Implémentation d'un langage de programmation logique d'ordre supérieur avec MALI

Pascal Brisset

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1119

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

IMPLEMENTATION D'UN LANGAGE DE PROGRAMMATION LOGIQUE D'ORDRE SUPERIEUR AVEC MALI

Pascal BRISSET

Novembre 1989

RR_1119

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication Interne n° 498 - Octobre 1989 - 28 Pages

Implémentation d'un langage de programmation logique d'ordre supérieur avec MALI

Pascal Brisset

Résumé

Nous présentons ici une implémentation d'un langage de programmation logique inspiré de λ-Prolog[Nad87]. Ce langage, comme Prolog, utilise la résolution mais, à la différence de ce dernier, manipule des termes d'ordre supérieur codés par des λ-termes typés. Pour cela, l'algorithme d'unification des termes d'ordre supérieur de Huet[Hue75] est employé. Un compilateur a été développé. Il génère du code utilisant la machine intermédiaire MALI[BCRU86].

Implementation of a language based on higher-order logic with MALI

Abstract

We present a logic programming language which follows the principles of λ-Prolog[Nad87]. This language, as Prolog does, uses SLD-resolution, but unlike Prolog, with higher-order terms coded on terms of the typed λ-calculus. The higher-order unification algorithm of Huet[Hue75] is used. A compiler for this language has been written. It generates code which uses the intermediate machine MALI[BCRU86].

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

Depuis l'invention de Prolog[Rou75], de nombreux reproches ont été faits à ce langage et de nombreuses solutions ont été proposées. On nomme généralement *extensions* ces dérivés du langage initial. Ces extensions consistent en une généralisation des caractéristiques originales (modification de la stratégie de recherche ou de l'unification par exemple) ou un ajout (évaluation fonctionnelle, contraintes, ...). Nous projetons d'intégrer diverses extensions dans la limite de leur compatibilité respective, à l'intérieur d'un seul langage baptisé Z. Le but est avant tout de proposer des maquettes permettant à l'utilisateur de tester et de choisir en fonction de ses besoins la spécificité qui lui convient.

Dans ce cadre, nous présentons ici une extension de Prolog où les termes manipulés ne sont plus les termes du premier ordre mais des termes d'ordre supérieur et où, de plus, la forme des buts est aussi généralisée.

Miller a mis en évidence la déficience des termes du premier ordre de Prolog pour manipuler des formules dans un démonstrateur automatique [MN87]. On peut généraliser cette observation à toutes les applications effectuant un calcul formel sur des structures de données contenant des variables qu'il est naturel de représenter par des variables logiques. En effet, Prolog ne possédant pas d'affectation, la liaison d'une variable est la seule manière de modifier un terme sans le recopier. Ces problèmes apparaissent par exemple pour les manipulations et les transformations de programmes.

Ceci a conduit Nadathur[Nad87] à définir un nouveau langage de programmation logique nommé **λ-Prolog** travaillant sur des λ-termes typés. λ-Prolog peut être vu comme une extension de Prolog. Des exemples de démonstrateurs ont été donnés dans [Fel87] prouvant l'adéquation de ce langage aux besoins de structures de données plus riches. Un interpréteur pour λ-Prolog a été écrit en Prolog. Cependant, il s'avère que les traitements nécessaires pour l'exécution d'un programme de ce langage sont importants, gourmands en mémoire et en temps. En travaillant sur des exemples réels, même de taille raisonnable, les temps de calcul sont prohibitifs, en particulier à cause d'une consommation de mémoire catastrophique.

Ces considérations et un intérêt général pour l'implémentation des langages de programmation logique nous ont poussés à écrire un compilateur pour un système inspiré de λ-Prolog. Ce travail de longue haleine a été facilité par l'emploi de la machine MALI, spécifique à l'implémentation des langages indéterministes et utilisant une recherche en profondeur.

Nous présentons dans un premier temps λ-Prolog et ses objectifs ainsi que l'algorithme d'unification en détail. Cet algorithme diffère beaucoup de ceux unifiant des termes du premier ordre car il est indéterministe.

Une présentation de la machine MALI vient ensuite. On y montre les différentes structures de données et de contrôle utiles dans notre implémentation, ainsi que les spécificités de la récupération mémoire qui y est utilisée.

Dans la partie suivante nous abordons les problèmes d'implémentation impliqués par les particularités du langage. Les choix effectués et les solutions adoptées y sont présentés.

Nous concluons tout en présentant les objectifs futurs.

2 λ-Prolog

2.1 Présentation

L'implémentation de démonstrateurs de théorèmes nécessitant généralement l'unification des termes manipulés, l'utilisation des langages de programmation logique y est tout naturellement adaptée. Il est en particulier très tentant de faire correspondre directement les termes traités par le démonstrateur et ceux du langage en utilisant un minimum de codage. Cependant cette correspondance devient délicate dès que les termes manipulés par le démonstrateur contiennent des variables. Celles-ci ne peuvent en général pas être représentées directement par des variables logiques du langage d'implémentation. Un tel codage conduit à l'utilisation de prédicats hors du domaine logique (comme **var** ou **free**) et mettant un peu trop en jeu la stratégie de sélection des buts et des clauses (autrement dit la recherche en profondeur et à gauche d'abord). La conclusion de cette constatation est que les termes du premier ordre ne sont pas adaptés à une représentation correcte des termes [MN87].

λ-Prolog est un langage de programmation logique manipulant des λ-termes typés à la place des termes du premier ordre classique. Un atout de λ-Prolog, dans le contexte des démonstrateurs abordé ici, est de pouvoir coder les variables des termes manipulés par des variables abstraites d'un λ-terme, et ainsi de pouvoir les traiter. Ceci permet par exemple de traiter simplement les quantifications existentielles et universelles sur les formules. Le principal avantage obtenu est que les programmes sont plus simples et plus clairs. En particulier, des propriétés sur ces programmes sont plus faciles à prouver.

Nous avons adopté ici une restriction de λ-Prolog. L'extension par rapport à un Prolog classique est double :

- Les termes manipulés sont des λ-termes typés.
- Un but peut-être quantifié universellement

A l'origine, λ-Prolog comprenait en outre :

- La disjonction sur les buts : on peut considérer ceci comme du sucre syntaxique, le OU de Prolog convenant parfaitement (pour traiter le but $A \vee B$, on ajoute au programme les deux clauses $X \vee Y :- call(X)$ et $X \vee Y :- call(Y)$).
- La quantification existentielle : par défaut, dans un but Prolog, toutes les variables libres sont quantifiées existentiellement.
- L'implication : cet aspect est un peu délicat à aborder dans le cadre d'un compilateur. Il suppose une modification dynamique du programme. En effet, la sémantique opérationnelle présentée par [FM88] est la suivante : pour montrer $\mathcal{D} \Rightarrow \mathcal{G}$, ajouter $\mathcal{D}$ au programme courant et montrer $\mathcal{G}$. Ceci pourra éventuellement être abordé dans le cadre d'un interpréteur développé sur notre compilateur.

La première extension adoptée nous conduit à utiliser l'unification d'ordre supérieur. Cette unification nécessite en particulier que les termes traités soient typés simplement (typage de Church [Chu40] avec un ensemble de constantes de type et un seul constructeur $\rightarrow$).

Cette recherche d'unification est seulement semi-décidable (c'est-à-dire que s'il y a une solution, elle peut être trouvée en un temps fini). Une recherche en profondeur d'abord suivant l'algorithme de Huet[Hue75] est adoptée.

La description de l'implémentation de la quantification universelle est assez floue dans la littérature. La solution la plus simple pour résoudre un but quantifié $\forall x G$ consiste à choisir un nouveau symbole c et à chercher à résoudre $G[c/x]$. Il faut vérifier ensuite que le nouveau paramètre c n'est pas capté par les variables logiques du but (c'est-à-dire qu'après résolution, c n'apparaît pas comme sous-terme du but résolu). Sans cette vérification, l'implémentation n'est pas correcte. Par exemple, pour la résolution du but $\forall x((F\ x) = x)$ où F est une variable, une constante c est choisie et on cherche à résoudre $(F\ c) = c$. L'unique solution retenue est $F \leftarrow \lambda u \cdot u$. La solution $F \leftarrow \lambda u \cdot c$ n'est pas valide à cause du test d'occurrence de c.

Un programme λ-Prolog est donc analogue à un programme Prolog classique. Il comporte en plus des déclarations de symboles accompagnés de leur type. On peut en outre déclarer des constructeurs de type. La syntaxe que nous avons adoptée, proche de celle de LISP, n'est pas celle de λ-Prolog, inspirée de celle d'Edimbourg.

2.2 L'unification d'ordre supérieur

Nous rappelons ici l'algorithme de semi-décision d'unification des λ-termes typés présenté et prouvé par Huet[Hue75] dont nous reprenons ici les notations.

