui construit à partir des paramètres. On suppose que les types des constructeurs et des primitives permettent de supprimer les évaluations incorrectes. Ces deux règles permettent donc de supprimer les cas d'erreur dans ces deux constructions. L'approximation, lors de la construction de message, est grossière ; tout message est de type $\mathcal{M}ess$.

$$\textsc{Fun}:\frac{\vdash p_i : \tau_i, \mathcal{E}_i, C_i \quad \mathcal{E} :: \mathcal{E}_i \vdash e_i : \tau_i', C_i' \quad \tau_{in} = newvar \quad \tau_{out} = newvar}{\mathcal{E} \vdash [\lambda p_1.e_1 ... \lambda p_n.e_n] : \tau_{in} \rightarrow \tau_{out}, \bigcup_i (C_i \cup C_i' \cup \{\tau_i = \tau_{in}\} \cup \{\tau_i' = \tau_{out}\})}$$

Durant le typage d'une fonction, on impose aux filtres d'une part, et aux expressions d'autre part, d'être tous de même type. Alors, le type de la fonction est le type fonctionnel dont le domaine est le type des filtres et le codomaine celui des expressions.

$$\textsc{App}:\frac{\mathcal{E} \vdash e_1 : \tau_1, C_1 \quad \mathcal{E} \vdash e_2 : \tau_2, C_2 \quad \tau_{in} = newvar \quad \tau_{out} = newvar}{\mathcal{E} \vdash e_1\ e_2 : \tau_{out}, C_1 \cup C_2 \cup \{\tau_1 = \tau_{in} \rightarrow \tau_{out}\} \cup \{\tau_2 = \tau_{in}\}}$$

$$\textsc{Seq}:\frac{\mathcal{E} \vdash e_1 : \tau_1, C_1 \quad \mathcal{E} \vdash e_2 : \tau_2, C_2}{\mathcal{E} \vdash e_1 ; e_2 : \tau_2, C_1 \cup C_2}$$

$$\textsc{Send}:\frac{\mathcal{E} \vdash e_1 : \tau_1, C_1 \quad \mathcal{E} \vdash e_2 : \tau_2, C_2}{\mathcal{E} \vdash \text{send } e_1 \text{ to } e_2 : \mathcal{U}nit, C_1 \cup C_2 \cup \{\tau_1 = \mathcal{M}ess\} \cup \{\tau_2 = \mathcal{A}ddr\}}$$

Classiquement, le traitement d'une application impose à la première partie d'être une fonction et à la seconde d'être compatible avec cette fonction. Cela supprime les erreurs d'applications durant l'évaluation. Pour typer une séquence, on se contente de vérifier que la première partie est «bien formée» et on retourne le résultat du typage de la seconde. L'envoi de message est une opération vide (qui renvoie $\mathcal{U}nit$) qui impose le «bon» type à ces composantes. On supprime ainsi les erreurs dynamiques d'envoi de message.

$$\textsc{Let}:\frac{\mathcal{E} \vdash e_1 : \tau_1, C_1 \quad \mathcal{E} :: \{x : Gen(\tau_1, C_1)\} \vdash e_2 : \tau_2, C_2}{\mathcal{E} \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2, C_2}$$

$$\textsc{Letrec}:\frac{\begin{array}{l}\forall i \neq j\ x_i \neq x_j\ \tau_i = newvar \\ \mathcal{E} :: \bigcup_i \{x_i : \tau_i\} \vdash e_i : \tau_i', C_i \qquad C_g = \bigcup_i (C_i \cup \{\tau_i' = \tau_i\}) \\ \mathcal{E} :: \bigcup_i \{x_i : Gen(\tau_i, C_g)\} \vdash e : \tau, C\end{array}}{\mathcal{E} \vdash \text{letrec}[x_1 = e_1, ..., x_n = e_n] \text{ in } e : \tau, C}$$

