José Manuel Gonçalves Silva Vieira

Controle de Fluxo em Prolog por Utilização de Primitivas de Suspensão

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Julho de 2010

José Manuel Gonçalves Silva Vieira

Controle de Fluxo em Prolog por Utilização de Primitivas de Suspensão

Dissertação submetida à Faculdade de Ciências da Universidade do Porto para a obtenção do grau de Mestre em Engenharia de Redes e Sistemas Informáticos

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Julho de 2010

Para a minha família.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, ao meu orientador, o Professor Ricardo Jorge Gomes Lopes da Rocha, por, durante todo este processo, ter estado sempre presente e me ter motivado e ajudado sempre que necessitei. As suas críticas sempre construtivas, sugestões e palavras de incentivo foram uma enorme ajuda para que esta tese se tornasse realidade. Muito obrigado por tudo.

Em segundo lugar aos meus companheiros de projecto, João Santos, Flávio Cruz, João Raimundo e Miguel Areias, pelo excelente ambiente de trabalho existente na sala e pela ajuda que, sempre que necessitei, me deram. Desejo-vos a melhor sorte para o vosso futuro pessoal e profissional.

Finalmente, queria agradecer à minha família e amigos que durante estes últimos meses nunca deixaram de me apoiar e foram sempre compreensivos comigo durante todo este processo. A todos vocês o meu muito obrigado.

Abstract

Tabling is an implementation technique that overcomes some of the well-known limitations of Prolog systems in dealing with infinite loops in programs that are syntactically correct. The tabling technique consists of storing intermediate answers for subgoals in a proper space, called *the table space*, so that they can be reused when a repeated subgoal appears. This technique can be implemented at low-level engine or it can be implemented at an high-level, using the Prolog language itself, by applying program transformation to add to the original program a set of control primitives that can be also implemented in Prolog.

In this thesis we discuss the problems, advantages and dissadvantages of using mechanisms to suspend and resume the computation at the Prolog level. These mechanisms can be used and are well suited for every kind of problems and algorithms that explore a search space. We will focus our attention to the problem of implementing the tabling technique at an high-level by using this mecanisms but, in shuch a way, that the execution model is very close to the execution model of the low-level engine.

Our study showed that the high-level tabling module has slightly worse execution times when compared to the YapTab system. These execution times were expected. However, the fact that our implementation reduce the complexity of the implementation, makes it quite attractive to experiment and try different execution strategies and/or new extensions for tabled evaluation.

Resumo

A *tabulação* é a técnica que melhor consegue resolver as limitações bem conhecidas dos sistemas Prolog, mais concretamente no que diz respeito a programas que, apesar de estarem sintacticamente correctos, entram em ciclo infinito. Esta técnica consiste em ir guardando as soluções intermédias encontradas para os subgolos num espaço próprio, a *tabela*, para que estas possam ser reutilizadas quando surgirem chamadas repetidas desses subgolos. A implementação desta técnica pode ser realizada a baixo nível, isto é, ao nível da máquina de execução, ou a alto nível, isto é, ao nível do próprio Prolog por utilização de um programa de transformação que adiciona ao programa original um conjunto de directivas de controle.

Nesta tese pretendemos abordar o problema de utilizar mecanismos que permitam suspender e recuperar a computação ao nível do próprio Prolog. Estes mecanismos podem ser utilizados em todos os problemas e algoritmos que exploram um espaço de procura. No caso concreto desta tese, iremos focar a nossa atenção no problema de implementar a técnica da tabulação a alto nível com recurso a estes mecanismos, mas de modo a que o modelo de execução seja muito próximo da implementação de baixo nível.

O estudo realizado demonstrou que a utilização do módulo desenvolvido, que implementa a técnica da tabulação a alto nível, apresenta, como seria de esperar, tempos de execução ligeiramente piores quando comparado com o sistema YapTab. Contudo, o facto da nossa abordagem reduzir consideravelmente a complexidade da implementação da técnica da tabulação, torna-a bastante atractiva para a experimentação de diferentes estratégias de execução e/ou novas extensões ao modelo base de execução.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A Programação em Lógica [12] é um paradigma de programação baseado nas Cláusulas Lógicas de Horn, um subconjunto da lógica de primeira ordem, que conduz a um código simples de programar, entender e transformar. A utilização da lógica como linguagem de programação permite que o programador concentre a sua atenção na especificação do problema que pretende resolver.

Actualmente, a linguagem de Programação em Lógica mais popular é o Prolog. A popularidade do Prolog deve-se, em grande parte, a David H. D. Warren. No ano de 1983, David Warren desenhou uma máquina abstracta para a execução de código Prolog compilado, a *Warren´s Abstract Machine* ou simplesmente WAM [31], que se tornou o standard para implementar compiladores em Prolog. Devido aos enormes avanços realizados na tecnologia de compilação, os compiladores de Prolog são capazes de executar programas com uma velocidade muito semelhante à de linguagens de programação imperativa como, por exemplo, a linguagem C [21]. Com a inclusão de alguns predicados que não estão presentes na lógica de primeira ordem como, por exemplo, predicados para efectuar operações de input/output, o Prolog tornou-se viável na resolução de problemas reais. Esta linguagem tem vindo a demonstrar, ao longo dos anos, todo o seu potencial em áreas como a Inteligência Artificial, o Processamento de Linguagem Natural, a Gestão de Base de Dados e Sistemas Inteligentes, entre outras.

Os sistemas Prolog são sensíveis à ordem pela qual as cláusulas de um predicado aparecem devido ao facto de utilizarem a resolução *Selective Linear Definite* (SLD) [12] para escolherem o subgolo que deve ser executado. Consoante essa ordem, a forma como a árvore de execução é percorrida e a ordem pela qual as soluções são encontradas

difere levando, em alguns casos, programas sintacticamente correctos do ponto de vista lógico a entrarem em ciclo infinito.

As limitações da resolução SLD são bem conhecidas de tal forma que, ao longo dos anos, têm sido propostas inúmeras técnicas para solucionar esses problemas. Uma das técnicas com melhores resultados é a *tabulação* [13] que, de forma sumária, consiste em ir guardando as soluções intermédias encontradas para os subgolos num espaço próprio, a *tabela*, para que estas possam ser reutilizadas quando surgirem chamadas repetidas desses subgolos. Com esta técnica para além de se conseguir evitar os ciclos infinitos, consegue-se ainda reduzir a memória utilizada e têm-se melhores propriedades de terminação comparativamente à resolução SLD [3].

A implementação da técnica da tabulação é realizada maioritariamente a baixo nível, mais concretamente ao nível da máquina de execução. Existem inúmeros sistemas que implementam a técnica da tabulação dessa forma como, por exemplo, o XSB Prolog [23, 16] e o YapTab [19]. Alguns desses sistemas diferem entre si no modelo de execução que utilizam. Actualmente, os principais modelos de execução são a *tabulação por suspensão da computação* e a *tabulação linear*. Na *tabulação por suspensão*, a execução é vista como uma sequência de sub-computações que podem ser suspensas e, mais tarde, recuperadas até se atingir um ponto fixo. Na *tabulação linear*, a execução é vista como uma única computação onde os subgolos tabelados são, de forma recursiva, reavaliados até se atingir um ponto fixo.

O sistema YapTab, por exemplo, implementa tabulação por suspensão da computação e, para isso, utiliza uma estratégia de congelamento das pilhas de execução. Este sistema, que tem como base o modelo de execução da SLG-WAM [22], introduz algumas alterações/extensões ao modelo da WAM. Uma dessas alterações/extensões é a utilização de uma nova estrutura de dados, a tabela, onde são guardados os subgolos tabelados e as respectivas soluções. A tabela é implementada com recurso a uma estrutura de dados compacta denominada *Tries* [14]. As *tries*, criadas inicialmente com o intuito de indexar dicionários [9], são uma estrutura de dados do tipo árvore muito compacta e que torna a inserção e a procura de termos bastante eficiente.

Outra forma de implementar a tabulação é a alto nível, mais concretamente ao nível do próprio Prolog por utilização de um programa de transformação que vai adicionar ao programa original um conjunto de directivas de controle. A abordagem original desta técnica da tabulação foi feita por Ramesh e Chen [15] em que o programa de transformação é escrito em Prolog e as directivas são implementadas na linguagem C com recurso ao interface para a linguagem C dos sistemas Prolog. Posteriormente,

surgiram outras propostas [24, 18, 4] mas todas elas tiveram como base a abordagem inicial de Ramesh e Chen.

1.1 Objectivo da Tese

Nesta tese pretendemos abordar o problema de utilizar mecanismos que permitam suspender e recuperar a computação ao nível do próprio Prolog. Como o Prolog tem uma ordem bem definida de atravessar o espaço de procura, através da resolução SLD, a utilização destes mecanismos permitirá implementar diferentes formas de explorar o espaço de procura. Em particular, todos os problemas e algoritmos que exploram um espaço de procura podem ser facilmente modulados para usar estes mecanismos de suspensão e recuperação da computação. No caso concreto desta tese, iremos focar a nossa atenção no problema de implementar a técnica da tabulação a alto nível seguindo, em parte, a ideia original de Ramesh e Chen, em que temos um programa de transformação e um módulo que implementa as directivas de controle, com recurso a estes mecanismos.

A nossa primeira abordagem, denominada *Chamada de Suspensão*, utiliza os predicados *built-in* do Yap Prolog, para guardar a informação necessária para retomar mais tarde uma computação suspensa. Apesar de esta abordagem ter demonstrado resultados positivos no problema de encontrar o caminho mais curto num grafo, tem algumas limitações no que diz respeito ao tipo de chamada de continuação a utilizar, como demonstrou Guzman et al [7], no caso de problemas onde é usada a técnica de tabulação implementada com recurso a mecanismos semelhantes. Quando os predicados tabelados são chamados a partir de predicados não tabelados (e vice-versa) corre-se o risco de se perder parte do ambiente de execução da chamada de continuação e assim, executar erradamente o programa transformado. No entanto, uma solução para este problema foi igualmente proposta por Guzman et al [7] que passa por utilizar um programa de transformação mais complexo que tenta manter todo o ambiente de execução associado à continuação da computação no momento de uma suspensão.

Uma outra abordagem, que evita aumentar a complexidade do programa de transformação, é implementar os mecanismos de suspensão e recuperação através de *Primitivas de Suspensão*. Nesta abordagem, onde também utilizamos predicados *built-in* do Yap Prolog, a suspensão e a recuperação da computação são efectuadas a baixo nível, mais concretamente, ao nível da máquina de execução. Esta abordagem demonstrou

resultados positivos tanto no problema de encontrar o caminho mais curto num grafo como em problemas que utilizam o módulo de tabulação.

O módulo desenvolvido com recurso às Primitivas de Suspensão permitiu-nos implementar a técnica de tabulação ao nível do próprio Prolog mas de modo a que o modelo de execução seja muito próximo da implementação de baixo nível. Esta abordagem é bastante interessante pois abre a possibilidade de, no futuro, experimentar novas extensões ao modelo de tabulação como, por exemplo, o suporte para a negação, mas sempre ao nível do próprio Prolog evitando assim a maior complexidade da implementação de baixo nível.

O nosso módulo, todo ele escrito na linguagem Prolog - programa de transformação e directivas da tabulação - suporta as duas estratégias de escalonamento com maior êxito actualmente - *local scheduling* e *batched scheduling*. O programa de transformação altera o programa do utilizador para um formato reconhecido pelo nosso módulo o que torna a utilização do módulo completamente transparente para o utilizador. De realçar ainda que o programa transformado só altera os predicados tabelados logo, o desempenho da execução de programas não tabelados não é afectado.

Como estruturas de suporte, utilizamos as *tries* para guardar os subgolos tabelados e as respectivas soluções, e um conjunto de estruturas de dados auxiliares para guardar a informação relativa à computação. Para implementar, ao nível do Prolog, o conjunto de estruturas de dados auxiliares fizemos três abordagens distintas: utilizando a base de dados interna do Yap Prolog; utilizando predicados dinâmicos; e utilizando vectores em conjunto com a base de dados interna do Yap Prolog ou com predicados dinâmicos.

1.2 Estrutura da Tese

Iremos em seguida fazer um pequeno sumário dos seis capítulos desta tese.

Capítulo 1: Introdução. O capítulo actual.

Capítulo 2: Programação em Lógica e Tabulação: Faz uma breve introdução à Programação em Lógica e à técnica da tabulação. No que diz respeito à Programação em Lógica vamo-nos focar na linguagem Prolog e na máquina de execução WAM. Relativamente à técnica da tabulação, abordamos um pouco mais em detalhe o sistema YapTab e a estrutura de dados Tries.

Capítulo 3: Mecanismos de Suspensão e Recuperação. Introduz as duas formas distintas de implementar mecanismos de suspensão e recuperação da computação com recurso a predicados *built-in* do Yap Prolog. Começa por abordar o problema sobre o qual nos baseamos e os detalhes da implementação de uma estrutura geral/principal onde vão ser, posteriormente, incluídos os mecanismos. Em seguida, aborda as estruturas de dados utilizadas e, finalmente, os detalhes da implementação de cada uma das versões.

Capítulo 4: Tabulação com Primitivas de Suspensão. Inicialmente, aborda os factores que nos motivaram a desenvolver este módulo. Depois, descreve todos os passos da implementação do módulo. Começa por descrever a versão base que reproduz, de forma minimalista, o sistema YapTab e, posteriormente descreve as diversas optimizações efectuadas: consumir soluções de forma incremental (*incremental answer resolution*); completar computações de forma incremental (*incremental completion*); utilização de outra estratégia de escalonamento (*batched scheduling*); e utilização de estruturas de dados alternativas para guardar a informação envolvida na computação.

Capítulo 5: Resultados Experimentais. Mostra um estudo detalhado do desempenho do módulo desenvolvido. Começamos por descrever, de forma sumária, os diferentes programas utilizados para realizar o estudo. Em seguida, mostramos e discutimos os resultados obtidos, mais concretamente os tempos de execução, pelas diferentes versões em alguns desses programas. Utilizando a versão com melhores resultados comparamos o nosso módulo com o sistema YapTab. O estudo dessa comparação permitiu-nos verificar, entre outras coisas, o custo real da utilização do módulo.

Capítulo 6: Conclusões e Trabalho Futuro. Sumaria o trabalho desenvolvido, destaca as principais contribuições do mesmo e sugere algumas ideias que poderão ser abordadas numa futura extensão ao trabalho desenvolvido.

Capítulo 2

Programação em Lógica e Tabulação

Este capítulo tem como principal objectivo dar uma visão geral das áreas de investigação abrangidas por esta tese, focando os aspectos mais relevantes de cada área. Iremos abordar a Programação em Lógica, com maior incidência na Warren´s Abstract Machine e no Prolog, e a técnica da tabulação, onde iremos focar o sistema YapTab e a estrutura de dados *Tries*.

2.1 Programação em Lógica

A utilização da lógica como linguagem de programação, denominada por Programação em Lógica [12], permite que o programador apenas se concentre no problema, deixando a sua resolução para o computador.

A Programação em Lógica é um paradigma de programação baseado em um subconjunto da lógica de primeira ordem denominado Cláusulas Lógicas de Horn. As Cláusulas Lógicas de Horn podem ser de três tipos:

Regras: a forma habitual em Prolog é

A :- B1, ..., Bn.

Uma regra é composta por uma cabeça (A) e por um corpo que pode ter um ou mais literais (cada Bi é um subgolo). A cabeça está separada do corpo pelo

símbolo ‘*:-*’ que pode ser lido como um “*se*”. Se houver mais do que um subgolo no corpo estes encontram-se separados pelo símbolo ‘,’ que pode ser lido como um “*e*”. Numa regra a cabeça só é verdade se o corpo for verdade.

Factos: a representação em Prolog é

A.

Um facto é uma regra em que o corpo é vazio. Neste tipo de cláusula a cabeça é sempre verdade.

Consultas: são representadas em Prolog por

:- B1, ..., Bn.

As consultas (*querys*) são regras em que a cabeça é vazia.

Cada literal ou subgolo (como, por exemplo, A) que compõe uma cláusula tem a forma

p(t1,...,tn)

onde p é o nome do predicado e todos os $ti\acute{}s$ são os termos usados como argumentos. Os predicados podem ser definidos por regras e/ou factos enquanto um termo pode ser:

- uma variável (cujo nome tem que começar obrigatoriamente por letra maiúscula);
- uma constante (cujo nome tem que começar obrigatoriamente por letra minúscula);
- um termo composto, da forma

  f(u1,...,un)

  onde f é um functor de aridade n e cada ui é também um termo. Habitualmente é representado por f/n onde f é o nome do functor e n é a aridade.

As variáveis só podem ser instanciadas uma única vez e permanecem sem um tipo definido até serem instanciadas. A instanciação ocorre via unificação, uma operação de *pattern matching*, que para dois termos tenta encontrar a instância mais comum.

A Programação em Lógica é um demonstrador de teoremas simples que, dada uma teoria (ou programa) e uma consulta, utiliza a teoria para procurar diferentes formas de satisfazer a consulta. A Programação em Lógica tem vantagens que levam a um código compacto que é fácil de programar, entender e transformar. Segundo Carlsson [2] essas vantagens são:

Semântica declarativa simples: um programa é uma colecção de cláusulas lógicas;

Semântica processual simples: um programa pode ser lido como um conjunto de procedimentos recursivos;

Elevado poder expressivo: os programas podem ser vistos como especificações executáveis que permitem implementar algoritmos complexos e eficientes;

Não determínismo inerente: como em geral várias cláusulas podem unificar com um golo, os problemas que envolvem pesquisa podem ser facilmente programados neste tipo de linguagens.

2.1.1 Warren´s Abstract Machine

O primeiro compilador de Prolog foi desenvolvido em 1977 por David H. D. Warren [30]. Este compilador tornou o Prolog mais viável e, por consequência, levou a um aumento do número de seguidores desta linguagem. Mais tarde, em 1983, David Warren desenhou uma nova máquina abstracta para a execução de código Prolog compilado que ficou conhecida como Warren´s Abstract Machine ou simplesmente WAM [31]. A WAM tornou-se o standard para implementar compiladores em Prolog.

Ao nível da arquitectura, a WAM está estruturada em diferentes áreas, como ilustra a Figura 2.1, com estruturas de dados simples e um conjunto de instruções de baixo nível. O estado da computação é obtido, a qualquer momento, pelo conteúdo que se encontra nas pilhas de execução e nos registos da WAM.

A WAM define as seguintes pilhas de execução:

PDL (ou *push down list*): área auxiliar utilizada pelo processo de unificação.

Trilha: guarda o endereço das variáveis da *stack* ou da *heap* que foram instanciadas e que devem ser recolocadas como variáveis quando a computação falha e retoma a execução num ponto anterior (designado por processo de *backtracking*). A trilha encontra-se organizada como um vector de endereços.

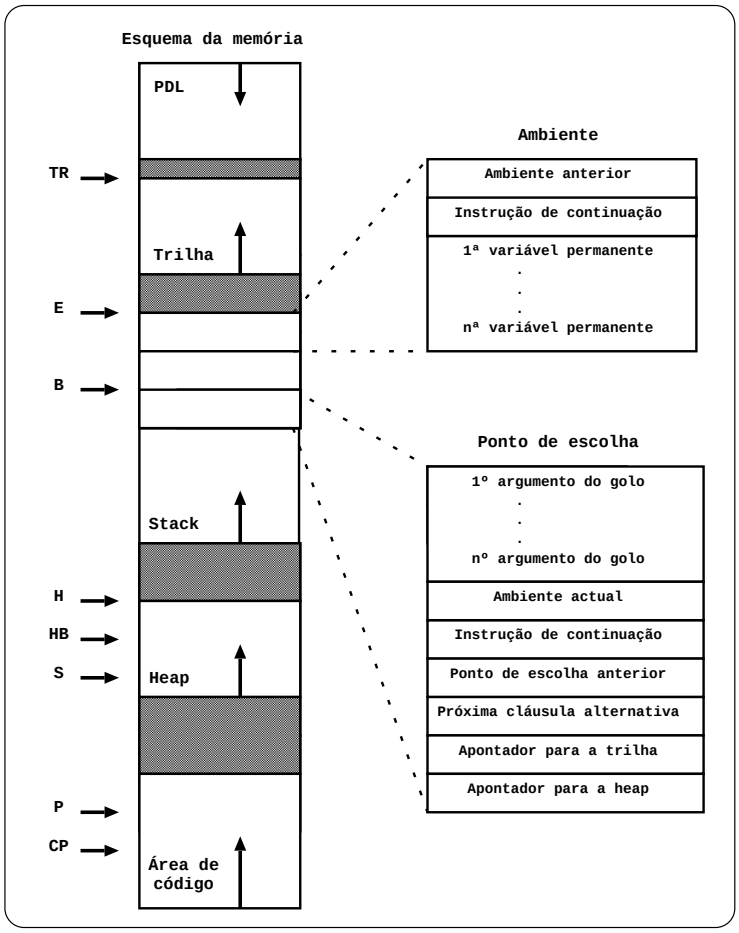

Figura 2.1: Arquitectura da WAM.

Stack (ou *local stack*): guarda as estruturas de dados utilizadas para representar os ambientes e os pontos de escolha. Um ambiente é colocado na pilha (*stack*) sempre que uma cláusula com vários subgolos é invocada para execução, e é retirado antes do último subgolo ser executado. Um ambiente guarda o endereço da *stack* referente ao ambiente anterior, que irá ser reposto caso seja retirado da pilha; o endereço da instrução de continuação, que será executado após o retorno com sucesso da cláusula invocada; e uma sequência de células que irão guardar as variáveis *permanentes* da cláusula invocada. As variáveis permanentes são variáveis que ocorrem em mais do que um subgolo da cláusula, logo as suas atribuições têm que ser mantidas para além da primeira vez que aparecem.

Um ponto de escolha é colocado na pilha quando um golo é chamado para execução e existe um conjunto de cláusulas candidatas, e é retirado quando a última cláusula desse conjunto for seleccionada para execução. Um ponto de escolha contém os dados necessários para restaurar o estado da computação para

o ponto anterior à criação do mesmo; e, para além disso, possui a referência para a próxima cláusula alternativa que deve ser testada caso a actual falhe.

Heap (ou *global stack*): é um vector de dados utilizado para guardar variáveis e termos compostos que não podem ser guardados na *stack*.

Área de código: contêm as instruções WAM referentes à forma compilada dos programas carregados.

As instruções da WAM estão organizadas em grupos. Desses diferentes grupos os mais importantes são:

Instruções de unificação: implementam versões específicas do algoritmo de unificação de acordo com o tipo e a posição dos argumentos. Existem instruções para efectuar a unificação da cabeça e dos sub-argumentos e instruções para preparar argumentos para subgolos. Estas instruções variam caso a variável ocorra pela primeira vez na cláusula, caso seja uma variável repetida, ou se aparecer na cláusula uma constante, uma lista ou um outro termo composto.

Instruções de controlo: têm a missão de alocar e remover ambientes e gerir as sequências *call*/*return* dos subgolos.

Instruções de pontos de escolha: são responsáveis por alocar e remover pontos de escolha e recuperar o estado da computação através dos dados guardados nos pontos de escolha.

Instruções de indexação: tornam o processo de determinar qual é a cláusula que unifica com a chamada (*call*) de um dado subgolo mais rápido.

Os registos mais importantes da WAM são o topo da trilha (TR); o ambiente actual (E); o ponto de escolha actual (B); o topo da heap (H); o apontador *backtrack* da heap (HB), que é usado para determinar se a instanciação de uma variável (*binding*) é condicional ou não; o apontador para o sub-termo a considerar (S), que é utilizado durante a unificação de termos compostos; o apontador para a instrução WAM que está a ser executada (P); e o apontador para onde retornar após a execução com sucesso da instrução actual (CP).

A WAM, apesar da sua aparente simplicidade, tem alguns detalhes na sua implementação bastante delicados. O tutorial de Ait-Kaci sobre a WAM [1], por exemplo, discute esses tópicos em detalhe.

2.1.2 Prolog

O Prolog, nome inventado por Colmerauer como abreviatura de *PROgramation en LOGic* [6], é considerada a linguagem de programação em lógica mais popular. Actualmente os compiladores de Prolog, devido aos enormes avanços efectuados na tecnologia de compilação nas últimas duas décadas, são capazes de executar programas quase com a mesma rapidez que linguagens de programação imperativas como o C [21]. Mas, para tornar o Prolog uma linguagem de programação para problemas reais, foi necessário introduzir alguns predicados que não estão na lógica de primeira ordem, tais como:

- Predicados para saber o estado da computação e para manipular termos (*meta-logical predicates*) como, por exemplo, os predicados *var/1* e *atom/1* que testam o estado dos argumentos;
- Predicados para efectuar operações de *input/output* e manipular a base de dados do Prolog (*extra-logical predicates*) como, por exemplo, os predicados *assert/1* e *retract/1* que adicionam e removem cláusulas da base de dados do Prolog, respectivamente;
- Predicados que efectuam operações de controlo, sendo o mais conhecido o predicado *cut* que permite descartar alternativas por explorar;
- Predicados que efectuam operações aritméticas e comparação de termos.

Para uma informação mais detalhada sobre o Prolog o leitor pode consultar livros sobre Prolog como [5, 12, 25].

Os sistemas Prolog utilizam a resolução *Selective Linear Definite* (SLD) [12]. Na resolução SLD o programa é percorrido até ser encontrada uma cláusula que unifique com um dos golos da consulta. Caso exista uma cláusula nessas condições, é gerada uma nova consulta composta pelo subgolos dessa cláusula (que unificou) e pelos restantes subgolos da consulta inicial. Este processo é aplicado de forma recursiva até ser gerada a consulta vazia. Se a unificação falhar, a execução volta atrás e tenta encontrar uma outra hipótese que satisfaça a consulta anterior.

No caso concreto da linguagem Prolog, a resolução SLD é utilizada como uma função de selecção em que o subgolo mais à esquerda é sempre escolhido em primeiro. Se existirem várias alternativas que unifiquem com o subgolo seleccionado, utiliza-se a ordem pela qual as cláusulas estão no programa, isto é, é escolhida a cláusula que

aparece em primeiro lugar. Desta forma a árvore de pesquisa é percorrida de cima para baixo e da esquerda para a direita.

De forma a ilustrar o funcionamento do Prolog, tomemos como exemplo o predicado *path/2* da Figura 2.2 que define se existe um caminho de X para Z num grafo arbitrário representado por factos *edge/2*.

```
path(X,Z) :- edge(X,Y), path(Y,Z).
path(X,Z) :- edge(X,Z).

edge(1,2).
edge(2,3).
```

Figura 2.2: Programa em Prolog para encontrar caminhos num grafo.

A primeira cláusula do predicado *path/2* verifica se existe uma ligação de X para Y e, caso exista, chama, de forma recursiva, o predicado *path/2* com o primeiro argumento a passar a ser Y. O resultado final desta cláusula é o resultado da chamada recursiva. Na segunda cláusula verifica-se se existe uma ligação de X para Z e, se for verdade, então existe um caminho de X para Z.

A primeira cláusula do predicado *edge/2* define que existe uma ligação de 1 para 2 e a segunda cláusula define que existe uma ligação de 2 para 3.

Como foi referido anteriormente, os sistemas em Prolog são sensíveis à ordem pela qual as cláusulas de um predicado aparecem. Essa ordem pode influenciar a forma como a árvore de execução é percorrida e a ordem pela qual as soluções são encontradas. Tomemos como exemplo a execução da consulta `path(1,Z)` tal como demonstrado na Figura 2.3. No lado esquerdo temos a árvore de execução se considerarmos o programa da Figura 2.2 e no lado direito temos a sequência de execução do mesmo programa mas com a ordem das cláusulas do predicado *path/2* invertida.

Uma das limitações do Prolog é a possibilidade de existirem ciclos infinitos. Quando a recursão é à direita, como no caso das duas árvores de execução da Figura 2.3, a existência de ciclos é menos provável, mas se a recursão estiver à esquerda, o programa facilmente entra em ciclo infinito. Por exemplo, se no programa da Figura 2.2 alterarmos a recursividade da direita para a esquerda, e se efectuarmos novamente a consulta `path(1,Z)` vemos que o programa entra em ciclo infinito (Figura 2.4). Uma das formas de resolver este tipo de problema é recorrendo à técnica da tabulação.