2.2.1 Les λ-termes typés

On considère un ensemble $\mathcal{C}_\mathcal{T}$ de constantes de type, $\mathcal{V}_\mathcal{T}$ un ensemble de variables de type et $\mathcal{T}_0 = \mathcal{C}_\mathcal{T} \cup \mathcal{V}_\mathcal{T}$ l'ensemble des types atomiques. On remarque ici que pour les besoins de l'unification une variable de type est assimilée à un type atomique. L'ensemble $\mathcal{T}$ des types est le plus petit ensemble contenant $\mathcal{T}_0$ clos par l'opérateur $\rightarrow$:

$$\alpha, \beta \in \mathcal{T} \Longrightarrow (\alpha \rightarrow \beta) \in \mathcal{T}$$

Un terme de type $(\alpha \rightarrow \beta)$ est une fonction, ayant pour domaine les termes de type α et d'image les termes de type β. Le type d'un terme e sera noté $\tau(e)$.

Soit $\mathcal{C}$ l'ensemble des constantes, $\mathcal{V}$ l'ensemble des variables, $\mathcal{I}$ l'ensemble des inconnues et $\mathcal{A} = \mathcal{C} \cup \mathcal{V} \cup \mathcal{I}$ l'ensemble des atomes. L'ensemble Λ des termes est le plus petit ensemble contenant $\mathcal{A}$ clos par les opération d'abstraction :

$$e \in \Lambda, \tau(e) = \beta, x \in \mathcal{V}, \tau(x) = \alpha \Longrightarrow \lambda x \cdot e \in \Lambda \text{ et } \tau(\lambda x \cdot e) = (\alpha \rightarrow \beta)$$

et d'application :

$$e_1 \in \Lambda, \tau(e_1) = (\alpha \rightarrow \beta), e_2 \in \Lambda, \tau(e_2) = \alpha \Longrightarrow (e_1\ e_2) \in \Lambda \text{ et } \tau((e_1\ e_2)) = \beta$$

Nous nous distinguons ici de la présentation de Huet en ajoutant l'ensemble $\mathcal{I}$ des inconnues correspondant aux variables logiques de Prolog. Ceci semble préférable pour éviter les confusions avec les variables abstraites des λ-termes. Ce choix implique aussi qu'aucun terme manipulé au niveau de Prolog ne peut contenir de variables libres (non sous la portée d'une

abstraction), c'est-à-dire, ce ne sont que des combinateurs. Cette caractéristique sera importante lors de l'étape de réduction. Ce souci de distinction correspond aussi ultimement à une représentation interne différente.

2.2.2 Règles de conversion et forme normale

Nous utilisons les trois relations d'équivalence sur les λ-termes, l'α-conversion qui traite le nommage des variables, la β-réduction qui est la seule règle de calcul et qui correspond à l'application d'une fonction à ses arguments et l'η-expansion qui sera utile pour normaliser les termes.

La α-conversion

$$\lambda x \cdot e \Longleftrightarrow \lambda y \cdot e[y/x]$$

où $e[t/x]$ désigne le terme e où toutes les occurrences libres de la variable x ont été remplacées par le terme t.

La β-réduction

$$(\lambda x \cdot e_1\ e_2) \Longleftrightarrow \overline{e}_1[e_2/x]$$

où $\overline{e}$ désigne le terme e où toutes les abstractions ont été renommées avec des nouvelles variables par α-conversion. Un terme $(\lambda x \cdot e_1\ e_2)$ susceptible d'être β-réduit est appelé *redex*.

La η-expansion

$$e \Longleftrightarrow \lambda x \cdot (e\ x) \text{ si x n'apparaît pas libre dans e}$$

On dira qu'un λ-terme typé est sous forme η-expansée si, intuitivement, sa représentation reflète son type, c'est-à-dire si :

$$e = \lambda x_1 \cdot \ldots \lambda x_n \cdot e' \Longrightarrow \tau(e') \in \mathcal{T}_0$$

Par économie et tradition, nous abrégerons :

$$\lambda x_1 \cdot \ldots \lambda x_n \cdot e \text{ en } \lambda x_1 \ldots x_n \cdot e, \text{ et } ((\ldots((e_1\ e_2)\ e_3)\ \ldots e_n) \text{ en } (e_1\ e_2 \ldots e_n)$$

Un terme ne comprenant pas de redex est dit sous forme β-normale. Ici, nous utilisons la forme normale de tête. Tout terme e peut être écrit sous la forme :

$$\lambda x_1 \ldots x_n \cdot (@\ e_1 \ldots e_p)$$

où @ est appelé la tête de e, $x_1 \ldots x_n$ le binder de e. Le terme est dit *flexible* si la tête est une inconnue ($@ \in \mathcal{I}$) et *rigide* sinon. Nous imposons ici une forme normale de tête η-expansée, c'est-à-dire telle que $\tau((@\ e_1 \ldots e_p)) \in \mathcal{T}_0$. Chaque λ-terme typé possède une unique représentation de cette forme (propriété de Church-Rosser).

2.2.3 Le problème de l'unification

Une substitution σ est un ensemble de couples $< v_i, e_i >$ où $v_i \in \mathcal{I}, e_i \in \Lambda, v_i \neq e_i$ et $\tau(v_i) = \tau(e_i)$ tel que $i \neq j \Rightarrow v_i \neq v_j$. On définit alors l'application d'une substitution $\sigma = \{< v_i, e_i >, i = 1 \ldots n\}$ à un terme e comme :

$$\sigma(e) = e[e_1/v_1] \ldots [e_n/v_n]$$

Soient e_1 et e_2 deux termes **de même type**. On dit qu'ils sont *unifiables* s'il existe une substitution σ telle que

$$\sigma(e_1) = \sigma(e_2)$$

On appelle alors σ *unificateur* de e_1 et e_2.

Contrairement au cas des termes de premier ordre (manipulés par les interpréteurs Prolog classiques), pour les termes d'ordre supérieur, il n'existe pas un unificateur minimal unique. Par exemple, les deux termes $(F\ 1)$ et 1 où $F \in \mathcal{I}$ sont unifiés par $\sigma_1 = < F, \lambda x \cdot x >$ ou $\sigma_2 = < F, \lambda x \cdot 1 >$ et les substitutions σ_1 et σ_2 ne sont pas comparables (c'est-à-dire il n'existe pas σ tel que $\sigma_1 = \sigma\sigma_2$ ou $\sigma_2 = \sigma\sigma_1$).

La recherche d'unificateurs se fait grâce à un *arbre d'unification* constitué de nœuds OU. Chaque nœud de cet arbre est un ensemble de couples de termes à unifier. La racine est constituée du couple des deux termes initiaux à unifier. Chaque arête de l'arbre est étiquetée par une substitution. On passe d'un nœud à son fils en lui appliquant la substitution portée par l'arête et en simplifiant l'ensemble des couples obtenus (ceci correspond intuitivement à une unification du premier ordre). Toutes les arêtes dérivant d'un même nœud sont étiquetées par des substitutions portant sur la même variable. Enfin les feuilles de l'arbre constituent des succès ou des échecs dans la recherche d'un unificateur. Un exemple est donné en figure 1.

L'algorithme est divisé en trois traitements :

TRIV Cette procédure résout les cas triviaux d'unification d'une inconnue et d'un terme de même type où l'inconnue n'apparaît pas dans le terme. Ce cas est très fréquent en résolution Prolog.

SIMPL C'est un traitement de simplification d'un ensemble de paires à unifier.

MATCH C'est la partie non-déterministe (au sens commun de l'acceptation du mot en programmation logique) de l'algorithme. Ici sont générées les substitutions qui décorent les arêtes de l'arbre d'unification.

Les traitements TRIV et SIMPL correspondent à l'unification du premier ordre. Par souci de simplicité, la procédure TRIV est incluse dans SIMPL.

2.2.4 Description de l'algorithme d'unification

SIMPL La procédure SIMPL prend comme argument une liste N de paires de termes de même type et rend comme résultat une liste de paires, un succès ou un échec. On note $\uplus$ l'union disjointe et $\mathcal{I}(e)$ l'ensemble des inconnues qui apparaissent dans le terme e.

1. Si $N = \emptyset$, retourner un succès.

2. (TRIV) Si $N = < v, e_2 > \uplus N', v \in \mathcal{I}, v \notin \mathcal{I}(e_2)$, alors $\sigma = \{< v, e_2 >\}$et $N \leftarrow \sigma(N')$ et revenir en 1.

3. Si N ne contient pas de paires rigide-rigide, retourner N.

4. Sinon $N = \{< e_1, e_2 >\} \uplus N'$ avec $< e_1, e_2 >$ rigide-rigide, $\tau(e_1) = \tau(e_2)$.

$$e_1 = \lambda x_1 \ldots x_n \cdot (@_1\ e_1^1 \ldots e_{p_1}^1)\ \ n \geq 0, p_1 \geq 0$$

$$e_2 = \lambda x_1 \ldots x_n \cdot (@_2\ e_1^2 \ldots e_{p_2}^2)\ \ p_2 \geq 0$$

Si $@_1 \neq @_2$, retourner un échec.
Sinon $N \leftarrow \{< \lambda x_1 \ldots x_n \cdot e_i^1, \lambda x_1 \ldots x_n \cdot e_i^2 >, i = 1 \ldots p_1\} \uplus N'$ (on a $p_1 = p_2$ car $\tau(e_1) = \tau(e_2)$).
Revenir en 1.

