Execução de Prolog com Alto Desempenho

Ricardo Nuno de Sousa Lopes

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

Dedicado à minha esposa e aos meus pais.

Agradecimentos

Desejo agradecer a colaboração e o apoio prestado por todos aqueles que em diferentes momentos e a diversos níveis contribuíram para uma melhor prossecução deste trabalho.

Em particular, desejo agradecer todo o trabalho de orientação do Prof. Doutor Vítor Santos Costa, pelo modo como a todos os níveis acompanhou e orientou todo o trabalho, bem como pela sua disponibilidade e atenção permanentes.

Gostaria também de agradecer ao Prof. Doutor Fernando Silva pela disponibilidade e pela paciente colaboração durante este trabalho, assim como agradeço ao Prof. Doutor Luís Damas pela atenção concedida em vários momentos deste trabalho.

Um agradecimento também especial ao Prof. José Maia Neves, pela 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 exprimir a minha gratidão ao LIACC e ao projecto PROLOPPE (PRAXIS 3/3.1/TIT/24/94) 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/2094/94.

Pretendo testemunhar a minha amizade ao Michel e ao Ricardo Rocha, que me acompanharam ao longo de todo este trabalho.

Desejo agradecer à minha esposa Vera pelo seu constante apoio e por estar sempre ao meu lado nos momentos mais difíceis.

Quero agradecer aos meus queridos pais, Albano e Fernanda, sem os quais este trabalho não teria sido possível, que sempre me apoiaram durante todo o tempo.

E por fim, agradeço a todos os meus colegas e amigos cujos nomes me é impossível citar, por falta de espaço, e que me ajudaram nas horas de maior dificuldade.

Resumo

Um dos grandes objectivos de qualquer implementação de Prolog é a de conseguir obter o máximo de performance possível na execução das aplicações. A compilação avançada é um dos factores essenciais para conseguir uma boa performance. Nos últimos anos presenciaram-se melhorias significativas na compilação do Prolog, resultando no retorno da ideia da execução em código nativo.

Nesta tese apresenta-se o sistema Native YAP, um sistema com geração de código nativo para a arquitectura Alpha, que usa como base o compilador de Prolog YAP.

O desenho do Native YAP envolveu um conjunto de novos passos para a compilação de Prolog, nomeadamente a aplicação do algoritmo 2-Streams na YAAM, a criação de uma nova linguagem intermédia para compilação avançada de Prolog (YAIL), optimizações para esta última linguagem, a geração de código nativo para a arquitectura Alpha, e um método para integrar o ficheiro objecto resultante da compilação do código nativo no YAP em tempo de execução.

O Native YAP obteve bons índices de performance comparativamente ao YAP, reforçando a ideia de que uma geração cuidada de código nativo para um programa Prolog pode trazer melhorias substanciais na execução relativamente às versões baseadas em emuladores e implementações com geração de código *C*.

Abstract

A major goal for any Prolog implementation is to attain very high performance for applications. One way to achieve this is through the use of advanced compilation techniques. The last few years have seen improvements in Prolog compilation technology resulting in a return to the idea of native code execution.

This thesis presents the Native Yap, a system capable of generating native code for the Alpha architecture. The new system is based on the pre-existing YAP Prolog compiler.

The design of Native Yap included the development of several new stages in the Prolog compilation, including the implementation of the 2-Streams algorithm in the YAAM, a new intermediate language for advanced Prolog compilation (YAIL), several optimisation phases to this new language, generation of native code for the Alpha architecture, and a method to integrate the resulting executable into YAP in run-time.

Preliminary results with the Native YAP system have shown that it achieves good performance gains comparing to the YAP system. These results substantiate our belief that in comparison with the emulated and *C* versions of Prolog compilers, a good native code generation strategy can significantly improve execution time.

Índice

Lista de Figuras 10

Lista de Tabelas 12

1 Introdução 13

1.1 Motivação 13

1.2 Objectivos 14

1.3 Contribuição da tese 15

1.4 A arquitectura alvo 15

1.5 Prolog 16

1.6 Estrutura da tese 17

2 Implementações de Prolog 19

2.1 Warren Abstract Machine 19

2.1.1 Organização da memória 20

2.1.2 Os registos da WAM 21

2.1.3 Representação de dados em memória com etiquetas 22

2.1.4 Instruções da WAM 23

2.1.5 Converter o Prolog para WAM 25

2.2 O algoritmo 2-Streams 26
2.3 O sistema YAP 29
2.3.1 Áreas de trabalho 30
2.3.2 Os registos 31
2.3.3 Representação de dados em memória com etiquetas 31
2.3.4 A YAAM 32
2.3.5 Compilação de um programa Prolog para YAAM 34
2.4 Implementações de Prolog com código nativo 35
2.4.1 Aquarius 35
2.4.2 Parma 36
2.4.3 SICStus Native 37

3 Novas linguagens intermédias para a compilação de Prolog 38

3.1 A YAAM 2-Streams 38
3.1.1 Registos da YAAM 2-Streams 39
3.1.2 Descrição das instruções da YAAM 2-Streams 39
3.1.3 Transformação do código YAAM em YAAM 2-Streams 40
3.1.4 Exemplo 1 43
3.1.5 Exemplo 2 44
3.1.6 Exemplo 3 45
3.1.7 Uso da YAAM 2-Streams 48
3.2 YAIL - Yet Another Intermediate Language 49
3.2.1 Descrição das instruções da YAIL 50
3.2.2 Organização dos registos na YAIL 52
3.2.3 Esquema de etiquetas escolhido para usar na arquitectura Alpha . . 53

3.2.4 Um exemplo 55

3.2.5 Comparação da YAIL com outras linguagens intermédias 56

4 O Sistema Native YAP **58**

4.1 Esquema de compilação do NYAP 59

4.1.1 Transformação do Prolog para YAAM 60

4.1.2 Transformação da YAAM para YAAM 2-Streams 60

4.1.3 Transformação da YAAM 2-Streams para YAIL 60

4.1.4 Optimização do código YAIL 61

4.1.5 Atribuição de registos 66

4.1.6 Geração do código assembly para as cláusulas 68

4.2 Compilação do código assembly gerado e ligação do executável resultante com o YAP 72

5 Performance do sistema NYAP **77**

5.1 Programas de teste 77

5.1.1 Performance do Native Yap versus Yap 78

5.1.2 Análise da performance do sistema Native YAP 79

5.1.3 Código nativo versus código *C* 81

5.1.4 Outros factores 84

5.2 Comparação de performance com o SICStus 85

6 Conclusões **86**

6.1 Síntese do Trabalho 87

6.2 Trabalho Futuro 89

6.3 Nota de Fecho 89

Referências **90**

A Os Programas de teste **92**

B Fases de compilação de um programa Prolog **99**

Índice de Figuras

1.1 Exemplo de um programa Prolog 17

2.1 Execução de Prolog usando WAM. 20

2.2 Áreas de memória da WAM 21

2.3 Representação de termos na WAM. 23

2.4 Compilação de Prolog para WAM 26

2.5 Uma forma simples de separar o código em duas áreas. 27

2.6 Disposição das áreas de memória no YAP. 30

2.7 Código WAM e YAAM para a estrutura `a(a(X,Y),X)`. 32

2.8 Código WAM e YAAM para a estrutura `a(a(X),a(X))`. 33

3.1 Algoritmo para transformar o código YAAM em YAAM 2-Streams. 42

3.2 Código YAAM para `estudante(menino(tiago))`. 43

3.3 Código YAAM 2-Streams para `estudante(menino(tiago))`. 43

3.4 Código YAAM para `append([X|L1],L2,[X|L3]) :- append(L1,L2,L3)`. 44

3.5 Código YAAM 2-streams da cláusula `append`. 45

3.6 Código YAAM para `a(b(c(d([x,y]),e(w))))`. 46

3.7 Código YAAM 2-Streams para `a(b(c(d([x,y]),e(w))))`. 47

3.8 Código YAAM para a estrutura `irmão(manuel,tiago)`. 50

3.9 Código *Pseudo-C* da instrução `put_unsafe`. 55
3.10 Código YAIL para a instrução `put_unsafe(Y[i],X[j])`, forma 1. 56
3.11 Código YAIL para a instrução `put_unsafe(Y[i],X[j])`, forma 2. 56

4.1 Fases de compilação do Native Yap 59
4.2 Tranformação da instrução `put_list` em instruções YAIL. 61
4.3 Optimização do código YAIL 62
4.4 Eliminação de instruções `label`. 62
4.5 Eliminação de código morto. 63
4.6 Redução de saltos. 63
4.7 Eliminação de saltos. 64
4.8 Substituição de instruções `jump` 64
4.9 Eliminação de atribuições não usadas. 64
4.10 Redução de atribuições para o mesmo registo. 65
4.11 Alteração da ordem das instruções. 65
4.12 Bloco de código por optimizar 66
4.13 Bloco de código já optimizado 66
4.14 Esquema para o código da instrução `deref(X)`. 69
4.15 Evitando saltos para endereços não alinhados 70
4.16 Novos `labels` gerados em cada cláusula 71
4.17 Organização do código de todas das cláusulas numa só função 73
4.18 Extrato do algoritmo de compilação do ficheiro com o código das cláusulas 73
4.19 Instruções na máquina abstracta para executar a próxima instrução 74
4.20 Ilustração da execução de uma cláusula (P2) 76

Índice de Tabelas

2.1 Código WAM para a unificação W=f(g(A),h(B)) 28

2.2 Código 2-Streams para a unificação W=f(g(A),h(B)). 29

3.1 Transformações na YAAM para obter código de leitura e escrita separado . 41

3.2 Instruções que acedem a posições de memória. 50

3.3 Instruções simples que trabalham apenas com registos. 51

3.4 Condições existentes para as instruções de comparação 51

3.5 Instruções complexas 52

3.6 Instruções relacionadas com a implementação de etiquetas. 52

3.7 Equivalência entre os registos da YAIL e da YAAM 53

5.1 Aceleração do NYAP sobre o YAP 79

5.2 Aceleração de várias versões do NYAP sobre o YAP 81

5.3 Tamanho do ficheiro objecto resultante da compilação de código nativo . . 82

5.4 Aceleração do YAPC sobre o YAP 83

5.5 Tamanho do ficheiro objecto, NYAP vs YAPC 83

5.6 Percentagem de tempo no emulador durante a execução no NYAP. 84

5.7 Aceleração do código Nativo sobre o emulador, SICStus vs YAP 85

Capítulo 1

Introdução

1.1 Motivação

A primeira implementação de Prolog foi um interpretador escrito por Roussel e Colmerauer no início dos anos 70 [CKPR73]. Em 1977, David Warren criou o primeiro compilador de Prolog, o DEC-10 Prolog [War77], que gerava código assembly para o DEC-10, estabelecendo desta forma o Prolog como uma linguagem prática, de performance comparável com o LISP. A investigação na implementação de Prolog continuou e em 1983 David Warren criou a WAM (Warren Abstract Machine) [War83], uma linguagem intermédia para a compilação do Prolog. A WAM oferecia várias vantagens como fácil compilação, portabilidade, boa performance e código compacto. Por essas razões, a WAM revelou-se desde cedo como sendo uma forma eficiente e elegante de permitir a execução de programas Prolog numa máquina sequencial, tornando-se rapidamente como modelo norma para implementações de Prolog.

A motivação original no desenho de máquinas abstractas era a construção de hardware para suportar Prolog eficientemente. Porém as instruções WAM realizam operações muito complexas, como seja a unificação, e o desenvolvimento das novas arquitecturas tem caminhado na direcção oposta [Hwa93], aparecendo cada vez mais arquitecturas com um pequeno número de instruções simples. As dificuldades em obter performance superior à possível em arquitecturas tradicionais e o custo de desenvolver hardware para suportar Prolog justificam que a WAM tenha sido principalmente usada em implementações por software. Estas implementações, tradicionalmente compilavam o código Prolog para o código de uma máquina abstracta que era depois emulado em tempo de execução. Como exemplo deste tipo de aproximação temos o Quintus [Qui86], YAP [DSCRA89], e o SICStus Prolog [Car87]. Contudo este tipo de sistemas não aproveitam ao máximo o desempenho da máquina. São duas as razões principais para tal facto, o *overhead* intrínseco em emular as instruções de outra arquitectura, e a complexidade das instruções, que não permitem que muitas optimizações se possam realizar.

Mais recentemente, sistemas como o Aquarius [Van90] e o Parma [And91] provaram que as linguagens lógicas podem ter desempenho comparável ao das linguagens imperativas. O bom desempenho destes sistema deve-se aos seguintes princípios:

- **Geração de código nativo**: foram criadas novas máquinas abstractas constituídas por instruções simples, mais próximas das instruções de uma máquina sequencial.
- **Análise global do programa**: permitindo especializar a unificação e explorar o determinismo em programas Prolog.

A geração directa de Prolog para código nativo é um problema complexo. Usualmente o problema é solucionado organizando a compilação num conjunto de fases especializadas. Um exemplo é o SICStus Prolog que transforma o código Prolog em código WAM, para depois o transformar numa nova linguagem intermédia, a SAM [Hay94].

Uma alternativa à geração directa de código nativo, é a geração de código *C* a partir do Prolog [LH94]. Um exemplo deste tipo de implementações é a `wamcc` [CD95]. A filosofia subjacente a este sistema, é deixar o trabalho de optimização para o compilador de *C*. Infelizmente o desempenho deste sistema não é muito impressionante, mesmo comparando com sistemas que usam emuladores.

1.2 Objectivos

O principal objectivo desta tese consiste na implementação de código nativo no YAP, desenvolvendo um método eficiente para tal. O YAP é um compilador de Prolog baseado na WAM. O código Prolog é compilado para YAAM (Yet Another Abstract Machine) [SC88], uma variante da WAM, que por sua vez é interpretado durante a execução.

Uma importante consideração no desenho do sistema é que ele deverá integrar-se transparentemente no sistema existente, permitindo que o vasto conjunto de predicados do YAP continue a ser usado. Em geral, o utilizador deverá poder escolher entre usar o código nativo ou código emulado.

Note-se que a geração ingénua de código nativo pode levar a vários problemas, particularmente resultar em excessivo tamanho de código conduzindo a pior localidade e eventualmente prejudicando o desempenho. Assim, um dos requesitos para a implementação do sistema é o de balançar as vantagens de usar código nativo com a necessidade de preservar o código compacto, possivelmente usando subrotinas.

1.3 Contribuição da tese

A principal contribuição desta tese é o desenvolvimento de um sistema de geração de código nativo usando como base o YAP. O novo sistema inclui:

- separação do código YAAM em duas áreas, uma com código em modo de leitura e a outra com o código em modo de escrita.
- o desenvolvimento de uma nova linguagem intermédia, a YAIL, capaz de suportar melhor a geração de código nativo.
- uma fase para optimização do código YAIL que, entre outros, elimina código desnecessário e reordena algumas instruções de forma a eliminar atribuições redundantes.
- geração de código assembly para a arquitectura Alpha a partir do código YAIL, usando sempre que possível os registos disponíveis na arquitectura de forma a reduzir o número de acessos à memória.
- integração do ficheiro objecto resultante da compilação do código assembly no YAP em tempo de execução, de forma a permitir que o código das cláusulas seja executado directamente pela máquina em vez de ser emulado.

1.4 A arquitectura alvo

Dado o nosso objectivo de suportar geração de código nativo para Prolog, tornava-se necessário escolher uma arquitectura alvo. A arquitectura escolhida foi a arquitectura

Alpha [Alp92]. A arquitectura Alpha é uma arquitectura *RISC* 64 bits, com acesso ao espaço de endereçamento de 43 bits na primeira versão. É superescalar, emissão de duas instruções por ciclo, com um pipeline de 7 níveis para operações inteiras e 10 níveis para operações de virgula flutuante [Hwa93].

A arquitectura Alpha dispõe de 32 registos inteiros e de 32 registos de virgula flutuante, ambos com o tamanho de 64 bits.

As instruções têm todas um tamanho fixo de 32 bits [Bha96]. Apenas as instruções `load` e `store` efectuam operações com a memória. Todas as outras operam sobre os registos.

As razões que nos levaram a escolher esta arquitectura para o protótipo do YAP com código nativo, foram:

- Ainda não existe nenhuma implementação de Prolog com suporte para geração de código nativo para a arquitectura Alpha. Desta forma o YAP será o primeiro a faze-lo.
- A Alpha é uma arquitectura de 64 bits, o que a torna bastante atractiva para implementações de Prolog, principalmente devido à necessidade do uso de etiquetas (TAGS).
- Com a evolução constante no mundo da informático, é de esperar que dentro de pouco tempo, a maioria das arquitecturas seja de 64 bits. Ao escolher a arquitectura Alpha, o YAP antecipou-se a essa evolução.

1.5 Prolog

Nesta secção será feita uma breve apresentação da linguagem Prolog. Informação mais completa pode ser obtida consultando [CM81]. Um programa Prolog (ver figura 1.1) é um conjunto de cláusulas de Horn [Llo87]. Um grupo de cláusulas com o mesmo nome e aridade define um predicado. Cláusulas podem ser constituídas por regras ou factos. As regras dividem-se em cabeça e corpo, onde a cabeça é a parte à esquerda do símbolo ':-' (leia-se *se*), e o corpo a parte à direita. O corpo por sua vez, é composto por objectivos, separados pelo símbolo ',' (leia-se *e*). Os factos são casos particulares das regras, em que o corpo é vazio.

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

Figura 1.1: Exemplo de um programa Prolog

Programas em Prolog trabalham sobre termos, que podem ser constantes, variáveis, ou termos complexos caracterizados por um nome, aridade, e os argumentos, que são por sua vez termos. Constantes podem ser números ou símbolos (átomos), representados por palavras em que a primeira letra é minúscula ou por palavras rodeadas por plicas. As variáveis, representadas por palavras em que a primeira letra é maiúscula, podem representar qualquer objecto. Inicialmente o valor das variáveis é desconhecido, podendo ficar conhecido através da instanciação. As variáveis só podem ser instanciadas uma vez, através da unificação. A execução falha se tentar instanciar uma variável já instanciada com outro valor.

A execução de Prolog tenta satisfazer um objectivo principal, isto é, verificar se é verdadeiro. A execução de uma cláusula consiste em testar o objectivo com a cabeça da cláusula. Se os dois unificarem, isto é, tiverem uma solução comum, Prolog tenta satisfazer cada um dos objectivos no corpo da cláusula seguindo a ordem esquerda para a direita. O Prolog tenta satisfazer as cláusulas seguindo a ordem pela qual estão dispostas no programa, e sempre que for invocado um predicado constituído por mais de uma cláusula é criado um ponto de escolha. Se a execução a certa altura falhar, o sistema recua para o ponto de escolha mais recente e tenta satisfazer a cláusula seguinte.

1.6 Estrutura da tese

Esta tese contém mais cinco capítulos. No próximo, capítulo 2, é feita uma apresentação da WAM, do algoritmo 2-Streams [Van94] o sistema YAP, e uma breve apresentação de alguns sistemas de Prolog que implementam código nativo.

No capítulo 3 apresentam-se duas novas linguagens intermédias para a compilação de Prolog que são usadas para obter código nativo eficiente.

No capítulo 4 descrevem-se as fases envolvidas durante a compilação de um programa Prolog para código nativo. É também apresentado o método usado para integrar o executável resultante da compilação com o YAP.

No capítulo 5 procede-se à análise do desempenho do sistema com código nativo usando para tal programas de teste conhecidos. A performance do sistema é comparada com a do YAP base e com uma versão do YAP com geração de código C.

No capítulo 6 apresentam-se as conclusões sobre o trabalho realizado, relembrando numa perspectiva global o mesmo. Perpectiva-se também algum trabalho futuro de continuação deste projecto.

No apêndice A, apresentam-se os programas Prolog usados para avaliar o novo sistema com código nativo, e no apêndice B dá-se um exemplo completo sobre a compilação de uma cláusula do programa de teste `nreverse.pl`. Para tal, apresenta-se o código das várias linguagens intermédias envolvidas na compilação, desde o Prolog até ao código nativo.

Capítulo 2

Implementações de Prolog

Neste capítulo será feita uma breve apresentação da WAM, seguindo-se de uma apresentação do algoritmo 2-Streams. Por último, serão apresentados alguns sistemas de Prolog, o sistema YAP que é o sistema usado como base no presente trabalho, e algumas implementações de código nativo.

2.1 Warren Abstract Machine

Em 1983, David Warren criou a WAM [War83], um modelo para a execução de Prolog que rapidamente se tornou norma nas implementações de Prolog. O relatório de Warren foi durante alguns anos, a única fonte de informação sobre a WAM. Nos dias de hoje já existem outros textos com explicações mais claras e simples sobre a WAM [GLLO85, AK91, Kog91].

São três as principais formas de implementação de Prolog usando a WAM (figura 2.1): implementações por hardware, implementações usando um emulador e implementações que traduzem o código WAM para código nativo. As implementações por hardware não tiveram muito sucesso devido à falta de mercado, pois são poucas as pessoas interessadas em comprar uma máquina apenas capaz de correr Prolog. Mesmo que o Prolog fosse muito popular, o hardware especializado teria muita dificuldade em conseguir competir pois os microprocessadores beneficiam de investimentos maiores. As implementações usando um emulador de WAM tornaram-se muito populares, pois são relativamente fáceis de implementar e têm um bom desempenho. As implementações de código nativo têm vindo a ganhar popularidade pois têm demonstrado serem melhores em termos de desempenho.

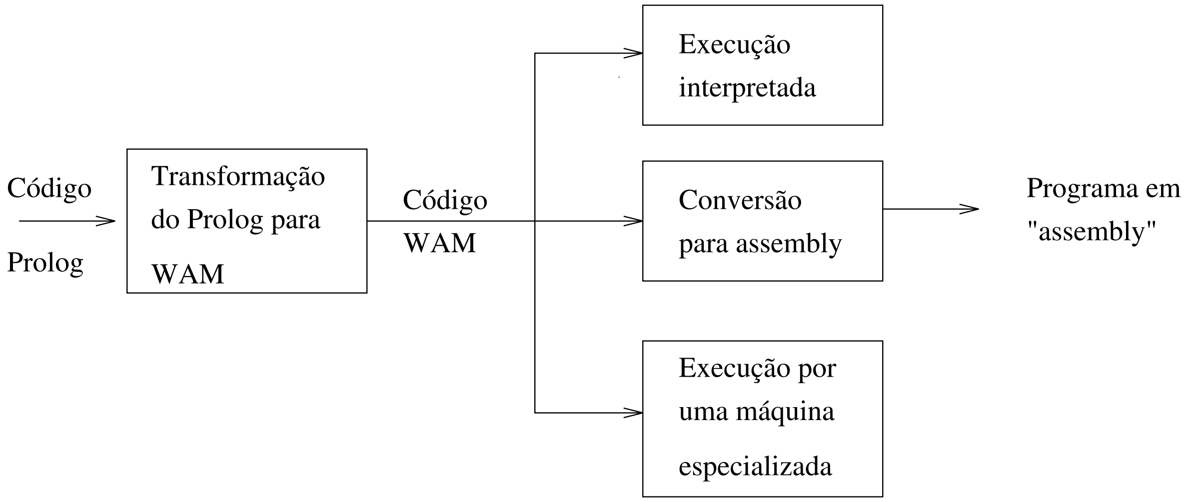

Figura 2.1: Execução de Prolog usando WAM.

2.1.1 Organização da memória

O estado externo da WAM é dividido em cinco áreas (figura 2.2):

- **pilha global**: nesta pilha são guardadas variáveis, termos compostos como listas, estruturas, e até mesmo átomos do programa Prolog.

- **pilha local**: esta pilha guarda os pontos de escolha e os ambientes. Nos ambientes são guardadas as variáveis locais para a cláusula. Nos pontos de escolha é guardada informação sobre o estado da execução para que se possa fazer retrocesso.

- **trilha**: nesta pilha é guardada informação acerca de variáveis cujo valor pode ser necessário recuperar aquando um retrocesso.

- **PDL(Push Down List)**: esta pilha é usada durante a unificação de termos compostos.

- **área de código**: nesta área é guardado o código do programa Prolog compilado.

