Um Compilador de Prolog com Expansão de Código

por

Michel Celestino Paiva Ferreira

Dissertação submetida à Universidade do Minho para obtenção do grau de Mestre em Informática na especialidade de Ciências da Computação

Departamento de Informática

Universidade do Minho

Julho de 1996

Aos meus pais.

Agradecimentos

Em primeiro lugar quero agradecer a excelente orientação que me deu o Professor Doutor Luís Damas, que me permitiu levar a bom termo este trabalho, demonstrando a todos os níveis uma disponibilidade e interesse irrepreensíveis.

Quero agradecer também ao Professor Doutor Fernando Silva que acompanhou e colaborou ao longo de todo este trabalho, demonstrando uma paciência, apoio e preocupação que nunca será demais enaltecer. Da mesma forma quero também agradecer ao Professor Doutor Vítor Costa pela colaboração prestada.

Um agradecimento também especial ao Professor Doutor José Maia Neves, pela atenção e disponibilidade demonstrada na cooperação entre o Departamento de Informática da Universidade do Minho e o Laboratório de Inteligência Artificial e de Ciência de Computadores da Universidade do Porto (LIACC).

Quero ainda exprimir a minha gratidão ao LIACC e ao projecto PROLOPPE pelas condições oferecidas e pelas facilidades concedidas para o desenvolvimento deste trabalho.

Quero agradecer o importante apoio financeiro que usufruí no âmbito do Programa PRAXIS XXI, pela concessão da bolsa de estudo com a referência BM/2093/94.

Um agradecimento especial vai para o Ricardo e para o Ricardo, companheiros de um mesmo percurso desde há vários anos, e que por isso contribuíram importantemente para o desenrolar deste trabalho.

Por fim, quero carinhosamente agradecer aos meus pais, à Sheila e ao Vasco pelo seu constante apoio e por me aturarem quando a disposição não era a melhor, nas alturas em que o trabalho corria pior.

Resumo

O objectivo desta dissertação é explorar uma nova forma de obter eficiência na compilação de programas Prolog. Esta compilação tem sido o tópico de muita investigação, atingindo um nível, no mínimo, tão elevado como o da compilação de linguagens imperativas. No entanto ainda existe uma diferença considerável entre o tempo de execução de um programa escrito numa linguagem imperativa e um programa equivalente escrito em Prolog. Esta diferença é fruto do código mais geral que se escreve numa linguagem de programação lógica comparativamente com as linguagens imperativas. Para ultrapassar isto é preciso gerar código especializado.

A forma de obter código especializado tem sido implementada recorrendo a interpretação abstracta para deduzir propriedades do programa, por forma a poder gerar esse código especializado. A abordagem explicada nesta dissertação mostra que não é necessária uma fase de interpretação abstracta. A propagação da informação implícita em cada instrução WAM é suficiente para permitir muitas das optimizações que eram suportadas pela interpretação abstracta. A propagação desta informação permite a obtenção de predicados especializados para os diferentes tipos de argumentos que aparecem nos *objectivos* das cláusulas. Mostra-se ainda que complementando as instruções WAM com informação acerca dos seus argumentos permite reduzir substancialmente a complexidade destas instruções, obtendo um efeito idêntico ao de definir uma nova máquina abstracta de um nível mais baixo do que a WAM.

As melhorias obtidas no código gerado comparativamente com o código geral, são equivalentes às melhorias possibilitadas por uma fase de interpretação abstracta nos sistemas que mais partido tiram desta fase de interpretação abstracta, como é o caso do Parma e Aquarius, resultando numa aceleração do tempo de execução do código de cerca de 5 vezes.

Abstract

The purpose of this dissertation is to explore a new way of obtaining efficiency in the compilation of Prolog programs. This compilation has been the subject of a lot of research, improving the level of logical languages compilation to, at least, as high as the one of imperative languages compilation. However, there is still a considerable gap between the execution time of a program written in an imperative language and an equivalent program written in Prolog. This gap is the result of a more general code that is written when using a logical language. To overcome this it is necessary to produce specialised code.

The traditional way of generating specialised code has been by using abstract interpretation to deduce program properties, that will allow the generation of such specialised code. The approach explained in this dissertation shows that it is not necessary a phase of abstract interpretation. The propagation of information implicit to each WAM instruction is enough to allow many of the optimizations that were possible due to abstract interpretation. The propagation of this information allows the creation of specialised predicates to the different type of arguments that appear in the goals of clauses. It is also showned that by adding information about their arguments to WAM instructions, the complexity of such instructions can be substancialy reduced, causing the same effect of defining a new abstract machine, more low-level than the WAM.

The improvements obtained in the code generated if compared with the general code, are equivalent to the improvements caused by a phase of abstract interpretation in the systems that take more advantages from this phase of abstract interpretation, like Parma or Aquarius, resulting in a speed-up of 5 times in the execution time of the code.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Tal como em qualquer trabalho de compilação o objectivo deste trabalho é traduzir o código de um programa escrito numa determinada linguagem para código que a máquina entenda e execute. A complexidade desta tarefa é substancialmente variável na razão de dois factores óbvios: a linguagem em que o programa está escrito, e a linguagem para a qual o programa é traduzido. Claramente, quanto mais próxima estiver a primeira linguagem da segunda mais fácil será a tarefa de tradução.

Muitas vezes a tradução não é feita directamente, mas gradualmente, passando por linguagens intermédias. Isto simplifica o trabalho, na medida em que há uma maior proximidade entre a linguagem original e a linguagem intermédia. Para além disso, para um novo processador executar o código de um programa escrito numa determinada linguagem de programação, basta escrever o compilador da linguagem intermédia para o conjunto de instruções do novo processador, reutilizando o compilador da linguagem original para a linguagem intermédia. Inversamente, para um mesmo processador pode-se reutilizar o compilador da linguagem intermédia para a linguagem executável, compilando diferentes linguagens para a mesma linguagem intermédia.

Quanto mais complexa é a tarefa de compilação mais difícil se torna de a fazer eficientemente. Daí que quando a eficiência é fundamental, os programas tendam a ser escritos numa linguagem o mais próxima possível da linguagem imediatamente entendível pela máquina.

As linguagens lógicas em geral, e o Prolog em particular, estão extremamente distantes da linguagem que a máquina entende. Esta distância é fruto não só do alto nível das linguagens lógicas, mas também da inadequação do conjunto de instruções dos processadores comuns para as linguagens lógicas. Este conjunto de instruções é muito mais apropriado para as linguagens imperativas, como o C ou o PASCAL, daí que a compilação de tais linguagens seja feita de forma mais eficiente, visto que há uma maior proximidade entre elas e a linguagem do processador. Existem no entanto arquitecturas de processadores concebidas expressamente para linguagens lógicas, sendo a PLM [Dob87] e a VLSI-BAM [Hol93] dois exemplos, mas sem generalizações comerciais.

A diferença existente entre o Prolog e o código executável dos processadores comuns, levou a que os primeiros sistemas de execução de Prolog fossem interpretadores [War77]. Em 1983 David Warren definiu a WAM [War83] (Warren Abstract Machine), uma linguagem intermédia para a compilação de Prolog, que rapidamente se tornou no modelo standard para a implementação desta linguagem. A razão para a popularidade da WAM na implementação de compiladores de Prolog é o facto de definir uma correspondência entre a terminologia lógica e a das linguagens imperativas, aproximando assim a linguagem Prolog do conjunto de instruções dos processadores comuns. Para além disso, a WAM constitui uma boa linguagem intermédia, na medida em que quer a passagem do código Prolog para código WAM, quer a construção de um emulador do código WAM são ambas tarefas relativamente simples.

Apesar da WAM ter contribuído grandemente para a aproximação do tempo de execução das linguagens lógicas ao das linguagens imperativas, tal diferença continua a ser substancial, na maioria dos compiladores de Prolog implementados. E no entanto tal não deveria acontecer, pois a execução do Prolog não é mais que uma generalização da execução das linguagens imperativas, como a WAM torna mais claro. E desde a definição da WAM que várias optimizações têm sido propostas e implementadas em compiladores de Prolog [Van94], elevando esse nível de compilação, no mínimo, ao nível de compilação das linguagens imperativas. Então qual a razão para esta diferença entre o tempo de execução de um programa em C ou PASCAL e um programa em Prolog?

Esta razão não deve ser procurada em aspectos ligados à forma como um programa é compilado, mas sim em aspectos ligados àquilo que é compilado, isto é, àquilo que se escreve nos programas Prolog.

As linguagens lógicas, das quais o Prolog é a mais generalizada, têm a grande virtude de permitir ao programador dizer aquilo que pretende sem ter de dizer exactamente como o obter. Ao contrário das linguagens imperativas em que a essência da programação é a descrição passo a passo das operações a fazer para se obter o que se pretende, em Prolog programam-se conceitos, definições, deixando os pormenores da sua execução para o compilador [CM81]. O problema dos programas Prolog é que a facilidade com que se escrevem estes conceitos e definições leva a que se escreva mais do que o necessário para obter o que se pretende.

Suponhamos que queriamos um programa que juntasse duas listas de elementos, por exemplo [1,2,3] e [4,5,6], numa única lista de elementos. Se tivéssemos que fazer o programa em C, ou em qualquer outra linguagem imperativa, para executar esta tarefa, provavelmente escreveriamos qualquer coisa do género: "pega no primeiro elemento da primeira lista e copia-o para o primeiro elemento de uma nova lista, depois pega no segundo elemento..., etc, até chegar ao fim da primeira lista; depois pega no primeiro elemento da segunda lista e copia-o para o lugar da nova lista imediatamente a seguir ao último elemento da primeira lista, depois pega no segundo..., etc, até chegar ao fim da segunda lista".

Em Prolog não se faz nada disto. Simplesmente dá-se a definição de junção de duas listas, e pede-se que se junte as listas pretendidas. O aspecto de um tal programa é o da figura 1.1. O predicado `append/3` é provavelmente o predicado mais compilado em Prolog, e aparecerá mais vezes durante esta dissertação.

É evidente a diferença entre este programa Prolog e o seu correspondente numa linguagem imperativa. Em primeiro lugar pela simplicidade do Prolog, sem preocupações de estruturas de dados, ou da forma como o programa é executado. Em segundo lugar, porque não fazem a mesma coisa. Enquanto que o programa na linguagem imperativa apenas sabe fazer a junção de duas listas, o programa em Prolog pode fazer muito mais, porque aprendeu a noção de junção de duas listas. Pode, por exemplo, fazer,

```
:- append([1,2,3],L,[1,2,3,4,5,6]).
```

que corresponde a perguntar o que é preciso juntar a [1,2,3] para obter [1,2,3,4,5,6], ou ainda,

```
:- append(L,[4,5,6],[1,2,3,4,5,6]).
```

com uma interpretação idêntica.

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

:- append([1,2,3],[4,5,6],R).
```

Figura 1.1: Junção de duas listas em Prolog

Esta é a grande beleza da programação em lógica, e em Prolog em particular. A descrição de conceitos ao invés da descrição sequencial dos passos a fazer para obter o resultado pretendido.

É claro que em termos de eficiência o programa da linguagem imperativa portar-se-á bem melhor, fruto da tarefa mais simples que tem que realizar. Levanta-se então a questão, se o que se pretende é somente juntar duas listas, então porquê gerar um código que vai fazer mais que isso?

A diferença entre o tempo de execução do Prolog e o tempo de execução das linguagens imperativas será esbatido quando se gerar código para o programa Prolog que faça somente aquilo que o programador tinha em mente, usando as poderosas características da programação lógica apenas onde forem necessárias.

1.2 Objectivos

Na execução de Prolog o sistema mais rápido existente é, sem dúvida, o compilador Aquarius [Van90] de Van Roy. Trata-se de um compilador extremamente complexo, concebido com o objectivo primordial de acelerar o mais possível a execução do código compilado, sem qualquer preocupação com o tempo que demora essa fase de compilação. A eficiência deste compilador é, segundo o autor, justificada em quatro princípios fundamentais:

1. Redução da granularidade das instruções.
2. Exploração do determinismo.
3. Especialização da unificação.
4. Análise global do programa.

Estes princípios são de facto determinantes na performance de um compilador de Prolog. Muitas das instruções da WAM são excessivamente complexas para permitir muitas optimizações possíveis e necessárias na geração de código eficiente. Daí a necessidade de dividir as instruções da WAM em instruções mais simples, que no caso do Aquarius levou à criação da BAM (Berkeley Abstract Machine), de forma a satisfazer o princípio da redução da granularidade das instruções.

A exploração do determinismo nos programas é importante porque a maioria dos predicados escritos são pensados para uma execução determinística. Estes são, no entanto, demasiadas vezes compilados ineficientemente usando *retrocesso* para simular a estrutura de *case* que as suas cláusulas normalmente representam.

A unificação é a essência do Prolog, e a sua forma geral é complexa visto que se podem tentar unificar quaisquer dois termos, sejam variáveis, átomos ou estruturas de qualquer tamanho. A sua concretização pode traduzir-se na passagem de argumentos, na atribuição de valores a variáveis, na reserva de memória, etc. Muitas vezes apenas uma destas acções é necessária, pelo que é importante simplificar o mecanismo para casos particulares que uma análise global do programa permite deduzir.

A importância da análise global de programas é a de ser a base de suporte para a extracção de determinismo e para a simplificação da unificação.

Não é objectivo deste trabalho a criação de um compilador mais rápido que o Aquarius. Nem sequer é nosso objectivo a criação completa de um compilador, desde a leitura do ficheiro de código, até ao desenvolvimento de um *debugger*. O que é objectivo deste trabalho é obter um compilador eficiente, que salvaguarde os quatro princípios descritos acima, e os implemente de uma forma simples e sem grandes custos adicionais, ao invés da complexa e custosa implementação do compilador Aquarius. É com esse objectivo que se implementa o *unfolding*, uma expansão do código WAM original, que, funcionando na sequência normal da geração de código, permite satisfazer estes quatro princípios fundamentais na compilação eficiente de Prolog.

Atendendo a que o objectivo deste trabalho se centra na geração de código eficiente, procurou-se evitar o desenvolvimento de um novo compilador de raíz. Foram estudados dois compiladores, o YAP [DSCRA89] (Yet Another Prolog) e a `wamcc` [CD95], optando-se por este último dado ser muito menos complexo, e mais simples a compreensão do código escrito, graças exactamente a estar totalmente escrito em Prolog.

Sendo o estudo dos resultados obtidos uma parte fundamental em qualquer trabalho deste género, este não poderia terminar pela geração de um código intermédio, que por muitas virtudes que tivesse não permitiria uma qualificação objectiva dos resultados obtidos, que só a medida do tempo de execução do código executável permite. Daí que fosse necessário prosseguir o trabalho para a geração desse código executável.

Uma das opções seria a construção de um emulador do código WAM expandido. Se bem que a sua construção fosse simples, os resultados que seriam de esperar eram razão suficiente para que não fosse essa a opção a seguir, uma vez que se tem a eficiência como factor primordial. Outra opção era a geração de código nativo para um determinado processador. Embora os resultados a esperar tivessem que ser bons, esta era uma hipótese demasiado trabalhosa, com preocupações de optimizações de muito baixo nível, que saiem do âmbito do objectivo desta dissertação. Por fim, e à semelhança da opção tomada no compilador `wamcc`, sobre o qual este trabalho é construído, decidiu-se gerar código C, deixando depois a geração de código executável optimizado para um compilador de C. Se a passagem de código WAM para C for feita com algum cuidado por forma a garantir eficiência, e em conjunto com as optimizações do código nativo que os compiladores de C fazem, os resultados a esperar são bons, tendo a vantagem extra em relação à geração de código nativo para um determinado processador de o código ser executável em qualquer máquina com um compilador de C.

O caminho a percorrer entre um sistema protótipo e um sistema completo e robusto é enorme e muitas vezes tortuoso. Nesta dissertação interessava experimentar algumas ideias na compilação de Prolog, testando a sua viabilidade e resultados, sem visar a construção definitiva de um tal sistema completo e robusto. O que se pretendia obter era um sistema que corresse um determinado número de programas Prolog, que permitissem tirar conclusões sobre as ideias implementadas.

1.3 Estrutura geral da tese

A estrutura desta dissertação é directamente influenciada pela estrutura do compilador implementado, seguindo os seus capítulos uma ordem que corresponde à ordem das diversas etapas do compilador. A figura 1.2 esquematiza estas etapas.

O capítulo 2 descreve a linguagem Prolog, que é o objecto de todo este trabalho, e a WAM, introduzindo as técnicas fundamentais para a compilação eficiente do Prolog. O capítulo 3

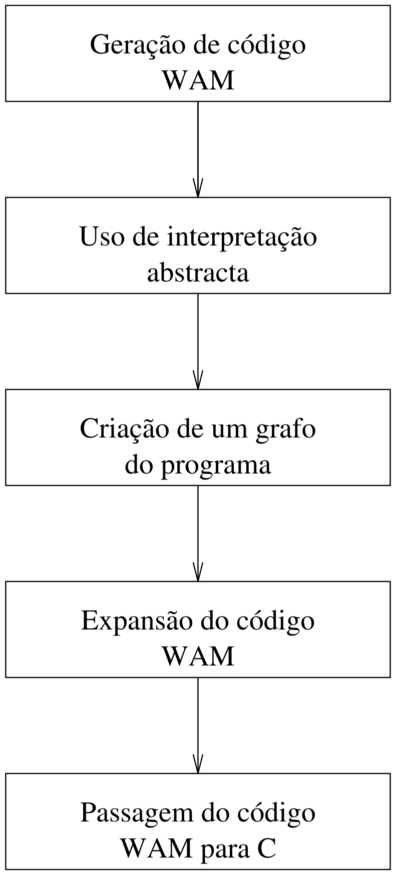

Figura 1.2: A estrutura do compilador

descreve a geração de informação para a expansão do código WAM, referindo a forma como o código WAM é gerado, falando da `wamcc` e da forma como este trabalho foi integrado nesse compilador. Descreve também o uso de interpretação abstracta para inferência dos tipos dos argumentos dos predicados à saída, e a criação de um grafo de um programa Prolog. O capítulo 4 descreve a forma como é feita a expansão do código WAM, e o capítulo 5 descreve a passagem deste código WAM para C, concretizando as optimizações possibilitadas pelo *unfolding*. O capítulo 6 faz uma avaliação das melhorias possibilitadas pelo *unfolding*. Finalmente, no capítulo 7 tiram-se conclusões sobre o trabalho e dão-se sugestões para trabalho futuro.

Capítulo 2

Prolog e a WAM

Este capítulo descreve a linguagem Prolog e os princípios da sua compilação eficiente, explicando os aspectos fundamentais da WAM e a forma genérica como um programa Prolog é compilado para código WAM.

2.1 A linguagem Prolog

Um *programa* Prolog é um conjunto de cláusulas lógicas que podem ser interpretadas como declarações *if-then*. Um *predicado* é um conjunto dessas cláusulas agrupadas pela seguinte relação: todas as cláusulas têm o mesmo nome e aridade (número de argumentos). Os predicados são normalmente identificados pelo par *nome/aridade*. A figura 2.1 representa o predicado `fib/2`, onde as duas primeiras cláusulas são chamadas *factos* e a terceira é chamada *regra*. As regras dividem-se em *cabeça* e *corpo*, sendo a cabeça a parte à esquerda do símbolo ':-' e o corpo a parte à direita. Cada uma das parcelas do corpo chama-se um *objectivo*. O símbolo ':-' quer dizer *se*, a vírgula ',' quer dizer *e*, e as palavras cuja primeira letra é maiúscula são variáveis.

A definição de `fib/2` é directamente executável pelo Prolog. Dependendo de qual argumento é entrada e qual é saída, a definição será executada diferentemente pelo Prolog. A execução procede como um simples demonstrador de teoremas. Dado um *inquérito* e um conjunto de cláusulas, o Prolog tenta arranjar valores para as variáveis do *inquérito* de forma a que este suceda. A semântica de execução do Prolog que testa as cláusulas na ordem em que aparecem e executa os *objectivos* do corpo das cláusulas da esquerda para a direita, torna