Les deux règles de liaisons lexicales ci-dessus consistent à typer leur corps dans l'environnement issu de l'ajout des nouvelles liaisons. De plus, on généralise les variables libres. Dans le traitement de la récursivité, l'opération de point fixe pour obtenir la valeur des types est contenu dans la contrainte $\tau = \tau'$. En effet, on obtient alors une équation de type f=F(f) dont la résolution se fait par un calcul de point fixe. Lors de l'introduction des nouveaux identificateurs dans l'environnement, on leur affecte un schéma de type $\forall.\tau$ que nous avons noté τ.

$$\textsc{Letactor} : \frac{\begin{array}{l}\forall i \neq j\ x_i \neq x_j \\ \mathcal{E} :: \bigcup_i \{x_i : \mathcal{A}ddr\} \vdash e_i : \tau_i, C_i \qquad \mathcal{E} :: \bigcup_i \{x_i : \mathcal{A}ddr\} \vdash e : \tau, C\end{array}}{\mathcal{E} \vdash \text{letactor}[x_1 = e_1, \ldots, x_n = e_n]\ \text{ in } e : \tau, \bigcup_i (C_i \cup \{\tau_i = \mathcal{B}eh\}) \cup C}$$

Pour l'écriture de la règle de création d'acteur, nous avons utilisé $\mathcal{A}ddr$ comme schéma associé aux nouveaux noms d'acteur, pour simplifier (on suppose le «vecteur» vide). On vérifie que les expressions associées aux noms sont des comportements fermés -ie auxquels on a fournit tous leurs arguments. Puis, on type le corps après avoir ajouté à l'environnement tous les identificateurs de noms d'acteur nouvellement définis.

$$\textsc{Become} : \frac{\text{ego} \in \text{dom}(\mathcal{E}) \quad \mathcal{E} \vdash e : \tau, C}{\mathcal{E} \vdash \text{become } e : \mathcal{U}nit, C \cup \{\tau = \mathcal{B}eh\}}$$

$$\textsc{Suicide} : \frac{\text{ego} \in \text{dom}(\mathcal{E})}{\mathcal{E} \vdash \text{suicide} : \mathcal{U}nit, \{\}}$$

$$\textsc{Set} : \frac{\text{ego} \in \text{dom}(\mathcal{E}) \quad x \in \text{dom}(\mathcal{E}) \quad Spe(\mathcal{E}(x)) = mut(t) \quad \mathcal{E} \vdash e : \tau, C}{\mathcal{E} \vdash x \leftarrow e : \mathcal{U}nit, C \cup \{\tau = t\}}$$

Lors de la construction de comportement, on met la variable ego dans l'environnement. Comme cette variable ne peut être définie que de cette manière, elle permet de savoir si on est bien dans une construction de comportement. Dans les trois règles (*Become*), (*Suicide*) et (*Set*), on teste la présence de ego dans l'environnement. On supprime donc l'erreur consistant à utiliser le changement de comportement, le suicide ou la mise à jour de champ privé en dehors d'un terme de formation de comportement. De plus, durant le changement de comportement, on impose que l'expression soit bien un comportement. On impose également à la variable que l'on souhaite affecter d'avoir le type *mut* (ie d'être modifiable).

$$\textsc{Beh} : \frac{\begin{array}{l}\forall (i \neq j \text{ or } k \neq j)\ x_i^k \neq x_j^l \quad m_i \neq m_j \quad \mathcal{E} \vdash e_i^1 : \tau_i^1, C_i^1 \quad \mathcal{E} \vdash e_i^2 : \tau_i^2, C_i^2 \\ \mathcal{E} :: \bigcup_i (\{x_i^1 : mut(\tau_i^1)\} \cup \{x_i^2 : \tau_i^2\}) :: \{ego : \mathcal{A}ddr\} \vdash e_i^3 : \tau_i^3, C_i^3 \\ \tau_i^4 = newvar \quad C = \bigcup_i (C_i^1 \cup C_i^2 \cup C_i^3 \cup \{\tau_i^3 = \tau_i^4 \rightarrow \mathcal{U}nit\})\end{array}}{\mathcal{E} \vdash \{\text{val}[x_1^1 = e_1^1, \ldots]\ , \text{valmut}[x_1^2 = e_1^2, \ldots]\ , \text{mess}[m_1 = e_1^3, \ldots]\ \} : \mathcal{B}eh, C}$$