MATCH La procédure MATCH prend une paire flexible-rigide $< e_1, e_2 >$ de deux termes de même type comme argument et rend comme résultat un ensemble (éventuellement vide) de substitutions.

$$e_1 = \lambda x_1 \ldots x_n \cdot (v\ e_1^1 \ldots e_{p_1}^1)\ \ v \in \mathcal{I},\ \tau(v) = \alpha_1 \rightarrow \ldots \rightarrow \alpha_{p_1} \rightarrow \beta$$

$$e_2 = \lambda x_1 \ldots x_n \cdot (@\ e_1^2 \ldots e_{p_2}^2)$$

Les termes construits pour les substitutions doivent être typés convenablement. Ces types ne sont pas décrits explicitement par souci de clarté. Par simplicité, pour éviter tous problèmes de nom, toutes les variables nécessaires à la construction des termes sont nouvelles.

1. $\Sigma = \emptyset$

2. Si $@ \in C$, alors il y a *imitation* :
   $\Sigma \leftarrow \Sigma \cup \{< v, \lambda w_1 \ldots w_{p_1} \cdot (@\ E_1 \ldots E_{p_2}) >\}$ avec $E_i = (h_i\ w_1 \ldots w_{p_1}), h_i \in \mathcal{I}$

3. Pour tout i tel que $\tau(e_i^1) = \gamma_1 \rightarrow \ldots \rightarrow \gamma_r \rightarrow \beta$, il y a *projection* :
   $\Sigma \leftarrow \Sigma \cup \{< v, \lambda w_1 \ldots w_{p_1} \cdot (w_i\ E_1 \ldots E_r) >\}$

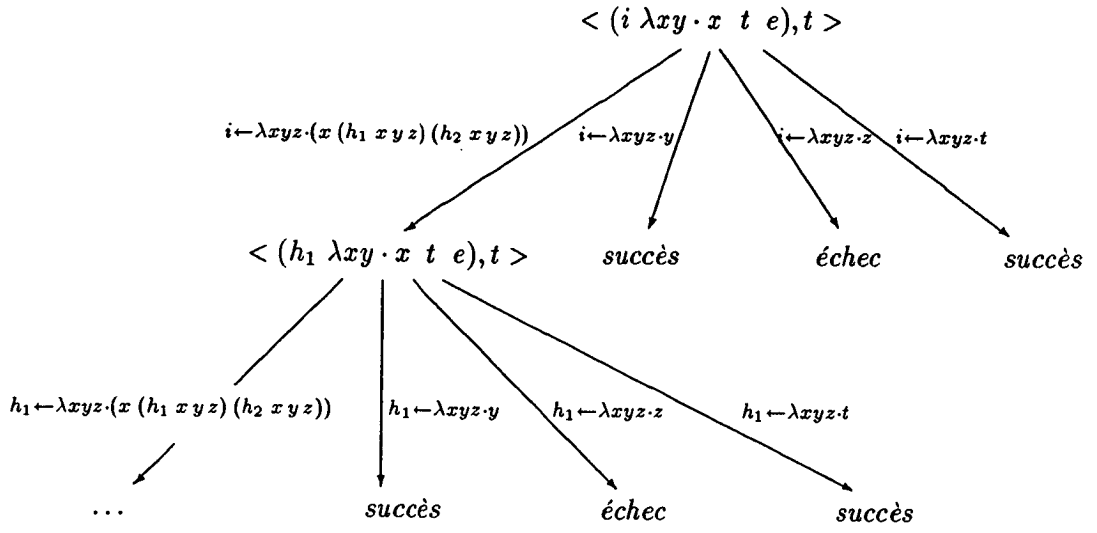
Lors de l'imitation (2) on substitue à l'inconnue du terme flexible un terme ayant le même constructeur que le terme rigide et les arguments les plus généraux possibles (les E_i). Quand il y a projection (3), on substitue à l'inconnue une fonction qui rend son i-ème argument. De même que pour l'imitation, il est appliqué à des arguments les plus généraux possibles.

Construction de l'arbre d'unification Soit à unifier les termes e_1 et e_2 de même type. On construit l'arbre $\mathcal{M}$.

1. La racine de l'arbre $\mathcal{M}$ est SIMPL$(< e_1, e_2 >)$.

2. S'il n'y a que des nœud de succès, d'échec, ou ne comprenant que des paires flexible-flexible, stopper.

Figure 1 : Exemple d'arbre d'unification



3. Sinon, choisir un nœud N et une paire flexible-rigide $< e_1, e_2 > \in N$ et calculer
   $\Sigma = \text{MATCH}(< e_1, e_2 >)$.
   Si $\Sigma = \emptyset$, remplacer le nœud N par un nœud d'échec.
   Sinon, pour tout $\sigma \in \Sigma$, dériver un fils
   $N' = \text{SIMPL}(\{< \sigma(e_1), \sigma(e_2) >, < e_1, e_2 > \in N\})$.
   Revenir en 2.

On peut remarquer que finalement, toutes les valeurs de liaisons construites par MATCH sont des combinateurs (sans variables libres).

2.2.5 Exemple d'unification

On se propose d'unifier les deux termes de même type e_1 et e_2.

$$e_1 = (i\ \lambda xy \cdot x\ \ t\ \ e), e_2 = t,\ \ i \in \mathcal{I},\ \ x, y \in \mathcal{V},\ \ t, e \in \mathcal{C}$$

$$\tau(x) = \tau(y) = \tau(t) = \tau(e) = \gamma \in \mathcal{T}_0,\ \tau(i) = (\gamma, \gamma \to \gamma) \to \gamma \to \gamma \to \gamma$$

L'arbre d'unification est donné en figure 1.

Toutes les substitutions proviennent de projections à part les deux branches les plus à droite. Il est clair que l'exploration de la branche la plus à gauche conduit à une boucle.

3 La machine MALI

Nous présentons ici succinctement la machine MALI sur laquelle le compilateur a été développé. Seuls les aspects essentiels à notre application apparaissent ici. Quelques spécificités

d'utilisation dues à λ-Prolog sont développées, en particulier à propos des variables à attribut et du contrôle du déclenchement du récupérateur.

3.1 MALI : une machine munie d'un récupérateur de mémoire adapté à la programmation logique

MALI[BCRU86] (Machine Adaptée aux Langages Indéterministes) est une machine abstraite dont la sémantique est appropriée à la mise en œuvre des langages "à la Prolog" avec un récupérateur de mémoire complet. Cette machine utilise une logique d'utilité qui a été prouvée spécifique aux langages sans affectation pratiquant une recherche en profondeur[Rid86]. Elle se comporte comme une mémoire qui aurait des fonctions plus diversifiées que seulement "lire" et "écrire". La mise en œuvre que nous utilisons ici est écrite en C.

Puisque la gestion de mémoire est spécifique à la recherche en profondeur, tout ce qui y a trait est défini au niveau de la machine MALI (i.e. installation et suppression de points de reprise, création et substitution de variable, et retour-arrière). Ainsi, tout ce qui ressemble à une pile de points de reprise ou à une traînée est caché et n'est pas de la responsabilité du concepteur d'interpréteur, simplifiant ainsi son travail. De plus, elle offre des possibilités de construction et parcours de terme qui sont le noyau de la représentation de valeur pour l'utilisateur.

Dans la suite, les identificateurs de concepts de MALI ont une lettre initiale majuscule.

3.2 Fonctions de base

Le type des arguments des fonctions de MALI est un type spécifique appelé "Nom". Une fonction, la "Désignation", associe un "Terme" à chaque Nom. Nom et Désignation sont les correspondants en MALI des notions d'adresse et de contenu en mémoire.

Le Nom indique la "Nature" et la "Sorte" d'un Terme. Les Natures possibles sont "Atome", "Construit", "Nuplet", "Variable", "Variable à Attribut" et "Niveau". La Sorte donne un typage élémentaire aux Atomes, Construits, Variables ou Variables à Attribut.

Nous présentons la sémantique de MALI à travers celle de ses fonctions. La signature des fonctions a la forme de "(E1,...,En) →(S1,...,Sn)", où les "Ei" (resp. "Sj") sont les types des paramètres d'entrée (resp. de sortie). Les premières lettres des identificateurs de Nature (A, C, Nu, V, VA, Ni) préfixent "Nom" ou "Sorte" pour contraindre le type d'argument ou de résultat.

Une fonction sert à analyser les Noms :

NatureSorte (Nom) →(Nature,Sorte)

Elle rend la Nature et la Sorte du terme nommé.

3.2.1 Structures de données

L'utilisateur connaît la structure d'un Nom d'Atome, et il peut en construire qui désigne ses propres valeurs d'atomes (en donnant au champ "Information" du nom, une valeur quelconque). Tous les autres Noms sont synthétisés par MALI.

Les fonctions suivantes servent à la représentation de valeurs complexes.