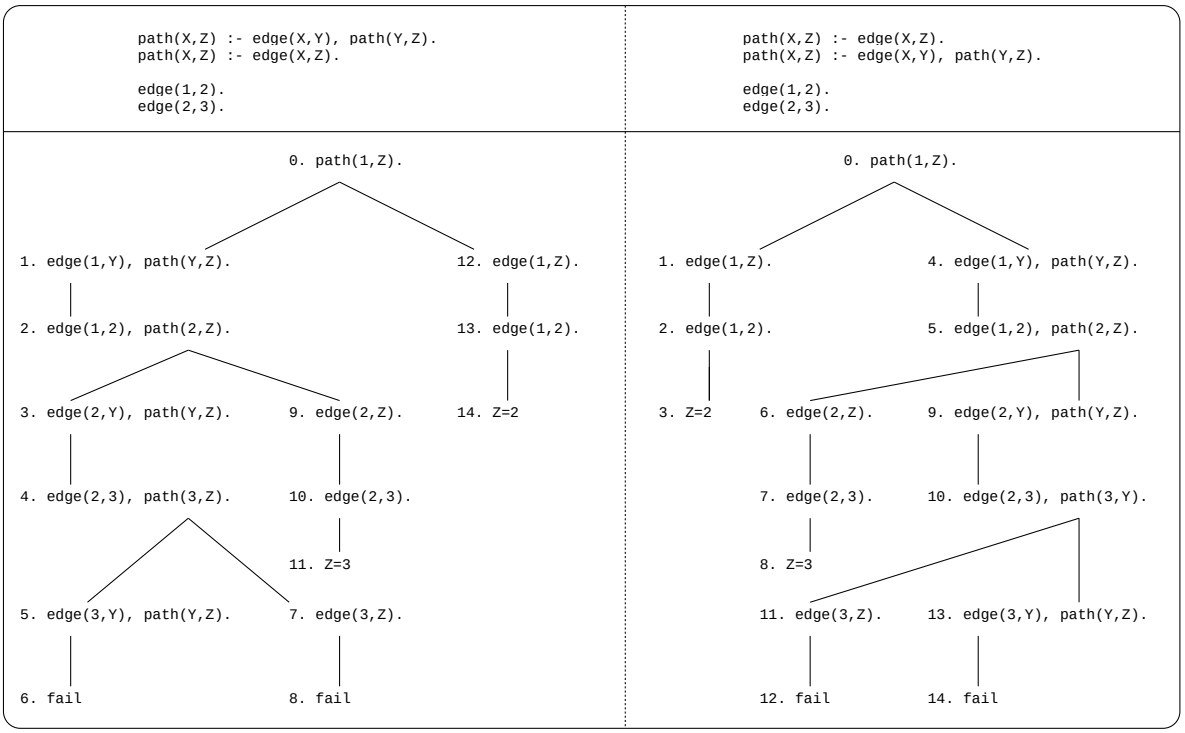

Figura 2.3: Árvores de execução da consulta `path(1,Z)` com recursividade à direita.

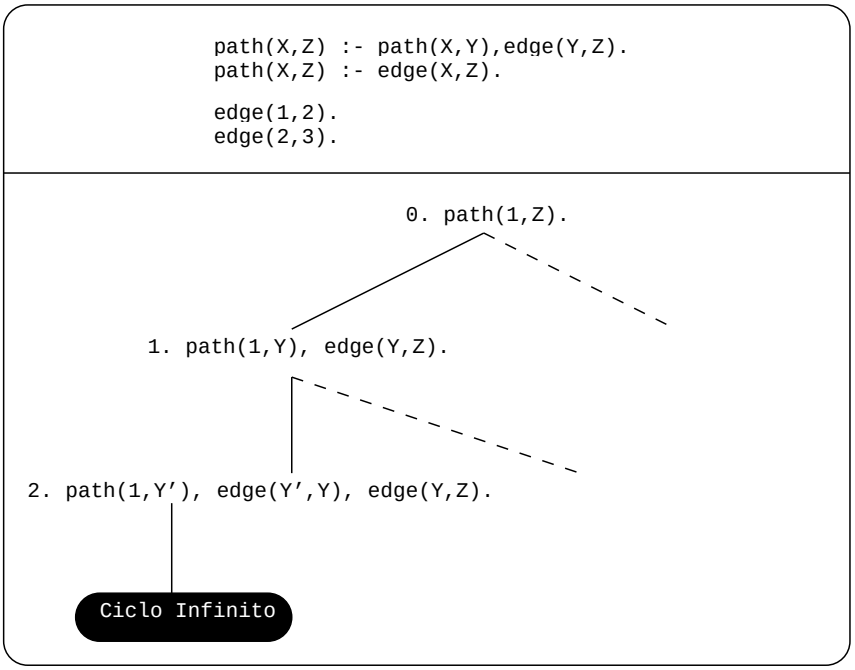

Figura 2.4: Árvore de execução da consulta `path(1,Z)` com recursividade à esquerda.

2.2 Tabulação

Desde o aparecimento da WAM que o Prolog tem vindo a demonstrar bons resultados. No entanto, a resolução SLD, utilizada pelo Prolog, sofre de fortes limitações quando estamos a lidar com computações redundantes e/ou quando estamos perante ciclos infinitos. Estas limitações são bem conhecidas e têm sido feitos bastantes esforços para tentar encontrar uma solução. Uma das técnicas propostas mais bem sucedidas para solucionar estes problemas é a tabulação [13] de tal modo que inicialmente foram propostos vários sistemas de tabulação. Os mais conhecidos são o OLDT [26], SLD-AL [28], Extension Tables [8], Backchain Iteration [29] e a resolução SLG [3].

A tabulação, de forma bastante sintetizada, consiste em ir guardando as soluções intermédias encontradas para os subgolos num espaço próprio, a *tabela*, de tal forma que estas possam ser reutilizadas quando surgirem chamadas repetidas desses subgolos. Desta forma consegue-se reduzir a memória utilizada, evitam-se ciclos infinitos e têm-se melhores propriedades de terminação relativamente à resolução SLD [3].

Tomemos como exemplo o programa da Figura 2.4 com recursão à esquerda. Se antes da definição do predicado *path/2* adicionarmos a declaração '*:- table path/2.*' estamos a dizer que o predicado *path/2* deve ser tabelado. A partir desse momento a tabulação vai ser aplicada para resolver as chamadas dos subgolos que constituem esse predicado. A Figura 2.5 ilustra a sequência de avaliação para a consulta `path(1,Z)` com tabulação.

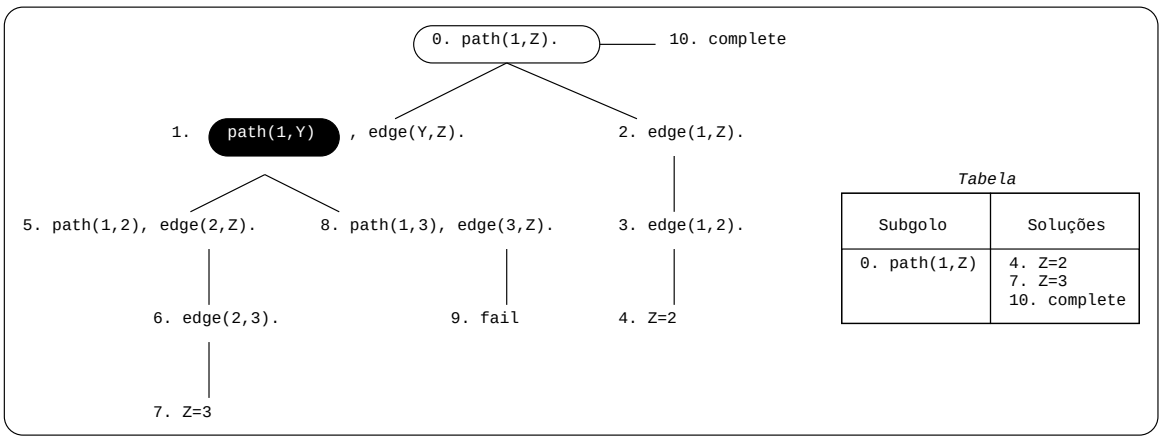

Figura 2.5: Árvore de execução da consulta `path(1,Z)` com tabulação.

Do lado esquerdo da figura temos a sequência de avaliação e do lado direito temos a tabela no final da avaliação. A avaliação começa por inserir uma nova entrada na tabela e alocar um nó gerador (representado por uma caixa branca) para representar

`path(1,Z)`. Depois, `path(1,Z)` unifica com a primeira cláusula do predicado *path/2* e chama `path(1,Y)` (passo 1). Como `path(1,Y)` é uma chamada variante relativamente a `path(1,Z)`, este é marcado como nó consumidor (representado por uma caixa preta). Tenta-se então consumir soluções da tabela mas, como ainda não existem soluções, a computação corrente é suspensa. Os nós consumidores devem ser suspensos porque podem ser encontradas novas soluções para essa chamada. Quando um nó consumidor é suspenso a computação passa, através de *backtracking*, para uma alternativa ainda por explorar. No caso concreto deste exemplo a alternativa em aberto é a segunda opção do nó 0 (passo 2).

O predicado *edge/2* como não é tabelado, é chamado de nó interior, e é resolvido normalmente. A primeira cláusula do predicado *edge/2* unifica com a chamada `edge(1,Z)` (passo 3) e a solução `Z=2` é encontrada e inserida na tabela (passo 4). Como não existem mais alternativas para explorar verificamos se a computação que havia sido suspensa pode ser retomada. A computação só pode ser retomada caso existam novas soluções para esse subgolo na tabela, como acontece com o nó 1 (passo 5). A computação continua com a chamada `edge(2,Z)` que unifica com a segunda cláusula desse predicado (passo 6) e uma nova solução, `Z=3`, é encontrada e inserida na tabela (passo 7). Devido a não existirem alternativas por explorar, retoma-se a computação que havia sido suspensa pois existe uma nova solução, `Z=3`, na tabela. A computação é retomada consumindo a nova solução da tabela (passo 8). A continuação da computação é agora feita com a chamada `edge(3,Z)` mas, como não existe nenhuma cláusula que unifique com a chamada, a computação falha (passo 9). Por fim, como não existem novas soluções e também não existem alternativas por explorar a computação é marcada como completa (passo 10).

Determinar se uma computação está completa pode ser bastante difícil, pois podem existir subgolos mutuamente dependentes e que, por consequência, formam um *Strongly Connected Component* (SCC) [27]. Por essa razão quando marcamos um subgolo de um SCC como completo temos que marcar todos os restantes. Habitualmente um SCC é representado pelo nó líder, isto é, o nó gerador mais novo que não depende de geradores mais antigos. O nó líder é também o nó mais antigo do seu SCC e define o ponto onde devemos tentar determinar se a computação do seu SCC está completa. No exemplo da Figura 2.5 o nó lider é o nó 0.

Uma das características principais deste modelo de execução é conseguir garantir que uma vasta classe de programas termine. Isto é bastante útil quando estamos a lidar com aplicações com predicados recursivos, como o predicado *path/2*, que podem levar a ciclos infinitos. Além disso, como nos modelos de tabulação um subgolo tabelado só

é computado uma vez, é possível não só reduzir a complexidade do programa como também a memória necessária à sua execução. Uma das formas que a tabulação tem para evitar computações desnecessárias e, em particular, ciclos infinitos é não guardar na tabela soluções duplicadas. Quando aparece uma solução duplicada a computação falha (de notar que esta situação não acontece no exemplo da Figura 2.5).

A decisão de continuar a execução (*forward execution*); ir por *backtracking* para um nó interior; consumir soluções em nós consumidores; ou completar um SCC é definida pela estratégia de escalonamento. A estratégia adoptada pode ter um impacto significativo no desempenho e pode influenciar a ordem das soluções. Na tabulação as estratégias com maior sucesso são a *batched scheduling* e a *local scheduling* [10].

A estratégia de *batched scheduling*, estratégia que foi adoptada no exemplo da Figura 2.5, dá prioridade à continuação da execução, seguindo-se o *backtracking* e, por fim, o consumir soluções ou completar. Esta estratégia, cuja execução é muito semelhante à da WAM, tem como principal objectivo reduzir a movimentação da computação pela árvore de procura.

A estratégia de *local scheduling*, cujo principal objectivo é completar SCCs o mais cedo possível, dá primazia ao *backtracking* e a completar os SCC, em seguida ao consumir de soluções e, finalmente, à continuação da execução. Em *local sheduling* quando surgem novas soluções a execução falha, ao contrário do *batched*, e quando um SCC é completo o líder do SCC age como um nó consumidor e começa a consumir as soluções, enquanto no *batched* quando um SCC é completo a execução falha.

Actualmente os principais modelos de execução da tabulação são a tabulação por *suspensão da computação* e a *tabulação linear*. Na tabulação por suspensão, a execução é vista com uma sequência de sub-computações que podem ser suspensas e mais tarde recuperadas até atingir um ponto fixo. Este é o modelo adoptado nos exemplos anteriores. A suspensão das sub-computações é conseguida através do congelamento das pilhas de execução, por cópia das pilhas de execução para áreas auxiliares, ou por combinação de ambas as técnicas. Os sistemas XSB Prolog [23, 16] e YapTab [19] utilizam suspensão por congelamento das pilhas de execução.

Na tabulação linear, a execução é vista como uma única computação onde os subgolos tabelados são, de forma recursiva, reavaliados até se atingir um ponto fixo sem que seja necessário suspender/recuperar sub-computações. A necessidade de utilizar re-computação para calcular pontos fixos pode ser considerado como um ponto fraco destes modelos. Os sistemas ALS-Prolog [11] e B-Prolog [32] utilizam tabulação linear.

2.2.1 O Sistema YapTab

O YapTab é um sistema de tabulação que, como já havia sido referido anteriormente, utiliza a suspensão por congelamento das pilhas de execução para implementar tabulação. O YapTab é baseado no modelo de execução da SLG-WAM [22] que, introduz as seguintes alterações/extensões ao modelo da WAM:

- Uma nova estrutura de dados, a tabela, onde são guardados os subgolos tabelados e as respectivas soluções. No YapTab, a tabela é implementada utilizando *Tries* [14], uma estrutura de dados compacta e que permite uma rápida procura e inserção de termos. Esta estrutura de dados torna o YapTab bastante eficiente.

- Uma extensão à trilha original, a *forward trail*, que permite restaurar as atribuições de uma variável para o estado anterior à suspensão da computação. Desta forma consegue-se retomar computações suspensas de forma bastante eficiente.

- Um novo conjunto de registos, os *freeze registers*, que permitem suspender e recuperar o estado da computação. O YapTab, que preserva o ambiente de uma computação suspensa através do congelamento das pilhas de execução, possui um conjunto de *freeze registers* para cada pilha de execução. Os *freeze registers*, responsáveis pela protecção do espaço das pilhas de uma computação suspensa, são actualizados quando a computação é suspensa e quando um SCC é marcado como completo.

- Quatro novas operações:

  1. *tabled subgoal call* - esta operação verifica se um subgolo está na tabela. Se não estiver adiciona-o e cria um novo nó gerador (nó 0 da Figura 2.5), caso contrário cria um nó consumidor e começa a consumir as soluções existentes na tabela (nó 1 da Figura 2.5).
  2. *new answer* - esta operação verifica se uma nova solução já faz parte da tabela. Se a solução existir a operação falha, senão insere-a na tabela (passos 4 e 7 na Figura 2.5).
  3. *answer resolution* - esta operação verifica, para um dado consumidor, se existem novas soluções para consumir. Se existirem consome (passos 5 e 8 da Figura 2.5), se não suspende a computação e, por *backtracking*, continua a execução numa alternativa por explorar (passo 2 da Figura 2.5).

As soluções são consumidas pela mesma ordem pela qual são inseridas na tabela.

4. *completion* - esta operação determina se um SCC está completamente avaliado. Uma tabela diz-se completa quando o conjunto de soluções guardadas na tabela representa todas as conclusões que podem ser tiradas do conjunto de factos e regras que constituem o programa (passo 10 da Figura 2.5). De outra forma a tabela está incompleta e a computação deve continuar a sua execução em alternativas que ainda não foram exploradas ou em consumidores com soluções por consumir.

As diferenças entre o YapTab e a SLG-WAM, tal como implementada no sistema XSB Prolog, encontram-se nas estruturas de dados e nos algoritmos utilizados para controlar o processo de detecção do nó líder e o escalonamento das soluções que ainda estão por consumir. Enquanto a SLG-WAM considera que o controlo deve ser efectuado ao nível das estruturas de dados correspondentes às primeiras chamadas dos subgolos tabelados, o YapTab considera que esse controlo deve estar a cargo das estruturas de dados correspondentes às chamadas repetidas dos subgolos tabelados. Para isso, o YapTab associa uma nova estrutura de dados aos nós consumidores que é utilizada para, de forma eficiente, percorrer os diversos nós consumidores que têm soluções para consumir ou determinar se um SCC está completo. Esta estrutura de dados é denominada por *dependency frame* [19].

O YapTab suporta uma estratégia dinâmica de avaliação de programas lógicos que junta o *batched scheduling* e o *local scheduling* [20]. Foi ainda o primeiro sistema de tabulação com suporte para tabulação incompleta [17], que permite evitar em algumas situações re-computação devido a utilizar as soluções previamente calculadas, e com suporte para a recuperação automática da memória da tabela [17], que coloca ao dispor do programador algoritmos eficazes de gestão de memória para escolher as tabelas que podem ser eliminadas quando existem problemas a este nível.

2.2.2 *Tries*

As *tries*, originalmente inventadas para indexar dicionários [9], são uma estrutura de dados do tipo árvore em que prefixos comuns são representados uma única vez. A ideia das *tries* é dividir um conjunto de termos `T` com base na sua estrutura para que a procura e a inserção desses termos seja bastante eficiente. A raiz da árvore representa o ponto de entrada e cada uma das folhas corresponde a um termo completo de `T`. Os

nós intermédios representam os símbolos dos termos. Vejamos como é fácil encontrar um termo através do exemplo da Figura 2.6.

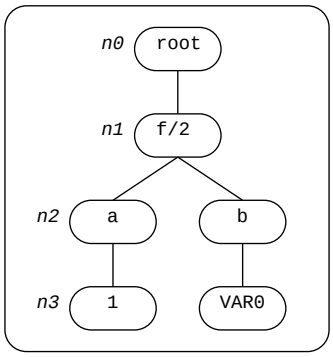

Figura 2.6: Representação na *trie* dos termos `f(a,1)` e `f(b,VAR0)`.

A Figura 2.6 apresenta uma *trie* onde estão representados os termos `f(a,1)` e `f(b,VAR0)`. Se quisermos procurar o termo `f(a,1)` começamos no nó raiz (*n0*) e, como só estamos a representar termos do tipo *f/2* na *trie*, avançamos para o próximo nó (*n1*). Neste nó, como temos duas opções, temos que verificar qual é o nó que corresponde à constante `a`. A opção da esquerda é a correcta (nó *n2*) e, em seguida, avançamos para o próximo nó (*n3*) onde a única opção existente corresponde à constante `1`.

Se quisermos adicionar um novo termo `g(X)` à *trie* temos que adicionar um novo nó para representar *g/1*, devido a este ser diferente de *f/2*, e um novo nó para representar a variável `X`. Cada variável é representada nas *tries* como uma constante distinta, isto é, para um conjunto de variáveis existentes em um termo t vai ser atribuído, da esquerda para a direita, uma constante do conjunto *(VAR0, VAR1, VAR2, ..., VARn)* a cada uma das variáveis. Por exemplo, no termo `f(g(X,Z),Y,X)` às variáveis `X`, `Z` e `Y` vão ser associadas as constantes *VAR0*, *VAR1* e *VAR2*, respectivamente.

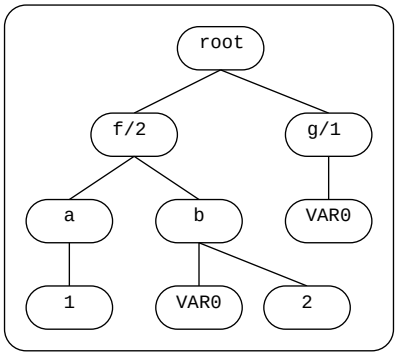

Figura 2.7: Adição dos termos `g(X)` e `f(b,2)` à *trie* da Figura 2.6.

Por outro lado, se quisermos adicionar o termo `f(b,2)`, como já existe na *trie* um nó para o functor *f/2* e um nó para a constante `b`, só precisamos de adicionar um nó para a constante `2` a partir do nó da constante `b`. A Figura 2.7 representa a *trie* anterior com a adição destes dois termos.

De forma a implementar predicados tabelados recorrendo às *tries* o YapTab utiliza dois níveis: a *subgoal trie* para guardar os subgolos tabelados; e a *answer trie* para guardar as soluções dos subgolos tabelados. Na *subgoal trie* cada *table entry* representa o ponto de entrada dos subgolos de um predicado. Na *answer trie* cada *subgoal frame* representa o ponto de entrada das soluções de um subgolo. A Figura 2.8 demonstra como a tabela do exemplo da Figura 2.5 pode ser estruturada utilizando *tries*.

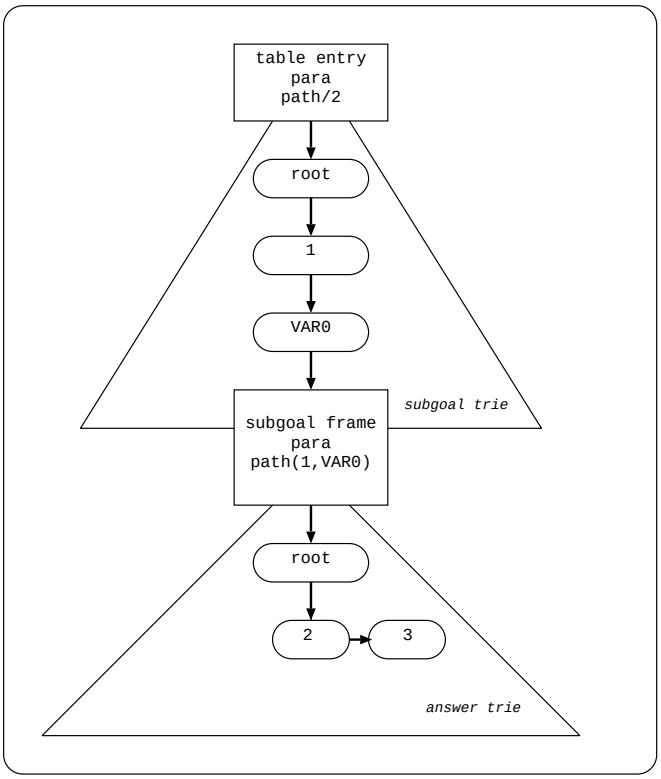

Figura 2.8: Utilizando *tries* para representar a tabela do exemplo da Figura 2.5.

De notar que na *answer trie* apenas são guardadas as substituições das variáveis existentes na chamada do subgolo correspondente (*substitution factoring*) [14]. Por exemplo, no caso da Figura 2.8 em vez de serem guardadas as soluções `path(1,2)` e `path(1,3)` apenas são guardadas as substituições `2` e `3` correspondentes à variável `X` na chamada `path(1,X)`.

As soluções são ainda mantidas numa lista pela ordem que foram encontradas. Os nós

consumidores, devido ao facto de guardarem a referência da última solução consumida, consomem novas soluções seguindo essa lista. No caso da Figura 2.8, por exemplo, primeiro é consumida a solução `2` e depois a solução `3`.

Para além do suporte para *tries* do YapTab, o sistema Yap possui uma biblioteca externa que permite manipular *tries*, independentemente da utilização da tabulação. As funções mais importantes dessa biblioteca são:

- *trie_open(T)* - inicia uma nova *trie* e devolve em T a sua referência;
- *trie_close(T)* - encerra a *trie* com a referência T.
- *trie_put_entry(T,E,R)* - adiciona o termo E à *trie* T e devolve em R a sua referência;
- *trie_check_entry(T,E,R)* - verifica se o termo E existe na *trie* T e, caso exista, devolve em R a sua referência;
- *trie_get_entry(R,E)* - devolve em E o termo representado pela referência R;
- *trie_remove_entry(R)* - remove o termo representado pela referência R.
- *trie_traverse(T,R)* - percorre a *trie* T a partir do primeiro termo e retorna em R a referência do próximo termo adicionado à *trie* T;
- *trie_traverse(T,R1,R2)* - percorre a *trie* T a partir do termo representado pela referência *R1* e retorna em *R2* a referência do próximo termo adicionado à *trie* T. De forma a isto ser possível, foram necessárias implementar duas novas funções:
  1. *trie_get_first_entry(T,R)* - retorna em R a referência do primeiro termo adicionado à *trie* T;
  2. *trie_get_last_entry(T,R)* - retorna em R a referência do último termo adicionado à *trie* T.

2.3 Sumário

Neste capítulo descrevemos os conceitos mais importantes da Programação em Lógica, da Warren´s Abstract Machine (WAM) e da linguagem de programação Prolog. De seguida abordamos a tabulação, que é uma das técnicas mais bem sucedidas para solucionar os problemas da resolução SLD utilizada pelo Prolog, e um pouco mais

em detalhe o sistema de tabulação YapTab. Por fim vimos as *tries*, uma estrutura de dados compacta e que permite uma rápida procura e inserção de termos, que é utilizada pelo YapTab para guardar os subgolos tabelados e as respectivas soluções.

No capítulo seguinte iremos abordar diferentes formas de implementar mecanismos de suspensão e recuperação da computação com recurso a predicados *buit-in* do Yap Prolog.

Capítulo 3

Mecanismos de Suspensão e Recuperação

Neste capítulo iremos abordar duas formas distintas de implementar mecanismos de suspensão e recuperação da computação com recurso a predicados *built-in* do Yap Prolog. Começaremos por abordar a ideia geral, incluindo o problema onde nos baseamos para implementar estas primitivas, e passaremos depois aos detalhes da sua implementação.

3.1 Ideia Geral

O principal objectivo a que nos propusemos com este trabalho foi desenvolver um módulo externo que implementasse a técnica da tabulação por transformação de programas e utilização de primitivas de suspensão (cujos detalhes da implementação irão ser abordados no próximo capítulo). Nesse sentido, o primeiro passo passou por desenvolver mecanismos que permitissem suspender e recuperar a computação. Esses mecanismos devem ser completamente transparentes para o utilizador, isto é, o utilizador deve apenas preocupar-se com o problema que pretende resolver e, caso necessite, deve invocar esses mecanismos sem ter a necessidade de proceder a qualquer tipo de implementação ou modificação.

De forma a tornar menos complexo o desenvolvimento dos mecanismos de suspensão e recuperação optamos por desenvolve-los no âmbito de um problema mais simples que, para além de nos garantir uma maior fiabilidade ao nível do funcionamento,

permite-nos também demonstrar que eles podem ser utilizados em outros problemas. Decidimos então utilizar o problema de encontrar o caminho mais curto num grafo. A escolha por este tipo de problema deveu-se ao facto de existirem diversos algoritmos de pesquisa para encontrar o caminho mais curto num grafo, o que nos permitiu experimentar diferentes estratégias de modo a verificar o correcto funcionamento dos mecanismos implementados.

Posto isto, procedemos ao desenvolvimento de uma estrutura geral/principal, tal como representada na Figura 3.1, composta pelas cláusulas que implementam o problema de encontrar o caminho mais curto num grafo, e invocam os mecanismos desenvolvidos.

```
principal_goal(X,Z,_):-
   search(X,Z,0,0).
principal_goal(X,Z,Distance):-
   %Get answer
   ...,
   %Clear data structures
   ....

search(X,Z,Distance,Min):-
   user_search(X,Z,Distance,Min).
search(_,_,_,_):-
   %Resume computation
   ....

user_search(X,X,Distance,_) :-
   !,
   make_answer(Distance),
   fail.
user_search(X,Z,PreviousDistance,PreviousMin):-
   edge(X,Y,Distance),
   CurrentDistance is PreviousDistance + Distance,
   %Search algorithm
   Min=...,
   %Suspend computation
   ....
```

Figura 3.1: Predicados que constituem a estrutura geral/principal para o problema de encontrar o caminho mais curto num grafo.

O predicado de topo da estrutura geral/principal é o predicado *principal_goal/3*, que é invocado pelo utilizador para realizar consultas. Este predicado é composto por duas cláusulas, e recebe como argumentos o nó de partida e o nó de chegada, e retorna a distância mínima entre os dois nós. A primeira cláusula é responsável por iniciar a pesquisa através da chamada ao predicado *search/4*. A segunda é responsável por retornar o caminho mínimo encontrado e limpar as estruturas de dados utilizadas durante a execução, tal como iremos ver mais à frente.

Dado estarmos a implementar o problema de encontrar o caminho mínimo num grafo através de mecanismos de suspensão, houve a necessidade de ter um predicado in-

termédio que fosse responsável por recuperar as computações suspensas. Esse predicado intermédio, que tem duas cláusulas, é o predicado *search/4*. A primeira cláusula chama o predicado *user_search/4* que vai explorar todo o espaço de procura da chamada correspondente. Quando isso acontecer, o predicado *search/4* executa a sua segunda cláusula. Aí, procede-se à recuperação da computação referente ao caminho mínimo encontrado pelo predicado *user_search/4*, dando origem a uma nova chamada ao predicado *search/4*. Este processo repete-se sucessivamente até atingirmos o nó de chegada, altura em que a computação regressa à segunda cláusula do predicado de topo *principal_goal/3*.

Por fim, temos o predicado onde o utilizador define o seu algoritmo de procura, denominado *user_search/4*. Este predicado, composto por duas cláusulas, tem como argumentos o nó corrente, o nó de chegada, a distância percorrida até aquele ponto e, mediante o algoritmo de pesquisa, como, por exemplo, no caso do algoritmo *Breadth-First*, este predicado recebe ainda como quarto argumento o valor mínimo até aquele momento. No caso específico deste algoritmo de pesquisa como a procura é realizada em largura, o valor mínimo corresponde ao nível onde o nó de chegada se encontra.