La construction de comportement est la seule opération qui introduit ego dans l'environnement et qui définit des valeurs modifiables (ie de type *mut*). On impose à tous les champs privés d'avoir des noms différents et aux réactions d'être uniques. On évite ainsi les cas de construction non valable de comportement.

7.3 Commentaires

Plutôt que de donner une preuve de correction et de complétude du système de type par rapport à la sémantique dynamique de mini-Act, nous allons discuter informellement leur corrélation.

Pour commencer remarquons que dans la sémantique présentée au chapitre précédent, il n'y a pas d'erreur d'évaluation possible au sens strict. En effet, toute «erreur» consiste uniquement en un arrêt de l'acteur contenant l'évaluation dans laquelle cette erreur s'est produite. On pourrait donc penser qu'un système de type est inutile puisque tout programme s'exécute sans produire d'erreur. Cependant, on ne peut considérer pour autant qu'une exécution erronée est souhaitée par le programmeur. Dans ce but, on a construit le système de type présenté dans la section 7.1. Nous allons examiner une à une les erreurs possibles et la façon dont elles sont traitées par le système de type.

1. La construction d'un filtre qui contient une variable plus d'une fois provoque une erreur. L'union disjointe utilisée dans la règle (*Const*) permet d'éliminer les programmes ne respectant pas cette linéarité.

2. Une construction de filtre erronée n'est pas directement détectée, mais elle génère un filtre qui échouera systématiquement. Or, ce type de comportement n'est pas souhaité, on peut donc le considérer comme un cas d'erreur implicite. Celui-ci sera traité par le typage, puisque l'on vérifie dans la règle (*Const*) que le filtre respecte le type du constructeur.

3. La terminaison d'une évaluation fonctionnelle dans un acteur n'a lieu que lorsque sa valeur est $\mathcal{U}nit$ (*End*). La règle de typage de la formation de comportement (*Beh*) permet d'éliminer tout programme dont les réactions ne se termineraient pas ainsi.

4. La règle d'accès à une variable impose que celle-ci soit un champ privé défini ou une variable définie. Dans le deuxième cas, lors de sa définition, elle est substituée et ne peut donc être dans le programme. Ainsi, on vérifie bien durant le typage que toutes les variables sont bien définies et que les champs privés sont bien introduits dans la mémoire.

5. Le typage de la construction de terme par la règle (*Const*) permet de vérifier la validité de celle-ci. L'erreur traitée par la règle (*Const-error*) de la sémantique ne peut donc plus se produire.

6. Le même traitement a lieu sur les primitives dans la règle (*Prim*).

7. La règle de typage d'une application impose au premier terme d'être une fonction et évite l'apparition d'un terme utilisant la règle (*Apply-error*) pour s'évaluer.

8. Notre système de type ne comporte pas d'analyse précise du filtrage, et donc il ne permet pas de déterminer si un filtrage est complet. C'est à dire qu'il ne peut prévenir l'usage de la règle (*Pm-failed*).

9. Le typage des liaisons lexicales récursives impose aux noms nouvellement définis d'être tous différents et prévient donc l'utilisation de la règle (*Letrec-error*).

10. La contrainte de type générée lors du typage du premier paramètre d'un envoi de message permet de supprimer les programmes dont l'évaluation utilisait la règle (*Send-error-1*).