Cons (C_Sorte,Nom,Nom) →(C_Nom)
Car, Cdr (C_Nom) →(Nom)

Le résultat de "Cons" est un Nom du Terme construit dont les Sous-termes sont désignés par les deux paramètres. L'accès aux Sous-termes se fait avec "Car" et "Cdr".

Nuplet (N_Sorte, [ ]Nom) →(Nu_Nom)
NiemeSousTerme (Nu_Nom,Index) →(Nom)

Le résultat de "Nuplet" est un Nom du Terme dont l'arité est la taille du tableau de Noms, et les Sous-termes sont désignés par les éléments du tableau. L'accès aux Sous-termes se fait avec "NiemeSousTerme".

3.2.2 Variables

Les fonctions précédentes ne permettent de modifier la Désignation qu'en étendant son domaine de définition. D'autres fonctions permettent de modifier la Désignation en des points de son domaine de définition.

Ces fonctions définissent l'utilisation des Variables. Elles correspondent aux variables logiques de Prolog.

Variable (V_Sorte) →(V_Nom)
Substituer (V_Nom,Nom) →()

La fonction "Variable" rend un Nom d'une nouvelle Variable.
La fonction "Substituer" substitue un Terme à une Variable.
Tout Nom qui désigne la Variable avant la substitution, désigne le Terme après. Un effet de la substitution est que plusieurs Noms désignent le même Terme.

VariableA (VA_Sorte,Nom) →(VA_Nom)
SubstituerA (VA_Nom,Nom) →()
Attribut (VA_Nom) →(Nom)

La fonction "VariableA" rend un Nom d'une nouvelle Variable à Attribut. Une Variable à Attribut fonctionne comme une Variable, sauf qu'elle a une valeur associée : un Terme appelé "Attribut".
La fonction "Attribut" rend un Nom de l'Attribut d'une Variable à Attribut .

Utilisation des variables à attribut

On distingue deux grandes utilisations des variables à attribut :

- La mise à jour d'un terme : MALI étant une machine sans affectation, l'utilisation des variables à attribut fait office d'affectation sensible au retour-arrière. Les variables à attribut seront alors utilisées de la manière suivante : chaque terme susceptible d'être réécrit sera en fait l'attribut d'une variable. S'il y a réécriture, à la variable sera substitué le nouveau terme (qui sera l'attribut d'une nouvelle variable à attribut s'il est susceptible, lui aussi, d'être mis à jour). Dans les parcours futurs du terme, la variable

sera traversée automatiquement par MALI et par conséquent l'attribut sera invisible à l'utilisateur. Ceci est utile lors de la manipulation des λ-termes. En effet lors de la normalisation d'un terme, il y a réécriture des sous-termes à β-réduire (les redex) ou à η-expanser (les abstractions).

- Ajout d'information à une variable : La deuxième utilisation typique des variables à attribut concerne le besoin d'attacher de l'information à certaines variables quand elles sont à l'état libre et qui est inutile quand elle passe à l'état lié. On peut citer par exemple, les besoins de typage et de contraintes sur les variables (traitement du `dif` ou du `freeze` de PrologII [Le 88] et des paires flexible-flexible dans l'application présente).

Conformément aux spéficités de MALI, les substitutions pourront être défaites s'il y a retour-arrière. De plus, si les conditions le permettent(cf 3.2.5), la mémoire occupée par l'attribut sera récupérée. Il faut bien noter que cette récupération serait impossible s'il était nécessaire d'ajouter un construit au dessus de la variable pour y associer un terme, celui-ci ne pouvant être détruit.

3.2.3 Contrôle de la recherche

Les fonctions suivantes sont une généralisation de la stratégie de recherche en profondeur d'abord de Prolog. Elles permettent de gérer une Pile de Recherche. C'est une pile dont les éléments sont des Termes.

Sauvegarder (Nom) $\rightarrow$()
Reprendre () $\rightarrow$(Nom)

La fonction "Sauvegarder" empile un Terme sur la Pile de Recherche. Elle sert à installer un point de reprise. Réciproquement, "Reprendre" dépile un Terme de la Pile de Recherche. Cela correspond à un retour-arrière. Après une reprise, la Désignation n'est spécifiée que pour le Nom rendu en résultat. L'utilisateur doit parcourir le Terme repris pour connaître complètement la Désignation.

Des fonctions permettent aussi de supprimer des niveaux. Le fait de supprimer les niveaux supérieurs de la pile de recherche correspond au "Cut" de Prolog.

3.2.4 Contrôle de la récupération de mémoire

La machine MALI est définie de telle sorte qu'un récupérateur de mémoire efficace soit facilement réalisable. La théorie de la gestion de mémoire de MALI impose de considérer la récupération de mémoire comme une procédure que l'utilisateur appelle en donnant en paramètre le Nom des Termes utiles.

La fonction d'appel de la procédure de récupération de mémoire est :

Réduire (Nom) $\rightarrow$(Nom)

Cette fonction permet d'indiquer au récupérateur que le Terme nommé en paramètre est le seul Terme utile de la désignation. La Désignation est réduite (i.e. aucun autre Nom n'est signifiant) à celle du Nom résultat. Celui-ci désigne le seul Terme utile.

L'effet algorithmique de Réduire dépend de la mise en œuvre du récupérateur, mais ne provoque quasiment jamais un appel de procédure stricto sensu. Si le récupérateur est séquentiel et déclenché à l'approche de la saturation, alors Réduire est sans effet si un certain seuil d'occupation de la mémoire n'est pas dépassé. Sinon, la procédure de récupération est exécutée, et Réduire ne rend la main qu'après.

On voit que le récupérateur n'itère pas automatiquement sa procédure ; il doit attendre un signal de l'utilisateur de MALI. Grâce à ce dispositif, celui-ci peut conserver des Noms dans sa propre mémoire, mais il doit soumettre périodiquement tous les Noms dont il se sert au récupérateur, pour que celui-ci puisse opérer.

Il est conseillé de soumettre les Noms utiles quand il y en a peu, mais le plus souvent possible. La période optimale de soumission est très dépendante de l'application. Par exemple, en Prolog, l'interpréteur peut conserver des Noms dans son contexte pendant l'unification ou l'exemplarisation (la construction d'une nouvelle instance de corps de clause) et ensuite n'a plus besoin que du Nom de la nouvelle résolvante. La période optimale pour Prolog est donc le pas de résolution.

La période doit être suffisamment courte pour éviter la saturation de la mémoire. C'est le cas en Prolog car la quantité de mémoire allouée pendant une période est petite comparée à la mémoire totale : il s'agit essentiellement des nouveaux buts venant d'un corps de clause, et des substitutions de variables.

Cas de λ-Prolog Pour l'implémentation présentée ici, le problème du déclenchement du récupérateur est assez délicat. En effet l'unification, à cause d'une part de la normalisation des termes unifiés et d'autre part de la construction des substitutions (contrairement au cas du premier ordre), est très grande consommatrice de mémoire. Cette consommation de mémoire est difficile à borner, et nous devons envisager d'implémenter un unificateur plus prudent vis-à-vis de la mémoire. Il faut pour cela augmenter le nombre d'appels au récupérateur. Ceci nécessite de réussir à isoler un petit nombre de noms désignant les termes utiles au sein même des procédures utilisées.

3.2.5 La récupération de mémoire

Nous allons présenter succinctement les caractéristiques de la récupération dans MALI.

Les occasions pour une représentation de termes de devenir inutile sont multiples, mais on peut les classer en deux catégories :

- Perte d'accès lors d'un réduire : la représentation des Termes qui ne sont pas Sous-termes de l'argument du Réduire est inutile si elle ne sert pas pour un Terme sauvegardé.
- Perte d'accès lors des modifications de la Pile de Recherche : la représentation (y compris Terme sauvegardé et Traînée) des sous-piles détruites par suppressions de niveaux et Reprendre est inutile si elle ne sert pas à d'autres sous-piles ou à la Désignation.

Toute la nouveauté de la gestion de mémoire des langages comme Prolog vient de ce que l'évolution de la Désignation est contrainte par la stratégie de recherche. Le parcours doit donc interpréter la représentation des Variables en fonction de la Pile de Recherche et de la Traînée. Cette nouveauté est entièrement prise en compte dans MALI.

Par exemple, le récupérateur sait détecter que la substitution d'une Variable est inutile, cas recontré si pour tous les termes utiles, une variable donnée est vue dans l'état libre. Pour cela, il compare les rangs des éléments de Traînée et de la représentation des Termes sauvegardés dans la Liste de Sauvegardes. Si tous les Termes sauvegardés dont la représentation utilise une Variable donnée sont plus anciens que l'élément de Traînée qui cite cette Variable, alors la substitution est inutile. Le récupérateur range la marque NonSubstituée dans la Variable, et détruit l'élément de Traînée. La représentation de la valeur de substitution sera détruite si elle n'est plus utilisée. On peut donc annuler des substitutions sans attendre l'exécution de Reprendre. C'est indispensable pour programmer un "processus perpétuel" [War82].