Saliente-se que é possível mudar a organização das áreas de memória sem que haja alterações substanciais na execução. Por exemplo, alguns sistemas dividem a *pilha local* em duas, uma para os ambientes e outra para os pontos de escolha.

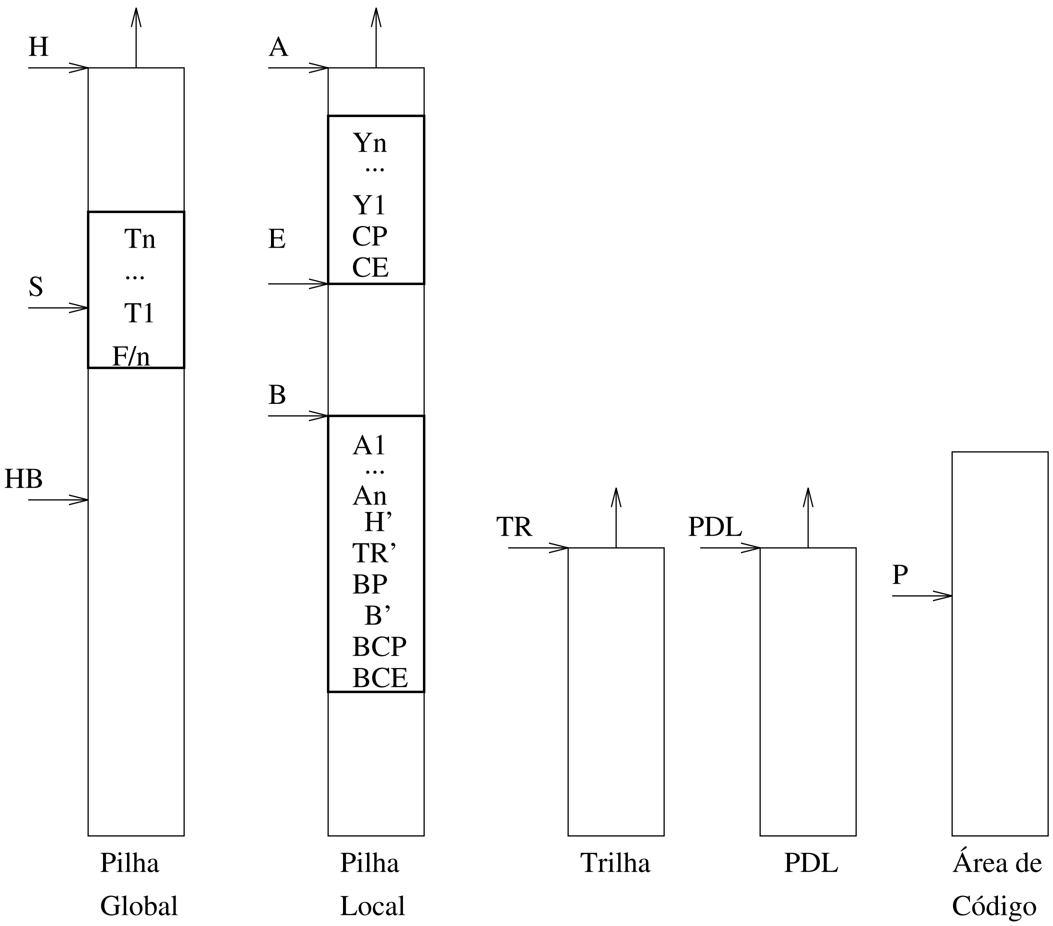

Figura 2.2: Áreas de memória da WAM

2.1.2 Os registos da WAM

O estado interno da WAM é guardado nos seguintes registos:

- **P**: localização do programa (*Program Counter*).
- **CP**: apontador para a continuação das cláusulas (*Continuation Pointer*).
- **E**: apontador para o ambiente actual.
- **B**: apontador para o último ponto de escolha.
- **A**: topo da pilha local.
- **TR**: topo da trilha.

- **H**: topo da pilha global (*Heap*).
- **HB**: valor de H no ponto de escolha mais recente.
- **S**: é usado pela unificação de termos compostos. Cada argumento do termo é acedido incrementando este registo.
- **Mode**: é usado para identificar o modo de execução de certas instruções, leitura ou escrita.
- **A1,A2,...**: registos dos argumentos.
- **X1,X2,...**: variáveis temporárias. Saliente-se que na WAM os Xi são equivalentes aos Ai.

Na WAM é usual a notação `Y[i]` para representar a i-gésima variável local da cláusula. Estas variáveis não fazem parte dos registos da máquina abstracta. Os `Y[i]` são elementos do ambiente actual (ver figura 2.2).

2.1.3 Representação de dados em memória com etiquetas

O Prolog é uma linguagem em que as variáveis são dinâmicas, ou seja, podem ter qualquer tipo durante a execução. Por essa razão é necessário que se possa determinar qual é o tipo de objecto que a variável representa em tempo de execução. Na WAM os termos são representados como palavras com etiquetas. As palavras consistem em duas partes: a etiqueta e o valor (ver figura 2.3). A etiqueta identifica o tipo do objecto. O valor é usado para diferentes propósitos conforme o tipo do objecto. Pode ter um valor inteiro, um identificador para átomos, ou um endereço de variáveis e termos complexos.

As variáveis indefinidas são representadas como sendo apontadores para elas mesmas. Como a semântica do Prolog permite que as variáveis se unifiquem com outras variáveis, são criadas cadeias de ligação de variáveis. Assim sendo, para saber o valor de uma variável é necessário seguir a cadeia até ao seu fim. Esta operação é designada por desreferenciação.

As estruturas são representadas em *n+1* palavras da máquina. A primeira indica o functor e a aridade do termo. As restantes *n* contêm os argumentos da estrutura.

As listas são representadas por pares, constituídos pela cabeça da lista mais o corpo da lista.

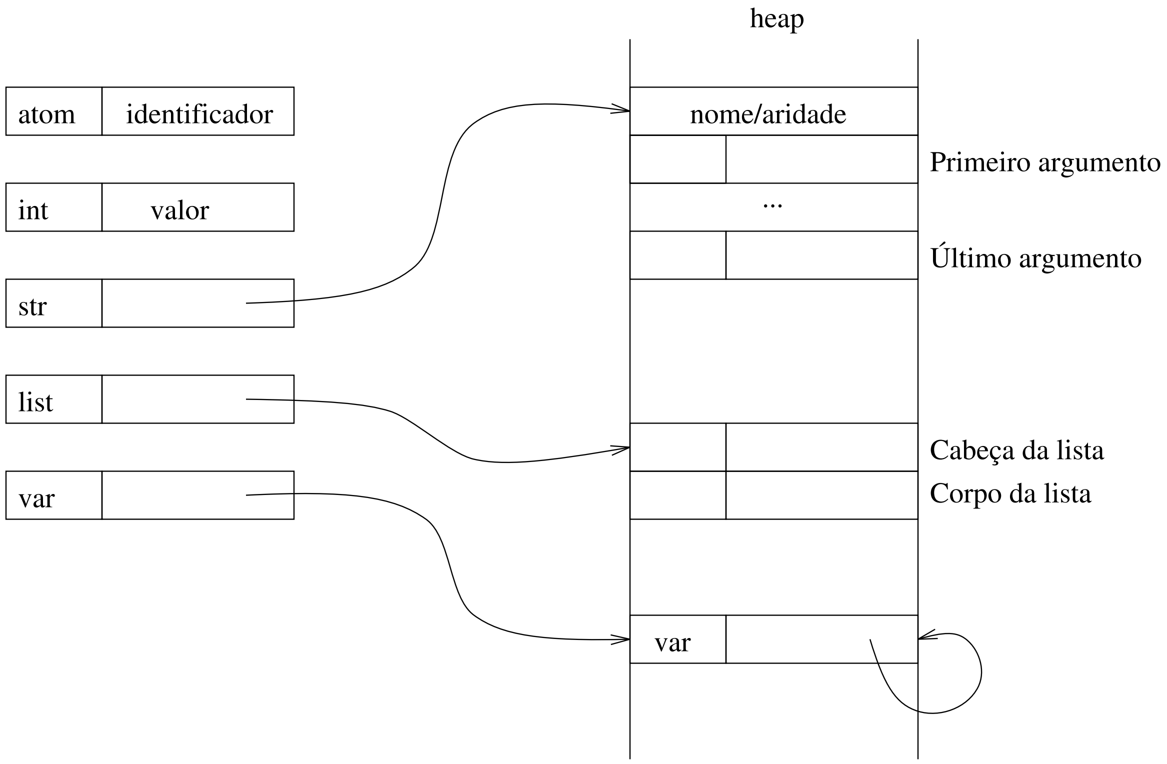

Figura 2.3: Representação de termos na WAM.

2.1.4 Instruções da WAM

Nesta secção apresentam-se as instruções básicas da WAM acompanhadas por uma breve descrição da operação de cada instrução. As instruções WAM podem ser divididas em cinco classes. Elas são (em algumas instruções é usada a notação Vn que denomina Xi ou Yi):

- **Instruções de procedimento** para controlar os pontos de escolha, o ambiente, e para transferir a execução para outras cláusulas.
  - **call P,N**: faz uma chamada ao predicado P de aridade N;
  - **execute P**: salta para o predicado P;
  - **proceed**: retorna de um predicado;
  - **allocate**: cria um ambiente na pilha local;
  - **deallocate**: retira o último ambiente da pilha local.
- **Instruções put**: actualizam os registos com os argumentos do próximo objectivo.

- **put_variable Vn,Ai**: cria uma nova variável e actualiza *Vn* e *Ai* com o seu valor;
- **put_value Vn,Ai**: copia *Vn* para *Ai*;
- **put_unsafe_value Yn,Ai**: copia *Yn* para *Ai* e globaliza *Yn*;
- **put_constant c,Ai**: copia *c* para *Ai*;
- **put_structure f,Ai**: inicializa uma estrutura e actualiza *Ai* com o seu valor;
- **put_list Ai**: cria um apontador para lista e actualiza *Ai* com o seu valor.

• **Instruções get** verificam se os argumentos de uma cláusula unificam com os actuais argumentos nos registos.

- **get_variable Vn,Ai**: copia *Ai* para *Vn*;
- **get_value Vn,Ai**: unifica *Vn* com *Ai*;
- **get_constant c,Ai**: unifica a constante *c* com *Ai*;
- **get_nil Ai**: unifica a constante *nil* com *Ai*;
- **get_structure f/n,Ai**: unifica a estrutura *f/n* com *Ai*, e actualiza o registo `Mode` para escrita ou leitura conforme o resultado da desreferenciação de `Ai` seja variável ou não;
- **get_list Ai**: unifica um apontador de lista com *Ai*, e actualiza o registo `Mode` para escrita ou leitura conforme o resultado da desreferenciação de `Ai` seja variável ou não.

• **Instruções unify**: para lidar com instruções `get` e `put` de elementos complexos, como sendo listas e estruturas.

- **unify_variable Vn**: copia o próximo elemento da estrutura para *Vn*;
- **unify_value Vn**: unifica *Vn* com o próximo elemento da estrutura;
- **unify_constant c**: unifica a constante *c* com o próximo elemento da estrutura;
- **unify_nil**: unifica a constante *nil* com o próximo elemento da estrutura;
- **unify_void n**: salta os próximos *n* elementos da estrutura.

• **Instruções de controle dos pontos de escolha** para controlar a sequência em que as cláusulas devem ser testadas ao executar um determinado predicado.

- **try Label**: cria um ponto de escolha e salta para Label;

- **retry Label**: actualiza o endereço no último ponto de escolha e salta para Label;
- **trust Label**: remove o último ponto de escolha e salta para Label;
- **try_me_else Label**: cria um ponto de escolha para Label e continua na instrução seguinte;
- **retry_me_else Label**: alteram o endereço no último ponto de escolha para Label e continua na instrução seguinte;
- **trust_me**: remove o ponto de escolha mais recente e continua na instrução seguinte.

• **Instruções de indexação** controlam qual a próxima cláusula a ser executada.

- **switch_on_term V,C,L,S**: é um salto decidido pelo tipo de termo de A1 para V se variável, C se constante, L se lista, ou S se termo complexo;
- **switch_on_constant N,Tbl**: caso *A1* seja uma constante faz um salto usando a tabela de Hash *Tbl* de tamanho *N*;
- **switch_on_structure N,Tbl**: caso *A1* seja uma estrutura faz um salto usando a tabela de Hash *Tbl* de tamanho *N*. Saliente-se que neste caso a indexação é realizada sempre pelo functor principal.

2.1.5 Converter o Prolog para WAM

A compilação de Prolog para WAM é relativamente simples, pois quase que existe uma transformação de um para um entre elementos no código de Prolog e instruções WAM. Na figura 2.4 apresenta-se um esquema para transformar predicados Prolog em WAM. Este código poderia ainda ser optimizado utilizando as instruções `switch` para evitar a criação de pontos de escolha em certos casos.

As cláusulas do predicado `p/3` são compiladas em blocos de código que são ligados pelas instruções `try` que controlam os pontos de escolha. Cada bloco consiste numa sequência de instruções `get`, para fazer a unificação dos argumentos da cabeça, seguido por uma sequência de instruções `put`, para preparar os argumentos de cada objectivo, e uma instrução `call` para executar o objectivo. No início e no fim do bloco estão as instruções `allocate` e `deallocate` que criam e retiram, respectivamente o ambiente para as variáveis permanentes.

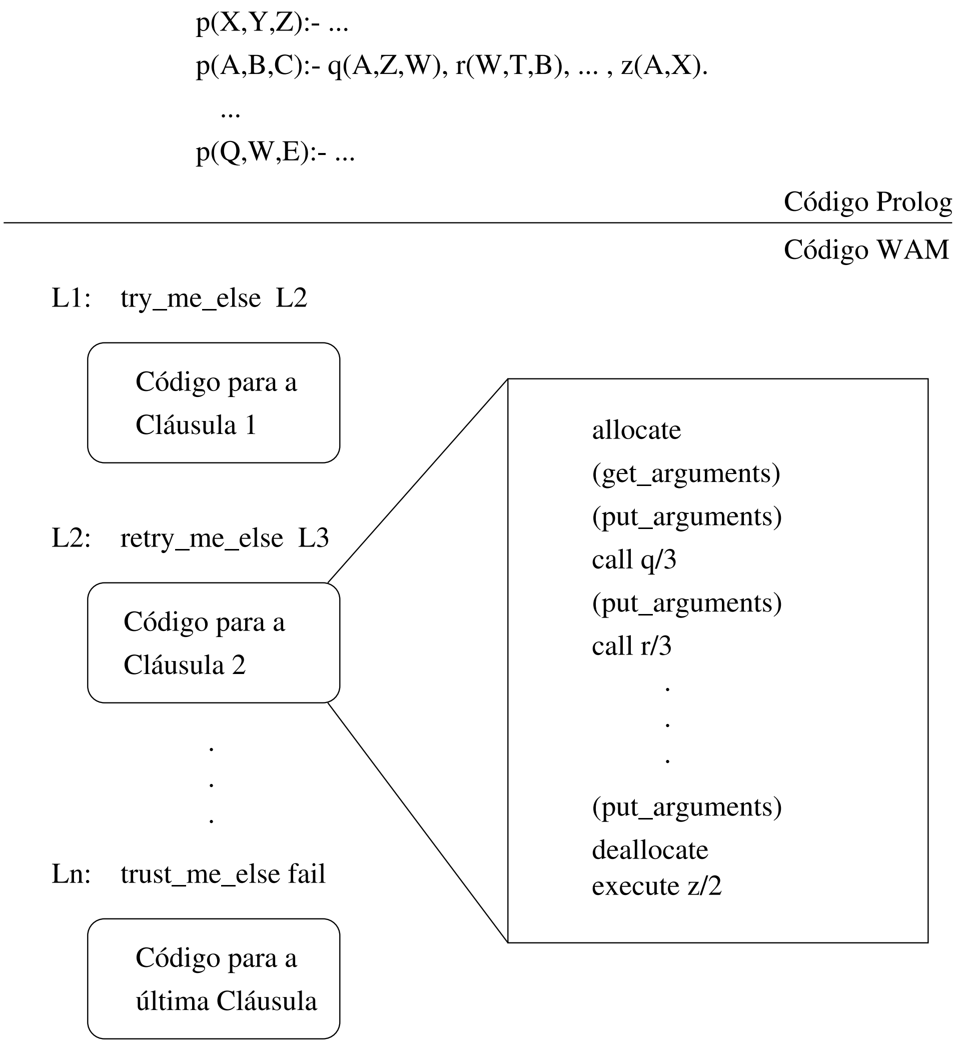

Figura 2.4: Compilação de Prolog para WAM

2.2 O algoritmo 2-Streams

A WAM compila a unificação numa simples sequência de instruções. Esta técnica tem alguns problemas:

- O modo de escrita não é propagado aos subtermos. Por exemplo, a unificação de M=h(f(a)) é compilada em duas unificações independentes, como M=h(N), N=f(a). Ora, se a variável M ainda não foi inicializada, N será também uma variável livre, facto este que não é propagado para a segunda unificação. Isto obriga a algumas operações desnecessárias, como seja a desreferenciação de N, a sua atribuição (binding) e verificação na trilha.

- As instruções têm dois modos de execução, modo de leitura e modo de escrita. O modo corrente é guardado num registo, que é inicializado na instrução `get_structure` ou na instrução `get_list` e verificado em todas as instruções `unify`.
- Pobre tradução para código nativo. Como as instruções `unify` têm dois modos o código de escrita e o de leitura ficam intercalados, resultando num grande número de saltos (`jumps`).

O algoritmo 2-Streams [Van94] constitui um elegante método de fazer unificação mais eficiente do que a WAM. A ideia é compilar a unificação em duas áreas de código, uma para o código de escrita e outra para o código de leitura, com saltos condicionais entre elas (ver figura 2.5).

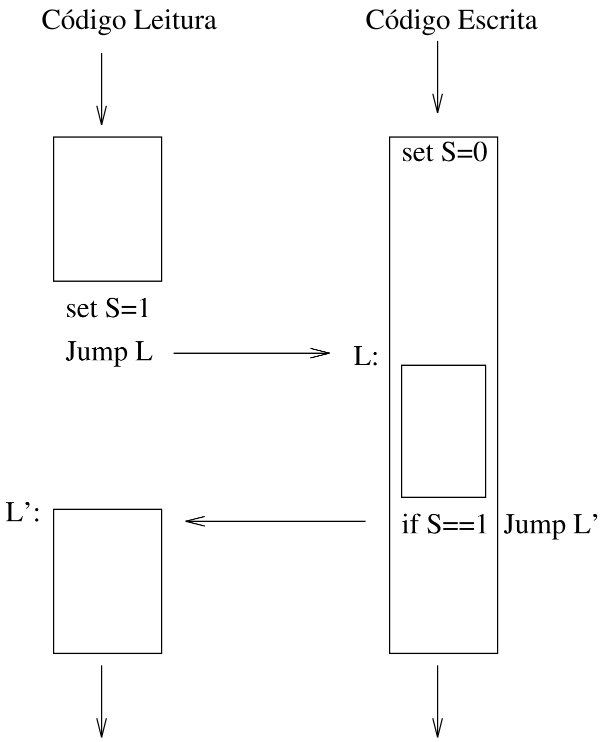

Figura 2.5: Uma forma simples de separar o código em duas áreas.

Durante a execução a unificação segue o código em modo de leitura saltando para o código de escrita quando os subtermos precisem de ser criados. Porém, antes de efectuar o salto do código de leitura para o código de escrita, uma variável, chamemos-lhe `S`, deverá ser actualizada com um identificador que permita que no código de escrita se possa determinar o momento em que se deverá retornar ao código de leitura. Este identificador pode ser um inteiro e deve ser criado de forma a reflectir o nível da profundidade da estrutura em questão. Uma forma de atribuir esse identificador é a de começar por referir a parte principal da estrutura por um inteiro `n`. Em seguida para cada um dos termos da estrutura é atribuído o inteiro seguinte, `n+1`. Para os subtermos dos termos da estrutura atribui-se o inteiro `n+2`, e por aí em diante. Por exemplo, a estrutura `f(h(a),b)` poderia ficar identificada da seguinte forma, `S=1` para o nível de `f()`, `S=2` para identificar o nível de `h()` e `b`, e `S=3` para o nível de `a`. Assim, durante a execução no código de escrita, só é necessário fazer uma verificação do valor de `S` para saber se a execução deverá regressar ao código de leitura.

Instruções WAM	**Operações**
`get_structure f/2,A1`	`W=(f/2)`
`unify_variable X4`	`W.1=U`
`unify_variable X5`	`W.2=Z`
`get_structure g/2,X4`	`U=(g/1)`
`unify_value A2`	`U.1=A`
`get_structure h/2,X5`	`Z=(h/1)`
`unify_value A3`	`Z.1=B`

Tabela 2.1: Código WAM para a unificação W=f(g(A),h(B))

As tabelas 2.1 e 2.2 apresentam a compilação para código WAM e para código 2-Streams, respectivamente, da unificação `W=f(g(A),h(B))`. Para melhor percepção do código as instruções WAM foram substituídas por uma notação em tipo de operações, as instruções do tipo `W=(f/2)` para representar as instruções `get` dos functores, e as instruções to tipo `W.1=U` para representar as instruções de unificação dos argumentos.

Note-se que o código da tabela 2.2 está conforme a descrição original por Peter Van Roy [Van94], mas pode ser melhorado, por exemplo agrupando as duas instruções `if S=2 jump Lz'` e `if S=1 jump Lx'` numa só, `if S>=1 jump Lx'`. Outra melhoria, não tão evidente, reside em criar um novo label `Lu'` antes da instrução `Z=(h/1)` no código de leitura e substituir a instrução `if S=2 jump Ly'` do modo de escrita, por uma nova instrução `if S=2 and notvar(Z) jump Lu'`.

Vantagens deste método:

- Propagação do modo de escrita. O modo de escrita de um termo é propagado para todos os seus subtermos durante a compilação. Não existem assim desreferenciações, verificações da trilha ou atribuições (*bindings*) desnecessários.

Instruções do modo de leitura	Instruções do modo de escrita
`if var(W) set S=1, jump Lx`	`Lx:`
`W=(f/2)`	`W=(f/2)`
`W.1=U`	`W.1=U`
`W.2=Z`	`W.2=Z`
`if var(U) set S=2, jump Ly`	`Ly:`
`U=(g/1)`	`U=(g/1)`
`U.1=A`	`U.1=A`
`Ly'`	`if S=2 jump Ly'`
`if var(Z) set S=2, jump Lz`	`Lz:`
`Z=(h/1)`	`Z=(h/1)`
`Z.1=B`	`Z.1=B`
`Lz'`	`if S=2 jump Lz'`
`Lx'`	`if S=1 jump Lx'`

Tabela 2.2: Código 2-Streams para a unificação W=f(g(A),h(B)).

- Tamanho do código linear. O número de instruções geradas é cerca do dobro das geradas pela WAM, mas as instruções têm metade (ou menos) da complexidade.
- Expansão para código nativo mais eficiente. Ora como o código de cada instrução é especifico para modo de leitura ou escrita, não necessita de ter saltos dentro das instruções `unify` para escolher o modo, pois este é seleccionado pelas instruções `get`, que fazem o salto para o código a ser executado. Saliente-se que um salto é uma operação com penalização muito elevada em termos de execução nas máquinas *RISC* actuais.

Este algoritmo obtém melhores resultados nas implementações de código nativo do que nas implementações baseadas em emulador, já que os emuladores têm que efectuar de qualquer das formas um salto no final das instruções da máquina abstracta. A maior vantagem do algoritmo 2-Streams para os emuladores é a propagação do modo de escrita. Contudo existem sistemas que fazem essa propagação do modo de escrita de outras formas. Um desses casos é o YAP.

2.3 O sistema YAP

O YAP [DSCRA89] é um compilador de Prolog desenvolvido na Universidade do Porto altamente portável, preparado para correr em várias arquitecturas e diferentes sistemas operativos. Como muitas outras implementações, o código Prolog é compilado para uma máquina abstracta, a YAAM (Yet Another Intermediate Language) [SC88], sendo depois emulado. A YAAM é uma linguagem intermédia baseada na WAM com algumas melhorias.

