Programmation Logique par Contraintes
Projet : Compilateur Prolog en Prolog

François Fages (fages@dmi.ens.fr)
Eric de la Clergerie (Eric.Clergerie@inria.fr)

1er avril 1997

Le but de ce projet est de réaliser en Prolog un mini-compilateur pour une sous-partie du langage Prolog. Ce compilateur va utiliser un jeu d'instructions restreint inspiré de celui de la Machine Abstraite de Warren (WAM). Ce jeu d'instructions permet d'exprimer (entre autres):

1. l'unification entre termes lors de la sélection d'une clause.
2. la construction de termes lors de l'appel d'un sous-but.
3. la gestion du non-déterminisme

Par exemple, considérons la version linéaire du prédicat `reverse` (étendue avec `test(A)`):

```
reverse(L,M) :- rev(L,M,[]).
rev([],L,L).
rev([A|L],Y,Z) :- test(A),rev(L,Y,[A|Z]).
```

Une compilation possible donne:

```
reverse/2                     % procédure pour reverse/2
                              % les arguments d'entrée sont dans A0 A1
    put_nil  A2               % charge le registre A1 avec []
    call     rev/3            % appelle rev/3 avec les arguments dans A0 A1 A2
    proceed                   % retour réussi pour reverse/2

rev/3                         % procédure pour rev/3
                              % les arguments d'entrée sont dans A0 A1 A2
    try_me_else clause2       % en cas d'echec dans la clause 1, aller à clause2
 clause1:                     % label marquant la clause 1 de rev/3
    get_nil        A0         % unifie A0 avec []
    get_x_var   X3 A1         % unifie A1 avec variable temporaire X3
    get_x_value X3 A2         % unifie A2 avec le contenu de X3 (d'où A1=A2)
    proceed                   % retour réussi
 clause2:                     % label marquant la clause 2 de rev/3
    trust_me_else_fail        % pas d'autre alternative apres clause2
```

```
allocate         4      % alloue 4 variables Y0 Y1 Y2 Y3
                        %               pour Z  A  Y  L
get_list         A0     % unifie A0 avec une liste [
unify_y_variable Y1     %                             Y1 |
unify_y_variable Y3     %                                  Y3 ]
get_y_variable   Y2 A1  % unifie Y2 avec A1
get_y_variable   Y0 A2  % unifie Y0 avec A2
put_y_value      Y1 A0  % charge A0 avec le contenu de Y1
call test/1             % appele test/1, argument dans A0
put_y_value      Y3 A0  % charge A0 avec le contenu de Y3
put_y_value      Y2 A1  % charge A1 avec le contenu de Y2
put_list            A2  % charge A2 avec la liste [
unify_y_value       Y1  %                            Y1 |
unify_y_local_value Y0  %           (globalisation de)  Y0 ]
call rev/3              % appelle rev/3, arguments dans A0 A1 A2
deallocate              % déallocation de Y0 Y1 Y2 Y3
proceed
```

Le jeu d'instructions complet est défini en Annexe A. La section suivante présente les bases de la machine abstraite sous-jacente. Un document plus complet décrivant plus en détail le jeu d'instructions ainsi que le modèle d'exécution de la WAM est également disponible.

1 La machine abstraite

La description de la machine abstraite de Warren (WAM) donnée ici est très succincte et a surtout pour but d'introduire le vocabulaire nécessaire à la lecture de ce document. Pour aider à la réalisation de ce projet, une description plus précise de la machine abstraite est fournie, extraite de la thèse de Daniel Diaz portant sur Wamcc, un compilateur Prolog générant du code C.

La WAM repose en premier lieu sur l'emploi de deux piles et d'un tas ("heap").

- Une **pile locale** où sont empilés (1) des **environnements** capturant l'information relative à chaque clause en cours de réfutation (position dans la clause, valeur courante des variables) et (2) les **points de choix**, qui indiquent où reprendre en cas d'échec [retour arrière].

- Une **pile de restauration** [trailing stack] qui garde une trace des variables liées devant être déliées en cas de retour-arrière au dernier point de choix.

- Le **tas** [heap] utilisé pour construire les termes logiques.

En plus de ces trois zones de données, la WAM comprends également un jeu de registres. En particulier, les **registres arguments** $A(i)$ sont utilisés pour passer les arguments lors d'un appel de prédicat. Ainsi, la compilation d'un appel $p(a, b)$ nécessite le chargement des registres $A(0)$ et $A(1)$ par respectivement a et b avant d'appeler la procédure relative au prédicat `p/2`.