Il détecte aussi qu'une Variable est inutile alors même que la valeur de substitution qui y est rangée est utile, ceci si cette variable doit être vue liée par tous les termes utiles. Si tous les Termes sauvegardés dont la représentation utilise une Variable sont plus récents que l'élément de Traînée qui la cite, alors cette Variable est inutile. Le récupérateur court-circuite la Variable en remplaçant toutes ses références par son contenu, et détruit l'élément de Traînée. De cette manière, la représentation des Variables qui n'ont servi qu'à construire des Termes est détruite.

3.2.6 Fonctions complexes

Les fonctions complexes sont fournies pour combler le fossé entre la sémantique de MALI décrite par les fonctions de base, et l'utilisation prévue, qui est la programmation d'interpréteurs.

Les besoins sont généralement les suivants :

- Construction descendante des Termes. Avec les commandes décrites précédemment, on doit construire un Terme des feuilles vers la racine. Une construction descendante peut être désirable et plus efficace car elle évite d'utiliser une pile.

- Parcours en profondeur des termes. Il faut utiliser une pile pour parcourir un Terme. Cela peut être un goulot d'étranglement pour des procédures comme l'unification. En conséquence une gestion correcte de cette pile est nécessaire.

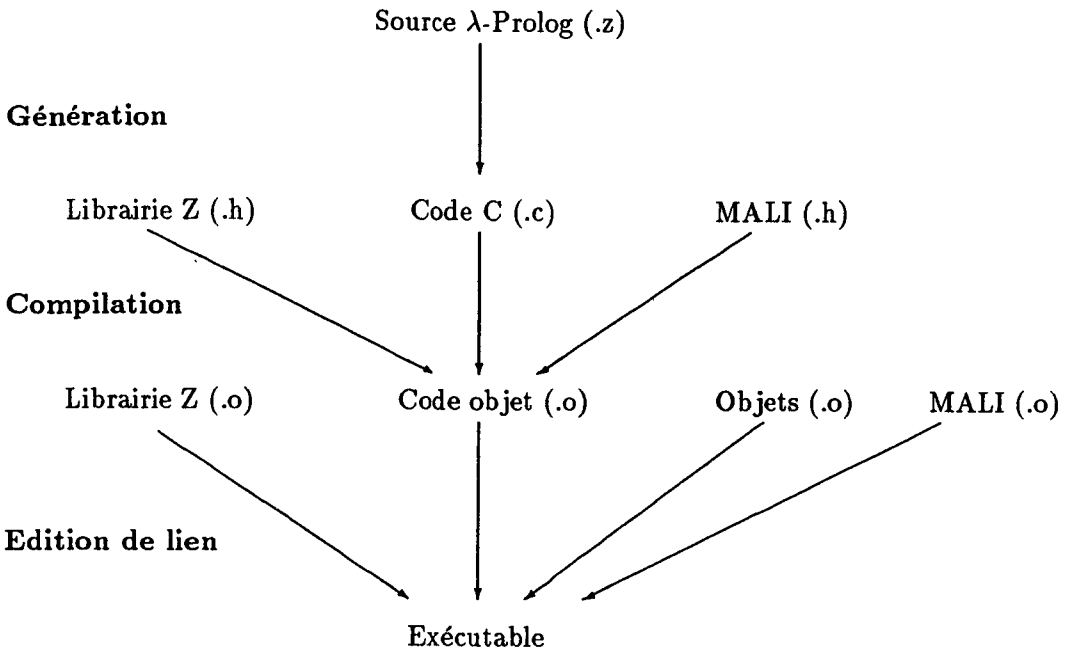
4 L'implémentation

Avec l'utilisation de la machine MALI, l'écriture d'un compilateur ne nécessite pas de suivre le modèle habituel de Warren[War83]. MALI gérant la manipulation des termes et l'indéterminisme, la compilation se fait directement vers du code C, le contrôle étant réalisé à ce niveau et non par la machine intermédiaire. Le compilateur suit alors le schéma 2. Il nous semble essentiel que les applications écrites en Z puissent être utilisées sans restriction au niveau du système et non par l'intermédiaire d'un interpréteur. Ce schéma permet aussi de faire de la compilation séparée simplement et d'interfacer avec n'importe quelle application écrite en C.

L'étape de génération transforme du source Z en code C. Celui-ci est constitué en grande partie par des macro-commandes soit de MALI soit d'une librairie spécifique. Lors de l'édition de lien, on assemble :

- le code objet du fichier à compiler

Figure 2 : Schéma de compilation



- le code de la machine MALI
- la librairie qui contient en particulier l'unificateur et le réducteur
- d'autres codes objets qui en l'occurrence peuvent provenir aussi d'une compilation d'un fichier Z.

4.1 La génération

La génération du code C à partir d'un source Z comporte quatre grandes parties. On commence par une phase de prétraitement syntaxique. Ceci permet en particulier à l'utilisateur de définir des macro-commandes avec une syntaxe identique à celle utilisée en C. Après une analyse lexicale et syntaxique, une vérification de type est réalisée, les termes sont normalisés et enfin, il y a production du code C. Tout ceci est provisoirement écrit en Prolog classique et sera traduit ultérieurement en Z. Ceci constituera en outre un réel exemple de programmation avec Z.

Prétraitement Afin de donner à l'utilisateur une plus grande souplesse de programmation, avant analyse, un fichier source Z est traité par une étape de prétraitement (comme pour les sources C). Ceci permet entre autre le nommage de termes et l'inclusion de fichier (de définitions d'un autre module, par exemple). Cette solution a l'avantage d'être peu coûteuse et simple tout en étant parfaitement claire à l'utilisateur.

Typage Dans un programme source, tous les symboles de constantes sont déclarés avec un type. Lors de l'analyse, la cohérence de type conformément aux règles de formation des termes (données en 2.2.1) est contrôlée. Les types des variables et des inconnues sont alors synthétisés. Il faut bien noter que ce contrôle statique de type n'assure que la correction de ces simples règles syntaxiques. Il est clair que ceci n'empêche pas des conflits de type dynamiques.

Normalisation La normalisation des termes comporte deux étapes, l'η-expansion et la β-réduction afin d'obtenir une forme normale de tête. Dans un premier temps, les termes sont donc η-expansés conformément aux types générés à l'étape précédente. Ensuite, les termes sont normalisés par β-réduction des redex.

Production C'est à cette étape que le code C est généré. Le modèle d'exécution donné en 4.2 étant relativement simple, la tâche essentielle consiste à exemplariser les termes. La traduction effectue deux opérations :

- Le codage des têtes de clause. On produit des structures statiques C. Ces structures sont parcourues et éventuellement exemplarisées en termes dynamiques (MALI) lors de l'unification.
- Le codage des corps de clauses. On fabrique une séquence de macro-commandes (définies dans une librairie) qui, lorqu'elles sont exécutées, produisent un exemplaire de ce corps de clause.

4.2 Le modèle d'exécution

Le modèle adopté est l'adaptation directe de l'implémentation d'un compilateur pour un Prolog *pur* écrite par Olivier Ridoux. Nous commencerons donc par décrire ce modèle pour Prolog. L'appel à la partie indéterministe de l'unificateur (MATCH 2.2.4) étant écrite elle-même en Z, ceci ne modifie que très légèrement le traitement du cas classique.

4.2.1 Prolog classique

L'exécution d'un programme Prolog consiste en la gestion d'une *résolvante* contenant les buts à effacer. Initialement la résolvante contient la *question* de l'utilisateur. Nous rompons avec les habitudes en n'utilisant pas ce concept de question. La résolvante initiale contiendra toujours le même but pour lequel un prédicat aura dû être défini par l'utilisateur. De plus les arguments de ce but sont les arguments de l'appel depuis le système. Ceci entre dans le cadre d'une utilisation directe et souple des programme écrits en Z, sans avoir recours à un interpréteur.

La gestion de cette résolvante doit prendre en compte l'indéterminisme de Prolog ; c'est-à-dire le retour-arrière. Ceci est entièrement relégué dans MALI.

Le contrôle principal est le suivant :

1. Sélection du premier but de la résolvante.
2. Résolution avec une clause d'un prédicat de même symbole de tête. Retour en 1.

Comme nous le verrons ultérieurement, pour des raisons d'optimisation, ce contrôle n'est pas toujours rigoureusement respecté.

A chaque prédicat Prolog correspond une procédure C. Elle transforme la résolvante et rend un pointeur vers une autre procédure correspondant à un autre prédicat. C'est une continuation. Chaque procédure est constituée par une séquence de blocs numérotés qui correspondent aux clauses du prédicat. Un tel bloc comporte :

- Un code de structures C statiques représentant la tête de la clause.
- Une préparation pour la sauvegarde de la résolvante.
- Les appels à l'unificateur (du premier ordre) pour les différents arguments.
- Le code d'exemplarisation du corps de la clause.

Sauvegarde La sauvegarde de la résolvante est nécessaire pour le retour-arrière. Elle doit être réalisée avant chaque effacement d'un but. Cependant, on peut économiser simplement cette sauvegarde :

- Elle peut être retardée jusqu'à la première modification de la résolvante, c'est-à-dire la première liaison de variable du but.
- Elle est inutile pour la dernière clause d'un prédicat.