```
fib(1,1).
fib(2,1).
fib(N,R) :- N>2, N1 is N-1, N2 is N-2,
            fib(N1,R1),
            fib(N2,R2),
            R is R1+R2.
```

Figura 2.1: Um exemplo de um programa Prolog

o Prolog num fraco demonstrador de teoremas, devido a esta restrita ordem de execução, mas útil como uma linguagem de programação, sobretudo porque esta restrita ordem de execução pode ser implementada eficientemente, o que não acontece com um demonstrador de teoremas mais geral.

2.1.1 A variável lógica

Uma variável representa qualquer objecto, e pode até representar outra variável. Inicialmente o valor de uma variável é desconhecido, tornando-se conhecido por *instanciação*. Cada variável só pode ser instanciada uma vez, falhando a execução se tentarmos atribuir um novo valor a uma variável. O facto de uma variável poder representar outra variável cria cadeias de ligação de variáveis, sendo que quando uma destas variáveis da cadeia é instanciada para um valor, este valor é visto por todas as variáveis da cadeia.

O tipo de objecto que uma variável representa pode ser qualquer, sendo comuns tipos como átomos (e.g. `michel`, `abcd`), inteiros, listas (representadas com parêntisis rectos, e.g. `[Head | Tail]`), e estruturas (e.g. `fib(1,1)`).

2.1.2 A unificação

A unificação é uma operação que tenta encontrar a instanciação comum mais geral de duas estruturas de dados. Uma definição formal de unificação é dada por Lloyd [Llo87]. Instanciações de variáveis a valores ou a outras variáveis são feitas por unificação. Ao tentar igualar dois termos, as variáveis nesses termos são instanciadas para se tornarem iguais. Por exemplo, unificar `s(X,Y,a)` e `s(Z,b,Z)` (Figura 2.2) iguala `X` com `Z`, `Y` com `b`, e `a` com `Z`. O termo resultante da unificação é `s(a,b,a)`.

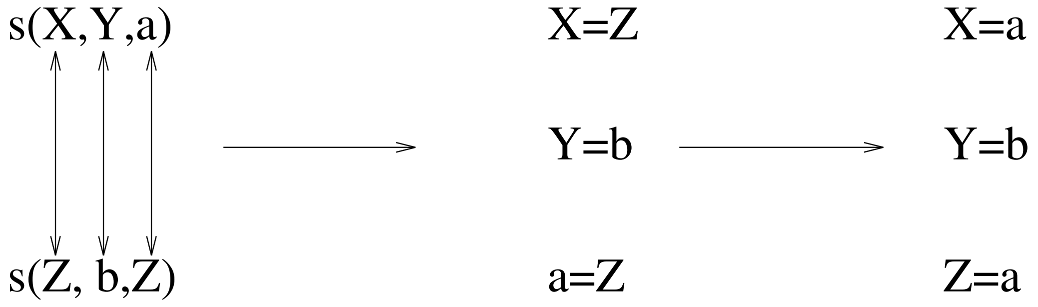

Figura 2.2: Um exemplo de unificação

2.1.3 O controlo do programa

A execução das cláusulas segue uma ordem fixa, determinada pela sua disposição no código do programa. Sempre que para um determinado predicado haja mais que uma cláusula possível, o Prolog executa a que primeiro aparece, tomando nota da existência de alternativas num *ponto-de-escolha*. Quando ao percorrer um determinado caminho a execução falha, o mecanismo de *retrocesso* é accionado, regressando ao ponto da última alternativa e desfazendo todas as atribuições a variáveis feitas no decorrer da alternativa falhada. Se houver mais alternativas a tentar a execução prossegue, caso contrário falha definitivamente. O Prolog dispõe de quatro formas de controlar a execução do programa: o *cut*, a disjunção, o *if-then-else* e a *negation-as-failure*.

O significado do *cut* é o de remover todos os *pontos-de-escolha* desde que se entrou na cláusula corrente até ao ponto onde se encontrou o *cut*. Quer dizer que o caminho que se percorreu é o correcto e que o *retrocesso* não deve explorar as alternativas criadas desde que se entrou nesta cláusula.

A disjunção é uma forma compacta de exprimir uma escolha de entre várias alternativas, e representa-se em Prolog por ';'. É equivalente a escrever cada alternativa como uma cláusula separada. Por exemplo o seguinte predicado `primo/1`:

```
primo(2).  primo(3).  primo(5).  primo(7).
```

pode ser escrito equivalentemente por:

```
primo(X) :- (X=2; X=3; X=5; X=7).
```

A ordem das cláusulas corresponde à ordem das parcelas da disjunção, e o funcionamento do *retrocesso* percorrerá os diferentes valores, (2,3,5,7), em ambas as representações.

O *if-then-else* denota uma escolha entre duas alternativas em que o resultado do *if* determina qual a alternativa a seguir, desprezando a outra com uma semântica idêntica à do *cut*. Representa-se por `->` `;` como ilustra o seguinte exemplo:

```
p(X,R) :- 0 is X mod 2 -> R = par ; R = ímpar.
```

A última forma de controlo do programa é a *negation-as-failure*, representada em Prolog por `\+` (Objectivo). Não se trata de uma verdadeira negação no sentido lógico de negação, daí a escolha de `\+` e não de *not*. O seu significado é de que um *objectivo* negado sucede se o mesmo *objectivo* falha, e falha se o *objectivo* sucede. Por exemplo:

```
par(X) :- \+ impar(X).
```

O predicado `par(X)` sucederá apenas se `impar(X)` falhar. Equivalentemente poderiamos escrever:

```
par(X) :- impar(X), !, fail.
par(X).
```

Ou seja, se `impar(X)` suceder o *cut* elimina o *ponto-de-escolha* de `r(A)` e a execução falha com o `fail`. Se `impar(X)` falha então como o *cut* não chega a ser executado a segunda cláusula de `r(A)` é tentada e sucede.

2.1.4 A sintaxe do Prolog

A figura 2.3 dá uma definição em Prolog da sintaxe de uma cláusula. Por uma razão de simplicidade foram omitidos desta definição os predicados primitivos (*built-ins*) do Prolog. Uma descrição destes predicados pode ser encontrada em [Dia94], de acordo com a sintaxe standard do Prolog descrita em [Cov93].

2.1.5 A semântica do Prolog

Nesta secção descreve-se a forma como um programa Prolog é executado na sua forma mais simples, isto é, sem optimizações introduzidas pela WAM, que modificam a forma de execução, nem detalhes de uma implementação em particular.

```
cláusula(H)         :- cabeça(H).
cláusula((H:-B))    :- cabeça(H), corpo(B).

cabeça(H)  :- termo_objectivo(H).

corpo(G)   :- controlo(G, A, B), corpo(A), corpo(B).
corpo(G)   :- objectivo(G).

objectivo(G)   :- \+controlo(G, _, _), termo_objectivo(G).

controlo((A;B), A, B).
controlo((A,B), A, B).
controlo((A->B), A, B).
controlo(\+(A), A, true).

termo(T)  :- var(T).
termo(T)  :- termo_objectivo(T).

termo_objectivo(T)  :- nonvar(T), functor(T, _, A), args_termo(1, A, T).

args_termo(I,A,_) :- I>A.
args_termo(I,A,T) :- I=<A, arg(I,T,X), termo(X), I1 is I+1, args_termo(I1,A,T).

% Predicados built-in necessários na definição:
functor(T, F, A) :- (O termo T tem functor F e aridade A).
arg(I, T, X) :- (O argumento I do termo T é X).
var(T) :- (O argumento T é uma variável livre).
nonvar(T) :- (O argumento T não é uma variável).
```

Figura 2.3: A sintaxe do Prolog

A execução do Prolog é a de um simples demonstrador de teoremas. O resolvente é uma lista de objectivos (G1,G2,...,Gn). O inquérito Q é o objectivo que começa a execução. Inicialmente atribui-se ao resolvente o inquérito Q. No processo de resolução do demonstrador de teoremas o resolvente é sucessivamente transformado até que:

1. Se torna vazio, caso em que a execução sucede;
2. Não restam hipóteses das cláusulas para testar, caso em que a execução falha;
3. O programa entra num ciclo infinito.

No processo de resolução escolhe-se um objectivo da lista de objectivos do resolvente e unifica-se com uma cláusula. O próximo resolvente obtém-se do corpo dessa cláusula, e assim sucessivamente num processo não determinístico. Para obter eficiência, o Prolog restringe este processo de resolução por dois lados: escolhendo sempre o primeiro objectivo do resolvente, e tentando as cláusulas pela ordem em que aparecem no programa. Se não se consegue unificar o objectivo actual com nenhuma cláusula então o programa faz retrocesso, recuperando um resolvente anterior e a execução continua. Estas restrições tornam o Prolog num fraco demonstrador de teorema, mas útil como linguagem de programação.

2.2 A WAM

Esta secção descreve a WAM, apresentando a sua organização da memória, o estado de execução e o seu conjunto de instruções. No fim dá-se um exemplo do código WAM gerado para um programa Prolog, e um esquema geral de compilação de Prolog para código WAM.

Uma descrição pedagógica, completa e pormenorizada da WAM é dada por [AK91] ou [GLLO85].

2.2.1 Estruturas de dados e organização da memória

Como já foi referido as variáveis do Prolog podem conter qualquer tipo de objecto durante a execução. Para determinar o tipo de um objecto é preciso por isso inspecioná-lo em tempo de execução. Na WAM os termos são representados por um par (etiqueta,valor), representando a etiqueta o tipo do objecto, podendo ser uma variável, um átomo, um inteiro, uma lista ou uma estrutura. O segundo elemento do par é usado com objectivos diferentes consoante o tipo do objecto: tem o valor dos inteiros, o endereço de variáveis e termos compostos, ou um valor único para cada átomo distinto. O endereço das variáveis livres aponta para elas próprias. Quando se unificam duas variáveis coloca-se o endereço de uma a apontar para a outra. Isto leva à criação de cadeias, pelo que por vezes para conhecer o valor de uma variável é necessário percorrer a cadeia até ao fim, isto é, até encontrar ou um objecto não variável ou uma variável livre (em que o endereço aponta para ela própria), numa operação denominada *desreferenciação*.

A figura 2.4 descreve o estado interno da WAM (guardado em registos). A função de cada registo é relativamente clara. O registo HB aponta para o topo da heap na altura do último

`P`	`Contador do programa`
`CP`	`Ponto de continuação`
`E`	`Apontador para o ambiente corrente`
`B`	`ponto-de-escolha mais recente`
`A`	`Topo da pilha local`
`TR`	`Topo da trilha`
`H`	`Topo da heap`
`HB`	`Ponto de backtrack da heap`
`S`	`Apontador para estruturas`
`Mode`	`Flag de modo`
`A1,A2,...`	`Registos de argumentos`
`X1,X2,...`	`Variáveis temporárias`

Figura 2.4: O estado interno da WAM

ponto-de-escolha e é usado na unificação para verificar se uma variável deverá ser incluída na trilha. O registo S é usado na unificação de termos compostos, isto é, com argumentos, e aponta para o argumento a ser unificado. A flag de modo é usada para determinar se a unificação é feita no modo de escrita ou no modo de leitura.

Em relação à memória, esta está dividida em seis àreas (ver figura 2.5): duas pilhas para dados, uma pilha para suportar a unificação, uma pilha para suportar a interacção entre a unificação e o retrocesso, uma àrea de código e uma àrea como tabela de símbolos:

- *A pilha global* ou *heap*. Esta pilha contém listas e estruturas, os termos compostos do Prolog.

- *A pilha local*. Esta pilha contém ambientes e pontos-de-escolha. Os ambientes contêm as variáveis locais de uma cláusula. Os pontos-de-escolha guardam o estado de execução para a tarefa de retrocesso.

- *A trilha*. Esta pilha é usada para guardar os endereços das variáveis instanciadas que têm que ser libertadas no retrocesso. Nem todas as variáveis têm que ser colocadas na trilha. Apenas aquelas cujo endereço na heap ou na pilha local seja anterior ao topo da respectiva pilha restaurado do último ponto-de-escolha.

- *A lista push-down (PDL)*. Esta pilha é usada na unificação de termos compostos.

- *A área de código*. Esta área contém o código compilado do programa.

- *A tabela de símbolos.* Esta área guarda informação acerca dos átomos e estruturas usados no programa.

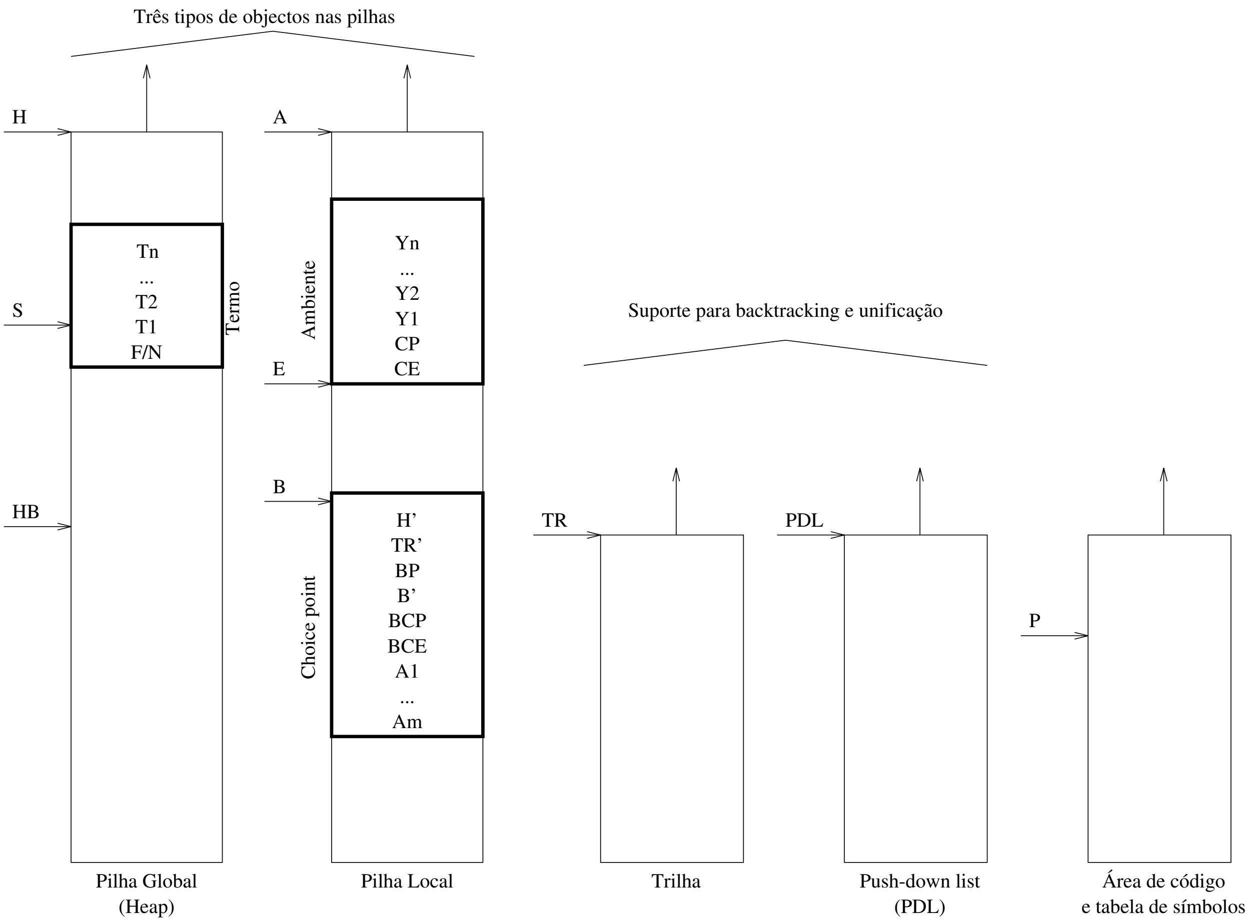

Figura 2.5: Organização da memória na WAM

2.2.2 O conjunto de instruções

A figura 2.6 apresenta o conjunto de instruções da WAM, descrevendo sumariamente o que cada uma destas instruções faz.

Uma variável guardada no ambiente corrente denota-se por Yi. Uma variável guardada num registo denota-se por Ai ou Xi. Um registo usado para passar argumentos denota-se por Ai. Um registo usado apenas internamente dentro de uma cláusula denota-se por Xi. A notação Vi é uma representação abreviada de Xi ou Yi. A notação Ri é uma representação abreviada de Xi ou Ai.