La WAM assure que les registres arguments sont restaurés lors des retour-arrières grâce à des sauvegardes dans les points de choix.

La WAM propose également un jeu de **registres temporaires** $X(i)$ pour stocker des valeurs intermédiaires. Cependant, en pratique, les registres arguments et temporaires désignent en fait les mêmes registres physiques (i.e. $A(i) = X(i)$).

Les variables devant être conservées le long de la réfutation d'une clause ont besoin d'être sauvegardées de manière plus permanentes, grâce à des **"registres" permanents** $Y(i)$, alloués lors de la création de l'environnement de la clause. Par exemple, la réfutation d'une clause "`q(X) :- p(Y), r(X).`" nécessite de stocker la valeur de X dans un tel registre permanent pour pouvoir la propager correctement de sa première occurrence en tête de clause à sa dernière occurrence dans `r(X)`. Attention cependant, bien qu'appelés permanents, ces "registres" disparaissent en fait lorsque l'environnement est désalloué à la fin de la réfutation de la clause associée.

Comme il peut arriver qu'une variable associée à un registre (temporaire ou permanent) doive survivre à la clause qui l'a créée, il est alors nécessaire de **globaliser** cette variable en l'installant dans le tas.

Une partie de la difficulté de la compilation pour la WAM consiste à déterminer les types de registres nécessaires et les variables **dangereuses** devant être globalisées. Une compilation efficace cherche à limiter les globalisations et les registres permanents qui consomment de la mémoire.

2 Sous-langage de Prolog

Dans un premier temps, vous pouvez vous restreindre à la compilation de clauses de Horn pures, c'est à dire sans négation, coupure `!`, conditionnelle `->`, arithmétique et plus généralement prédicats "built-in" (comme `dif` ou `write`). Bien entendu, l'ajout de fonctionnalités n'est pas interdit !

3 Analyse Syntaxique

La réalisation d'un compilateur démarre généralement par celle d'un analyseur syntaxique permettant de lire vos fichiers sources. Dans votre cas, vous pouvez soit utiliser directement le "reader" de Sicstus ou réécrire (pour les puristes) un tel analyseur à l'aide des Grammaires de Clauses Définies (DCG) de Sicstus (voir le document de référence de Sicstus).

Signalons que l'approche DCG permet éventuellement de traiter dans le même temps la génération de code (section suivante) en associant des actions sémantiques aux règles de grammaires.

4 Génération de code

Partant de votre programme structuré comme un ensemble de clauses, elles-mêmes structurées comme une liste de termes, vous devez générer une liste d'instructions. Informellement, cette génération comprend les phases suivantes:

1. Une itération sur l'ensemble des clauses pour les grouper par **paquets** en fonction du prédicat de tête de clause. En effet, une unique "procédure" WAM est créée pour chaque prédicat.

2. Une itération sur les clauses de chaque paquet pour compiler chaque clause (point 3) et installer les reprises en cas d'échec (instructions `try_me_else`, `retry_me_else`, ou `trust_me_else_fail`).

3. Pour chaque clause $q(t_1, \ldots, t_n) :- \ldots$, allocation (éventuelle) d'un environnement pour certaines variables de la clause (`allocate`), compilation de l'unification entre les registres arguments $A(i)$ et les sous-termes t_{i+1} de la tête de clause (instructions `get_*` et `unify_*`) et compilation (point 4) du corps de la clause.

4. Pour chaque atome $p(t_1, \ldots, t_n)$ dans le corps de clause, construction des arguments t_{i+1}, mise en place de ces arguments dans des registres arguments $A(i)$ et appel de la procédure p/n (instructions `put_*`, `unify_*` et `call`) Après traitement du dernier atome de corps de clause, il faut déallouer (`deallocate`) et retourner à la clause appelante (`proceed`).

La compilation de la lecture (avec unification) ou de l'écriture d'un argument composé $t_i = f(r(m(a)), b)$ se fait par **aplatissement** récursifs en introduisant des variables intermédiaires (dans des registres temporaires) pour nommer les sous-termes composés de t_i. Ainsi, l'unification $t_i = f(r(m(a)), b)$ est remplacée par la série d'unifications $\{t_i = f(X1, b)\ X1 = r(B)\ X2 = m(a)\}$ où $X1, X2$ sont de nouvelles variables intermédiaires. Le code résultant (sans optimisation) donne:

```
get_structure f/2 A0    % A0 = f(
unify_x_variable  X1    %          X1,
unify_constant    b     %              b )
get_structure r/1 X1    % X1 = r(
unify_x_variable  X2    %          X2 )
get_structure m/1 X2    % X2 = m(
unify_constant    a     %          a  )
```

5 Emission du code

Le code WAM généré doit ensuite être émis vers un fichier d'extension `.wam`. De manière quelque peu arbitraire, ces fichiers `.wam` devront respecter la syntaxe donnée en Annexe B qui permet de considérer ces fichiers comme des (pseudo) programmes Prolog.