En plus de la résolvante, la sauvegarde contient le numéro de la prochaine clause à sélectionner.

Unification Elle consiste en un parcours simultané d'un argument du but et du terme statique de l'argument correspondant de la tête de clause. Elle peut conduire à une exemplarisation dynamique du terme statique et aussi à l'unification de deux termes dynamiques. D'autre part, comme décrit plus haut, une liaison de variable du but provoque la sauvegarde (si celle-ci n'a pas encore été réalisée). Un échec de l'unification conduit soit à la sélection de la clause suivante avec le but courant (si la sauvegarde n'a pas encore été faite), soit à la reprise de la sauvegarde.

Exemplarisation du corps Si l'unification de la tête de la clause et du but courant réussit, on doit ajouter les littéraux du corps à la résolvante. Pour cela ces littéraux sont exemplarisés. Cependant, on se détourne ici du contrôle global décrit plus haut ; le premier but du corps de la clause étant le prochain qui doit être résolu, il est traité de façon particulière pour préparer l'appel. Il n'est pas ajouté effectivement à la résolvante. De plus, si l'appel est un appel récursif au même prédicat, l'appel de procédure est remplacé par un simple saut.

4.2.2 Le cas de λ-Prolog

A cause de l'indéterminisme de l'unification des termes d'ordre supérieur, il ne suffit pas de changer la procédure d'unification du modèle précédent. Pour prendre en compte cet indéterminisme, nous avons écrit les différents appels de MATCH à l'imitation et à la projection en

Z. Nous avons intégré les traitements SIMPL et TRIV au niveau de l'unification du premier ordre décrite plus haut. Cette unification produit donc éventuellement un ensemble de paires flexible-rigide. Dans ce cas, l'appel au premier but du corps de la clause est remplacé par un appel à MATCH et le premier but est traité alors comme un but de corps de clause.

4.3 La représentation des termes

Nous présentons ici la représentation des termes adoptée. Les structures différentes codées sur des termes MALI de même nature utilisent des sortes distinctes. Ceci est indispensable pour les distinguer statiquement et dynamiquement.

Les types Une des grandes nouveauté des λ-termes par rapport aux termes du premier ordre est la présence de types. Ces types sont nécessaires pour l'unification d'ordre supérieur. Leur représentation et leur manipulation dynamique est donc inévitable. Ils ne constituent donc pas seulement un outil de programmation utile à l'écriture de programmes corrects, mais un élément utile au calcul.

La première observation que l'on peut faire, c'est qu'une représentation systématique du type pour chaque terme conduit à de nombreuses redondances. En effet conformément aux règles de formation des λ-termes, si on a :

$$(f\ e_1 \ldots e_n),\ \tau(f) = \alpha_1 \to \ldots \to \alpha_n \to \beta$$

alors on a forcément :

$$\tau(e_i) = \alpha_i,\ i = 1 \ldots n$$

De plus, les termes étant représentés sous forme η-expansée, nous avons de même, si :

$$e = \lambda x_1 \ldots x_n \cdot e'$$

alors :

$$\tau(e') = \beta \in \mathcal{T}_0,\ \tau(e) = \tau(x_1) \to \ldots \to \tau(x_n) \to \beta$$

Ces premières observations nous laissaient espérer qu'il n'était pas nécessaire de représenter ces types. En effet, il apparaît que le type est intrinsèquement contenu dans la représentation en forme normale du terme (η-expansée). Il restait à réduire tous les types atomiques à un seul type atomique γ, et l'économie de représentation était totale. Malheureusement cette écomomie est trop grande et un problème se pose : une variable qui n'a d'occurrence que dans le binder d'un terme, n'est jamais η-expansée. Ceci implique que son type n'est pas connu. Par exemple, l'unification de $\lambda x \cdot 1$ et $\lambda y \cdot 1$ réussit toujours même si x et y n'ont pas le même type. Ne pouvant pas caractériser l'ensemble des termes que nous traitions correctement avec cette représentation, nous avons décidé d'abandonner cette économie. Nous conservons finalement des informations de type complètes, cependant avec un seul type atomique γ [1]. Toujours pour des raisons de représentation sous forme normale, seul le type des variables et des inconnues doit être représenté. Un type est donc, soit :

- un type atomique : il est représenté par un atome de MALI.

[1] Cette solution est provisoire, le choix du typage n'étant pas définitif (voir conclusion).

- un type construit : il est représenté par un construit.
- une inconnue de type : elle est représentée par une variable.

Les inconnues de type sont indispensables avec le typage élémentaire présenté ici pour permettre un minimum de polymorphisme (un prédicat de concaténation doit être utilisable pour des listes d'entiers comme pour des listes de symboles ou de termes d'un type donné quelconque). Ces inconnues de type (ou variables de type, puisqu'il n'y a pas de confusion possible) conduisent cependant à des problèmes de représentation normale. En effet, d'une liaison d'une variable de type peut provenir un changement d'arité de la tête d'un terme (si c'est une inconnue ou une variable). Ce changement d'arité remet en cause la normalisation statique du terme et demande une η-expansion dynamique. Voici un exemple d'η-expansion :

$$e = \lambda x \cdot x \text{ avec } \tau(x) = v \in \mathcal{V}_\mathcal{T}$$

Si la variable de type v est liée à $\gamma \rightarrow \gamma$ alors e doit être réécrit en :

$$e \leftarrow \lambda xy \cdot (x\ y)$$

Les abstractions La représentation interne des abstractions correspond directement à la forme normale de tête ; toutes les abstractions emboitées les unes sous les autres sont réunies. Ainsi $\lambda x_1 \ldots x_n \cdot t$ est représenté par un nuplet de MALI d'arité $n+1$. Les n premiers arguments du nuplets sont les variables du binder et le dernier est le terme sous l'abstraction. Chaque variable est un construit d'un numéro d'identification et du type de la variable.

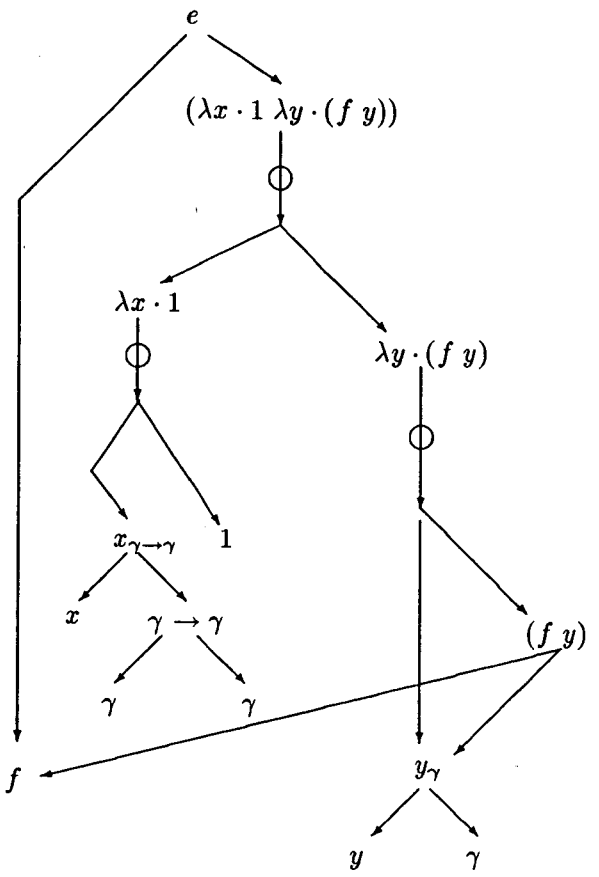
D'autre part, à cause des types variables, une abstraction peut devoir subir une η-expansion. Nous codons donc une abstraction avec l'attribut d'une variable à attribut de MALI. Ceci permet la modification du terme (cf 3.2.2, première utilisation des Varas), en conservant une gestion de mémoire optimale.

Les applications Les applications sont aussi codées par des nuplets de MALI. Le premier argument du nuplet est la tête de l'application et les éléments suivants sont les arguments de l'application. Pour distinguer les termes du premier ordre afin de les traiter plus efficacement, les applications dont la tête est une constante sont codés par des nuplets de nature différente. Les autres, c'est-à-dire celles commençant par une inconnue ou une variable sont susceptible d'être β-réduites. Elles sont donc l'attribut d'une variable à attribut pour permettre la réécriture (cf 3.2.2, première utilisation des Varas).

Les inconnues Les inconnues doivent comporter une information de type. Cette information de type devient inutile lors de la liaison de cette inconnue. Naturellement, c'est encore la variable à attribut de MALI qui va permettre le codage ; le type d'une inconnue est représenté dans l'attribut de la variable la représentant (cf 3.2.2, deuxième utilisation des Varas).