Nas secções que se seguem serão descritas as características do YAP que são relevantes para esta tese.

2.3.1 Áreas de trabalho

À semelhança dos sistemas que se baseiam na WAM, o YAP dispõe de cinco pilhas para área de trabalho (figura 2.6). Elas são:

- **Área de código** ou **Heap**: nesta área é guardada informação estática, ou seja, informação cujo tempo de vida não depende da execução do programa. Essencialmente corresponde a informação sobre átomos, predicados e código associado, e também à base de dados interna.
- **Pilha Local**: na pilha local são guardados os pontos de escolha e os ambientes.
- **Pilha Global** ou **Pilha de Termos**: esta pilha serve para guardar informação com tamanho variável (termos complexos), e variáveis cuja vida pode ultrapassar a vida do ambiente onde elas foram criadas.
- **Trilha**: nesta área mantêm-se informação necessária ao retrocesso, nomeadamente para recuperar o estado de variáveis.
- **Área Auxiliar**: esta área inclui uma pilha auxiliar usada pelo compilador, algoritmo de indexação e base de dados interna. Inclui também uma pilha de unificação usada internamente para unificação de estruturas.

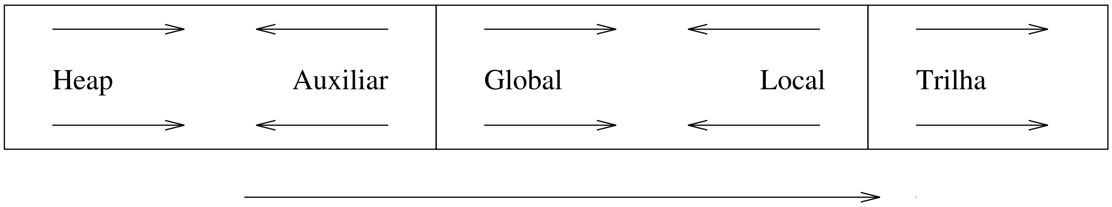

Figura 2.6: Disposição das áreas de memória no YAP.

2.3.2 Os registos

Os principais registos usados na YAAM são:

- **P**: localização do programa, ou Program Counter.
- **CP**: apontador para a continuação da cláusula activa.
- **YENV**: apontador para o ambiente actual.
- **H**: apontador para a pilha global.
- **S**: usado pelo algoritmo de unificação para percorrer estruturas.
- **B**: apontador para o último ponto de escolha.
- **HB**: valor de H na altura do último ponto de escolha. É usada pelo algoritmo de unificação para verificar se um variável deverá ser incluída na trilha.
- **TR**: apontador para o topo da trilha.
- **N**: número de argumentos do objectivo actual.
- **A1...AN**: argumentos da cláusula em execução.
- **Reading**: é usado para identificar o modo de execução de certas instruções, leitura ou escrita.

Saliente-se que na prática a YAAM usa outros registos que se destinam a suportar instruções especiais, como o corte.

2.3.3 Representação de dados em memória com etiquetas

O YAP pode ser executado em diferentes arquitecturas com a representação dos dados em memória variando dependendo do tamanho da palavra da arquitectura em questão. Por essa razão, no esquema de etiquetas do YAP que a seguir se descreve deve-se ter em conta que o tamanho da palavra pode ser de 32 bits (família i386, SPARC, MIPS,...) ou de 64 bits (ALPHA).

- **0mr...00** endereço de uma célula.
- **0mr...00** endereço de uma célula a apontar para ela mesma.
- **1mr...01** apontador para uma lista.
- **1mr...10** apontador para uma estrutura.
- **1m11...00** valor inteiro.
- **1m10...00** valor de virgula flutuante.
- **1m01...00** referencia para um átomo.
- **1m00...00** referencia para a base de dados.

m e **r** representam bits reservados para serem usados pelo *garbage collector*.

2.3.4 A YAAM

A YAAM é a linguagem intermédia do YAP, baseada na WAM. A principal diferença está no modo como é realizada a unificação de estruturas compostas. O código da WAM corresponde a uma visita em largura dos termos, enquanto que o código da YAAM corresponde a uma visita em profundidade (ver figura 2.7).

WAM	**YAAM**

```
WAM                          YAAM

get_struct Ai,a/2            get_struct Ai,a/2
        unify_var Xj                 unify_struct a/2
        unify_var Rk                         unify_var Rj
get_struct Xj,a/2                            unify_var Rk
        unify_val Rk                         pop
        unify_var Rm                 unify_val Rj
```

Figura 2.7: Código WAM e YAAM para a estrutura `a(a(X,Y),X)`.

A instrução `pop` é usada para recuperar o registo `S` e o registo `reading`. O `push` destes registos é efectuado pelas instrução `unify_struct` e `unify_list`.

A aproximação da YAAM tem algumas vantagens sobre a da WAM, nomeadamente:

- gera um número menor ou igual de instruções. Facilmente se conclui que uma instrução `pop` corresponde a uma instrução `unify_var` na WAM. Para além disso, na WAM é sempre necessário gerar os `unify_var` correspondentes a cada subestrutura enquanto na YAAM é possível agregar várias instruções `pop` na instrução `popn` e, em certos casos, é possível não os gerar. Por exemplo:

  A lista `[a,a,a]` dará origem ao seguinte código na YAAM:

```
get_list Xi
        unify_atom a
        unify_last_list
                unify_atom a
                unify_last_list
                        unify_atom a
                        unify_atom []
```

  Neste caso não é necessário gerar nenhuma instrução `pop` porque não há necessidade de continuar com uma das estruturas superiores. Saliente-se que as instruções `unify_last_list` e `unify_last_struct` diferem das instruções `unify_list` e `unify_struct` no facto de não efectuarem um `push` dos registos `S` e `reading`.

- uma outra vantagem diz respeito à propagação do modo de leitura ou de escrita. Por exemplo, considere-se a estrutura `a(a(X),a(X))` (figura 2.8).

WAM	**YAAM**
`get_struct a/2, Aj`	`get_struct a/2,Aj`
`        unify_var Rk`	`        unify_struct a/1`
`        unify_var Rl`	`                unify_var Ri`
`get_struct a/1,Rk`	`                pop`
`        unify_var Rm`	`        unify_struct a/1`
`get_struct a/1,Rl`	`                unify_val Ri`
`        unify_var Rm`	

Figura 2.8: Código WAM e YAAM para a estrutura `a(a(X),a(X))`.

Se o predicado `a/2` unificar com uma variável, entrando-se portanto no modo de escrita, a YAAM usa essa informação para simplificar a construção das duas subestruturas. Na WAM isso não é possível, o compilador tem que testar o valor de `Rk` para voltar a determinar se está em modo de escrita ou de leitura.

A YAAM difere também da WAM na execução das instruções `get_struct`, `get_list`, `unify_struct` e `unify_list`. Na YAAM se se passar para o modo de escrita durante a execução destas instruções, a heap é inicializada, facto que não acontecia na WAM. Com isso, a YAAM consegue evitar que as instruções `unify_void` e `unify_var` que aparecem no fim, sejam geradas. A desvantagem é que há casos onde essas instruções não aparecem, tornando a inicialização da heap numa operação supérflua.

Concluindo, a principal vantagem da YAAM está no uso da instrução `pop` que recupera o registo `reading` durante a unificação de estruturas, enquanto que na WAM é necessário realizar uma desreferenciação para decidir em que modo a execução vai continuar.

2.3.5 Compilação de um programa Prolog para YAAM

As cláusulas de um programa Prolog são compiladas isoladamente pelo YAP. Cada cláusula é transformada em código para máquina abstracta YAAM passando por cinco fases. Elas são:

- compilação da cabeça de uma cláusula;
- compilação do corpo de uma cláusula;
- classificação de variáveis e optimização das variáveis temporárias;
- eliminação das instruções `pop` e `unify_var` supérfluos;
- e por último, preparação do código para ser corrido pela máquina abstracta.

As instruções que fazem a ligação entre as cláusulas e as instruções de indexação só são geradas numa fase posterior, depois de ter todas as cláusulas do programa compiladas. O Yap ao compilar o código para as cláusulas separado do código de indexação, torna o processo de asserção de novas cláusulas relativamente simples. Esta é uma das vantagens dos sistemas de Prolog baseados em emuladores, a fácil implementação do `assert` e do `retract`. Nos sistemas com geração de código nativo a implementação destes predicados é mais complicada.

2.4 Implementações de Prolog com código nativo

As implementações de Prolog de código nativo têm vindo a ganhar popularidade porque têm vindo a conseguir aumentos significativos de performance na execução de programas Prolog. As implementações de código nativo podem ser divididas em duas grandes classes:

- Sistemas que compilam Prolog directamente para código nativo. Como exemplo deste tipo de aproximação temos o Aquarius e o Parma. A compilação directa para Prolog é um problema complexo. Usualmente este problema é resolvido dividindo a compilação em vários passos intermédios.

- Sistemas que geram o código nativo a partir da WAM. Este tipo de aproximação é bastante útil quando já existe à partida um sistema baseado na WAM, evitando que se crie um sistema a partir do zero. Como exemplo deste tipo de aproximação temos o SICStus Prolog.

2.4.1 Aquarius

O Aquarius é um exemplo de um sistema que evita a WAM. Este sistema utiliza a Berkeley Abstract Machine (BAM) [Van90] como linguagem intermédia na compilação de Prolog. A BAM na verdade é baseada nos mesmos princípios da WAM. Fundamentalmente a sua ideia base é a de dividir as instruções WAM num conjunto de instruções de granularidade mais reduzida. A BAM é uma linguagem de muito baixo nível, desenhada de forma a ser facilmente traduzida para uma arquitectura *RISC*.

A eficiência deste compilador é, segundo o autor, justificada em quatro princípios fundamentais:

- Redução da granularidade das instruções.

- Exploração do determinismo.

- Especialização da unificação.

- Análise global do programa.

A BAM, que foi criada de forma a satisfazer o primeiro principio, pode ser classificada por três classes de instruções :

- **Instruções simples** são instruções de suporte a comparações, atribuições, manipulação de pilhas, indexação, e instruções de controle (tipo `call` e `return`).
- **Instruções complexas** são instruções de suporte a operações comuns como desreferenciação, operações com a trilha, unificação, e retrocesso.
- **Instruções embutidas**. Estas instruções, *pragmas*, destinam-se a dar informação para ajudar o optimizador de código.

O Aquarius implementa várias optimizações na BAM como passo final, eliminação de código morto, eliminação de saltos, eliminação de código duplicado, redução de atribuições, entre outras.

Um conceito importante usado na BAM é o uso de variáveis por inicializar, que permite que se poupe trabalho a inicializar variáveis que se sabe que vão ser unificadas.

2.4.2 Parma

O Parma foi desenvolvido por A. Taylor e é contemporâneo do Aquarius. Muitas das suas ideias base são similares às usadas pelo Aquarius.

A compilação no Parma é processada em vários passos. Os mais relevantes são:

- O sistema começa por transformar o código prolog, simplificando a unificação na cabeça das cláusulas.
- Em seguida o Parma faz uma análise global do código de forma a conseguir obter informação sobre os argumentos dos predicados.
- O programa é transformado para um código intermédio com instruções de 3 endereços, que inclui instruções `load` e `store`, operações aritméticas e lógicas, testes condicionais e saltos.

- O código intermédio é traduzido para assembly do MIPS, usando técnicas standard de compilação.

Tanto o Parma como o Aquarius evitam a WAM, e devem a sua boa performance principalmente ao passo de análise global.

2.4.3 SICStus Native

O SICStus native é um exemplo de um sistema que em vez de gerar código nativo directamente do Prolog, usa a WAM como passo intermédio. Este sistema traduz o código WAM para uma nova máquina abstracta, a SAM.

A SAM é muito simples, e inclui poucas instruções: `ld`, `st`, `stz` para atribuições, operações aritméticas e lógicas, e instruções de controle. As operações complexas como a unificação, são implementadas como um salto para código pré-programado em código assembly (Kernel). A SAM utiliza o algoritmo 2-Streams para simplificar a unificação de termos compostos. A SAM é traduzida para a RISS, uma representação intermédia para máquinas *RISC*, ou directamente para código do 68k.

A implementação do código nativo no sistema YAP seguirá uma aproximação do tipo da do SICStus native. No próximo capítulo apresentar-se-ão as novas linguagens intermédias criadas para o sistema YAP de forma a se conseguir uma geração de código nativo eficiente.

Capítulo 3

Novas linguagens intermédias para a compilação de Prolog

A WAM foi um grande passo para a execução eficiente de Prolog. Contudo as suas instruções são muito complexas não permitindo que um grande número de optimizações se realizem. Por exemplo, muitas das instruções da WAM fazem uma desreferenciação, mesmo que fosse possível determinar que tal operação não era necessária. A YAAM, descrita na secção 2.3.4, inclui alguns melhoramentos sobre a WAM, mas mantém as mesmas características. Neste capítulo apresentar-se-ão duas novas linguagens intermédias, a YAAM 2-Streams, que na verdade é uma adaptação da YAAM como o próprio nome o indica, e a YAIL. Ambas foram criadas para obter uma geração de código nativo eficiente no YAP. As novas linguagens intermédias não pretendem substituir a YAAM na compilação de Prolog mas sim complementá-la como novos passos intermédios na geração do código final. Estas novas linguagens intermédias usam áreas de trabalho idênticas à da YAAM/WAM.

3.1 A YAAM 2-Streams

A linguagem YAAM 2-Streams, como o próprio nome indica, deriva de aplicar o algoritmo 2-Streams (ver secção 2.2) à linguagem intermédia do YAP, a YAAM. Relembra-se ainda que a maior vantagem de ter o código WAM dividido em duas áreas, uma de leitura e outra de escrita, é a propagação do modo de escrita dos termos aos subtermos. Contudo a YAAM já faz essa propagação, logo a implementação do algoritmo em questão não tem qualquer ganho nesse aspecto. As vantagens de alterar a YAAM para funcionar com duas áreas de código, têm a ver com a diminuição da complexidade das instruções e com a redução de saltos (`jumps`). Na verdade, usando este novo algoritmo, deixa de ser necessário no início das instruções `unify` fazer um teste para decidir se estamos em modo de escrita ou em modo de leitura. Passam a existir apenas alguns testes que controlam o momento em que se deve saltar do código de leitura para o código de escrita ou vice-versa. Esta diminuição de saltos internos é importante porque nas máquinas *RISC*, no presente, um salto é uma operação com uma penalização muito elevada em termos de execução. Saliente-se que a YAAM 2-Streams foi criada com o objectivo de ser usada numa implementação com geração de código nativo, pois num emulador é necessário efectuar um salto no final de cada instrução de qualquer das formas. O custo associado ao uso de duas áreas de código é o aumento do tamanho do código intermédio a ser tratado para aproximadamente o dobro, mas observe-se que as instruções passam a ter metade ou menos da complexidade.

3.1.1 Registos da YAAM 2-Streams

Os registos da YAAM 2-Streams são idênticos aos da YAAM, excepto no registo `Reading`. A YAAM usava este registo para controlar o modo de execução. Nesta nova implementação esse registo já não é necessário. Passa porém a ser necessário um novo registo somente no código de escrita para determinar se já se fez a unificação dos n argumentos do termo complexo e se se deve regressar ao modo de leitura. Este registo foi designado por `M` (de modo).

3.1.2 Descrição das instruções da YAAM 2-Streams

A YAAM 2-Streams difere da YAAM apenas nas instruções que lidam com os termos compostos, `get_XXX` e `unify_XXX`, fazendo uso do registo `Reading` para indicar se está a ler um termo ou a escrever um novo termo. As novas instruções da YAAM 2-Streams podem ser classificadas em três classes distintas. Elas são:

1. Instruções do tipo `get_XXX_read` e `get_XXX_write` que resultam da divisão das instruções `get_XXX` em duas.

2. Instruções do tipo `unify_XXX_read` e `unify_XXX_write` que resultam da divisão das instruções `unify_XXX` em duas.

3. Instruções de controle para decidir o momento em que a execução deverá passar do modo de leitura para o modo de escrita ou vice-versa.

   - `deref_read Xi,labelX`. Esta instrução que aparece somente no código de leitura, faz a desreferenciação de `Xi` e caso se verifique ser variável salta para o `labelX` que fica no código de escrita.
   - `deref_write Xi,labelY`. Esta instrução que aparece somente no código de escrita, faz a desreferenciação de `Xi` e caso se verifique não estar instanciada salta para o `labelY` que fica no código de leitura.
   - `deref_S i,labelW`. Esta instrução que aparece somente no código de leitura, faz a desreferenciação do registo `S` e caso se verifique ser variável faz `M=i` e salta para o `labelW` que fica no código de escrita.
   - `if_M_ge i,labelZ`. Esta instrução que aparece somente no código de escrita, salta para o `labelZ` no código de leitura caso se verifique que o valor no registo `M` é maior ou igual ao valor de `i`.

3.1.3 Transformação do código YAAM em YAAM 2-Streams

O código YAAM gerado a partir do Prolog é usado para criar dois códigos, um de leitura e outro de escrita. Apenas as instruções `get_XXX` e `unify_XXX` são alteradas (tabela 3.1).

A figura 3.1 apresenta-se uma parte representativa do algoritmo de transformação do código YAAM para o código YAAM 2-Streams. Foram escolhidas 4 instruções, cada uma delas pertencendo a uma das 4 classes distintas de instruções apresentadas na tabela 3.1. Observe-se que o algoritmo começa por chamar a função `two_streams()` que irá invocar por sua vez a função `transforma_instrução()`. Esta última função é que faz a divisão das instruções. O nível da estrutura é controlado pelo parâmetro desta função (`m`), e é incrementado aquando do aparecimento de um `unify` de uma estrutura ou lista, altura essa em que é feita uma chamada recursiva à função.

Características principais desta implementação:

- Todo o código é duplicado, mesmo que seja igual nos dois modos. A parte de optimização posterior tratará de eliminar o código desnecessário (ver capítulo 4).

- O código gerado deverá começar por ser executado em modo de leitura e pode mudar de modo conforme ocorram as seguintes situações:

Instruções YAAM	**Código de leitura**	**Código de escrita**

`get_atom` ou
`get_num` ou
`get_value`
	`deref_read labelY`
	`labelX:`
	`get_XXX_read`
		`deref_write labelX`
		`labelY:`
		`get_XXX_write`

`get_struct`
ou
`get_list`
	`deref_read labelY`
	`labelX:`
	`get_XXX_read`
		`deref_write labelX`
		`labelY:`
		`get_XXX_write`

`unify_struct`
ou
`unify_list`
	`deref_S n,labelY`
	`unify_XXX_read`
	...
	unify's
	...
	`labelZ:`
		`labelY:`
		`unify_XXX_write`
		...
		unify's
		...
		`if_M_ge n,labelZ`

`unify_atom` ou `unify_value` ou `unify_var` ou `unify_val` ou `unify_local`	`unify_XXXX_read`	`unify_XXXX_write`

Tabela 3.1: Transformações na YAAM para obter código de leitura e escrita separado

- em modo de leitura, na presença de uma instrução `get_struct` ou `get_list`, o argumento da instrução deverá ser desreferenciado usando a instrução `deref` e caso se verifique ser variável passa-se para o modo de escrita.
- em modo de escrita, na presença de uma instrução `get_struct` ou `get_list`, o argumento da instrução deverá ser desreferenciado e caso se verifique não ser variável passa-se para o modo de leitura.
- em modo de leitura, na presença de uma instrução `unify_struct` ou `unify_list`, o registo `S` deverá ser desreferenciado e caso se verifique ser variável passa-se para o modo de escrita. Retorna-se para o modo de leitura no final das instruções `unify` requeridos pelo termo complexo.
- em modo de escrita, na presença de um `unify_XXX` não se faz qualquer desreferenciação e continua-se em modo de escrita.
- em modo de escrita, na presença de uma instrução `if_M_ge i,labelZ` se o valor do registo `M` for superior ou igual a `i` a execução passa para o código de leitura, nomeadamente para o `labelZ`.

• O algoritmo é linear e requer uma só passagem pelo código YAAM trazendo deste modo pouco acréscimo ao tempo de compilação.

```
two_streams() {
(...)
        enquanto_houver_instruções faça
        transforma_instrução(0);
(...)
}

void transforma_instrução(inteiro m) {

switch(instrução_actual) {

    case (get_atom (Xi,at)) :
        emitir_inst_leitura  deref_read (Xi,++label)
        emitir_inst_leitura  ++label:
        emitir_inst_leitura  get_atom_read(Xi,a)
        emitir_inst_escrita  deref_write(Xi,label)
        emitir_inst_escrita  label-1:
        emitir_inst_escrita  get_atom_write (Xi,a)

    case (get_struct (Xi,st)) :
        emitir_inst_leitura  deref_read (Xi,++label)
        emitir_inst_leitura  ++label:
        emitir_inst_leitura  get_struct_read (Xi,st)
        emitir_inst_escrita  deref_write (Xi,label)
        emitir_inst_escrita  label-1:
        emitir_inst_escrita  get_struct_write (Xi,st)

        for i=0 to arity_da_struct
            transforma_instrucao(m)

    case (unify_atom (at)):
        emitir_inst_leitura  unify_atom_read(at)
        emitir_inst_escrita  unify_atom_write(at)

    case (unify_struct (st)):
        emitir_inst_leitura  derefS (m+1,++label)
        emitir_inst_leitura  unify_struct_read (st)
        emitir_inst_escrita  label:
        emitir_inst_escrita  unify_struct_write (st)

        for i=0 to arity_da_struct
            transforma_instrucao(m+1)

        emitir_inst_leitura  ++label:
        emitir_inst_escrita  if_M_ge (m,label)

    (...)
}
```

Figura 3.1: Algoritmo para transformar o código YAAM em YAAM 2-Streams.

3.1.4 Exemplo 1

Seguem-se agora alguns exemplos sobre a transformação do código YAAM em YAAM 2-Streams. Observe-se que os exemplos não pretendem ter alguma utilidade prática, servindo apenas para ilustrar a geração do novo código. Em cada exemplo será feita uma breve descrição para comparar os dois códigos.

Neste primeiro exemplo, temos uma cláusula muito simples `estudante(menino(tiago))`. O código YAAM é apresentado na figura 3.2. O código YAAM 2-Streams é apresentado na figura seguinte, 3.3.

```
name              estudante/1,1
get_struct        menino/1,A1     (1)
unify_atom        tiago           (2)
proceed
```

Figura 3.2: Código YAAM para `estudante(menino(tiago))`.

```
           name              estudante/1,1
           deref_read        A1,label1       (3)
label2:    get_struct_read   menino/1,A1     (4)
           unify_atom_read   tiago           (5)
           proceed

write_code                                   (6)
           name              estudante/1,1 (*)
           deref_write       A1,label2       (*)
label1:    get_struct_write  menino/1,A1     (7)
           unify_atom_write  tiago           (8)
           proceed
```

Figura 3.3: Código YAAM 2-Streams para `estudante(menino(tiago))`.

No código normal, YAAM, a instrução `get_struct (1)` faz a desreferenciação de `A1` e actualiza o registo `Reading` conforme seja variável ou não. A instrução `unify_atom (2)` faz um teste ao valor do registo `Reading` para decidir qual o código a executar.

A instrução `deref_read (3)` faz a desreferenciação e caso `A1` seja variável salta para o `label1` continuando a execução em modo de escrita. Caso não seja variável, continua a execução em modo de leitura executando a instrução seguinte, `get_struct_read (4)`.

As instruções assinaladas com (*) nunca chegam a ser executadas, visto o código começar sempre a ser executado em modo de leitura, e são eliminadas durante a fase optimização do código. Nos exemplos seguintes estas instruções já não serão apresentadas.