En pratique, vous pouvez vous familiariser avec cette syntaxe grâce à l'utilitaire `pl2wam` qui produit du code WAM (optimisé) à partir d'un programme logique. Notez que cet utilitaire vous permet également de jauger votre code !

```
TD-PLC> pl2wam reverse.pl
TD-PLC> cat reverse.wam
```

```
def_predicate(reverse/2,private).       % début de définition de reverse/2
wam(0, [
    put_nil(2),                         % put_nil A0
    execute(rev/3)]).

def_predicate(rev/3,private).           % début de définition de rev/3
wam(0, [                                % liste des instructions entre
    switch_on_term(1,2,fail,4,fail)]).  % les labels (locaux) 0 et 1
wam(1, [
    try_me_else(3)]).                   % try_else_else label(3)
wam(2, [                                % instructions entre 2 et 3
    get_nil(0),
    get_x_value(1,2),                   % get_x_value X1 A2 (notez que X1=A1)
    proceed]).
    ...
```

6 Evaluation du code

Deux alternatives vous sont proposées pour évaluer votre code WAM.

6.1 Wamcc (facile)

La solution la plus immédiate consiste à utiliser la partie terminale ("back-end") de Wamcc (développé par Daniel Diaz et déjà évoqué plus haut). Vous disposez d'un utilitaire (`wam2exe`) prenant en entrée le fichier `.wam` pour le lier avec la boucle top-level de Wamcc. Le résultat est un exécutable avec lequel vous pourrez tester votre programme.

```
TD-PLC> wam2exe reverse.wam       % => création d'un executable reverse
TD-PLC> ./reverse
wamcc                                INRIA Rocquencourt - ChLoE Project
Version 2.21                                        Daniel Diaz - 1994
| ?- reverse([a,b,c],L).
L = [c,b,a]
(0 ms) yes
```

6.2 Emulateur (dur)

La seconde alternative est d'écrire (en Prolog) en émulateur de code WAM pouvant lire les fichiers `.wam`. Cet émulateur est essentiellement une boucle qui sélectionne à chaque étape une instruction, la décode et l'exécute.

Le point délicat dans la réalisation d'un tel émulateur est l'implantation de la machine abstraite sous-jacente pour pouvoir exécuter les instructions ! Cependant, comme votre émulateur est écrit en Prolog, vous héritez directement de l'unification et de la gestion du non-déterminisme.

7 Optimisations et extensions

Maintenant que votre compilateur fonctionne, vous pouvez songer à l'optimiser et à l'étendre. Quelques idées (classiques) sont ici suggérées, mais de nombreuses autres n'attendent que vous !

Certaines de ces optimisations sont illustrées par la version suivante de `rev/3`:

```
rev/3
    switch_on_term Lvar clause1 fail Llist fail % indexation sur A0
 Lvar:                         % si A0 est une variable,
    try_me_else clause2        % essayer clause1, puis clause2
 clause1:                      % si A0 est une constante
    get_nil              A0
    get_x_value       X1 A2 % unifie X1 (=A1) avec A2
    proceed
 clause2:
    trust_me_else_fail
 Llist:                        % si A0 est une liste
    allocate 4
        ...
    deallocate                 % déallocation avant le dernier appel
    execute          rev/3 % saut terminal à rev/3 (plutôt qu'appel)
```

7.1 Gestion des registres

L'identification des registres arguments $A(i)$ et temporaires $X(i)$ permet de nombreuses optimisations, comme montré dans l'exemple ci-dessus pour `clause1` avec `get_x_value X1 A2`.

Plus généralement, l'analyse des utilisations et durées de vie des valeurs dans les registres (lors des unifications ou des constructions des arguments) permet de réduire grandement le nombre de registres et mouvements entre registres. De plus, une compilation efficace cherche à réduire le nombre de registres permanents alloués ainsi que le nombre de variables dangereuses. Ceci peut parfois être obtenu en intervertissant l'ordre d'unification ou de construction des arguments.

Ce travail d'analyse peut éventuellement se formaliser à l'aide de contraintes sur les registres. Un document fournissant les bases d'un tel algorithme est disponible.

7.2 Appels terminaux

Une cause majeure d'inefficacité d'un compilateur est le mauvais traitement des appels terminaux. Un **appel terminal** correspond à l'appel du dernier littéral d'une clause.

Le traitement d'une clause commence normalement l'empilement d'un environnement dans la pile locale, pour stocker les valeurs des variables de la clause. Or, après construction des arguments de l'appel terminal, nous n'avons plus besoin d'accéder aux variables de la clause. En conséquence, l'environnement peut être détruit juste avant l'appel terminal.

Par exemple, dans le cas de la seconde clause de `rev/3`, nous pouvons déallouer plus tôt et remplacer le dernier appel à `rev/3` par un saut (`execute`).

La mise en oeuvre de cette optimisation permet de réduire la consommation de la pile locale et d'éviter en particulier des débordements de pile pour des programmes très récursifs.

Cependant, bien qu'il n'y ait plus de références explicites aux variables présentes dans une clause après l'appel terminal, des références implicites peuvent exister au travers des arguments passés à l'appel terminal. Avant de libérer l'environnement et au moment du chargement des registres arguments, il faut alors globaliser certaines variables dangereuses.

7.3 Indexation

Classiquement, la WAM utilise un mécanisme d'indexation sur la nature du premier argument t_1 de chaque prédicat $p(t_1, \ldots, t_n)$ qui réduit le nombre de clauses testées. Par exemple, en cas d'appel à `q(1,X)`, l'indexation permet de ne tester que les clauses pour `q\2` dont le premier argument est soit un entier, soit une variable, mais pas celles dont le premier argument est une liste.

L'indexation se fait en insérant en début de procédure une instruction `switch_on_term`. D'autres instructions permettent de raffiner après ce premier niveau, en indexant sur la valeur du premier argument (`switch_on_constant`, ...).

7.4 Prédicats "Built-in"

Pour ceux qui utilisent l'utilitaire `wam2exe`, vous pouvez à peu de frais compiler des appels aux prédicats "built-in" traditionnels de Prolog (arithmétique, `is`, `write`, ...) à l'aide des instructions `builtin_n` et `function_n`.

A Jeu d'instructions

A.1 Lecture/écriture des registres arguments

Vous disposez d'une série d'instructions `get_*` pour lire avec unification les registres arguments et d'une deuxième série miroir `put_*` pour charger les registres arguments.

Les instructions qui lisent ou chargent une variable sont dédoublées en `*_x_*` ou `*_y_*` selon le type de registre utilisé pour stocker cette variable. Enfin, la première occurrence d'une variable est traitée par une instruction de type `*_variable`, tandis que les autres occurrences utilisent des instructions `*_value`.

`get_constant C Aj`   `put_constant C Aj`

Lecture ou écriture dans Aj de la constante C

`get_integer I Aj`   `put_integer I Aj`

Lecture ou écriture dans Aj de l'entier I

`get_nil Aj`   `put_nil Aj`

Lecture ou écriture dans Aj de la constante []

`get_list Aj` `put_list Aj`

Commence la lecture ou l'écriture dans Aj d'une liste

`get_structure F/N Aj` `put_structure F/N Aj`

Commence la lecture ou l'écriture dans Aj d'une structure de foncteur F/N

`get_x_variable Xi Aj` `put_x_variable Xi Aj`
`get_y_variable Yi Aj` `put_y_variable Yi Aj`

Lecture ou écriture dans Aj d'une variable chargée dans le registre Xi ou Yi.

`get_x_value Xi Aj` `put_x_value Xi Aj`
`get_y_value Yi Aj` `put_y_value Yi Aj`

Lecture ou écriture dans Aj d'une valeur chargée dans le registre Xi ou Yi.

`put_y_unsafe_value Yi Aj`

Similaire à `put_y_value` mais utilisée pour des variables dangereuses.

A.2 Lecture/construction des termes composées

La lecture ou la construction des arguments t_i d'un terme composé $f(t_1, \ldots, t_n)$ introduit par exemple par une instruction `get_structure` se fait à l'aide des instructions `unify_*`. Ces instructions fonctionnent soit en lecture (avec unification) soit en écriture en fonction d'un drapeau booléen. Nous retrouvons une série très similaire à la série `put_*` avec les dédoublements en fonction du type de registres (temporaire ou permament), première occurrence ou non, variable dangereuse ou non.

Les instructions suivantes utilisent un registre interne de la WAM qui indique l'argument courant du terme composé en cours d'unification ou de construction.

`unify_void  Nb`

Saute les Nb arguments suivants si ce sont des variables anonymes

`unify_constant C`
`unify_integer I`

Unifie l'argument courant avec la constante C ou l'entier I.

`get_list Xj`
`get_structure F/N Xj`

Commencent l'unification du registre Xj avec une structure ou une liste. Ce sont les mêmes instructions que celles vues dans la section A.1 car Xj et Aj dénote en fait le même registre.

`unify_x_variable Xj`
`unify_y_variable Yj`

Unifie l'argument courant avec une variable contenue dans Xj ou Yj.

`unify_x_value    Xj`

`unify_y_value    Yj`

Unifie l'argument courant avec la valeur contenue dans Xj ou Yj.

`unify_x_local_value Xj`
`unify_y_local_value Yj`

Similaires aux instructions précédentes mais utilisés si la valeur de Xj ou Yj doit être globalisés (en cas de référence à une variable dangereuse).

A.3 Contrôle

`allocate Nb`

Alloue un environnement de Nb registres permanents sur la pile locale.

`deallocate`

Libère l'environnement au sommet de la pile locale.

`call F/N`

Appelle la procédure associée au prédicat `F/N`.

`proceed`

Provoque un retour réussi vers la clause appelante.

A.4 Gestion du non-déterminisme

`try_me_else L`

Empile un point de choix et stocke le label L comme adresse de reprise en cas d'échec.

`retry_me_else L`

Positionne à L l'adresse de reprise dans le point de choix courant.

`trust_me_else_fail`

Libère le dernier point de choix, indiquant qu'il n'y a plus d'autre alternative pour l'appel en cours.

A.5 Extensions

D'autres instructions sont disponibles en vue d'extensions de votre compilateur.

`fail`

Provoque explicitement un échec.

`switch_on_term Lvar Lcst Lint Llst Lstc`

Installe un aiguillage en fonction de la nature du premier argument d'un appel (chargé dans A0). Selon que A0 est une variable, constante, entier, liste ou structure, l'exécution continue ensuite au label `Lvar`, `Lcst`, `Lint`, `Llst` ou `Lstc`. Un label peut aussi valoir `fail` pour dénoter un échec pour le type d'argument correspondant.

```
switch_on_constant  NbCst  (C,L)   ...
switch_on_integer   NbInt  (I,L)   ...
switch_on_structure NbStc  (F/N,L) ...
```

Installe un aiguillage en fonction de la valeur du premier argument d'un appel en énumérant les constantes (entier ou foncteurs) possibles. Cet aiguillage complète le premier niveau d'aiguillage obtenu avec `switch_on_term`.

```
try   L
retry L
trust L
```

Similaires aux instructions `try_me_else`, `retry_me_else` et `trust_me_else_fail`, sauf que le label L désigne maintenant l'instruction suivante à exécuter. Le point de reprise à stocker est en fait l'instruction suivante.

`builtin_n C Xi1 .. XiN`

Appelle une fonction built-in de nom C retournant un booléen avec les arguments contenus dans les registres Xi1 à XiN. Un échec est déclaré si la valeur retournée est fausse.

`function_n C Xi Xi1 .. XiN`

Appelle une fonction built-in de nom C avec les arguments contenus dans les registres Xi1 à XiN. La valeur retournée par cette fonction est unifié avec le contenu de Xi.

B Format d'emission

Le mieux pour se familiariser avec le format d'emission est de tester quelques exemples avec l'utilitaire `pl2wam`.

- Le début de la procédure définissant le prédicat `F/N` est marqué par l'émission du fait Prolog:

  ```
  def_predicate(F/N,private).
  ```

- Un bloc d'instructions entre deux labels est émis sous la forme du fait Prolog:

  ```
  wam(<label_debut>,<liste_des_instructions>).
  ```

- Une instruction WAM `<instruction> <arg1> .. <argN>` est émise sous la forme du terme Prolog:

  ```
  <instruction>(<arg1>,...,<argN>)
  ```

- Une liste d'arguments (par exemple dans l'instruction `switch_on_constant`) est émise sous forme d'une liste PROLOG d'arguments.
- Les registres Ai, Xi ou Yi sont émis comme l'entier i, l'instruction permettant de déterminer la nature du registre.
- les entiers, constantes et noms de foncteurs F/N sont émis tels quels.
- les labels sont des entiers remis à 0 pour chaque procédure. Le label 0 est réservé pour pour le début d'une procédure. Les autres labels d'une procédure doivent être référencés par une instruction avant d'être définis. Enfin, les labels d'une procédure doivent être définis consécutivement (sans trous) et dans l'ordre croissant à partir de 0.