Se a computação unificar com a primeira cláusula do predicado *user_search/4*, isso significa que foi encontrado um caminho válido pois o nó corrente e de chegada são o mesmo. Nesse caso, a segunda cláusula não deve ser executada e para isso utilizamos o predicado *built-in !* (*cut*) que, como já foi referido, permite descartar alternativas por explorar. Quando é encontrado um caminho, temos que guardar a distância percorrida caso ainda não tenha sido encontrado nenhum caminho ou se esta distância for menor que a existente. Este controlo é efectuado pelo predicado *make_answer/1* cujo esboço podemos ver na Figura 3.2, e que iremos abordar em detalhe mais à frente.

```
make_answer(NewDistance):-
   (%Get current answer, if any
   ... ->
     CurrentDistance > NewDistance,
     %Current answer is worse than new answer, so remove it
     ...
   ;
     %First answer
     true
   ),
   %Store new answer
   ....
```

Figura 3.2: Esboço do predicado *make_answer/1*.

A segunda cláusula do predicado *user_search/4* é responsável pela procura de todos os caminhos directos existentes a partir do nó corrente. Para cada caminho actualiza

o custo (CurrentDistance) adicionando à distância já percorrida (PreviousDistance), o custo do caminho directo entre o nó corrente e um nó intermédio (Distance). De seguida, define o valor mínimo (Min), consoante o algoritmo de pesquisa (no caso do algoritmo de *Dijkstra* o valor mínimo é igual ao custo actualizado, e no caso do algortimo do algoritmo *Breadth-First* o valor mínimo é o nível onde o nó de chegada se encontra), e suspende a computação. É nesta cláusula que o utilizador define, de forma bastante simples, o algoritmo de pesquisa a utilizar como demonstra a Figura 3.3.Em cima temos o predicado *user_search/4* com o algoritmo de *Dijkstra* e em baixo com o algoritmo *Breadth-First*.

```
user_search(X,X,Distance,_) :-
   !,
   make_answer(Distance),
   fail.
user_search(X,Z,PreviousDistance,_):-
   edge(X,Y,Distance),
   CurrentDistance is PreviousDistance + Distance,
   %Dijkstra search algorithm
   Min=CurrentDistance,
   %Suspend computation
   ....
```

```
user_search(X,X,Distance,_) :-
   !,
   make_answer(Distance),
   fail.
user_search(X,Z,PreviousDistance,PreviousMin):-
   edge(X,Y,Distance),
   CurrentDistance is PreviousDistance + Distance,
   %BreathFirst search algorithm
   Min is PreviousMin + 1,
   %Suspend computation
   ....
```

Figura 3.3: Predicado *use_search/4* implementado com dois algoritmos de pesquisa diferentes: *Dijkstra* (em cima) e *Breadth-First* (em baixo).

Após o desenvolvimento desta estrutura geral/principal ficaram reunidas todas as condições para procedermos à implementação dos mecanismos de suspensão e recuperação da computação. Iremos, em seguida, abordar todos os detalhes referentes à implementação desses mecanismos.

3.2 Estruturas de Dados Auxiliares

Para além dos predicados *built-in* do Yap Prolog, os mecanismos de suspensão e recuperação da computação foram desenvolvidos com recurso a estruturas de dados

auxiliares para guardar a informação necessária à recuperação da computação. Para o problema de encontrar o caminho mais curto num grafo foram utilizadas as seguintes estruturas:

- *answer/1* - responsável por guardar a melhor solução encontrada (*Distance*);
- *waiting/3* - responsável por guardar a informação referente às computações suspensas, nomeadamente o valor mínimo (*Min*), a distância percorrida (*Distance*) e o caminho entre o nó corrente e o nó de chegada (*path(Y,Z)*). Na versão primitivas de suspensão, onde a suspensão é realizada ao nível da máquina de execução, temos ainda que guardar o ponto de escolha associado à computação suspensa (*CP*) - logo a estrutura passa a ter aridade *4*;
- *done/2* - responsável por guardar os caminhos que já foram visitados (*path(Y,Z)*) juntamente com a respectiva distância (*Distance*).

Estas estruturas de dados foram implementadas utilizando predicados dinâmicos. Este tipo de predicados permitem ao programador alterar a base de dados das cláusulas durante a execução do programa. Para definir um predicado P com aridade n como predicado dinâmico devemos utilizar a declaração *dynamic(P/n)*.

Se pretendermos adicionar uma cláusula C a um predicado dinâmico e não nos interessar o local onde é colocada, utilizamos o predicado *built-in assert(C)*. Caso nos interesse inserir a cláusula C no início do programa utilizamos o predicado *asserta(C)*. Se, por outro lado, nos interessar inserir no fim utilizamos o predicado *assertz(C)*.

Para remover uma cláusula C de um predicado dinâmico utilizamos o predicado *built-in retract(C)* mas, se quisermos remover todas as cláusulas que unifiquem com a cabeça de um golo G, utilizamos o predicado *retractall(G)* . Os predicados de inserção e remoção de cláusulas em predicados dinâmicos são, como já foi referido no capítulo 2 (secção 2.1.2), predicados de manipulação da base de dados do Prolog (*extra-logical predicates*).

As consultas aos predicados dinâmicos são efectuadas da mesma maneira que aos restantes predicados.

Por exemplo, para declarar a estrutura de dados auxiliar *answer/1* como predicado dinâmico utilizamos *dynamic(answer/1)*. Para inserir uma nova solução efectuamos um *assert(answer(Distance))* e para remover realizamos um *retract(answer(Distance))*. Se quisermos consultar a melhor solução efectuamos a consulta *answer(Distance)*.

Recorrendo a estas estruturas de dados foram desenvolvidos os mecanismos de suspensão e recuperação de duas formas distintas. A primeira abordagem, denominada *Chamada de Continuação*, consistiu em utilizar apenas predicados *built-in* do Yap Prolog e a segunda, denominada *Primitivas de Suspensão*, consistiu em fazer a suspensão e recuperação ao nível da máquina de execução.

No entanto, em ambas as abordagens as cláusulas e os predicados que manipulam, por exemplo, a estrutura de dados *answer/1* são iguais. Na Figura 3.4 podemos ver como o predicado *make_answer/1* manipula a estrutura de dados *answer/1*. Este predicado é responsável por, caso ainda não exista nenhuma solução, inseri-la na estrutura de dados *answer/1*. No caso de já existir, mas for encontrada uma nova solução melhor que a existente, a solução existente é removida e, posteriormente, é inserida a nova solução encontrada. Este predicado, como já foi referido, é invocado na primeira cláusula do predicado *user_search/4*.

```
make_answer(NewDistance):-
   (%Get current answer, if any
   answer(CurrentDistance) ->
     CurrentDistance > NewDistance,
     %Current answer is worse than new answer, so remove it
     retract(answer(CurrentDistance))
   ;
     %First answer
     true
   ),
   %Store new answer
   assert(answer(NewDistance)).
```

Figura 3.4: Predicado *make_answer/1*.

No final da execução, altura em que o grafo já foi todo percorrido e não há mais nenhuma computação suspensa, a execução volta para a segunda cláusula do predicado *principal_goal/3*. Aí, é retirada da estrutura de dados *answer/1* a melhor solução encontrada, e eliminada toda a informação auxiliar que existe nas estruturas de dados *waiting/3* e *done/2* através da invocação do predicado *clear_data_structures/0*. Podemos ver na Figura 3.5 o predicado *principal_goal/3* e o predicado *clear_data_structures/0*.

3.2.1 Chamada de Continuação

Nesta versão dos mecanismos de suspensão e recuperação da computação, quando se suspende a computação, guarda-se toda a informação necessária à sua posterior recuperação no predicado dinâmico *waiting/3*. Essa informação vai ser crucial para

```
principal_goal(X,Z,_):-
   search(X,Z,0,0).
principal_goal(_,_,Distance):-
   %Get answer
   retract(answer(Distance)),
   %Clear data structures
   clear_data_structures.

clear_data_structures:-
   %Clear waiting/3 data structure
   retractall(waiting(_,_,_)),
   %Clear done/2 data structure
   retractall(done(_,_)).
```

Figura 3.5: Predicados *principal_goal/3* e *clear_data_structures/0*.

retomar a computação na medida em que, nesta abordagem, como não há congelamento das pilhas de execução (*stack*, *heap* e trilha), todas as atribuições a variáveis vão perder-se.

O predicado *suspend/3*, que podemos ver na Figura 3.6, é responsável pela suspensão da computação. Este predicado, invocado pela segunda cláusula do predicado *user_search/4*, recebe como argumentos o factor que permite decidir qual é o caminho mínimo (*Min*), a distância percorrida até ao momento (*Distance*) e o caminho que falta percorrer (*Call*). Esta informação só é guardada se esse caminho ainda não tiver sido visitado. Para efectuar essa verificação, utilizamos o predicado $\backslash + \ P$ que, dado um caminho e a distância correspondente, verifica se existe na estrutura *done/2* um predicado P com a associação desses dois elementos. Se existir, a execução falha levando a que a suspensão da computação não seja realizada, evitando assim caminhos repetidos. Caso não exista, a computação é suspensa e a informação é guardada na estrutura *waiting/3*.

Após termos guardado a informação a execução falha e volta ao predicado *user_search/4*. Desta forma, garantimos que são encontrados todos caminhos directos a partir do nó corrente que foi passado como argumento ao predicado *user_search/4*. A Figura 3.6 mostra a parte do código que permite efectuar a suspensão da computação nesta versão.

A recuperação da computação é efectuada na segunda cláusula do predicado *search/4*. Nesse predicado é invocado o predicado *wake_min/3*, cujo objectivo é recuperar a informação relativa à computação com valor mínimo. Este predicado, presente na Figura 3.7, vai percorrer todas as computações suspensas, isto é, todas as cláusulas da estrutura *waiting/3*, e criar uma lista com a informação de cada uma dessas computações, nomeadamente, o valor mínimo (*Min*), distância percorrida (*Distance*)

```
user_search(X,X,Distance,_) :-
   !,
   make_answer(Distance),
   fail.
user_search(X,Z,PreviousDistance,_):-
   edge(X,Y,Distance),
   CurrentDistance is PreviousDistance + Distance,
   %Dijkstra search algorithm
   Min=CurrentDistance,
   %Save suspended info and fail
   suspend(Min,CurrentDistance,path(Y,Z)).

suspend(Min,CurrentDistance,Call):-
   %Check if the path have been already visited with this cost
   \+ done(CurrentDistance,Call),
   %Save suspend info and fail
   asserta(waiting(Min,CurrentDistance,Call)),
   fail.
```

Figura 3.6: Suspensão da computação na versão *Chamada de Continuação*.

e o caminho em falta (*Call*). Isto é conseguido pelo predicado *built-in findall(T,G,L)* que coloca numa lista L todas as instanciações do termo T que satisfazem o golo G. Neste caso concreto, T é o termo composto `Min-Distance-Call`, e G é o golo `waiting(Min,Distance,Call)`.

Posteriormente, a lista é ordenada pelo valor mínimo recorrendo ao predicado *built-in sort(L,S)* que, dada uma lista L, ordena-a e unifica a lista ordenada com S. No caso de existirem duas computações suspensas com o mesmo valor mínimo, o factor de desempate é a distância percorrida.

Depois de a lista ter sido ordenada, é invocado o predicado *get_elem/2* (Figura 3.7), que recebe como argumento essa lista e retorna os dados (o valor mínimo, a distância percorrida e o caminho em falta) da computação suspensa que tem o valor mínimo. Se o caminho que havia sido suspenso já foi visitado e a distância percorrida guardada (CurrentDistance) for menor que a da computação que pretendemos acordar (New-Distance), a computação é ignorada e é retirada outra computação em espera da lista (segunda cláusula do predicado *get_elem/2*).

Caso o caminho não tenha sido ainda visitado ou, no caso de já ter sido, a distância percorrida guardada for maior que a da computação que pretendemos retomar, situação que leva a uma remoção do caminho anteriormente visitado da estrutura *done/2*, guardamos a distância juntamente com o caminho e retornamos os dados desta computação. Após o predicado *get_elem/2* retornar os dados de uma computação suspensa, a execução do predicado *wake_min/3* sucede e a execução volta para a segunda cláusula do predicado *search/4*. Nessa altura, é invocado de forma recursiva

```
search(X,Z,Distance,Min):-
   user_search(X,Z,Distance,Min).
search(_,_,_,_):-
   %Resume computation
   wake_min(NewMin,NewDistance,path(X,Z)),
   search(X,Z,NewDistance,NewMin).

wake_min(NewMin,NewDistance,Call):-
   findall(Min-Distance-Goal,waiting(Min,Distance,Goal),List),
   sort(List,SortedList),
   get_elem(SortedList,NewMin-NewDistance-Call).

get_elem([NewMin-NewDistance-Call|T],NewMin-NewDistance-Call):-
   %Pick up the next suspended computation
   retract(waiting(NewMin,NewDistance,Call)),
   (done(CurrentDistance,Call)->
      %The path has already been visited, so check if the new distance is better
      CurrentDistance > NewDistance,
      %The new distance is better, so remove the previous one from done/2
      retract(done(CurrentDistance,Call))
   ;
      %The path has not yet been visited
      true
   ),
   !,
   %Mark the path as visited
   asserta(done(NewDistance,Call)).
get_elem([H|T],Call):-
   %Try the next suspended computation
   get_elem(T,Call).
```

Figura 3.7: Recuperação da computação na versão *Chamada de Continuação*.

o predicado *search/4* com os argumentos que correspondem aos dados da computação que tem o valor mínimo retornados pelo *wake_min/3*.

Quando não existir mais nenhuma computação suspensa, altura em que o predicado *wake_min/3* falha, a computação volta para o predicado de topo *principal_goal/3*, mais concretamente para a segunda cláusula, de forma a retornar a distância mínima encontrada.

Tomemos como exemplo as cláusulas do predicado *edge/3* que dão origem ao grafo da Figura 3.8.

Se quisermos saber qual o caminho mínimo de a para c temos que efectuar a consulta `principal_goal(a,c,Distance)`. Podemos ver na Figura 3.9 a execução desta consulta, onde utilizamos o algoritmo de *Dijsktra*, com os mecanismos desenvolvidos nesta versão.

Os passos 4, 6 e 9 correspondem à suspensão da computação e, por consequência, à inserção na estrutura *waiting/3* da informação relacionada com cada uma das computações suspensas (o valor mínimo, a distância percorrida e o caminho).

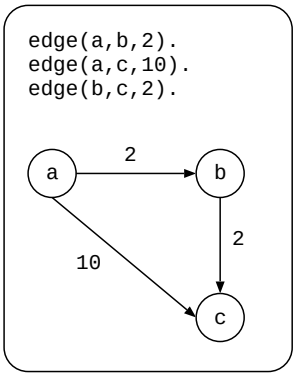

Figura 3.8: Exemplo de um grafo definido por cláusulas *edge/3*.

Os passos 7 e 10 correspondem à recuperação da computação suspensa com valor mínimo e, por isso, à remoção das respectivas entradas na estrutura *waiting/3* e à marcação desses caminhos como visitados, através da inserção na estrutura *done/2* do caminho e da respectiva distância. De notar que a computação suspensa no passo 6 não é recuperada (é descartada) pois, esse caminho já havida sido visitado com um menor custo (esse caminho foi marcado como visitado no passo 10). Por essa razão, a tentativa de recuperar uma computação suspensa realizada no passo 12 leva a que a execução falhe (passo 13) logo, a execução vai voltar para a segunda cláusula do predicado de topo onde, depois de retornar a distância mínima e limpar as estruturas *waiting/3* e *done/2*, termina (passo 14).

3.2.2 Primitivas de Suspensão

A segunda versão dos mecanismos de suspensão e recuperação da computação foi implementada com recurso a dois predicados de baixo nível que já se encontravam implementados na máquina de execução. Esses predicados são o *freeze_choice_point/1* e o *wake_choice_point/1* que permitem, respectivamente, preservar as atribuições a variáveis efectuadas até ao momento da suspensão de modo a que seja possível repor o seu estado no momento da recuperação.

Para cada computação que é suspensa a função *freeze_choice_point/1* cria um ponto de escolha e protege as pilhas de execução utilizando para isso um conjunto de *freeze_registers*, mais concretamente, o *B_FZ* para a *stack*, o *H_FZ* para a *heap* e o *TR_FZ* para a trilha. O ponto de escolha é retornado e serve como identificador da computação suspensa.

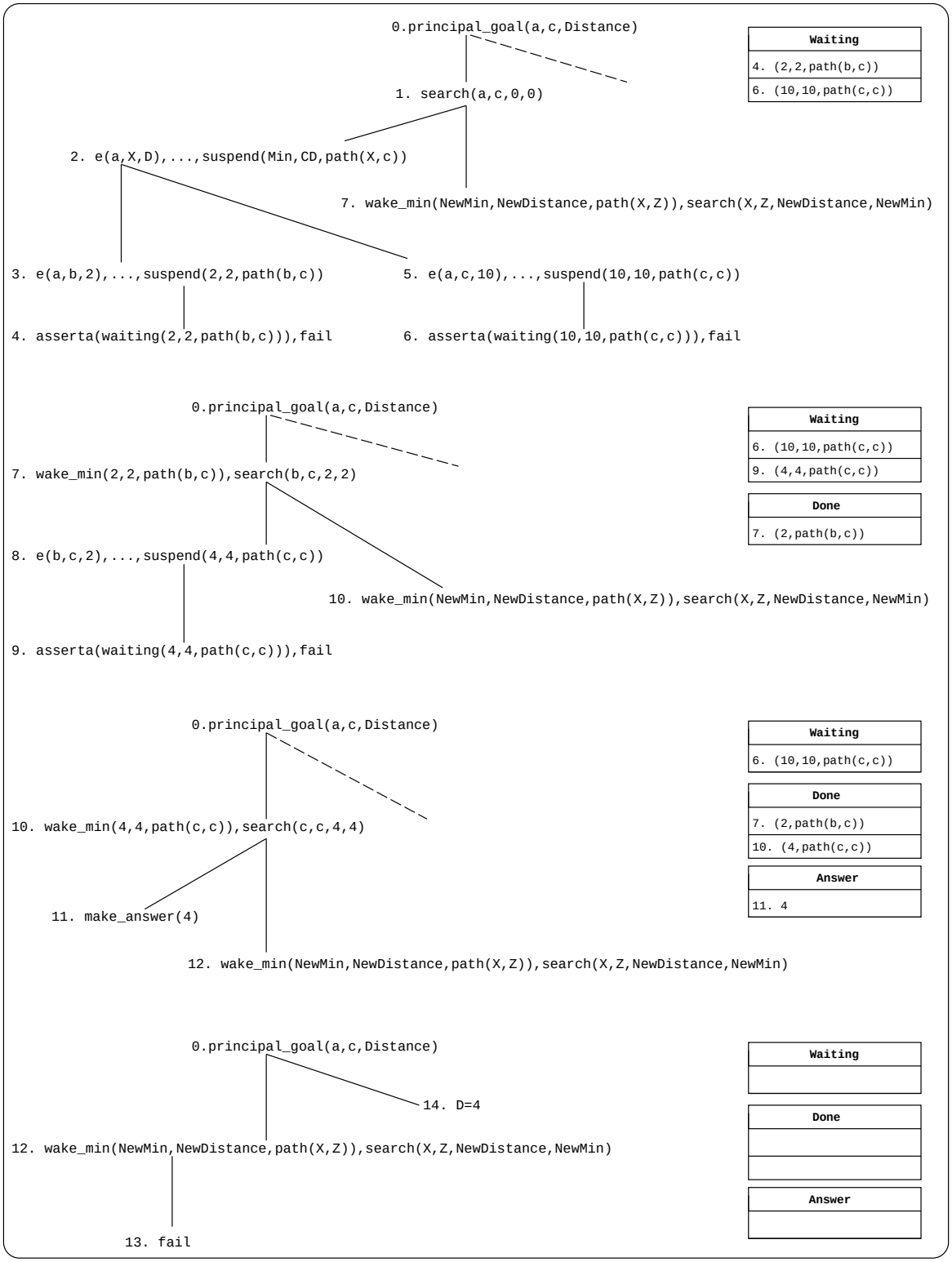

Figura 3.9: Árvore de execução da consulta `principal_goal(a,c,Distance)` para o algoritmo de *Dijsktra* e utilizando os mecanismos de suspensão e recuperação da versão *Chamada de Continuação*.

A função *wake_choice_point/1* recebe como argumento o identificador de uma computação suspensa e restaura todas as atribuições condicionais dessa computação. O restauro das atribuições condicionais é conseguido por utilização da *forward trail*, uma extensão à trilha da WAM usada pelo YapTab.

Para suspender uma computação criamos dois predicados auxiliares: o predicado *freeze_and_suceed_once/2* e o *freeze_and_fail_once/2*. Estes predicados recebem como argumento um golo G e retornam o ponto de escolha (CP) criado pela função *freeze_choice_point/1*. O golo G, que corresponde à inserção de informação em uma estrutura de dados auxiliar, é executado através do predicado *built-in call(G)*. A Figura 3.10 mostra o código dos predicados *freeze_and_suceed_once/2* e *freeze_and_fail_once/2*.

```
freeze_and_suceed_once(CP,Goal):-
   freeze_choice_point(CP),
   call(Goal),
   true
   ;
   fail.

freeze_and_fail_once(CP,Goal):-
   freeze_choice_point(CP),
   call(Goal),
   fail
   ;
   true.
```

Figura 3.10: Predicados *freeze_and_suceed_once/2* e *freeze_and_fail_once/2* desenvolvidos na versão *Primitivas de Suspensão*.

Devido ao facto da função *freeze_choice_point/1* criar um ponto de escolha (CP), necessitamos de ter nos predicados *freeze_and_suceed_once/2* e *freeze_and_fail_once/2* código para ser executado na altura da suspensão, que é executado apenas uma vez, e código para ser executado quando retomarmos a computação, que é executado mediante o número de vezes que a computação for retomada. Isto é conseguido através do uso do predicado *built-in* P ; Q que irá executar o golo P apenas uma vez e o golo Q tantas vezes quantas o número de vezes que a computação for retomada.

A principal diferença entre os dois predicados reside no facto de, após suspendermos a computação e guardarmos a informação relevante, a execução suceder ou falhar. Mediante isso, quando a computação for recuperada vai ter o comportamento contrário, ou seja, se na suspensão a execução falhar, quando acordarmos a execução vai suceder.

No predicado *freeze_and_suceed_once/2*, por exemplo, quando suspendemos a computação a execução vai suceder e quando retomamos a execução vai falhar, enquanto que no predicado *freeze_and_fail_once/2* acontece o contrário, quando suspendemos a

execução falha e quando retomamos a execução sucede.

Como estamos a suspender a computação ao nível da máquina de execução, não podemos deixar que a computação falhe por mais do que uma vez por *backtracking* sobre um ponto de escolha (*CP*) suspenso, porque da segunda vez podemos estar a aceder a caminhos já explorados, e por isso não válidos. Sendo assim, para além de suspendermos a computação na segunda cláusula do predicado *user_search/4* (através da chamada ao predicado *suspend/3*), que serve para guardar as alternativas que ainda se encontram por explorar, necessitamos também de suspender a computação no predicado de topo *principal_goal/3* de modo a voltar a esse predicado quando não houver mais alternativas. Se não suspendermos a computação no predicado de topo e não houver mais alternativas por explorar, a execução por *backtracking* pode aceder a áreas de memória já exploradas, e por isso inválidas, causando a interrupção abrupta do programa.

O predicado *principal_goal/3* continua a ser composto por duas cláusulas, das quais apenas a primeira sofreu alterações. A Figura 3.11 mostra o predicado *principal_goal/3* nesta versão.

```
principal_goal(X,Z,_):-
   %Suspend computation before begin the execution
   freeze_and_suceed_once(CP,asserta(waiting_top(CP))),
   search(X,Z,0,0).
principal_goal(X,Z,Distance):-
   %Get answer
   retract(answer(Distance)),
   %Clear data structures
   clear_data_structures.

clear_data_structures:-
   %Clear waiting/4 data structure
   retractall(waiting(_,_,_,_)),
   %Clear done/2 data structure
   retractall(done(_,_)).
```

Figura 3.11: Predicado *principal_goal/3* na versão *Primitivas de Suspensão*.

Na primeira cláusula passamos a invocar o predicado *freeze_and_suceed_once/2* de forma a suspender a computação e, dado que a execução sucede após a suspensão, a possibilitar o início da pesquisa através da chamada do predicado *search/4*. O predicado *freeze_and_suceed_once/2* recebe como argumento o golo que vai permitir guardar numa nova estrutura de dados auxiliar, denominada *waiting_top/1*, o ponto de escolha criado pela chamada à função *freeze_choice_point/1*. A utilização de uma estrutura de dados auxiliar diferente para o ponto de escolha de topo permite diferenciá-lo dos restantes pontos de escolha suspensos durante a execução do programa e guardados

na estrutura de dados *waiting/4*.

O predicado *suspend/3*, que continua a ser invocado na segunda cláusula do predicado *user_search/4*, nesta versão continua a receber os mesmos três argumentos porque, devido ao problema que estamos a resolver, precisamos do factor que permite decidir qual é o caminho mínimo (*Min*) e da distância percorrida até ao momento (*Distance*) para ordenar as computações suspensas. Necessitamos também do caminho entre dois nós (*Call*) que, juntamente com a distância percorrida, permite verificar se o caminho já foi visitado. Caso o caminho ainda não tenha sido visitado, é chamado o predicado *freeze_and_fail_once/2* que recebe como argumento o golo que permite guardar na estrutura de dados auxiliar *waiting/4* o valor mínimo, a distância percorrida, o caminho e, como foi referido anteriormente, o ponto de escolha (*CP*) retornado pela função *freeze_choice_point/1*. Neste predicado utilizamos o *freeze_and_fail_once/2* pois, como após a suspensão a execução falha, torna possível visitar todos caminhos directos a partir do nó inicial que é passado como argumento ao predicado *user_search/4*. Na Figura 3.12 temos o predicado *suspend/3* nesta versão.

```
suspend(Min,Distance,Call):-
   %Check if the path have been already visited with this cost
   \+ done(Distance,Call),
   %Suspend computation
   freeze_and_fail_once(CP,asserta(waiting(Min,Distance,Call,CP))).
```

Figura 3.12: Predicado *suspend/3* na versão *Primitivas de Suspensão*.

A recuperação de uma computação continua a ser realizada na segunda cláusula do predicado *search/4* (Figura 3.13). Aí invocamos o predicado *wake_min/1* (cujo funcionamento é igual ao da versão anterior) para, com recurso ao predicado *get_elem/2* (cujo funcionamento é igual ao da versão anterior), retornar o identificador da computação suspensa com valor mínimo. Se o predicado *wake_min/1* suceder, continuamos a explorar o grafo; caso contrário a execução tem de voltar para o predicado de topo de forma a retornar a solução mínima encontrada. Podemos ver o código que realiza a recuperação da computação nesta versão na Figura 3.13.

Para continuarmos a explorar o grafo a partir de uma das computações suspensas, invocamos o predicado de baixo nível *wake_choice_point/1* que recebe como argumento o identificador retornado pelo predicado *wake_min/1*. Como já foi referido anteriormente, esta função vai restaurar todas as atribuições condicionais e retomar a execução a partir do predicado *freeze_and_fail_once/2*. Neste predicado, vai ser executado o predicado *built-in true*, que é a alternativa que está por explorar, levando a que a execução do predicado *freeze_and_fail_once/2* suceda. Como a execução sucede, o

```
search(X,Z,Distance,Min):-
   user_search(X,Z,Distance,Min).
search(_,_,_,_):-
   %Resume Computation
   (wake_min(CP) ->
      wake_choice_point(CP)
   ;
      retract(waiting_top(TopCP)),
      wake_choice_point(TopCP)
   ).

wake_min(CP):-
   findall(Min-Distance-Goal-CPoint,waiting(Min,Distance,Goal,CPoint),List),
   sort(List,SortedList),
   get_elem(SortedList,CP).

get_elem([NewMin-NewDistance-Call-CP|T],CP):-
   %Pick up the next suspended computation
   once(retract(waiting(NewMin,NewDistance,Call,CP))),
   (done(CurrentDistance,Call)->
      %The path has already been visited, so check if the new distance is better
      CurrentDistance > NewDistance,
      %The new distance is better, so remove the previous one from done/2
      retract(done(CurrentDistance,Call))
   ;
      %The path has not yet been visited
      true
   ),
   %Mark the path as visited
   asserta(done(NewDistance,Call)).
get_elem([H|T],Call):-
   %Try the next suspended computation
   get_elem(T,Call).
```

Figura 3.13: Recuperação da computação na versão *Primitivas de Suspensão*.

predicado *suspend/3*, que havia invocado esse predicado, também vai suceder voltando a execução para o predicado *user_search/4*. Aqui, vai ser invocado agora o predicado *search/4* com o novo nó corrente, o mesmo nó final, a distância percorrida actualizada e o novo valor mínimo como argumentos. Isto só é possível porque o predicado *wake_choice_point/1* consegue recuperar todas as atribuições efectuadas previamente até ao predicado *user_search/4*. Na Figura 3.14 podemos ver o predicado *user_search/4* nesta versão.