3.1.5 Exemplo 2

Neste segundo exemplo, temos uma cláusula muito conhecida dos programadores de Prolog, `append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).` Na figura 3.4 apresenta-se o código YAAM, na figura 3.5 o código YAAM 2-Streams.

```
name            append/3,3
get_list        A1              (9)
unify_var       X4              (10)
unify_var       X1              (11)
get_list        A3              (12)
unify_val       X4              (13)
unify_var       X3              (14)
execute         user:append/3
```

Figura 3.4: Código YAAM para `append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).`

Neste exemplo, a instrução `get_list (9)` foi dividida nas duas instruções `deref_read` e `get_list_read ((15)+(16))` para o modo de leitura e `get_list_write (23)` para o modo de escrita. A nova instrução `deref_read (15)` faz a desreferenciação de `A1` e caso seja variável salta para `label1` entrando em modo de escrita. Caso contrário, continua a execução com `get_list_read (16)` permanecendo em modo de leitura.

O segundo `get_list (12)` foi dividido nas instruções `deref_read` e `get_list_read ((19)+(20))` para o modo de leitura e `deref_write (26)` e `get_list_write (27)` para o modo de escrita. A instrução `deref_read (19)` faz a desreferenciação de `A3` e caso seja variável salta para `label2` entrando em modo de escrita. Caso contrário, permanece em modo de leitura e executa `get_list_read (20)`. A instrução `deref_write (26)` funciona

```
           name               append/3,3
           deref_read         A1,label1     (15)
           get_list_read      A1            (16)
           unify_var_read     X4            (17)
           unify_var_read     X1            (18)
           deref_read         A3,label2     (19)
label3:    get_list_read      A3            (20)
           unify_val_read     X4            (21)
           unify_var_read     X3            (22)
           execute            user:append/3
```

write_code

```
label1:    get_list_write     A1            (23)
           unify_var_write    X4            (24)
           unify_var_write    X1            (25)
           deref_write        A3,label3     (26)
label2:    get_list_write     A3            (27)
           unify_val_write    X4            (28)
           unify_var_write    X3            (29)
           execute            user:append/3
```

Figura 3.5: Código YAAM 2-streams da cláusula `append`.

de modo inverso a `deref_read (19)`. Faz a desreferenciação de `A3` e salta para `label3` caso não seja variável.

3.1.6 Exemplo 3

Neste exemplo, uma cláusula um pouco mais elaborada é apresentada para que se entenda como e quando são realizados os saltos entre o código de leitura e escrita. Na figura 3.6 apresenta-se o código YAAM para `a(b(c(d([x,y]),e(w))))`. Na figura seguinte, 3.7, apresenta-se o código YAAM 2-Streams para a mesma cláusula.

Neste exemplo, a instrução `get_struct (30)` foi transformada nas instruções `deref_read` e `get_struct_read ((41)+(42))` para o modo de leitura e `get_struct_write (58)` para o modo de escrita. A instrução `deref_read (41)` faz a desreferenciação de `A1` e caso seja variável salta para `label1` entrando no modo de escrita onde continuará até ao final do

```
name                  a/1,1
get_struct            b/1,A1       (30)
unify_last_struct     c/2          (31)
unify_struct          d/1          (32)
unify_last_list                    (33)
unify_atom            x            (34)
unify_last_list                    (35)
unify_atom            y            (36)
unify_atom            []           (37)
pop                   1            (38)
unify_last_struct     e/1          (39)
unify_atom            w            (40)
proceed
```

Figura 3.6: Código YAAM para `a(b(c(d([x,y]),e(w))))`.

código. Caso não seja variável continua em modo de leitura executando `get_struct_read` `(42)`.

A instrução `unify_last_struct (31)` foi transformada em duas instruções, `derefS` e `unify_last_struct_read ((43)+(44))`, para o código de leitura e, para o código de escrita, na instrução `unify_last_struct_write (59)`. A instrução `derefS (43)` faz a desreferenciação do registo `S` e caso seja variável actualiza o registo `M` para 1 e salta para o `label2` entrando para o modo de escrita. Neste caso o registo `M` só é usado pela instrução `unify_last_struct_write (59)` para saber se acabou de entrar no modo de escrita ou se já lá estava.

As instruções `unify_XXX (32)`,`(33)`, `(35)` foram transformadas nas instruções `derefS` e `unify_XXX_read ((45)+(46))`,`((47)+(48))`,`((50)+(51))` para o código de leitura e `unify_XXX_write (60)`, `(61)`, `(63)` para o código de escrita. As instruções `derefS` fazem a desreferenciação do registo `S` e caso seja variável actualizam o registo `M`. Este registo será usado pela instrução `if_t_ge (67)` para saber se deve retornar para o código de leitura. Saliente-se que que a instrução `if_M_ge 2,label6` é a junção das três instruções:

```
if_M_ge 4,label6;
if_M_ge 3,label6;
if_M_ge 2,label6;
```

```
          name                       a/1,1
          deref_read                 A1,label1      (41)
          get_struct_read            b/1,A1         (42)
          derefS                     M=1,label2     (43)
          unify_last_struct_read     c/2            (44)
          derefS                     M=2,label3     (45)
          unify_last_struct_read     d/1            (46)
          derefS                     M=3,label4     (47)
          unify_last_list_read                      (48)
          unify_atom_read            x              (49)
          derefS                     M=4,label5     (50)
          unify_last_list_read                      (51)
          unify_atom_read            y              (52)
          unify_atom_read            []             (53)
          pop                        1              (54)
label6:   derefS                     M=2,label7     (55)
          unify_last_struct_read     e/1            (56)
          unify_atom_read            w              (57)
          proceed
```

write_code

```
label1:   get_struct_write           b/1,A1         (58)
label2:   unify_last_struct_write    c/2,M=1        (59)
label3:   unify_struct_write         d/1,M=2        (60)
label4:   unify_last_list_write      M=3            (61)
          unify_atom_write           x              (62)
label5:   unify_last_list_write      M=4            (63)
          unify_atom_write           y              (64)
          unify_atom_write           []             (65)
          pop                        1              (66)
          if_M_ge                    2,label6       (67)
label7:   unify_last_struct_write    e/2,M=2        (68)
          unify_atom_write           w              (69)
          proceed
```

Figura 3.7: Código YAAM 2-Streams para `a(b(c(d([x,y]),e(w))))`.

numa só, significando que a execução deverá continuar em modo de leitura, se o valor do registo `M` for maior ou igual a `2`.

Por último, observe-se que a instrução `derefS` só é necessária no código de leitura. Se, por exemplo, for executado `a(X)`, a sequência do código a executar será `( 41 ; 58` até `69)`, não sendo necessário fazer nenhuma desreferenciação do registo `S`.

Um outro exemplo da transição de um programa Prolog para a linguagem YAAM 2-Streams é dado no apêndice B.

3.1.7 Uso da YAAM 2-Streams

A linguagem intermédia YAAM 2-Streams foi criada com o objectivo de ser usada na compilação de Prolog com geração de código nativo. Contudo pode ser usada numa implementação com emulador de código tendo a vantagem de ter instruções mais simples do que a YAAM/WAM e a desvantagem do código ter aproximadamente o dobro do tamanho, ocupando por isso mais memória.

O sistema SICStus Prolog aplica o algoritmo 2-Streams de uma forma ligeiramente diferente à usada na YAAM 2-Streams:

- A SAM não precisa de um novo registo. Na YAAM 2-Streams foi necessário criar um novo registo `M`, para ser usado no código de escrita por forma a controlar quando se deve retornar ao código de leitura. Na SAM o registo `S` é usado para esse efeito, pois nesta implementação aquele registo está livre durante o código de escrita. O registo `S` é actualizado com o valor que o registo `H` deverá ter quando terminar a criação do subtermo. Desta forma a condição de retorno é a simples comparação do registo `S` com o registo `H`. Se forem iguais a execução passa para o código de leitura.

- A YAAM 2-Streams gera menos saltos do que a SAM. Na SAM no final da construção de um termo em código de escrita a execução retorna sempre para o modo de leitura.

  Por exemplo, considere-se a cláusula `pai(menino(manuel),menina(maria))`. Considere-se também que é invocado o predicado `pai` com duas variáveis `pai(A,B)`.

  Na SAM a execução vai ter a seguinte ordem: como o primeiro argumento é variável a execução salta para o código de escrita fazendo-se a unificação de `A` com `menino(manuel)`. Em seguida, a execução regressa para o modo de leitura. Aí, como o segundo argumento é novamente uma variável, a execução salta novamente para o código de escrita fazendo-se a unificação de `B` com `menina(maria)`.

Na YAAM 2-Streams como o primeiro argumento é variável a execução salta para o código de escrita, onde faz a unificação de `A` com `menino(manuel)`. Em seguida como o segundo argumento é uma variável a execução não retorna ao modo de leitura. Continua em modo de escrita para fazer a unificação de `B` com `menina(maria)`. Neste caso a YAAM 2-Streams poupa um salto em relação a SAM. Na YAAM 2-Streams a execução só retornaria ao modo de leitura caso o segundo argumento já estivesse instanciado.

3.2 YAIL - Yet Another Intermediate Language

A criação de uma nova linguagem intermédia está normalmente ligada a dois objectivos. O primeiro é o de isolar muitas das dependências do compilador em relação à máquina a que se destina. O segundo é o de criar uma representação capaz de melhorar as transformações e optimizações que são difíceis (mesmo impossíveis) na linguagem de origem e destino. O segundo factor prevaleceu sobre o primeiro na criação da linguagem intermédia aqui apresentada.

As instruções foram escolhidas de forma a serem o mais independentes possível umas das outras, para se poder na maior parte dos casos alterar a sua ordem, por forma a permitir um maior leque de possíveis optimizações. Teve-se também em conta as instruções da arquitectura Alpha, de forma a que a transformação para esta arquitectura seja feita com o menor custo possível, aproveitando ao máximo os seus registos e as suas instruções. Apesar disso, tentou-se não inviabilizar uma possível implementação numa outra máquina. Procurou-se igualmente minimizar o trabalho realizado pelo compilador de traduzir o código YAAM para YAIL.

No conjunto de instruções da linguagem intermédia YAAM existe uma série de optimizações que não são possíveis de se realizar. Considere-se por exemplo a estrutura `irmão(manuel,tiago).` e o resultado da sua compilação para YAAM (figura 3.8). O conjunto de instruções YAAM fazem com que seja difícil obter código mais optimizado. Sabemos porém que a inicialização do `Heap` feita pela instrução `get_struct` quando entra em modo de escrita é desnecessária, pois as instruções `unify` que se lhe seguem realizam essa operação. Esta optimização já pode ser facilmente realizada a nível da YAIL.

```
get_struct  irmão
      unify_atom  manuel
      unify_atom  tiago
```

Figura 3.8: Código YAAM para a estrutura `irmão(manuel,tiago)`.

A YAIL é constituída por instruções muito simples (ver 3.2.1). Todas as instruções são facilmente traduzidas para código nativo da arquitectura Alpha (na verdade é facilmente traduzida para o assembler de qualquer máquina *RISC*). Esta nova linguagem tem a vantagem de simplificar e tornar mais eficiente a geração código nativo para máquinas convencionais. A simplificação advém de o conjunto de instruções usado na YAIL ter transformação quase imediata para código nativo. A eficiência resulta de se permitir ter um maior controlo sobre o uso dos registos, o que na prática se traduz numa melhoria substancial no tempo de execução dos programas.

3.2.1 Descrição das instruções da YAIL

Na YAIL somente duas instruções têm acesso à memória, `load` e `store`. Todas as outras funcionam apenas com registos. As instruções podem ser classificadas em quatro classes distintas. Elas são:

- **Instruções que acedem a posições de memória** (tabela 3.2). São apenas duas, e são directamente traduzidas para uma instrução da arquitectura Alpha.

Instrução	Significado
`Load(Ra,Rb,disp)`	Ra = *(Rb+disp)
`Store(Ra,Rb,disp)`	*(Rb+disp)=Ra

Tabela 3.2: Instruções que acedem a posições de memória.

- **Instruções simples que trabalham apenas com registos** (tabela 3.3). São instruções de âmbito geral que apenas fazem operações entre registos, não tendo acesso a posições de memória. Todas estas instruções são traduzidas em uma única instrução da Alpha.

  As instruções `test`, `compare`, `comparei`, `move_cond` e `move_condi` precisam de um parâmetro extra que indique qual é a condição a usar. Na tabela 3.4 estão descritas quais as condições possíveis para estas instruções.

Instrução	**Significado**
`test(cond,Ra, label)`	if cond(Ra) jump label
`compare(cond,Ra,Rb,Rc)`	Rc= (Ra cond Rb)
`comparei(cond,Ra n,Rc)`	Rc= (Ra cond n)
`move_cond(cond,Ra,Rb,Rc)`	if cond(Ra) Rc=Rb
`move_condi(cond,Ra,n,Rc)`	if cond(Ra) Rc=n
`set(Ra,Rb,n)`	Ra=Rb+n (n com 16 bits)
`add(Ra,Rb,Rc)`	Ra=Rb+Rc
`addv(Ra,Rb,n)`	Ra=Rb+n
`mul(Ra,Rb,Rc)`	Ra=Rb*Rc
`mulv(Ra,Rb,n)`	Ra=Rb*n
`shift(Ra,Rb,n)`	Ra=Rb$\gg$ n

Tabela 3.3: Instruções simples que trabalham apenas com registos.

Condições para test e move_cond	**Condições para o compare**
`Equal to Zero`	`Equal`
`Greater or Equal than Zero`	`Less or Equal`
`Greater Than Zero`	`Less Than`
`Low Bit Clear`	`Unsigned Less or Equal`
`Low Bit Set`	`Unsigned Less Than`
`Less Than Zero`	
`Less or Equal to Zero`	
`Not Equal to Zero`	

Tabela 3.4: Condições existentes para as instruções de comparação

- **Instruções complexas** (tabela 3.5). Apesar de todas estas instruções poderem ser escritas sobre a forma de um conjunto de instruções simples, assim não foi feito porque iria tornar a sua detecção mais difícil. Por exemplo, num caso em que uma variável é desreferenciada duas vezes, a segunda desreferenciação pode ser removida. Este caso seria muito difícil de detectar se as instruções fossem expandidas. Para além disso, as instruções como o `unify` são bastante complexas, e a sua expansão levaria a que o código crescesse muito. Ao não expandir este código, torna-se possível usar este tipo de instruções como sendo uma subrotina. Saliente-se que apesar destas instruções operarem sobre registos, todas as instruções, excepto as instruções `failX`, podem durante a sua execução aceder a posições de memória.

- **Instruções relacionadas com o esquema de etiquetas** (tabela 3.6). Novamente, todas estas instruções poderiam ser escritas como um conjunto de instruções simples. O modo como as etiquetas são implementadas influência substancialmente o tempo

Instrução	**Significado**
`deref(Ra,Rb)`	Ra fica com o resultado da desreferenciação de Rb
`failz(Ra)`	Se Ra for zero falha
`failnz(Ra)`	Se Ra for diferente de zero falha
`trail(Ra)`	faz o trail de Ra se Ra for menor que `BH` e maior que `B`
`trailglobal(Ra)`	faz o trail de Ra se Ra for menor do que `BH`
`traillocal(Ra)`	faz o trail de Ra se Ra for maior do que `B`
`unify(Ra,Rb)`	unifica as células Ra e Rb
`unifybound(Ra,Rb)`	tenta unificar Ra com Rb e retorna 1 se conseguir
`push(Ra)`	push(Ra)
`pop(Ra)`	pop(Ra)

Tabela 3.5: Instruções complexas

de execução de um programa Prolog. Assim sendo é natural que a implementação do esquema de etiquetas varie de máquina para máquina. Ao criar um conjunto de instruções relacionadas com a implementação das etiquetas melhorou-se a portabilidade deste sistema para outra máquina.

Instrução	**Significado**
`IsVar(Ra,Rb)`	Retorna 0 em Ra se Rb não for uma variável
`IsAppl(Ra,Rb)`	Retorna 0 em Ra se Rb não for uma estrutura
`IsPair(Ra,Rb)`	Retorna 0 em Ra se Rb não for uma lista
`RepAppl(Ra,Rb)`	Ra toma o valor de Rb sem a etiqueta de estrutura
`RepPair(Ra,Rb)`	Ra toma o valor de Rb sem a etiqueta de lista
`AbsAppl(Ra,Rb)`	Ra toma o valor de Rb mais a etiqueta de estrutura
`AbsPair(Ra,Rb)`	Ra toma o valor de Rb mais a etiqueta de lista

Tabela 3.6: Instruções relacionadas com a implementação de etiquetas.

3.2.2 Organização dos registos na YAIL

A YAIL tem por definição um número ilimitado de registos, contudo na última fase de compilação, depois da optimização, os registos serão restringidos aos existentes na máquina destino. Os registos na YAIL têm a seguinte organização;

- `R0`, registo com o valor sempre em zero.
- `R1` - `R9`, registos de âmbito geral, para controle do estado da execução (ver tabela 3.7). Estes registos por simplicidade em exemplos posteriores serão designados como

R_ mais o nome usado na YAAM, por exemplo, R_S representará o registo da YAIL R1.

- R10 - no código de escrita, este registo é equivalente ao registo M da YAAM 2-Streams, serve para controlar o momento em que se deve saltar para o código de leitura. No código de leitura é usado como registo temporário para outras operações. Este registo, por simplicidade será designado por R_M.
- R11,R12 - registos temporários, são na maior parte das vezes usados para receber o resultado de operações de desreferenciação. Estes dois registos serão designados por R_D1 e R_D2 respectivamente.
- R13 - registo temporário usado em outras operações, principalmente comparações. Este registo será designado por R_T.
- R14 em diante, registos X[i]. Estes registos serão designados por R_X+i.

YAIL	YAAM
R1	S
R2	H
R3	Y
R4	ENV
R5	B

YAIL	YAAM
R6	HB
R7	TR
R8	CP
R9	P

Tabela 3.7: Equivalência entre os registos da YAIL e da YAAM

Nesta tese os registos serão sempre referidos pela sua designação com letras, ou seja, mantendo uma notação parecida com a da YAAM/WAM, para simplificar a leitura.

3.2.3 Esquema de etiquetas escolhido para usar na arquitectura Alpha

Uma boa implementação de etiquetas pode trazer uma melhoria substancial ao tempo de execução de um programa Prolog.

O sistema de etiquetas aqui apresentado foi criado tendo em conta as características da linguagem intermédia YAIL, e as características da arquitectura Alpha. A arquitectura Alpha é uma máquina de 64 bits, que permite aceder a qualquer endereço múltiplo de 4 para ler uma palavra de 32 ou 64 bits. Contudo existe uma grande penalização quando o acesso a uma palavra de 64 bits é feito num endereço não múltiplo de 8. Como a implementação do YAP na Alpha usa palavras de 64 bits, deve usar-se apenas endereços múltiplos de 8, ficando assim os 3 bits inferiores livres para representar as etiquetas.

O sistema de etiquetas escolhido para a Alpha é o seguinte:

- `000` - Variável
- `101` - Lista
- `011` - Estrutura
- `001` - Inteiro, Virgula Flutuante ou Átomo

Esta escolha teve em conta os dois pontos seguintes:

- a instrução IsVar é usada muitas vezes. Assim sendo, é importante que este teste seja o mais simples possível. Como existem os teste `TLBS` (Test Low Bit Set) e `TLBC` (Test Low Bit Clear) é natural que se use o bit mais a direita para distinguir as variáveis, de modo que a instrução `IsVar` corresponda a apenas uma instrução da Alpha. A escolha 000 é natural pois permite que o acesso a endereços possa ser feito por um múltiplo de 8.

- é importante que o teste, retirar ou criar de uma etiqueta de lista ou estrutura seja o mais simples possível. Assim reservou-se o terceiro bit para representar uma lista e o segundo para representar uma estrutura. Deste modo para um elemento que não seja variável, basta fazer um `and` com 4 ou 2 e ver se o resultado é diferente de zero para se concluir se é lista ou estrutura respectivamente. Com esta representação torna-se também muito simples criar ou retirar a etiqueta, pois basta para tal somar ou subtrair o valor da etiqueta.

Uma característica interessante que advém do uso destas etiquetas é que em certos casos não é preciso retirar a máscara para aceder a um endereço. Por exemplo, suponha-se que no registo `Rx` está o endereço de uma estrutura. Como a YAIL acede a uma posição de memória mais um deslocamento de 16 bits, para aceder à estrutura na posição de memória indicada por `Rx`, usa-se a instrução `Load(Ry,(Rx),-3)`. Um apontador para uma estrutura tem a máscara 011 nos três bits mais à direita, o que representa 3. Ao acedermos ao endereço `Rx-3` estamos a retirar a etiqueta de `Rx` sem para isso ser necessário efectuar outra operação. Saliente-se que este tipo de optimização é fácil de implementar na YAIL, mas não na WAM ou YAAM.

3.2.4 Um exemplo

Para se ilustrar o modo como se transforma o código YAAM para YAIL, apresenta-se como exemplo a transformação da instrução da YAAM, `put_unsafe`, para a YAIL. Na figura 3.9 é apresentado o código da instrução `put_unsafe` em *Pseudo-C-Português*.

```
put_unsafe( i : inteiro, j : inteiro )
{
temporario Z;
        Z= desreferencia (Y[i]);
        se (for_variavel(Z))  e Z > H e  Z < Y ) {
               bind_local(Z,H);
                X[j]= H;
                *H= H++;
        }
        senão { X[j]= Z; }
}
```

Figura 3.9: Código *Pseudo-C* da instrução `put_unsafe`.

O código YAIL para a instrução `put_unsafe` pode ser gerado de duas formas diferentes. Na primeira forma, figura 3.10, o código está optimizado máquinas que dispõem de uma instrução que faça uma atribuição condicional (como é o caso da arquitectura Alpha).

Nos exemplos que se seguem `CELLSIZE` representa o tamanho da palavra da máquina destino. Por questões de legibilidade, os registos estão representados com letras em vez de números.

Na segunda forma, figura 3.11, apresenta-se o código YAIL optimizado para máquinas que não dispõem de uma instrução que faça uma atribuição condicional.

Os dois códigos diferem no modo como se faz a comparação do resultado da desreferenciação com o registo `H` e `Y`. Na segunda forma é feito um teste para cada uma das comparações. Na primeira forma as duas comparações são realizadas uma a seguir à outra, e o resultado é combinado usando a instrução `movei_cond`. Desta forma passa a ser necessário apenas um salto condicional.

No apêndice B apresenta-se um exemplo completo do código YAIL obtido por compilação

```
load                          R_T, R_Y, i*CELLSIZE
deref                         R_X+j, R_T
isvar                         R_T, R_X+j
test  equal to zero           R_T, labelX
compare unsigned less than    R_H, R_X+j, R_T
compare unsigned less than    R_X+j, R_Y, R_D1
movei_cond equal to zero      R_D1, 0, R_T
test equal to zero            R_T, labelX
store                         R_H, R_X+j, 0
traillocal                    R_X+j
set                           R_X+j, R_H, 0
store                         R_H, R_H, 0
set                           R_H, R_H, CELLSIZE
labelX:
```

Figura 3.10: Código YAIL para a instrução `put_unsafe(Y[i],X[j])`, forma 1.

```
load                          R_T, R_Y, i*CELLSIZE
deref                         R_X+j, R_T
isvar                         R_T, R_X+j
test  equal to zero           R_T, labelX
compare unsigned less than    R_H, R_X+j, R_T
test equal to zero            R_T, labelX
compare unsigned less than    R_X+j, R_Y, R_T
test equal to zero            R_T, labelX
store                         R_H, R_X+j, 0
traillocal                    R_X+j
set                           R_X+j, R_H, 0
store                         R_H, R_H, 0
set                           R_H, R_H, CELLSIZE
labelX:
```

Figura 3.11: Código YAIL para a instrução `put_unsafe(Y[i],X[j])`, forma 2.

de um programa Prolog.

3.2.5 Comparação da YAIL com outras linguagens intermédias

Nesta secção far-se-á uma comparação entre a YAIL e outras linguagens intermédias para geração de código nativo, como a SAM, BAM, e a linguagem intermédia do Parma. A YAIL tem uma estrutura mais parecida com a BAM do sistema Aquarius, incluindo também instruções simples e complexas, embora o esquema geral de implementação do Native YAP seja mais parecido com o usado para o SICStus Nativo.

As diferenças fundamentais entre a YAIL e as outras linguagens intermédias são:

- A YAIL, ao contrário das outras linguagens intermédias, dispõe de um conjunto de instruções especificas para a implementação de etiquetas. Este facto torna o sistema mais independente da máquina destino, tornando-o mais portável.

- A YAIL dispõe de um conjunto de instruções simples, e complexas. O mesmo acontece com a SAM e a BAM. Na linguagem intermédia do Parma não existem instruções complexas.

  Na BAM as instruções complexas são traduzidas directamente para instruções simples. Na SAM as instruções complexas são implementadas no "Kernel" do sistema. Na YAIL existe uma maior flexibilidade, as instruções complexas podem ser expandidas ou podem ser implementadas como funções, a decisão é feita num passo posterior.

- A YAIL não dispõe de instruções de salto condicional baseados na comparação de dois registos como acontece na BAM e no Parma. Na YAIL para fazer um salto baseado no resultado da comparação de dois registos, são necessárias duas instruções. Uma primeira (`compare`) faz a comparação dos registos atribuindo o resultado a um terceiro registo (`1` verdadeiro, `0` falso), a segunda instrução (`test`) faz o teste ao valor do terceiro registo para decidir se deve saltar.

- A YAIL não dispõe de "pragmas", como acontece na BAM, pois para tal seria necessário o YAP realizar análise global ao programa Prolog de forma a passar informação ao código intermédio.

- Por último, ao contrário das outras linguagens intermédias, a YAIL suporta instruções disponíveis em arquitecturas modernas, que podem ser emuladas em arquitecturas anteriores, como seja a atribuição condicional.

O objectivo inerente à criação da YAIL foi a implementação de código nativo no YAP. No próximo capítulo apresentar-se-á o sistema YAP adaptado à geração de código nativo.

Capítulo 4

O Sistema Native YAP

O YAP compila programas Prolog para uma máquina abstracta, a YAAM (ver 2.3.4), e emula este código através de um emulador rápido escrito na linguagem C. A desvantagem de emular o código é a menor performance do sistema, já que todas as instruções da máquina abstracta no fim têm de realizar um salto para o código da próxima instrução. Para além disso, torna-se impraticável ter instruções optimizadas para todos os casos possíveis. Neste capítulo apresenta-se o sistema Native YAP (NYAP), que como o nome indica, é um sistema baseado no YAP que traduz o código Prolog em várias linguagens intermédias, de forma a obter código nativo para ser executado directamente pela máquina.

O YAP divide a compilação de um programa Prolog em duas fases. Na primeira fase, são compiladas as cláusulas isoladamente, enquanto que na segunda fase é criado o código para ligação e indexação dos predicados. Nesta primeira implementação do sistema Native YAP apenas são traduzidas para código nativo as instruções que resultam da compilação das cláusulas. O tratamento das instruções de indexação e ligação dos predicados manteve-se inalterado. Assim, a execução de um programa Prolog no Native YAP é realizada transitando entre o emulador do YAP e o executável resultante da compilação do código nativo.

Neste capítulo falar-se-á principalmente sobre a fase de compilação das cláusulas, pois é nesta fase que o Native YAP difere do YAP.

4.1 Esquema de compilação do NYAP

Considerando a grande separação entre o Prolog e as instruções de uma máquina convencional, é evidente que para gerar um bom código a transição tenha que ser feita usando várias linguagens intermédias. O sistema NYAP faz a transição do Prolog para o código nativo, usando três linguagens intermédias distintas (ver figura 4.1). A maior vantagem em usar três linguagens intermédias de níveis diferentes é a de permitir obtermos um maior leque de possíveis optimizações do que seria possível se tivéssemos usado apenas uma linguagem intermédia.

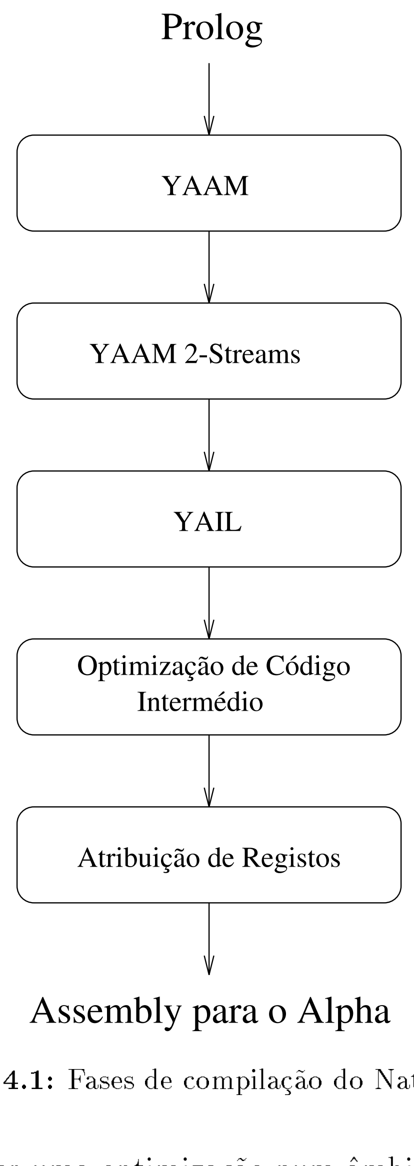

Figura 4.1: Fases de compilação do Native Yap

Com a YAAM conseguimos ter uma optimização num âmbito mais superior, classificação de variáveis (globais ou locais a nível da WAM), optimização das variáveis temporárias, e eliminação de instruções desnecessárias do tipo `unify_var` e `pop`.

Na YAAM 2-Streams consegue-se simplificar o código reduzindo o número de saltos existentes no seio das instruções da YAAM.

Com a YAIL conseguimos ter uma optimização de código a mais baixo nível, eliminando entre outros, atribuições desnecessárias e fazendo um uso mais eficaz dos registos da máquina destino.

Outra vantagem de dividir o processo de compilação em várias fases, é a de se tornar relativamente simples alterar uma fase sem ter que para isso remodelar as outras.

4.1.1 Transformação do Prolog para YAAM

A fase de compilação do Prolog para a YAAM não foi alterada, mantendo-se igual à do YAP (ver secção 2.3).

4.1.2 Transformação da YAAM para YAAM 2-Streams

A transformação da YAAM para a YAAM 2-Streams já foi descrita no capítulo 3.

O uso desta linguagem intermédia é importante pois vem permitir uma redução de saltos (`jumps`) entre as instruções.

4.1.3 Transformação da YAAM 2-Streams para YAIL

O algoritmo que faz a passagem do código YAAM 2-Streams para a YAIL, é um algoritmo simples, que transforma cada instrução da YAAM 2-Streams num conjunto de instruções da YAIL. Durante esta tradução são realizadas algumas optimizações no código de forma a reduzir o número de atribuições necessárias e a optimizar o uso dos registos.

Como exemplo apresenta-se na figura 4.2 a transformação da instrução `put_list` da YAAM 2-Streams para instruções da YAIL. Saliente-se que na instrução `put_list` da YAAM era sempre necessário guardar o valor do registo `Xi` para o caso de aparecer mais tarde a instrução `save_exp`. Na YAIL só é guardado o valor caso se verifique que a instrução

```
case put_list_op Xi:
       emitir_instr(abspair_op,R_Xi,R_H);
       emitir_instr(set_op,R_S,R_H,0);
       emitir_instr(set_op,R_H,R_H,2*CELLSIZE);
       if (aparece_instr_save_exp())
             emitir_instr(set_op,R_D1,R_Xi,0);
       break;
```

Figura 4.2: Tranformação da instrução `put_list` em instruções YAIL.

realmente aparece.

4.1.4 Optimização do código YAIL

A optimização de código tem vindo a acompanhar a evolução dos compiladores. Como resultado existe muita literatura sobre este assunto, [AVA86, DFB94], apresentando muitas técnicas diferentes no âmbito de melhorar o código. Em seguida serão apresentadas as técnicas escolhidas para optimizar o código YAIL. Todas elas podem ser usadas para optimizar outro tipo linguagens, como por exemplo o *C*.

Depois de compilado o código Prolog para o código YAIL, são realizadas oito transformações na tentativa de melhorar o código (ver figura 4.3). As transformações são aplicadas repetidamente ao código até que não se verifiquem mais alterações. Cada transformação foi cuidadosamente estudada de forma a que o código resultante seja melhor, e de forma a que exista um ponto fixo. Entenda-se por código melhor como sendo código mais rápido, mais pequeno, ou ambos.

Segue-se a descrição das oito transformações usadas para melhorar o código. Para cada transformação é apresentado um simples exemplo. Nestes exemplos por simplicidade vai ser usada a instrução `jump(label)` que é equivalente à instrução `test(equal,R0,label)` da YAIL. Note-se que o registo `R0` da YAIL está sempre com o valor zero.

1. **Eliminação de instruções** `label` (figura 4.4). Todos as instruções `label` que não sejam referenciadas por outras instruções no código, podem ser eliminadas. No caso de duas instruções `label` aparecerem seguidas, uma delas pode ser eliminada, fazendo com que todas as suas referências apontem para o outro `label`.

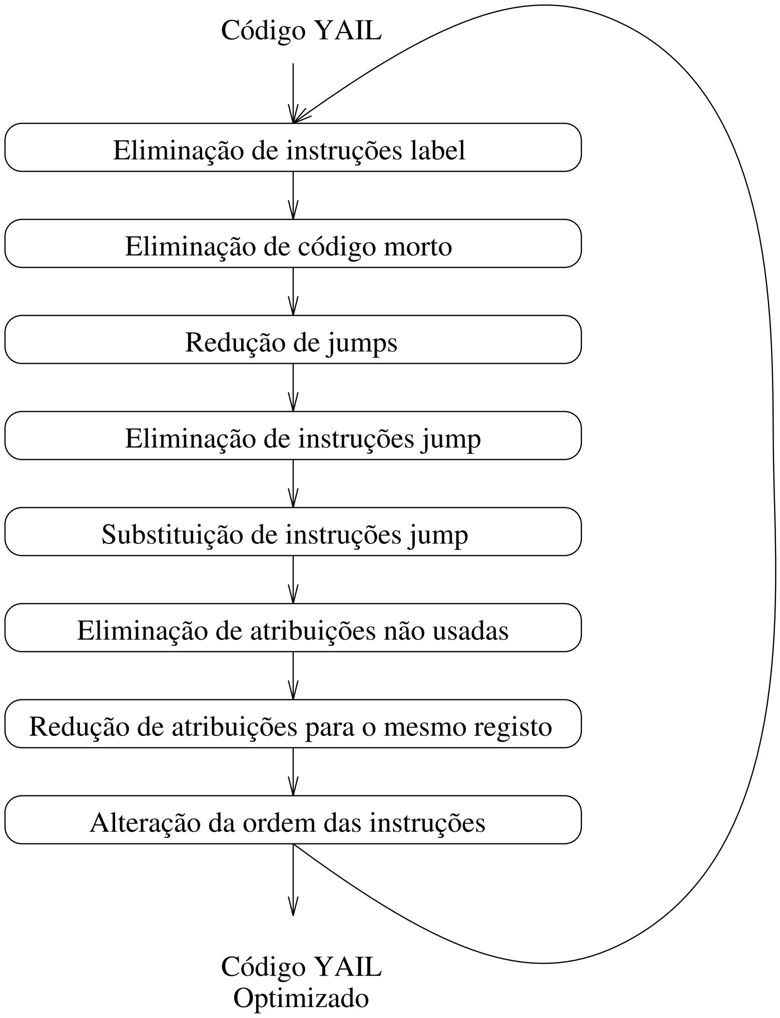

Figura 4.3: Optimização do código YAIL

```
<...>                                      <...>
labely:                                    labelx:
labelx:                         --->       <...>
<...>                                      jump labelx
jump labely
```

Figura 4.4: Eliminação de instruções `label`.

2. **Eliminação de código morto** (figura 4.5). Toda a instrução diferente de `label` que estiver a seguir a um `jump`, `proceed`, `execute` ou `failz R0` pode ser eliminada. Todas as instruções que não alteram o estado da execução do programa podem ser eliminadas. Alguns exemplos deste tipo de instruções são:

- `failnz R0`
- `set Rx,Rx,0`
- `add Rx,Rx,R0`
- `addv Rx,Rx,0`
- `mulv Rx,Rx,1`

Depois de se executar qualquer destas instruções, o estado da execução do programa fica sempre igual ao estado em que o programa estava antes de as executar.

```
<...>                                            <...>
jump labelx                                      jump labelx
<ins. diferentes de label>                       labely:
labely:                                          <...>
<...>                        --->                <...>
labelw:                                          labelw:
failnz R0                                        <...>
<...>                                            <...>
```

Figura 4.5: Eliminação de código morto.

3. **Redução de saltos** (figura 4.6). Aparecendo um `label` seguido por um `jump`, pode--se eliminar o `label` e alterar todas as referências deste ultimo para apontar para o mesmo local apontado pelo `jump`.

```
<...>                                            <...>
labely:                                          <...>
jump labelx                                      jump labelx
<...>                        --->                <...>
jump labely                                      jump labelx
<...>                                            <...>
```

Figura 4.6: Redução de saltos.

4. **Eliminação de saltos** (figura 4.7). Uma instrução `jump` ou `test` seguida pelo `label` para o qual aponta, pode ser eliminada.
5. **Substituição de instruções** `jump` (figura 4.8). Uma instrução `jump` que aponta para

```
<...>                                <...>
jump labelx                ---->     <...>
labelx:                              labelx:
<...>                                <...>
```

Figura 4.7: Eliminação de saltos.

um pequeno bloco de código pode ser substituída por esse mesmo código. Entenda-se por pequeno bloco de código como sendo um conjunto de instruções com dimensão menor que um valor pre-estabelecido. A ideia inerente a esta optimização, é que um salto é uma operação de custo elevado em termos de tempo de execução, assim sendo deve-se evitar ao máximo esse tipo de instruções.

```
<...>                                <...>
jump labelx                          proceed
<...>                      ---->     <...>
labelx:                              labelx:
proceed                              proceed
<...>                                <...>
```

Figura 4.8: Substituição de instruções `jump`

6. **Eliminação de atribuições não usadas** (figura 4.9). Uma instrução que faça uma atribuição para um registo pode ser eliminada se, o valor desse registo não for usado até aparecer uma instrução que faça uma outra atribuição para o mesmo registo.

```
<...>                                <...>
deref Rn,Ra                ---->     <...>
<ins. que não usam Rn>               <sem alteração>
set Rn,Rb,m                          set Rn,Rb,m
<...>                                <...>
```

Figura 4.9: Eliminação de atribuições não usadas.

7. **Redução de atribuições para o mesmo registo** (figura 4.10). Uma instrução que actualiza o registo `Rx` pode ser removida se lhe seguir um conjunto de instruções

que usam `Rx` mais um deslocamento, mais uma outra instrução que atribui um valor a Rx.

```
<...>                                   <...>
set Rx,Rw,n1                            <...>
set Ry,Rx,m              --->           set Ry,Rw,m+n1
set Rx,Rx,n2                            set Rx,Rw,n1+n2
<...>                                   <...>
```

Figura 4.10: Redução de atribuições para o mesmo registo.

8. **Alteração da ordem das instruções** (figura 4.11). A ordem das instruções pode ser alterada de forma a possibilitar que se concentre o maior número de instruções `set` seguidas, tornando assim possível a eliminação de algumas destas instruções.

```
<...>                                   <...>
set Rx,Ry,n              --->           store Rw,Ry,n+m
store Rw,Rx,m                           set Rx,Ry,n
<...>                                   <...>
```

Figura 4.11: Alteração da ordem das instruções.

Saliente-se que todos estes passos não funcionam isoladamente mas sim como um conjunto. Os primeiros cinco passos têm como objectivo reduzir o tamanho do código, eliminando código que nunca chega a ser usado. As últimas três fases da optimização concentram-se no facto de que algumas instruções podem usar o valor de um registo mais um inteiro como parâmetro, eliminando assim a necessidade de efectuar certas atribuições.

Para demonstrar as potencialidades deste tipo de optimização, apresenta-se na figura 4.12 um extracto de código usual em programas de Prolog, que é obtido aquando o uso de listas. Ora no código YAAM (descrito à direita) não era possível realizar qualquer optimização. No código YAIL, como as instruções `load` e `store` acedem a uma posição de memória cujo endereço é indicado pela soma do valor de um registo mais um deslocamento, torna possível que a ordem de certas instruções seja alterada.

Neste caso o novo código, figura 4.13, ficou com menos duas instruções que o código original. Neste extracto de código a percentagem de instruções eliminadas é bastante significativa,

```
set   R_S,R_H,0              \  fim do código da instrução
set   R_H,R_H,2*CELLSIZE     /  unify_last_list_write
addv  R_T,R_Z,valor_átomo_1  \
store R_T,R_S,0               > unify_atom_write átomo_1
set   R_S,R_S,CELLSIZE       /
addv  R_T,R_Z,valor_átomo_[] \
store R_T,R_S,0               > unify_atom_write []
set   R_S,R_S,CELLSIZE       /
```

Figura 4.12: Bloco de código por optimizar

```
addv  R_T,R_Z,valor_átomo_1
store R_T,R_H,0
addv  R_T,R_Z,valor_átomo_[]
store R_T,R_H,CELLSIZE
set   R_S,R_H,2*CELLSIZE
set   R_H,R_H,2*CELLSIZE
```

Figura 4.13: Bloco de código já optimizado

representando 25% do código total, mas num programa completo Prolog é de esperar um redução de código na média dos 5%, com este tipo de optimização.

4.1.5 Atribuição de registos

Até esta fase, a compilação assume que um número ilimitado de registos estão disponíveis. Esta fase de compilação distribui estes registos *virtuais* pelos registos existentes na arquitectura Alpha.

A arquitectura Alpha dispõe de 32 registos inteiros (R0 - R31) e de 32 registos vírgula flutuante (F0 - F31) ambos de 64 bits. O registo inteiro R31 e o registo de vírgula flutuante F31 estão sempre a zero. Os registos Rx são usados pelo Native YAP para âmbito geral, incluindo armazenamento de valores dos registos da YAAM e operações temporárias. Os registos Fx só são usados para operações de vírgula flutuante, não sendo usados para armazenamento temporário de valores. A razão para tal deve-se ao facto de na arquitectura Alpha não ser possível transferir directamente valores entre os registos Rx e os registos Fx. Assim sendo, se fosse usado qualquer registo Fx para armazenar um valor que em seguida fosse necessário passar para um registo Rx, seria preciso usar uma posição de memória para efectuar essa atribuição. Esta operação ficaria assim com uma penalização em termos de tempo de execução superior ao simples uso de posições de memória para armazenamento de dados.

Para se usar os registos da Alpha no Native YAP foi necessário respeitar as seguintes regras impostas pelo `gcc` [Sta93]:

1. Os registos `R29`,`R30`,`R31` e `F31` são registos reservados e como tal não podem ser usados.
2. O registo `R26` é usado para retornar o `PC`.
3. O registo `R0` é usado para retornar valores entre funções.
4. Os registos `R21` até `R16` são usados para passar os argumentos de funções. Por simplicidade passaremos a designar estes registos por *registos argumentos*.
5. Apenas os registos `R9` até `R15` e `F2` até `F9` estão disponíveis para serem usados entre chamadas de funções. Por simplicidade passaremos a designar estes registos por *registos globais*.
6. Os restantes registos devem ser usados com cuidado porque o seu conteúdo não é guardado. Por simplicidade passaremos a designar estes registos por *registos locais*.

Facilmente se conclui que, apesar da arquitectura Alpha dispor de um bom número de registos, não é possível manter todos os registos da YAAM nos registos da máquina. O NYAP dispõe os registos da YAIL nos registos da Alpha da seguinte forma:

- os registos `R_H`, `R_S`, `R_ENV`, `R_Y`, `R_TR` e `R_P` são usados nos registos globais `R10` até `R15`. Estes registos da YAIL são usados da mesma forma pelo código nativo e pelo emulador.
- Os registos `R_B`, `R_HB` e `R_CP` são usados nos registos argumentos `R20` até `R22` durante o código nativo, e são usados como posições de memória durante o emulador.
- os registos `R_T`, `R_D1`, `R_D2` e `R_M`, são usados nos registos argumentos `R16` até `R19`. Estes registos da YAIL são usados apenas no código nativo.
- Para os restantes registos da YAIL, `R_X[i]`, o Native YAP dispõe de uma área de memória onde guarda o valor desses registos. O endereço do início dessa área é mantido no registo global `R9` da arquitectura Alpha. Sempre que é necessário ler ou

atribuir um valor a esses registos da YAIL, é feito um `load` ou `store`, respectivamente usando o registo `R9` mais um deslocamento.

Saliente-se que se fosse executado apenas código nativo, poder-se-ia usar mais registos da Alpha obtendo assim um código mais optimizado. Usar os registos `X[i]` como sendo registos da Alpha poderia dar uma boa melhoria ao tempo de execução do programa, mas tal só seria possível se a máquina abstracta do YAP estivesse escrita em código nativo também.

4.1.6 Geração do código assembly para as cláusulas

Nesta fase o código YAIL optimizado já só usa os registos disponíveis na arquitectura Alpha, tornando o processo de gerar código nativo relativamente simples.

As instruções simples da YAIL (ver tabelas 3.2 e 3.3), são transformadas directamente em instruções do Alpha. As instruções relativas à implementação de etiquetas (ver tabela 3.6) são transformadas também em uma ou duas instruções da Alpha. Quanto às instruções complexas (ver tabela 3.5), a sua transformação é realizada de forma diferente, conforme a instrução em questão. As instruções `fail_X`, `trail_XXX`, `pop` e `push` são transformadas directamente para um pequeno conjunto de instruções da arquitectura Alpha. As instruções `unify_XXX` são transformadas numa chamada a uma função, que se encontra implementada no YAP sobre o mesmo nome da instrução YAIL. A instrução `deref` é tratada de uma forma especial. Durante a geração e optimização do código YAIL, tem-se o cuidado de manter junto a esta instrução os testes que fazem uso do resultado da desreferenciação. Desta forma no código assembly gerado para o `deref` realiza-se imediatamente o salto para o local onde a execução deve continuar conforme se conclua que o resultado da desreferenciação é variável ou não. Na figura 4.14 pode-se ver os esquemas usados para esta instrução. À direita contempla-se o caso em que o código que segue a instrução é para `X var`. À esquerda contempla-se o caso inverso.

Ao mesmo tempo que se gera o código assembly para a arquitectura Alpha, realiza-se uma análise às instruções que se vão gerando, por forma a tentar melhorar a organização destas tendo em conta os dois pontos seguintes:

- A primeira versão do processador Alpha (DECShip 21064) processa duas instruções de cada vez. Todas as instruções têm um tamanho fixo de 32 bits. Para obter a

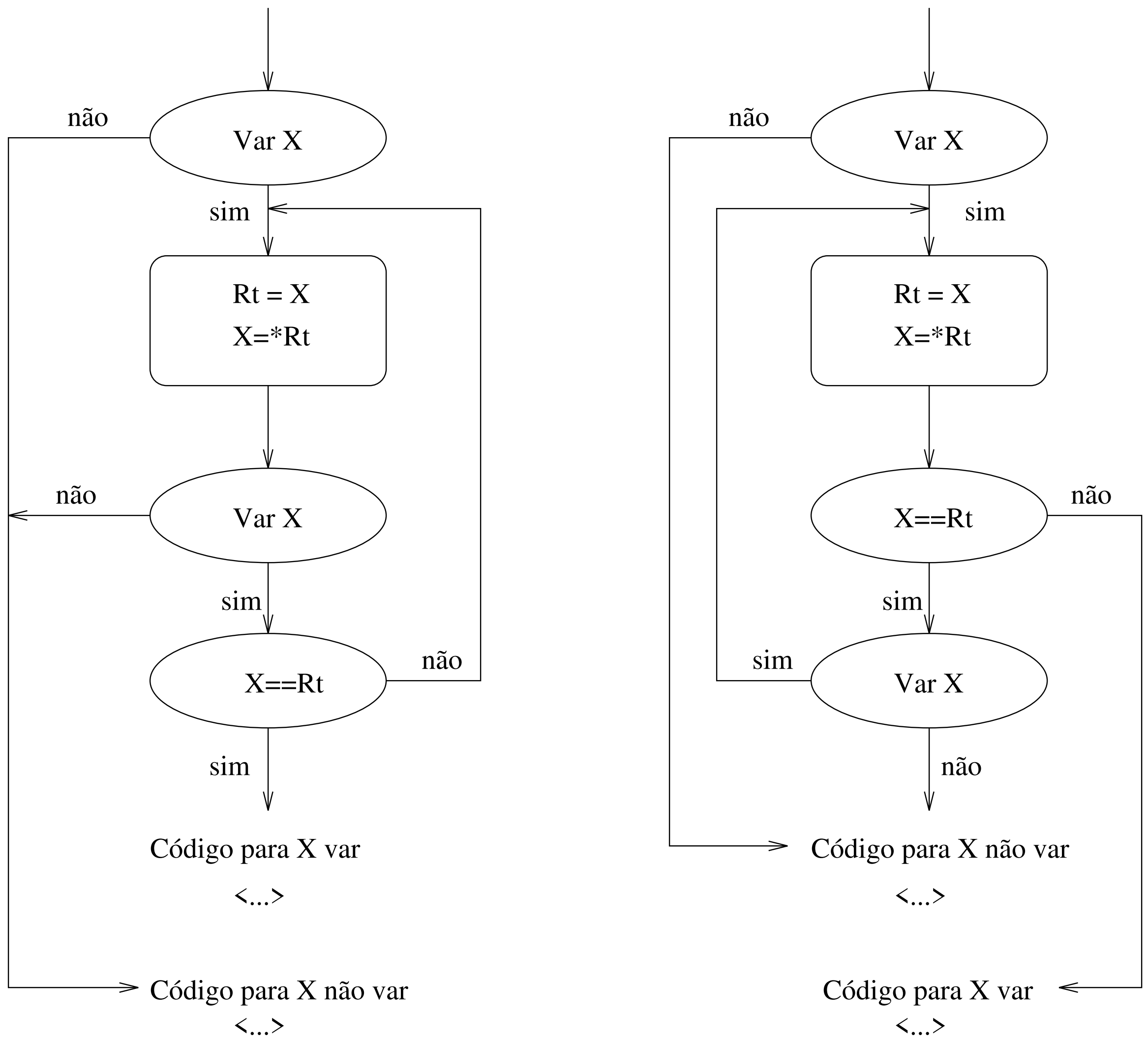

Figura 4.14: Esquema para o código da instrução `deref(X)`.

máxima performance do CPU o ideal é que todos os saltos sejam realizados para um endereço múltiplo de 8, evitando assim problemas de realinhamento das instruções (8 bytes= 2*32= 64 bits). Antes de gerar uma instruções onde se sabe à partida que vai ser um ponto de entrada (segue um `label`), é confirmado se o endereço vai ser múltiplo de 8 ou não. Se sim, a instrução é gerada imediatamente. Caso contrário, antes de gerar a instrução em questão insere-se a instrução `fnop` [1] que não tem nenhuma influência no estado da execução da máquina, e é usada apenas para forçar o alinhamento das instruções.

Na figura 4.15 à esquerda podemos ver o caso em que uma instrução (`instd`) faz um

[1] `fnop` é a designação reduzida para a instrução `cpys $f31,$f31,$f31`

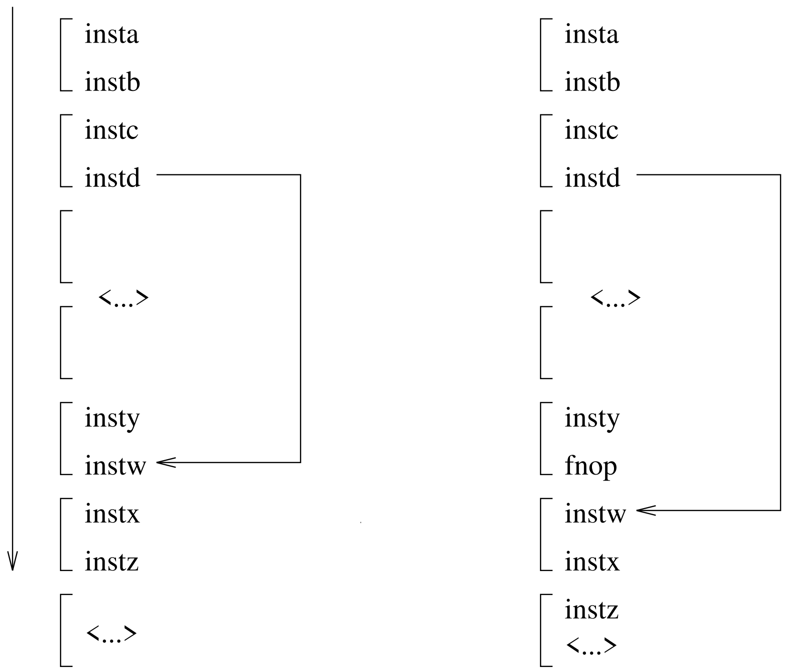

Figura 4.15: Evitando saltos para endereços não alinhados

salto para uma instrução não alinhada (`instw`). Para que este salto não traga uma penalização muito grande no tempo de execução do programa, introduzimos nestes casos a instrução `fnop` forçando a instrução `instw` a ficar num endereço múltiplo de 8 (ver figura à direita).

Na implementação do Native YAP criaram-se dois níveis para a inserção das instruções `fnop`, a saber:

- **Alinhar apenas as instruções a seguir a um** `jump`: neste método apenas são alinhadas as instruções no código que possam ser pontos de entrada e que apareçam imediatamente a seguir a uma instrução `jump`. A vantagem deste método é não inserir instruções em áreas de código executáveis. Por outro lado tem a desvantagem de que nem todas as instruções do código, que são pontos de entrada, ficam alinhadas.

  Por exemplo, na figura 4.15 a instrução `fnop` só seria inserida caso a instrução `insty` fosse uma instrução de salto. Neste caso a instrução `fnop` ficaria numa área "não executável".

- **Todas as instruções alinhadas**: neste método são alinhadas todas as instruções do código que possam ser pontos de entrada. A vantagem deste método é de que todos os saltos são feitos para instruções alinhadas. A desvantagem é, de por vezes, ser necessário inserir instruções `fnop` em locais executáveis.

Por exemplo, na figura 4.15, supondo que a instrução `insty` não seja um salto, depois de inserir a instrução `fnop` passar a existir mais uma instrução.

Por defeito o Native YAP usa o primeiro método, embora nenhum dos dois métodos seja claramente superior, como será evidenciado no capítulo 5.

- A primeira versão do processador Alpha tem três unidades de processamento das instruções, *Fbox*, *Abox* e *Ebox*. A unidade *Fbox* executa todas as operações relacionadas com os registos de vírgula flutuante. Esta unidade de processamento não chega a ser usada no código nativo, pois todas as operações com vírgula flutuante estão a cargo do YAP. A unidade *Abox* executa todas as instruções do tipo `load` e `store`. A unidade *Ebox* executa as restantes instruções sobre os registos inteiros, incluindo as operações aritméticas, operações lógicas, operações de comparação e as instruções de saltos.

  Cada uma destas unidades de processamento só pode processar uma instrução de cada vez. Porém, as três unidades podem estar a processar instruções em paralelo. O ideal seria as instruções poderem ser organizadas em pares que usassem unidades de processamento diferentes. No entanto tal não é possível, porque como seria de esperar, as instruções mais frequentes no código nativo são as instruções da unidade *Ebox*. O caso mais frequente, que pode ser optimizado, é o aparecimento de duas instruções da unidade *Abox* seguidas. Neste caso o que se faz é reordenar uma dessas instruções de forma a que uma das instruções do tipo *Ebox* fique no meio.

Nesta fase, para além de transformar o código YAIL em código assembly, é também gerado uma série de `labels`, (ver figura 4.16), um no início do código da cláusula e um a seguir de cada instrução `call`.

```
início_clausula:
       ...
       call ()
ponto_retorno1:
      ...
      call ()
ponto_retorno2:
      ...
proceed
```

Figura 4.16: Novos `labels` gerados em cada cláusula

O endereço `início_clausula` servirá para que se possa saltar para o início da cláusula quando se pretender executá-la. Os endereços `ponto_retorno` servem para que a instrução `call` possa actualizar o registo `CP` com o endereço de retorno.

4.2 Compilação do código assembly gerado e ligação do executável resultante com o YAP

Depois de compiladas todas as cláusulas do programa em Prolog para código assembly, torna-se necessário invocar o assemblador para esse código de forma a ser possível executar o código nativo dentro do YAP. Nesta secção descreve-se como tal está implementado.

Nas secções anteriores deixou-se intencionalmente a dúvida sobre se o código de cada cláusula era gerado em ficheiros/funções diferentes, ou tudo no mesmo ficheiro, numa só função. Ambos os métodos foram testados, e ambos obtiveram tempos de execução idênticos. Optou-se por gerar todo o código para apenas uma função, pois numa futura implementação em que a indexação seja incluída no código nativo, o salto entre as cláusulas pode ser efectuado localmente. Torna-se também desta forma, mais fácil criar uma nova implementação em que o código nativo não necessite da máquina abstracta para correr.

O ficheiro com o código dos predicados nesta primeira implementação do Native YAP é organizado obedecendo aos seguintes parâmetros (ver exemplo na figura 4.17):

- O código de todas as cláusulas é inserido dentro de uma só função.
- No início da função é criado um vector com os endereços onde a execução pode continuar. Estes pontos incluem os endereços que marcam o início das cláusulas, e todos os endereços que são usados para marcar um ponto de continuação, usualmente pontos de retorno para as instruções `call`.
- A função recebe um parâmetro de entrada. Este parâmetro servirá para indicar qual a cláusula a executar ou qual o ponto em que a execução deverá continuar.
- A primeira instrução da função é um salto para um endereço existente no vector com os endereços, cuja posição é indicada pelo parâmetro da função.
- A função retorna o valor zero para representar um `fail`, ou qualquer outro valor para indicar que a execução deve continuar na máquina abstracta do YAP com a

```
int funcao(int par) {
static void *labels[]= {
       &&início_clausula_1,
       ...
       &&início_clausula_n,
       &&ponto_retorno_n_1,
       &&...
       &&ponto_retorno_n_m,
       ...
}

goto *((void *) labels[par]);

...

início_clausula_n:
       ...
       /* call () */
       CP=ponto_retorno_n_1;
       return;
ponto_retorno_n_1:
       ...
       /* call () */
       CP=ponto_retorno_n_m;
       return;
ponto_retorno_n_m:

...

} /* fim da funcao */
```

Figura 4.17: Organização do código de todas das cláusulas numa só função

instrução indicada pelo registo `P`.

O ficheiro com o código nativo é compilado para um ficheiro objecto, que é carregado dinamicamente (durante a execução) para a memória usando a função `dlopen` [2] (ver figura 4.18). O endereço da nova função é guardado na variável `endereço`.

```
system("gcc codigo -c -fpic");
system("ld codigo.o -o codigo.so -shared -expect_unresolved "*" ");
handle=dlopen("./codigo.so",RTLD_NOW);
endereço=dlsym(handle,"funcao");
```

Figura 4.18: Extrato do algoritmo de compilação do ficheiro com o código das cláusulas

[2] o `dlopen` é uma função definida na biblioteca `ldl` de ligação dinamica do sistema UNIX V

O código da função fica disponível para ser chamado a partir de qualquer área do YAP, como se de uma função já implementada se tratasse. Basta para tal invocar o comando `(*endereço)(n);`.

Para que a máquina abstracta do YAP pudesse usar o código nativo foi necessário alterá-la. Em primeiro lugar foi necessário criar uma nova instrução que estabelecesse a ligação com o código nativo. Essa instrução, que foi designada por `Run_Native`, recebe obrigatoriamente um parâmetro inteiro que serve para identificar o endereço do início da cláusula. Durante a compilação do programa Prolog, o "velho" código YAAM de cada cláusula é substituído por essa instrução.

```
#define execute_abstract()
     goto *P;

#define execute_native()
     goto Run_Native;

#define execute_instrution()
     if condição(P) execute_native() else execute_abstract();

#define execute_next_instruction()
     { P += Tamanho_Instrucao;
     execute_abstract();     }

#define execute(NEWP)
     { P = NEWP;
     execute_instruction();   }
```

Figura 4.19: Instruções na máquina abstracta para executar a próxima instrução

Foi também necessário alterar as instruções da máquina abstracta que fazem o salto para a próxima instrução. Isto porque a próxima instrução no Native YAP, tanto pode ser na máquina abstracta como no código nativo. São cinco (ver figura 4.19) as instruções que controlam como vai ser executada a próxima instrução na máquina abstracta:

- `execute_abstract()`. Esta instrução faz com que a execução passe para a instrução da máquina abstracta apontada por `P`.
- `execute_native()`. Esta instrução indica que a execução vai continuar no código nativo. Como tal, é efectuado um salto para o endereço `Run_Native` onde é feita a chamada ao código nativo. O parâmetro a usar na chamada do código nativo é o registo `P`.

- `execute_instruction()`. Esta instrução faz um teste ao valor do registo `P` para decidir se vai continuar na máquina abstracta ou se passa para o código nativo. Esta instrução é usada quando não se sabe se a execução vai continuar em código nativo ou na máquina abstracta.

- `execute_next_instruction`. Esta instrução é usada sempre que se sabe que a execução vai continuar na máquina abstracta, não havendo por isso necessidade de testar o registo `P`.

- `execute(NEWP)`. Esta instrução actualiza `P` com o valor que recebe como parâmetro. Como nada se sabe quanto ao parâmetro faz um `execute_instruction()` para decidir se continua na máquina abstracta ou se salta para o código nativo.

Resumindo, para executar uma cláusula, o NYAP começa por executar a máquina abstracta, só que em vez de ter uma sequência de instruções YAAM do tipo `get_XXX`, `unify_XXX`, `put_XXX`, etc., tem apenas uma instrução, `Run_Native`. Esta instrução ao ser executada passa o controle para o código nativo. No começo da função do código nativo é feito um salto para o início do código da cláusula a executar. A partir daí o código é executado em sequência, voltando para a máquina abstracta ou no fim do código da cláusula ou se precisar de executar uma outra cláusula (`call`).

Na figura 4.20 está ilustrado este processo para a execução da cláusula `P2`. O código começa na máquina abstracta onde é executada a instrução `Run_Native` com o parâmetro `2` que serve para identificar qual a cláusula que se quer executar. A instrução `Run_Native` faz a chamada à função do código nativo passando o seu parâmetro. Já dentro da função, o que a primeira instrução faz, é saltar para o início do código da cláusula cujo endereço é identificado pelo parâmetro de entrada. Em seguida o código é executado em sequência até encontrar um `return`, que faz com que a execução retorne à máquina abstracta. Saliente-se que neste exemplo muito simples, assume-se que a cláusula `P2` não faz nenhuma chamada a outra cláusula.

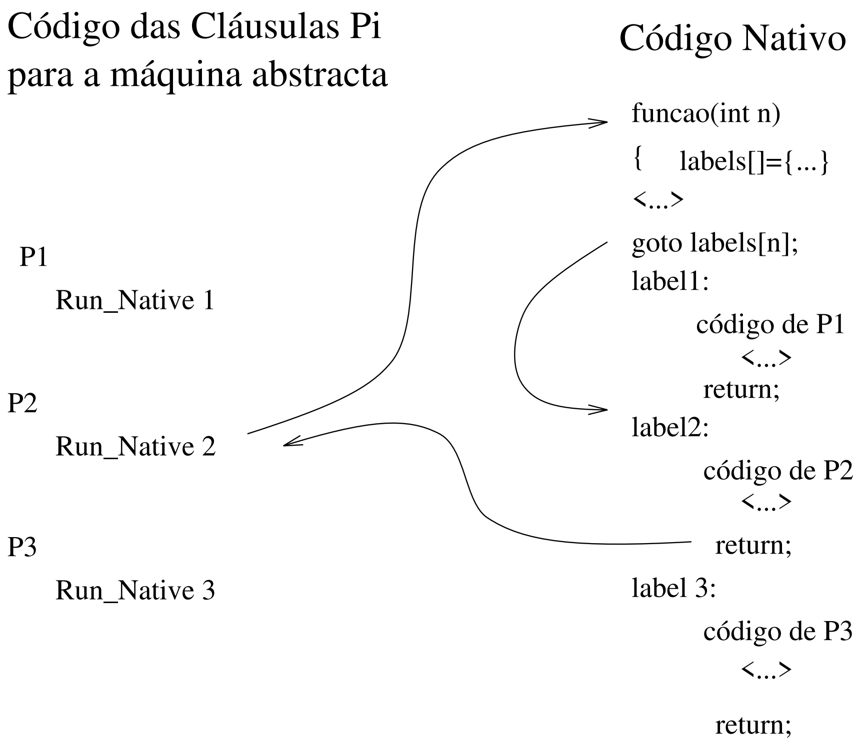

Figura 4.20: Ilustração da execução de uma cláusula (P2)

Capítulo 5

Performance do sistema NYAP

Neste capítulo, procede-se a uma avaliação preliminar da performance do sistema desenvolvido. Para isso, apresentam-se e analisam-se os resultados obtidos pela execução de um amplo conjunto de programas de teste. Todos os resultados foram obtidos numa máquina da Digital, modelo DECsystem 3000, com um processador DECChip 21064 [1] a 175Mhz com 8Kb+8Kb de cache interna, 64 Mbytes de memória RAM e 512Kb de cache externa. O sistema operativo usado foi o OSF1 V3.0 (rev 347).

5.1 Programas de teste

O conjunto de programas usado é constituído por programas de teste habitualmente usados pela comunidade científica do Prolog, tendo, a maior parte deles, sido usados na avaliação de outros sistemas, como seja no Aquarius e no Parma. Segue-se uma breve descrição de cada um dos programas de teste usados (no apêndice A apresenta-se o código Prolog).

- **ham**: encontra os caminhos hamiltonianos num grafo.
- **houses**: resolve o problema "quem possui a zebra" através da verificação de um conjunto de restrições.
- **magic**: resolve o problema do cubo mágico pelo método de geração e teste.
- **mu**: prova um teorema no sistema de Hofstadter "mu-math".

[1] Usualmente designado por Alpha

- **nreverse**: inverte uma lista com 30 elementos.
- **nsort**: ordena uma lista com oito elementos pelo método de geração e teste.
- **poly_10**: uso de aritmética simbólica para elevar um polinómio à décima ordem.
- **qsort**: ordena uma lista com 50 elementos pelo método *quick-sort*.
- **queens**: resolve o problema da colocação das 8 rainhas através da permutação de duas listas.
- **queens_8**: resolve o problema das 8 rainhas por procura exaustiva.
- **query**: pesquisa numa base de dados para responder a questões, envolvendo aritemética inteira.
- **tak**: programa altamente recursivo. A recursão é controlada por operações inteiras simples.
- **zebra**: puzzle lógico baseado em restrições que é resolvido por procura exaustiva.

Gostaríamos de poder afirmar que os programas de teste usados são programas Prolog típicos, mas na verdade, é difícil saber o que é um programa Prolog típico. Porém, estamos conscientes que o conjunto de programas de teste usado é adequado para conseguirmos mostrar um panorama amplo sobre o uso da geração de código nativo na compilação de Prolog.

5.1.1 Performance do Native Yap versus Yap

Na tabela 5.1 apresentam-se os tempos que cada programa de teste demora a correr no sistema YAP e no sistema Native YAP. Os tempos deste último sistema são apresentados sobre a forma de aceleração relativamente aos tempos do primeiro. Quanto maior for o valor, melhor será a performance. Os resultados representam os menores tempos obtidos num conjunto de 5 ou mais execuções.

O tempo de execução de alguns dos nossos programas de teste é muito reduzido, o que torna a obtenção de resultados coerentes mais complicada. Esse problema foi resolvido fazendo-se múltiplas execuções. O número de execuções variou entre 1 e 5000.

Programas de Teste	YAP		NYAP
	milisegundos	escalado	escalado
ham	2268,00	1	1,46
houses	4,67	1	1,34
magic	2488,00	1	1,74
mu	2,80	1	1,24
nreverse	1,06	1	1,91
nsort	2029,00	1	1,58
poly_10	0,13	1	1,64
qsort	2,16	1	1,87
queens	12101,00	1	1,66
queens_8	25,60	1	1,64
query	3,61	1	1,17
tak	732,00	1	1,27
zebra	130,00	1	1,23
Média		1	1,52

Tabela 5.1: Aceleração do NYAP sobre o YAP

Pela última linha da tabela 5.1 podemos concluir que o Native YAP tem uma aceleração média de `1.52` sobre o YAP, o que é muito bom considerando o facto de que apenas o código das cláusulas é executado em código nativo.

A diferença que existe entre os resultados nos vários programas pode explicar-se pelo facto de que alguns programas de teste passam uma grande parte do tempo a fazer chamadas à biblioteca de predicados do YAP. Ora como esse código é igual para o YAP e Native YAP, não pode haver uma grande melhoria. Outros programas de teste, como seja o `nreverse`, essencialmente o que fazem é manipular listas, não requerendo tantas chamadas ao YAP, e por isso conseguem obter uma melhor performance.

5.1.2 Análise da performance do sistema Native YAP

Nesta secção faz-se uma análise mais detalhada da performance do sistema NYAP. Na tabela 5.2 apresentam-se os tempos de execução dos diversos programas de teste, obtidos em sete variações do NYAP.

As primeiras três versões diferem na forma como o alinhamento das instruções assembly é realizado (ver secção 4.1.6):

- **ver1**: esta versão, usa o parâmetro por defeito, que é gerar instruções `fnop` apenas para alinhar as instruções que seguem uma instrução `jump`.
- **ver2**: nesta versão, todas as instruções que são pontos de entrada de código são forçados a ficar num endereço múltiplo de 8, inserindo para isso a instrução `fnop`.
- **ver3**: nesta versão, nenhuma instrução é alinhada, ou seja, não se geram instruções `fnop`.

As restantes quatro versões diferem na fase de optimização do código YAIL:

- **ver4**: nesta versão, as primeiras quatro fases da optimização do código YAIL estão desactivadas (ver figura 4.3).
- **ver5**: nesta versão, a quinta fase de optimização do código YAIL, "substituição de `jumps`", está desactivada. Relembra-se que esta fase substitui, quando possível, as instruções `jump` pelo bloco de código para onde apontam. Este bloco deve ter um número de instruções menor que um valor pre-estabelecido. Nas outras versões esse valor não foi limitado.
- **ver6**: nesta versão, as últimas três fases de optimização do código YAIL estão desactivadas.
- **ver7**: nesta versão, todas as fases de optimização do código YAIL estão desactivadas.

Saliente-se que existem outras optimizações de código que aqui não podem ser avaliadas, devido à impossibilidade da sua desactivação.

Comparando os resultados da tabela 5.2, podemos verificar alguns aspectos interessantes:

- O método de alinhamento da versão 1 é aquele que maior performance obteve. Porém, apesar da versão 2 ser mais lenta, conseguiu em alguns programas de teste obter a velocidade mais elevada (`queens`,`queens_8` e `query`).
- A versão 7, que tem a optimização do código YAIL desactivada, é claramente a mais lenta como seria de esperar.
- A versão 5, que tem desactivada a fase de optimização que "substitui as instruções `jump`", obteve em três casos os melhores resultados (`ham`, `houses` e `zebra`). Supõe-se

Programas	ver1	ver2	ver3	ver4	ver5	ver6	ver7
ham	1,42	1,41	1,42	1,34	**1,46**	1,24	1,45
houses	1,30	1,22	1,31	1,15	**1,34**	1,27	1,20
magic	**1,74**	1,56	1,73	1,71	1,68	1,71	1,69
mu	**1,24**	1,23	1,20	1,22	1,24	1,16	1,24
nreverse	**1,91**	1,87	1,90	1,87	1,89	1,83	1,38
nsort	**1,58**	1,34	1,56	1,43	1,45	1,47	1,42
poly_10	**1,64**	1,55	1,57	1,56	1,62	1,55	1,61
qsort	**1,87**	1,53	1,86	1,69	1,86	1,60	1,77
queens	1,65	**1,66**	1,60	1,44	1,54	1,53	1,39
queens_8	1,57	**1,64**	1,45	1,34	1,37	1,40	1,18
query	1,15	**1,17**	1,15	1,10	1,13	1,09	0,98
tak	**1,27**	1,20	1,20	1,22	1,06	1,16	1,05
zebra	1,18	1,20	1,17	1,12	**1,23**	1,19	1,19
Média	**1,50**	1,43	1,47	1,40	1,45	1,40	1,35

Tabela 5.2: Aceleração de várias versões do NYAP sobre o YAP

que este comportamento fique a dever-se ao crescimento de código que a "substituição de `jumps`" provoca, levando a um maior conflito na cache. Por exemplo, no programa de teste `ham` a "substituição dos `jumps`", provocou um crescimento na ordem dos 2,3 (ver tabela 5.3). Saliente-se que a versão 5 e 7 são as únicas que têm esta fase de optimização desactivada. Este facto comprova que a quinta fase da optimização do código YAIL, deve ser usada com cuidado, pois pode conduzir a uma redução na performance.

5.1.3 Código nativo versus código *C*

Um compilador de Prolog com geração de código nativo tem o inconveniente de não ser facilmente portável para outras máquinas. Uma das alternativas ao uso do código nativo é o uso do código *C*. Nesta secção faz-se uma comparação da performance entre versões do YAP com geração de código nativo e geração de código *C* (ver tabela 5.4).

Para que se perceba a importância de um bom uso dos registos, foram criadas três versões diferentes do YAP com geração de código *C*:

- **YAPC1**: esta versão gera código *C* independente da máquina. Os registos da YAIL são usados como variáveis normais. A atribuição dos registos é deixada a cargo do

Programas	**ver1**	**ver2**	**ver3**	**ver4**	**ver5**	**ver6**	**ver7**
ham	187K	202K	187K	170K	82K	187K	91K
houses	56K	56K	56K	56K	48K	56K	48K
magic	213K	230K	213K	221K	213K	222K	222K
mu	47K	47K	47K	47K	39K	47K	47K
nreverse	39K	39K	39K	39K	31K	39K	39K
nsort	38K	39K	38K	39K	30K	39K	39K
poly_10	65K	65K	65K	73K	56K	65K	65K
qsort	39K	39K	39K	48K	39K	39K	40K
queens	39K	47K	39K	47K	39K	47K	47K
queens_8	39K	39K	39K	47K	38K	39K	39K
query	58K	58K	58K	66K	49K	58K	58K
tak	30K	30K	30K	38K	30K	30K	30K
zebra	48K	57K	48K	56K	47K	48K	48K

Tabela 5.3: Tamanho do ficheiro objecto resultante da compilação de código nativo

compilador *C*.

- **YAPC2**: esta versão gera código *C* usando alguns registos da YAIL directamente como registos do processador. Uma grande parte da atribuição dos registos do processador é ainda deixada a cargo do compilador *C*.

- **YAPC3**: esta versão usa ainda mais registos da máquina destino. Para isso foi necessário usar algumas instruções assembly em certos pontos críticos, para evitar que o compilador de *C* alterasse o conteúdo dos registos.

Analisando os resultados apresentados na tabela 5.4 verifica-se que os melhores resultados nas versões com geração de código *C* se obtêm quando se faz uso do maior número de registos da YAIL directamente como registos do processador. Pode-se também verificar que a versão YAPC1, que usa os registos da YAIL como variáveis, teve uma perda muito grande de performance em relação ao YAP.

Verifica-se também que o código *C* ainda não consegue alcançar a performance obtida pelo uso de código nativo. Isto deve-se principalmente ao facto do compilador *C* gerar em muitos casos instruções supérfluas. Por exemplo, quando o compilador *C* encontra uma instrução `if`, muitas vezes gera dois saltos quando um é suficiente. Na tabela 5.5 pode-se verificar que o tamanho do ficheiro objecto resultante pela compilação do código *C* (versão YAPC3) é superior ao tamanho do ficheiro objecto resultante do código nativo.

Programas	NYAP	YAPC1	YAPC2	YAPC3
ham	1,46	0,70	0,94	1,17
houses	1,34	0,65	0,81	1,00
magic	1,74	0,49	1,07	1,07
mu	1,24	0,59	0,85	0,85
nreverse	1,91	0,74	0,95	1,31
nsort	1,58	0,71	0,98	0,99
poly_10	1,64	0,74	1,08	1,30
qsort	1,87	0,77	1,23	1,44
queens	1,66	0,81	1,19	1,35
queens_8	1,64	0,98	1,21	1,23
query	1,17	0,93	1,00	1,11
tak	1,27	0,87	1,07	1,03
zebra	1,23	0,85	0,97	1,02
Média	1,52	0,76	1,03	1,14

Tabela 5.4: Aceleração do YAPC sobre o YAP

Programas	NYAP	YAPC
ham	187K	219K
houses	56K	65K
magic	213K	334K
mu	47K	55K
nreverse	39K	40K
nsort	38K	39K
poly_10	65K	82K
qsort	39K	48K
queens	39K	48K
queens_8	39K	47K
query	58K	75K
tak	30K	38K
zebra	48K	57K

Tabela 5.5: Tamanho do ficheiro objecto, NYAP vs YAPC

5.1.4 Outros factores

Nesta secção descrevem-se alguns factores que contribuíram para que a aceleração do Native YAP sobre o YAP não fosse mais representativa. Esses factores são:

- Apenas o código das cláusulas é transformado em código nativo.
- A execução passa em média 50,61% do seu tempo no emulador (ver tabela 5.6). O restante tempo de execução, 49,39%, é passado no código nativo (inclui o tempo que se perde a saltar entre o emulador e o código nativo).
- O emulador no Native YAP ficou cerca de 10% mais lento devido à necessidade de se efectuar um teste ao valor do registo `P`, quando da execução da próxima instrução, para determinar se a execução continua no emulador ou se passa para o código nativo.
- Não foi possível explorar ao máximo os registos da arquitectura Alpha, já que a passagem entre o código nativo e o emulador poderia alterar o valor de alguns registos. Uma melhoria significativa seria obtida com o uso dos registos da YAAM `X[i]` directamente nos registos da arquitectura Alpha, mas tal só seria possível se mais instruções da YAAM fossem executadas em código nativo.

Programas de Teste	percentagem de tempo no emulador
ham	52,14%
houses	52,28%
magic	29,28%
mu	49,27%
nreverse	46,59%
nsort	51,86%
poly_10	49,16%
qsort	51,57%
queens	50,96%
queens_8	59,99%
query	59,95%
tak	57,50%
zebra	48,13%
Média	50.61%

Tabela 5.6: Percentagem de tempo no emulador durante a execução no NYAP.

5.2 Comparação de performance com o SICStus

Uma comparação directa entre a aceleração do Native YAP sobre o YAP e a aceleração do SICStus emulado sobre o SICStus Nativo não é possível porque este último gera código nativo para o programa Prolog todo, enquanto o Native YAP só gera código nativo para as cláusulas. De qualquer forma, para dar uma ideia geral, efectuamos uma comparação relativa, usando para isso a aceleração que o Native YAP obteve sobre o YAP no código das cláusulas.

Na tabela 5.7 apresentam-se os valores de aceleração do SICStus Native sobre o SICStus emulado [Hay94]. Apresenta-se também a aceleração que o Native YAP obteve sobre o YAP considerando apenas o código das cláusulas. Para exemplificar como estes valores são obtidos, apresentam-se os cálculos realizados para o programa de teste nreverse (idêntico para os outros): o nreverse demorou 1063 microsegundos a executar no YAP e 557 microsegundos no Native YAP. Neste último, 46,59% desse tempo foi passado no emulador, (557×0.4659) 260 microsegundos, e 297 segundos no código nativo. Temos então que o YAP demorou pelo menos 803 microsegundos ($1063 - 260$) a executar o código das cláusulas, (pelo menos porque o emulador do Native YAP é mais lento que o do YAP). A aceleração do Native YAP sobre o YAP neste caso foi de 2,70 ($803 \div 297$). Saliente-se que esta avaliação não é cem por cento correcta, mas dá-nos um panorama geral.

Programas de Teste	SICStus Nativo na SPARC	SICStus Nativo na MIPS	Native YAP na Alpha
nreverse	1.18	2.60	2.70
qsort	1.06	1.91	2.80
query	1.02	2.77	1.42

Tabela 5.7: Aceleração do código Nativo sobre o emulador, SICStus vs YAP

Podemos assim concluir que a aceleração que o Native YAP obteve sobre o YAP é boa, pois ultrapassou os valores do SICStus na SPARC, e também ultrapassou a aceleração obtida no SICStus na MIPS nos programas de teste nreverse e qsort.

Capítulo 6

Conclusões

Um dos principais objectivos associados a qualquer implementação de Prolog é a de conseguir obter o máximo da performance. Nos últimos anos presenciaram-se melhorias significativas na compilação de Prolog, resultando no retorno da ideia da execução em código Nativo. São duas as razões principais que levaram os sistemas recentes a evitar a emulação:

- **Redução do *overhead***: a compilação para código nativo elimina o *overhead* intrínseco em emular as instruções de uma máquina abstracta, que ocupa uma boa percentagem do tempo de execução.

- **Aumento da flexibilidade**: a compilação para código nativo permite uma especialização das instruções, facto que na WAM requeria a construção de novas instruções para a máquina abstracta. Note-se que a redução da granularidade das instruções WAM não é muito boa ideia, pois leva ao aumento do *overhead*.

A principal motivação desta tese foi a implementação de um sistema Prolog com geração código nativo usando como base o sistema YAP. O YAP é um compilador de Prolog baseado na WAM. O código Prolog é compilado para a linguagem YAAM, uma variante da WAM, que por sua vez é emulada durante a execução. Dado o nosso objectivo de suportar geração de código nativo para Prolog, tornou-se necessário escolher uma arquitectura alvo. A arquitectura escolhida foi a arquitectura Alpha.

6.1 Síntese do Trabalho

Para que o YAP gerasse o código nativo para as cláusulas de uma forma eficiente foi necessário adicionar novas fases na compilação do Prolog de forma a se conseguir obter um maior leque de possíveis optimizações. Essas novas fases são:

- **Transformação do código YAAM em código YAAM 2-Streams**. A YAAM 2-Streams é uma nova linguagem intermédia que resulta de separar a YAAM em duas áreas de código, uma primeira com o código em modo de leitura, e uma segunda com o código em modo de escrita. O uso desta nova linguagem intermédia veio permitir uma redução significativa nos saltos dentro das instruções que tinham dois modos de execução, tipo `unify`. Esta redução do número de saltos não teria muita importância num emulador porque no final das instruções nos emuladores é sempre necessário realizar um salto de qualquer das formas. Numa implementação de código nativo essa redução é bastante importante já que nas arquitecturas *RISC* actuais, um salto é uma operação com uma penalização muito elevada em termos de tempo de execução.

- **Transformação do código YAAM 2-Streams em código YAIL**. Apesar do uso da YAAM 2-Streams ter simplificado algumas instruções da YAAM, nomeadamente as instruções `unify` e `get`, elas no seu conjunto continuam a ser bastante complexas, não permitindo que um grande número de optimizações se possa realizar. Nesse sentido foi criada a linguagem intermédia YAIL.

  A YAIL é composta por instruções simples e complexas. As instruções simples são instruções que seguem uma estrutura semelhante às instruções das arquitecturas *RISC*. As instruções complexas são instruções usuais nas implementações em Prolog como a desreferenciação, a unificação, e outras do tipo.

  A YAIL foi desenhada de forma a permitir um grande número de optimizações, incluindo a reordenação de instruções.

- **Optimização do código YAIL**. Para melhorar o código YAIL são realizadas repetidamente oito transformações até que não se verifiquem mais alterações no código. Estas transformações incluem entre outras a eliminação de código morto, redução de saltos, eliminação de atribuições não usadas, e reordenação de algumas instruções.

- **Transformação do código YAIL em assembly da Alpha**. Para se poder transformar o código YAIL em código assembly da Alpha foi necessário desenvolver um método para fazer a atribuição dos registos da YAIL para os registos disponíveis na arquitectura Alpha. Foi também desenvolvido um método para evitar que as instruções que são pontos de entrada (`label`) fossem colocadas em endereços múltiplos de 8 para evitar problemas de alinhamento.

Foi necessário também desenvolver um método para estabelecer a ligação entre o emulador do YAP e o código resultante da compilação do código nativo. Para isso foi necessário:

- **Criar uma nova instrução para o emulador**. Esta nova instrução, designada por `Run_Native`, veio substituir o código YAAM das cláusulas, e a sua função ao ser executada, é passar o controle para o código nativo.

- **Alterar as instruções do emulador que fazem o salto para a próxima instrução**. Foi necessário alterar as instruções da máquina abstracta que fazem o salto para a próxima instrução. Isto porque a próxima instrução no Native YAP, tanto pode ser na máquina abstracta como pode ser no código nativo.

- **Criar um esquema para identificar o início das cláusulas**. Ao gerar várias cláusulas num ficheiro com o código nativo, tornou-se necessário criar um esquema capaz de identificar o início de cada cláusula e os possíveis pontos de entrada.

Os resultados da comparação entre o YAP e o Native YAP, mostraram que mesmo gerando código nativo apenas para as cláusulas, o Native YAP conseguiu obter uma aceleração entre 1.17 e 1.91, para o pior e melhor caso respectivamente, com uma média na aceleração superior a 1.5. Demonstrou-se também que essa aceleração não atingia valores mais elevados porque a execução de uma aplicação Prolog ainda passa em média 50% do seu tempo no emulador.

Foi criada uma versão idêntica ao Native YAP, que em vez de gerar código nativo gera código *C*. Ao comparar estas duas versões, ficou saliente mais uma vez, que a geração de código *C* não permite obter um desempenho tão elevado como o uso directo de assembly.

6.2 Trabalho Futuro

Nesta secção apresentam-se algumas ideias sobre uma possível continuação de trabalho na linha do que já foi realizado:

- **Expansão da geração do código nativo para os predicados de indexação.** A versão Native YAP desenvolvida suporta apenas a geração de código nativo para as cláusulas. A indexação e os predicados que fazem a ligação das cláusulas ainda são executados no emulador. O alargamento da geração do código nativo de forma a incluir todo o código para os predicados, pode trazer uma boa melhoria na performance do sistema, pois dessa forma, os saltos realizados entre cláusulas podem ser efectuados directamente evitando uma ida ao emulador.

- **Implementação de análise global ao código Prolog.** Com o desenho e a implementação da YAIL, uma linguagem intermédia com uma granularidade mais reduzida do que a WAM, foi dado o primeiro passo para que o YAP possa, à semelhança do que acontece no Aquarius, fazer uma análise global do código Prolog, explorar o determinismo e especializar a unificação.

- **Refinamento do conjunto de instruções das linguagens intermédias.** Para que a YAIL possa tirar todo o proveito de uma análise global de um programa Prolog, torna-se necessário que haja um modo de passar informação entre as diversas fases de compilação. Um desses métodos passa pela criação de novas instruções "*pragmas*", não executáveis, servindo apenas para guardar informação sobre o código. Desta forma as fases de optimização podem usar essa informação para conseguir obter melhor código.

6.3 Nota de Fecho

A principal contribuição desta tese foi a implementação de código nativo no sistema YAP. Para atingir este objectivo, foram criados novos passos para a compilação de Prolog, com destaque para a criação de uma nova linguagem intermédia para compilação avançada de Prolog, a YAIL.

Os bons resultados obtidos constituem uma forte motivação para uma possível continuação do trabalho realizado.

Referências

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

[Alp92] DECChip 21064 - AA RISC Microprocessor Preliminary Data Sheet. Technical note, Digital Equipment Corporation, 1992.

[And91] Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, June 1991.

[AVA86] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[Bha96] Dileep Bhandarkar. *Alpha Implementes and Architecture Complete Reference and Guide.* Digital Press, 1996.

[Car87] Mats Carlsson. Internals of Sicstus Prolog version 0.6. Internal Report, Gigalips Project, November 1987.

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

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme–machine en francais. Technical report cri 72-18, Groupe Intelligence Artificielle, Université Aix-Marseille II, October 1973.

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

[DFB94] Oliver J. Sharp David F. Bacon, Susan L. Graham. Compiler transformations for high-performance computing. *ACM computing surveys*, 26(4):345–375, 1994.

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

[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.

[Hay94] R. C. Haygood. Native code compilation in SICStus Prolog. MIT Press, June 1994.

[Hwa93] Kai Hwang. *Advanced Computer Architecture Parallelism Scalability Programmability.* McGraw-Hill, 1993.

[Kog91] Peter M. Kogge. *The Architeture of Symbolic Computers.* McGraw-Hill, 1991.

[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.

[Qui86] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[SC88] Vitor Santos Costa. Implementação de Prolog. Provas de Aptidão Pedagógica e Capacidade Científica, Universidade do Porto, Dezembro 1988.

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

[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. 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.

Apêndice A

Os Programas de teste

ham.pl

```
%    HAMILTONIAN GRAPHS
%    about 60 solutions
go:- ham(_),fail.
go.
ham(X):-
        cycle_ham([a,b,c,d,e,f,g,h,i,j,k,
                   l,m,n,o,p,q,r,s,t],X),
        write(X), nl.
cycle_ham([X|Y],[X,T|L]):-
        chain_ham([X|Y],[],[T|L]),
        edge(T,X).
chain_ham([X],L,[X|L]).
chain_ham([X|Y],K,L):-
        delete(Z,Y,T),
        edge(X,Z),
        chain_ham([Z|T],[X|K],L).
delete(X,[X|Y],Y).
delete(X,[U|Y],[U|Z]):-
        delete(X,Y,Z).
edge(X,Y):-
        connect(X,L),
        el(Y,L).
el(X,[X|L]).
el(X,[Y|L]):- el(X,L).
connect(0,[1,2,3,4,5,6,7,8,9]).
connect(1,[0,2,3,4,5,6,7,8,9]).
connect(2,[0,1,3,4,5,6,7,8,9]).
connect(3,[0,1,2,4,5,6,7,8,9]).
connect(4,[0,1,2,3,5,6,7,8,9]).
connect(5,[0,1,2,3,4,6,7,8,9]).
connect(6,[0,1,2,3,4,5,7,8,9]).
connect(7,[0,1,2,3,4,5,6,8,9]).
connect(8,[0,1,2,3,4,5,6,7,9]).
connect(9,[0,1,2,3,4,5,6,7,8]).
connect(a,[b,j,k]).
connect(b,[a,c,p]).
connect(c,[b,d,l]).
connect(d,[c,e,q]).
connect(e,[d,f,m]).
connect(f,[e,g,r]).
connect(g,[f,h,n]).
connect(h,[i,g,s]).
connect(i,[j,h,o]).
connect(j,[a,i,t]).
connect(k,[o,l,a]).
connect(l,[k,m,c]).
connect(m,[l,n,e]).
connect(n,[m,o,g]).
connect(o,[n,k,i]).
connect(p,[b,q,t]).
connect(q,[p,r,d]).
connect(r,[q,s,f]).
connect(s,[r,t,h]).
connect(t,[p,s,j]).
```

houses.pl

```
%Version : "coCOKE 1.1 mod"
%          from September 1987
go:- go_hs(_).
%% memb/2
%%-------
% all, unordered, eager
memb(X2,[X1|X0]):- memb(X2,X0).
memb(X1,[X1|X0]).
%% testp/4
%%--------
```

```
% all, unordered, eager
testp(X3,X2,[X3|X1],[X2|X0]).
testp(X5,X4,[X3|X2],[X1|X0]) :-
        testp(X5,X4,X2,X0).
%% on_right/4
%%----------
% all, unordered, eager
on_right(X4,X3,[X4|X2],[X1,X3|X0]).
on_right( X5,X4,[X3|X2],[X1|X0]) :-
        on_right(X5,X4,X2,X0).
%% on_left/4
%%---------
% all, unordered, eager
on_left(X4,X3,[X2,X4|X1],[X3|X0]).
on_left(X5,X4,[X3|X2],[X1|X0]) :-
        on_left(X5,X4,X2,X0).
%% beside/4
%%--------
% all, unordered, eager
beside(X3,X2,X1,X0) :-
        on_right(X3,X2,X1,X0).
beside(X3,X2,X1,X0) :-
        on_left(X3,X2,X1,X0).
%% go3/1
%%------
% all, ordered, lazy
go_hs(config(X27,X26,X25,X24,X23)) :-
        X27=[norway,X22,X21,X20,X19],
        X26=[X18,X17,X16,X15,X14],
        X25=[X13,X12,X11,X10,X9],
        X24=[X8,X7,milk,X6,X5],
        X23=[X4,X3,X2,X1,X0],
        testp(england,red,X27,X26),
        testp(japan,craven,X27,X23),
        beside(norway,blue,X27,X26),
        on_right(white,green,X26,X26),
        testp(yellow,kool,X26,X23),
        testp(spain,dog,X27,X25),
        testp(old_gold,snail,X23,X25),
        testp(gitane,wine,X23,X24),
        testp(ukr,tea,X27,X24),
        testp(green,coffe,X26,X24),
        beside(chesterfield,fox,X23,X25),
        beside(kool,horse,X23,X25),
        memb(zebra,X25),
        memb(water,X24).
```

magic.pl

```
%    resolve o problema
%    do cubo mágico
%    pelo método da força
%    bruta
%
%    Ricardo Rocha, 1996
%
go4(S):- problem4(Y),
         solve(4,Y,S).
problem4([ [2,1,1,5,1,1,5,2,6],
           [2,4,6,2,6,6,3,1,3],
           [3,3,6,3,3,6,5,4,6],
           [1,5,5,1,5,4,1,3,4],
           [4,2,2,2,2,3,3,5,5],
           [2,6,4,5,4,4,1,6,4] ]).
movimento(1,0).
movimento(2,1).
movimento(3,2).
movimento(4,3).
movimento(5,4).
movimento(6,5).
movimento(7,6).
solve(_,[ [1,1,1,1,1,1,1,1,1],
  [2,2,2,2,2,2,2,2,2],
  [3,3,3,3,3,3,3,3,3],
  [4,4,4,4,4,4,4,4,4],
  [5,5,5,5,5,5,5,5,5],
  [6,6,6,6,6,6,6,6,6] ],[]).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [dir1|L]):-
  movimento(N,N1),
  solve(N1,[
  [L41,L42,L43,L14,L15,L16,L17,L18,L19],
  [L11,L12,L13,L24,L25,L26,L27,L28,L29],
  [L21,L22,L23,L34,L35,L36,L37,L38,L39],
  [L31,L32,L33,L44,L45,L46,L47,L48,L49],
  [L53,L56,L59,L52,L55,L58,L51,L54,L57],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69] ],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [dir2|L]):-
  movimento(N,N1),
  solve(N1,[
  [L11,L12,L13,L44,L45,L46,L17,L18,L19],
  [L21,L22,L23,L14,L15,L16,L27,L28,L29],
  [L31,L32,L33,L24,L25,L26,L37,L38,L39],
  [L41,L42,L43,L34,L35,L36,L47,L48,L49],
  [L53,L56,L59,L52,L55,L58,L51,L54,L57],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [dir3|L]):-
  movimento(N,N1),
  solve(N1,[
```

```
  [L11,L12,L13,L14,L15,L16,L47,L48,L49],
  [L21,L22,L23,L24,L25,L26,L17,L18,L19],
  [L31,L32,L33,L34,L35,L36,L27,L28,L29],
  [L41,L42,L43,L44,L45,L46,L37,L38,L39],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L63,L66,L69,L62,L65,L68,L61,L64,L67] ],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [lat1|L]):-
  movimento(N,N1),
  solve(N1,[
  [L67,L12,L13,L64,L15,L16,L61,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L57,L34,L35,L54,L37,L38,L51],
  [L43,L46,L49,L42,L45,L48,L41,L44,L47],
  [L11,L52,L53,L14,L55,L56,L17,L58,L59],
  [L33,L62,L63,L36,L65,L66,L39,L68,L69] ],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [lat2|L]):-
  movimento(N,N1),
  solve(N1,[
  [L11,L68,L13,L14,L65,L16,L17,L62,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L58,L33,L34,L55,L36,L37,L52,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L12,L53,L54,L15,L56,L57,L18,L59],
  [L61,L32,L63,L64,L35,L66,L67,L38,L69] ],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [lat3|L]):-
  movimento(N,N1),
  solve(N1,[
  [L11,L12,L69,L14,L15,L66,L17,L18,L63],
  [L27,L24,L21,L28,L25,L22,L29,L26,L23],
  [L59,L32,L33,L56,L35,L36,L53,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L13,L54,L55,L16,L57,L58,L19],
  [L61,L62,L31,L64,L65,L34,L67,L68,L37] ],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [frn1|L]):-
  movimento(N,N1),
  solve(N1,[
  [L17,L14,L11,L18,L15,L12,L19,L16,L13],
  [L57,L22,L23,L58,L25,L26,L59,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L67,L44,L45,L68,L47,L48,L69],
  [L51,L52,L53,L54,L55,L56,L49,L46,L43],
  [L61,L62,L63,L64,L65,L66,L27,L24,L21] ],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [frn2|L]):-
  movimento(N,N1),
  solve(N1,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L54,L23,L24,L55,L26,L27,L56,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L64,L43,L44,L65,L46,L47,L66,L49],
  [L51,L52,L53,L48,L45,L42,L57,L58,L59],
  [L61,L62,L63,L28,L25,L22,L67,L68,L69] ],L).
solve(N,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L23,L24,L25,L26,L27,L28,L29],
  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
  [L51,L52,L53,L54,L55,L56,L57,L58,L59],
  [L61,L62,L63,L64,L65,L66,L67,L68,L69]
  ], [frn3|L]):-
  movimento(N,N1),
  solve(N1,[
  [L11,L12,L13,L14,L15,L16,L17,L18,L19],
  [L21,L22,L51,L24,L25,L52,L27,L28,L53],
  [L33,L36,L39,L32,L35,L38,L31,L34,L37],
  [L61,L42,L43,L62,L45,L46,L63,L48,L49],
  [L47,L44,L41,L54,L55,L56,L57,L58,L59],
  [L29,L26,L23,L64,L65,L66,L67,L68,L69] ],L).
```

mu.pl

```
% generated: 9 November 1989
% option(s):
%
%   mu
%
%   derived from Douglas R. Hofstadter,
%   "Godel, Escher, Bach," pages 33-35.
%

%   prove "mu-math" theorem
go :- theorem([m,u,i,i,u], 5, _), !.
theorem([m,i], _, [[a|[m,i]]]).
theorem(R, Depth, [[N|R]|P]) :-
    Depth > 0,
    D is Depth-1,
    theorem(S, D, P),
    rule(N, S, R).
```

```
rule(1, S, R) :- rule1(S, R).
rule(2, S, R) :- rule2(S, R).
rule(3, S, R) :- rule3(S, R).
rule(4, S, R) :- rule4(S, R).
rule1([i], [i,u]).
rule1([H|X], [H|Y]) :-
    rule1(X, Y).
rule2([m|X], [m|Y]) :-
    append(X, X, Y).
rule3([i,i,i|X], [u|X]).
rule3([H|X], [H|Y]) :-
    rule3(X, Y).
rule4([u,u|X], X).
rule4([H|X], [H|Y]) :-
    rule4(X, Y).
append([], X, X).
append([A|B], X, [A|B1]) :-
    append(B, X, B1).
```

nreverse.pl

```
% generated: 25 October 1989
% option(s):
%
%   nreverse
%   David H. D. Warren
%
%   "naive"-reverse a list
%    of 30 integers
go:- nreverse([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],_).
nreverse([X|L0],L):-
            nreverse(L0,L1),
            concatenate(L1,[X],L).
nreverse([],[]).
concatenate([X|L1],L2,[X|L3]) :-
         concatenate(L1,L2,L3).
concatenate([],L,L).
```

nsort.pl

```
%  NAIVE - SORT program
%  (from Delgado-Rannauro's Thesis)
%
%  The program sorts a list by
%  brute force! The list is in
%  the worst case (reverse-order).
go :- nsort([7,8,6,5,4,3,2,1],_).
nsort(L1,L2) :-
    nsort_permutation(L1,L2),
    nsort_sorted(L2).
nsort_sorted([X,Y|Z]) :-
    X =< Y,
    nsort_sorted([Y|Z]).
nsort_sorted([_]).
nsort_permutation([],[]).
nsort_permutation(L,[H|T]):-
    nsort_delete(H,L,R),
    nsort_permutation(R,T).
nsort_delete(X,[X|T],T).
nsort_delete(X,[Y|T],[Y|T1]) :-
        nsort_delete(X,T,T1).
```

poly_10.pl

```
%   Ralph Haygood
%
%   raise a polynomial (1+x+y+z)
%   to the 10th power (symbolically)
:-op(700,xfx,less_than).
% test polynomial definition
go(_) :-
    poly_add(poly(x,[term(0,1),term(1,1)]),
                    poly(y,[term(1,1)]),Q),
    poly_add(poly(z,[term(1,1)]),Q,P).
% 'less_than'/2 for x, y, z
less_than(x,y).
less_than(y,z).
less_than(x,z).
% polynomial addition
poly_add(poly(Var,Terms1),
  poly(Var,Terms2),poly(Var,Terms)):- !,
  term_add(Terms1, Terms2, Terms).
poly_add(poly(Var1,Terms1),
  poly(Var2,Terms2), poly(Var1,Terms)):-
  less_than(Var1,Var2), !,
  add_to_order_zero_term(Terms1,
  poly(Var2,Terms2), Terms).
poly_add(Poly, poly(Var,Terms2),
  poly(Var,Terms)) :- !,
  add_to_order_zero_term(Terms2,Poly,Terms).
poly_add(poly(Var,Terms1), C,
  poly(Var,Terms)) :- !,
  add_to_order_zero_term(Terms1,C,Terms).
poly_add(C1,C2,C):- C is C1+C2.
% term addition
term_add([], X, X) :- !.
term_add(X, [], X) :- !.
```

```
term_add([term(E,C1)|Terms1],
  [term(E,C2)|Terms2],[term(E,C)|Terms])
  :- !, poly_add(C1, C2, C),
  term_add(Terms1,Terms2,Terms).
term_add([term(E1,C1)|Terms1],
  [term(E2,C2)|Terms2],[term(E1,C1)|Terms])
  :- E1 < E2, !,
  term_add(Terms1,[term(E2,C2)|Terms2],Terms).
term_add(Terms1,
  [term(E2,C2)|Terms2],[term(E2,C2)|Terms])
  :- term_add(Terms1, Terms2, Terms).
add_to_order_zero_term([term(0,C1)|Terms],
  C2, [term(0,C)|Terms]):- !,
  poly_add(C1, C2, C).
add_to_order_zero_term(Terms,C,
  [term(0,C)|Terms]).
% polynomial exponentiation
poly_exp(0, _, 1) :- !.
poly_exp(N, Poly, Result) :-
  M is N>>1,
  N is M<<1, !,
  poly_exp(M, Poly, Part),
  poly_mul(Part, Part, Result).
poly_exp(N, Poly, Result) :-
  M is N-1,
  poly_exp(M, Poly, Part),
  poly_mul(Poly, Part, Result).
% polynomial multiplication
poly_mul(poly(Var,Terms1),
  poly(Var,Terms2), poly(Var,Terms)) :- !,
  term_mul(Terms1, Terms2, Terms).
poly_mul(poly(Var1,Terms1),
  poly(Var2,Terms2), poly(Var1,Terms)) :-
  less_than(Var1,Var2), !,
  mul_through(Terms1,poly(Var2,Terms2),Terms).
poly_mul(P,poly(Var,Terms2),poly(Var,Terms))
  :- !,mul_through(Terms2, P, Terms).
poly_mul(poly(Var,Terms1),C,poly(Var,Terms))
  :- !,mul_through(Terms1,C,Terms).
poly_mul(C1, C2, C) :- C is C1*C2.
term_mul([], _, []) :- !.
term_mul(_, [], []) :- !.
term_mul([Term|Terms1], Terms2, Terms):-
  single_term_mul(Terms2, Term, PartA),
  term_mul(Terms1, Terms2, PartB),
  term_add(PartA, PartB, Terms).
single_term_mul([], _, []) :- !.
single_term_mul([term(E1,C1)|Terms1],
  term(E2,C2), [term(E,C)|Terms]) :-
  E is E1+E2,
  poly_mul(C1, C2, C),
  single_term_mul(Terms1,term(E2,C2),Terms).
mul_through([], _, []) :- !.
mul_through([term(E,Term)|Terms],
  Poly, [term(E,NewTerm)|NewTerms]) :-
  poly_mul(Term, Poly, NewTerm),
  mul_through(Terms, Poly, NewTerms).
```

qsort.pl

```
% generated: 16 November 1989
% option(s): SOURCE_TRANSFORM_1
%
%   qsort
%
%   David H. D. Warren
%
%   quicksort a list of 50 integers
go :- 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]) :-
      partition(L,Y,L1,L2).
partition([],_,[],[]).
```

queens.pl

```
%    Solve the queens problem.
go:- q8,q9,q10.
q8:- q([1,2,3,4,5,6,7,8],C),
     write(C), nl.
q9:- q([1,2,3,4,5,6,7,8,9],C),
     write(C), nl.
q10:-q([1,2,3,4,5,6,7,8,9,10],C),
     write(C), nl.
q(L,C):-
     perm(L,P),
     pair(L,P,C),
     safe([],C).
perm([],[]).
perm(Xs,[Z|Zs]):-
     select(Z,Xs,Ys),
     perm(Ys,Zs).
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-
     select(X,Ys,Zs).
pair([],[],[]).
pair([X|Y],[U|V],[p(X,U)|W]):- pair(Y,V,W).
safe(X,[]).
```

```
safe(X,[Q|R]):-
     test(X,Q),
     safe([Q|X],R).
test([],X).
test([R|S],Q):-
     test(S,Q),
          nd(R,Q).
nd(p(C1,R1),p(C2,R2)):-
     C  is C1-C2,
     R  is R1-R2, C=\=R,
     NR is R2-R1, C=\=NR.
```

queens_8.pl

```
% generated: 10 November 1989
% option(s):
%
%   (queens) queens_8
%
%   from Sterling and Shapiro,
%   "The Art of Prolog," page 211.
%
go:- queens(8,_).
queens(N,Qs) :-
      range(1,N,Ns),
      queens(Ns,[],Qs).
queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :-
      select(UnplacedQs,UnplacedQs1,Q),
      not_attack(SafeQs,Q),
      queens(UnplacedQs1,[Q|SafeQs],Qs).
not_attack(Xs,X) :-
      not_attack(Xs,X,1).
not_attack([],_,_) :- !.
not_attack([Y|Ys],X,N) :-
      X =\= Y+N, X =\= Y-N,
      N1 is N+1,
      not_attack(Ys,X,N1).
select([X|Xs],Xs,X).
select([Y|Ys],[Y|Zs],X) :-
      select(Ys,Zs,X).
range(N,N,[N]) :- !.
range(M,N,[M|Ns]) :-
      M < N,
      M1 is M+1,
      range(M1,N,Ns).
```

query.pl

```
% generated: 17 November 1989
% option(s): SOURCE_TRANSFORM_1
%
%   query
%
%   David H. D. Warren
%
%   query population and area
%   database to find countries
%   of approximately equal
%   population density
go:- query(_).
query([C1,D1,C2,D2]) :-
    density(C1,D1),
    density(C2,D2),
    D1 > D2,
    T1 is 20*D1,
    T2 is 21*D2,
    T1 < T2.
density(C,D) :-
    pop(C,P),
    area(C,A),
    D is (P*100)//A.
% populations in 100000's
pop(china,8250).
pop(india,5863).
pop(ussr,2521).
pop(usa,2119).
pop(indonesia,1276).
pop(japan,1097).
pop(brazil,1042).
pop(bangladesh, 750).
pop(pakistan, 682).
pop(w_germany, 620).
pop(nigeria, 613).
pop(mexico, 581).
pop(uk, 559).
pop(italy, 554).
pop(france, 525).
pop(philippines, 415).
pop(thailand, 410).
pop(turkey, 383).
pop(egypt, 364).
pop(spain, 352).
pop(poland, 337).
pop(s_korea, 335).
pop(iran, 320).
pop(ethiopia, 272).
pop(argentina, 251).
% areas in 1000's of square miles
area(china,     3380).
area(india,     1139).
area(ussr,      8708).
area(usa,       3609).
area(indonesia,  570).
area(japan,      148).
area(brazil,    3288).
area(bangladesh,  55).
area(pakistan,   311).
area(w_germany,   96).
```

```
area(nigeria,    373).
area(mexico,     764).
area(uk,          86).
area(italy,      116).
area(france,     213).
area(philippines, 90).
area(thailand,   200).
area(turkey,     296).
area(egypt,      386).
area(spain,      190).
area(poland,     121).
area(s_korea,     37).
area(iran,       628).
area(ethiopia,   350).
area(argentina, 1080).
```

tak.pl

```
% generated: 17 November 1989
% option(s): SOURCE_TRANSFORM_1
%
%   tak
%
%   Evan Tick
%   (from Lisp version by R. P. Gabriel)
%   (almost) Takeuchi function
%   (recursive arithmetic)
go:- tak(18,12,6,U),
tak(X,Y,Z,A):-
        X =< Y,
        Z = A.
tak(X,Y,Z,A):-
        X > Y,
        X1 is X - 1,
        tak(X1,Y,Z,A1),
        Y1 is Y - 1,
        tak(Y1,Z,X,A2),
        Z1 is Z - 1,
        tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).
```

zebra.pl

```
% Where does the zebra live?
% Puzzle solution written by Claude Sammut.
go:- houses(Houses),
     member(house(red, english, _, _, _), Houses),
     member(house(_, spanish, dog, _, _), Houses),
     member(house(green, _, _, coffee, _), Houses),
     member(house(_, ukrainian, _, tea, _), Houses),
     right_of(house(green,_,_,_,_), house(ivory,_,_,_,_), Houses),
     member(house(_, _, snails, _, winstons), Houses),
     member(house(yellow, _, _, _, kools), Houses),
     Houses = [_, _, house(_, _, _, milk, _), _,_],
     Houses = [house(_, norwegian, _, _, _)|_],
     next_to(house(_,_,_,_,chesterfields), house(_,_,fox,_,_), Houses),
     next_to(house(_,_,_,_,kools), house(_,_,horse,_,_), Houses),
     member(house(_, _, _, orange_juice, lucky_strikes), Houses),
     member(house(_, japanese, _, _, parliaments), Houses),
     next_to(house(_,norwegian,_,_,_), house(blue,_,_,_,_), Houses),
     member(house(_, _, zebra, _, _), Houses),
     member(house(_, _, _, water, _), Houses),
     print_houses(Houses).
houses([house(_, _, _, _, _),
     house(_, _, _, _, _),
     house(_, _, _, _, _),
     house(_, _, _, _, _),
     house(_, _, _, _, _)]).
right_of(A,B,[B,A|_]).
right_of(A,B,[_|Y]) :-
     right_of(A,B,Y).
next_to(A,B,[A,B|_]).
next_to(A,B,[B,A|_]).
next_to(A,B,[_|Y]) :-
     next_to(A,B,Y).
member(X,[X|_]).
member(X,[_|Y]) :-
     member(X,Y).
```

Apêndice B

Fases de compilação de um programa Prolog

Por razões de espaço apresentam-se os códigos das fases de compilação de um programa Prolog no Native YAP, apenas para uma cláusula do programa de teste `nreverse`.

```
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

Código YAAM

```
name concatenate/3,3
get_list   A1
unify_var  X4
unify_var  X1
get_list   A3
unify_val  X4
unify_var  X3
execute primitives:concatenate/3
```

Código YAAM 2-Streams

```
name concatenate/3,3
deref_read      A1,label1
label2:
get_list_read   A1
unify_var_read  X4
unify_var_read  X1
deref_read      A3,label3
label4:
get_list_read   A3
unify_val_read  X4
unify_var_read  X3
execute primitives:concatenate/3
write_code
deref_write     A1,label2
label1:
get_list_write  A1
unify_var_write X4
unify_var_write X1
deref_write     A3,label4
label3:
get_list_write  A3
unify_val_write X4
unify_var_write X3
execute primitives:concatenate/3
```

Código YAIL já optimizado

```
name        concatenate/3,3
deref       R11,R15
set         R10,R0,0
isvar       R13,R11
test        NEQ,R13,label1
ispair      R13,R11
failz       R13
reppair     R1,R11
load        R18,R1,0
load        R15,R1,8
set         R1,R1,16
deref       R11,R17
set         R10,R0,0
isvar       R13,R11
test        NEQ,R13,label3
label4:
ispair      R13,R11
failz       R13
reppair     R1,R11
deref       R11,R1
deref       R12,R18
isvar       R13,R12
test        EQ,R13,label5
isvar       R13,R11
test        EQ,R13,label6
unify       R12,R11
load        R17,R1,8
set         R1,R1,16
execute primitives:concatenate/3
label6:
store       R11,R12,0
trail       R12
load        R17,R1,8
set         R1,R1,16
execute primitives:concatenate/3
label5:
isvar       R13,R11
test        EQ,R13,label7
store       R12,R11,0
trailglobal R11
load        R17,R1,8
set         R1,R1,16
execute primitives:concatenate/3
label7:
unifybound  R11,R12
load        R17,R1,8
set         R1,R1,16
execute primitives:concatenate/3
label1:
set         R1,R2,0
abspair     R12,R1
store       R12,R11,0
set         R2,R2,16
trail       R11
store       R1,R1,0
set         R18,R1,0
set         R1,R1,8
store       R1,R1,0
set         R15,R1,0
set         R1,R1,8
deref       R11,R17
isvar       R13,R11
test        EQ,R13,label4
label3:
set         R1,R2,0
abspair     R12,R1
store       R12,R11,0
set         R2,R2,16
trail       R11
store       R18,R1,0
set         R1,R1,8
store       R1,R1,0
set         R17,R1,0
set         R1,R1,8
execute primitives:concatenate/3
```

Código assembly para a Alpha

```
.verstamp 3 11 2 7 2
.set noreorder
.set volatile
.set noat
.file 1 "code.c"
gcc2_compiled.:
__gnu_compiled_c:
.data
.align 3
labels.2:
.quad $34
.quad $35
.rdata
.quad 0
.align 3
$C32:
.ascii "erro"
.text
.align 3
.globl funcao
.ent funcao
funcao:
ldgp $29,0($27)
funcao..ng:
lda $30,-32($30)
.frame $15,32,$26,0
stq $26,0($30)
.mask 0x4000000,-32
bis $30,$30,$15
.prologue 1
stq $16,24($15)
ldl $3,24($15)
stl $3,16($15)
$123454321:
lda $2,labels.2
s8addq $16,$2,$1
ldq $2,0($1)
jmp $31,($2),0
$36:
$123456789:
lda $0,0
bis $15,$15,$30
ldq $26,0($30)
addq $30,32,$30
ret $31,($26),1
$35:
bis $31,$31,$20
```

```
ldq $22,40 ($9)
bis $22,$22,$19
blt $19,$555555556
$555555557:
bis $19,$19,$23
ldq $19,0 ($23)
cmpeq $19, $23, $23
bne $23,$10001
bgt $19,$555555557
$555555556:
$10002:
blbc $19,$123456789
lda $1,-64
srl $1,4,$1
and $19,$1,$11
ldq $21,0 ($11)
stq $21,64 ($9)
lda $11,8 ($11)
ldq $21,0 ($11)
stq $21,40 ($9)
lda $11,8 ($11)
bis $31,$31,$20
ldq $22,56 ($9)
bis $22,$22,$19
blt $19,$555555558
$555555559:
bis $19,$19,$23
ldq $19,0 ($23)
cmpeq $19, $23, $23
bne $23,$10003
bgt $19,$555555559
$555555558:
$10004:
blbc $19,$123456789
lda $1,-64
srl $1,4,$1
and $19,$1,$11
ldq $22,64 ($9)
bis $22,$22,$18
blt $18,$10005
$555555560:
bis $18,$18,$23
ldq $18,0 ($23)
blt $18,$10005
cmpeq $18, $23, $23
beq $23,$555555560
bis $11,$11,$19
blt $19,$10006
$555555561:
bis $19,$19,$23
ldq $19,0 ($23)
blt $19,$10006
cmpeq $19, $23, $23
beq $23,$555555561
bis $18,$18,$16
bis $19,$19,$17
jsr $26,UnifyCells
ldgp $29,0($26)
br $31,$10008
cpys $f31,$f31,$f31
$10006:
stq $19,0 ($18)
ldq $21,16($9)
cmpule $18,$21,$22
beq $22,$555555562
ldq $21,24($9)
cmpult $18,$21,$22
beq $22,$555555563
$555555562:
subq $14,8,$14
stq $18,0($14)
$555555563:
br $31,$10008
$10005:
bis $11,$11,$19
blt $19,$10007
$555555564:
bis $19,$19,$23
ldq $19,0 ($23)
blt $19,$10007
cmpeq $19, $23, $23
beq $23,$555555564
stq $18,0 ($19)
ldq $21,24($9)
cmpult $19,$21,$22
beq $22,$555555565
subq $14,8,$14
stq $19,0($14)
$555555565:
br $31,$10008
$10007:
bis $19,$19,$16
bis $18,$18,$17
jsr $26,IUnify
ldgp $29,0($26)
beq $0,$123456789
$10008:
lda $11,8 ($11)
ldq $21,0 ($11)
stq $21,56 ($9)
lda $11,8 ($11)
ldq $21,16($9)
stq $21,-24($13)
bis $15,$15,$30
lda $11,3632
ldah $11,513($11)
addq $11,8,$2
zapnot $2,15,$1
ldq $15,0($1)
lda $0,1
ldq $26,0($30)
addq $30,32,$30
ret $31,($26),1
ldq $22,40 ($9)
bis $22,$22,$19
blt $19,$10002
$555555566:
bis $19,$19,$23
ldq $19,0 ($23)
blt $19,$10002
cmpeq $19, $23, $23
beq $23,$555555566
$10001:
bis $10,$10,$11
lda $10,16 ($10)
lda $1,-4
srl $1,1,$1
```

```
ornot $31,$1,$1
bis $11,$1,$18
stq $18,0 ($19)
ldq $21,16($9)
cmpule $19,$21,$22
beq $22,$555555567
ldq $21,24($9)
cmpult $19,$21,$22
beq $22,$555555568
$555555567:
subq $14,8,$14
stq $19,0($14)
$555555568:
stq $11,0 ($11)
stq $11,64 ($9)
lda $11,8 ($11)
stq $11,0 ($11)
stq $11,40 ($9)
lda $11,8 ($11)
ldq $22,56 ($9)
bis $22,$22,$19
blt $19,$10004
$555555569:
bis $19,$19,$23
ldq $19,0 ($23)
blt $19,$10004
cmpeq $19, $23, $23
beq $23,$555555569
$10003:
bis $10,$10,$11
lda $10,16 ($10)
lda $1,-4
srl $1,1,$1
ornot $31,$1,$1
bis $11,$1,$18
stq $18,0 ($19)
ldq $21,16($9)
cmpule $19,$21,$22
beq $22,$555555570
ldq $21,24($9)
cmpult $19,$21,$22
beq $22,$555555571
$555555570:
subq $14,8,$14
stq $19,0($14)
$555555571:
ldq $22,64 ($9)
stq $22,0 ($11)
lda $11,8 ($11)
stq $11,0 ($11)
stq $11,56 ($9)
lda $11,8 ($11)
ldq $21,16($9)
stq $21,-24($13)
bis $15,$15,$30
lda $11,3632
ldah $11,513($11)
addq $11,8,$2
zapnot $2,15,$1
ldq $15,0($1)
lda $0,1
ldq $26,0($30)
addq $30,32,$30
ret $31,($26),1
$34:
lda $16,$C32
jsr $26,printf
ldgp $29,0($26)
bis $31,1,$16
jsr $26,exit
ldgp $29,0($26)
$33:
bis $15,$15,$30
ldq $26,0($30)
addq $30,32,$30
ret $31,($26),1
.end funcao
```