```
      Carregamento dos registos de argumentos (antes de uma chamada)
put_variable Vn, Ri     Cria uma nova variável, colocada em Vn e Ri
put_value Vn, Ri        Move Vn para Ri
put_constant C, Ri      Move a constante C para Ri
put_nil Ri              Move a constante nil para Ri
put_structure F/N, Ri   Cria o functor F/N e coloca-o em Ri
put_list Ri             Cria um apontador para uma lista e coloca-o em Ri

        Unificação com Registos (unificação da cabeça)
get_variable Vn, Ri     Move Ri para Vn
get_value Vn, Ri        Unifica Vn com Ri
get_constant C, Ri      Unifica a constante C com Ri
get_nil Ri              Unifica a constante nil com Ri
get_structure F/N, Ri   Unifica o functor F/N com Ri
get_list Ri             Unifica um apontador para lista com Ri

        Unificação com Argumentos de Estruturas (unificação da cabeça)
unify_variable Vn       Move o próximo argumento da estrutura para Vn
unify_value Vn          Unifica Vn com o próximo argumento da estrutura
unify_constant C        Unifica a constante C com o próximo argumento da estrutura
unify_nil               Unifica a constante nil com o próximo argumento da estrutura
unify_void N            Salta os próximos N argumentos da estrutura

                       Gestão de variáveis unsafe
put_unsafe_value Vn, Ri Move Vn to Ri e globaliza
unify_local_value Vn    Unifica Vn com o próximo argumento da estrutura e globaliza

                               Controlo

call P, N               Chama o predicado P, com N variáveis no ambiente
execute P               Salta para o predicado P
proceed                 Retorna
allocate                Cria um novo ambiente
deallocate              Remove um ambiente

               Selecção de cláusulas (Salto condicional)

switch_on_term V,C,L,S  Salto de 4 alternativas no tipo de A1
switch_on_constant N,T  Salto ditado por uma tabela de hashing T de tamanho N em A1
switch_on_structure N,T Salto ditado por uma tabela de hashing T de tamanho N em A1

              Retrocesso (gestão de pontos-de-escolha).

try_me_else L           Cria um ponto-de-escolha para L, e continua a execução
retry_me_else L         Muda o endereço do ponto-de-escolha para L e continua a execução
trust_me_else fail      Remove o último ponto-de-escolha e continua a execução
try L                   Cria um ponto-de-escolha e salta para L
retry L                 Muda o endereço do ponto-de-escolha e salta para L
trust L                 Remove o último ponto-de-escolha e salta para L
```

Figura 2.6: O conjunto de instruções da WAM

2.2.3 Um exemplo do código WAM de um programa Prolog

A figura 2.7 representa o código Prolog e o código WAM do predicado `append/3` introduzido no capítulo 1. A instrução `switch` salta para a cláusula ou grupo de cláusulas correcto, com base no tipo do primeiro argumento (a WAM só faz indexação no primeiro argumento). As instruções de ponto-de-escolha (`try`) agrupam um conjunto de cláusulas. As instruções `get` fazem a unificação com os argumentos da cabeça, enquanto que as instruções `unify` fazem a unificação com os argumentos das estruturas.

As instruções `unify` comportam-se diferentemente consoante o modo em que se está. Este modo pode ser de *leitura* ou de *escrita* e é guardado numa flag *mode*. É inicializado pelas instruções `get`. Por exemplo, se a instrução `get_list Ai` vê uma variável livre no argumento, inicializa a flag para o modo de escrita. Se vê um argumento lista, então inicializa a flag de modo para leitura.

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
-----------------------------------------------------------------------------

append/3:  switch_on_term V1,C1,C2,fail   Salta se variável, constante,lista
                                          ou estrutura
      V1:  try_me_else V2                 Cria um ponto-de-escolha

      C1:  get_nil A1                     Unifica A1 com nil

           get_value A2,A3                Unifica A2 e A3

           procceed                       Retorna ao local de chamada

      V2:  trust_me_else fail             Remove o ponto-de-escolha

      C2:  get_list A1                    Início da unificação da lista em A1

           unify_variable X4              Unificação da cabeça: move a cabeça
                                          da lista para X4
           unify_variable A1              Unificação da cauda: move a cauda da
                                          lista para A1
           get_list A3                    Início da unificação da lista em A3

           unify_value X4                 Unificação da cabeça: unifica a cabeça
                                          da lista com X4
           unify_variable A3              Unificação da cauda: move a cauda da
                                          lista para A3
           execute append/3               Salta para o início
```

Figura 2.7: O código Prolog e WAM do predicado `append/3`

2.2.4 Esquema geral de compilação de Prolog para WAM

A passagem de Prolog para WAM é simples porque existe uma correspondência clara entre os objectos do código Prolog e as instruções da WAM. A figura 2.8 esquematiza um processo geral de compilação de código Prolog para WAM, sem qualquer pensamento em optimizações.

As cláusulas do predicado `p/3` são compiladas em blocos de código agrupados por instruções `try`. Cada bloco é constituído por uma sequência de instruções `get` que fazem a unificação dos argumentos da cabeça da cláusula, seguida de uma sequência de instruções `put` para dispôrem os argumentos para os objectivos do corpo, e de um `call` para executar o objectivo, a não ser que se trate da última chamada, caso em que o `call` é substituído por um `execute`. No início de cada bloco uma instrução `allocate` cria um ambiente, e no fim uma instrução `deallocate` remove esse ambiente.

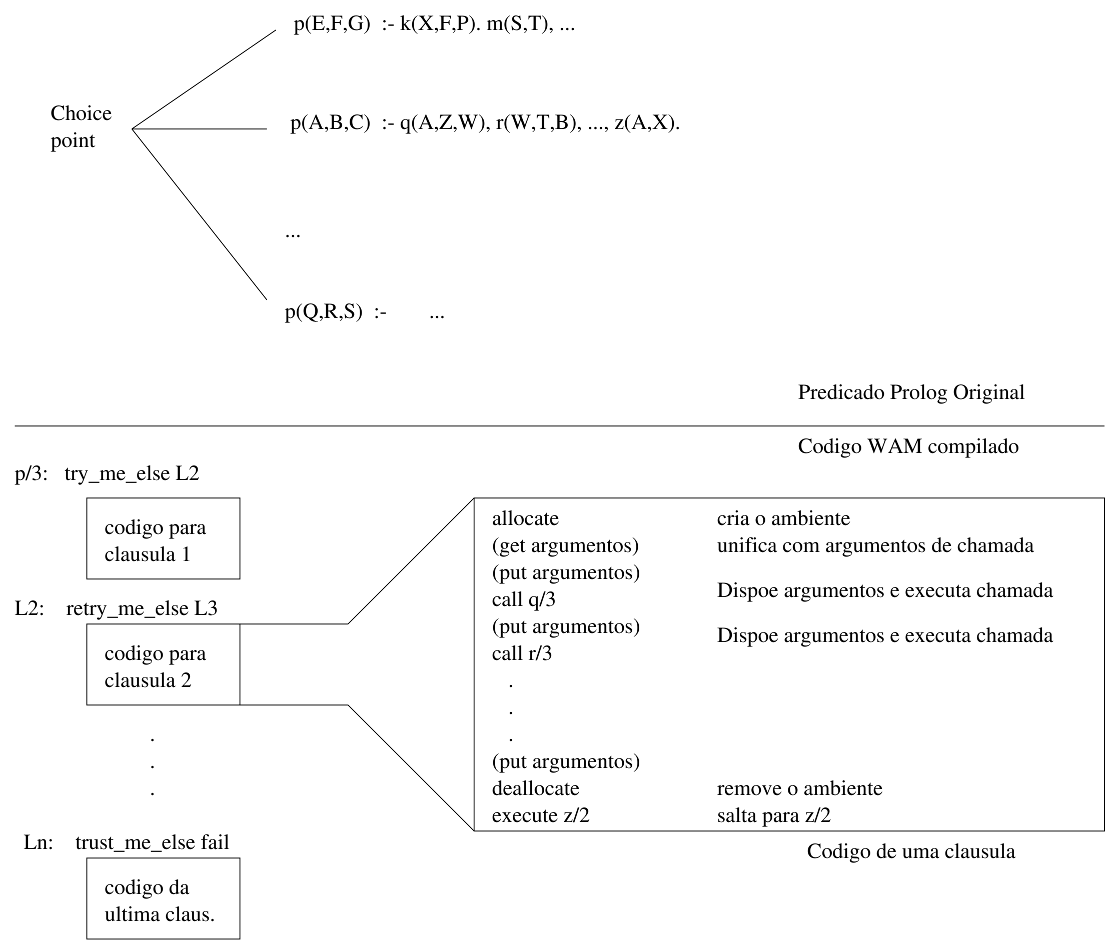

Figura 2.8: Esquema geral de passagem de Prolog para WAM

Capítulo 3

Geração de informação para *unfolding*

Este capítulo descreve a forma como é gerada a informação que servirá de base ao *unfolding*. A primeira, e mais importante, informação necessária para a expansão do código é o código em si, ou seja, o código WAM das cláusulas do programa. Outra informação necessária é a dos tipos dos argumentos dos predicados à saída. Finalmente, um grafo dos predicados do programa é gerado para permitir seleccionar os predicados a expandir.

3.1 Geração de código WAM

Ao invés de se desenvolver um compilador de Prolog para WAM, usou-se um compilador genérico para executar essa tarefa, uma vez que as ideias fundamentais deste trabalho vão executar-se após a geração do código WAM.

Seguidamente descreve-se a estrutura do compilador `wamcc` e a forma como este trabalho foi integrado nesse compilador.

3.1.1 A `wamcc`

A `wamcc` é um compilador de Prolog escrito por Daniel Diaz, que gera um código intermédio WAM e depois o converte para C. Todo o código até à emissão de C está escrito em Prolog, ocupando cerca de 3000 linhas. O facto de não ser um programa excessivamente grande, de estar escrito de uma forma clara e em Prolog, e estar estruturado em fases independentes, levaram à escolha deste compilador como plataforma sobre a qual este trabalho foi construído. A simplicidade deste compilador foi um dos objectivos no seu desenvolvimento, de forma a permitir o seu uso como plataforma experimental para extensões. Para além da extensibilidade, a portabilidade, a eficiência e a modularidade foram os outros requisitos deste sistema.

O código Prolog do compilador `wamcc` é constituído por 10 módulos: `wamcc.pl`, `wamcc0.pl`, `wamcc1.pl`, `wamcc2.pl`, `wamcc3.pl`, `wamcc4.pl`, `wamcc5.pl`, `wamcc6.pl`, `wamcc7.pl` e `wamcc8.pl`. Quando se desenvolve uma grande aplicação é recomendável dividi-la em vários pequenos módulos. Cada módulo tem o seu próprio espaço de nomes de predicados. Apenas os predicados são locais aos módulos, enquanto que os termos são globais. Cada predicado, quer seja um *built-in* ou um predicado definido pelo utilizador pertence a um módulo. Por defeito, um predicado só é visível no módulo onde está definido (predicado *private*) a não ser que seja declarado como *public*. Estes predicados são visíveis em todos os módulos, como é o caso dos *built-ins*.

A estruturação em módulos do compilador, e à excepção do módulo `wamcc` que controla os restantes módulos, é guiada pelas diferentes fases do processo de geração de código. A figura 3.1 representa o fluxo destas fases de compilação.

Como já foi dito, o módulo `wamcc` é o responsável pelo controlo das diversas fases de compilação. É este módulo que trata as opções da linha de comando, que inicializa os ficheiros a gerar e define o ciclo que comporta a execução de todos os outros módulos. A figura 3.2 mostra os dois predicados básicos que definem o ciclo de compilação.

A cada um dos restantes módulos está associada uma fase da compilação. No caso do módulo `wamcc0` faz-se a leitura do ficheiro, colocando as cláusulas do programa em memória, fazendo a verificação da sua correcção sintáctica. No módulo `wamcc1` removem-se determinadas estruturas sintácticas das cláusulas, como *cuts*, *if_then_else* e disjunções, de forma obter um código sintacticamente mais simples, facilitando as próximas fases. No módulo seguinte, `wamcc2`, as cláusulas em memória são colocadas num formato interno. É sobre este formato interno que funcionarão os próximos módulos, `wamcc3` e `wamcc4`. No primeiro faz-se a classificação das variáveis, segundo os princípios descritos na WAM, e no segundo gera-se código WAM para as cláusulas. No módulo `wamcc5` faz-se a atribuição de registos temporários e no módulo `wamcc6` faz-se a indexação do código WAM em dois níveis, primeiro com a indexação com `try_me_else`, `retry_me_else` e `trust_me_else_fail`, e depois a indexação de segundo nível com `switch_on_term`. Os dois últimos módulos fazem a emissão de código WAM e de C, respectivamente.

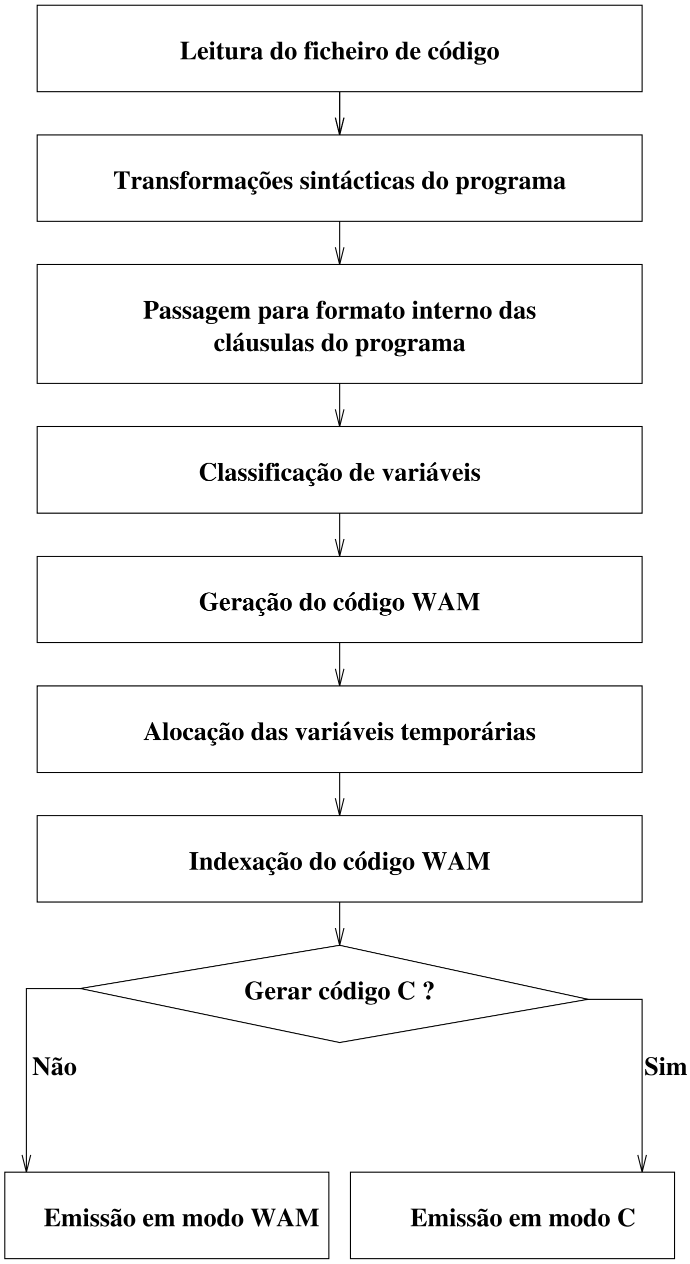

Figura 3.1: As diversas fases da `wamcc`.

Em termos de eficiência, e segundo o autor, a `wamcc` está algures entre o SICStus emulado e o SICStus nativo.

```
compiler(LSrc,Main,LPPublic) :-
        g_read(mode_c,ModeC),
        exec_passes(LSrc,LClComp),
        cree_paquets(LClComp,LPaq),
        exec_indexation(LPaq,LICode),
        (ModeC==t
          ->  c_emmit(LICode)
          ;   wam_emmit(LICode)).


 exec_passes([ClSrc|LSrc],[clcomp(Pred,Arg1,W1)|LClComp]) :-
        sucre_syntaxique(ClSrc, LSrc, ClSrc1,LSrc1),
        format_interne(ClSrc1,Arg1,T,C,NbChunk),
        classif_variables(T,C),
        generation_code(T,C,NbChunk,Pred,W1),
        allocation_varsX(W1),
        exec_passes(LSrc,LClComp).
```

Figura 3.2: Os dois predicados fundamentais do módulo `wamcc`.

3.1.2 Estrutura do compilador implementado e sua integração na `wamcc`

O facto da estruturação da `wamcc` ser baseada em módulos independentes facilitou grandemente o presente trabalho, pois permitiu a utilização de alguns módulos sem ser necessário fazer grandes alterações a esses módulos. Tal é o caso, por exemplo, do módulo de leitura do ficheiro de código, que foi utilizado na íntegra.

Como foi referido nos objectivos, o compilador a implementar baseia a sua eficiência nos quatros princípios descritos, sendo implementados pelo *unfolding*. O *unfolding* é uma expansão do código WAM original para um código WAM complementado com informações sobre o tipo dos argumentos, das variáveis e sobre o modo das estruturas. Essa expansão faz-se naturalmente após a geração do código WAM original. Mas antes de se iniciar a tarefa da expansão do código é preciso recolher determinada informação prévia.

Uma dessas informações é a determinação dos tipos dos argumentos dos predicados à saída. Será essa informação que permitirá melhorar a informação deduzida pelo *unfolding* nas cláusulas com mais do que um *objectivo* no corpo. Isso é feito recorrendo a interpretação abstracta, que opera sobre o formato interno das cláusulas em memória.

Uma outra informação que é necessária antes de se efectuar o *unfolding*, e dado o crescimento exponencial do código em termos de número de argumentos dos predicados e número de elementos do domínio de interpretação abstracta, é um grafo dos predicados do programa a compilar, que permita seleccionar determinados predicados de acordo com certos critérios. Também aqui se opera sobre o formato interno das cláusulas em memória.

Tendo esta informação reunida pode-se começar a expandir o código WAM gerado pela `wamcc`. Esta expansão vai gerando informação sobre os argumentos dos predicados e modo das estruturas, e as chamadas a predicados são complementadas com esta informação. Uma vez que se tem informação sobre os argumentos, pode-se fazer uma indexação do código WAM bem mais inteligente que a da `wamcc`, que faz essa indexação sempre sobre o primeiro argumento. Faz-se um estudo sobre o melhor argumento para indexar o código, indexando as cláusulas sobre esse argumento. Este estudo sobre os argumentos permite também controlar melhor o crescimento do código gerado pelo *unfolding*, pois permite detectar casos particulares de especialização do código de um predicado sem interesse, e substituí-los pelo caso geral. Finalmente obtemos um código WAM expandido e indexado, que se converte para C.

A figura 3.3 representa a estrutura do compilador implementado, para ser comparada com a da estrutura da `wamcc` da figura 3.1 .

3.2 Uso de interpretação abstracta

Esta secção descreve a aplicação de interpretação abstracta para inferir os tipos dos argumentos dos predicados à saída, ou seja, depois de ter sido efectuada a chamada.

Muito de tem escrito acerca da aplicação de interpretação abstracta a programas lógicos, apesar de não haver muitas implementações desses algoritmos em compiladores, prevalecendo a teoria sobre a prática. Uma explicação formal dos métodos de interpretação abstracta, e dos seus fundamentos matemáticos, é dada por [CC92].

3.2.1 Transformação do programa original.

Como em qualquer implementação de interpretação abstracta, também esta aplicação se baseia numa transformação do programa inicial numa representação mais simples, em

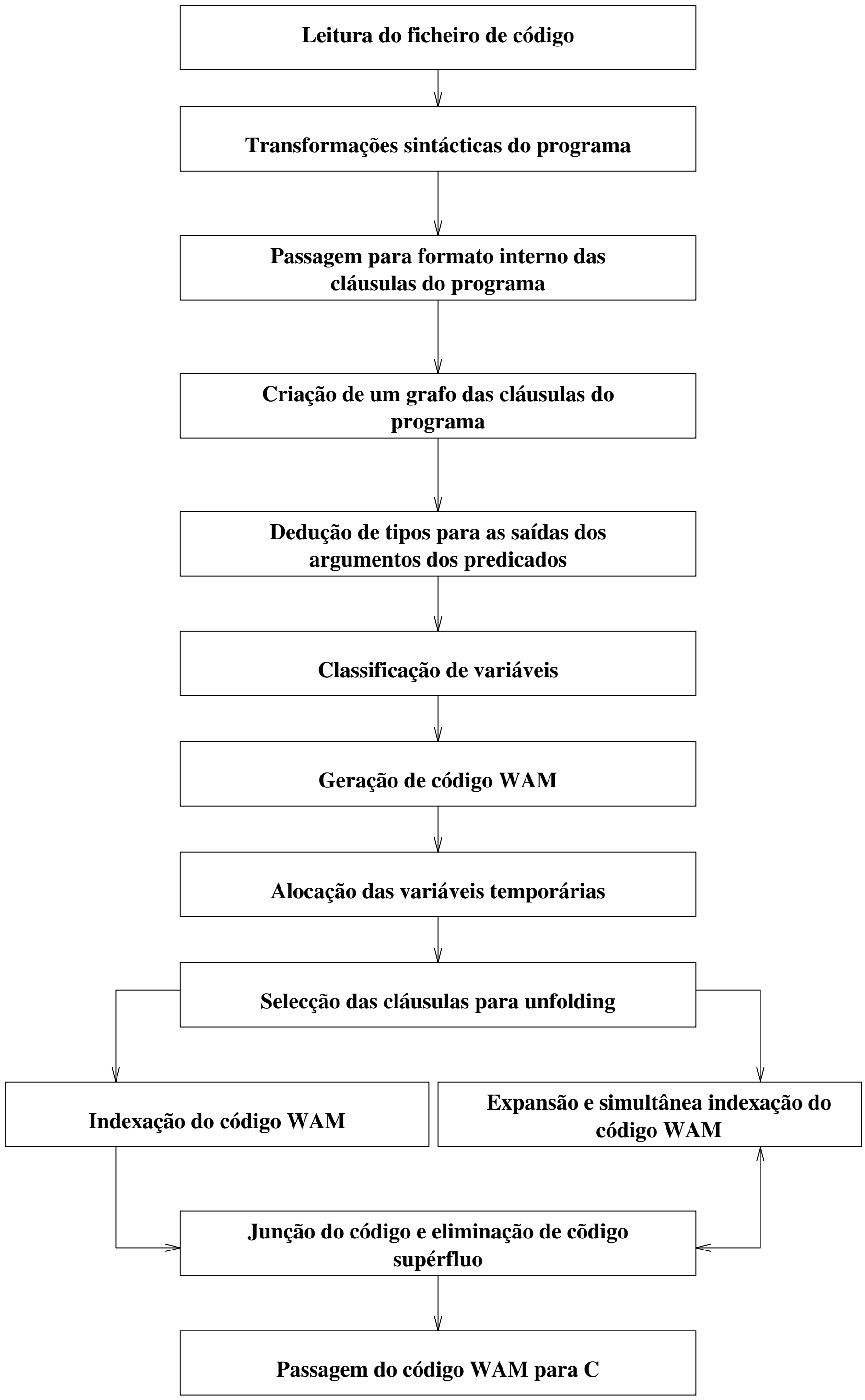

Figura 3.3: A estrutura do compilador implementado.

que prevaleçam as propriedades importantes para o que se pretende obter. Neste caso, pretendemos obter informação sobre os argumentos dos predicados, mais concretamente sobre estes argumentos à saída do predicado. Dado o número extremamente elevado, potencialmente infinito, de possíveis valores que os argumentos de um predicado podem tomar, torna-se muito difícil obter informação relevante acerca destes argumentos, se o programa estiver na sua forma original. Daí que o algoritmo de análise não opere sobre o programa original, mais sim sobre uma representação deste onde todos os possíveis valores dos argumentos são mapeados num reduzido número de valores, que constituem o domínio de análise.

A transformação do programa para análise, é portanto muito simples. Para todos os predicados, e para todas as cláusulas de um predicado, obtemos uma representação equivalente com os valores dos argumentos substituídos pelo valor correspondente ditado pela função de mapeamento. Considerando o seguinte predicado `fib/2` com as suas três cláusulas, e tendo um domínio de análise onde *int* represente os inteiros, obteríamos a seguinte representação para aplicação do algoritmo de análise:

Programa original:

```
fib(1,1).
fib(2,1).
fib(N,R) :- N>2, N1 is N-1, N2 is N-2,
            fib(N1,R1),
            fib(N2,R2),
            R is R1+R2.
```

Programa transformado:

```
fib(int,int).
fib(int,int).
fib(N,R) :- N>int, N1 is N-int, N2 is N-int,
            fib(N1,R1),
            fib(N2,R2),
            R is R1+R2.
```

Todos os diferentes inteiros que ocorriam no programa original foram mapeados num único elemento, *int*, do domínio de análise. As variáveis não sofrem qualquer transformação, pois o seu nome é importante, pois é através dele que a informação dos tipos é propagada.

3.2.2 O domínio de análise.

A análise de tipos em Prolog difere da das linguagens com tipos pré-declarados, na medida em que, ao invés de fazer a verificação da correcção destes tipos, tenta deduzi-los. A informação mais importante que pode ser deduzida acerca de um determinado argumento é se ele é usado como entrada ou como saída, isto é, o *modo* desse argumento. Para além disso, interessa saber o conjunto de valores que esse argumento pode tomar, ou seja, o seu *tipo*. A definição do domínio de análise é guiada por duas linhas concorrentes. Por um lado interessa que o número de valores do domínio seja reduzido, de forma a facilitar o trabalho de análise, mantendo essa análise rápida e pouco complicada. Por outro, para que se tire o máximo partido da análise abstracta, interessa que esse domínio seja bem mais numeroso. Nos extremos de cada linha directora obteríamos por um lado um domínio vazio e por outro um domínio que mantivesse o programa original na mesma forma. Daí que o domínio resultante seja um compromisso entre estes dois aspectos.

A figura 3.4 representa o domínio de análise escolhido. Neste domínio, *?* representa todos os valores possíveis de ocorrerem num argumento, *struct* representa as estruturas ou termos compostos, *list* representa a estrutura de aridade dois cujo functor é .,*const* representa as estruturas de aridade zero, *nil* representa a lista vazia, `[]`, e *int* representa os números inteiros.

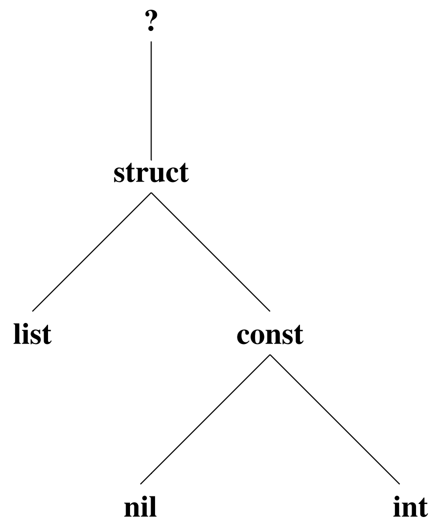

Figura 3.4: O domínio de análise.

A estrutura especialmente simples deste domínio de interpretação, bastante mais simples do que a usada em outros sistemas que também usam interpretação abstracta, como o

Aquarius de Van Roy e o Parma [Tay91] de Taylor, não se deve somente, nem sequer principalmente, à necessidade de manter a análise rápida e pouco complicada, uma vez que poderiam ser acrescentados mais valores a este domínio sem que a análise feita se tornasse demasiado pesada. De facto, e durante a evolução da implementação do analisador, foram usados domínios diferentes deste, onde se usavam tipos como *ground+struct* e *ground+list*, e fizeram-se experiências com o tipo *deref*, que garantia que um determinado argumento era acessível sem necessidade de percorrer uma cadeia de apontadores. A razão principal para manter o domínio de análise o mais pequeno possível tem haver com razões que se tornarão claras mais à frente, quando falarmos do *unfolding*, de forma a controlar o crescimento exponencial do código gerado.

3.2.3 Um exemplo simples para dedução do tipo *int*.

Este exemplo pretende dar uma ideia de como o algoritmo de análise vai funcionar, deixando para a próxima secção a descrição detalhada do mesmo.

Relembremos o predicado `fib/2` referido atrás:

```
fib(1,1).
fib(2,1).
fib(N,R) :- N>2, N1 is N-1, N2 is N-2,
            fib(N1,R1),
            fib(N2,R2),
            R is R1+R2.
```

Dos primeiros dois factos do predicado o analisador conclui que, à saída, ambos os argumentos deste predicado são do tipo *int*. Para a terceira cláusula, e começando pelo corpo, observa-se que este é constituído por seis objectivos, sendo quatro *built-ins* e os outros dois definidos pelo programador. Para os *built-ins* existem declarações do tipo dos seus argumentos à saída. Para os outros dois, temos a informação deduzida com base nas duas primeiras cláusulas. Assim, com base na informação até aí gerada, conclui-se que `N1` é do tipo *int*, `R1` é também do tipo *int*, e o mesmo para `N2` e `R2` do outro *objectivo*. Como `R` é *int+int* é também do tipo *int*. Das declarações dos *built-ins* conclui-se que N é também do tipo *int*. Finalmente, propaga-se esta informação para a cabeça, concluindo que também nesta cláusula ambos os argumentos são do tipo *int* à saída. Tendo todas as cláusulas analisadas, conclui-se que os valores dos argumentos do predicado `fib/2` à saída são do tipo *int*. Se nesta última cláusula se concluísse que os argumentos eram algo que fosse diferente de (*int*, *int*), teríamos que repetir a análise da última cláusula, pois havia que usar a nova informação nos dois objectivos de chamada ao predicado `fib/2`, até atingirmos um ponto fixo.

3.2.4 O algoritmo de análise.

O algoritmo usa uma estratégia *bottom-up*, começando pelos factos e percorrendo as cláusulas no sentido inverso àquele em que o Prolog as executa, isto é, primeiro pelo corpo e só depois propagando a informação para a cabeça da cláusula. A análise dos factos, ou cláusulas sem corpo, é absolutamente linear, limitando-se o algoritmo a colher informação sobre os argumentos, mapeando os objectos nos argumentos do predicado em elementos do domínio de análise. Se ainda não existia informação sobre os argumentos do predicado da cláusula em causa, guarda-se a informação gerada. Se já existia, reune-se a nova informação sobre os argumentos, escolhendo para cada argumento o menor valor do domínio de análise que cubra os valores encontrados.

Para as restantes cláusulas, isto é, aquelas que têm corpo, se houver informação acerca dos argumentos de todos os *objectivos* que constituem o seu corpo, e se houve alguma alteração nos valores dos argumentos desses *objectivos* desde que a cláusula foi analisada pela última vez, substituem-se as variáveis que ocorrem nos argumentos dos *objectivos* do corpo de acordo com a informação disponível, intersectando os possíveis valores para uma mesma variável, escolhendo o valor mais pequeno do domínio de análise. A análise dos *objectivos* do corpo não pode ser da esquerda para a direita, numa só passagem, uma vez que determinada informação pode só estar disponível no fim da cláusula, pelo que é necessário propagar para trás a informação deduzida no fim da cláusula. Finalmente, propaga-se essa informação para a cabeça, actualizando a informação relativa ao predicado da cláusula em causa. O processo repete-se enquanto houver alterações nos argumentos da cabeça das cláusulas, escolhendo as cláusulas influenciadas por estas alterações para repetir a análise, até que não haja mais alterações e se atinja um ponto fixo. Se as condições correctas forem respeitadas (isto é, a actualização de tipos é monótona e a informação sobre esses tipos é propagada correctamente) o ponto fixo é o *ponto fixo mínimo* e os tipos resultantes dão informação correcta acerca do programa original.

A figuras 3.5 dá uma descrição detalhada em pseudo-código do algoritmo.

```
Função análise(Cláusulas):
Cláusulas: conjunto de cláusulas do programa;
início
    Recolhe informação sobre os factos;
    Marca esses factos como modificados;
    Recolhe informação sobre os builtins;
    Marca esses builtins como modificados;
    Recolhe informação sobre predicados externos;
    Marca esses predicados como modificados;
    Enquanto pelo_menos_um(predicado) foi modificado
    início
        selecciona_cláusulas_a_analisar(Cláusulas,Cláusulas_A_Analisar);
        analisa_cláusulas(Cláusulas_A_Analisar);
    fim
fim

Função selecciona_cláusulas_a_analisar(Cláusulas,Cláusulas_A_Analisar):
Cláusulas: conjunto de cláusulas do programa;
Cláusulas_A_Analisar: conjunto de cláusulas;
início
    Para todas as cláusulas com corpo
    início
        Cláusula := cabeça :- objectivo1,...,objectivoN;
        Se (objectivo1,...,objectivoN têm tipos conhecidos
            e  pelo_menos_um(objectivo1,...,objectivoN) foi modificado)
             acrescenta_cláusula(Cláusula,Cláusulas_A_Analisar);
    fim
 fim

Função analisa_cláusulas(Cláusulas_A_Analisar):
Cláusulas_A_Analisar: conjunto de cláusulas;
início
    Para cada Cláusula em Cláusulas_A_Analisar
    início
        Cláusula : = cabeça :- corpo;
        analisa_cláusula(Cláusula);
        Se (actualiza_args_pred(cabeça) se modifica) marca cabeça modificado;
    fim
fim

Função analisa_cláusula(Cláusula):
Cláusula: uma cláusula;
início
    Cláusula : = cabeça :- objectivo1,...,objectivoN;
    Para cada objectivo em (objectivo1,...,objectivoN)
    início
        se  objectivo_de_unificação(Objectivo) unifica_tipos_args(Objectivo);
        senão atribui_tipos_às_vars(Objectivo);
    fim
    propaga_informação_para_trás_no_corpo();
    propaga_informação_para_cabeça(head);
fim
```

Figura 3.5: O algoritmo de análise.

```
incl_2([],C,[C]).
incl_2([A|E],C,D) :- incl_3(C,A,E,D).

incl_3(C,A,E,[A|E]) :- incl(E,C,D).
```

Figura 3.6: Um programa exemplo para análise

3.2.5 Análise de um *objectivo*.

Quando um *objectivo* é encontrado ao percorrer o corpo de uma cláusula três situações são possíveis:

1. Trata-se de *objectivo* definido no programa;
2. Trata-se de um *objectivo* não definido no programa, um built-in ou um predicado de library;
3. Trata-se de um *objectivo* de unificação.

Na primeira situação, se durante os passos anteriores da análise se deduziu que os argumentos do predicado são de determinado tipo, atribui-se às variáveis presentes nos argumentos o tipo correspondente, caso contrário abandona-se a análise da cláusula, por ainda não se a poder fechar. O segundo caso é idêntico ao primeiro, excepto que os tipos dos seus argumentos não foram deduzidos pela análise do programa, mas foram recolhidos de uma declaração prévia e, portanto, ao contrário dos primeiros, não sofrem alterações durante a análise. No caso de um *objectivo* de unificação unificam-se as variáveis dos argumentos, de forma a que as variáveis unificadas apontem para o menor valor em termos do domínio de análise.

3.2.6 Um exemplo de análise.

Os predicados `incl_2/3` e `incl_3/4` da figura 3.6 foram extraídos de uma definição de inclusão de conjuntos.

São necessários três passos na análise até se atingir o ponto fixo. Pela análise da primeira cláusula conclui-se que os seus argumentos são (*nil,any,list*). Neste primeiro passo a segunda cláusula não pode ser analisada pois não são conhecidos os tipos dos argumentos do *objectivo* do seu corpo, e salta-se para a terceira cláusula. Com base na informação deduzida sabe-se que `E` será do tipo *nil*, `C` do tipo *any* e `D` do tipo *list*. Da propagação desta informação para a cabeça resulta que os argumentos de `incl_3/4` são (*any*, *any*,*nil*,*list*).

No segundo passo já se tem informação sobre o *objectivo* no corpo da segunda cláusula, concluindo-se que `C` é do tipo *any*, `A` é do tipo *any*, `E` é do tipo *nil* e `D` é do tipo *list*. Propagando para a cabeça vem que os argumentos nesta cláusula serão (*list*, *any*,*list*). Esta informação será reunida à informação deduzida para a primeira cláusula do mesmo predicado, no passo 1: (*nil*,*any*,*list*). Como resultado temos (*struct*, *any*,*list*), modificando a informação anterior para este predicado `incl_2/3`, o que obriga a voltar a analizar a terceira cláusula. Agora, nesta cláusula, `E` é do tipo *struct*, `C` é do tipo *any* e `D` é do tipo *list*. Propagando para a cabeça vem que os tipos dos argumentos de `incl_3/4` são (*any*,*any*,*struct*,*list*), modificando e generalizando a dedução anterior para este predicado, obrigando a um terceiro passo.

No terceiro passo voltamos a analisar a segunda cláusula, e voltamos a concluir que os seus argumentos à saída serão do tipo (*list*,*any*,*list*), atingindo o ponto fixo e terminando a análise.

A tabela 3.1 descreve a mudança dos argumentos dos predicados durante os passos da análise.

incl_2(A,B,C)			incl_3(A,B,C,D)			
A	B	C	A	B	C	D
nil	*any*	*list*	*any*	*any*	*nil*	*list*
struct	*any*	*list*	*any*	*any*	*struct*	*list*
struct	*any*	*list*	*any*	*any*	*struct*	*list*

Tabela 3.1: Evolução dos tipos durante a análise

3.3 Determinação do grafo de chamadas dos predicados de um programa.

Pretende-se obter uma representação das cláusulas de um programa que reflicta o fluxo das chamadas entre os predicados. Representando as cláusulas do programa por um grafo, será fácil detectar ciclos recursivos e outras estruturas que permitirão seleccionar determinados predicados, com um objectivo que se explicará mais à frente.

Uma tradução bastante clara das cláusulas de um programa para um grafo, é fazer corresponder a cada cabeça de cláusula um vértice do grafo, sendo os *objectivos* que aparecem no corpo os ramos desse grafo, que ligam o vértice da cabeça aos vértices respectivos da ou das cláusulas que representam.

Assim, considere-se o seguinte programa:

```
nrev([],[]).
nrev([H|T],R) :- nrev(T,M), append(M,[H],R).
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

A representação do grafo que o traduz é a seguinte:

```
vertice(nrev/2, 1).
vertice(nrev/2,2).    ramo(2,1,1). ramo(2,1,2). ramo(2,2,3). ramo(2,2,4).
vertice(append/3,3).
vertice(append/3,4).  ramo(4,1,3). ramo(4,1,4).
```

O segundo argumento de `ramo( )` traduz o número do *chunk* do corpo da cláusula, onde foi encontrado o *objectivo*. A figura 3.7 dá uma representação gráfica do grafo descrito.

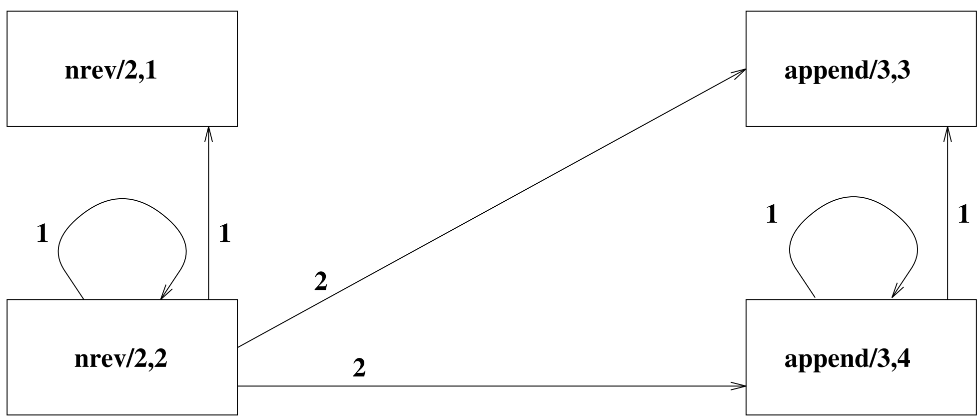

Figura 3.7: O grafo do `nrev`.

O critério de selecção de predicados para *unfolding* é escolher os predicados que possuem cláusulas recursivas, isto é, que apontam para elas próprias, como é o caso, neste exemplo, do predicado `nrev` e do predicado `append`. Este critério pode ser estendido aos ciclos de tamanho dois, isto é, aqueles predicados que apontam para eles próprios percorrendo dois ramos do grafo. Estes predicados definem ciclos, e portanto é mais provável que a execução do programa passe mais tempo nestes predicados. É claro que tal pode não acontecer, mas um critério objectivo só pode ser conseguido executando verdadeiramente o programa, fazendo o seu *profiling*, e então concluindo quais os melhores predicados a expandir.

Capítulo 4

Expansão do código WAM - o *unfolding*

Neste capítulo explica-se como é usada a informação referida no capítulo anterior, descrevendo a forma como o código WAM é expandido, como é feita a sua indexação, e explicando as implicações destas optimizações na execução eficiente do programa. No fim explica-se detalhadamente um exemplo de *unfolding* de um programa Prolog.

4.1 As estruturas de dados usadas pelo *unfolding*

Após se ter reunido a informação que foi descrita no capítulo anterior, estamos em condições de iniciar a expansão do código WAM.

Neste momento existem três estruturas de dados importantes em memória:

1. O código WAM gerado pela `wamcc`, com as cláusulas de cada predicado agrupadas.
2. Uma lista com os modos de saída inferidos para cada predicado (e.g. `fib/2(int,int)`).
3. Uma lista com a indicação de quais os predicados a expandir.

Destas três estruturas a terceira vai dividir a primeira separando os predicados para os quais se vai fazer *unfolding* dos restantes. Para os primeiros é feita uma indexação tradicional, a mesma que a `wamcc` fazia para todos os predicados. Terminada a execução do *unfolding* este código indexado é agrupado ao código expandido e também indexado que resultou da execução do *unfolding*. A segunda estrutura de dados é usada durante a expansão dos predicados, para actualizar a informação acerca das variáveis depois de uma instrução de `call`.

4.2 O funcionamento do *unfolding*

Nesta secção explica-se o funcionamento do *unfolding* nas suas duas vertentes, a expansão de um predicado em vários predicados especializados e a especialização de instruções gerais em instruções especializadas num determinado tipo de argumentos.

4.2.1 Criação de novos predicados

Quando iniciamos o *unfolding* de determinado predicado não sabemos nada acerca dos seus argumentos à entrada. É quando percorremos o código WAM de uma das suas cláusulas que vamos recolhendo esta informação. Basta olhar para as instruções WAM que codificam a cláusula, como `get_list(ArgNo)`, ou `put_integer(ArgNo)`, ou `get_constant(ArgNo)`, etc. Em cada um dos casos concluiríamos que o argumento número ArgNo era do tipo *list*, *int* ou *const*, respectivamente.

A nossa ideia básica é que a informação implícita em cada instrução WAM, que se traduz em variantes de instruções do género `xxxx_var`, `xxxx_value`, `xxxx_list`, `xxxx_const`, etc, permite simpificar não só a própria instrução (razão da sua implementação na WAM), mas também as próximas instruções WAM de codificação da cláusula.

Assim, vamos guardando a informação que as instruções WAM fornecem, quer acerca de variáveis globais e locais, quer sobre o *modo* das instruções (leitura ou escrita).

Percorrendo o código WAM das cláusulas, e se estas tiverem corpo, encontraremos as instruções de `call(P/N)` ou `execute(P/N)`. Se `P/N` representar um predicado que também se encontra na terceira estrutura descrita na secção anterior, ou seja, um predicado para o qual também se fará *unfolding*, terá todo o interesse usar a informacão recolhida até então acerca de variáveis globais e temporárias e modo de estruturas. Esta informação permite simplificar a execução deste *objectivo*, pois ao invés de o executarmos na sua forma geral, i.e. quando não sabemos nada acerca dos seus argumentos de chamada, podemos executar uma variante especializada para o tipo de argumentos com que sabemos ser executada esta chamada.

Esta é uma das ideias fundamentais do *unfolding*: expandir o código WAM das cláusulas de um predicado em código WAM de cláusulas especializadas para determinado tipo de argumentos de chamada.

Assim, quando encontramos um *objectivo* no corpo de uma cláusula substituímos esta chamada a um determinado predicado por uma chamada especializada, de acordo com a informação existente acerca do modo das estruturas e das variáveis. Por exemplo, no predicado `fib/2` ao encontrar a instrução WAM `call(fib/2)`, e se tivéssemos concluído das instruções WAM anteriores (`put_integer`, put_variable) que os seus argumentos eram do tipo, respectivamente, *int* e *var*, o *unfolding* da instrução `call(fib/2)` resultaria em `call(fib(int,var))`.

Se ainda não temos código expandido para a chamada resultante, iremos criar o código para esse par (predicado,argumentos). Começamos por procurar o código WAM geral das cláusulas do predicado. Este código será expandido, usando o conhecimento dos tipos dos seus argumentos de chamada.

O *unfolding* de um determinado predicado necessita da seguinte informação:

- O código WAM das cláusulas do predicado
- O modo da chamada, isto é, os tipos dos seus argumentos de chamada (no predicado que primeiro se expande estes argumentos são de tipo desconhecido, *?*)
- O código WAM das cláusulas dos restantes predicados, para passar ao *unfolding* dos *objectivos* do predicado corrente
- Uma lista dos predicados a expandir, para saber se os *objectivos* do predicado corrente são para expandir
- Os modos de saída dos predicados, para actualizar os tipos das variáveis depois de um `call`

Na expansão de um predicado começa-se por guardar a informação transmitida nos seus argumentos de chamada. É isso que faz o predicado `init_status/2` da figura 4.1. Seguidamente escolhe-se o argumento de indexação, de uma forma que se explicará mais à frente. O código WAM do predicado é indexado e faz-se o *unfolding* das instruções resultantes.

A figura 4.1 mostra o código Prolog para *unfolding* de um predicado, que se explicou nos parágrafos anteriores.

```
unfold(PCode,Mode,Code,PrToUnf,OutMs,ECode0,ECode) :-
   init_status(Mode,Status0),
   select_arg_to_index(Mode,PCode,Arg),
   indexation(Arg,PCode,PCode1),
   unfold(PCode1,Status0,Code,PrToUnf,OutMs,UPCode,[Mode:UPCode|ECode0],ECode).
```

Figura 4.1: *Unfolding* de um predicado

4.2.2 Complementação das instruções WAM

Para além da complementação das chamadas a predicados com a informação do tipo dos seus argumentos (instruções `call` e `execute`), também todas as outras instruções WAM são complementadas com esta informação, tirando grande partido disso. Por exemplo, se ao percorrer o código WAM de uma cláusula concluíssemos que `ArgNo` era do tipo *nil*, e a seguir encontrássemos a instrução `get_nil(ArgNo)`, a complementação desta instrução com informação acerca de `ArgNo` resultaria na instrução `get_nil(ArgNo,nil)`, cuja execução é muito mais simples do que quando não há qualquer informação sobre o argumento. Neste caso temos que fazer a desreferenciação do argumento `ArgNo`, fazer um `switch` para o caso de se encontrar na `heap` uma variável ou outro valor, neste último caso comparar esse valor com o átomo *nil*, e, finalmente incrementar o contador do programa.

No caso de `get_nil(ArgNo,nil)` apenas temos que incrementar o contador do programa.

Desta forma simples o *unfolding* altera profundamente o conjunto de instruções da WAM, dotando-a de um conjunto bem mais vasto, e de uma maior simplicidade de cada uma destas instruções, reduzindo grandemente o número de operações e de testes que uma instrução especializada para determinado tipo de argumentos tem que realizar em comparação com a instrução geral.

4.3 Um exemplo de *unfolding* de uma instrução WAM

O código WAM das cláusulas de um programa vai sendo expandido instrução a instrução. O que se faz ao fazer *unfolding* de uma instrução é, fundamentalmente, guardar informação acerca dos seus argumentos ou modo da estrutura, propagar essa informação para a expansão das instruções seguintes, e no regresso combinar as instruções WAM complementadas com a informação deduzida.

A figura 4.2 representa o código Prolog para expansão da instrução `get_list(I)`.

```
unfold([get_list(I)|CCode],Status0,Code,UPCode,ECode0,ECode) :-
    get_a_mode(I,ModeI,Status0),
    ( ModeI == list ->
        set_s_mode(read,Status0,Status1),
        unfold(CCode,Status1,Code,UPCode1,ECode0,ECode),
        combine(get_list(I,list),UPCode1,UPCode);
    ( ModeI == var; ModeI == ovar) ->
        set_s_mode(write,Status0,Status1),
        unfold(CCode,Status1,Code,UPCode1,ECode0,ECode),
        combine(get_list(I,ModeI),UPCode1,UPCode);
    ModeI == ?  ->
        set_s_mode(read,Status0,Status1),
        unfold(CCode,Status1,Code,UPCode1,ECode0,ECode1),
        set_s_mode(write,Status0,Status2),
        unfold(CCode,Status2,Code,UPCode2,ECode1,ECode),
        UPCode = [switch_on_mode(I,?,UPCode1,UPCode2)]
    ).
```

Figura 4.2: Código Prolog para *unfolding* da instrução `get_list`

Começa-se por procurar na informação guardada o tipo do argumento de `get_list(I)`. Se este argumento for *list* conclui-se que o modo da estrutura é *read*, e propaga-se esta informação para as instruções seguintes. O resultado da expansão destas instruções acrescenta-se à instrução `get_list(I,list)`. Se o argumento for *var* então conclui-se que a lista está no modo *write*, e o procedimento é o mesmo. Se não soubermos nada sobre o tipo do argumento, expandem-se ambas as hipóteses de modo, *read* ou *write*, e combinam-se numa instrução `switch_on_mode(I,?,UPCode1,UPCode2)`.

4.4 Chamadas a um predicado

A instrução WAM que traduz uma chamada a um predicado pode ser um `execute` ou um `call`, consoante a cláusula tem um ou mais *objectivos* no seu corpo. Em qualquer dos casos duas situações podem ocorrer: o predicado chamado consta da lista de predicados a expandir ou não consta. No segundo caso, o *unfolding* termina por aí visto não se querer expandir o predicado. No outro caso, constrói-se o modo dos argumentos de chamada do predicado com base na informação recolhida até então, e expande-se o par (predicado,argumentos).

Em qualquer das situações descritas acima existem diferenças consoante a chamada ao predicado é feita com `call` ou com `execute`. Se for feita com `execute` nada há a referir, pois um `execute` termina o código da cláusula. Já se se tratar de um `call` a situação é diferente, pois há que alterar a informação relativa às variáveis, aquelas que não se encontravam instanciadas na altura de chamada do predicado (tipo *var*). É por essa razão que se usou interpretação abstracta para deduzir os tipos dos argumentos dos predicados à saída. Assim, após o `call` há que fazer a actualização das variáveis de acordo com a informação deduzida na análise descrita no capítulo 3. Se esta análise não fosse feita nada poderíamos concluir acerca dos argumentos não instanciados do predicado chamado, tendo que promovê-los a *?*, ao invés da melhoria da informação sobre estes argumentos que a análise permite.

4.5 Uma nota sobre o tipo *var*

Existem dois elementos do domínio que representam uma variável: o tipo *var* e o tipo *ovar*. O tipo *var* representa as variáveis para as quais ainda não há referências, ou seja, são as variáveis que aparecem pela primeira vez, e são equivalentes às variáveis *uninitialized* tais como descritas por Van Roy. Para estas variáveis não é necessária a operação de desreferenciação. O tipo *ovar* representa as outras variáveis, que já apareceram anteriormente na cláusula, bem como as variáveis que se devem incluir na trilha.

A razão porque não se referiram estes dois elementos na descrição do domínio de análise é porque estes elementos apenas fazem sentido como tipos para as entradas dos predicados, porque não se podem tirar conclusões sobre se à saída um argumento tem tipo *var*, concluindo sim que o seu tipo será *?*.

4.6 Implicações do *unfolding* na eficiência

Tendo percebido o funcionamento do *unfolding*, é fácil de ver as consequências que provoca nos quatros princípios de eficiência descritos no início deste trabalho. Relembremos estes princípios:

1. Redução da granularidade das instruções.
2. Exploração do determinismo.
3. Especialização da unificação.
4. Análise global do programa.

A redução da granularidade das instruções é conseguida não através da criação de novas instruções e a consequente definição de uma nova máquina abstracta, mas complementando as instruções WAM com a informação que vai sendo deduzida.

A exploração do determinismo é fruto da substituição das chamadas gerais a predicados por chamadas especializadas, e por uma indexação mais inteligente de que se falará seguidamente.

O facto de se ter informação acerca das variáveis e do modo das estruturas permite simplificar grandemente a unificação.

O uso de interpretação abstracta como suporte dos princípios acima foi reduzido a um mínimo, traduzindo-se num algoritmo simples que apenas tem que deduzir os tipos dos argumentos à saída dos predicados.

4.7 Controlo do crescimento do código provocado pelo *unfolding*

O número de predicados especializados em que um predicado pode ser expandido é fruto de dois factores: o número de argumentos do predicado e o número de elementos do domínio. Se bem que seja necessário um programa muito especial para um determinado argumento poder tomar todos os possíveis valores do domínio de análise, tal pode acontecer. No pior caso possível, todos os argumentos de um determinado predicado podem tomar todos os possíveis valores do domínio de análise. Neste caso, o número de predicados especializados em que um predicado é expandido é o número de elementos do domínio elevado ao número de argumentos do predicado. Se o predicado tiver aridade 3 e o domínio de análise tiver 6 elementos, poderão ser gerados 216 predicados especializados. É claro que esta situação é praticamente impossível de ocorrer, visto que tendo em conta o nosso domínio de análise, um argumento poderá tomar no máximo 2 ou três valores desse domínio.

No entanto torna-se claro que se deve tentar limitar sempre que possível a expansão provocada pelo *unfolding*.

Este controlo é feito em três vertentes:

1. Limitação dos predicados a expandir
2. Limitação do número de elementos do domínio de análise
3. Substituição de entradas especializadas (pred,argumentos) sem interesse, pelo caso geral

A limitação dos predicados a expandir é um ponto crucial na estratégia do *unfolding*. É claro que em programas de dimensão real a expansão da totalidade dos seus predicados é perfeitamente inviável. Por outro lado, a eficiência é obtida pela expansão dos predicados, pelo que esta limitação dos predicados a expandir não deve incidir sobre os predicados fundamentais em tempo de execução do programa. A forma automática de selecção dos predicados a expandir descrita no capítulo 3 e baseada num grafo do programa, segue um critério que não é objectivo em termos do tempo de execução. Daí que seja permitido ao programador fornecer a indicação dos predicados a expandir, cabendo-lhe a ele a obtenção de um compromisso entre o tempo de compilação e tamanho do código executável, e o tempo de execução do programa. Durante o desenvolvimento do programa, e as suas sucessivas compilações, pode-se decidir não expandir nenhum predicado, deixando para a compilação final a indicação de uma lista mais ou menos numerosa dos predicados a expandir.

A segunda forma de controlar o *unfolding* é pela limitação do número de elementos do domínio de possíveis tipos para os argumentos. Esta limitação é importante porque o número de predicados especializados criados a partir do predicado original é fruto de dois factores: o número de argumentos do predicado (a sua aridade) e o número de possíveis valores que cada um desses argumentos pode tomar (o número de elementos do domínio). Se quanto ao primeiro destes dois factores nada se pode fazer, já no segundo factor podemos tentar reduzir a extensão do domínio de forma a que o número de predicados especializados não seja excessivo.

Não são gerados novos predicados para cada combinação de possíveis tipos dos argumentos com cada predicado. Apenas as hipóteses que vão surgindo da expansão do código. Assim, se em vez de expandirmos um predicado na sua forma geral, caso em que serão geradas todas as hipóteses, o expandirmos sabendo que é dessa forma que se inicia a execução, serão imediatamente limitadas uma série de hipóteses de expansão, resultando num muito menor número de predicados criados. Tal é o caso dos programas com uma directiva, que indica a forma como o programa irá começar a ser executado.

A terceira forma de controlar o *unfolding* baseia-se na observação de que determinado predicado especializado não traz melhorias na execução comparativamente com o caso geral. Neste caso substituem-se as referências para este predicado por referências ao caso geral.

4.8 A indexação

4.8.1 Vantagens do *unfolding* na indexação.

Tal como originalmente descrita na WAM, a indexação faz-se no primeiro argumento dos predicados. Esta escolha do argumento tem bastante lógica dada a forma estruturada com que os programadores de Prolog dispõem os argumentos dos predicados, visando a que este primeiro argumento seja chamado instanciado, permitindo bons resultados na selecção das cláusulas. Pode no entanto acontecer que algumas chamadas a um predicado sejam feitas com o primeiro argumento não instanciado, e outro argumento sim. Como em tempo de compilação não se sabe nada sobre os argumentos não haveria maneira de mudar o argumento da indexação. Tal não é o caso ao compilarmos o código e fazermos o seu *unfolding*. Aí temos informação sobre os argumentos de cada chamada a um predicado, pelo que se pode simplesmente alterar o argumento sobre o qual se faz a indexação.

A figura 4.3 mostra um exemplo típico de pesquisa numa base de dados onde, para garantir eficiência, a indexação deve ser feita em diferentes argumentos consoante a chamada ao predicado `rio/2`. No predicado `procura_por_cidade/0` o *objectivo* `rio/2` é chamado com o primeiro argumento instanciado e o segundo livre, pelo que a indexação deve ser feita no primeiro argumento. Já no predicado `procura_por_rio/0` a situação inverte-se, pelo que a indexação deveria ser feita no segundo argumento.

Usando o *unfolding*, deduz-se pela interpretação abstracta que à saída do predicado `read/1`, o argumento é o tipo *const*, sabendo-se nas duas diferentes chamadas ao predicado `rio/2`, qual dos argumentos está instanciado. Indexando as cláusulas sobre este argumento, gera-se um `switch_on_constant` que permite aceder directamente à cláusula pretendida, ao invés de ter que percorrer todas as cláusulas com `try`, `retry` e `trust`, se se indexasse sobre uma variável livre.

```
:- write('Como quer procurar a Base de Dados?  (1.Cidade, 2.Rio)'),
    read(Op),
    (Op == 1 -> procura_por_cidade ; procura_por_rio).

procura_por_cidade :- write('Nome da cidade:'),
    read(Cidade), rio(Cidade,Rio), write(Rio).

procura_por_rio :- write('Nome do rio:'),
    read(Rio), rio(Cidade,Rio), write(Rio).

rio('Lisboa','Tejo').
rio('Porto','Douro').
rio('Coimbra','Mondego').
```

Figura 4.3: Consulta a uma base de dados

4.8.2 Mudando o argumento de indexação

O pior argumento de chamada para fazer indexação é aquele que é uma variável livre. Este argumento unificaria com o argumento respectivo de todas as cláusulas do predicado. O melhor argumento para fazer indexação é aquele que permite seleccionar apenas uma das cláusulas do predicado.

A selecção do argumento de indexação faz-se olhando para os argumentos de chamada e para os argumentos das cláusulas.

A partir de agora quando falarmos em argumento de chamada referimo-nos a um dos argumentos de chamada, aquele sobre o qual se está a fazer a indexação.

O objectivo é minimizar o número de alternativas a tentar, ou seja, a criação de *pontos-de-escolha*. Se o argumento de chamada é *var* então o número de cláusulas a tentar é o número de cláusulas do predicado, pelo que não pode haver pior argumento de indexação. Se não soubermos nada sobre o argumento de chamada, isto é, se for do tipo *?*, estamos numa situação melhor do que se soubermos que é *var*, pois podemos gerar indexação de segundo nível, com instruções `switch`. Neste caso olhamos para as cláusulas do predicado, recolhendo informação sobre o argumento respectivo, para calcular o número de alternativas que se vão formar. Se o argumento das cláusulas só tiver inteiros todos diferentes, então com um `switch_on_integer` não geramos *pontos-de-escolha*. Se houver cláusulas com o argumento do tipo *var*, então cada uma destas cláusulas criará uma alternativa. Também se houver mais que uma cláusula com argumento do tipo *list* serão criadas mais alternativas. Se existirem apenas duas cláusulas, uma com argumento *nil* e outra com argumento *list*, não é criado nenhum *ponto-de-escolha*, graças à instrução `switch_on_term`.

Se soubermos que o argumento de chamada é do tipo *nil*, *int*, *const*, *list* ou *struct*, estamos na melhor situação, pois vamos seleccionar apenas as cláusulas do predicado cujo argumento satisfaz o tipo do argumento de chamada. Em particular podemos ficar reduzidos a apenas uma cláusula.

Após seleccionar estas cláusulas geramos instruções de indexação apenas para estas cláusulas e não para todas as do predicado.

Calculando para cada argumento do predicado o número de *pontos-de-escolha* que a indexação sobre ele criaria, escolhe-se aquele que minimize esse número de *pontos-de-escolha*.

A figura 4.4 descreve o algoritmo de selecção do argumento de indexação.

4.8.3 Indexação para os predicados sobre os quais não se faz *unfolding*

A estratégia descrita acima poderia também ser aplicada aos predicados sobre os quais não se faz *unfolding*. O estudo sobre as cláusulas do predicado seria feito para o caso em que não se sabe nada sobre o argumento de chamada. Seria escolhido o argumento que minimizasse a criação de *pontos-de-escolha*. No entanto todo este trabalho deixaria de fazer qualquer sentido se o argumento à chamada não estivesse instanciado, pois a indexação de segundo nível gerada não seria utilizada. Daí que para estes predicados se faça uma indexação no primeiro argumento, pois apesar de teoricamente se poderem criar menos *pontos-de-escolha* indexando noutro argumento, a probabilidade de o primeiro argumento vir instanciado é maior dada a tendência dos programadores, o que contrabalança a hipótese de escolha de outro argumento para indexação.

```
selecciona_arg_a_indexar(pred(tipo1,...,tipoN),arg):
início
   para cada arg em pred(tipo1,...,tipoN)
   início
      se (var) num_pts_esc = número_de_cláusulas;
      se (?) recolhe_info_cláusulas(todas_cláusulas);
      senão selecciona_cláusulas_unif(arg,cláusulas),
         recolhe_info_cláusulas(cláusulas);
   fim
   arg = menor(num_pts_esc);
fim

recolhe_info_cláusulas(Cláusulas):
início
   se (var) num_pts_esc++;
   se (int) e não único(int) num_pts_esc++;
   se (nil) e não único num_pts_esc++;
   se (list) e não único num_pts_esc++;
   se (struct) e não único(struct) num_pts_esc++;
fim
```

Figura 4.4: O algoritmo para selecção do argumento de indexação

4.9 Um exemplo de *unfolding* de um programa

Esta secção concretiza o que foi explicado nas secções anteriores através de um exemplo de *unfolding* de um programa. A figura 4.5 ilustra o código Prolog do programa exemplo, o seu código WAM e os tipos dos seus argumentos à saída.

A directiva presente no programa Prolog é transformada pela `wamcc` no predicado `$exec_1`.

O código apresentado tem algumas simplificações em relação ao código realmente gerado pelo compilador, mas sem qualquer interferência no que o exemplo pretende explicar.

A expansão do programa começa pelo predicado `$exec_1`, uma vez que também a execução do programa começará por aí. Como este predicado tem aridade 0 não há qualquer informação a guardar. Pela mesma razão também não se faz a selecção do argumento de indexação. Como o predicado só tem uma cláusula também não se faz indexação. Começa-se portanto a expandir as instruções WAM da única cláusula do predicado.

```
Prolog Code:

append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

:- append([1,2],[3,4],R).


Output types:
append(struct,any,any)
```

```
WAM Code:

append/3: [get_nil(1),
           get_x_value(2,3),
           procceed],
          [get_list(1),
           unify_x_variable(4),
           unify_x_variable(1),
           get_list(3),
           unify_x_value(4),
           unify_x_variable(3),
           execute(append/3)]

$exec_1:   put_list(2),
           unify_integer(2),
           unify_nil,
           put_list(1),
           unify_integer(1),
           unify_x_value(2),
           put_list(3),
           unify_integer(4),
           unify_nil,
           put_list(2),
           unify_integer(3),
           unify_x_value(3),
           put_x_variable(3,3),
           execute(append/3)
```

Figura 4.5: Um programa exemplo para *unfolding*

A primeira instrução é `put_list(2)`. Desta instrução guarda-se a seguinte informação: o *modo* corrente é de escrita, e o registo de argumento 2 é do tipo *list*. A instrução resultante do *unfolding* é exactamente a mesma.

A próxima instrução é `unify_integer(2)`. Como sabemos que o *modo* em que se está é de escrita, complementa-se esta instrução com essa informação, resultando na instrução `unify_integer(2,write)`.

O *unfolding* da próxima instrução, `unify_nil`, resulta na instrução `unify_nil(write)`, exactamente como a instrução anterior.

Seguidamente temos uma nova sequência de instruções `put_list(1)`, `unify_integer(1)`, com exactamente os mesmos resultados: o *modo* continua a ser de escrita, o registo de argumento 1 é do tipo *list*, e a expansão destas instruções resultará em `put_list(1)` e `unify_integer(1,write)`.

A próxima instrução é um `unify_x_value(2)` que coloca na cauda da lista corrente a lista apontada pelo registo 2. Como o *modo* corrente é de escrita, complementa-se esta instrução com informação sobre o *modo* e sobre o seu argumento, 2, que se sabe ser do tipo *list*. A instrução resultante é `unify_x_value(2,write,list)`.

Até este momento tratamos do primeiro argumento do *objectivo* da directiva, que contém a lista [1,2]. O código para o segundo argumento trata-se exactamento da mesma forma, e o seu *unfolding* é igual, variando apenas os inteiros e o número dos registos usados. É isso que fazem as 6 instruções seguintes do código WAM, em tudo indênticas às anteriormente explicadas.

A próxima instrução é um `put_x_variable(3,3)`. Guarda-se a informação de que o registo de argumento 3 é uma variável nova, para a qual ainda não há referências. A instrução resultante do *unfolding* é exactamente a mesma.

Finalmente, e terminando o código deste predicado, vem a instrução `execute(append/3)`. Com base na informação guardada sobre os registos de argumentos, contrói-se o modo como é feita a chamada. Neste caso, os argumentos são do tipo *list*, *list*, *var*, pelo que a instrução resultante do *unfolding* é `execute(append(list,list,var))`. O *unfolding* desta instrução força ao *unfolding* do predicado `append(list,list,var)` chamado por esta instrução.

Começa-se agora o *unfolding* do predicado append(list,list,var). Guarda-se a informação transmitida pelos seus argumentos de chamada, e selecciona-se o argumento de indexação. O argumento escolhido é o argumento 1, que por se saber ser do tipo *list* permite seleccionar apenas uma cláusula, pelo que novamente não é necessário proceder à indexação.

A primeira instrução do código da cláusula seleccionada é `get_list(1)`. Sabe-se que o argumento 1 tem tipo *list*, pelo que se guarda a informação de que o *modo* corrente é de leitura. A instrução resultante é `get_list(1,list)`.

As duas próximas instruções são `unify_x_variable(4)` e `unify_x_variable(1)`. Como o *modo* em que estamos é de leitura, teremos que promover a informação sobre os tipos dos registos 4 e 1 a *?*, porque os seus valores serão lidos da `heap`, e não sabemos o que pode vir da `heap`. As instruções resultantes são `unify_x_variable(4,read)` e `unify_x_variable(1,read)`.

Seguidamente temos um `get_list(3)`. Como o registo 3 tem tipo *var*, o *modo* passa a ser de escrita, e complementa-se a instrução com a informação sobre o argumento, ficando `get_list(3,var)`.

A próxima instrução é um `unify_x_value(4)` em modo de escrita, resultando na instrução `unify_x_value(4,write,?)`, pois o registo 4 tinha sido promovido a *?*.

Agora vem a instrução `unify_x_variable(3)`, em modo de escrita, guardando-se a informação de que o registo 3 é uma variável nova. O *unfolding* desta instrução resulta em `unify_x_variable(3,write)`.

Finalmente, e para terminar o código do predicado `append(list,list,var)` vem a instrução `execute(append/3)`. Esta instrução leva ao *unfolding* de um novo predicado, `append(?,list,var)`, por ser expandida em `execute(append(?,list,var))`.

Este predicado é uma nova especialização do predicado geral `append/3`. A selecção do argumento a indexar escolhe o primeiro argumento, por ser este o que menos *pontos-de-escolha* poderá criar, graças à indexação feita com `switch_on_term`.

A diferença entre o código deste predicado e o código do predicado anterior é que agora temos que desreferenciar o primeiro argumento e, dependendo do seu valor, saltar ou para o caso em que este é *nil*, *list* ou *var*. Se o argumento for *list*, e como esta informação é propagada para as próximas instruções, estamos no mesmo caso do predicado append(list,list,var), explicado anteriormente.

Se o argumento for *nil* a instrução a expandir é `get_nil(1)`. O facto do tipo do argumento ter sido propagado permite saber agora que é do tipo *nil*, resultando o *unfolding* desta instrução em `get_nil(1,nil)`.

A próxima instrução é `get_x_value(2,3)`. Sabe-se que o argumento 2 tem tipo *list* e o argumento 3 tem tipo *var*, verificando-se a compatibilidade destes tipos, e concluindo que o argumento 3 passará a ter tipo *list*, fruto da unificação feita. A instrução resultante é `get_x_value(2,3,list,var)`.

O *unfolding* da próxima instrução, `procceed`, é a própria instrução.

Falta o caso em que o argumento desreferenciado é do tipo *var*. Este caso nunca chega a acontecer, mas o *unfolding* não tem maneira de saber isso, pelo que se gera código para ele.

Ambas as cláusulas do predicado terão que ser testadas. Em determinada altura encontrar-se-á no código WAM da segunda cláusula a instrução `execute(append/3)`, sendo os argumentos desta vez (*var,list,var*). Isto levará à criação de um último predicado especializado, `append(var,list,var)`, cujo código nunca será percorrido, mas que é gerado pela razão explicada acima.

A figura 4.6 representa o código final resultante do *unfolding* do programa original.

```
conc(var,list,var):
   try_me_else(L1)
      get_nil(0,ovar)
      get_x_value(1,2,list,ovar)
      procceed
   L1:
   trust_me_else_fail
      get_list(0,var)
      unify_x_variable(3,write)
      unify_x_variable(0,write)
      get_list(2,var)
      unify_x_value(3,write,var)
      unify_x_variable(2,write)
      execute(conc(var,list,var))
conc(?,list,var):
   switch_on_term(0,?)
   const:
   int:
   nil:
      get_nil(0,nil)
      get_x_value(1,2,list,var)
      procceed
   list:
      get_list(0,list)
      unify_x_variable(3,read)
      unify_x_variable(0,read)
      get_list(2,var)
      unify_x_value(3,write,?)
      unify_x_variable(2,write)
      execute(conc(?,list,var))
   struct:
   var:
      try_me_else(L2)
         get_nil(0,ovar)
         get_x_value(1,2,list,ovar)
         procceed
      L2:
      trust_me_else_fail
         get_list(0,var)
         unify_x_variable(3,write)
         unify_x_variable(0,write)
         get_list(2,var)
         unify_x_value(3,write,var)
         unify_x_variable(2,write)
         execute(conc(var,list,var))
conc(list,list,var):
   get_list(0,list)
   unify_x_variable(3,read)
   unify_x_variable(0,read)
   get_list(2,var)
   unify_x_value(3,write,?)
   unify_x_variable(2,write)
   execute(conc(?,list,var))
$exec_1:
   put_list(1)
   unify_integer(2,write)
   unify_nil(write)
   put_list(0)
   unify_integer(1,write)
   unify_x_value(1,write,list)
   put_list(2)
   unify_integer(4,write)
   unify_nil(write)
   put_list(1)
   unify_integer(3,write)
   unify_x_value(2,write,list)
   put_x_variable(2,2)
   execute(conc(list,list,var))
```

Figura 4.6: Código resultante do *unfolding*

Capítulo 5

Passagem do código WAM para C

Este capítulo descreve a forma como o código WAM resultante do *unfolding* é convertido para código C. É neste capítulo que se concretizam as optimizações possibilitadas pela complementação das instrucões WAM com a informação descrita no capítulo anterior, comparando o código C de uma instrução WAM geral e o código C das instruções resultantes do *unfolding*.

5.1 Motivação da tradução de WAM para C

Desde a criação da WAM que existem duas vertentes na forma de tornar executável este código WAM. A primeira, e mais generalizada, é a construção de um emulador de código WAM. As vantagens desta vertente são a simplicidade da escrita de um tal emulador (cerca de 3000 linhas de C, para uma versão não optimizada), a portabilidade do código, se um tal emulador for escrito numa linguagem tão generalizada como C, e a facilidade com que se lida com a modificação e criação de novo código WAM. As desvantagens deste método são a sua menor eficiência, devido ao ciclo *fetching*, *decoding*, *executing*, presente na execução de cada instrução WAM, e o facto de não se gerar código autonomamente executado pela máquina, tendo o emulador que estar sempre presente.

A segunda vertente é a tradução do código WAM para código nativo. Este método apresenta a vantagem de não ter a sobrecarga da emulação das instruções, e portanto ser mais eficiente. Por outro lado, a geração desse código eficiente é uma tarefa complexa e exige um conhecimento profundo da máquina para a qual se está a gerar código [Lop96], de forma a tirar o máximo partido das características de uma máquina em particular. Uma outra desvantagem é o facto de o código não ser executável em máquinas diferentes, e de a adaptação do compilador de uma máquina para outra poder ser uma tarefa difícil.

Uma outra abordagem na execução de código WAM é a tradução para C [LH94]. Existem já alguns sistemas que usaram esta abordagem, como a `wamcc`, Janus [GDBS92], KL1 [CFY93] e Erlang [Hau94]. As vantagens desta abordagem cifram-se pela obtenção de um equilíbrio entre as duas abordagens descritas acima: por um lado temos uma maior simplicidade na geração de C do que de código nativo, e obtemos portabilidade, graças à generalização de compiladores de C, por outro deixamos de ter a sobrecarga da emulação das instruções, pois obtemos código executável fornecido pelo compilador de C. É natural que se perca em eficiência comparativamente com a geração de código nativo para uma máquina em particular, pois o compilador de C não pode fazer tudo, daí que se trate da obtenção de um equilíbrio.

5.2 Controlo da execução do programa

Um problema fundamental na tradução de WAM para C é a forma como é feito o controlo da execução do programa. O código WAM é um código sequencial, onde as transferências de execução são feitas por saltos. Isto difere um pouco da linguagem C, dotada de estruturas de controlo mais complexas, como funções, fornecendo pouco para controlo de baixo nível. Daí que um ponto crucial é a tradução dos saltos, tais como são feitos nas instruções WAM.

Estes saltos na WAM são de dois tipos distintos: saltos directos e saltos indirectos. Os saltos directos são aqueles cujo endereço de salto é uma *etiqueta* conhecida. Traduzem-se na WAM pelas instruções `call` e `execute`. Os saltos indirectos são aqueles cujo endereço de salto é o conteúdo de uma variável, e acontecem quando na WAM, por exemplo, encontramos uma instrução `procceed`(conteúdo da variável `CP`).

A ideia básica em todos os sistemas de tradução de WAM para C de implementar a passagem de execução do programa é traduzir um salto WAM por um salto em C, isto é, uma instrução `goto`. Se bem que isto não traga dificuldades nos saltos directos, já nos saltos indirectos surgem problemas, porque no C standard (ANSI) estes saltos indirectos não estão disponíveis. Para além disso, o `goto` só pode aceder a código dentro da mesma função.

Para resolver estes problemas várias soluções foram propostas. No sistema Janus gera-se um programa C que consiste numa única função, formada por uma instrução `switch` para simular os saltos indirectos. Já no sistema KL1, e para evitar gerar uma tão grande função em C, cada predicado é transformado numa função em C, sendo estas funções controladas por uma função supervisora, que assegura a continuação das chamadas. Desta forma cada chamada WAM é feita por um regresso à função supervisora, e depois por uma chamada a uma função. Isto é claramente muito menos eficiente do que o `goto` que seria gerado em código nativo. Para contrariar esta perda de eficiência, no KL1 optou-se por colocar todos os predicados do mesmo módulo numa única função em C, resultando, no caso de só termos um módulo, numa situação idêntica à do sistema Janus.

A solução que foi por nós adoptada foi a seguida no sistema Erlang, com a particularidade de nós apenas gerarmos uma função. A ideia é usar as facilidades do compilador GNU C (gcc) [Sta93], que permite guardar o endereço de uma *etiqueta* num apontador, e seguidamente efectuar a chamada para o valor guardado nesse apontador.

A figura 5.1 exemplifica a forma como são feitos os saltos directos e indirectos.

```
Programa Prolog:        Código C gerado:
p :- q, r.              void wam_code()
q.                      {
                        p:
                          push(CP);   /*allocate*/
                          CP = &&p1;
                          goto q;
                        p1:
                          pop(CP);    /*deallocate*/
                          goto r;
                        q:
                          goto *CP;
                        }
```

Figura 5.1: Implementação dos saltos directos e indirectos

Assim, ao gerarmos código C, para além de gerarmos as *etiquetas* que marcam o início do código de cada predicado (no exemplo acima `p` e `q`), temos que criar novas *etiquetas* para colocar imediatamente a seguir a um `call`, de forma a que essa *etiqueta* defina o ponto de continuação para o regresso da chamada (no exemplo acima `p1`).

5.3 Uma nota sobre a implementação das *tags*

A implementação de tipos dinâmicos em Prolog é feita através de *tags*. Cada objecto é representado de forma a caber num registo (32 bits). Esta representação consiste em duas partes: uma parte para a *tag*, que determina o tipo do objecto, e uma parte para o valor, que é usada com objectivos diferentes consoante o tipo do objecto.

Na geração de código nativo a implementação das *tags* tem um papel importante na eficiência dos programas compilados. Isto porque os testes de bits próprios do conjunto de instruções de cada processador devem influenciar a representação da *tag* no registo, de forma a minimizar quer os testes de tipo de um objecto, quer a recolha do seu valor.

Em C, ou em qualquer outra linguagem que não seja a linguagem do processador, a abstracção é maior, pelo que a implementação das *tags* já não é influenciada pelo conjunto de instruções da máquina.

Escolhemos usar os três bits mais à esquerda para representar a *tag*. Para cada tipo as *tags* escolhidas foram:

- `000` - Variável
- `100` - Inteiro
- `101` - Lista
- `110` - Estrutura
- `111` - Átomo

As preocupações com a escolha destas *tags* foram basicamente influenciadas pelo tipo variável. Como o teste de se um determinado objecto é variável aparece muitas vezes, com esta representação resume-se a testar se o registo é maior que 0. Também a recolha do valor de um registo do tipo variável é simplificada, pois como a representação de variável são os primeiros 3 bits, e são 0, não é preciso fazer nenhuma operação sobre o registo para aceder ao seu valor.

Em relação ao tipo átomo criou-se uma função que calcula com base no átomo um valor distinto para cada um desses átomos, usando-se esse valor na representação do átomo.

5.4 Comparação das instruções gerais e das instruções resultantes do *unfolding*

As optimizações possibilitadas pelo *unfolding* manifestam-se de duas formas: pela expansão de um predicado em vários predicados especializados, e pela complementação das instruções WAM com informação sobre os seus argumentos e modo de unificação. O primeiro caso resulta também de se complementar as instruções `call` e `execute` com informação acerca dos seus argumentos, levando então à criação dos novos predicados.

Das optimizações possibilitadas por esta complementação das instruções WAM quase que ainda não se falou. É nesta secção que se concretizam tais optimizações, descrevendo a forma como quer a instrução geral quer a instrução complementada são implementadas em C.

A tradução da instrução geral para C é inspirada na descrição destas instruções WAM feita no livro de Ait-Kaci [AK91]. São dados apenas alguns exemplos da tradução de instruções, pelo que se deve consultar o livro referido para encontrar a descrição das outras instruções. Esta tradução é algo bem conhecido, pelo que o interesse desta secção é a comparação entre a implementação da instrução geral e das intruções complementadas com informação acerca dos argumentos e modo das estruturas e perceber os ganhos em termos de eficiência.

As instruções que a seguir se explicam aparecem no exemplo do `append/3` descrito no capítulo anterior.

5.4.1 A instrução `get_nil(I)`

A figura 5.2 representa o código C para implementação da instrução `get_nil(1)` na sua forma genérica.

Começa-se por testar se a *tag* do registo `1` é variável. Se for vamos procurar o valor no fim da possível cadeia, desreferenciando o argumento, e atribuímos a `d1` esse valor. Se o registo `1` não for variável, copiamos para `d1` o conteúdo de `X[1]`. No caso de `d1` ser uma variável não instanciada, movemos para o endereço apontado por `d1` o átomo *nil*, e copiamos para a trilha o endereço `d1`. Se `d1` estiver instanciado comparamos o seu valor com o átomo *nil*, falhando e executando o *backtraking* se a comparação falhar.

As instruções especializadas para determinado tipo de argumentos da instrução `get_nil(I)`

```
if (IsVar(X[1])) d1 = deref(X[1]);
else d1 = X[1];
if (IsVar(d1))
{
   *((int *) d1) = atom_nil;
   trail(d1);
}
else if (d1 != atom_nil) fail=1;
if (fail) backtrak();
```

Figura 5.2: Código C para implementação da intrução `get_nil(1)`

são quatro: `get_nil(I,nil)`, `get_nil(I,var)`, `get_nil(I,ovar)` e `get_nil(I,?)`. A razão porque não há mais instruções, uma vez que há mais elementos no domínio, tipo `get_nil(I,int)`, é porque a execução destas instruções falharia obviamente, pelo que se ao fazermos o *unfolding* da instrução `get_nil(I)` surgir um caso destes, o resultado do *unfolding* desta instrução é a instrução `fail`.

Para a primeira instrução, `get_nil(I,nil)`, o C que se gera não é nenhum, pois esta instrução não tem que fazer absolutamente nada, nem mesmo que incrementar o `P` (contador do programa) para a próxima instrução, como fazem todas as instruções descritas em [AK91]. Isto porque o facto de não se emular uma máquina abstracta, mas sim gerar-se código directamente executável pela máquina, fornece um contador de programa automaticamente gerado pelo compilador ao tratar o código original. A razão porque nesta instrução não se tem que fazer absolutamente nada é óbvia olhando para o código da figura 5.2, pois o facto de sabermos que o argumento é *nil* torna desnecessários todos os diferentes casos e comparações desse código.

A segunda instrução é `get_nil(I,var)`. Como `var` caracteriza as variáveis não inicializadas o código C fica muito simplificado, bastando fazer:

```
*((int *) X[1]) = atom_nil;
```

A próxima instrução é `get_nil(I,ovar)`. A diferença em relação à instrução anterior é que não se trata agora de uma variável não inicializada, pelo que o código C a gerar é:

```
d1 = deref(X[1]);
*((int *) d1) = atom_nil;
trail(d1);
```

A última instrução é `get_nil(I,?)` que corresponde ao caso geral descrito na figura 5.2. O C que é gerado é exactamente o dessa figura.

5.4.2 A instrução `get_list(I)`

A figura 5.3 representa o código C gerado para o caso geral da instrução `get_list(1)`.

```
if (IsVar(X[1]) d1 = deref(X[1]);
else d1=X[1];
if (IsVar(d1))
{
   trailbind(d1,MakeList((int) H)); H++;
   mode = write;
}
else if (IsList(d1))
     {
         S = (int *) Value(d1);
         mode = read;
     }
     else fail = 1;
if (fail) backtrak();
```

Figura 5.3: Código C para implementação da intrução `get_list(1)`

Esta instrução é especializadas em quatro instruções, correspondentes aos tipos que o argumento pode ter: `get_list(1,list)`, `get_list(1,var)`, `get_list(1,ovar)`, `get_list(1,?)`.

No primeiro caso, `get_list(1,list)`, o C que se gera é simplesmente

```
S = (int *) Value(X[1]);
```

porque se sabe que `X[1]` tem tipo *list*. Nem sequer é preciso fazer a atribuição do *modo* para *read*, pois este *modo* é propagado nas instruções.

Seguidamente temos as instruções `get_list(1,var)`, `get_list(1,ovar)`. A diferença entre ambas foi explicada na secção anterior. O código C gerado para cada uma destas instruções é, respectivamente:

```
*((int *) X[1]) = MakeList((int) H);
```

e

```
d1 = deref(X[1]);
trailbind(d1,MakeList((int) H)); H++;
```

5.4.3 As instruções unify

A simplificação fundamental das instruções `unify` cifra-se pela inclusão nestas instruções do *modo* em que é feita a unificação. Assim, e considerando por exemplo o código geral da instrução `unify_integer(15)` da figura 5.4, divide-se esta instrução em duas instruções, `unify_integer(15,read)` e `unify_integer(15,write)`.

```
if (mode == read)
{
   d1 = deref(S);
   if (IsVar(d1))
   {
      *((int *) d1) = MakeInteger(15);
      trail(d1);
   }
   else if (d1 != MakeInteger(15)) fail=1;
}
else
{
   *H = MakeInteger(15); H++;
}
if (fail) backtrak();
```

Figura 5.4: Código C para implementação da intrução `unify_integer(15)`

O código de cada uma destas instruções é, respectivamente:

```
d1 = deref(S);
if (IsVar(d1))
{
   *((int *) d1) = MakeInteger(15);
   trail(d1);
}
else if (d1 != MakeInteger(15)) fail=1;
if (fail) backtrak();
```

e

```
*H = MakeInteger(15); H++;
```

As instruções unify_value(I), para além da informação sobre o *modo* de unificação, são ainda complementadas com informação acerca do registo I. Isto permite simplificar a unificação, especializando-a para os casos em que conhecemos o tipo de um dos valores a unificar. Por exemplo, enquanto que o código geral de unify_x_value(1) é o seguinte:

```
if (mode == read) unify(X[1],S);
else
{
   *H = X[1]; H++;
}
S++;
if (fail) backtrak();
```

o código de unify_x_value(1,read,var) é simplesmente

```
*X[1] = *S++;
```

não sendo necessário chamar a função de unificação geral.

Estes exemplos descritos ao longo das últimas três subsecções mostram de forma clara que o *unfolding* permite simplificar profundamente a execução das instruções WAM. Esta simplificação traduz-se numa redução do número de instruções C para implementar a instrução WAM pretendida. Esta redução chega a ser de 100% em relação à implementação geral (caso de get_nil(I,nil)), sendo em muitas instruções de 80% e 50%.

5.5 Desenvolvimento futuro do tradutor para C

A maior fonte de incompletude deste trabalho é a implementação das instruções WAM em C. Esta implementação cobre apenas uma parte das instruções do código WAM expandido, e é extremamente incompleta na implementação do predicados internos (*built-ins*) do Prolog. Esta foi uma opção tomada. A parte central e fundamental do trabalho era na geração de código intermédio. A razão porque se implementou o tradutor para C foi para poder testar objectivamente a melhoria em eficiência que o *unfolding* provoca, compilando um pequeno conjunto de pequenos programas Prolog.

Futuramente, e motivados pelos resultados que o *unfolding* mostrou nos programas testados, este tradutor terá que ser completado e melhorado de forma a poderem-se testar aplicações reais. É natural que se abandone até a geração de código C, e se passe para a geração directa de código nativo. A escolha da linguagem C neste trabalho foi justificada pela maior simplicidade na sua geração comparativamente com a geração de código nativo, sendo suficiente para testar o que se pretendia. No futuro, para melhorar ainda a eficiência, e não tendo as limitações de tempo do presente trabalho, é natural que se enverede pela geração de código nativo.

Capítulo 6

A performance do *unfolding*

Este capítulo faz-se uma avaliação sumária da melhoria em termos de eficiência que o *unfolding* do código WAM traz. É feita uma comparação entre a execução do código WAM sem expansão e o código WAM expandido. Esta comparação é feita sobre dois parâmetros: o tempo de execução e o tamanho do código C gerado.

6.1 Os programas de teste

A escolha dos programas de teste foi extremamente condicionada pelo facto de a implementação em C das instruções WAM não ser uma implementação completa. Por exemplo, não foram implementadas as instruções `switch_on_constant` e `switch_on_integer`, que através de uma função de *hashing* forneceriam uma *label* para onde saltar.

Também em termos de *built-ins* a implementação em C é muito incompleta. Isto porque as ideias descritas ao longo deste trabalho não se traduziriam em qualquer melhoria na implementação destes *built-ins*, pelo que não interessava perder tempo com tal implementação, sobretudo porque não se pretendia a obtenção de um compilador completo, mas sim a obtenção de um meio para testar algumas ideias.

As próprias estruturas de dados da memória não são muito robustas, implementando-se apenas alguns testes para verificar a não confluência de àreas de memória, pelo que um programa de maior envergadura poderá ter alguns problemas em correr. Na perspectiva que nos guiava, o interesse de grandes programas de teste era pouco, pois é comum que um programa de 1000 linhas passe 80% do tempo a executar o predicado `append/3`, sendo pouco mais útil que um programa de teste que apenas execute o predicado `append/3`, e muito menos conveniente.

Assim, os programas escolhidos para quantificarem, quer em termos do tempo de execução, quer em termos do crescimento do código C gerado, as ideias implementadas neste trabalho, foram quatro:

- `append.pl` - Faz a junção de duas listas de 15 elementos (figura 6.1);
- `nrev.pl` - Faz a inversão de uma lista de 30 elementos (figura 6.2);
- `qsort.pl` - Faz a ordenação de uma lista de 50 elementos (figura 6.3);
- `fib.pl` - Calcula o `fib` (número de Fibonachi) de 100 (figura 6.4);

As quatro figuras que se seguem representam o código Prolog dos quatro programas de teste.

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

:- append([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
          [16,17,18,19,20,21,22,23,24,25,26,27,28,29,30],R).
```

Figura 6.1: O programa `append.pl`

```
:- nrev([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
         16,17,18,19,20,21,22,23,24,25,26,27,28,29,30],R).

nrev([],[]).
nrev([H|T],R) :- nrev(T,M), append(M,[H],R).

append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

Figura 6.2: O programa `nrev.pl`

A código WAM resultante da compilação de qualquer um destes programas traduz-se em instruções WAM que foram implementadas em C, tendo sido algumas delas descritas no

```
:- qsort([27,74,17,33,94,18,46,83,65,2,32,53,28,85,99,47,28,
          82,6,11,55,29,39,81,90,37,10,0,66,51,7,21,85,27,31,
          63,75,4,95,99,11,28,61,74,18,92,40,53,59,8],_,[]).

qsort([X|L],R,R0) :-
        partition(L,X,L1,L2),
        qsort(L2,R1,R0),
        qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :-
        X =< Y,
        partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
        X > Y,
        partition(L,Y,L1,L2).
partition([],_,[],[]).
```

Figura 6.3: O programa `qsort.pl`

```
:- fib(100,R).

fib(N,1) :- N =< 2.
fib(N,R) :- N > 2, N1 is N-1, N2 is N-2,
            fib(N1,R1),
            fib(N2,R2),
            R is R1+R2.
```

Figura 6.4: O programa `fib.pl`

capítulo anterior. Em termos de *built-ins*, no caso dos dois primeiros programas não temos qualquer chamada a predicados que não sejam predicados definidos pelo programador. No programa `qsort.pl` e no programa `fib.pl` temos apenas os *built-ins* `=<`/2, `>`/2, `-`/2 e `+`/2, que foram facilmente implementados em C.

6.2 Melhoria do tempo de execução do *unfolding*

A melhor forma de testar objectivamente a melhoria em termos de tempo de execução que o *unfolding* provoca é através de duas implementações idênticas, com a diferença de que numa se faz *unfolding* do programa e na outra não se faz. Isto é extremamente simples de implementar no nosso trabalho, visto que podemos seleccionar quais os predicados a expandir, e em particular podemos escolher não expandir nenhum predicado. Se comparassemos a nossa implementação de um programa fazendo o *unfolding* do código, com uma outra implementação que não faz esta expansão, os resultados poderiam ser influenciados pelas diferenças relativas a pormenores de implementação, camuflando as verdadeiras melhorias que o *unfolding* provoca.

Foi por esta razão que evitamos comparar este trabalho com a `wamcc`, que tem, por lhe ter servido de base, muitas semelhanças com a nossa implementação. No entanto, na implementação em C do código WAM existem grandes diferenças nos dois sistemas. Foi nosso objectivo na geração de C tentar gerar um código o mais *in-line* possível, evitando a sobrecarga das chamadas a funções. Isto não foi seguido no tradutor para C da `wamcc`, onde cada instrução WAM é implementada por uma função em C. Este aspecto pode explicar alguma da diferença de performance entre os dois sistemas, mas não pode justificar tudo. Daí que tenhamos que concluir que parte da pior performance do compilador `wamcc` se fique a dever ao facto de ser um sistema completo e robusto, com toda a série de sobrecargas do código que isso acarreta.

A tabela 6.1 representa o quociente entre o tempo de execução da versão sem *unfolding* e o tempo de execução do programa expandido.

Programa de Teste	Quociente do Tempo de Execução
`append.pl`	5.1
`nrev.pl`	4.8
`qsort.pl`	4.3
`fib.pl`	2.9

Tabela 6.1: Versão sem *unfolding*/Versão com *unfolding*

Estes resultados são bastante interessantes pois traduzem acelerações importantes em relação à versão em que não se faz *unfolding* do código. Esta aceleração é de cerca de 5 vezes nos três primeiros programas, sendo um pouco pior no programa `fib.pl`. A razão da pior performance neste programa é que a informação que o *unfolding* deduz cifra-se em saber que as chamadas ao predicado `fib/2` são feitas com o primeiro argumento do tipo *int* e o segundo argumento do tipo *var*. O código do predicado é expandido em apenas um predicado especializado, `fib_int_var/2`. Apenas o facto de conhecer o tipo destes dois argumentos permite tornar o código cerca de três vezes mais rápido, eliminando uma série de testes, desreferenciações e operações desnecessárias.

Estes resultados são ainda mais interessantes quando comparados com os resultados de sistemas que suportam muitas das optimizações possibilitadas pelo *unfolding* através de interpretação abstracta. Tal é o caso do sistema Parma.

Neste sistema, e correndo um conjunto de 19 programas de teste, incluindo alguns equivalentes aos nossos 4 programas de teste, a aceleração possibilitada pela informação deduzida pela interpretação abstracta era da ordem de 2.8 vezes. Se considerarmos apenas os programas de teste equivalentes aos nossos programas de teste esta ordem é de 5.4 vezes, sendo semelhante aos nossos resultados. É preciso, no entanto, não esquecer que enquanto que a aceleração de 5 vezes por nós conseguida é obtida apenas pela propagação de informação, sem algoritmos complexos de interpretação abstracta do código dos programas, já no sistema Parma a obtenção da informação que possibilita a melhoria do tempo de execução, resulta de uma fase de interpretação abstracta extremamente complexa (cerca de 4500 linhas de SICStus Prolog).

6.3 Crescimento do tamanho do código

Um outro aspecto que é muito importante avaliar é o crescimento do código gerado, comparando o tamanho do código C gerado para a versão com *unfolding* e o tamanho do código C gerado para a versão sem *unfolding*.

A tabela 6.2 representa o quociente entre o tamanho do código C gerado nestas duas versões.

Programa de Teste	Quociente do Tamanho do código C
`append.pl`	2.1
`nrev.pl`	2.8
`qsort.pl`	4.1
`fib.pl`	0.7

Tabela 6.2: Versão com *unfolding*/Versão sem *unfolding*

Os valores da tabela 6.2 mostram um crescimento aceitável do tamanho do código C gerado. A moderação destes valores deve-se em parte ao número reduzido de argumentos dos predicados presentes em qualquer dos programas de teste. Em programas com predicados com um grande número de argumentos, a expansão de tais predicados implicará um crescimento do tamanho de código maior do que o observado nestes 4 programas de teste.

Outro factor que contribui para a moderação do crescimento do tamanho de código é o facto destes programas começarem a ser executados pela directiva. Desta forma o *unfolding* do código inicia-se com o conhecimento do tipo de argumentos com que é feita a primeira chamada. Isto pode cortar uma série de possíveis casos que teriam que ser considerados se não se soubesse como é iniciada a execução.

O crescimento do código é também controlado pelo facto de as instruções WAM resultantes do *unfolding*, isto é, complementadas com informação sobre os seus argumentos e modo de unificação, serem mais simples de implementar do que a instrução geral, e portanto precisarem de um número menor de instruções C para as implementar. Esta diferença pode ser muito significativa, dependendo do tipo do argumento. Isto contrabalança em parte a multiplicação de um predicado em vários predicados especializados. Em particular, quando um predicado é transformado em poucos predicados especializados o tamanho do código C gerado pode ficar mais pequeno do que o tamanho do código C gerado para a versão sem *unfolding*. É isto que acontece no programa `fib.pl` onde o predicado `fib/2` apenas é especializado no predicado `fib_int_var/2`.

Capítulo 7

Conclusão

Começamos este trabalho pelo estudo de um compilador, a `wamcc`. Este sistema revelou-se extremamente interessante, fruto da sua simplicidade e excelente nível de código. Essa mesma simplicidade tem no entanto consequências importantes na eficiência deste compilador, tornando bastante mais lento do que outros compiladores de Prolog existentes. Se bem que o autor reclame um tempo de execução algures entre o SICStus emulado e o SICStus nativo, os nossos testes mostraram melhorias apenas sobre o SICStus interpretado.

Foram estes dois factores (a simplicidade do código e a falta de performance) que motivaram o nosso trabalho. Começamos por reutilizar as boas partes do compilador `wamcc`, introduzindo-lhe componentes de forma a poder gerar informação que permitisse melhorar o tempo de execução dos programas.

A melhoria da eficiência foi baseada em três princípios fundamentais:

- Redução da granularidade das instruções;
- Extracção de determinismo;
- Especialização da unificação.

A implementação destes princípios veio melhorar profundamente o tempo de execução dos programas, conseguindo nos 4 programas de teste descritos no capítulo anterior uma aceleração de 25 vezes em média sobre o tempo de execução dos programas no compilador `wamcc`. Esta diferença não é, no entanto, totalmente credível sendo inflacionada pelo facto de a nossa implementação não ser uma implementação completa e robusta, ao contrário da `wamcc`.

A implementação destes princípios seguiu também um aspecto relevantemente salvaguardado no compilador `wamcc`, a simplicidade.

Assim, e ao contrário de muitos sistemas onde a redução da granularidade das instruções é conseguida pela definição de uma nova máquina abstracta, de um nível mais baixo do que a WAM, com toda a complexidade que isso acarreta, na nossa implementação este princípio foi atingido apenas pela complementação das instruções WAM existentes com informação acerca do tipo dos seus argumentos e modo de unificação. Esta estratégia reduziu o número de instruções do código gerado (neste caso C) necessárias para implementar as instruções WAM em até 100%, reduzindo em muitos casos em 80 e 50%.

Foi na extracção de determinismo que o *unfolding* trouxe mais ganhos. Os *objectivos* que aparecem no corpo das cláusulas, e tendo recolhida informação sobre o tipo dos seus argumentos, levam à criação de novos predicados, especializados naquele tipo de argumentos, que unificarão directamente com o *objectivo* (isto tem algumas semelhanças com o que é feito na VAM (Vienna Abstract Machine) [KN90], onde os *objectivos* e as cabeças dos predicados são unificados directamente, com a importante vantagem de que nós não necessitamos da criação de uma nova máquina abstracta para implementar isto). Aliado a esta criação de predicados especializados, o facto de se conhecerem os tipos dos argumentos nas chamadas dos predicados permitiu indexar mais eficientemente o código dos predicados, seleccionando o argumento que menos *pontos de escolha* criasse.

Para manter correcta a informação sobre as variáveis locais e globais que se ia propagando, e depois de efectuar uma chamada a um predicado, foi necessário recorrer a um algoritmo simples de interpretação abstracta, que determinasse os tipos dos argumentos dos predicados à saída. Esta foi a única informação em que foi necessário recorrer a interpretação abstracta sobre o programa para a deduzir. Toda a restante informação foi obtida apenas por propagação, constituindo esta independência de uma fase de interpretação abstracta um importante resultado desta tese.

Ainda por causa dos diferentes tipos de argumentos nos *objectivos* que aparecem no corpo das cláusulas, e porque cada diferente par (predicado/tipo de argumentos) gera um novo predicado, houve que definir uma heurística para selecção dos predicados a expandir, dada a impossibilidade de em programas de dimensão real expandir todos os seus predicados. O critério implementado de seleccionar as cláusulas que definem ciclos parece-nos bastante razoável, bastando considerar os ciclos de ordem 1 ou ordem 2, pois é raro a execução de programas Prolog definir ciclos pela chamada a mais do que dois predicados. É também comum que grandes programas Prolog passem a grande maioria do tempo em pequenos predicados recursivos.

As optimizações possibilitadas pelo *unfolding* traduziram-se numa aceleração da execução do código de cerca de 5 vezes. Isto constitui um excelente resultado, comparável ao obtido em sistemas como o Parma graças a uma crucial e complexa fase de interpretação abstracta, baseada num domínio de análise extenso, e implementada em 4500 linhas de Prolog.

A forma como as instruções WAM foram implementadas em C permitiu também ganhar substancialmente em performance em relação à `wamcc`. A geração de uma só função em C, sem penalizações para saltos directos ou indirectos, e a geração de código *in-line*, minimizando as chamadas a funções, ao contrário da `wamcc` em que cada instrução WAM é implementada por uma função em C, permitiu acelerar substancialmente a execução do código. Esta forma como o código C é gerado aproxima-se de uma implementação em código nativo.

No entanto, esta implementação em C é também a grande fonte de incompletude deste trabalho. A implementação apenas parcial quer dos *built-ins* quer das instruções WAM, fazem com que muitos programas ainda não possam ser compilados.

7.1 Contribuições da tese

Uma contribuição importante desta tese é a de que é possível obter um compilador simples que seja simultaneamente eficiente. Demonstra-se ainda que a geração de código C como código final não inibe a obtenção dessa eficiência. Sistemas como a `wamcc` tornam-se assim mais interessantes, constituindo uma alternativa para futuras implementações de Prolog.

No entanto, a maior contribuição desta tese é na forma em como esta eficiência é obtida. Até agora a obtenção de código especializado era suportada por uma fase de interpretação abstracta, para detectar propriedades do programa que permitissem essa especialização do código. Neste trabalho mostrou-se uma forma de obter código especializado apenas pela propagação de informação presente nas instruções WAM, reduzindo o uso de interpretação abstracta a um algoritmo simples para inferência dos tipos dos argumentos dos predicados à saída. Isto traz grandes implicações pois são conhecidas as dificuldades da interpretação abstracta em programas de maior dimensão.

7.2 Trabalho futuro

Em termos de trabalho futuro é importante expandir o número de programas testados, e em particular testar aplicações de dimensão real. Isto implica desenvolver o tradutor de WAM para C. Será importante verificar os resultados do *unfolding* em programas onde a interpretação abstracta dá maus resultados.

Para tirar o máximo partido da informação disponibilizada pelo *unfolding* será preferível a geração directa de código nativo sem passar pelo passo intermédio da geração de C. O sistema resultante deverá competir com os sistemas mais rápidos existentes, visto que a geração directa de código nativo possibilita optimizações que se traduzem numa boa aceleração do tempo de execução do código.

Em termos da geração de código intermédio e de melhorias do *unfolding* será importante acrescentar um elemento ao domínio de análise da interpretação abstracta para inferência dos tipos dos argumentos à saída. Este tipo será o *deref* que garantirá que se à entrada uma determinada variável era desreferenciada então à saída também é desreferenciada. Este é um problema complexo, mas que traz alguns ganhos, pois segundo Van Roy, o número de variáveis desreferenciadas deduzidas pela fase de análise do seu compilador é de 17%.

A informação deduzida pelo *unfolding* poderá, para além de de ser usada para a especialização do código gerado a ser invocado através da detecção de casos especiais, servir também para a especialização das próprias estruturas de dados envolvidas, levando, por exemplo, ao uso de pontos de escolha especializados em predicados recursivos que não necessitem de variáveis locais, ou de versões especializadas de predicados que utilizem a representação nativa para os seus argumentos inteiros.

Referências

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction.* MIT Press, 1991.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[CD95] Philippe Codognet and Daniel Diaz. wamcc: Compiling Prolog to C. In *12th International Conference on Logic Programming.* MIT Press, 1995.

[CFY93] Takashi Chikayama, Tetsuro Fujise, and Hiroshi Yashiro. A portable and reasonably efficient implementation of KL1. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, page 833, Budapest, Hungary, 1993. The MIT Press.

[CM81] William F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Cov93] Michael A. Covington. ISO Prolog: A Summary of the Draft Proposed Standard. Technical note, University of Georgia, 1993.

[Dia94] Daniel Diaz. *wamcc 2.21 User's Manual*, July 1994.

[Dob87] T.P. Dobry. *A High Performance Architecture for Prolog.* PhD thesis, University of California at Berkeley, April 1987.

[DSCRA89] L. Damas, V. Santos Costa, R Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1989.

[GDBS92] D. Gudeman, K. De Bosschere, and Debray S. jc: An efficient and portable sequential implementation of janus. In *Joint International Conference and Symposium on Logic Programming*, Washington, 1992. "MIT Press".

[GLLO85] J. Gabriel, T. Lindholm, E. L. Lusk, and R. A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Research Paper ANL-84-84, Argonne National Laboratory, June 1985.

[Hau94] B. Haussman. Turbo Erlang: Approaching the Speed of C. In Evan Tick, editor, *Implementations of Logic Programming Systems*. Kluwer, 1994.

[Hol93] B.K. Holmer. *Automatic Design of Computer Instruction Sets*. PhD thesis, University of California at Berkeley, 1993.

[KN90] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *International Workshop in Programming Language Implementation and Logic Programming (PLIP'90)*, number 456 in Lecture Notes in Computer Science, pages 121–135. Sweeden, Springer-Verlag, 1990.

[LH94] M.R. Levy and R.N. Horspool. Translating Plolog to C: a WAM-based approch. Technical note, University of Victoria, 1994.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[Lop96] R. Lopes. Execução de Prolog com Alto Desempenho. Master's thesis, Universidade do Minho - Departamento de Informática, July 1996.

[Sta93] Richard M. Stallman. Using and porting gcc. Technical report, The Free Software Foundation, 1993.

[Tay91] A. Taylor. *High Performance Prolog Implementation*. PhD thesis, University of Sydney, June 1991.

[Van90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[Van94] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming*, 19/20, May/July 1994.

[War77] D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

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