Se não existir mais nenhuma alternativa por explorar (altura em que o predicado *wake_min/1* falha) temos que acordar o ponto de escolha referente ao predicado de topo. Para isso, retiramos o ponto de escolha do predicado de topo da estrutura *waiting_top/1*. Posteriormente, invocamos a função *wake_choice_point/1*, com esse ponto de escolha como argumento, e a execução volta para o predicado *freeze_and_suceed_once/2*. Aí vai ser executado o predicado *built-in fail* fazendo com que a execução deste predicado falhe. A execução passa assim para a segunda cláusula do predicado *principal_goal/3* (Figura 3.11), encarregue de retornar a solução mínima encontrada e limpar as estruturas *waiting/4* e *done/2*.

```
user_search(X,X,Distance,_) :-
   !,
   make_answer(Distance),
   fail.
user_search(X,Z,PreviousDistance,_):-
   edge(X,Y,Distance),
   CurrentDistance is PreviousDistance + Distance,
   %Dijkstra search algorithm
   Min=CurrentDistance,
   %Suspend computation
   suspend(Min,CurrentDistance,path(Y,Z)).
   %Code to be executed when the computation is resumed
   search(Y,Z,CurrentDistance,_).
```

Figura 3.14: Predicado *user_search/4* na versão *Primitivas de Suspensão.*

A Figura 3.15 mostra a execução da consulta `principal_goal(a,c,Distance)` sobre o grafo da Figura 3.8, com os mecanismos desenvolvidos nesta versão. A primeira suspensão realiza-se no predicado de topo mal a execução começa e, por consequência, existe a introdução do ponto de escolha na estrutura *waiting_top/1* (passo 0). No decorrer da execução são realizadas mais três suspensões (passos 4, 6 e 9) da execução que vão levar à introdução na estrutura *waiting/4* da informação relacionada com cada uma das computações suspensas (o valor mínimo, a distância percorrida, o caminho e o ponto de escolha) e ao congelamento das pilhas de execução.

Os passos 7 e 10 correspondem à recuperação de computações suspensas ao longo da execução e que têm o valor mínimo. Logo, existe a remoção das respectivas entradas na estrutura *waiting/4* e a marcação desses caminhos como visitados, através da inserção na estrutura *done/2* do caminho e da respectiva distância. De notar que a computação suspensa no passo 6 não é recuperada (é descartada), tal como acontecia na versão anterior, pois, esse caminho já havida sido visitado com um menor custo (esse caminho foi marcado como visitado no passo 10). Por essa razão, no passo 12 em vez de acordamos uma computação suspensa ao longo da execução vamos acordar o ponto de escolha referente ao predicado de topo (que está guardado na estrutura *waiting_top/1*). A execução vai então para a segunda cláusula do predicado de topo onde, depois de retornar a distância mínima e limpar as estruturas *waiting/4* e *done/2*, termina (passo 14).

A Figura 3.16 mostra um exemplo que ilustra o que acontece às pilhas de execução quando ocorre uma suspensão da computação. Aquando da primeira suspensão, que ocorre no nó F1, é criado um ponto de escolha (CP_F1) cuja função é proteger as pilhas de execução dos nós N1 e N2, mais concretamente, impedir que essas pilhas sejam apagadas. Isto faz com que na continuação da execução os novos nós, por exemplo,

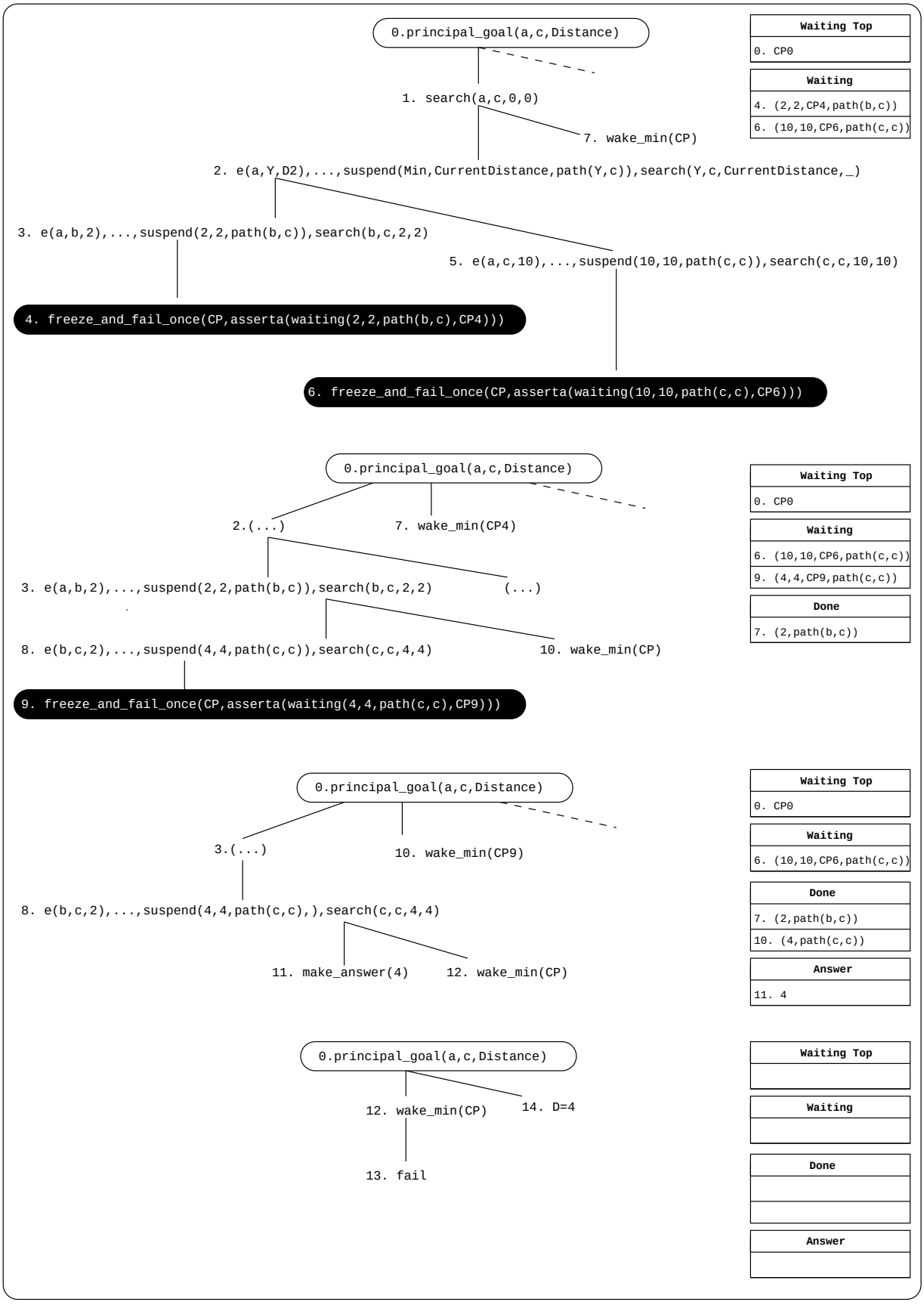

Figura 3.15: Árvore de execução da consulta `principal_goal(a,c,Distance)` para o algoritmo de *Dijsktra* utilizando os mecanismos de suspensão e recuperação da versão *Primitivas de Suspensão*.

o nó N3 (cujo ponto de escolha é CP_N3), sejam guardados em posições posteriores ao ponto de escolha suspenso CP_F1. O registo *B_FZ* aponta para o último ponto de escolha suspenso, que na situação (a) é CP_F1 e na situação (b) passa a ser CP_F2.

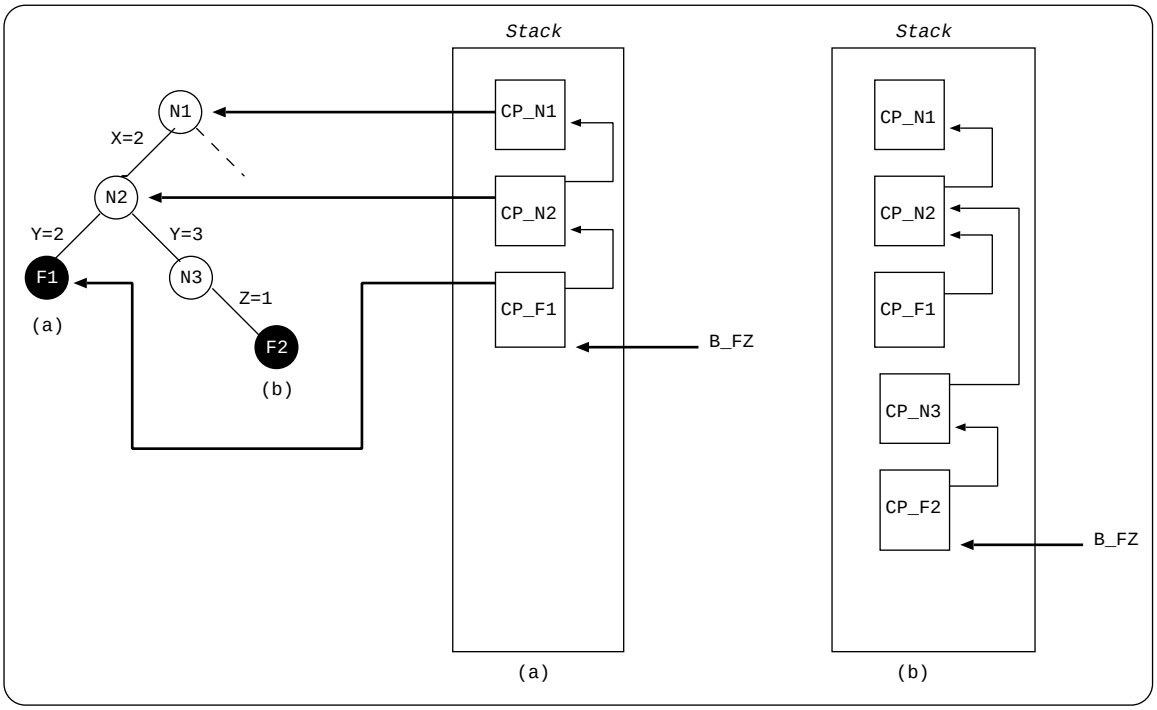

Figura 3.16: Suspensão das pilhas de execução.

Se na continuação, recuperarmos a computação de CP_F2 para CP_F1, todas as atribuições efectuadas até F2 devem ser removidas e todas as atribuições efectuadas até F1 devem ser recolocadas. No entanto, basta realizar estas operações - remover e recolocar atribuições - até se atingir o primeiro nó comum (no caso da Figura 3.16 o nó comum é o nó N2). Vejamos na Figura 3.17, o que acontece ao nível da trilha.

Quando ocorrer a recuperação da computação de CP_F2 para CP_F1, a remoção de todas as atribuições efectuadas até F2 é realizada a partir do registo TR guardado no ponto de escolha CP_F2. As atribuições efectuadas até F1 são recolocadas a partir do registo TR guardado no ponto de escolha CP_F1. Os dois registos avançam no sentido ascendente até atingirem a primeira entrada comum, que no caso da Figura 3.17 é a entrada referente ao registo TR guardado no nó N2 (X=2). Nessa altura, todas as atribuições referentes a F1 já foram recolocadas e todas as atribuições referentes a F2 removidas.

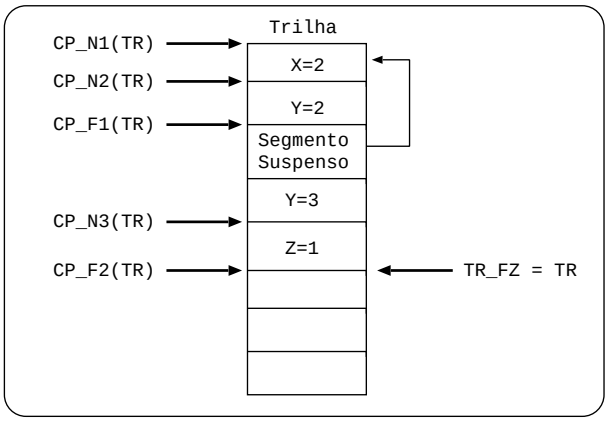

Figura 3.17: Conteúdo da trilha (*forward trail*) após a suspensão no nó F2 da Figura 3.16.

3.3 Sumário

Neste capítulo foram descritas duas formas de implementar mecanismos de suspensão e recuperação da computação por utilização de predicados *built-in* do Yap Prolog. Começamos por abordar a ideia base e o problema onde nos baseamos para desenvolver estes mecanismos. Posteriormente, descrevemos em detalhe cada uma das implementações efectuadas. A primeira versão, denominada *Chamada de Suspensão*, apenas utiliza predicados *built-in* do Yap Prolog, sendo por isso necessário guardar toda a informação necessária para retomar uma computação. Na segunda versão, denominada *Primitivas de Suspensão*, a suspensão e recuperação da computação são efectuadas ao nível da máquina de execução. Nesta versão - onde também são utilizados predicados *built-in* do Yap Prolog - dado que as pilhas de execução são protegidas, as atribuições efectuadas até à suspensão são preservadas. Assim, ao recuperamos a computação, essas atribuições são também recuperadas.

Os mecanismos desenvolvidos neste capítulo, mais concretamente os mecanismos desenvolvidos na segunda versão, vão ser de extrema importância na implementação do controlo de fluxo que iremos abordar no próximo capítulo.

Capítulo 4

Tabulação com Primitivas de Suspensão

Neste capítulo iremos abordar o desenvolvimento de um módulo externo que implementa a técnica da tabulação por transformação de programas e utilização de primitivas de suspensão. Inicialmente abordaremos os factores que nos motivaram a desenvolver este módulo - a ideia geral. Depois, iremos descrever, em detalhe, todos os passos da implementação do módulo. Começaremos pela versão base e, posteriormente, passaremos às diversas optimizações efectuadas: consumir soluções de forma incremental (*incremental answer resolution*); completar computações de forma incremental (*incremental completion*); utilização de outra estratégia de escalonamento (*batched scheduling*); e utilização de estruturas de dados alternativas para guardar a informação envolvida na computação.

4.1 Ideia Geral

A técnica da tabulação, como foi referido anteriormente, foi proposta para solucionar algumas das limitações que a resolução SLD, utilizada pelo Prolog, tem quando estamos a lidar com computações redundantes e/ou quando estamos perante ciclos infinitos. Actualmente, a maioria dos sistemas de tabulação existentes estão implementados ao nível da máquina de execução. Contudo, alguns desses sistemas só são activados se a máquina de execução for compilada com parâmetros específicos. Por exemplo, no caso do sistema YapTab, que serviu de modelo à nossa implementação, necessitamos de compilar a máquina de execução com a opção - - *enable-tabling*.

Outra forma de implementar tabulação é ao nível do próprio Prolog, por utilização de um programa de transformação que adiciona ao programa original, um conjunto de directivas de controle. Este é o caso das propostas apresentadas em [24, 4, 18], todas elas baseadas na proposta original de Ramesh e Chen [15] em que o programa de transformação é escrito em Prolog e as directivas são implementadas na linguagem C, mais concretamente utilizando o interface para linguagem C dos sistemas Prolog.

A nossa abordagem, tal como as anteriores, passa por implementar tabulação ao nível do próprio Prolog. No entanto, tanto o programa de transformação como as directivas de controle são implementadas ao nível do Prolog. Para utilizar a tabulação na nossa abordagem, o utilizador necessita de passar o seu programa pelo programa de transformação e, posteriormente, carregar o programa transformado juntamente com o módulo que implementa as directivas da tabulação. A Figura 4.1 esquematiza o que acabamos de descrever.

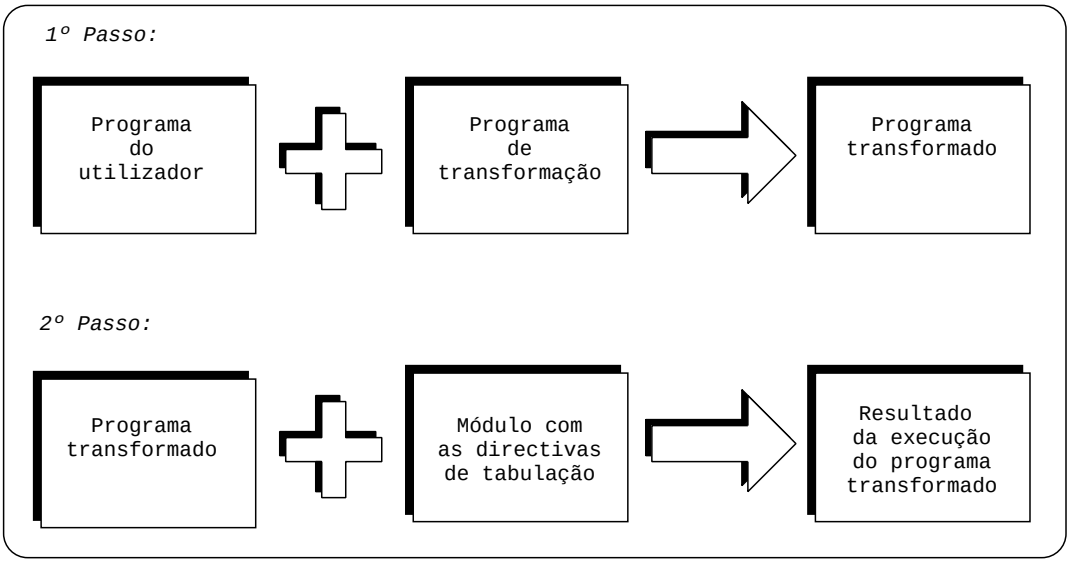

Figura 4.1: Passos que o utilizador deve seguir para utilizar tabulação na nossa abordagem.

O programa de transformação, que recebe como input o programa do utilizador e retorna um novo programa cujo formato é reconhecido pelo módulo, torna a utilização do módulo completamente transparente para o utilizador na medida em que o utilizador continua a realizar as consultas da mesma forma. Vejamos o exemplo da Figura 4.2.

Do lado esquerdo da figura temos o programa original onde podemos ver que o predicado *path/2* é o único predicado tabelado. Do lado direito temos o programa transformado. De realçar que a transformação só incide sobre os predicados tabelados,

```
:-yap_flag(tabling_mode,local).
:-table path/2.

path(X,Z) :-
   edge(X,Y),
   path(Y,Z).
path(X,Z) :-
   edge(X,Z).

edge(1,2).
edge(2,3).
```

```
path(X,Z) :-
   tabled_call(pathtrans(path(X,Z),AnsTrie)).

pathtrans(path(X,Z),AnsTrie) :-
   edge(X,Y),
   path(Y,Z),
   new_answer(path(X,Z),AnsTrie).
pathtrans(path(X,Z),AnsTrie) :-
   edge(X,Z),
   new_answer(path(X,Z),AnsTrie).

edge(1,2).
edge(2,3).

:- trie_open(SubTrie),
   recorda(table_entry,[path/2,local|SubTrie],_).
```

Figura 4.2: Programa em Prolog para encontrar caminhos num grafo antes e depois da aplicação do programa de transformação.

logo o predicado *edge/2* não sofre qualquer alteração (apenas o predicado *path/2* é transformado). O predicado *path/2* no programa transformado passa a ter apenas uma cláusula, responsável por despoletar o mecanismo de tabulação através da chamada do predicado *tabled_call/1*, que faz parte do módulo desenvolvido e, por essa razão, irá ser abordado em detalhe mais à frente neste capítulo.

O predicado *pathtrans/2* é a versão transformada do predicado *path/2* presente no programa original. Este novo predicado, cujo número de cláusulas é o mesmo que o predicado *path/2* original, recebe agora como argumentos o subgolo da chamada a executar e a referência para a *answer trie* onde devem ser guardadas as soluções encontradas. As duas cláusulas deste predicado definem os mesmos procedimentos das cláusulas do predicado *path/2* do programa original e, no final, inserem a nova solução encontrada na *answer trie*. Por exemplo, na segunda cláusula verifica-se se existe uma ligação de X para Z e, caso exista, insere-se na *answer trie* a solução *path(X,Z)*. A inserção de novas soluções na *answer trie* está a cargo do predicado *new_answer/2* que faz parte do módulo desenvolvido e, por isso, será abordado mais adiante.

Quando o programa transformado for carregado, para cada predicado tabelado vai ser iniciada uma nova *trie* e guardado numa estrutura auxiliar, denominada *table_entry*, o nome e a aridade do predicado tabelado, a estratégia de escalonamento e a referência da *trie* que acabou de ser iniciada. Por exemplo, no caso da Figura 4.2 é guardado na estrutura auxiliar o predicado tabelado *path/2* que vai ser executado com estratégia de escalonamento *local scheduling*, e a referência da *trie* iniciada para esse predicado.

De forma a ser possível efectuar a suspensão e a recuperação da computação, o nosso módulo tira partido dos mecanismos desenvolvidos na versão primitivas de suspensão cujos detalhes da implementação foram abordados no capítulo anterior.

O nosso módulo utiliza ainda a estrutura de dados *tries*, para guardar os subgolos tabelados e as respectivas soluções (tal como descrito na secção 2.2.2), bem como um conjunto de outras estruturas de dados auxiliares, que iremos abordar mais à frente, para guardar a informação relativa ao controle da computação com tabulação.

Tal como em qualquer sistema de tabulação, no nosso módulo temos nós geradores e nós consumidores. Um subgolo é denominado nó gerador quando corresponde à primeira chamada de um subgolo. O nó gerador é responsável por criar as estruturas de dados necessárias na tabela e iniciar a computação. As funções de um nó gerador estão ainda relacionadas com a estratégia de escalonamento adoptada. Por exemplo, no caso da estratégia *local scheduling*, um nó gerador é ainda responsável por, no final da computação, consumir as soluções encontradas.

Dentro dos nós geradores temos ainda os nós líderes que, para além de efectuarem as funções de um nó gerador, são responsáveis por reactivar as computações suspensas no *Strongly Connected Component* (SCC) ou marcar como completos os nós que formam o seu SCC. O nó líder é guardado numa estrutura de dados diferente dos restantes nós, de forma a garantir que este nó apenas é retomado quando todos os subgolos pertencentes ao seu SCC estiverem totalmente avaliados.

Se uma chamada a um subgolo já existir na *trie*, então é denominado nó consumidor. A única função deste tipo de nó é consumir as soluções encontradas até ao momento. Quando não existirem mais soluções, a computação é suspensa podendo vir a ser reactivada mais tarde por um nó líder.

Iremos agora abordar os detalhes da implementação da versão base que serviu para, posteriormente, irmos efectuando, de forma incremental, diversas optimizações.

4.2 Versão Base

A primeira versão do nosso módulo consistiu em reproduzir, de forma minimalista, o mecanismo de tabulação do YapTab utilizando a estratégia de escalonamento *local scheduling*. Para isso foi necessário, para além de utilizar a estrutura de dados *tries*, criar diferentes estruturas de dados auxiliares para guardar a informação relativa à

computação. As estruturas criadas foram as seguintes:

- *table_entry* - guarda o nome e a aridade de cada um dos predicados tabelados, a estratégia de escalonamento e o respectivo identificador da *trie*.

- *subgoal_frame* - guarda a informação que nos permite navegar na *subgoal trie* e na *answer trie* de cada subgolo tabelado, nomeadamente o *subgoal identifier* (SgId), a referência do subgolo na *trie* (SgFr), o identificador da *answer trie* (AnsTrie) e o estado da computação (State) correspondente. O SgId é um número único usado para distinguir/identificar cada subgolo tabelado.

- *frozen_cp* - guarda a informação necessária para recuperar computações suspensas, mais concretamente o *frozen identifier* (FzId), o ponto de escolha que permite acordar uma computação suspensa (CP), o identificador da *answer trie* (AnsTrie) e o *subgoal identifier* (SgId) correspondente. É necessário haver um número único, o FzId, associado a cada computação suspensa visto que podem existir várias computações suspensas para o mesmo subgolo.

- *leader_cp* - guarda as referências para os pontos de escolha associados a nós líderes.

- *waiting_cp* - guarda as referências para os pontos de escolha das computações associadas a nós consumidores que têm que ser retomadas.

Estas estruturas de dados foram implementadas, nesta primeira versão, com recurso à base de dados interna do Yap Prolog. Cada conjunto de termos guardados vai ter associado um termo chave distinto dos restantes. Cada termo pertencente a um conjunto tem associado uma referência única. Os predicados *built-in* do Yap Prolog que permitem a inserção, consulta e remoção de informação na base de dados interna são:

- *recorda(K,T,R)* - insere o termo T na estrutura K, ambos passados como argumento, e retorna a referência R. O termo T passa a ser o primeiro termo associado a K.

- *recordz(K,T,R)* - insere o termo T na estrutura K tal como o predicado *recorda*, mas o termo T passa a ser o último termo associado a K.

- *recorded(K,T,R)* - procura na base de dados interna por um termo que esteja associado ao termo chave K, passado como argumento, que unifique com o termo T (também passado como argumento). Este predicado retorna a referência R.

- *erase(R)* - remove o termo ao qual está associado a referência R passada como argumento. Para obtermos a referência R necessitamos de, anteriormente, procurar o termo que pretendemos remover.

- *eraseall(K)* - remove todos os termos pertencentes ao termo chave K.

Cada subgolo tabelado, cuja informação está armazenada na estrutura de dados *subgoal_frame*, tem associado a si o estado actual da computação. A computação, nesta primeira versão, passa por quatro estados:

1. *ready* - a computação encontra-se neste estado quando um subgolo é inserido na *trie*. Apenas subgolos com este estado podem dar início à computação;

2. *eval* - a computação passa para este estado antes de um subgolo dar início à computação e quando o líder do *Strongly Connected Component* (SCC) correspondente decide escalonar o subgolo para ser reavaliado;

3. *eval_new* - a computação passa para este estado quando são encontradas novas soluções;

4. *complete* - a computação passa para este estado quando todos os subgolos que formam o SCC estão completamente avaliados. A passagem para este estado é realizada pelo líder do SCC.

Para um subgolo ser considerado como nó gerador, então o estado da computação tem que estar a *ready*. Nesta versão, só existe um único líder para toda a computação, que é o primeiro subgolo gerador (com o SgId igual a zero). Subgolos com o estado a *eval* ou a *eval_new* são considerados nós consumidores.

A inserção de novas soluções está a cargo do predicado *new_answer/2*, que podemos ver na Figura 4.3, cujos argumentos são o subgolo referente à nova solução e a referência para a *answer trie*. Para inserirmos uma nova solução procuramos na estrutura *subgoal_frame* por um termo que inclua a referência da *answer trie* passada como argumento e, caso o estado da computação seja *eval_new*, inserimos a nova solução na *answer trie* através da função *trie_put_entry/3*. Se o estado for diferente

de *eval_new*, verificamos se a solução já existe na *answer trie* através da função *trie_check_entry/3*. Caso a solução não exista, inserimo-la e alteramos o estado da computação para *eval_new*. Como a estratégia de escalonamento adoptada inicialmente na implementação deste módulo foi a estratégia *local scheduling*, após adicionarmos uma nova solução a execução deve falhar.

```
new_answer(Call,AnsTrie) :-
   recorded(subgoal_frame,[SgId,SgFr,AnsTrie|SgState],Ref),
   (SgState = eval_new ->
      trie_put_entry(AnsTrie,Call,_)
   ;
      (trie_check_entry(AnsTrie,Call,_) ->
         true
      ;
         trie_put_entry(AnsTrie,Call,_),
         %Update subgoal state to eval_new
         erase(Ref),
         recordz(subgoal_frame,[SgId,SgFr,AnsTrie|eval_new],_)
      )
   ),
   fail.
```

Figura 4.3: Predicado *new_answer/2* na versão base.

O predicado *tabled_call/1* (Figura 4.4) como tem como argumento a chamada ao predicado correspondente à versão transformada do predicado tabelado (no caso do programa da Figura 4.2 o argumento é a chamada ao predicado *pathtrans/2*), necessita de efectuar o tratamento do *input* de forma a conseguir obter o predicado tabelado, mais concretamente o nome e a aridade, e a referência para a *answer trie*. Isto é conseguido por utilização do predicado *built in* do Yap Prolog T =.. L que constrói uma lista L com o functor e os argumentos do termo T. Por exemplo, no caso do programa da Figura 4.2 o termo seria `pathtrans(path(A,B),AnsTrie)` e a lista resultante seria `[pathtrans,path(A,B),AnsTrie]`.

```
tabled_call(TransCall):-
   %Parse the input
   TransCall =.. [_,Call,AnsTrie],
   Call =.. [Name|Args},
   length(Args,Arity),
   %Get tabled predicate info
   recorded(table_entry,[Name/Arity,_|SubTrie],_),
   %Check/insert the subgoal in the subgoal trie SubTrie and return the subgoal info
   tabled_call_check_insert(SubTrie,Call,SgId,SgFr,AnsTrie,SgState),
   %Begin execution
   tabled_call_options(TransCall,Call,SgId,SgFr,AnsTrie,SgState).
```

Figura 4.4: Predicado *tabled_call/1* na versão base.