Les atomes Toutes les constantes doivent être déclarées et typées par l'utilisateur. Si une constante possède un type fonctionnel, cette déclaration de type fixe l'arité de la constante. Grâce à cette restriction et à l'η-expansion statique, il n'est pas nécessaire de représenter le type des constantes. Ainsi, une constante c de type $\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \gamma$ aura toujours ses occurrences en tête d'une application $(c\ t_1 \ldots t_n)$ avec $\tau(t_i) = \alpha_i$.

Figure 3 : Représentation interne de $e = (f\ (\lambda x \cdot 1\ f))$, $f \in \mathcal{C}$, $\tau(f) = \gamma \to \gamma$, $x \in \mathcal{V}$



Exemple Nous donnons en figure 3 un exemple de représentation. Chaque variable à attribut est représentée graphiquement par une arête décorée par un cercle qui sera noirci si elle est liée. Les partages de structures sont montrés. On peut remarquer l'η-expansion d'une occurrence de la constante f et les abstractions et applications susceptibles d'être réécrites sous la coupe de variables à attribut. Les types attachés aux variables apparaissent.

4.4 Le réducteur

La mise sous forme normale des λ-termes est indispensable pour les besoins de l'unificateur. Cette mise sous forme normale comprend la réduction des redex et l'η-expansion des inconnues et des variables.

En fait, sauf pour des facilités de lecture suite à un affichage, il n'est jamais nécessaire de réduire entièrement un terme. Nous avons donc simplement défini des commandes de parcours des λ-termes analogues aux commandes de MALI, mais qui assure que l'utilisateur ne verra, en utilisant ces commandes, que des termes sous forme normale. On peut faire le

parallèle entre MALI qui ne montre pas les variables qui ont été liées et ces commandes qui ne montrent pas de redex mais leur forme réduite. Cet ensemble de commandes de parcours constitue une interface entre MALI et l'utilisateur de λ-termes (l'unificateur). Cette solution utilise donc des réductions par nécessité.

Ces commandes de parcours sont identiques à celles de MALI. La seule différence est que le réducteur est invoqué lors de chaque accès à un sous-terme. Ce réducteur réduit les redex et η-expanse si nécessaire. Grâce au mécanisme des variables à attribut(3.2.2), c'est une opération semblable au remplacement physique d'un sous-terme qui est effectuée lors d'une réécriture. Nous utilisons un algorithme de réduction de graphe analogue à ceux décrits dans[Jon86]. Comme présenté en 2.2.2, la β-réduction nécessite le renommage et donc la recopie (à cause des partages) du terme auquel on substitue une variable. Nous minimisons certaines recopies de termes en reconnaissant certains combinateurs (en particulier les termes fabriqués par MATCH pour les substitutions). Cette économie intervient de deux façons :

- si le terme auquel on substitue une variable est un combinateur, il n'y a rien à faire (puisque le terme est clos, il ne contient pas d'occurrence de la variable), et il y a simplement partage du terme.

- si l'argument à substituer à la variable est un combinateur, le renommage n'est pas nécessaire (ceci ne dispense pas de copier la structure). Il y a alors économie de fabrication de nouvelles variables. On peut noter que dans l'implémentation présente, nous n'économisons pas seulement des noms de variable mais aussi des construits (du numéro et du type de la variable).

Nous montrons dans les figures 4 et 5 deux exemples de réduction et expansion conduisant à une réécriture, c'est-à-dire la liaison d'une variable à attribut. Dans la réduction on peut observer la copie du terme auquel est substitué une variable ainsi que le partage (optimum dans cet exemple) obtenu.

4.5 L'unificateur

Nous présentons ici un peu plus en détail l'implémentation de l'unificateur. Il est découpé en trois grande parties :

- UNIF l'unification des termes du premier ordre telle qu'on la trouve dans tout interpréteur ou compilateur Prolog classique. Elle n'est pas du tout compilée, c'est-à-dire que l'on fait appel à une procédure unique d'unification. Elle a été décrite en 4.2.1.

- SIMPL qui est éventuellement appelé par l'unificateur précédent s'il y a occurrence de termes d'ordre supérieur.

- MATCH qui est la partie indéterministe et qui est écrite en Z.

En fait, la partie UNIF est surabondante étant donné que SIMPL traite les termes du premier ordre. Nous l'avons conservé uniquement pour des raisons d'efficacité nécessaire sur les termes du premier ordre. Si, pendant une telle unification, un terme d'ordre supérieur est rencontré, la paire de termes à unifier est stockée et le parcours continue. Si la procédure UNIF termine avec un succès, la liste de paires stockées est fournie à SIMPL. Ce schéma a l'avantage de

Figure 4 : β-réduction de $(\lambda x \cdot (f\ x)\ 1)$, $\tau(f) = \gamma \to \gamma$

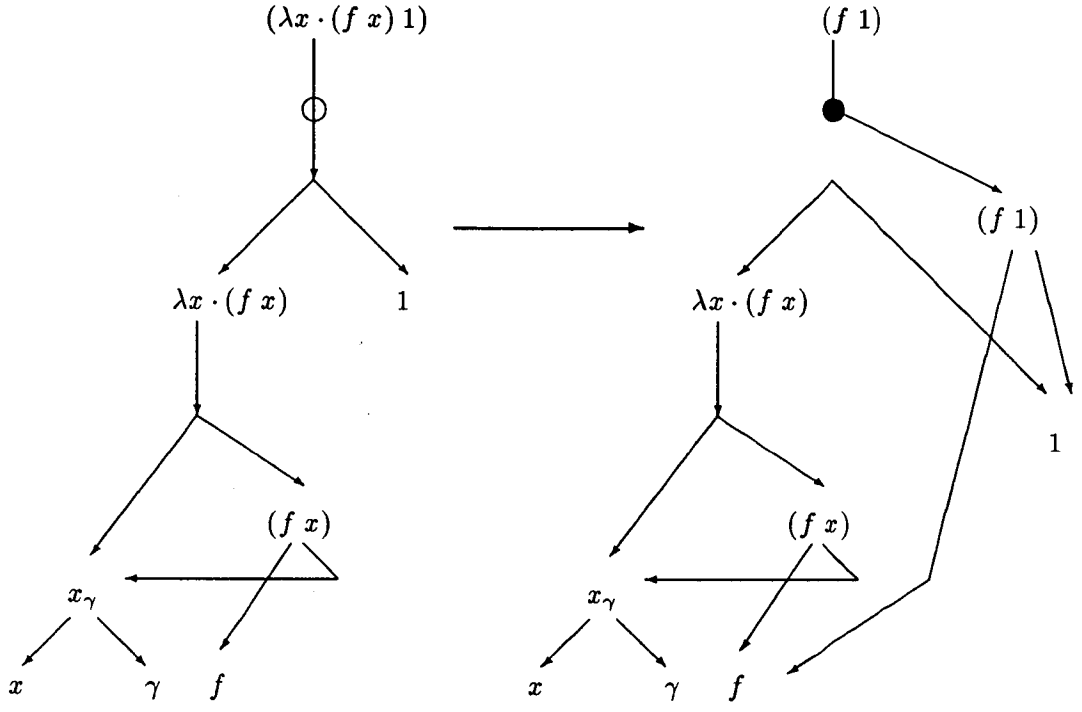
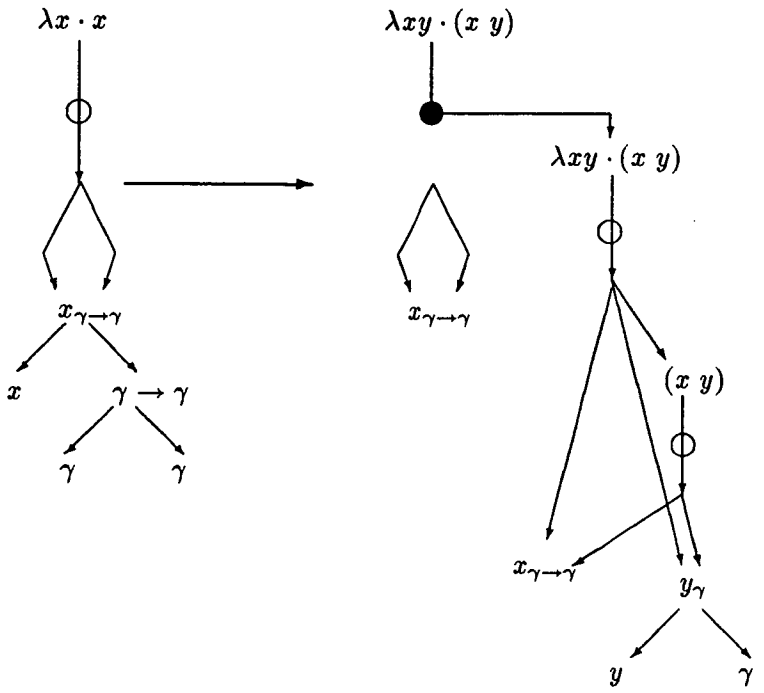


Figure 5 : η-expansion de $\lambda x \cdot x$ avec $\tau(x) = \gamma \to \gamma$ en $\lambda xy \cdot (x\ y)$



s'adapter simplement à l'ajout d'autres programmes d'unifications plus ou moins spécialisées. On peut très bien imaginer en effet, que selon le type de certains termes, une unification particulière est appelée, UNIF n'étant là que pour sélectionner l'unification adaptée.