11. Il en va de même pour la vérification concernant le destinataire et donc la règle (*Send-error-2*).

12. La vérification de la présence de ego dans la mémoire impose aux changements de comportement, suicides et mises à jour de n'être utilisés que lors de la formation de comportement. Cela permet d'éliminer les programmes qui les contiennent dans des parties purement fonctionnelles. Et ceci, même si ce n'est pas un cas d'erreur explicite de la sémantique.

13. Le changement de comportement peut mal se dérouler via la règle (*Become-error*). Pour éviter cela le système de type vérifie que l'on fournit bien une fermeture comportementale lors d'un changement d'état.

14. Un programme bien typé ne pourra essayer de mettre à jour qu'un champ privé bien défini. La règle (*Set-error*) n'apparaîtra donc plus dans les évaluations.

15. Le typage de la construction comportement permet de prévenir l'apparition d'un comportement comportant deux noms identiques ou deux réactions à un même message. Et donc, on évite (*Beh-error*).

16. L'erreur due à l'emploi du même nom plus d'une fois dans la règle (*Actor-alloc-error*) ne peut se produire dans le cas d'un programme bien typé. En effet, la règle (*Letactor*) prévient de ces doublons.

17. Cette même règle vérifie également que l'expression qui représente le comportement est effectivement un comportement.

De façon plus générale, remarquons que l'approximation grossière faite sur les messages ne permet pas de détecter des constructions de message erronées. En effet, pour cela, il faudrait attribuer à chaque message un type contenant le type de ces arguments, et vérifier que chaque acteur à qui l'on envoi un message saura le traiter.

Or, cette analyse est complexe et fait partie des travaux de thèse de JL. Colaço [Col97], aussi nous ne nous y sommes pas intéressés. On peut remarquer également que lors du typage des liaisons lexicales récursives, on n'impose pas que les noms soient liés à des fonctions. Dans le cadre d'un langage de programmation, on impose que le calcul de point fixe se fasse uniquement sur des fonctions pour diminuer la probabilité qu'il ne se termine pas. Mais, formellement, étant incapable de vérifier si le calcul de point fixe se termine, même lorsqu'on le fait sur une fonction, on ne s'intéresse pas à cette limitation.

7.4 Conclusion

Le système de type construit reste classique en ce qui concerne les aspects fonctionnels de mini-Act. Dans le cadre du chapitre 2, on a cité comme possible extension, la transformation des contraintes d'égalité en contraintes d'inclusion. Les règles de typage présentées ici ne subiraient alors, que peu de modifications. La forme donnée leurs confère donc une grande flexibilité vis à vis du système de type. Celui-ci comporte actuellement deux «trous» qu'il est prévu de «boucher».

D'une part, l'approximation actuelle faite sur les messages ne nous permet pas de disposer d'informations suffisamment précises sur leur forme. Afin de remédier à ce défaut, il est prévu d'améliorer la précision du type d'un message et, à partir des travaux de JL.Colaço, d'effectuer une analyse beaucoup plus pertinente des paramètres de message. En fait, on cherche à atteindre deux buts: avoir la possibilité de polymorphisme sur les messages et disposer d'informations sur les messages qui ne pourront jamais être traités.

D'autre part, le système de filtrage tel qu'il est conçu ne permet déterminer ni les cas superflus, ni les filtrages incomplets ; tous deux souvent synonymes d'erreur du programmeur. Une extension du système de type actuel est donc envisagée afin de remédier à cela.

Enfin, signalons que dans le cadre d'une formalisation plus complète du langage, la preuve d'adéquation du système de type est également prévue.

Chapitre 8

CONCLUSION

Les recherches au sein de l'équipe sont essentiellement centrées sur l'analyse de programmes d'acteurs. Dans le cadre de ces travaux, j'ai été chargé de concevoir un langage fonctionnel d'acteurs.

Le langage développé, ML-ACT, est, dans sa version actuelle relativement simple et minimal. Mais ses analyseurs (lexical et syntaxique) ont été réalisé avec un souci d'extensibilité importante. Partant de ce noyau, la suite du projet consistera à en faire un langage relativement complet permettant la programmation d'applications intéressantes. Pour cela, il serait également avantageux d'étendre le champs des primitives de communication en y ajoutant par exemple des instructions de synchronisation. Ces extensions permettraient d'augmenter grandement l'expressivité de ML-ACT.

Son système d'analyse statique est réduit et mériterait d'être développé. En effet, la forme actuelle de l'analyse de la communication qui consiste en l'extraction d'un terme CAP, impose trop de limitations aux programmes correctement typés. La politique d'analyse doit donc être revue. Plusieurs pistes sont actuellement envisagée : soit une extension de CAP pour diminuer les contraintes de la traduction, soit passer outre CAP et générer directement des contraintes de type qui sont normalement extraites du terme.

Enfin, la génération de code exécutable, qui n'est pas vraiment le but du projet, n'est pas implantée. On a uniquement conçu et testé «à la main» différentes politiques possibles. L'étape suivante consiste donc en une implantation de cette traduction afin de disposer d'un compilateur complet.

Toutes ces orientations n'ont pas la même priorité. En effet vu le cadre des recherches de l'équipe, je pense que le travail devra se concentrer principalement sur la partie typage. Le but est donc le développement d'un outil puissant permettant un diagnostique pertinent des communications dans un programme d'acteurs.

Bibliographie

[Agh86] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, Mass., 1986.

[AH92] G. Agha and C. Houck. Hal: A high-level actor language and its distributed implementation. In *21st International Conference on Parallel Processing (ICPP'92), vol II*, pages 158–165, August 1992.

[AW92] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proc. of the IEEE Symp. on LICS*, 1992.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proc. of the ACM Symp. on FPCA*, June 1993.

[Col97] JL. Colaço. *Analyses statiques d'un calcul d'Acteurs par typage.* PhD thesis, Institut National Polytechnique de Toulouse, October 1997.

[CPS96] J-L. Colaço, M. Pantel, and P. Sallé. CAP: An actor dedicated process calculus. In *ECOOP'96 Workshop on Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[CPSS95] J-L. Colaço, M. Pantel, P. Sallé, and A. Senteni. Un calcul d'acteurs primitifs (CAP). In *Actes des Journées du Groupement De Recherche en Programmation*, November 1995.

[GAP92] W. Kim G. Agha and R. Panwar. Actor language for specification of parallel computations. In *the Proceedings of the workshop on DIMACS '94*, 1992.

[GMM+78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pages 119–130. ACM, ACM, January 1978.

[Kah88] G. Kahn. Natural semantics. In *Programming of Future Generation Computers.* Elsevier, 1988.

[Lal90] R. Lalement. *Logique Réduction Résolution.* ERI MASSON., 1990.

[Ler96] X. Leroy. *The Objective Caml system release 1.01, Documentation and user's manual.* INRIA Rocquencourt, June 1996.

[McD93] Raymond C. McDowell. The relatedness and comparative utility of various approaches to operational semantics. Technical Report MS-CIS-93-16/LINC LAB 246, Department of Computer and Information Science, University of Pennsylvania, January 1993.

[MMS88] A. Marcoux, C. Maurel, and P. Sallé. A language for distributed applications. In *IEEE Workshop on Future Trends of Distributed Systems in the 90's*, 1988.

[Pan94] M. Pantel. *Représentation et Transformation : Un modèle de la réutilisabilité dans les langages fonctionnels à objets.* PhD thesis, Institut National Polytechnique de Toulouse, February 1994.

[Plo81] G. Plotkin. A structural approach to operationnal semantics. Technical Report DAIMI FN 19, Aahrus Univ., September 1981.

[Thi94] X. Thirioux. Inférence de type par résolution de contraintes ensemblistes pour un langage fonctionnel à objets (fol). Rapport de DEA,ENSEEIHT, June 1994.