Após ter sido efectuado o tratamento do *input*, procuramos na estrutura *table_entry* a referência da *trie* do predicado tabelado e verificamos se o subgolo já foi inserido na

trie. A tarefa de verificar se um subgolo já existe na *trie* e, caso não exista, inserilo está a cargo do predicado *tabled_call_check_insert/6* que podemos ver na Figura 4.5. Se já existir, com a referência do subgolo na *trie* conseguimos pesquisar na estrutura *subgoal_frame* a informação associada a esse subgolo (informação essa que nos permite navegar tanto na *subgoal trie* como na *answer trie* e saber o estado da computação) para ser retornada. Caso contrário, inserimos esse subgolo na *trie* do predicado tabelado; criamos uma nova *answer trie*; obtemos um novo *subgoal identifier* (SgId) para esse subgolo; inserimos essa informação (o SgId, a referência do subgolo na *trie* e o identificador da *answer trie*) na estrutura *subgoal_frame* juntamente com o estado da computação (*ready*); e, por fim, retornamos toda a informação inserida.

```
tabled_call_check_insert(SubTrie,Call,SgId,SgFr,AnsTrie,SgState):-
   trie_check_entry(SubTrie,Call,SgFr),
   %The subgoal already exists, so get subgoal info
   recorded(subgoal_frame,[SgId,SgFr,AnsTrie|SgState],_),
   !.
tabled_call_check_insert(SubTrie,Call,SgId,SgFr,AnsTrie,SgState):-
   %The subgoal is new, so insert subgoal info
   trie_put_entry(SubTrie,Call,SgFr),
   trie_open(AnsTrie),
   get_new_id(subgoal_frame_counter,SgId),
   recordz(subgoal_frame,[SgId,SgFr,AnsTrie|ready],_).

get_new_id(Counter,Id):-
   %Get an Id and update the counter
   get_value(Counter,Id),
   NewId is Id + 1,
   set_value(Counter,NewId).
```

Figura 4.5: Predicado *tabled_call_check_insert/6* na versão base.

Posteriormente a termos verificado se um subgolo existe na *trie* e, por consequência, termos obtido a informação referente ao subgolo tabelado, invocamos o predicado *tabled_call_options/6* passando como argumento toda essa informação. A Figura 4.6 mostra o predicado *tabled_call_options/6*.

Sumariamente, a primeira cláusula deste predicado é responsável por, se o estado for diferente de complete, efectuar a suspensão da computação. A segunda cláusula é exclusiva para os nós geradores, isto é, nós cujo estado seja igual a *ready*. Aqui, os geradores actualizam o seu estado para *eval* e iniciam a computação através da chamada, no caso, por exemplo, do programa transformado da Figura 4.2, ao predicado *pathtrans/2*. Se o gerador for também o líder (isto é, se o *subgoal identifier* for igual a zero) então, depois de ter dado início à computação e de todos os nós terem concluído a primeira volta, vai construir a primeira lista de computações suspensas e acordar o primeiro elemento dessa lista (através da chamada ao predicado *wake_next_frozen_cp/0*). Na última cláusula do predicado *tabled_call_options/6* é onde todo o tipo de nó,

```
tabled_call_options(_,Call,SgId,_,AnsTrie,SgState) :-
   (SgState \= complete ->
      (SgId=0, SgState=ready ->
         freeze_and_fail_once(CP,recordz(leader_cp,CP,_)),
         complete_all,
         fail
      ;
         get_new_id(frozen_cp_counter,FzId),
         freeze_and_fail_once(CP,recordz(frozen_cp,[FzId,CP,AnsTrie|SgId],_)),
         wake_frozen_cp(FzId,Call)
      )
   ).
tabled_call_options(TransCall,_,SgId,SgFr,AnsTrie,ready) :-
   once(recorded(subgoal_frame,[SgId,SgFr,AnsTrie|ready],Ref)),
   erase(Ref),
   recordz(subgoal_frame,[SgId,SgFr,AnsTrie|eval],_),
   (
      call(TransCall)
   ;
      SgId=0,
      wake_next_frozen_cp
   ).
tabled_call_options(,Call,_,_,AnsTrie,_) :-
   consume_answer(Call,AnsTrie,0).
```

Figura 4.6: Predicado *tabled_call_options/6* na versão base.

independentemente do seu estado, consome todas as soluções encontradas até ao momento através da chamada ao predicado *consumer_answer/3*.

A suspensão é realizada ao nível da máquina de execução através do predicado *freeze_and_fail_once/2*, tal como apresentado na secção 3.2.2. O golo passado como argumento ao predicado *freeze_and_fail_once/2* varia consoante o tipo de nó. Se estivermos perante o nó líder da computação, o golo corresponde à inserção na estrutura *leader_cp* do ponto de escolha referente ao líder. Caso contrário, o golo corresponde à inserção - através do predicado *built-in recordz* de forma a termos as computações congeladas por ordem inversa à ordem que foram suspensas - na estrutura *frozen_cp* do *frozen identifier* (obtido antes de invocarmos o predicado *freeze_and_fail_once/2*), do ponto de escolha, da referência para a *answer trie* e do *subgoal identifier* do subgolo tabelado.

Como estamos a suspender a computação ao nível da máquina de execução, quando retomarmos uma computação suspensa esta irá executar o código que está depois da chamada ao predicado *freeze_and_fail_once/2* correspondente.

Quando o nó líder for acordado, isso significa que devemos completar todos os subgolos da computação, e para isso basta alterar o estado da computação de todas as entradas existentes na estrutura *subgoal_frame* para *complete* e, quando não houver mais entradas, remover toda a informação auxiliar utilizada para recuperar computações - nomeadamente eliminar todos os termos da estrutura *frozen_cp* e *leader_cp*. Esta

operação está a cargo do predicado *complete_all/0*, tal como podemos ver na Figura 4.7.

```
complete_all:-
   recorded(subgoal_frame,[SgId,SgFr,AnsTrie|eval],Ref),
   erase(Ref),
   recordz(subgoal_frame,[SgId,SgFr,AnsTrie|complete],_),
   fail.
complete_all:-
   eraseall(leader_cp),
   eraseall(frozen_cp).
```

Figura 4.7: Predicado *complete_all/0* na versão base.

Posteriormente a termos completado os subgolos em avaliação,voltamos à primeira cláusula do predicado *tabled_call_options/6* e aí provocamos a falha da execução (através da chamada ao predicado *built-in fail*) de forma a possibilitar que o líder execute a última chamada do predicado *tabled_call_options/6* e, assim, consuma as soluções encontradas.

Por outro lado, se o nó acordado não for o líder, é invocado o predicado *wake_frozen_cp/2*. Este predicado é composto por duas cláusulas, e recebe como argumentos o *frozen identifier* (FzId) e o subgolo tabelado (Figura 4.8). Na primeira cláusula vamos consumir todas as soluções encontradas até ao momento. Para isso precisamos de procurar na base de dados interna pelo termo existente na estrutura *frozen_cp* que unifique com o FzId, passado como argumento, de modo a obtermos a referência da *anwer tri*e do subgolo tabelado. Com essa referência e com o subgolo tabelado invocamos o predicado *consume_answer/3* (Figura 4.9). Como na versão base, sempre que se acorda um nó consumidor suspenso, vão ser sempre consumidas todas as soluções encontradas até ao momento, o predicado *consume_answer/3* é invocado com a referência da última solução consumida (terceiro argumento) igual a zero. Este predicado vai percorrer a *answer trie* do subgolo tabelado (através da função *trie_traverse/3*) e, para cada solução encontrada, devolve o respectivo termo (o termo é depois construído pelo predicado *trie_get_entry/2* tal como descrito no capítulo 2).

```
wake_frozen_cp(FzId,Call) :-
   recorded(frozen_cp,[FzId,_,AnsTrie|_],_),
   consume_answer(Call,AnsTrie,0).
wake_frozen_cp(_,_) :-
   wake_next_frozen_cp.
```

Figura 4.8: Predicado *wake_frozen_cp/2* na versão base.

Na segunda cláusula do predicado *wake_frozen_cp/2* vamos retomar a computação suspensa seguinte através da chamada ao predicado *wake_next_frozen_cp/0*. O predi-

```
consume_answer(Call,AnsTrie,LastRef):-
   trie_traverse(AnsTrie,LastRef,AnsRef),
   trie_get_entry(AnsRef,Call).
```

Figura 4.9: Predicado *consume_answer/3* na versão base.

cado *wake_next_frozen_cp/0*, primeiro verifica se ainda existem pontos de escolha na estrutura *waiting_cp* e, nesse caso, retiramos o próximo e, em seguida, acordamo-lo. Caso contrário, tentamos construir uma nova lista de pontos de escolha referentes a computações que devem ser acordadas. Para isso invocamos o predicado *build_waiting_list/0* que percorre todos os subgolos existentes na estrutura *subgoal_frame* em busca de subgolos com novas soluções, isto é, de subgolos cujo estado seja *eval_new*. Para subgolos nessas condições actualizamos o seu estado para *eval* e, em seguida, utilizamos o *subgoal identifier* para procurar por todos os pontos de escolha (CP) suspensos na estrutura *frozen_cp*. Posteriormente, inserimos esses pontos de escolha na estrutura *waiting_cp* para serem acordados e, em seguida, provocamos a falha da execução (através do predicado *built-in fail*) de forma a possibilitar a continuação da procura de todos os subgolos com novas soluções. Quando não houver mais subgolos a execução passa para a segunda cláusula do predicado *build_waiting_list/0* que simplesmente sucede. A Figura 4.10 mostra o predicado *buil_waiting_list/0* nesta versão.

```
build_waiting_list :-
   recorded(subgoal_frame,[SgId,SgFr,AnsTrie|eval_new],Ref),
   erase(Ref),
   recordz(subgoal_frame,[SgId,SgFr,AnsTrie|eval],_),
   recorded(frozen_cp,[_,CP,_|SgId],_),
   recorda(waiting_cp,CP,_),
   fail.
build_waiting_list.
```

Figura 4.10: Predicado *buil_waiting_list/0* na versão base.

Após construirmos a lista verificamos se existe alguma computação para ser retomada, isto é, se existe algum termo na estrutura *waiting_cp*. Se existir, removemos a primeira entrada e retomamos essa computação. Caso contrário, estão reunidas as condições para acordar o líder (o ponto de escolha do líder encontra-se guardado na estrutura *leader_cp*). Estas três hipóteses para retomar uma computação suspensa no predicado *wake_next_frozen_cp/0* - acordar um ponto de escolha existente na estrutura *waiting_cp*; criar uma lista de computações para serem retomadas e acordar o primeiro elemento; e acordar o líder quando não existem computações para serem retomadas - encontram-se detalhadas na Figura 4.11.

```
wake_next_frozen_cp:-
   %Resume next choice point, if any
   once(recorded(waiting_cp,CP,Ref)),
   !,
   erase(Ref),
   wake_choice_point(CP).
wake_next_frozen_cp:-
   %No more choice points to be resumed, so lets build a new list
   build_waiting_list,
   %Resume first choice point, if any
   once(recorded(waiting_cp,CP,Ref)),
   !,
   erase(Ref),
   wake_choice_point(CP).
wake_next_frozen_cp:-
   %No more computations to be resumed, so lets resume the leader
   once(recorded(leader_cp,LeaderCP,_)),
   wake_choice_point(LeaderCP).
```

Figura 4.11: Predicado *wake_next_frozen_cp/0* na versao base.

Desta forma, conseguimos implementar uma versão minimalista do mecanismo de tabulação do YapTab. O passo seguinte foi proceder à implementação progressiva de diversas optimizações de forma a melhorar o desempenho do nosso módulo.

4.3 Optimizações

Após termos uma versão base estável procedemos à implementação de diversas optimizações tendo em vista a melhoria do desempenho. Mais concretamente, foram realizadas optimizações ao nível do mecanismo de tabulação, nomeadamente, o consumo de soluções de forma incremental e a marcação de computações como completas de forma incremental; ao nível da estratégia de escalonamento, nomeadamente, a implementação da estratégia *batched scheduling*; e ao nível das estruturas de dados utilizadas para guardar a informação envolvida na computação, nomeadamente, o uso de predicados dinâmicos e o uso de vectores.

Em seguida, iremos descrever em detalhe cada uma destas optimizações e, sempre que possível, mostrar excertos de código onde será possível ver, de forma clara, as alterações efectuadas.

4.3.1 Consumir Soluções de Forma Incremental

A primeira optimização implementada foi modificar a forma como são consumidas as soluções. Na versão base, sempre que uma computação é retomada são consumidas

todas as soluções disponíveis. Com esta optimização pretende-se que, quando uma computação é retomada, sejam consumidas apenas as novas soluções encontradas a partir da última solução consumida na iteração anterior. Por exemplo, se tivermos na *answer trie* as soluções S1, S2 e S3, da primeira vez que a computação for retomada consumimos todas as soluções. Se, no seguimento da computação, forem adicionadas as soluções S4 e S5 à *answer trie*, da segunda vez que a computação for retomada vamos apenas consumir as soluções S4 e S5, que são as soluções que se encontram depois da última solução consumida na iteração anterior - a solução S3.

Tendo em vista a implementação desta optimização, foi necessário alterar a estrutura *frozen_cp* - responsável por guardar a informação necessária para recuperar computações suspensas. Este termo chave passou a guardar, para além do *frozen identifier* (FzId), do ponto de escolha que permite acordar uma computação suspensa (CP) e da referência da *answer trie* (AnsTrie), a referência da última solução consumida (LastCons) em lugar do *subgoal identifier* (SgId).

Esta optimização permitiu melhorar a forma como são inseridas novas soluções na *answer trie*. Antes desta optimização quando era encontrada uma nova solução, altura em que era invocado o predicado *new_answer/2*, era necessário garantir que o estado do subgolo respectivo passaria a *eval_new* caso a solução encontrada não estivesse já na *answer trie*. Com esta optimização a solução pode ser inserida sem efectuar qualquer tipo de teste sobre a *answer trie* já que as *tries* têm a capacidade de, quando uma solução é repetida, não a inserir novamente. Por consequência, deixou de ser necessário existir o estado *eval_new* e a inserção de soluções passou a ser bastante mais simples. A Figura 4.12 mostra o predicado *new_answer/2* após esta optimização.

```
new_answer(Call,AnsTrie) :-
   trie_put_entry(AnsTrie,Call,_),
   fail.
```

Figura 4.12: Predicado *new_answer/2* na versão com consumo incremental de soluções.

Com esta optimização, cada cláusula do predicado *tabled_call_options/6*, que podemos ver na Figura 4.13, abrange um estado específico da computação. A primeira cláusula passa apenas a ser executada por nós cujo o estado da computação seja igual a *complete*. Estes nós apenas têm que consumir todas as soluções existentes na *answer trie* respectiva.

A segunda cláusula do predicado *tabled_call_options/6* apenas é executada por nós consumidores, logo o estado da computação é igual a *eval*. Aí, através da invocação

do novo predicado *tabled_consumer_node/4* presente na Figura 4.13, começamos por consumir todas as soluções encontradas até ao momento para, quando procedermos à suspensão da computação, guardarmos a referência da última solução consumida (obtida através da função *trie_get_last_entry/2*). Caso não existam soluções para consumir, a referência da última solução fica igual a zero (valor definido por nós para indicar que ainda não foi consumida nenhuma solução).

```
tabled_call_options(,Call,_,_,AnsTrie,complete) :-
   consume_answer(Call,AnsTrie,0).
tabled_call_options(_,Call,SgId,_,AnsTrie,eval) :-
   get_new_id(frozen_cp_counter,FzId),
   tabled_consumer_node(Call,SgId,AnsTrie,FzId).
tabled_call_options(_,Call,0,_,AnsTrie,ready) :-
   freeze_and_fail_once(CP,recordz(leader_cp,CP,_)),
   !,
   complete_all,
   consume_answer(Call,AnsTrie,0).
tabled_call_options(TransCall,_,SgId,SgFr,AnsTrie,ready) :-
   once(recorded(subgoal_frame,[SgId,SgFr,AnsTrie|ready],Ref)),
   erase(Ref),
   recordz(subgoal_frame,[SgId,SgFr,AnsTrie|eval],_),
   (
      call(TransCall)
   ;
      (SgId = 0 ->
         wake_next_frozen_cp
      ;
         get_new_id(frozen_cp_counter,FzId),
         tabled_consumer_node(Call,SgId,AnsTrie,FzId)
      )
   ).


tabled_consumer_node(Call,_,AnsTrie,_):-
   consume_answer(Call,AnsTrie,0).
tabled_consumer_node(Call,SgId,AnsTrie,FzId):-
   %Update last consumed answer before suspend
   (trie_get_last_entry(AnsTrie,LastAns)->
      freeze_and_fail_once(CP,recordz(frozen_cp,[FzId,CP,AnsTrie|LastAns],_))
   ;
      freeze_and_fail_once(CP,recordz(frozen_cp,[FzId,CP,AnsTrie|0],_))
   ),
   wake_frozen_cp(FzId,Call).
```

Figura 4.13: Predicado *tabled_call_options/6* na versão com consumo incremental de soluções e o novo predicado *tabled_consumer_node/4* criado nesta versão.

O consumo de soluções e a actualização da referência da última solução consumida também se realizam quando uma computação de um nó consumidor, anteriormente suspenso, for retomada. Estas operações são realizadas antes de ser invocado o predicado *wake_next_froozen_cp/0* que, como foi referido anteriormente, pode acordar um ponto de escolha existente na estrutura *waiting_cp*; ou criar uma lista de computações para serem retomadas e acordar o primeiro elemento; ou acordar o líder caso não existem computações para serem retomadas. Logo, o predicado responsável por estas

operações é o predicado *wake_frozen_cp/2* que agora é invocado na segunda cláusula do novo predicado *tabled_consumer_node/4*. Podemos ver na Figura 4.14 o predicado *wake_frozen_code/2* após esta optimização.

```
wake_frozen_cp(FzId,Call) :-
   %We now consume starting from the last consumed answer
   recorded(frozen_cp,[FzId,_,AnsTrie|LastCons],_),
   consume_answer(Call,AnsTrie,LastCons).
wake_frozen_cp(FzId,_) :-
   %Update last consumed answer before waking next frozen cp
   once(recorded(frozen_cp,[FzId,CP,AnsTrie|_],Ref)),
   erase(Ref),
   trie_get_last_entry(AnsTrie,LastAns),
   recordz(frozen_cp,[FzId,CP,AnsTrie|LastAns],_)),
   wake_next_frozen_cp.
```

Figura 4.14: Predicado *wake_frozen_code/2* na versão com consumo incremental de soluções.

Na versão base não era possível guardar as computações que deviam ser retomadas por ordem decrescente na estrutura *waiting_cp*, isto é, da última computação suspensa para a primeira, pois o controle sobre se existiam novas soluções era realizado através da estrutura *subgoal_frame*. Por exemplo, se as computações com *frozen identifier* (FzId) igual a 10, 7 e 4 tivessem novas soluções não era possível garantir que as computações fossem retomadas por essa ordem (do FzId 10 para o FzId 4). Isto acontece porque, como a inserção está dependente do *subgoal identifier* (SgId), se o FzId 4 e 10 pertencerem ao subgolo com SgId 2 e o FzId 7 pertencer ao SgId 3 então, o FzId 4 e 10 seriam inseridos na estrutura *waiting_cp* antes do FzId 7. Com esta optimização, o controle sobre se existem novas soluções passou a ser realizado através da estrutura *frozen_cp*. Logo, conseguimos guardar as computações que devem ser retomadas (mais concretamente os pontos de escolha) na estrutura *waiting_cp* por ordem decrescente. A construção da lista de computações suspensas, que se mantém a cargo do predicado *build_waiting_list/0*, tornou-se assim bastante mais simples - em vez de consultarmos as estruturas *subgoal_frame* e *frozen_cp*, agora basta consultar a estrutura *frozen_cp* - como podemos ver na Figura 4.15.

Na terceira cláusula do predicado *tabled_call_options/6* efectuamos a suspensão do líder, que é o subgolo com *subgoal_identifier* igual a zero e com o estado da computação igual a *ready*. Por consequência, o código que deve ser executado quando o líder for retomado, isto é, completar todos os subgolos da computação e consumir todas as soluções encontradas, também se encontra nesta cláusula.

Por fim, na quarta cláusula do predicado *tabled_call_options/6* os nós geradores, cujo

```
build_waiting_list :-
   %We now check if the last consumed answer
   %is different from the last found answer
   recorded(frozen_cp,[FzId,CP,AnsTrie|LastCons],_),
   trie_get_last_entry(AnsTrie,LastAns),
   LastAns \== LastCons,
   recorda(waiting_cp,CP,_),
   fail.
build_waiting_list.
```

Figura 4.15: Predicado *build_waiting_list/0* na versão com consumo incremental de soluções.

estado tem que ser obrigatoriamente igual a *ready*, actualizam o seu estado para *eval* e dão inicio à computação. Quando todos os nós terminarem a primeira volta, verifica-se se o nó gerador é também líder. Se for, então é invocado o predicado *wake_next_frozen_cp/0* para construir a primeira lista de computações suspensas que devem ser retomadas. Caso contrário, é invocado o predicado *tabled_consumer_node/4* pois esse nó vai, a partir deste momento, passar a ser um nó consumidor.

O consumo de soluções de forma incremental é uma das optimizações mais importantes pois permite melhorar o desempenho do nosso módulo na maioria dos casos. Em seguida iremos abordar uma outra optimização - completar computações de forma incremental - que também consegue melhorar o desempenho em alguns casos.

4.3.2 Completar Computações de Forma Incremental

O próximo passo no desenvolvimento do nosso módulo externo foi a implementação de uma optimização que permite completar computações de forma incremental. Esta nova optimização envolve a alteração da forma como é escolhido o nó líder durante a computação. Tomemos como exemplo o programa da Figura 4.16. Se efectuássemos a consulta `path(a,Z)`, na versão base, o nó líder do grafo seria o subgolo *path(a,Z)*, que corresponde ao subgolo de topo da computação, isto é, com *subgoal identifier* igual a zero. Por consequência, todas as computações só seriam marcadas como completas no final da execução.

Com esta optimização, para além de deixarmos de ter necessariamente apenas um nó líder, o líder pode ir variando ao longo da computação. No caso concreto da consulta `path(a,Z)`, e numa fase inicial, todas as chamadas aos subgolos *path(a,Z)*, *path(b,Z)* e *path(c,Z)* são líderes. Mas, como existe um ciclo no grafo, devido ao nó *c* ter uma ligação para o nó *b*, o subgolo *path(c,Z)* vai deixar de ser líder pois ao

```
:-yap_flag(tabling_mode,local).
:-table path/2.

path(X,Z) :-
   edge(X,Y),
   path(Y,Z).
path(X,Z) :-
   edge(X,Z).

edge(a,b).
edge(b,c).
edge(c,b).
```

Figura 4.16: Programa para encontrar caminhos num grafo onde existe um ciclo.

chamar *path(b,Z)* irá criar uma dependência para um subgolo anterior. Logo, sempre que existem dependências entre os nós da computação pode existir alterações nos nós líderes. Mais concretamente, todos os nós com *subgoal identifier* maior que o *subgoal identifier* do nó para o qual existe uma dependência não podem ser líderes - neste caso o subgolo *path(c,Z)* deixará de ser líder. Quando a computação relativa ao nó líder *path(b,Z)* estiver completamente explorada, este, para além de completar a sua computação e a dos nós que a ele estão ligados (*path(c,Z)* no exemplo), vai auto excluir-se da lista de líderes. Resta então apenas um líder na lista - o subgolo *path(a,Z)* - que, devido à computação a partir do nó *b* já estar totalmente explorada e, por consequência, marcada como completa, apenas tem que completar a sua própria computação.

Esta optimização levou a que tivessem de ser efectuadas alterações nos predicados *tabled_call_check_insert/6* e *tabled_call_options/6* tal como podemos ver nas Figuras 4.17 e 4.18.

De forma a conseguirmos lidar com a existência de diferentes líderes, que podem variar ao longo da computação, criamos duas novas estruturas auxiliares: a estrutura *leader*; e a estrutura *back_leader*. A estrutura *leader* é utilizada para guardar os *subgoal identifiers* dos nós líderes. O *subgoal identifier* (SgId) de um nó é inserido neste termo chave quando o subgolo é chamado pela primeira vez - esta operação é realizada na segunda cláusula do predicado *tabled_call_check_insert/6*. Quando um subgolo for chamado repetidamente dando origem a nós consumidores, isto é, quando o seu estado é igual a *eval*, temos que remover todos os *subgoals identifiers* maiores que o SgId deste nó da estrutura *leader*. Esta remoção de entradas é realizada pelo predicado *update_leader/1*, invocado na primeira cláusula do predicado *tabled_call_check_insert/6*, que podemos ver na Figura 4.17.

Devido ao facto de um nó poder deixar de ser líder a meio de uma computação,

```
tabled_call_check_insert(SubTrie,Call,SgId,SgFr,AnsTrie,SgState):-
   trie_check_entry(SubTrie,Call,SgFr),
   %The subgoal already exists, so get subgoal info
   recorded(subgoal_frame,[SgId,SgFr,AnsTrie|SgState],_),
   (SgState \= complete ->
      %New subgoal dependency, so update leader/1 info
      update_leader(SgId)
   ;
      true
   ),
   !.
tabled_call_check_insert(SubTrie,Call,SgId,SgFr,AnsTrie,SgState):-
   %The subgoal is new, so insert subgoal info
   trie_put_entry(SubTrie,Call,SgFr),
   trie_open(AnsTrie),
   get_new_id(subgoal_frame_counter,SgId),
   recordz(subgoal_frame,[SgId,SgFr,AnsTrie|ready],_),
   %By default now a new subgoal is a leader subgoal
   recorda(leader,SgId,_).

update_leader(SgId):-
   %Remove all subgoals with an identifier greater than
   %the new subgoal dependecy
   recorded(leader,LeaderId,Ref),
   (LeaderId > SgId ->
      erase(Ref),
      fail
   ;
      !
   ).
```

Figura 4.17: Predicados *tabled_call_check_insert/6* e *update_leader/1* na versão com detecção incremental de subgolos completos.

houve a necessidade de guardar o ponto de escolha referente aos líderes que iniciam o processo de acordar computações suspensas numa outra estrutura de dados pois, caso contrário, como pode haver uma chamada ao predicado *update_leader/1*, a informação referente ao último líder que iniciou o processo de acordar computações suspensas pode deixar de existir e tornar inviável acordar mais tarde o ex-líder correcto, quando não existirem mais computações suspensas por acordar. A estrutura criada para esse efeito foi o *back_leader* que guarda a referência para o ponto de escolha associado aos líderes que iniciam o processo de acordar computações suspensas e o respectivo *frozen_identifier* (FzId). A inclusão do FzId vai permitir garantir que não são acordados nós consumidores que não pertençam ao líder que iniciou o processo de acordar computações suspensas.

A primeira e a segunda cláusula do predicado *tabled_call_options/6* mantêm-se inalteradas. No entanto, o predicado *tabled_consumer_node/4*, invocado na segunda cláusula do predicado *tabled_call_options/6* para proceder à suspensão da computação dos nós consumidores, passa agora a guardar as computações suspensas sempre no topo da estrutura de dados *frozen_cp*, logo a última computação suspensa fica a ser a primeira

```
tabled_call_options(_,Call,_,_,AnsTrie,complete):-
   consume_answer(Call,AnsTrie,0).
tabled_call_options(_,Call,SgId,_,AnsTrie,eval):-
   get_new_id(frozen_cp_counter,FzId),
   tabled_consumer_node(Call,SgId,AnsTrie,FzId).
tabled_call_options(TransCall,Call,SgId,SgFr,AnsTrie,ready):-
   once(recorded(subgoal_frame,[SgId,SgFr,AnsTrie|ready],Ref)),
   erase(Ref),
   recordz(subgoal_frame,[SgId,SgFr,AnsTrie|eval],_),
   get_new_id(frozen_cp_counter,FzId),
   (
     call(TransCall)
   ;
     check_if_is_leader(Call,SgId,AnsTrie,FzId)
   ).

tabled_consumer_node(Call,_,AnsTrie,_):-
   consume_answer(Call,AnsTrie,0).
tabled_consumer_node(Call,SgId,AnsTrie,FzId):-
   %Update last consumed answer before suspend
   (trie_get_last_entry(AnsTrie,LastAns)->
      freeze_and_fail_once(CP,recorda(frozen_cp,[FzId,CP,AnsTrie|LastAns],_))
   ;
      freeze_and_fail_once(CP,recorda(frozen_cp,[FzId,CP,AnsTrie|0],_))
   ),
   wake_frozen_cp(FzId,Call).
```

Figura 4.18: Predicado *tabled_call_options/6* na versão com detecção incremental de subgolos completos.

entrada da estrutura de dados *frozen_cp* (para isto ser possível passamos a utilizar o predicado *built-in recorda/3* em vez do predicado *built-in recordz/3*). Esta ordenação será importante para o predicado *build_waiting_list/0* como veremos mais à frente.

Quando um nó consumidor é acordado, vai ser invocado o predicado *wake_frozen_cp/2* que podemos ver na Figura 4.19. Este predicado é em tudo semelhante ao predicado existente até esta optimização. A única diferença prende-se com o facto de, após se proceder à actualização da última solução consumida, não ser invocado o predicado *wake_next_frozen_cp/0* (que com esta optimização deixa de existir). Em vez disso, verificamos directamente se existe alguma computação para ser retomada. Se existir, e o ponto de escolha pertencer ao último líder que iniciou o processo de acordar computações suspensas, guardado na estrutura *back leader*, então acordamos esse ponto de escolha. Caso contrário, isto é, se não existir nenhum ponto de escolha na lista ou o ponto de escolha não pertencer ao último líder que iniciou o processo de acordar computações suspensas, então acordamos o ponto de escolha guardado na estrutura *back leader*.

Como com esta optimização podemos ter diferentes líderes, a terceira cláusula do predicado *tabled_call_options/6* existente na versão anterior deixou de existir e a quarta cláusula (que é nesta versão a terceira cláusula) sofreu algumas modificações.

```
wake_frozen_cp(FzId,Call) :-
   recorded(frozen_cp,[FzId,_,AnsTrie,_|LastCons],_),
   consume_answer(Call,AnsTrie,LastCons).
wake_frozen_cp(FzId,_) :-
   once(recorded(frozen_cp,[FzId,CP,AnsTrie,SgId|_],Ref)),
   erase(Ref),
   trie_get_last_entry(AnsTrie,LastAns),
   recordz(frozen_cp,[FzId,CP,AnsTrie,SgId|LastAns],_)),
   %We now resume the youngest choice point between
   %the back leader cp and the next frozen cp, if any
   once(recorded(back_leader,[LeaderFzId|BackCP],_)),
   (once(recorded(waiting_cp,[LeaderFzId|NextCP],NextRef)) ->
      erase(NextRef),
      wake_choice_point(NextCP)
   ;
      wake_choice_point(BackCP)
   ).
```

Figura 4.19: Predicado *wake_frozen_cp/2* na versão com detecção incremental de subgolos completos.

Na terceira cláusula do predicado *tabled_call_options/6*, após um nó gerador dar início à computação, é agora invocado um novo predicado denominado *check_if_is_leader/4*, que podemos ver na Figura 4.20. Aí verificamos se esse nó é líder através da estrutura *leader* (em vez de verificarmos se o *subgoal identifier* é igual a zero). Caso o *subgoal identifier* (SgId) do gerador exista nessa estrutura, é criada a primeira lista de computações suspensas que devem ser acordadas através da chamada do predicado *build_waiting_list/2* que, com esta optimização, passa a receber dois argumentos e passa a guardar na estrutura *waiting_cp*, para além do ponto de escolha do nó consumidor que vai ser acordado, o *frozen identifier* (FzId) do líder corrente. Como só queremos percorrer os nós consumidores que estão relacionados com esse líder (isto é, que foram suspensos depois da chamada a esse nó líder), necessitamos de especificar concretamente o seu FzId e devemos começar a percorrer a estrutura *frozen_cp* do último FzId até ao FzId imediatamente a seguir ao nó líder. A Figura 4.21 mostra o predicado *build_waiting_list/2* nesta optimização.

Se existirem computações para serem retomadas, então retira-se a primeira computação da lista de computações suspensas e, em seguida, invoca-se o novo predicado *suspend_leader_cp/4* que podemos ver na Figura 4.22. Este predicado é responsável por realizar a suspensão do líder, através do predicado *freeze_and_fail_once/2* cujo golo, passado como argumento, permite guardar na estrutura *back_leader* o FzId do líder e o ponto de escolha que permite retomar a computação. Por consequência, o código executado quando o líder é acordado também se encontra nesta cláusula. Mais concretamente, quando a computação do líder é retomada temos, em primeiro lugar, que verificar se o nó continua a ser líder. Se continuar, isto é, se o SgId ainda existir

```
check_if_is_leader(Call,SgId,AnsTrie,FzId):-
   (recorded(leader,SgId,LeaderRef) ->
      get_value(frozen_cp_counter,CounterFzId),
      LastFzId is CounterFzId - 1,
      build_waiting_list(FzId,LastFzId),
      (once(recorded(waiting_cp,[FzId|ConsumerCP],WaitRef)) ->
         erase(WaitRef),
         (
           suspend_leader_cp(Call,SgId,AnsTrie,FzId)
         ;
           wake_choice_point(ConsumerCP)
         ;
           fail
         )
      ;
         get_value(subgoal_frame_counter,LastSgId),
         complete_scc(SgId,LastSgId),
         erase(LeaderRef),
         set_value(subgoal_frame_counter,SgId),
         set_value(frozen_cp_counter,FzId),
         abolish_frozen,
         consume_answer(Call,AnsTrie,0)
      )
   ;
      tabled_consumer_node(Call,SgId,AnsTrie,FzId)
   ).
```

Figura 4.20: Predicado *check_if_is_leader/4* criado na versão com detecção incremental de subgolos completos.

```
build_waiting_list(LeaderFzId,LeaderFzId):-
   !.
build_waiting_list(LeaderFzId,FzId):-
   (recorded(frozen_cp,[FzId,CP,AnsTrie|LastCons],_),
    trie_get_last_entry(AnsTrie,LastAns), LastAns \= LastCons ->
      recordz(waiting_cp,[LeaderFzId|CP],_)
   ;
      true
   ),
   PreviousFzId is FzId - 1,
   build_waiting_list(LeaderFzId,PreviousFzId).
```

Figura 4.21: Predicado *build_waiting_list/2* na versão com detecção incremental de subgolos completos.

na estrutura *leader*, então construímos a lista de computações suspensas e, caso exista pelo menos um elemento, acordamos o primeiro elemento dessa lista. Caso não exista nenhuma computação para ser retomada, o líder está em condições de completar o seu *Strongly Connected Component* (SCC). A operação de completar é realizada pelo predicado *complete_scc/2* (Figura 4.23) que deverá completar os nós a partir do SgId do líder (inclusivé). Para além disso, deverá remover a informação associada a cada um dos nós completos, nomeadamente a informação que permite recuperar computações suspensas. Para isso ser possível, houve a necessidade de guardar o SgId na estrutura *frozen_cp* de modo a identificar os nós consumidores de um determinado subgolo. A operação de remover a informação da estrutura auxiliar *frozen_cp* é realizada pelo predicado *remove_frozen_cps/1*, invocado pelo predicado *complete_scc/2*, que podemos ver na Figura 4.23.

```
suspend_leader_cp(Call,SgId,AnsTrie,FzId):-
   freeze_and_fail_once(CP,recorda(back_leader,[FzId|CP],_)),
   (recorded(leader,SgId,LeaderRef) ->
      get_value(frozen_cp_counter,CounterFzId),
      LastFzId is CounterFzId - 1,
      build_waiting_list(FzId,LastFzId),
      (once(recorded(waiting_cp,[FzId|ConsumerCP],WaitRef))->
         erase(WaitRef),
         wake_choice_point(ConsumerCP)
      ;
         get_value(subgoal_frame_counter,LastSgId),
         complete_scc(SgId,LastSgId),
         erase(LeaderRef),
         recorded(back_leader,[FzId|_],BackRef),
         erase(BackRef),
         set_value(subgoal_frame_counter,SgId),
         set_value(frozen_cp_counter,FzId),
         abolish_frozen,
         consume_answer(Call,AnsTrie,0)
      )
   ;
      recorded(back_leader,[FzId|_],BackRef),
      erase(BackRef),
      tabled_consumer_node(Call,SgId,AnsTrie,FzId)
   ).
```

Figura 4.22: Predicado *suspend_leader_cp/4* criado na versão com detecção incremental de subgolos completos.

Após completar a computação e antes de consumir todas as soluções encontradas, o líder vai remover ainda a sua entrada das estruturas *leader* e *back_leader*, e, de forma a minimizar a memória utilizada, actualizar o contador de subgolos (o *subgoal_frame_counter*), o contador de computações suspensas (o *frozen_cp_counter*) e invocar o predicado *abolish_frozen/0*. O predicado *abolish_frozen/0* invoca o predicado *abolish_frozen_choice_points/1*, implementado ao nível da máquina, que recoloca os *freeze_registers*, mais concretamente o *B_FZ* para a *stack*, o *H_FZ* para a *heap* e o

```
complete_scc(LastSgId,LastSgId):-
   !.
complete_scc(SgId,LastSgId):-
   (recorded(subgoal_frame,[SgId,SgFr,AnsTrie|eval],Ref) ->
      erase(Ref),
      recordz(subgoal_frame,[SgId,SgFr,AnsTrie|complete],_),
      remove_frozen_cps(SgId)
   ;
      true
   ),
   NextSgId is SgId + 1,
   complete_scc(NextSgId,LastSgId).

remove_frozen_cps(SgId):-
   recorded(frozen_cp,[_,_,_SgId|_],Ref),
   erase(Ref),
   fail.
remove_frozen_cps(_).
```

Figura 4.23: Predicados *complete_scc/2* e *remove_frozen_cps/1* na versão com detecção incremental de subgolos completos.

TR_FZ para a trilha, a apontar para a computação que foi suspensa antes do líder.

Caso o nó durante o decorrer da computação deixe de ser líder, então, quando a computação for retomada, vai ser removida a entrada na estrutura *back_leader* referente ao seu FzId e invocado o predicado *tabled_consumer_node/4* pois, a partir deste momento, este nó passou a ser um nó consumidor, logo tem que ser suspenso como tal.

Quando é invocado pela primeira vez o predicado *suspend_leader_cp/4*, a execução desse predicado vai falhar visto que, para suspendermos a execução do líder, invocamos o predicado *freeze_and_fail_once/2*. Nessa altura, é invocada como alternativa o predicado *wake_choice_point/1* que vai retomar a primeira computação da primeira lista de computações suspensas criada antes da invocação do predicado *suspend_leader_cp/4*.

Após o gerador ter completado a computação e consumido todas as soluções encontradas, no predicado *suspend_leader_cp/4*, a execução vai suceder. Nessa altura, é executada a alternativa que ainda se encontra por explorar no predicado *check_if_is_leader/4*. Esta alternativa é o predicado *built-in fail* que vai permitir efectuar o *backtracking*.

Se a invocação do predicado *build_waiting_list/2*, no predicado *check_if_is_leader/4*, tiver criado uma lista de computações suspensas vazia, isto é, se não existirem computações para serem retomadas, então estamos em condições de completar a execução do líder e dos nós a ele associados. De realçar que, nesta situação, a execução do líder não tem que ser suspensa. Logo, após o processo de completar computações estar concluído, apenas não se procede à remoção da sua entrada na estrutura *back_leader* visto esta não ter sido criada.

Por fim, se no predicado *check_if_is_leader/4* verificarmos que o nó não é líder, então é invocado o predicado *tabled_consumer_node/4* para suspender a execução desse nó pois estamos perante um nó consumidor.

Após a implementação desta optimização, passamos à implementação de outra estratégia de escalonamento - a estratégia *batched scheduling* - que iremos descrever em seguida.

4.3.3 A Estratégia *Batched Scheduling*

Como foi referido anteriormente, a estratégia de escalonamento pode ter um impacto significativo no desempenho e influenciar a ordem das soluções encontradas. Por isso, decidimos experimentar diferentes estratégias e implementar a estratégia de escalonamento *batched scheduling*. Esta estratégia, como foi mencionado na secção 2.2, dá prioridade à continuação da execução, seguindo-se o *backtracking* e, por fim, o consumir de soluções e o marcar computações como completas.

A implementação da estratégia *batched scheduling* teve como base a versão final do módulo com a estratégia *local scheduling*, isto é, com a implementação do consumo de soluções de forma incremental e a marcação de computações como completas de forma incremental.

Para suportar esta estratégia, começamos por modificar o predicado *new_answer/2*. Na parte de cima da Figura 4.24 temos o predicado *new_answer/2* na estratégia *local scheduling* e, na parte de baixo, temos o mesmo predicado na estratégia *batched_scheduling*. Na estratégia *batched scheduling* após introduzirmos uma nova solução na *answer trie*, a execução deve suceder e não falhar, como acontece em *local scheduling*, de forma a permitir que a solução encontrada seja imediatamente propagada para todos os subgolos. Por essa razão, após a inserção de uma nova solução não é invocado o predicado *built-in fail*. Para além disso, como a estratégia *batched scheduling* dá prioridade à continuação da execução, é necessário controlar se a solução já está guardada na *answer trie* pois, caso contrário, podem ser retornadas soluções repetidas ao utilizador.

Foi também necessário efectuar alterações no predicado *check_if_is_leader/4*. Após o líder marcar como completo o seu *Strongly Connected Component* (SCC) não necessita de consumir todas as soluções encontradas visto que, na altura da inserção, a solução já foi propagada. Em vez disso, a execução deve falhar. A outra alteração prende-

```
%Predicate new_answer/2 in local scheduling
new_answer(Call,AnsTrie) :-
   trie_put_entry(AnsTrie,Call,_),
   fail.

- - - - - - - - - - - - - - - - - - - - - - - - -

%Predicate new_answer/2 in batched scheduling
new_answer(Call,AnsTrie) :-
   (trie_check_entry(AnsTrie,Call,_)->
      fail
   ;
      trie_put_entry(AnsTrie,Call,_)
   ).
```

Figura 4.24: Predicado *new_answer/2* na versão com a estratégia *local scheduling* (em cima) e na versão com a estratégia *batched scheduling* (em baixo).

se com o facto de, caso o nó gerador não seja líder, a execução falhar ao invés de suspender a execução desse nó como consumidor (através da invocação do predicado *tabled_consumer_node/4*). Na parte de cima da Figura 4.25 temos o predicado *check_if_is_leader/4* na estratégia *local scheduling* e, na parte de baixo, temos o mesmo predicado na estratégia *batched_scheduling*.

Por fim, modificamos o predicado *suspend_leader_cp/4* onde, tal como acontece no predicado *check_if_is_leader/4*, o líder após marcar como completo o seu SCC não necessita de consumir todas as soluções encontradas. Em vez disso, a execução deve falhar. Caso o nó durante a computação deixe de ser líder, a execução deve falhar em vez de ocorrer a suspensão desse nó como consumidor (através da invocação do predicado *tabled_consumer_node/4*). Na parte de cima da Figura 4.26 temos o predicado *suspend_leader_cp/4* na estratégia *local scheduling* e, na parte de baixo, temos o mesmo predicado na estratégia *batched_scheduling*.

Com a implementação da estratégia *batched scheduling* no nosso módulo, conseguimos reproduzir quase na totalidade o sistema YapTab. Por isso, as próximas optimizações realizadas, que iremos abordar em seguida, são ao nível das estruturas de dados utilizadas para guardar a informação envolvida na computação.

4.3.4 Estruturas de Dados Alternativas

A estrutura de dados utilizada inicialmente para guardar a informação relativa à computação foi, como já foi referido, a base de dados interna do Yap Prolog. No entanto esta estrutura de dados sofre de limitações quando utilizada como suporte a

```
%Predicate check_if_is_leader/4 in local scheduling
check_if_is_leader(Call,SgId,AnsTrie,FzId):-
   (recorded(leader,SgId,LeaderRef) ->
      get_value(frozen_cp_counter,CounterFzId),
      LastFzId is CounterFzId - 1,
      build_waiting_list(FzId,LastFzId),
      (once(recorded(waiting_cp,[FzId|ConsumerCP],WaitRef)) ->
         erase(WaitRef),
         (
           suspend_leader_cp(Call,SgId,AnsTrie,FzId)
         ;
           wake_choice_point(ConsumerCP)
         ;
           fail
         )
      ;
         get_value(subgoal_frame_counter,LastSgId),
         complete_scc(SgId,LastSgId),
         erase(LeaderRef),
         set_value(subgoal_frame_counter,SgId),
         set_value(frozen_cp_counter,FzId),
         abolish_frozen,
         consume_answer(Call,AnsTrie,0)
      )
   ;
      tabled_consumer_node(Call,SgId,AnsTrie,FzId)
   ).
```

```
%Predicate check_if_is_leader/4 in batched scheduling
check_if_is_leader(Call,SgId,AnsTrie,FzId):-
   (recorded(leader,SgId,LeaderRef) ->
      get_value(frozen_cp_counter,CounterFzId),
      LastFzId is CounterFzId - 1,
      build_waiting_list(FzId,LastFzId),
      (once(recorded(waiting_cp,[FzId|ConsumerCP],WaitRef)) ->
         erase(WaitRef),
         (
           suspend_leader_cp(Call,SgId,AnsTrie,FzId)
         ;
           wake_choice_point(ConsumerCP)
         ;
           fail
         )
      ;
         get_value(subgoal_frame_counter,LastSgId),
         complete_scc(SgId,LastSgId),
         erase(LeaderRef),
         set_value(subgoal_frame_counter,SgId),
         set_value(frozen_cp_counter,FzId),
         abolish_frozen,
         fail
      )
   ).
```

Figura 4.25: Predicado *check_if_is_leader/4* na versão com a estratégia *local scheduling* (em cima) e na versão com a estratégia *batched scheduling* (em baixo).

```
%Predicate suspend_leader_cp/4 in local scheduling
suspend_leader_cp(Call,SgId,AnsTrie,FzId):-
   freeze_and_fail_once(CP,recorda(back_leader,[FzId|CP],_)),
   (recorded(leader,SgId,LeaderRef) ->
      get_value(frozen_cp_counter,CounterFzId),
      LastFzId is CounterFzId - 1,
      build_waiting_list(FzId,LastFzId),
      (once(recorded(waiting_cp,[FzId|ConsumerCP],WaitRef))->
         erase(WaitRef),
         wake_choice_point(ConsumerCP)
      ;
         get_value(subgoal_frame_counter,LastSgId),
         complete_scc(SgId,LastSgId),
         erase(LeaderRef),
         recorded(back_leader,[FzId|LeaderCP],BackRef),
         erase(BackRef),
         set_value(subgoal_frame_counter,SgId),
         set_value(frozen_cp_counter,FzId),
         abolish_frozen,
         consume_answer(Call,AnsTrie,0)
      )
   ;
      recorded(back_leader,[FzId|_],BackRef),
      erase(BackRef),
      tabled_consumer_node(Call,SgId,AnsTrie,FzId)
   ).
```

```
%Predicate suspend_leader_cp/4 in batched scheduling
suspend_leader_cp(Call,SgId,AnsTrie,FzId):-
   freeze_and_fail_once(CP,recorda(back_leader,[FzId|CP],_)),
   (recorded(leader,SgId,LeaderRef) ->
      get_value(frozen_cp_counter,CounterFzId),
      LastFzId is CounterFzId - 1,
      build_waiting_list(FzId,LastFzId),
      (once(recorded(waiting_cp,[FzId|ConsumerCP],WaitRef))->
         erase(WaitRef),
         wake_choice_point(ConsumerCP)
      ;
         get_value(subgoal_frame_counter,LastSgId),
         complete_scc(SgId,LastSgId),
         erase(LeaderRef),
         recorded(back_leader,[FzId|LeaderCP],BackRef),
         erase(BackRef),
         set_value(subgoal_frame_counter,SgId),
         set_value(frozen_cp_counter,FzId),
         abolish_frozen,
         fail
      )
   ;
      recorded(back_leader,[FzId|_],BackRef),
      erase(BackRef),
      fail
   ).
```

Figura 4.26: Predicado *suspend_leader_cp/4* na versão com a estratégia *local scheduling* (em cima) e na versão com a estratégia *batched scheduling* (em baixo).

uma implementação da técnica da tabulação a alto nível na medida em que, durante a execução, existe muita informação que é passível de actualização. No caso concreto da nossa implementação o estado da computação (guardado na estrutura *subgoal_frame*) e, no caso das versões com optimizações, a última resposta consumida (guardada na estrutura *frozen_cp*) e a informação referente ao líder (guardada nas estruturas *leader* e *back_leader*) são as informações que durante a execução do programa são actualizadas. Destas três, as operações mais críticas são a actualização do estado da computação e a actualização da última resposta consumida na medida em que, para fazer a respectiva actualização temos que primeiro proceder a uma consulta para obter a informação, em seguida apagar a entrada respectiva da base de dados interna (através da referência retornada pela consulta) e, finalmente, voltar a inserir essa informação na base de dados (no caso da actualização do líder só temos que remover a informação dos nós que deixaram de ser líderes, o que é mais pacifico). Isto deve-se ao facto de não existir no Yap Prolog um predicado *built-in* que efectue a actualização de uma entrada na base de dados interna, logo o custo associado a uma operação de actualização é maior.

Um dos problemas com que nos deparamos quando estávamos a desenvolver a versão base foi o problema da variável que contêm a referência para a *answer trie* de um subgolo (AnsTrie) e que é passada como argumento ao predicado *tabled_call_options/6*. A primeira instanciação dessa variável é realizada na segunda cláusula do predicado *tabled_call_check_insert/6*, altura em que o subgolo surge pela primeira vez (logo é um nó gerador) e, por consequência, toda a sua informação é guardada na estrutura *subgoal_frame*. A execução passa então para a primeira cláusula do predicado *tabled_call_options/6* onde, no caso do subgolo não ser o nó líder, vai ser realizada a suspensão da computação e, subsequentemente, o armazenamento na estrutura *frozen_cp* da informação necessária para recuperar a computação. Posteriormente, a execução vai para a segunda cláusula do predicado *tabled_call_options/6* onde o nó gerador actualiza o seu estado de *ready* para *eval* antes de iniciar a computação. Essa actualização vai fazer com que, quando a execução desse subgolo for retomada na primeira cláusula do predicado *tabled_call_options/6*, a atribuição da variável AnsTrie esteja corrompida. Isto acontece porque cada entrada tem uma referência única logo quando voltamos a inserir a informação esta passa a ter uma nova referência na base de dados.

Mal nos deparamos com este problema prontamente tentamos solucioná-lo. A primeira abordagem passou por guardar a referência para a *answer trie* também na estrutura *frozen_cp*. Esta solução levou a que tivéssemos que associar um número único a cada entrada dessa estrutura - o *frozen identifier* - pois podemos ter mais do que uma

entrada referente a um subgolo nesta estrutura. Logo, como não podemos correr o risco de estar a retomar uma computação errada, o número único pareceu-nos a melhor solução. De realçar que, nas versões com optimizações, a presença da referência para a *answer trie* na estrutura *frozen_cp* é crucial pois essa referência é necessária para construir a lista de computações suspensas (mais concretamente para ir à *answer trie* buscar a última solução).

Uma outra alternativa para solucionar este problema passou por utilizar predicados dinâmicos para guardar a informação envolvida na computação. Os predicados dinâmicos, como foi referido na secção 3.2, permitem ao programador alterar a base de dados das cláusulas durante o decorrer da computação. Todas as estruturas criadas passam agora a ser implementadas com predicamos dinâmicos. Por exemplo, a estrutura *subgoal_frame* à qual estavam associado quatro termos - o *subgoal identifier* (SgId), a referência do subgolo na *trie* (SgFr), o identificador da *answer trie* (AnsTrie) e o estado da computação (State) - passa agora a ser definida por um predicado dinâmico de aridade quatro através da declaração *dynamic(subgoal_frame/4)*.

Com a utilização de predicados dinâmicos, a remoção de informação fica mais simples, pois para remover uma entrada de um predicado dinâmico utilizamos directamente o predicado *built-in retract/1* que permite efectuar a remoção de predicados dinâmicos. A Figura 4.27 mostra o predicado *wake_frozen_cp/2* com predicados dinâmicos.

```
wake_frozen_cp(FzId,Call) :-
   frozen_cp(FzId,_,AnsTrie,_,LastCons),
   consume_answer(Call,AnsTrie,LastCons).
wake_frozen_cp(FzId,_) :-
   once(retract(frozen_cp(FzId,CP,AnsTrie,SgId,_))),
   trie_get_last_entry(AnsTrie,LastAns),
   assertz(frozen_cp(FzId,CP,AnsTrie,SgId,LastAns)),
   once(back_leader(LeaderFzId,BackCP)),
   (once(waiting_cp(LeaderFzId,NextCP)) ->
      retract(waiting_cp(LeaderFzId,NextCP)),
      wake_choice_point(NextCP)
    ;
      wake_choice_point(BackCP)
    ).
```

Figura 4.27: Predicado *wake_frozen_cp/2* com predicados dinâmicos.

No entanto, os predicados dinâmicos também não conseguem preserver a atribuição da variável AnsTrie. Por essa razão, e de forma a evitar guardar em duas estruturas de dados distintas a referência para a *answer trie*, optamos por utilizar vectores em conjunto com os predicados dinâmicos ou com a base de dados interna do Yap Prolog. O tamanho dos vectores, cujo tamanho é aumentado mediante as necessidades, para guardar a informação passível de actualização. Nomeadamente, vamos guardar o

estado de cada subgolo num vector denominado *subgoal_frame*, e a última solução consumida num outro vector designado *frozen_cp*. Logo, essa informação deixou de constar, respectivamente, nas estruturas *subgoal_frame* e *frozen_cp*. Para inserirmos pela primeira vez ou actualizarmos, por exemplo, o estado da computação utilizamos o predicado *built-in update_array(Name, Index, Value)* que recebe em *Name* o nome do vector (neste caso *subgoal_frame*), em *Index* o índice da posição no vector que pretendemos alterar (neste caso o índice corresponde ao *subgoal identifier* do subgolo) e em *Value* o novo elemento que queremos inserir (neste caso o novo estado).

Se pretendermos obter a informação que se encontra numa dada posição do vector utilizamos o predicado *built-in array_element(Name,Index,Value)* que recebe em *Name* o nome do vector e em *Index* o índice da posição no vector, e retorna em *Value* essa informação. Por exemplo, para sabermos o estado da computação de um determinado subgolo, invocamos o predicado *array_element* passando como argumento o nome do vector (*subgoal_frame*) e o *subgoal identifier* referente a esse subgolo.

O teste para verificar se o tamanho do vector *subgoal_frame* e *frozen_cp* ainda é adequado, é realizado no predicado *get_new_id/2* que podemos ver na Figura 4.28. Caso o tamanho do vector, que se encontra guardado numa varíavel global, for igual ao identificador que vai ser atribuído, então utilizamos o predicado *built-in* do Yap Prolog *resize_static_array(Name,Size,NewSize)* onde *Name* é o nome do vector, *Size* é o tamanho actual do vector, e *NewSize* é o tamanho do vector após a actualização. Optamos por actualizar o tamanho do vector sempre para o dobro do valor actual.

Com os vectores conseguimos, finalmente, resolver o problema do *binding* da variável AnsTrie. Por consequência, o predicado que contêm o código que deve ser executado quando um consumidor é acordado - *wake_frozen_cp/2* - passou a receber como argumento, para além do *frozen identifier* e do subgolo tabelado, a referência da *answer trie* referente ao subgolo que está a ser executado - passando a *wake_frozen_cp/3*. Desta forma, evitamos ter que efectuar uma pesquisa na estrutura *frozen_cp* para obter essa referência. A Figura 4.29 mostra o predicado *wake_frozen_cp/3* com vectores e predicados dinâmicos.

De modo a verificar as possíveis melhorias que os predicados dinâmicos e os vectores poderiam trazer, esta optimização foi implementada em todas as versões do módulo.

```
get_new_id(subgoal_frame_counter,Id) :-
   get_value(subgoal_frame_counter,Id),
   NewId is Id + 1,
   set_value(subgoal_frame_counter,NewId),
   get_value(subgoal_array_size,Size),
   (Id = Size ->
      NewSize is 2 * Size,
      resize_static_array(subgoal_frame,Size,NewSize),
      set_value(subgoal_array_size,NewSize)
   ;
      true
   ).

get_new_id(frozen_cp_counter,Id) :-
   get_value(frozen_cp_counter,Id),
   NewId is Id + 1,
   set_value(frozen_cp_counter,NewId),
   get_value(frozen_array_size,Size),
   (Id = Size ->
      NewSize is 2 * Size,
      resize_static_array(frozen_cp,Size,NewSize),
      set_value(frozen_array_size,NewSize)
   ;
      true
   ).
```

Figura 4.28: Predicado *get_new_id/2* alterado para funcionar com vectores.

```
wake_frozen_cp(FzId,Call,AnsTrie) :-
   array_element(frozen_cp,FzId,LastCons),
   consume_answer(Call,AnsTrie,LastCons).
wake_frozen_cp(FzId,_,AnsTrie) :-
   trie_get_last_entry(AnsTrie,LastAns),
   update_array_element(frozen_cp,FzId,LastAns),
   once(back_leader(LeaderFzId,BackCP)),
   (once(waiting_cp(LeaderFzId,NextCP)) ->
      retract(waiting_cp(LeaderFzId,NextCP)),
      wake_choice_point(NextCP)
    ;
      wake_choice_point(BackCP)
    ).
```

Figura 4.29: Predicado *wake_frozen_cp/3* com predicados dinâmicos e vectores.

4.4 Sumário

Neste capítulo descrevemos em detalhe a implementação de um módulo externo, baseado no sistema YapTab, que implementa a técnica da tabulação por transformação de programas e utilização de primitivas de suspensão. A implementação foi dividida em diversas fases. O primeiro passo foi construir um módulo estável que reproduzisse, de forma minimalista, o sistema YapTab.

Posteriormente foram realizadas diversas optimizações com o objectivo de melhorar o módulo. A primeira optimização foi redefinir a forma como as novas soluções eram consumidas quando ocorre uma recuperação da computação. Inicialmente, eram sempre consumidas todas as soluções. Com esta optimização passou-se a consumir as novas soluções a partir da última solução consumida.

A segunda optimização envolveu modificar a detecção do líder que, por consequência, permitiu marcar, de forma incremental, as computações como completas. De realçar, que antes desta optimização, como o líder era sempre o primeiro subgolo, as computações só eram marcadas como completas no final da execução.

Após a implementação destas duas optimizações, dotamos o módulo com uma outra estratégia de escalonamento, nomeadamente, a estratégia *batched scheduling* - a estratégia adoptada até esta optimização era a estratégia local *scheduling*. Note-se que, como as optimizações foram graduais, só implementados a estratégia *batched scheduling* na versão do módulo com consumo de soluções de forma incremental e marcação de computações como completas de forma incremental. Com esta optimização conseguimos reproduzir quase na totalidade o sistema YapTab no nosso módulo.

Devido ao facto de existirem alguns problemas ao nível da memória, a última optimização realizada foi ao nível da implementação das estruturas de dados utilizadas para guardar a informação necessária durante a computação. Inicialmente utilizamos a base de dados interna do Yap Prolog mas, como verificamos alguns problemas, passamos a usar predicados dinâmicos. Contudo, alguns problemas continuaram a existir e, por essa razão, resolvemos utilizar vectores juntamente com uma das outras duas hipóteses. Como com esta optimização resolvemos alguns dos problemas ao nível da memória, ela foi reproduzida em todas as versões do módulo.

No próximo capítulo iremos apresentar os resultados dos testes realizados ao módulo desenvolvido.

Capítulo 5

Resultados Experimentais

Este capítulo mostra uma análise detalhada do desempenho do módulo que implementa a técnica da tabulação por transformação de programas e utilização de primitivas de suspensão. Começamos por descrever, de forma sumária, o conjunto de programas utilizados para realizar a análise do desempenho. Em seguida, mostramos o tempo de execução que cada versão do módulo - versão base (V1); versão com consumo incremental de respostas (V2); e versão com detecção incremental de subgolos completos (V3) - obtém na execução desses programas e comparamos cada uma delas com o sistema YapTab. Com recurso à melhor das versões, mostramos também o tempo de execução dessa versão com as diferentes estruturas de dados - base de dados interna do Yap Prolog (*Recorded*); base de dados interna do Yap Prolog e vectores; predicados dinâmicos (*Assert*); e predicados dinâmicos e vectores - e comparamos cada uma delas com o sistema YapTab. Por fim, mostramos uma análise estatística a um grupo mais restrito de programas que, em nossa opinião, permite entender melhor os resultados obtidos.

5.1 Conjunto de Programas

Após a implementação do módulo externo estar terminada passamos à experimentação do mesmo num vasto conjunto de programas que recorrem à técnica da tabulação. Para isso, utilizamos um conjunto de testes, denominado *Test Suite*, que engloba diferentes tipos de programas. Um dos problemas incluídos no *Test Suite* é o problema de encontrar caminhos num grafo (predicado *path/2*). Este problema foi implementado utilizando as seis alternativas mais conhecidas para o predicado *path/2*:

- Recursão à esquerda com a cláusula recursiva em primeiro lugar:

```
path(X,Z):- path(X,Y), edge(Y,Z).
path(X,Z):- edge(X,Z).
```

- Recursão à esquerda com a cláusula recursiva em último lugar:

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
```

- Recursão à direita com a cláusula recursiva em primeiro lugar:

```
path(X,Z):- edge(X,Y), path(Y,Z).
path(X,Z):- edge(X,Z).
```

- Recursão à direita com a cláusula recursiva em último lugar:

```
path(X,Z):- edge(X,Z).
path(X,Z):- edge(X,Y), path(Y,Z).
```

- Dupla recursão com a cláusula recursiva em primeiro lugar:

```
path(X,Z):- path(X,Y), path(Y,Z).
path(X,Z):- edge(X,Z).
```

- Dupla recursão com a cláusula recursiva em último lugar:

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), path(Y,Z).
```

Para definir as ligações dos grafos (predicado *edge/2*) foram utilizadas as seguintes configurações: árvores; ciclos (o último nó tem uma ligação para o primeiro nó); grelhas (se existir uma ligação de um nó para outro, então essa ligação é mútua); e pirâmides (todos os nós tem ligação para um nó a baixo e para um nó à sua esquerda). Mediante o tipo de teste, o número de predicados que definem as ligações do grafo (*edge/2*) pode variar. A Figura 5.1 mostra alguns exemplos dessas configurações.

No *Test Suite* existe também um conjunto de problemas baseados no predicado *path/2* com recursão à esquerda onde os factos *edge/2* definem diferentes especificações de problemas de *Model Checking*.

Por fim, o *Test Suite* inclui um vasto conjunto de programas desenvolvidos pelo projecto *Open Rule Bench*[1]. Este projecto é uma comunidade de recursos aberta

[1]http://rulebench.projects.semwebcentral.org

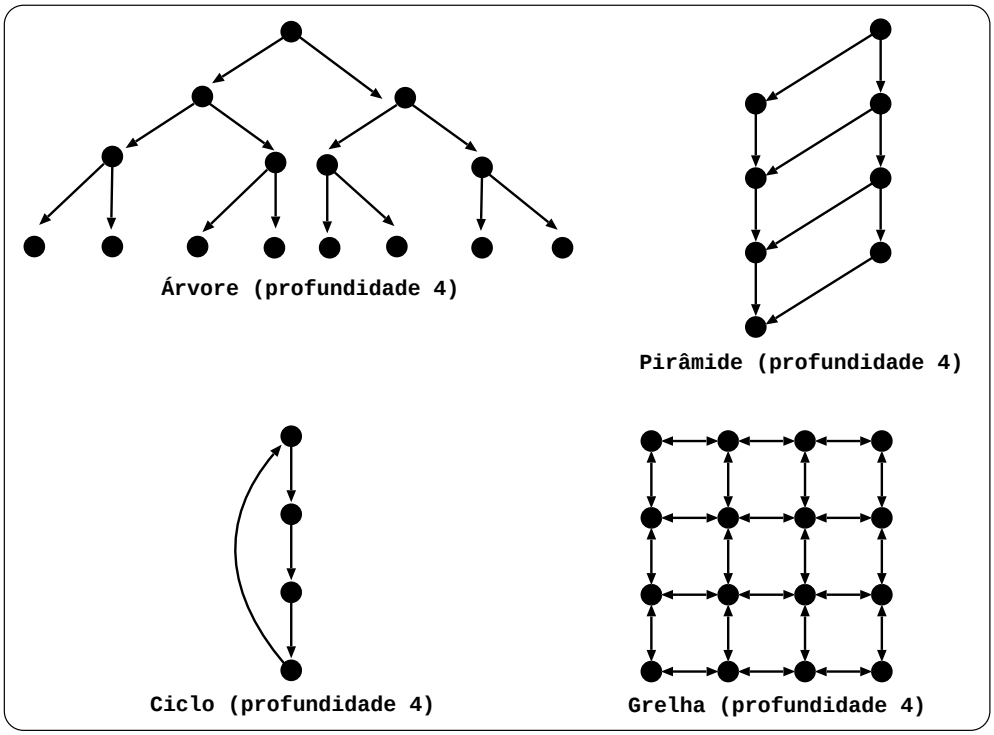

Figura 5.1: Tipos de configurações utilizadas para o predicado *edge/2*.

que tem vindo a implementar um conjunto de programas diversificado que permite analisar o desempenho e a escalabilidade de diferentes tipos de sistemas utilizados para *Semantic Web*. Dentro desse variado conjunto de testes, optamos por utilizar os testes *lubm*, *recursion_sg*, *recursion_tc* e *negation_win*.

Com os diferentes tipos de teste incluídos no *Test Suite* conseguimos averiguar a robustez do módulo, verificar pequenos erros que poderiam existir e analisar o seu desempenho.

5.2 Avaliação do Desempenho

De forma a avaliar o desempenho do nosso módulo executamos os diferentes programas, que constituem o nosso conjunto de testes, e medimos os tempos de execução em milissegundos. Para uma melhor análise do desempenho, decidimos comparar os tempos de execução obtidos com os do sistema YapTab, que implementa a técnica da tabulação ao nível da máquina de execução, utilizando a mesma estratégia de escalonamento utilizada na execução do módulo. De referir ainda que todos os tempos de execução foram obtidos no mesmo ambiente - um computador com um processador

Quad Core a 2.83 Ghz com 4 Gb de memória RAM a correr o sistema operativo Mandriva 2009.1 (versão 32 bits) e a versão 6.0.6 do Yap Prolog.

Neste capítulo apenas apresentamos os tempos de execução médios para cada grupo de programas que constituem o nosso conjunto de testes (os detalhes de todos os tempos de execução podem ser consultados no Apêndice A). Começamos então por analisar as diferentes versões do módulo - versão base (V1); versão com consumo incremental de respostas (V2); e versão com detecção incremental de subgolos completos (V3) - que utilizam como estrutura de dados a base de dados interna do Yap Prolog. Esta análise permite-nos demonstrar as melhorias que cada uma das optimizações pode trazer.

A Tabela 5.1 mostra os tempos médios de execução, em milissegundos, do sistema YapTab para os diferentes grupos de programas e, para cada uma das três versões, a proporção relativamente ao YapTab utilizando a estratégia de *local scheduling*. As duas últimas linhas da tabela mostram a proporção média global de cada uma das versões relativamente ao sistema YapTab, isto é, o tempo médio de execução de todos os programas para cada uma das versões a dividir pelo tempo médio de execução de todos os programas para o sistema YapTab; e o número de testes que não foram bem sucessidos - um teste pode não suceder caso a execução aborte por algum motivo, causando um *execution aborted*, ou então porque excedeu o tempo limite de 1 hora. Considerando os 7 grupos de programas, o total de testes é de 104 testes.

Grupo de Programas	**YapTab**	**Módulo/YapTab**		
		V1	**V2**	**V3**
Path Big	9.944	221.28	56.04	5.67
Path Medium	1.297	131.41	4.08	3.64
Model Checking	5.015	2.64	1.33	1.29
Lubm	621	1.00	1.00	1.00
Recursion Sg	10.013	11.40	3.78	3.80
Recursion Tc	44.385	115.72	13.40	5.14
Stratified Negation Win	1.084	31.98	11.00	11.28
Média Global		76.22	13.06	4.86
Testes Falhados		19	3	0

Tabela 5.1: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling*.

Como podemos ver, a versão V1 tem tempos bastante superiores relativamente às outras duas versões e é, em média, cerca de 76 vezes pior relativamente ao sistema

YapTab, além de que não sucedeu em 19 testes. A versão V2 permite, em média, uma melhoria de cerca de 6 vezes nos tempos de execução comparada com a versão V1 mas, para além de continuar a não suceder em todos os testes (falha em 3 testes), a diferença para o sistema YapTab ainda é bastante significativa (cerca de 13 vezes pior). A última versão V3 consegue ser, em média, apenas cerca de 5 vezes pior do que o sistema YapTab algo que já podemos considerar aceitável na medida em que, existem vários níveis de comunicação entre o Prolog e a máquina de execução, algo que não acontece no YapTab.

Na Tabela 5.2 podemos ver os tempos médios de execução, em milissegundos, do sistema YapTab para os diferentes grupos de programas e, para a melhor versão do módulo, a versão V3, a proporção relativamente ao YapTab utilizando a estratégia de *batched scheduling*. As duas últimas linhas da tabela mostram a proporção média global da versão V3 relativamente ao sistema YapTab e o número de testes que não foram bem sucessidos.

Grupo de Programas	**YapTab**	**Módulo/YapTab**
Path Big	11.599	5.20
Path Medium	1.148	3.93
Model Checking	4.945	1.28
Lubm	624	1.00
Recursion Sg	10.069	3.84
Recursion Tc	56.659	4.25
Stratified Negation Win	1.058	10.92
Média Global		4.20
Testes Falhados		1

Tabela 5.2: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da versão V3 do módulo utilizando a estratégia de *batched scheduling*.

Quando utilizamos a estratégia de escalonamento *batched scheduling*, os tempos de execução do nosso módulo são, em média, cerca de 4 vezes pior relativamente ao sistema YapTab. Contudo, com esta estratégia existe um teste que não sucede para o grupo de programas *Stratified Negation Win*.

De forma a averiguar se a estrutura de dados utilizada pode ou não influenciar os tempos de execução, o próximo passo do nosso estudo passou por medir os tempos de execução da versão V3 com as diferentes estruturas de dados existentes: base de dados interna do Yap Prolog (*Recorded*); base de dados interna do Yap Prolog e vectores;

predicados dinâmicos (*Assert*); e predicados dinâmicos e vectores. A Tabela 5.3 mostra os tempos médios de execução, em milissegundos, do sistema YapTab para os diferentes grupos de programas utilizando *local scheduling*, e a proporção da versão V3 implementada com as diferentes estruturas de dados. Na última linha da tabela podemos ver a proporção média global.

Grupo de Programas	**YapTab**	**Módulo/YapTab**			
		Recorded		**Assert**	
		~~Array~~	**Array**	**~~Array~~**	**Array**
Path Big	9.944	5.67	5.54	5.70	5.62
Path Medium	1.297	3.64	3.51	3.76	3.54
Model Checking	5.015	1.29	1.29	1.33	1.32
Lubm	621	1.00	1.00	1.00	1.00
Recursion Sg	10.013	3.80	3.74	3.79	3.80
Recursion Tc	44.385	5.14	5.21	5.18	5.16
Stratified Negation Win	1.084	11.28	6.84	16.47	8.89
Média Global		4.86	4.88	4.90	4.87

Tabela 5.3: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da versão V3 do módulo utilizando a estratégia de *local scheduling* com diferentes estruturas de dados.

Como podemos ver, a diferença entre utilizar como estrutura de dados a base de dados interna do Yap Prolog, com ou sem vectores, ou os predicados dinâmicos, com ou sem vectores, é, em média, pouco significativa. O único grupo de testes onde a diferença é mais notória é no caso dos *Stratified Negation Win*. Nesse caso concreto, se utilizarmos vectores juntamente com a base de dados interna do Yap Prolog conseguimos reduzir, em média, a proporção para o sistema YapTab para aproximadamente 7 vezes. Por essa razão, se tivéssemos que eleger a melhor versão do módulo, optaríamos pela versão V3 que utiliza como estrutura de dados a base de dados interna do Yap Prolog e vectores.

Utilizando a versão V3 com a base de dados interna do Yap Prolog e vectores como estrutura de dados, procedemos a uma análise estatística, que podemos observar na Tabela 5.4, para tentar perceber quais os factores que podem levar a um aumento da proporção do módulo relativamente ao sistema YapTab. Para efectuar essa análise seleccionamos alguns dos programas de teste com melhores (< 4) e piores (> 4) proporções e recolhemos a seguinte informação: o número de subgolos tabelados (ST); o número de líderes suspensos (LR); o número de líderes recuperados (LR); o número

de consumidores suspensos (CS); o número de consumidores recuperados (CR); e o número de soluções consumidas (SC). Na Tabela 5.4, a proporção de cada programa de teste aparece antes do nome do programa.

Programas de Teste	**ST**	**LS:LR**	**CS:CR**	**SC**
Proporção < 4				
1.16 : model_checking_reach_left_last_data_trans_sieve	1	0 : 0	1 : 0	760
2.35 : path_medium_left_last_data_edge_pyramid_1500	1	0 : 0	1 : 0	6.748.500
3.38 : recursion_sg_ff_data_d1000_parsize10000_sibsize2000_cyc	1.001	1 : 2	10.000 : 4.694	21.000.000
Proporção ≥ 4				
4.10 : recursion_sg_bf_data_d500_parsize10000_sibsize2000_cyc	500	1 : 2	10.000 : 4.799	5.000.500
5.13 : path_medium_right_first_data_edge_grid_35	1.226	1 : 4	4.760 : 8.186	13.162.625
6.28 : recursion_tc_fb_data_d1000_parsize500000_nocyc	1.000	0 : 0	409.851 : 0	0
7.50 : recursion_tc_fb_data_d1000_parsize50000_nocyc	994	0 : 0	31.634 : 0	0

Tabela 5.4: Valores estatísticos da versão V3 do módulo utilizando a estratégia de *local scheduling* com a base de dados interna do Yap Prolog e vectores.

Observando a Tabela 5.4 não é possível encontrar uma clara relação entre os factores para os quais recolhemos informação e as proporções dos programas de teste seleccionados. No entanto, os factores que maior contribuição parecem ter para o aumento da proporção do módulo relativamente ao sistema YapTab são o número de subgolos tabelados (ST) e o número de consumidores suspensos (CS). O número de consumidores recuperados (CR) também parece ser um factor com alguma relevância, apesar dos valores nulos obtidos para os programas com maiores proporções, o *recursion_tc_fb_data_d1000_parsize500000_nocyc* e o *recursion_tc_fb_data_d1000_parsize-50000_nocyc*.

Por outro lado, e ao contrário do que se poderia esperar, o número de soluções consumidas (SC) parece não ser um factor muito relevante como parecem sugerir os valores obtidos para os programas *path_medium_left_last_data_edge_pyramid_1500* e *recursion_sg_ff_data_d1000_parsize10000_sibsize2000_cyc*.

Capítulo 6

Conclusões e Trabalho Futuro

Este último capítulo sumariza o trabalho desenvolvido nesta tese. Começaremos por destacar os principais contributos deste trabalho e, em seguida, iremos sugerir algumas possibilidades de trabalho futuro.

6.1 Principais Contribuições

O trabalho desenvolvido nesta tese focou-se no problema de utilizar mecanismos que permitam suspender e recuperar a computação ao nível do próprio Prolog. Todos os problemas e algoritmos que exploram um espaço de procura podem ser facilmente modulados para utilizar estes mecanismos de suspensão e recuperação da computação. No caso concreto do nosso trabalho concentramos a nossa atenção no problema de implementar a técnica da tabulação a alto nível, mas de modo a que o modelo de execução fosse muito próximo da implementação de baixo nível. Realizamos duas abordagens, uma onde utilizamos os predicados *built-in* do Yap Prolog para guardar a informação necessária para retomar mais tarde uma computação suspensa, e uma outra onde a suspensão e a recuperação são efectuadas a baixo nível.

O estudo realizado demonstrou que a utilização do nosso módulo, desenvolvido com recurso às Primitivas de Suspensão e que nos permitiu implementar a técnica de tabulação ao nível do próprio Prolog, quando comparado com o sistema YapTab apresenta, como seria de esperar, tempos de execução ligeiramente piores. Estes resultados eram espectáveis já que o nosso módulo está todo ele escrito em Prolog e daí termos que pagar o custo associado aos vários níveis de comunicação entre o Prolog

e a máquina de execução, o que não acontece com o YapTab. Contudo, uma das mais valias do nosso módulo relativamente ao sistema YapTab é precisamente o facto de todo ele, primitivas de suspensão e directivas da tabulação, estar implementado em Prolog. Isto reduz drasticamente a complexidade da implementação, o que torna o nosso módulo mais apelativo e mais interessante para utilizadores menos experientes e para a sua experimentação com diferentes estratégias de execução e/ou novas extensões.

As principais contribuições deste trabalho são:

Primitivas de suspensão e recuperação da computação. Estas primitivas foram implementadas inicialmente com recurso aos predicados *built-in* do Yap Prolog que permitem guardar a informação necessária para retomar mais tarde uma computação suspensa. No entanto esta primeira abordagem, denominada *Chamada de Suspensão*, tem algumas limitações no que diz respeito ao tipo de chamada de continuação, como demonstrou Guzman et al [7], no caso de problemas onde é utilizada a técnica de tabulação implementada com recurso a mecanismos semelhantes. Uma solução para este problema foi igualmente proposta por Guzman et al [7] que passa por aumentar a complexidade do programa de transformação. Uma outra abordagem, que evita o aumento da complexidade do programa de transformação, é implementar os mecanismos de suspensão e recuperação através de *Primitivas de Suspensão*. Nesta abordagem, onde também são utilizados predicados *built-in* do Yap Prolog, a suspensão e a recuperação da computação são efectuadas a baixo nível.

Programa de transformação. A implementação do programa de transformação teve como base a abordagem original feita por Ramesh e Chen [15]. Este programa de transformação, escrito em Prolog, torna a utilização do módulo que implementa a tabulação ao nível do próprio Prolog completamente transparente para o utilizador na medida em que o utilizador continua a realizar as consultas da mesma forma. De referir ainda que este programa de transformação suporta as duas estratégias de escalonamento que actualmente têm maior êxito - *local scheduling* e *batched scheduling*.

Módulo que implementa as directivas da tabulação. Este módulo, desenvolvido com recurso às Primitivas de Suspensão, permitiu-nos implementar a técnica de tabulação ao nível do próprio Prolog mas de modo a que o modelo de execução fosse muito próximo da implementação de baixo nível. O módulo suporta igualmente as estratégias de escalonamento *local scheduling* e *batched*

scheduling e, ao contrário da abordagem de Ramesh e Chen [15], implementa as directivas de controle da tabulação totalmente em Prolog.

6.2 Trabalho Futuro

No nosso entender o facto da nossa abordagem reduzir consideravelmente a complexidade da implementação da técnica da tabulação, torna-a bastante atractiva tanto para utilizadores menos experientes como para utilizadores mais experientes. Utilizadores menos experientes sentir-se-ão mais confiantes para tentar compreender os detalhes envolvidos na implementação de um sistema deste género, e eventualmente experimentar alterar partes do código. Utilizadores mais experientes poderão facilmente estudar e rapidamente experimentar diferentes estratégias de execução da tabulação, optimizações e/ou extensões sem que para isso seja necessário perceber todos os detalhes da implementação de baixo nível.

Em particular para a equipa de desenvolvimento do YapTab, permitirá fazer experimentação de novas optimizações/extensões antes de as concretizar na implementação de baixo nível. Uma dessas extensões é o suporte para negação com tabulação. Um vasto grupo de aplicações que usam a tabulação requer a expressividade garantida pela possibilidade de manipular subgolos negativos. A possibilidade de expandir o nosso módulo para suportar de forma eficiente a negação, algo que pode ser bem mais simples por ser realizada ao nível do próprio Prolog, pode ser uma enorme vantagem para que, mais tarde, esta venha a ser integrada na versão oficial do YapTab.

Apêndice A

Tempos de Execução

A.1 Local Scheduling

A.1.1 Comparação Entre Versões

Programas	YapTab	Módulo/YapTab		
		V1	V2	V3
Path Big				
double_first_data_edge_btree_18	7.750	(*e.a.*)	(*e.a.*)	6.22
doube_first_data_edge_cycle_2000	8.920	242.78	88.25	6.11
double_first_data_edge_grid_35	9.190	233.32	85.00	5.99
double_first_data_edge_pyramid_1500	13.790	(>1h)	108.06	4.94
double_last_data_edge_btree_18	7.810	(*e.a.*)	(*e.a.*)	5.73
double_last_data_edge_cycle_2000	8.920	110.24	11.82	6.11
double_last_data_edge_grid_35	9.290	106.33	11.37	6.25
double_last_data_edge_pyramid_1500	13.880	340.11	5.32	4.91
Média	9.944	221.28	56.04	5.67

Tabela A.1: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling* no grupo de programas *Path Big* onde (*e.a.*) significa *execution aborted* e (>1h) significa que a execução excedeu o tempo limite de 1 hora.

Programas	YapTab	Módulo/YapTab		
		V1	V2	V3
Path Medium				
left_first_data_edge_btree_18	1.240	5.32	2.83	2.79
left_first_data_edge_cycle_2000	1.210	5.31	2.76	2.78
left_first_data_edge_grid_35	960	6.54	2.58	2.54
left_first_data_edge_pyramid_1500	1.320	4.85	2.39	2.39
left_last_data_edge_btree_18	1.320	4.97	2.65	2.67
left_last_data_edge_cycle_2000	1.320	4.98	2.50	2.64
left_last_data_edge_grid_35	940	6.68	2.59	2.60
left_last_data_edge_pyramid_1500	1.320	4.83	2.39	2.41
right_first_data_edge_btree_18	1.760	733.01	7.53	4.96
right_first_data_edge_cycle_2000	1.360	8.07	4.55	4.48
right_first_data_edge_grid_35	1.150	15.81	5.03	4.83
right_first_data_edge_pyramid_1500	1.230	10.03	3.98	3.99
right_last_data_edge_btree_18	1.730	750.71	7.45	5.01
right_last_data_edge_cycle_2000	1.410	8.03	4.41	4.30
right_last_data_edge_grid_35	1.170	18.14	4.80	4.72
right_last_data_edge_pyramid_1500	1.310	9.38	3.73	3.72
Média	1.297	131.41	4.08	3.64

Tabela A.2: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling* no grupo de programas *Path Medium*.

Programas	YapTab	Módulo/YapTab		
		V1	V2	V3
Model Checking				
reach_left_first_data_trans_iproto	1.240	3.38	1.73	1.74
reach_left_first_data_trans_leader	2.060	3.26	1.67	1.63
reach_left_first_data_trans_sieve	11.540	2.54	1.25	1.21
reach_left_last_data_trans_iproto	1.260	3.33	1.72	1.71
reach_left_last_data_trans_leader	2.070	3.24	1.65	1.65
reach_left_last_data_trans_sieve	11.920	2.38	1.21	1.17
Média	5.015	2.64	1.33	1.29

Tabela A.3: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling* no grupo de programas *Model Checking*.

Programas	YapTab	Módulo/YapTab		
		V1	V2	V3
Lubm				
query_14_data_10_univs	40	0.75	0.75	0.75
query_14_data_50_univs	210	1.00	1.00	1.00
query_1_data_10_univs	120	1.00	0.83	1.00
query_1_data_50_univs	1.180	1.00	1.01	1.00
query_2_data_10_univs	150	0.93	0.93	0.93
query_2_data_50_univs	1.060	0.84	0.84	0.84
query_9_data_10_univs	310	0.94	1.00	1.03
query_9_data_50_univs	1.900	1.11	1.11	1.11
Média	621	1.00	1.00	1.00

Tabela A.4: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling* no grupo de programas *Lubm*.

Programas	YapTab	Módulo/YapTab		
		V1	V2	V3
Recursion Sg				
bf_data_d1000_parsize10000_sibsize2000_cyc	6.730	10.08	3.86	3.85
bf_data_d1000_parsize10000_sibsize2000_nocyc	2.280	12.61	3.75	3.74
bf_data_d1000_parsize5000_sibsize1000_cyc	2.000	9.33	3.69	3.96
bf_data_d1000_parsize5000_sibsize1000_nocyc	50	8.40	2.80	3.00
bf_data_d500_parsize10000_sibsize2000_cyc	5.810	11.04	4.12	4.22
bf_data_d500_parsize10000_sibsize2000_nocyc	3.910	13.32	3.95	3.95
bf_data_d500_parsize20000_sibsize4000_cyc	22.360	10.77	4.93	4.12
bf_data_d500_parsize20000_sibsize4000_nocyc	19.690	13.64	4.02	4.03
bf_data_d500_parsize5000_sibsize1000_cyc	1.650	9.99	4.07	3.86
bf_data_d500_parsize5000_sibsize1000_nocyc	540	13.22	4.13	4.06
fb_data_d1000_parsize10000_sibsize2000_cyc	7.510	9.98	3.81	3.80
fb_data_d1000_parsize10000_sibsize2000_nocyc	4.280	12.09	3.69	3.63
fb_data_d1000_parsize5000_sibsize1000_cyc	2.370	9.08	3.65	3.63
fb_data_d1000_parsize5000_sibsize1000_nocyc	450	10.31	3.56	3.51
fb_data_d500_parsize10000_sibsize2000_cyc	6.270	10.56	3.89	4.05
fb_data_d500_parsize10000_sibsize2000_nocyc	5.050	13.33	3.95	3.94
fb_data_d500_parsize20000_sibsize4000_cyc	22.920	10.94	4.15	4.14
fb_data_d500_parsize20000_sibsize4000_nocyc	21.460	13.74	3.99	4.04
fb_data_d500_parsize5000_sibsize1000_cyc	1.780	10.11	4.01	3.96
fb_data_d500_parsize5000_sibsize1000_nocyc	990	12.85	3.91	3.86
ff_data_d1000_parsize10000_sibsize2000_cyc	15.600	9.79	3.38	3.91
ff_data_d1000_parsize10000_sibsize2000_nocyc	9.370	10.58	3.26	3.25
ff_data_d1000_parsize5000_sibsize1000_cyc	4.680	9.58	3.22	3.26
ff_data_d1000_parsize5000_sibsize1000_nocyc	950	9.36	3.16	3.13
ff_data_d500_parsize10000_sibsize2000_cyc	13.480	11.10	3.61	3.59
ff_data_d500_parsize10000_sibsize2000_nocyc	11.370	11.55	3.45	3.44
ff_data_d500_parsize20000_sibsize4000_cyc	51.470	10.98	3.69	3.86
ff_data_d500_parsize20000_sibsize4000_nocyc	49.490	11.76	3.33	3.46
ff_data_d500_parsize5000_sibsize1000_cyc	3.700	10.38	3.53	3.52
ff_data_d500_parsize5000_sibsize1000_nocyc	2.170	11.41	3.43	3.39
Média	10.013	11.40	3.78	3.80

Tabela A.5: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling* no grupo de programas *Recursion Sg*.

Programas	YapTab	Módulo/YapTab		
		V1	V2	V3
Recursion Tc				
bf_data_d1000_parsize250000_cyc	15.710	47.90	16.73	6.99
bf_data_d1000_parsize250000_nocyc	6.370	94.18	5.97	5.65
bf_data_d1000_parsize500000_cyc	31.380	79.88	27.87	8.13
bf_data_d1000_parsize500000_nocyc	13.230	164.05	5.76	5.49
bf_data_d1000_parsize50000_cyc	3.290	17.06	7.37	6.74
bf_data_d1000_parsize50000_nocyc	1.080	23.95	5.96	5.80
bf_data_d2000_parsize1000000_cyc	133.870	76.23	26.83	6.31
bf_data_d2000_parsize1000000_nocyc	52.150	169.94	5.59	5.42
bf_data_d2000_parsize500000_cyc	66.190	45.04	16.75	6.53
bf_data_d2000_parsize500000_nocyc	25.360	98.41	5.70	5.61
fb_data_d1000_parsize250000_cyc	250	7999.72	360.00	14.04
fb_data_d1000_parsize250000_nocyc	230	9.74	14.17	3.39
fb_data_d1000_parsize500000_cyc	590	13589.93	381.31	12.56
fb_data_d1000_parsize500000_nocyc	610	8.70	10.84	2.72
fb_data_d1000_parsize50000_cyc	50	1385.60	52.00	14.00
fb_data_d1000_parsize50000_nocyc	40	13.00	14.75	3.75
fb_data_d2000_parsize1000000_cyc	1.460	(>1h)	1172.48	11.04
fb_data_d2000_parsize1000000_nocyc	1.350	12.90	12.36	2.51
fb_data_d2000_parsize500000_cyc	670	(>1h)	793.60	11.25
fb_data_d2000_parsize500000_nocyc	600	8.77	11.15	2.75
ff_data_d1000_parsize250000_cyc	43.460	(>1h)	8.63	5.18
ff_data_d1000_parsize250000_nocyc	18.670	(>1h)	4.52	4.24
ff_data_d1000_parsize500000_cyc	89.330	(>1h)	12.48	5.03
ff_data_d1000_parsize500000_nocyc	43.180	(>1h)	3.90	3.81
ff_data_d1000_parsize50000_cyc	8.840	(>1h)	5.30	5.10
ff_data_d1000_parsize50000_nocyc	2.920	(>1h)	4.99	4.87
ff_data_d2000_parsize1000000_cyc	368.520	(>1h)	12.28	4.78
ff_data_d2000_parsize1000000_nocyc	157.570	(>1h)	3.97	4.11
ff_data_d2000_parsize500000_cyc	175.230	(>1h)	8.91	5.10
ff_data_d2000_parsize500000_nocyc	69.350	(>1h)	4.54	4.53
Média	44.385	115.72	13.40	5.14

Tabela A.6: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling* no grupo de programas *Recursion Tc* onde (>1h) significa que a execução excedeu o tempo limite de 1 hora.

Programas	YapTab	Módulo/YapTab		
		V1	V2	V3
Stratified Negation Win				
data_upper1000000	2.940	(*e.a.*)	(*e.a.*)	11.83
data_upper100000	240	(*e.a.*)	16.21	11.54
data_upper250000	680	(*e.a.*)	17.56	10.66
data_upper500000	1.460	31,98	20.54	10.18
data_upper50000	100	(*e.a.*)	18.80	14.60
Média	1.084	31.98	11.00	11.28

Tabela A.7: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa das diferentes versões do módulo utilizando a estratégia de *local scheduling* no grupo de programas *Stratified Negation Win* onde (*e.a.*) significa *execution aborted*.

A.1.2 Comparação Entre Estruturas de Dados

Programas	YapTab	Módulo/YapTab			
		Recorded		Assert	
		~~Array~~	Array	~~Array~~	Array
Path Big					
double_first_data_edge_btree_18	7.750	6.22	5.73	6.26	5.86
doube_first_data_edge_cycle_2000	8.920	6.11	6.00	6.19	6.07
double_first_data_edge_grid_35	9.190	5.99	5.92	6.17	6.07
double_first_data_edge_pyramid_1500	13.790	4.94	4.98	5.04	5.11
double_last_data_edge_btree_18	7.810	5.73	5.39	5.98	5.51
double_last_data_edge_cycle_2000	8.920	6.11	6.01	6.03	6.03
double_last_data_edge_grid_35	9.290	6.25	5.98	5.93	6.00
double_last_data_edge_pyramid_1500	13.880	4.91	4.94	4.87	4.94
Média	9.944	5.67	5.54	5.70	5.62

Tabela A.8: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da Versão 3 do módulo com as diferentes estruturas de dados utilizando a estratégia de *local scheduling* no grupo de programas *Path Big*.

Programas	YapTab	Módulo/YapTab			
		Recorded		Assert	
		~~Array~~	Array	~~Array~~	Array
Path Medium					
left_first_data_edge_btree_18	1.240	2.79	2.75	2.83	2.81
left_first_data_edge_cycle_2000	1.210	2.78	2.73	2.69	2.73
left_first_data_edge_grid_35	960	2.54	2.56	2.51	2.52
left_first_data_edge_pyramid_1500	1.320	2.39	2.33	2.38	2.35
left_last_data_edge_btree_18	1.320	2.67	2.63	2.67	2.61
left_last_data_edge_cycle_2000	1.320	2.64	2.51	2.56	2.70
left_last_data_edge_grid_35	940	2.60	2.56	2.54	2.55
left_last_data_edge_pyramid_1500	1.320	2.41	2.35	2.33	2.41
right_first_data_edge_btree_18	1.760	4.96	4.14	6.00	4.65
right_first_data_edge_cycle_2000	1.360	4.48	4.40	4.15	4.18
right_first_data_edge_grid_35	1.150	4.83	5.13	4.71	4.77
right_first_data_edge_pyramid_1500	1.230	3.99	4.41	4.11	4.12
right_last_data_edge_btree_18	1.730	5.01	4.20	6.10	4.67
right_last_data_edge_cycle_2000	1.410	4.30	4.27	4.03	4.05
right_last_data_edge_grid_35	1.170	4.72	4.58	4.65	4.72
right_last_data_edge_pyramid_1500	1.310	3.72	3.79	3.85	3.74
Média	1.297	3.64	3.51	3.76	3.54

Tabela A.9: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da Versão 3 do módulo com as diferentes estruturas de dados utilizando a estratégia de *local scheduling* no grupo de programas *Path Medium*.

Programas	YapTab	Módulo/YapTab			
		Recorded		Assert	
		~~Array~~	Array	~~Array~~	Array
Model Checking					
reach_left_first_data_trans_iproto	1.240	1.74	1.72	1.73	1.77
reach_left_first_data_trans_leader	2.060	1.63	1.66	1.67	1.66
reach_left_first_data_trans_sieve	11.540	1.21	1.20	1.27	1.22
reach_left_last_data_trans_iproto	1.260	1.71	1.70	1.72	1.71
reach_left_last_data_trans_leader	2.070	1.65	1.66	1.63	1.63
reach_left_last_data_trans_sieve	11.920	1.17	1.16	1.19	1.23
Média	5.015	1.29	1.29	1.33	1.32

Tabela A.10: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da Versão 3 do módulo com as diferentes estruturas de dados utilizando a estratégia de *local scheduling* no grupo de programas *Model Checking*.

Programas	YapTab	Módulo/YapTab			
		Recorded		Assert	
		~~Array~~	Array	~~Array~~	Array
Lubm					
query_14_data_10_univs	40	0.75	0.75	0.75	0.75
query_14_data_50_univs	210	1.00	1.05	1.05	1.05
query_1_data_10_univs	120	1.00	0.83	0.75	0.92
query_1_data_50_univs	1.180	1.00	1.00	1.01	1.00
query_2_data_10_univs	150	0.93	0.93	1.00	0.93
query_2_data_50_univs	1.060	0.84	0.84	0.84	0.86
query_9_data_10_univs	310	1.03	1.00	0.87	1.00
query_9_data_50_univs	1.900	1.11	1.09	1.11	1.10
Média	621	1.00	1.00	1.00	1.00

Tabela A.11: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da Versão 3 do módulo com as diferentes estruturas de dados utilizando a estratégia de *local scheduling* no grupo de programas *Lubm*.

Programas	YapTab	Módulo/YapTab			
		Recorded		Assert	
		~~Array~~	Array	~~Array~~	Array
Recursion Sg					
bf_data_d1000_parsize10000_sibsize2000_cyc	6.730	3.85	4.04	3.84	4.07
bf_data_d1000_parsize10000_sibsize2000_nocyc	2.280	3.74	3.73	3.73	3.79
bf_data_d1000_parsize5000_sibsize1000_cyc	2.000	3.96	3.71	3.65	3.69
bf_data_d1000_parsize5000_sibsize1000_nocyc	50	3.00	3.00	3.00	3.00
bf_data_d500_parsize10000_sibsize2000_cyc	5.810	4.22	4.10	4.15	4.13
bf_data_d500_parsize10000_sibsize2000_nocyc	3.910	3.95	3.98	3.97	3.97
bf_data_d500_parsize20000_sibsize4000_cyc	22.360	4.12	4.12	4.26	4.45
bf_data_d500_parsize20000_sibsize4000_nocyc	19.690	4.03	4.04	4.02	4.14
bf_data_d500_parsize5000_sibsize1000_cyc	1.650	3.86	3.91	4.01	3.92
bf_data_d500_parsize5000_sibsize1000_nocyc	540	4.06	3.98	4.37	4.09
fb_data_d1000_parsize10000_sibsize2000_cyc	7.510	3.80	3.82	3.81	3.89
fb_data_d1000_parsize10000_sibsize2000_nocyc	4.280	3.63	3.65	3.56	3.90
fb_data_d1000_parsize5000_sibsize1000_cyc	2.370	3.63	3.86	3.63	3.63
fb_data_d1000_parsize5000_sibsize1000_nocyc	450	3.51	3.64	3.58	3.62
fb_data_d500_parsize10000_sibsize2000_cyc	6.270	4.05	4.03	4.01	4.25
fb_data_d500_parsize10000_sibsize2000_nocyc	5.050	3.94	3.99	3.94	3.94
fb_data_d500_parsize20000_sibsize4000_cyc	22.920	4.14	4.13	4.13	4.14
fb_data_d500_parsize20000_sibsize4000_nocyc	21.460	4.04	3.98	4.01	3.99
fb_data_d500_parsize5000_sibsize1000_cyc	1.780	3.96	3.97	4.00	3.98
fb_data_d500_parsize5000_sibsize1000_nocyc	990	3.86	3.91	3.89	3.88
ff_data_d1000_parsize10000_sibsize2000_cyc	15.600	3.91	3.38	4.25	3.36
ff_data_d1000_parsize10000_sibsize2000_nocyc	9.370	3.25	3.25	3.22	3.24
ff_data_d1000_parsize5000_sibsize1000_cyc	4.680	3.26	3.22	3.25	3.20
ff_data_d1000_parsize5000_sibsize1000_nocyc	950	3.13	3.14	3.17	3.11
ff_data_d500_parsize10000_sibsize2000_cyc	13.480	3.59	3.60	3.63	3.60
ff_data_d500_parsize10000_sibsize2000_nocyc	11.370	3.44	3.44	3.44	3.47
ff_data_d500_parsize20000_sibsize4000_cyc	51.470	3.86	3.64	3.65	3.74
ff_data_d500_parsize20000_sibsize4000_nocyc	49.490	3.46	3.46	3.45	3.46
ff_data_d500_parsize5000_sibsize1000_cyc	3.700	3.52	3.61	3.54	3.55
ff_data_d500_parsize5000_sibsize1000_nocyc	2.170	3.39	3.43	3.40	3.44
Média	10.013	3.80	3.74	3.79	3.80

Tabela A.12: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da Versão 3 do módulo com as diferentes estruturas de dados utilizando a estratégia de *local scheduling* no grupo de programas *Recursion Sg*.

Programas	YapTab	Módulo/YapTab			
		Recorded		Assert	
		~~Array~~	Array	~~Array~~	Array
Recursion Tc					
bf_data_d1000_parsize250000_cyc	15.710	6.99	6.95	6.89	6.94
bf_data_d1000_parsize250000_nocyc	6.370	5.65	5.82	5.62	5.77
bf_data_d1000_parsize500000_cyc	31.380	8.13	6.95	7.06	7.01
bf_data_d1000_parsize500000_nocyc	13.230	5.49	5.73	5.51	5.66
bf_data_d1000_parsize50000_cyc	3.290	6.74	6.65	6.69	6.85
bf_data_d1000_parsize50000_nocyc	1.080	5.80	5.94	6.25	5.93
bf_data_d2000_parsize1000000_cyc	133.870	6.31	6.52	6.64	6.51
bf_data_d2000_parsize1000000_nocyc	52.150	5.42	5.66	5.38	5.43
bf_data_d2000_parsize500000_cyc	66.190	6.53	6.57	6.62	6.60
bf_data_d2000_parsize500000_nocyc	25.360	5.61	5.71	5.56	5.75
fb_data_d1000_parsize250000_cyc	250	14.04	12.60	14.04	12.60
fb_data_d1000_parsize250000_nocyc	230	3.39	6.91	3.09	7.17
fb_data_d1000_parsize500000_cyc	590	12.56	11.24	11.32	10.54
fb_data_d1000_parsize500000_nocyc	610	2.72	6.28	2.39	6.26
fb_data_d1000_parsize50000_cyc	50	14.00	12.60	14.60	13.20
fb_data_d1000_parsize50000_nocyc	40	3.75	7.50	3.75	7.75
fb_data_d2000_parsize1000000_cyc	1.460	11.04	9.93	10.88	9.87
fb_data_d2000_parsize1000000_nocyc	1.350	2.51	5.13	2.30	4.98
fb_data_d2000_parsize500000_cyc	670	11.25	10.09	11.48	9.94
fb_data_d2000_parsize500000_nocyc	600	2.75	5.35	2.57	5.27
ff_data_d1000_parsize250000_cyc	43.460	5.18	4.98	5.14	5.19
ff_data_d1000_parsize250000_nocyc	18.670	4.24	4.30	4.22	4.35
ff_data_d1000_parsize500000_cyc	89.330	5.03	5.04	5.27	5.10
ff_data_d1000_parsize500000_nocyc	43.180	3.81	3.83	3.81	3.86
ff_data_d1000_parsize50000_cyc	8.840	5.10	5.13	5.11	5.10
ff_data_d1000_parsize50000_nocyc	2.920	4.87	4.88	4.77	4.88
ff_data_d2000_parsize1000000_cyc	368.520	4.78	4.85	4.84	4.88
ff_data_d2000_parsize1000000_nocyc	157.570	4.11	4.21	4.01	3.97
ff_data_d2000_parsize500000_cyc	175.230	5.10	5.32	5.22	5.11
ff_data_d2000_parsize500000_nocyc	69.350	4.53	4.54	4.50	4.54
Média	44.385	5.14	5.21	5.18	5.16

Tabela A.13: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da Versão 3 do módulo com as diferentes estruturas de dados utilizando a estratégia de *local scheduling* no grupo de programas *Recursion Tc*.

Programas	YapTab	Módulo/YapTab			
		Recorded		Assert	
		~~Array~~	Array	~~Array~~	Array
Stratified Negation Win					
data_upper1000000	2.940	11.83	6.92	16.68	8.74
data_upper100000	240	11.54	7.42	17.79	10.04
data_upper250000	680	10.66	6.75	16.38	9.15
data_upper500000	1.460	10.18	6.51	15.55	8.71
data_upper50000	100	14.60	8.70	21.30	11.80
Média	1.084	11.28	6.84	16.47	8.89

Tabela A.14: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da Versão 3 do módulo com as diferentes estruturas de dados utilizando a estratégia de *local scheduling* no grupo de programas *Stratified Negation Win*.

A.2 Batched Scheduling

A.2.1 Versão V3

Programas	YapTab	Módulo/YapTab
Path Big		
double_first_data_edge_btree_18	9.730	5.27
doube_first_data_edge_cycle_2000	10.780	5.43
double_first_data_edge_grid_35	10.260	6.23
double_first_data_edge_pyramid_1500	15.430	4.84
double_last_data_edge_btree_18	10.040	4.72
double_last_data_edge_cycle_2000	10.810	5.43
double_last_data_edge_grid_35	10.190	5.61
double_last_data_edge_pyramid_1500	15.550	4.56
Média	11.599	5.20

Tabela A.15: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da ultima versão do módulo utilizando a estratégia de *batched scheduling* no grupo de programas *Path Big*.

Programas	YapTab	Módulo/YapTab
Path Medium		
left_first_data_edge_btree_18	1.030	3.22
left_first_data_edge_cycle_2000	880	3.67
left_first_data_edge_grid_35	840	2.90
left_first_data_edge_pyramid_1500	990	3.05
left_last_data_edge_btree_18	970	3.44
left_last_data_edge_cycle_2000	870	3.62
left_last_data_edge_grid_35	830	2.98
left_last_data_edge_pyramid_1500	960	3.48
right_first_data_edge_btree_18	1.780	4.84
right_first_data_edge_cycle_2000	1.160	4.07
right_first_data_edge_grid_35	1.230	4.60
right_first_data_edge_pyramid_1500	1.190	4.00
right_last_data_edge_btree_18	1.790	4.75
right_last_data_edge_cycle_2000	1.290	3.64
right_last_data_edge_grid_35	1.270	4.68
right_last_data_edge_pyramid_1500	1.280	3.81
Média	1.148	3.93

Tabela A.16: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da última versão do módulo utilizando a estratégia de *batched scheduling* no grupo de programas *Path Medium*.

Programas	YapTab	Módulo/YapTab
Model Checking		
reach_left_first_data_trans_iproto	1.150	1.84
reach_left_first_data_trans_leader	2.080	1.53
reach_left_first_data_trans_sieve	11.600	1.18
reach_left_last_data_trans_iproto	1.150	1.84
reach_left_last_data_trans_leader	2.070	1.54
reach_left_last_data_trans_sieve	11.620	1.18
Média	4.945	1.28

Tabela A.17: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da ultima versão do módulo utilizando a estratégia de *batched scheduling* no grupo de programas *Model Checking*.

Programas	YapTab	Módulo/YapTab
Lubm		
query_14_data_10_univs	30	0.67
query_14_data_50_univs	200	1.05
query_1_data_10_univs	110	1.00
query_1_data_50_univs	1.190	1.00
query_2_data_10_univs	140	1.00
query_2_data_50_univs	880	1.02
query_9_data_10_univs	330	0.91
query_9_data_50_univs	2.110	1.00
Média	624	1.00

Tabela A.18: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da ultima versão do módulo utilizando a estratégia de *batched scheduling* no grupo de programas *Lubm*.

Programas	YapTab	Módulo/YapTab
Recursion Sg		
bf_data_d1000_parsize10000_sibsize2000_cyc	6.940	3.90
bf_data_d1000_parsize10000_sibsize2000_nocyc	2.320	3.85
bf_data_d1000_parsize5000_sibsize1000_cyc	2.080	3.72
bf_data_d1000_parsize5000_sibsize1000_nocyc	40	4.00
bf_data_d500_parsize10000_sibsize2000_cyc	5.880	4.20
bf_data_d500_parsize10000_sibsize2000_nocyc	3.910	4.07
bf_data_d500_parsize20000_sibsize4000_cyc	22.190	4.26
bf_data_d500_parsize20000_sibsize4000_nocyc	19.720	4.19
bf_data_d500_parsize5000_sibsize1000_cyc	1.630	4.10
bf_data_d500_parsize5000_sibsize1000_nocyc	570	3.98
fb_data_d1000_parsize10000_sibsize2000_cyc	7.820	3.85
fb_data_d1000_parsize10000_sibsize2000_nocyc	4.300	3.84
fb_data_d1000_parsize5000_sibsize1000_cyc	2.540	3.54
fb_data_d1000_parsize5000_sibsize1000_nocyc	440	3.70
fb_data_d500_parsize10000_sibsize2000_cyc	6.260	4.19
fb_data_d500_parsize10000_sibsize2000_nocyc	5.050	4.03
fb_data_d500_parsize20000_sibsize4000_cyc	23.100	4.48
fb_data_d500_parsize20000_sibsize4000_nocyc	21.360	4.07
fb_data_d500_parsize5000_sibsize1000_cyc	1.830	3.98
fb_data_d500_parsize5000_sibsize1000_nocyc	980	4.02
ff_data_d1000_parsize10000_sibsize2000_cyc	15.740	3.50
ff_data_d1000_parsize10000_sibsize2000_nocyc	9.650	3.26
ff_data_d1000_parsize5000_sibsize1000_cyc	4.710	3.37
ff_data_d1000_parsize5000_sibsize1000_nocyc	950	3.25
ff_data_d500_parsize10000_sibsize2000_cyc	13.560	3.68
ff_data_d500_parsize10000_sibsize2000_nocyc	11.400	3.52
ff_data_d500_parsize20000_sibsize4000_cyc	51.590	3.73
ff_data_d500_parsize20000_sibsize4000_nocyc	49.640	3.53
ff_data_d500_parsize5000_sibsize1000_cyc	3.720	3.78
ff_data_d500_parsize5000_sibsize1000_nocyc	2.140	3.57
Média	10.069	3.84

Tabela A.19: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da última versão do módulo utilizando a estratégia de *batched scheduling* no grupo de programas *Recursion Sg*.

Programas	YapTab	Módulo/YapTab
Recursion Tc		
bf_data_d1000_parsize250000_cyc	21.240	5.35
bf_data_d1000_parsize250000_nocyc	6.490	5.50
bf_data_d1000_parsize500000_cyc	37.700	5.99
bf_data_d1000_parsize500000_nocyc	13.350	5.48
bf_data_d1000_parsize50000_cyc	5.910	4.27
bf_data_d1000_parsize50000_nocyc	1.100	5.55
bf_data_d2000_parsize1000000_cyc	174.270	5.28
bf_data_d2000_parsize1000000_nocyc	52.940	5.29
bf_data_d2000_parsize500000_cyc	103.400	4.71
bf_data_d2000_parsize500000_nocyc	25.770	5.56
fb_data_d1000_parsize250000_cyc	260	10.92
fb_data_d1000_parsize250000_nocyc	240	3.21
fb_data_d1000_parsize500000_cyc	580	11.31
fb_data_d1000_parsize500000_nocyc	590	2.80
fb_data_d1000_parsize50000_cyc	50	11.00
fb_data_d1000_parsize50000_nocyc	50	3.00
fb_data_d2000_parsize1000000_cyc	1.400	10.24
fb_data_d2000_parsize1000000_nocyc	1.360	2.51
fb_data_d2000_parsize500000_cyc	620	10.39
fb_data_d2000_parsize500000_nocyc	610	2.74
ff_data_d1000_parsize250000_cyc	58.760	4.01
ff_data_d1000_parsize250000_nocyc	18.910	4.13
ff_data_d1000_parsize500000_cyc	101.840	4.47
ff_data_d1000_parsize500000_nocyc	43.390	3.71
ff_data_d1000_parsize50000_cyc	14.910	3.49
ff_data_d1000_parsize50000_nocyc	2.940	5.03
ff_data_d2000_parsize1000000_cyc	499.570	3.82
ff_data_d2000_parsize1000000_nocyc	159.300	3.94
ff_data_d2000_parsize500000_cyc	282.020	3.69
ff_data_d2000_parsize500000_nocyc	70.200	4.42
Média	56.659	4.25

Tabela A.20: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da última versão do módulo utilizando a estratégia de *batched scheduling* no grupo de programas *Recursion Tc*.

Programas	**YapTab**	**Módulo/YapTab**
Stratified Negation Win		
data_upper1000000	3.180	10.99
data_upper100000	240	10.54
data_upper250000	710	10.45
data_upper500000	1.520	(*e.a.*)
data_upper50000	100	12.90
Média	1.058	10.92

Tabela A.21: Tempos de execução, em milissegundos, do sistema YapTab e proporção relativa da última versão do módulo utilizando a estratégia de *batched scheduling* no grupo de programas *Stratified Negation Win* onde (*e.a.*) significa *execution aborted*.

Referências

[1] H. Aït-Kaci. Warren's Abstract Machine – A Tutorial Reconstruction. The MIT Press, 1991.

[2] M. Carlsson. Design and Implementation of an OR-Parallel Prolog Engine. PhD thesis, The Royal Institute of Technology, 1990.

[3] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM, 43(1):20–74, 1996.

[4] P. Chico, M. Carro, M. V. Hermenegildo, C. Silva, and R. Rocha. An Improved Continuation Call-Based Implementation of Tabling. In International Symposium on Practical Aspects of Declarative Languages, number 4902 in LNCS, pages 197–213. Springer-Verlag, 2008.

[5] W. Clocksin and C. Mellish. Programming in Prolog. Springer-Verlag, fourth edition, 1994.

[6] 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, 1973.

[7] Pablo Chico de Guzman, Manuel Carro, and Manuel V. Hermenegildo. Towards a Complete Scheme for Tabled Execution Based on Program Transformation. In International Symposium on Practical Aspects of Declarative Languages, number 5418 in LNCS, pages 224–238. Springer Berlin/Heidelberg, 2009.

[8] S. Dietrich. Extension Tables for Recursive Query Evaluation. PhD thesis, Department of Computer Science, State University of New York, 1987.

[9] E. Fredkin. Trie Memory. Communications of the ACM, 3:490–499, 1962.

[10] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In International Symposium on Programming Language Implementation and Logic Programming, number 1140 in LNCS, pages 243–258. Springer-Verlag, 1996.

[11] Hai-Feng Guo and G. Gupta. A Simple Scheme for Implementing Tabling based on Dynamic Reordering of Alternatives. In Conference on Tabulation in Parsing and Deduction, pages 141–154, 2000.

[12] J. W. Lloyd. Foundations of Logic Programming. Springer-Verlag, 1987.

[13] D. Michie. Memo Functions and Machine Learning. Nature, 218:19–22, 1968.

[14] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming, 38(1):31–54, 1999.

[15] R. Ramesh and W. Chen. Implementation of Tabled Evaluation with Delaying in Prolog. IEEE Transactions on Knowledge and Data Engineering, 9(4):559–574, 1997.

[16] P. Rao, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In International Conference on Logic Programming and Non-Monotonic Reasoning, number 1265 in LNCS, pages 431–441. Springer-Verlag, 1997.

[17] R. Rocha. Efficient Support for Incomplete and Complete Tables in the YapTab Tabling System. In Colloquium on Implementation of Constraint and LOgic Programming Systems, pages 2–17, 2006.

[18] R. Rocha, C. Silva, and R. Lopes. Implementation of Suspension-Based Tabling in Prolog using External Primitives. In Local Proceedings of the 13th Portuguese Conference on Artificial Intelligence, pages 11–22, 2007.

[19] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In Conference on Tabulation in Parsing and Deduction, pages 77–87, 2000.

[20] R. Rocha, F. Silva, and V. Santos Costa. Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In International Conference on Logic Programming, number 3668 in LNCS, pages 250–264. Springer-Verlag, 2005.

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

[22] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems, 20(3):586–634, 1998.

[23] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In ACM SIGMOD International Conference on the Management of Data, pages 442–453. ACM Press, 1994.

[24] C. Silva. On Applying Program Transformation to Implement Tabled Evaluation in Prolog. Master's thesis, University of Porto, 2007.

[25] L. Sterling and E. Shapiro. The Art of Prolog. The MIT Press, 1994.

[26] H. Tamaki and T. Sato. OLDT Resolution with Tabulation. In International Conference on Logic Programming, number 225 in LNCS, pages 84–98. Springer-Verlag, 1986.

[27] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.

[28] L. Vieille. Recursive Query Processing: The Power of Logic. Theoretical Computer Science, 69(1):1–53, 1989.

[29] A. Walker. Backchain Iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. Journal of Automated Reasoning, 11(1):1–23, 1993.

[30] D. H. D. Warren. Applied Logic – Its Use and Implementation as a Programming Tool. PhD thesis, Edinburgh University, 1977.

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

[32] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a Linear Tabling Mechanism. In Practical Aspects of Declarative Languages, number 1753 in LNCS, pages 109–123. Springer-Verlag, 2000.