SIMPL Dès que les termes d'ordre supérieur sont abordés, se pose le problème de la cohérence des types des termes à unifier. En effet nous pouvons remarquer, que, sauf pour les liaisons de variables, UNIF n'a aucune vérification à faire. Les paires de termes qui sont fournies à SIMPL ont donc subi préalablement une vérification de type qui se traduit par une unification des types des termes en présence. Une erreur est déclenchée s'il y a échec. Cette vérification de type ne doit pas être incluse à l'intérieur de SIMPL. En effet, après MATCH, il y a un appel à SIMPL avec des paires de termes qui sont de même type (ceci est assuré par les substitutions générées par MATCH). Il est alors inutile de le contrôler.

SIMPL, à la différence de UNIF, utilise les commandes de parcours spécialisées aux λ-termes, pour pouvoir en avoir une vue normalisée.

L'implémentation suit la description formelle donnée(cf 2.2.4). Une liste de paires flexible-rigide est fabriquée. Les paires flexible-flexible sont attachées comme des contraintes aux deux inconnues des deux termes flexibles. Ultérieurement, s'il y a liaison d'une de ces inconnues, la contrainte sera traitée. Encore une fois, c'est la variable à attribut de MALI qui va permettre ce codage : la paire flexible-flexible est ajoutée à l'attribut des inconnues (qui contenait déja le type). Ainsi, une inconnue peut être affublée d'une liste de contraintes qu'il faudra résoudre lors d'une liaison de cette inconnue.

MATCH La procédure MATCH est en fait relativement facile à mettre en œuvre. La seule difficulté provient du fait que c'est un traitement indéterministe. Ce problème est simplement solutionné en écrivant cet algorithme directement en Z. L'interface entre Z et C étant directe (par inclusion de blocs C dans le source), la tâche est aisée. MATCH prend comme argument une liste de paires flexibles et les résout successivement. Les procédures d'imitation et de projection sont alors appelées. Elles construisent les substitutions, effectuent la liaison de l'inconnue du terme flexible et appellent SIMPL avec la nouvelle paire rigide-rigide ainsi obtenue.

4.6 La quantification universelle

La quantification universelle sur les buts donne une facilité d'expression supplémentaire au langage. Ce qui est particulièrement remarquable, c'est qu'elle peut être utilisée en correspondance directe avec les quantifications universelles qui apparaissent dans les formules manipulées à l'intérieur d'un démonstrateur [Fel87]. Intuitivement, la quantification universelle force l'unification à ne pas être triviale, c'est-à-dire aux fonctions à consommer leurs arguments.

L'exécution d'un but quantifié universellement consiste à choisir une nouvelle constante du type approprié et à résoudre le but où la variable quantifiée est substituée par la nouvelle constante. On vérifie ensuite qu'aucune des variables libres du but n'a été liée à la nouvelle constante ou à un terme la contenant. L'implémentation impose donc simplement de disposer du `call` des Prologs classiques.

5 Conclusion

Nous avons présenté une implémentation d'un compilateur pour le langage Z. Cette implémentation a été réalisé grâce à l'utilisation de la machine MALI. Les enseignements à tirer de cette expérimentation sont multiples :

- la facilité et la rapidité avec lesquelles a été développé le compilateur sont dues entièrement à la grande généralité de MALI. Cette expérience supplémentaire d'implémentation prouve sa robustesse et son adéquation à la programmation logique. En particulier, cette tâche aurait été beaucoup plus délicate si nous avions utilisé la machine de Warren. Ce modèle étant beaucoup plus restrictif, il aurait été nécessaire de le modifier. Il est cependant clair que, pour les tests classiques en Prolog (du premier ordre) la généralité de MALI se paye considérablement en temps d'exécution.

- les quelques mesures de performance réalisées sont encourageantes. Bien que non comparables aux performances atteintes par les compilateurs actuels en ce qui concerne le premier ordre, elles dépassent de loin celles de l'interpréteur λ-Prolog de Miller dont nous disposons (version 2.7 sur Quintus-Prolog). Les raisons apparentes (non analysées en détail) concernent la gestion de la mémoire.

- la compilation ne paraît pas un enjeu important dans cette implémentation. En effet, l'unificateur d'ordre supérieur étant entièrement interprété, le gain par rapport à un interpréteur qui serait écrit avec la même technologie est faible. Cependant, ceci permet de conserver des performances raisonnables pour le traitement du premier ordre.

- des constatations peuvent être faites sur la représentation et le traitement des λ-termes :

  - dans le système de typage adopté (typage élémentaire de Church avec un seul type atomique), il n'est pas nécessaire de représenter exhaustivement tous les types ; si une forme normale η-expansée est prise, seules les variables et les inconnues doivent comporter un type.
  - l'η-expansion dynamique est rendue nécessaire à cause de la présence de variables dans les types.
  - la mise sous forme normale totale d'un terme n'est pas nécessaire. Un parcours effectuant des normalisations paresseuses est suffisant (forme normale de tête).

Pour améliorer les performances et la souplesse de programmation nous envisageons plusieurs modifications :

- le typage élémentaire utilisé jusqu'à maintenant paraît beaucoup trop limité. Nous projetons donc d'adopter des types plus élaborés prenant en compte en particulier :

  - le polymorphisme : ceci afin de préciser le typage. La seule manière de typer un prédicat polymorphe avec le système existant est d'utiliser des variables de type qui sont *a priori* souvent beaucoup trop générales.
  - l'inclusion de type : ceci pour structurer les types en treillis.

- pour améliorer les performances, il semble judicieux de compiler plus de traitements. En particulier, on peut compiler simplement l'unification du premier ordre, ceci à une profondeur donnée à choisir (pour l'instant nous compilons à la profondeur zéro). Ceci ne modifierait certes pas les performances pour la manipulation des termes d'ordre supérieur mais serait profitable pour les termes du premier ordre.

Bibliographie

[BCRU86] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: a memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symposium on Logic Programming*, IEEE, 1986.

[Chu40] A. Church. A formulation of the simple theory of types. *J. Symb. Logic*, 5(1):56–68, 1940.

[Fel87] A. Felty. Implementing theorem provers in logic programming. 1987.

[FM88] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In *CADE-88*, 1988.

[Hue75] G. Huet. A unification algorithm for typed $\bar{\lambda}$-calculus. *Theoretical Computer Science*, (1):27–57, 1975.

[Jon86] S.L. Peyton Jones. *The Implementation of Functional Programming Languages. International Series in Computer Science*, Prentice-Hall, 1986.

[Le 88] S. Le Huitouze. *Mise en œuvre de PrologII/MALI.* Thèse, Université de Rennes I, 1988.

[MN87] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, 1987.

[Nad87] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* Thesis, University of Pennsylvania, 1987.

[Rid86] O. Ridoux. *Gestion de mémoire temps-réel des langages de programmation relationnelle.* Thèse, Université de Rennes I, 1986.

[Rou75] P. Roussel. *Prolog : manuel de référence et d'utilisation.* Technical Report, G.I.A. Université Aix-Marseille, 1975.

[War82] D.H.D. Warren. Perpetual processes — an unexploited Prolog technique. *Logic Programming Newsletter*, (3), 1982.

[War83] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 491 **PHYSIQUE QUALITATIVE : PRESENTATION ET COMMENTAIRES**
Qinghua ZHANG
48 Pages, Septembre 1989.

PI 492 **SPARSE MATRIX MULTIPLICATION ON VECTOR COMPUTERS**
Jocelyne ERHEL
20 Pages, Septembre 1989.

PI 493 **THE SUPERIMPOSITION OF ESTELLE PROGRAMS : A TOOL FOR THE IMPLEMENTATION OF OBSERVATION AND CONTROL ALGORITHMS**
Benoît CAILLAUD
30 Pages, Septembre 1989.

PI 494 **EMPLOI DU TEMPS : PROBLEME MATHEMATIQUE OU PROBLEME POUR LA PROGRAMMATION EN LOGIQUE AVEC CONTRAINTES**
Xavier COUSIN
64 Pages, Septembre 1989.

PI 495 **NUMERICAL METHODS IN MARKOV CHAIN MODELING**
Bernard PHILIPPE, Youcef SAAD, William J. STEWART
46 Pages, Septembre 1989.

PI 496 **PLANIFICATION EN UNIVERS MONO ET MULTI-AGENTS (Définitions, concepts, objectifs, présentation d'un planificateur)**
Philippe PORTEJOIE
92 Pages, Octobre 1989.

PI 497 **LE TRAITEMENT D'EXCEPTIONS - ASPECTS THEORIQUES ET PRATIQUES**
Valérie ISSARNY
80 Pages, Octobre 1989.

PI 498 **IMPLEMENTATION D'UN LANGAGE DE PROGRAMMATION LOGIQUE D'ORDRE SUPERIEUR AVEC MALI**
Pascal BRISSET
28 Pages, Octobre 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique