UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica

por

JORGE LUIS VICTÓRIA BARBOSA

Dissertação submetida à avaliação, como
requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, julho de 1996

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Barbosa, Jorge Luis Victória

GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica / Jorge Luis Victória Barbosa. - Porto Alegre: CPGCC da UFRGS, 1996.

167 p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1996. Orientador: Geyer, Cláudio Fernando Resin.

1. Granulosidade. 2. Análise de Granulosidade. 3. Processamento Paralelo. 4. Paralelismo na Programação em Lógica. I. Geyer, Cláudio Fernando Resin. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Hélgio Trindade

Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Cláudio Scherer

Diretor do Instituto de Informática: Prof. Roberto Tom Price

Coordenador do CPGCC: Prof. Flávio Rech Wagner

Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Nada é mais nobre do que a **criação**, nem mais sublime do que a **inspiração**. Jamais sacrifique alguma delas em nome da **produção**.

Jorge Luis Victória Barbosa

Agradecimentos

Inicialmente, gostaria de agradecer à todos que fazem parte da irmandade que costumamos denominar comunidade científica. Estas pessoas que trabalham em conjunto, trocam idéias, orientam, são orientadas, criticam, são criticadas, cobram, são cobradas e dedicam incontáveis horas a pesquisa. Obrigado por terem me acolhido. Obrigado pelo contínuo incentivo oriundo de sua estimulante natureza criadora. Obrigado por colocarem a minha disposição o principal fruto da ciência, ou seja, o conhecimento.

Queridos familiares, obrigado pela maravilhosa experiência proporcionada pelo nosso convívio. Obrigado aos meus pais, Nadir e Zita. Obrigado a minha segunda mãe, Marlene. Obrigado a mana Bia, a seu companheiro André e sua filhinha Victória (ainda por nascer). Obrigado ao mano Sérgio, sua companheira Celina e seus filhos Antônio e Helena. Vocês compõem a base da minha existência. Amo vocês.

Agradeço ao Professor Cláudio Geyer, meu amigo e orientador. Obrigado pelo incentivo, pelo exemplo, pela orientação, pelo convívio e principalmente pelo seu companheirismo. Uma mão orientadora é muito importante quanto temos que percorrer um árduo caminho como um curso de pós-graduação. Obrigado pela mão.

Professor Adenauer, grande amigo e orientador, agradecimentos específicos jamais poderiam transmitir minha imensa gratidão. Sendo assim, muito obrigado compadre.

Agradeço aos meus grandes amigos de Pelotas: Clara, Helena, Eduardo, Ednilson e Fernando. Vocês estão sempre comigo. Obrigado pela Amizade.

Obrigado aos meus amigos e colegas de mestrado: Eduardo Bezerra (vulgo Careca ou mão-suja), João Schramm (vulgo Schramm), Cláudio Félix (vulgo Cafelix) e Carlos Alberto Teixera Junior (vulgo CATJ ou Junior). Nossa parceria foi uma experiência maravilhosa. Jamais esquecerei vocês.

Agradeço aos meus colegas do projeto OPERA (OPERArios): Otília, Denise, Patrícia, Charles, Vinícius, Cristiano e Luis Fernando. Obrigado pelo apoio, companheirismo e pelo auxílio no desenvolvimento deste trabalho.

Obrigado aos meus colegas de mestrado. Vocês são muito especiais e sempre farão parte das minhas melhores lembranças do CPGCC.

Agradeço aos professores e funcionários do CPGCC. Obrigado pelas aulas, orientação, convivência, apoio técnico e administrativo. Tenho certeza que continuaremos nos encontrando.

Caros colegas professores e funcionários da Universidade Católica de Pelotas, obrigado pelo apoio e incentivo. Companheiros da Escola de Informática, formamos um grupo muito especial. Obrigado pelo privilégio da convivência diária com vocês.

Obrigado à todos os meus alunos, os quais constituem a principal motivação para meu contínuo aperfeiçoamento.

Agradeço à CAPES pelo apoio financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

AAD	Algoritmo para Análise de Dependências
AC	Analisador de Complexidade
ACE	Analisador de Complexidade E
ACO	Analisador de Complexidade OU
ACRPL	Acumulador da Complexidade das Resolvantes Pendentes Locais
ADG	Algoritmo para Determinação de Grãos
AG	Adaptador de Grafos
AGP	Anotação de Grãos em Potencial
AGR	Analisador de Grãos
AGL	Analisador Global
AIG	Anotação de Informações de Grãos
CC	Complexidade das Cláusulas
CGE	*Conditional Graph Expression*
CRPL	Complexidade das Resolvantes Pendentes Locais
DG	Determinação de Granulosidade
DGR	Determinação da Granulosidade
DTS	Determinação do Tamanho das Saídas
ECL	Entrada de Custo limite
ETL	Entrada de Tamanho Limite
GAC	Gerador de Anotação de Complexidade
GAG	Gerador de Anotação de Grãos
GD	Grafo de Dependências
GRANLOG	*Granularity Analyzer for Logic Programming*
LDP	Lista Descritiva do Programa
LEI	LIsta de Expressões Incondicionais
LGD	Lista de Grafos de Dependência
PC	Potencial de Compartilhamento
POP	Pilha de Objetivos Paralelos
RAP	*Restricted AND Parallelism*
TABV	Tabela de Variáveis
UGE	*Unconditional Graph Expression*
WAM	*Warren's Abstract Machine*

Resumo

A exploração do **paralelismo na programação em lógica** é considerada uma alternativa para simplificação da programação de máquinas paralelas e para aumento do desempenho de programas em lógica. Desta forma, a integração da programação em lógica e sistemas paralelos tornou-se nos últimos anos um centro de atenções da comunidade científica.

Dentre os problemas que devem ser solucionados para exploração adequada do paralelismo, encontra-se a **análise de granulosidade**. A análise de granulosidade determina o tamanho dos **grãos**, ou seja, a complexidade dos módulos que deverão ser executados seqüencialmente num único processador. Basicamente, esta análise consiste de uma refinada identificação dos grãos, visando a máxima eficiência na exploração do paralelismo. Neste sentido, devem ser realizadas considerações sobre dependências, complexidade dos grãos e custos envolvidos na paralelização. Recentemente, a análise de granulosidade na programação em lógica tem recebido atenção especial por parte dos pesquisadores.

Os grãos podem ser identificados pelo programador através de primitivas de programação ou podem ser detectados automaticamente pelo sistema paralelo. Na programação em lógica, a **exploração automática do paralelismo** é estimulada, devido ao **paralelismo implícito** existente na avaliação das expressões lógicas. Além disso, a programação em lógica permite uma clara distinção entre a semântica e o controle da linguagem, proporcionando uma abordagem distinta entre a descrição do problema e o caminho para obtenção das soluções. A detecção automática do paralelismo permite o aproveitamento de programas já existentes, além de liberar o programador do encargo de paralelizar o problema.

Este trabalho dedica-se ao estudo da **análise automática de granulosidade na programação em lógica**. O texto propõe um modelo para geração de informações de granulosidade, denominado **GRANLOG** (**GR**anularty **AN**alyzer for **LOG**ic Programming). O GRANLOG realiza uma análise estática de um programa em lógica. Dessa análise resulta o **programa granulado**, ou seja, o programa original acrescido da **anotação de granulosidade**. Esta anotação contém diversas informações que contribuem de forma significativa com a exploração adequada do paralelismo na programação em lógica.

Durante o desenvolvimento do GRANLOG foram exploradas diversas áreas de pesquisa da programação em lógica, dentre as quais destacam-se: análise de modos, análise de tipos, análise de medidas para mensuração do tamanho de termos, interpretação abstrata, análise de dependências e análise de complexidade. A integração destes tópicos torna o GRANLOG uma rica fonte de pesquisa. Além disso, a organização modular da proposta permite o aprimoramento independente de suas partes, tornando a estrutura do modelo uma base para o desenvolvimento de novos trabalhos.

Além do modelo, o texto descreve a implementação de um protótipo e propõe duas aplicações para as informações de granulosidade, ou seja, auxílio a decisões de escalonamento e simulação da execução de programas. O texto apresenta ainda uma proposta para integração do GRANLOG a um modelo para execução paralela de programas em lógica, denominado OPERA. O OPERA dedica-se a exploração do paralelismo na programação em lógica e possui atualmente um protótipo para execução

paralela de programas em lógica em redes de computadores. Os bons resultados obtidos com a integração OPERA-GRANLOG demonstram a relevância das informações geradas pelo modelo proposto neste trabalho.

Encontra-se ainda neste texto uma proposta para inclusão do GRANLOG numa interface gráfica, denominada XOPERA. Esta interface permite a execução do protótipo OPERA e, a partir deste trabalho, gerencia também o protótipo GRANLOG. A inclusão da gerência do GRANLOG na interface XOPERA, contribui de forma substancial para a integração OPERA-GRANLOG.

PALAVRAS-CHAVES: Granulosidade, Análise de Granulosidade, Processamento Paralelo, Paralelismo na Programação em Lógica

TITLE: "GRANLOG: A MODEL FOR AUTOMATIC GRANULARIY ANALYSIS IN LOGIC PROGRAMMING"

Abstract

The exploitation of parallelism in logic programming is considered an alternative for simplifying the task of programming parallel machines. Also, it provides a way to increase the performance of logic programs. Because of this, integrating parallel systems with parallel programming has been a topic of much interest in the scientific comunity, in the last years.

Among the problems that must be solved for the adequate exploitation of parallelism, there is the granularity analysis. Granularity analysis determines the size of the grains, that is, the complexity of the modules that must be sequentially executed in a single processor. Basically, this analysis consists of a refined identification of the grains, aiming the maximum efficiency in the parallelism exploitation. In this sense, considerations must be taken about dependencies, grain complexity and costs involved in the parallelizing process. Recently, many researchers have given special attention to the granularity analysis of logic programming.

The grains may be identified by the programmer via programming primitives, or they may be automatically detected by the parallel system. In logic programming, the automatic exploitation of parallelism is stimulated, because of the implicit parallelism that exists in the evaluation of the logic expressions. Besides, logic programming allows a clear distinction between the semantics and the control of the language, providing a distinct approach between the problem description and the way to obtain the results. The automatic detection of parallelism permits the utilization of already written programs, also freeing the programmer from parallelizing the program by hand.

This work is dedicated to the study of automatic granularity analysis in logic programming. The text proposes a model for generating granularity informations, called GRANLOG (GRanularity Analyzer for LOGic Programming). GRANLOG performs a static analysis of a logic program. From this analysis, it results a granulated program, that is, the original program increased by the granularity annotation. This annotation has several informations that contribute in a significant way to the adequate exploitation of parallelism in logic programming.

During the development of GRANLOG, several research areas have been explored, namely, mode analysis, type analysis, measure analysis for measuring the size of terms, abstract interpretation, dependencies analysis and complexity analysis. The integration of these topics makes GRANLOG a good source for researchs. Besides, the modular organization proposed permits the independent improvement of its parts, making of the model structure, a base for the development of new works.

Besides the model, the text describes the implementation of a prototype and proposes two applications for the granularity informations, namely, help in scheduling decisions and program execution simulation. It also presents a proposal for integrating GRANLOG to a parallel logic execution model for logic programming, called OPERA. OPERA is dedicated to the exploitation of parallelism in logic programming and, at the present time, has a prototype for parallel execution of logic programming in computer

networks. The good results obtained by integrating OPERA and GRANLOG show the importance of the information generated by the model proposed in this work.

There is, also, in this work, a proposal for including GRANLOG in a graphical interface, called XOPERA. This interface allows the execution of the OPERA prototype and, from now on, also manages the GRANLOG prototype. The inclusion of GRANLOG in the XOPERA interfaces substantially contributes to the OPERA-GRANLOG integration.

KEYWORDS: Granularity, Granularity Analysis, Parallel Processing, Parallelism in Logic Programming.

1 Introdução

1.1 Tema

O tema desta dissertação é a análise automática de granulosidade na programação em lógica. A abordagem deste tema estabelece o estudo de diversas áreas de pesquisa relacionadas com a programação em lógica, dentre as quais destacam-se: interpretação abstrata, análise de dependências, análise de modos, análise de tipos, análise de medidas para mensuração de termos, análise de complexidade e finalmente, a exploração do paralelismo nos programas em lógica. Visando concretizar este estudo, é proposto um modelo para geração de informações de granulosidade para programas em lógica, denominado GRANLOG (GRanularity ANalyzer for LOGic programming).

1.2 Motivação

Nos últimos anos o processamento paralelo vem sendo indicado como uma das melhores soluções para aumento do desempenho dos computadores. Dentre as alternativas atualmente consideradas, o paralelismo apresenta uma das melhores relações custo/benefício. No entanto, a maioria dos sistemas computacionais existentes ainda são baseados no paradigma de processamento seqüencial. Além disso, a cultura do processamento seqüencial está difundida na comunidade científica e acadêmica, gerando uma inércia natural que deve ser vencida para disseminação do processamento paralelo. A adoção do paralelismo como solução para aumento do desempenho dos computadores dependerá das facilidades de utilização deste novo paradigma.

A utilização do paralelismo exige a pesquisa e a construção de novas arquiteturas de computadores (*hardware*) e novos programas (*software*). Atualmente, a existência de uma grande variedade de arquiteturas paralelas estimula a utilização do paralelismo como solução para evolução dos computadores. No entanto, o desenvolvimento de *softwares* paralelos não acompanha a evolução das arquiteturas paralelas, estabelecendo um empecilho para adoção definitiva do paralelismo. Este problema foi denominado pela comunidade científica de a **crise do *software* paralelo**. Em grande parte, esta crise é gerada pela complexidade adicional introduzida no desenvolvimento de programas para computadores paralelos. Esta complexidade resulta da necessidade de tratamento de novos problemas introduzidos pelo paralelismo. Dentre os principais problemas que devem ser solucionados para exploração eficiente do paralelismo, encontra-se a análise de granulosidade, ou seja, a determinação do tamanho adequado dos módulos que serão executados seqüencialmente num único processador (grãos).

Do ponto de vista do usuário, existem basicamente duas abordagens para a análise de granulosidade, ou seja, a análise realizada pelo programador (paralelismo explícito) e a análise realizada automaticamente pelo sistema (paralelismo implícito). Na primeira abordagem, o usuário utiliza linguagens paralelas que possuam primitivas de paralelização ou linguagens convencionais acrescidas de bibliotecas para exploração do paralelismo. Por outro lado, na segunda abordagem, o usuário não participa da paralelização do problema, ficando esta tarefa sob responsabilidade do sistema. Sendo assim, na análise automática as linguagens de programação não incorporam qualquer estrutura para exploração do paralelismo.

Conforme afirmam Pancake ([PAN91]) e Barbosa ([BAR93]), grande parte dos usuários de computadores não possuem interesse em envolver-se na paralelização de suas aplicações, pois o paralelismo aumenta a complexidade do desenvolvimento e depuração dos programas. Na verdade, os usuários desejam apenas o aumento de desempenho proporcionado pelo processamento paralelo. Portanto, a análise automática de granulosidade é o caminho natural para utilização do paralelismo. Além disso, a exploração transparente do paralelismo permite a paralelização de programas seqüenciais existentes, estimulando desta forma a adoção do paradigma paralelo.

Constata-se nos últimos anos um crescente interesse na exploração do paralelismo em linguagens baseadas em paradigmas de programação não convencionais, tais como a programação em lógica, a programação funcional e a programação orientada a objetos. Este fato deve-se, em grande parte, às dificuldades encontradas na exploração do paralelismo nas linguagens convencionais (linguagens imperativas). As linguagens convencionais dificultam a exploração automática do paralelismo, pois obrigam o programador a explicitar o fluxo de execução do programa, gerando assim, uma série de dependências de dados e de controle. A dificuldade para exploração automática do paralelismo nestas linguagens é considerada uma das principais causas da crise do *software* paralelo.

Por outro lado, as linguagens não convencionais fornecem aos programadores modelos conceituais que possibilitam sua dedicação à especificação do problema computacional, independentemente da forma com que será realizado seu processamento. Deve-se ressaltar ainda, que os paradigmas não convencionais introduzem características que aumentam o potencial de exploração do paralelismo implícito, ou seja, facilitam a exploração automática do paralelismo.

Segundo Yamin ([YAM92]), a programação em lógica, devido ao paralelismo implícito na avaliação das expressões lógicas, libera o programador do encargo de gerenciar explicitamente o paralelismo do problema. Além disso, a programação em lógica permite uma clara distinção entre a semântica e o controle da linguagem, proporcionado uma abordagem distinta entre a descrição do problema e o caminho para obtenção das soluções. Estas características possibilitam a programação de computadores paralelos, em nível de dificuldade semelhante à da programação seqüencial, e ainda facilitam a migração de programas entre máquinas paralelas e seqüenciais. Neste sentido, também as afirmações de Bianchini ([BIA90]) confirmam que a separação da semântica e controle, inerente à programação em lógica, e o crescente domínio da tecnologia de processamento paralelo fizeram com que a execução paralela de programas em lógica tenha se tornado um centro de atenções.

Deve-se ressaltar ainda, que as linguagens de programação em lógica caracterizam-se pela sua ineficiência de execução ([PAA88], [REI88]), a qual pode ser atenuada com a utilização de sistemas paralelos. Portanto, pode-se citar como principais objetivos da integração da programação em lógica e processamento paralelo:

- simplificar a programação de computadores paralelos, contribuindo assim para solucionar a crise do *software* paralelo;
- aumentar o desempenho da programação em lógica, contribuindo desta forma para aumentar sua aceitação junto à comunidade de usuários.

A união do paralelismo e da programação em lógica vem sendo considerada pela comunidade científica como uma promissora alternativa para o desenvolvimento dos

futuros computadores. O Projeto Japonês de Computadores de Quinta Geração ([SHA93]) obteve resultados estimulantes com a exploração do paralelismo na programação em lógica. A maioria das linhas de pesquisa, que exploram a união da programação em lógica e do processamento paralelo, estudam como paralelizar programas em linguagem Prolog, existindo um grande número de artigos publicados, protótipos implementados e alguns produtos disponíveis no mercado.

Destaca-se ainda como fonte de motivação, a possibilidade de integração deste trabalho ao projeto OPERA ([BRI90], [BRI90a], [GEY92], [YAM93], [WER94], [WER94a], [YAM94]). O projeto OPERA iniciou suas atividades no Laboratório de Génie Informatique (Universidade de Joseph Fourier em Grenoble / França). Atualmente, encontra-se em desenvolvimento na UFRGS uma ramificação deste projeto. Nessa ramificação procura-se integrar o paralelismo AND restrito ([DEG84], [DEG87], [DEB89]) e o paralelismo OU multiseqüencial ([GEY91]). A integração do presente trabalho ao projeto OPERA, possibilita seu desenvolvimento num contexto de pesquisa atualizado, onde os resultados alcançados são aplicados de forma imediata. Além disso, os resultados obtidos são utilizados para expansão do protótipo OPERA, contribuindo para a ampliação do projeto e servindo de estímulo para o desenvolvimento de novos trabalhos.

1.3 Objetivos

O objetivo geral do trabalho é desenvolver um estudo sobre a análise automática de granulosidade na programação em lógica. Este estudo está baseado na proposta de um modelo para geração de informações de granulosidade e na demonstração da importância destas informações para exploração eficiente do paralelismo na programação em lógica.

Pode-se citar como objetivos específicos desse trabalho:

- Pesquisar genericamente a análise de granulosidade e especificamente sua exploração na programação em lógica;
- Propor um modelo para análise automática de granulosidade na programação em lógica, denominado GRANLOG;
- Modelar e implementar um protótipo para validação da proposta;
- Propor a utilização do GRANLOG em duas aplicações, ou seja, auxílio a decisões de escalonamento e simulação da execução de programas;
- Integrar o GRANLOG ao projeto OPERA, utilizando as informações de granulosidade no protótipo OPERA e incluindo o GRANLOG na interface XOPERA.

1.4 Contribuição do Autor

Tendo como base os objetivos do trabalho e as atividades desenvolvidas, pode-se destacar as seguintes contribuições do autor:

- estabeleceu os três princípios básicos que orientam o desenvolvimento deste trabalho, ou seja: independência do sistema paralelo (arquitetura paralela e plataforma para execução), exploração automática do paralelismo (paralelismo

implícito) e máxima exploração do paralelismo nos programas em lógica (paralelismo E/OU). Esta etapa envolveu o estudo do estado da arte das pesquisas sobre paralelismo, visando a detecção das tendências científicas nesta área. Em função deste estudo foram estabelecidos os três princípios básicos do GRANLOG;

- propôs o GRANLOG, modelo para análise automática de granulosidade na programação em lógica. Esta atividade envolveu a pesquisa de diversas áreas de estudo da programação em lógica, tais como: interpretação abstrata, análise de complexidade, análise de dependências, análise de tipos, análise de medidas e análise de modos. Baseado neste estudo, o autor selecionou quais trabalhos da comunidade científica serviriam de base para a proposta e determinou a organização modular do GRANLOG;
- trabalhou no desenvolvimento de um protótipo para o modelo proposto. Nesta contribuição o autor participou da modelagem e implementação de um protótipo. Basicamente, esta tarefa consistiu na definição de estruturas de dados, algoritmos e organização do sistema;
- demonstrou aplicações para o modelo proposto. Nesta atividade o autor pesquisou diversas aplicações para informações de granulosidade. Deste estudo, resultou a proposta de duas aplicações para o GRANLOG, ou seja, auxílio a decisões de escalonamento e simulação da execução de programas;
- definiu a integração do GRANLOG e do OPERA. Esta tarefa consistiu no estudo do modelo proposto pelo projeto OPERA e na definição de sua integração com o GRANLOG;
- trabalhou na adaptação do protótipo OPERA. Nesta atividade o autor participou da adaptação do protótipo OPERA para utilização das informações geradas pelo GRANLOG. O autor participou ainda da realização de testes e análise dos resultados;
- participou da expansão da interface XOPERA. Nesta contribuição, o autor definiu a inclusão do GRANLOG na interface gráfica do projeto OPERA e participou da implementação desta expansão.

1.5 Estrutura da Dissertação

A dissertação está organizada em 9 capítulos, distribuídos de forma a introduzir gradativamente o leitor no tema proposto. Inicialmente, são apresentados conceitos que servem de suporte para leitura do restante do texto. Logo após, é descrito o modelo GRANLOG. Esta descrição abrange uma visão genérica do modelo e uma visão específica de cada uma das três etapas da análise de granulosidade. A cada uma destas etapas equivale um módulo do modelo. Finalmente, é descrito o protótipo e a integração dos modelos OPERA e GRANLOG.

O capítulo 2 introduz conceitos sobre paralelismo, programação em lógica e exploração do paralelismo na programação em lógica. Estes conceitos permitem ao leitor uma visão básica sobre o tema proposto, facilitando a compreensão dos próximos capítulos.

No capítulo 3 são apresentados os princípios básicos da proposta, uma visão genérica da sua organização e uma comparação com outros modelos. Ainda neste capítulo, são propostas duas aplicações para o GRANLOG, ou seja, auxílio a decisões de escalonamento e simulação da execução de programas.

O capítulo 4 descreve a primeira fase da análise de granulosidade, ou seja, a **análise global**. Neste capítulo é discutido o primeiro módulo do modelo, denominado **Analisador Global**. Além disso, são discutidos conceitos sobre análise de modos, análise de tipos, análise de medidas para mensuração de termos e análise de dependências. Ainda no capítulo 4 é proposto um algoritmo para análise de dependências e são apresentados os modos, tipos e medidas utilizadas pelo GRANLOG.

No capítulo 5 é descrita a segunda etapa da análise de granulosidade, ou seja, a **análise de grãos**. Este capítulo aborda o segundo módulo do modelo, denominado **Analisador de Grãos**. Ainda neste capítulo, propõem-se uma classificação para os possíveis grãos existentes na programação em lógica e um algoritmo para determinação destes grãos. Destaca-se ainda no capítulo 5, a descrição da **anotação de grãos** gerada pelo GRANLOG.

O capítulo 6 aborda o terceiro e último estágio da análise de granulosidade, ou seja, a **análise de complexidade**. Este capítulo apresenta o terceiro módulo do GRANLOG, denominado **Analisador de Complexidade**. O capítulo discute ainda conceitos relacionados com análise de complexidade, propõe uma classificação para tipos de complexidade na programação em lógica e descreve as metodologias propostas para realização da análise. Como complemento, é apresentada a **anotação de complexidade** gerada pelo GRANLOG.

No capítulo 7 é descrito o protótipo. Este capítulo apresenta as decisões de implementação, a organização do sistema, as simplificações em relação ao modelo proposto, as estruturas de dados e algoritmos.

O capítulo 8 apresenta a integração do GRANLOG e do OPERA. Neste capítulo é descrito o modelo proposto no projeto OPERA, as alterações necessárias para a integração OPERA-GRANLOG e os resultados alcançados.

Finalmente, no capítulo 9 estão as conclusões deste trabalho.

2 Programação em Lógica e Paralelismo

Neste capítulo são apresentados conceitos básicos e definições que servem de suporte para o restante do texto. O seção 2.1 descreve tópicos relacionados com paralelismo. Dentre estes tópicos destacam-se as subseções sobre análise de granulosidade e exploração do paralelismo em diferentes paradigmas de programação. Na seção 2.2 são apresentados conceitos sobre programação em lógica. A seção 2.3 aborda a exploração do paralelismo na programação em lógica. Destaca-se a subseção 2.3.4, que resume e discute propostas para análise de granulosidade nos programas em lógica. Finalmente, na seção 2.4 são apresentadas as conclusões deste capítulo.

2.1 Paralelismo

Esta seção apresenta tópicos relacionados com a exploração do paralelismo no universo dos computadores. Aconselha-se o clássico de Hwang ([HWA84]) e o recente livro de Moldovan ([MOL93]) para aprofundamento nos estudos sobre paralelismo e os livros de Quinn ([QUI87]), Jaja ([JAJ92]) e Leighton ([LEI92]) para estudos relacionados com algoritmos paralelos. Como complemento, encontra-se em [WIL93] um glossário da terminologia de processamento paralelo.

2.1.1 Problemas na Exploração do Paralelismo

Segundo Kruatrachue e Lewis ([KRU88]), a exploração eficiente do paralelismo existente em um algoritmo depende do seu particionamento em módulos e no escalonamento destes módulos entre os processadores do sistema. No texto [KRU88] este problema é dividido em duas abordagens distintas:

- Problema do particionamento do programa: Como particionar um programa em módulos a serem executados em paralelo, visando o menor tempo de execução, e consequentemente, qual o tamanho adequado para cada um destes módulos ?
- Problema do escalonamento dos módulos no sistema paralelo: Como distribuir os módulos pelos processadores do sistema, aproveitando ao máximo a arquitetura disponível ?

Também Mccreary e Gill ([MCC89]) afirmam que o principal problema para execução paralela de programas consiste na determinação eficiente dos módulos paralelos e no escalonamento destes módulos de forma a minimizar o tempo de execução.

De forma complementar, Hwang ([HWA84]) destaca que a efetiva utilização do paralelismo está limitada pela falta de metodologias para projeto de programas paralelos. Neste sentido, Hwang afirma ainda que o principal problema consiste na decomposição do algoritmo para execução paralela. Esta decomposição pode ser dividida em duas abordagens, ou seja, particionamento e atribuição. O particionamento consiste na divisão do algoritmo em procedimentos, módulos e processos. A atribuição refere-se a alocação destas unidades nos processadores. Finalmente, Hwang afirma que estes problemas residem entre os mais difíceis e mais importantes do processamento paralelo.

Destaca-se ainda, o trabalho de Sarkar ([SAR89]) que identifica três problemas fundamentais a serem resolvidos para exploração do paralelismo, ou seja:

- Identificação do paralelismo no programa;
- Particionamento do programa em tarefas seqüenciais;
- Escalonamento das tarefas nos processadores.

Constata-se em [SAR89] que Sarkar salienta a **identificação do paralelismo**, distinguindo este problema do **particionamento do programa**. A identificação do paralelismo consiste na determinação das partes do programa que podem ser executadas em paralelo. Esta determinação é realizada através da análise de dependências.

Conforme afirma Moldovan ([MOL93]), o paralelismo existente no programa depende da natureza do problema e do algoritmo utilizado pelo programador. A identificação do paralelismo é usualmente independente da máquina paralela. Por outro lado, o particionamento e escalonamento são projetados para minimizar a execução do programa num computador paralelo e normalmente dependem de parâmetros, tais como número de processadores, desempenho dos processadores, custo de comunicação, custo de escalonamento e outros.

Os cinco textos referenciados nesta subseção destacam os problemas introduzidos pelo paralelismo. Estes problemas devem ser solucionados para viabilização dos computadores paralelos. Considerando os pontos de vista apresentados em [HWA84], [KRU88] e [MCC89], o presente trabalho dedica-se ao estudo e a solução do primeiro problema na exploração do paralelismo, ou seja, o particionamento do programa. Por outro lado, considerando o ponto de vista de [SAR89] e [MOL93], o presente estudo dedica-se a solução dos dois primeiros problemas da exploração do paralelismo, ou seja, identificação do paralelismo e particionamento do programa. Na verdade, a análise de granulosidade, tema desta dissertação, dedica-se a determinação dos grãos através da identificação do paralelismo e particionamento do programa. A próxima subseção discute o conceito de grão e define análise de granulosidade no contexto deste trabalho.

2.1.2 Análise de Granulosidade

A análise de granulosidade destaca-se como um dos principais problemas a serem solucionados para exploração eficiente do paralelismo. As técnicas para solução deste problema variam de acordo com o paradigma de programação em estudo. No decorrer dos últimos anos, surgiram vários trabalhos sobre a análise de granulosidade no paradigma convencional ([KRU88], [MCC89], [SAR89], [GIR92]) e no paradigma funcional ([HUD85], [RAB90]). Recentemente, a análise de granulosidade na programação em lógica tem recebido atenção especial por parte da comunidade científica ([TIC88], [DEB90], [KIN90], [TIC90], [KIN92], [ZHO92], [DEB93], [TIC93], [GAR94]). Além disso, deve-se ressaltar que os estudos nos paradigmas convencional e funcional serviram de base para as pesquisas na programação em lógica. A subseção 2.3.4 aborda especificamente a análise de granulosidade nos programas em lógica. Por sua vez, o texto [BAR94] resume e compara várias propostas para análise de granulosidade no paradigma convencional e paradigma da programação em lógica.

Na definição de Kruatrachue e Lewis ([KRU88]), um grão é uma tarefa composta por um ou mais módulos de um programa, a qual será executada num único processador. Um grão inicia sua execução tão logo disponha de todas suas entradas e termina a execução somente após obter todas suas saídas. Em [MCC89], grão é definido como um conjunto de passos de um programa, os quais serão executados seqüencialmente por um

único processador. Em ambos os textos, encontra-se a idéia de particionamento inerente ao conceito de grão, ou seja, o conceito de grão está vinculado com o particionamento de um programa para sua execução paralela. Portanto, neste trabalho, define-se **grão** como uma tarefa resultante do particionamento de um programa, a qual deverá ser executada seqüencialmente num único processador. Esta definição torna clara a atomicidade dos grãos, ou seja, os grãos são os átomos (unidades indivisíveis) componentes do programa paralelo.

Define-se **granulosidade** como o tamanho do grão, ou seja, o esforço computacional necessário para o processamento do grão (complexidade do grão). Portanto, a **análise de granulosidade** consiste na análise do tamanho dos grãos, ou seja, a determinação da complexidade adequada para os módulos que deverão ser executados seqüencialmente num único processador. Na verdade, esta análise consiste basicamente numa refinada identificação dos grãos, visando a máxima eficiência na exploração do paralelismo. Neste sentido, **devem** ser realizadas considerações sobre dependências, complexidade dos grãos e custos envolvidos na paralelização.

Neste ponto, deve-se ressaltar a definição de Hwang ([HWA84]) para o particionamento. Hwang afirma que o particionamento consiste na especificação do conjunto de unidades de programa que implementam da **maneira mais eficiente** um algoritmo numa determinada arquitetura paralela. Essa especificação determina o tamanho adequado das unidades de programa (grãos), ou seja, consiste na análise de granulosidade. Destaca-se na definição de Hwang, a ênfase na máxima eficiência inerente ao particionamento. O particionamento de um programa somente poderá ser o mais eficiente, quando levar em consideração aspectos sobre complexidade dos grãos e custos de paralelização. Nem todas técnicas de particionamento consideram estes fatores. Desta forma, a definição de **análise de granulosidade** confunde-se com o **particionamento visando máxima eficiência** definido por Hwang, podendo-se inclusive afirmar que as duas definições são equivalentes. Sendo assim, conclui-se que toda análise de granulosidade é um particionamento (visando máxima eficiência), mas nem todo particionamento é uma análise de granulosidade.

A determinação da granulosidade adequada depende de uma série de características. Se o grão é muito grande, o número de grãos tende a ser pequeno, podendo ocasionar perda do potencial de paralelismo do programa. Por outro lado, se o grão é muito pequeno, o número de grãos tende a ser grande, podendo ocasionar uma perda de desempenho devido aos altos custos de comunicação. Portanto, a execução eficiente de um programa paralelo, depende do dimensionamento adequado dos grãos. Esta é a tarefa da análise de granulosidade.

Na medida em que distribui-se os grãos para o maior número de processadores disponíveis, com o intuito de não deixá-los ociosos, aumenta-se a tendência para um atraso gerado pela maior necessidade de comunicação (sincronização e informação). Desta forma, a obtenção do melhor desempenho depende de uma negociação entre a máxima paralelização e a mínima comunicação. A esta negociação, Kruatrachue e Lewis ([KRU88]) denominam **problema max-min** (máxima paralelização / mínima comunicação). Em [HWA84] este problema recebe o nome **negociação computação-comunicação**. O principal objetivo da análise de granulosidade é resolver este problema, através do equilíbrio entre paralelização e comunicação.

Segundo Sarkar ([SAR89]), a análise de granulosidade pode ser realizada em tempo de compilação (**análise estática**), tempo de execução (**análise dinâmica**) ou em

ambos (**análise combinada**). Sendo realizada em tempo de compilação, a análise não introduz custo computacional (*overhead*) na execução. No entanto, a complexidade de um grão normalmente depende do tamanho de suas entradas, o que não pode ser previsto antes da execução. Portanto, a análise estática não permite uma determinação precisa da granulosidade. A análise dinâmica é mais precisa, introduzindo no entanto um considerável custo na execução do programa. Recentes abordagens da análise de granulosidade indicam que a análise combinada (combinação da análise estática e dinâmica) produz resultados mais promissores. Esta metodologia pretende o equilíbrio entre as análises estática e dinâmica, visando baixo custo adicional na execução e alta precisão nas informações de granulosidade.

2.1.3 Paralelismo Explícito e Implícito

Conforme afirmam Mccreary e Gill ([MCC89]), do ponto de vista do usuário existem duas abordagens para exploração do paralelismo:

- Detecção do paralelismo pelo programador (paralelismo explícito);
- Detecção automática do paralelismo (paralelismo implícito).

No paralelismo explícito, a linguagem de programação contém mecanismos para paralelização do programa. Neste caso, o programador utiliza linguagens paralelas ou linguagens seqüenciais estendidas para suportar o paralelismo. Desta forma, o programador pode utilizar seu conhecimento empírico para explorar ao máximo o potencial de paralelização de suas aplicações. No entanto, afirmam Kruatrachue e Lewis ([KRU88]) que a utilização de mecanismos explícitos pode levar a uma exploração inadequada do potencial de paralelismo. Além disso, conforme ressalta Karp ([KAR88]), grande parte do trabalho necessário para paralelização de programas é muito difícil para ser realizado por pessoas. Por exemplo, Karp afirma que somente compiladores são confiáveis para realização da análise de dependências em sistemas paralelos com memória compartilhada. Por outro lado, deve-se ressaltar que o paralelismo explícito diminui a complexidade dos compiladores paralelizadores, pois elimina a necessidade da detecção automática do paralelismo em tempo de compilação.

No paralelismo implícito, a linguagem de programação não contém mecanismos para paralelização dos programas. Nesta abordagem, o programador utiliza linguagens seqüenciais e a exploração do paralelismo fica sob responsabilidade do sistema computacional. A principal vantagem deste método consiste na liberação do programador do envolvimento com a paralelização de suas aplicações. Além disso, o paralelismo implícito aumenta a portabilidade de programas entre sistemas paralelos, eliminando a necessidade da alteração do código fonte em função da arquitetura paralela a ser utilizada. Outra característica interessante da exploração automática consiste no aproveitamento tanto dos programas seqüenciais já existentes quanto dos ambientes de desenvolvimento (depuração) direcionados para o paradigma seqüencial.

Analisando-se as vantagens do paralelismo explícito e implícito, constata-se que ambos os métodos possuem méritos e devem continuar sendo pesquisados. No entanto, dois pontos relacionados com a exploração automática devem ser ressaltados:

- as vantagens obtidas na exploração automática do potencial de paralelismo inerente às linguagens não convencionais (programação em lógica, programação

orientada a objetos e programação funcional), as quais cada vez mais são destacadas pela comunidade científica;

- a tendência para simplificação do uso dos computadores e especificamente a necessidade de simplificar o desenvolvimento de programas para computadores paralelos. Sabe-se que a complexidade de desenvolvimento de programas para plataformas paralelas é um das principais causas da **crise do *software* paralelo** (atraso do desenvolvimento do *software* paralelo em relação ao *hardware* paralelo).

O primeiro ponto será discutido genericamente na subseção 2.1.4 e especificamente para a programação em lógica na seção 2.3. O segundo ponto assume uma posição de destaque na medida em que interfere na tendência das pesquisas para criação de novas linguagens e no desenvolvimento de novos sistemas para computadores paralelos. O segundo ponto será discutindo nos próximos parágrafos.

Segundo Pancake ([PAN91]) e Barbosa ([BAR93]), os usuários de computadores paralelos não estão interessados em envolver-se na exploração do paralelismo, desejando apenas o aumento de desempenho proporcionado por esta nova tecnologia. O envolvimento com o paralelismo aumenta a complexidade de desenvolvimento e depuração de programas. Sendo assim, a exploração automática do paralelismo torna-se a opção mais interessante. Desta forma, espera-se que a exploração do paralelismo siga a tendência natural de simplificar a utilização dos computadores, a qual vem sendo aplicada na maioria das áreas da computação (interfaces gráficas para sistemas operacionais, sistemas para gerenciamento de dados e outros).

Inevitavelmente, ficando a paralelização a cargo do sistema, a complexidade dos compiladores aumenta, o que exige a utilização de computadores mais poderosos para viabilização da análise e tradução dos programa fontes em tempos razoáveis. No entanto, a aplicação do paralelismo nos projetos de arquiteturas de computadores vem possibilitando um aumento significativo na velocidade dos sistemas computacionais. Além disso, verifica-se atualmente um crescente aumento da capacidade de armazenamento em disco magnético e na memória principal. Sendo assim, o aumento da complexidade dos compiladores não representará obstáculo na tendência para exploração automática do paralelismo. Neste contexto, deve-se salientar que a **análise automática de granulosidade** permitirá a criação de sistemas paralelos eficientes e de simples utilização. Estas características são atraentes para o mercado de computadores e tendem a estimular a utilização do processamento paralelo.

2.1.4 Exploração do Paralelismo nos Paradigmas de Programação

Segundo Sterling e Shapiro ([STE86]), embora os computadores tenham sido criados para interagir com seres humanos, as dificuldades para construi-los fizeram com que as primeiras linguagens de programação fossem projetadas a partir da perspectiva dos engenheiros de computadores. Desta forma, estas linguagens sofreram forte influência dos conceitos de Von Neumann, os quais são utilizados para construção da maioria destes equipamentos. Os usuários que não possuem conhecimento da arquitetura Von Neumann, encontram dificuldades na utilização destas linguagens. Além disso, devido a sua orientação para o *hardware*, estas linguagens diferem substancialmente da forma com que o homem pensa e expressa seus conhecimentos. As linguagens de programação que enquadram-se nesta perspectiva são denominadas **linguagens**

convencionais. Este paradigma de programação recebe ainda as seguintes denominações: **programação imperativa**, **programação procedimental** ou **programação tradicional**.

Na medida em que aumentava a compreensão dos problemas envolvidos na criação dos computadores, a atenção dos cientistas voltava-se para os problemas relacionados com a utilização destes. Desta forma, a procura por linguagens de programação mais adequadas ao homem assumiu uma posição de destaque. Grande parte dos esforços foram direcionados para criação de novos paradigmas de programação que permitissem ao usuário expressar o problema de forma mais humana e menos direcionada para a máquina. Assim, nasceram os **paradigmas de programação não convencionais**, tais como a **programação declarativa** e a **programação orientada a objetos**. A programação declarativa pode ser dividida basicamente em dois grupos, ou seja, **programação funcional** (baseada no Cálculo Lambda ([CHU41]) e inicialmente implementada na linguagem LISP ([MCJ65])) e **programação em lógica** (baseada na lógica ([CAS86], [ROB92]), proposta em [KOW74], inicialmente implementada na linguagem Prolog ([ROU69]) e reconhecida como ferramenta prática de programação através do compilador/intepretador Prolog proposto por Warren ([PER78])). A figura 2.1 resume a organização dos paradigmas de programação

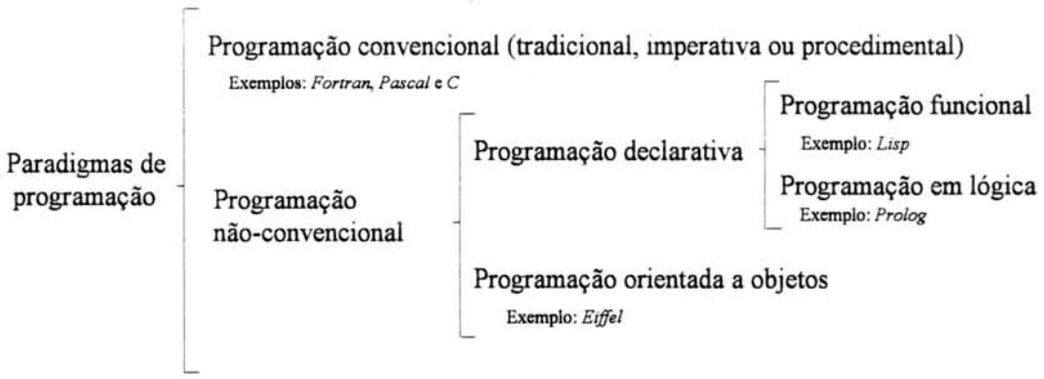


FIGURA 2.1 - Paradigmas de programação

As linguagens orientadas a objeto propõem a diminuição da distância (*gap semântico*) existente entre a modelagem computacional e o mundo real. Neste sentido, surgem os **objetos** e suas diversas características (herança, polimorfismo, encapsulamento, etc). Desta forma, a orientação a objetos alcança um nível de abstração que aproxima a programação de computadores à forma usual do homem interagir com o mundo. Por sua vez, as linguagens declarativas propoem a programação de computadores através da descrição formal da estrutura do problema. Através deste formalismo, o programador expressa seu conhecimento a respeito do problema, sem preocupar-se com o fluxo de execução. Este conhecimento pode ser expresso na forma de funções (programação funcional) ou como sentenças da lógica de predicados de primeira ordem (programação em lógica). Alguns dos benefícios introduzidos pelas linguagens declarativas são apresentados por Backus em [BAC78].

O surgimento do paralelismo trouxe esperanças para o aumento do desempenho dos computadores, o qual estava ameaçado pelos limites impostos pela utilização de apenas um processador. Com a utilização da tecnologia disponível para construção de computadores seqüenciais, foram criadas arquiteturas paralelas que estimularam a

utilização do paralelismo. Seguindo esta tendência, utilizou-se a tecnologia disponível em linguagens seqüenciais para criar linguagens paralelas. A atenção voltou-se para exploração do paralelismo em linguagens imperativas, as quais dominavam o mercado de computadores. No entanto, deste cedo notou-se que a utilização de linguagens imperativas para programação de arquiteturas paralelas possui algumas limitações.

As linguagens imperativas possuem uma proposta de **controle explícito**, ou seja, o usuário deve descrever os caminhos que levam à solução do problema. Isto é feito pela descrição do fluxo de execução através de comandos de controle. A utilização deste paradigma na programação paralela cria uma tendência ao controle explícito do paralelismo. Mesmo assim, existem vários pesquisadores que dedicam-se a exploração automática do paralelismo em linguagens imperativas ([KRU88], [MCC89], [SAR89]). Conforme discutido na subseção 2.1.3, a utilização do paralelismo explícito introduz dificuldades no desenvolvimento e depuração de programas.

Um dos principais problemas para paralelização de programas consiste nas **dependências** de dados e de controle ([PAD86], [GAL91], [ZIM91], [BAR93], [BAR94]). Nas linguagens imperativas a criação de dependências é estimulada pela proposta de controle explícito e pelo escopo global de variáveis. Normalmente nestes casos, a análise de dependências torna-se complexa, dificultando a paralelização dos programas.

Por outro lado, os paradigmas de programação não convencionais possuem características que solucionam naturalmente os problemas criados pelo controle explícito e pelas dependências. Desta forma, estes paradigmas tornam-se atraentes para exploração do paralelismo. Uma das principais características dos paradigmas não convencionais reside na nova proposta de modelos conceituais. Baseados neste modelos, os programadores dedicam-se à especificação do problema computacional, independentemente da forma como será realizado o processamento. Desta forma, estes paradigmas propõem um **controle implícito** da execução dos programas. Neste sentido, a organização destas linguagens tornou-se naturalmente modular, diminuindo o escopo das variáveis e tornando transparente o controle do fluxo de execução. Estas características facilitam o tratamento das dependências de dados e de controle.

A orientação a objetos introduz o conceito de encapsulamento e mensagens, atingindo altos níveis de modularidade e níveis de abstração que tornam o controle implícito. A programação funcional introduz o conceito de função. Em linguagens funcionais puras, uma função é chamada com entradas e retorna um resultado, não podendo ocasionar efeitos colaterais, tais como, alteração de variáveis globais. Desta forma, atinge-se um alto nível de modularidade e restringe-se as dependências. Na programação em lógica, a avaliação das expressões lógicas torna o controle implícito, fazendo com que o programador dedique-se apenas ao problema e não à forma como ele será resolvido. Não existem comandos de controle do fluxo de execução. Em linguagens de programação em lógica puras, uma chamada de procedimento não ocasiona efeitos colaterais. Além disso, nesse paradigma o escopo das variáveis está restrito a apenas uma cláusula, facilitando o tratamento das dependências.

Deve-se ressaltar ainda, que os novos modelos conceituais dos paradigmas não convencionais introduzem novos problemas que devem ser solucionados para viabilizar a exploração do paralelismo. Por exemplo, a orientação a objetos deve solucionar o problema de herança em ambientes paralelos. Por sua vez, a programação em lógica introduz problemas na execução paralela das cláusulas de um procedimento (coerência

das pilhas da máquina abstrata) e na execução paralela das metas no corpo de um cláusula (conflitos de ligação de variáveis). No entanto, os novos problemas introduzidos pelos paradigmas não convencionais tornam-se estimulantes quando comparados com os antigos problemas da paralelização de linguagens convencionais.

Destacam-se como referências bibliográficas para estudo da exploração do paralelismo nos diversos paradigmas de programação, a revista [GEL86] e o artigo [BAL89].

2.2 Programação em Lógica

Esta seção apresenta tópicos sobre programação em lógica, relevantes para a compreensão das idéias apresentadas no restante do texto. Não são descritos conceitos básicos e não é abordada a sintaxe de nenhuma linguagem de programação. No entanto, os exemplos apresentados nesta seção e no restante do texto são baseados na linguagem Prolog ([CLO87]). Os tópicos foram selecionados de acordo com sua importância para compreensão da análise automática de granulosidade na programação em lógica. Em cada subseção é destacada a relevância do tópico para o presente estudo. Aconselha-se para estudos sobre conceitos da programação em lógica os textos [KOW79], [CAS86] e [ROB92]). O Livro de Hogger ([HOG84]) apresenta uma introdução a aspectos teóricos e de implementação da programação em lógica. Os textos [CLO81] e [STE86] servem como tutoriais para programação Prolog. Um interessante histórico da evolução da programação em lógica é apresentado em [ROB92].

2.2.1 Lógica e Controle

Segundo Hermenegildo ([HER86]), existem dois elementos na execução de programas em lógica, ou seja:

- **Programa**: Conjunto de regras e fatos fornecidos pelo usuário;
- **Avaliador**: Responsável pela avaliação do programa para derivação de conclusões consistentes com os fatos e regras estabelecidos pelo usuário.

De forma suscinta, Kowalski ([KOW79]) representa estes elementos através da seguinte equação:

ALGORITMO = LÓGICA + CONTROLE

Sendo assim, torna-se clara a distinção entre a expressão do problema (programa ou lógica) e a forma com que este será resolvido (avaliador ou controle). Portanto, o controle da execução dos programas em lógica é implícito e transparente para o usuário. O usuário utiliza a lógica para expressar o problema, abstraindo-se da execução. Por sua vez, o avaliador executa o programa na arquitetura disponível. Na maioria das vezes, esta arquitetura segue o modelo Von Neumann.

Afirma ainda Manuel Hermengildo ([HER86]), que o avaliador possui um enorme grau de liberdade (não-determinismo) na escolha dos caminhos de dedução que serão

percorridos para solucionar o problema. A política utilizada pelo avaliador para escolha destes caminhos é denominada **estratégia de controle**.

Do ponto de vista do paralelismo, a distinção entre programa e controle é conveniente e estimulante. Esta distinção possibilita a criação de avaliadores com diferentes tipos de estratégias. Uma destas estratégias pode incluir a exploração do paralelismo na execução dos programas. Desta forma, a exploração do paralelismo fica restrita ao controle e portanto transparente ao usuário. O usuário dedica-se ao problema (programa) e o avaliador explora o paralelismo.

2.2.2 Semântica Declarativa e Procedimental

Conforme descrito em [STE86], a semântica atribui significado para um programa e permite uma compreensão do seu comportamento. Basicamente, existem duas interpretações semânticas para um programa em lógica, ou seja, **semântica declarativa** e **semântica procedimental**. Na semântica declarativa o programa é interpretado do ponto de vista da lógica de primeira ordem. Por outro lado, na semântica procedimental o programa é interpretado considerando-se o controle de execução, ou seja, do ponto de vista do avaliador.

Na intepretação declarativa um programa é composto por um conjunto de **predicados** lógicos. Um predicado é um grupo de **cláusulas** (**fatos** ou **regras**) que possuam o mesmo **nome de predicado** (***functor***) e o mesmo número de argumentos (**aridade**). No ponto de vista procedimental, um programa é composto por um conjunto de **procedimentos**. Nesta interpretação, cada procedimento equivale a um predicado lógico.

Na programação em lógica uma cláusula possui o seguinte formato:

$$A \leftarrow B_1, B_2, ..., B_n \qquad n \geq 0$$

onde os ***Bi's*** são objetivos.

Na intepretação declarativa uma cláusula deve ser lida da seguinte forma:

A é verdadeiro se todos os *Bi's* forem verdadeiros

Por outro lado, na semântica procedimental a cláusula é interpretada do ponto de vista do controle de execução (avaliador) e portanto deve ser lida da seguinte forma:

Execute o procedimento A chamando todos os procedimentos *Bi's*

Conforme citado na seção 2.2.1, o avaliador de programas em lógica normalmente é executado sobre uma plataforma Von Neumann. Desta forma, apesar do programador utilizar uma linguagem declarativa para expressar o problema, o avaliador deve adaptar-

se ao modelo convencional de execução. Este modelo está baseado na chamada de procedimentos. Portanto, a semântica procedimental aproxima-se da forma como realmente será executado o programa em lógica. Por sua vez, o usuário pode optar entre as duas interpretações. A intepretação declarativa segue a proposta da linguagem e portanto permite ao usuário desfrutar do poder implícito na lógica. No entanto, os usuários acostumados às linguagens imperativas normalmente sentem-se mais seguros com a interpretação procedimental.

A distinção entre intepretação declarativa e procedimental será relevante na descrição das diversas análises (complexidade, modos, tipos, medidas, dependências e granulosidade) apresentadas a partir do capítulo 3.

2.2.3 Estratégia de Controle

O projeto de um avaliador para programação em lógica consiste basicamente na determinação de duas políticas, ou seja:

- **Política de escolha de meta (regra de computação)**: Na execução de uma resolvante, como escolher a próxima meta a ser reduzida ?
- **Política de escolha de cláusula (regra de busca)**: Na execução de um procedimento, como escolher a próxima cláusula a ser solucionada ?

O avaliador da linguagem Prolog utiliza as seguintes políticas:

- **Política de escolha de meta**: Executar as metas da esquerda para a direita com retorno no caso de falha (***backtracking***);
- **Política de escolha de cláusula**: Executar as cláusulas de cima para baixo.

Desta forma, o avaliador Prolog pode adotar uma **política de pilhas**. Esta política estabelece que a resolvante deve ser tratada como uma pilha, ou seja, a execução do programa desempilha a meta do topo para redução e empilha as metas derivadas.

A política de redução adotada pelo Prolog possibilita uma interessante visão procedimental do compartilhamento de variáveis pelas metas de uma cláusula. Deste ponto de vista, as metas podem ser classificadas como **produtoras** e **consumidoras**. O procedimento *nrev*, apresentado na figura 2.2, será utilizado para exemplificar esta classificação.

```
nrev([],[]).
nrev([HL],R) :- nrev(L,R1), append(R1,[H],R).
```

FIGURA 2.2 - Procedimento *nrev*

A segunda cláusula do procedimento *nrev* possui duas metas. Estas metas compartilham a variável *R1*. Além disso, a política de redução do Prolog estabelece a execução da primeira meta (*nrev*) antes da segunda (*append*). Desta forma, a meta *nrev* produz um valor para a variável *R1* e a meta *append* consome este valor. Sendo assim, a meta *nrev* atua como produtora e a meta *append* como consumidora. A figura 2.3 mostra a relação de produção e consumo entre estas duas metas.

Conforme afirmam Sterling e Shapiro ([STE86]), a **computação** de uma meta G baseada num programa Prolog P deve gerar todas as soluções de G em relação a P. Em

termos dos conceitos da programação em lógica, a computação Prolog de uma meta G é uma completa travessia de uma particular **árvore de busca** das soluções para G. Seguindo a estratégia de controle do Prolog, esta árvore é sempre percorrida em profundidade e da esquerda para a direita. Esta característica resulta das políticas de redução e não-determinismo adotadas pelo Prolog. A figura 2.4 demonstra uma árvore de busca gerada pela estratégia de controle adotada na linguagem Prolog.

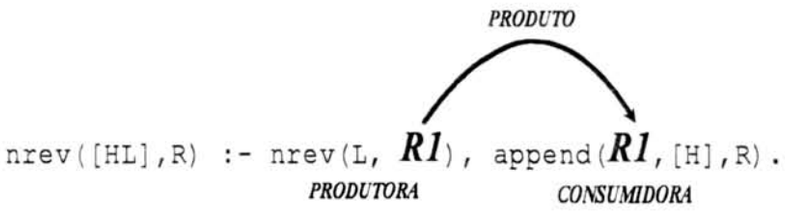


FIGURA 2.3 - Metas produtoras e consumidoras

Geralmente, diferentes estratégias de controle geram diferentes árvores de busca. No entanto, independentemente da estratégia adotada, o conjunto de soluções deve ser o mesmo. Por outro lado, o tamanho da árvore de busca depende da estratégia escolhida. Sendo assim, esta escolha influencia na eficiência da computação ([LIN93]).

As considerações apresentadas nesta seção serão importantes para discussão posterior sobre análise de complexidade e análise de dependências. A escolha de uma estratégia de controle estabelece um conjunto de premissas que devem ser utilizadas durante as análises estáticas (compilação) de programas em lógica. Neste trabalho, adota-se a estratégia de controle da linguagem Prolog.

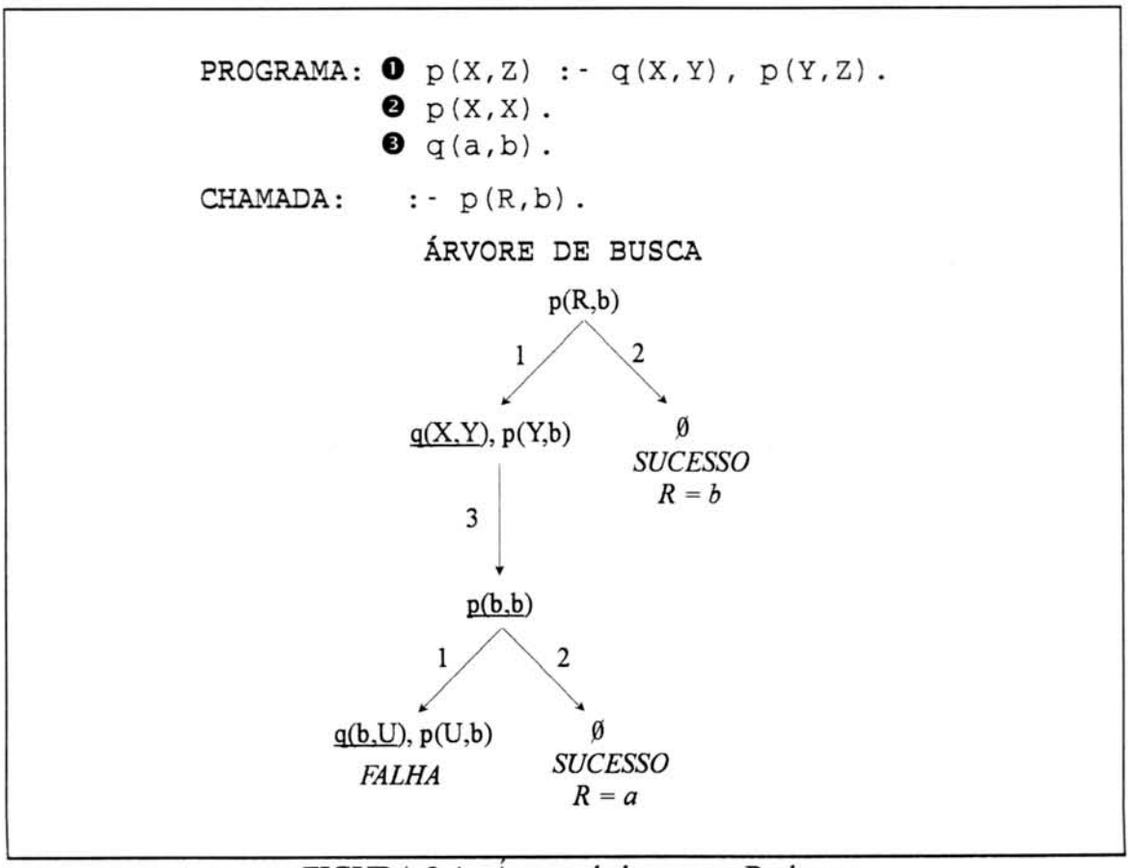


FIGURA 2.4 - Árvore de busca em Prolog

2.2.4 Recursividade

Grande parte do trabalho executado pelos programas em lógica é baseado na **recursividade**. Do ponto de vista procedimental, a recursividade consiste num procedimento chamando a si mesmo. Do ponto de vista declarativo, a recursividade é transparente.

Um exemplo clássico de recursividade em Prolog é o procedimento *append*, mostrado na figura 2.5. A segunda cláusula do procedimento *append* chama-se recursivamente e a primeira cláusula estabelece o término para a recursividade.

```
append([],L,L).
append([HL],L1,[HR]) :- append(L,L1,R).
```

FIGURA 2.5 - Procedimento *append*

A utilização da recursividade confere às linguagens de programação em lógica um alto poder de expressão, pois de forma declarativa o usuário expressa o problema sem preocupar-se com a implementação recursiva feita pelo avaliador.

A recursividade torna-se relevante para este trabalho devido principalmente aos seguintes fatores:

- **Freqüência de utilização**: A recursividade é freqüentemente utilizada na programação em lógica, assumindo desta forma grande importância para compreensão dos mecanismos deste paradigma de programação;
- **Complexidade dos programas**: A recursividade é responsável por grande parte do trabalho executado por um programa em lógica. Desta forma, a complexidade de um programa sofre forte influência das estruturas recursivas utilizadas pelo programador;
- **Análise estática**: A recursividade é um dos principais objetos de estudo da análise estática de programas em lógica. As chamadas recursivas estabelecem grande parte da dinâmica de um programa e introduzem algumas das principais dificuldades para previsão do comportamento da execução.

2.2.5 Variável Lógica e Unificação

Segundo Shapiro ([SHA89]), a principal diferença entre a programação em lógica e a programação convencional consiste na **variável lógica** e sua manipulação através da **unificação**. As principais características da variável lógica são as seguintes:

- **Referência a termos**: Nos programas em lógica uma variável referencia um **termo** e não um local de armazenamento na memória (linguagens convencionais). Os termos são as estruturas de dados básicas da programação em lógica ([STE86]);
- **Atribuições construtivas**: Após especificar um termo, a variável não pode ser alterada para referenciar outro termo. Em outras palavras, a programação em lógica não permite a alteração do conteúdo de uma variável inicializada, conforme ocorre na programação convencional (atribuições destrutivas);

- **Escopo de cláusula**: Uma variável lógica possui seu escopo restrito a cláusula na qual é utilizada, ou seja, não existem variáveis globais;
- **Atipada**: Uma variável lógica não necessita definição, ou seja, não possui um tipo definido. As linguagens convencionais normalmente são tipadas, ou seja, necessitam da definição de tipos para suas variáveis;
- **Atribuição de variáveis**: Uma variável lógica pode referenciar outra variável ou um termo contendo várias variáveis.

Conforme afirmam Sterling e Shapiro ([STE86]), a manipulação de dados na programação em lógica é realizada através da unificação. Do ponto de vista de Shapiro ([SHA86]), compreender a programação em lógica equivale à compreender o poder da unificação. Basicamente, a unificação de dois termos envolve encontrar uma atribuição de valores para suas variáveis que torne os dois símbolos idênticos. Desta forma, a unificação pode ser definida como um simples e poderoso processo de casamento de padrões.

Em [SHA86] encontra-se uma comparação entre o processo de unificação e primitivas de manipulação de dados nas linguagens convencionais. Nesta comparação, Shapiro ressalta a equivalência entre a unificação de uma meta com a cabeça de uma cláusula e a passagem de parâmetros nas linguagens convencionais. Esta visão é compatível com a semântica procedimental da programação em lógica, onde a solução de uma meta equivale a chamada de um procedimento. Neste caso, a unificação implementa a passagem de parâmetros, onde os argumentos da meta equivalem aos parâmetros atuais e os argumentos da cabeça da cláusula equivalem aos parâmetros formais. A figura 2.6 mostra, através de um diagrama de módulos ([JON88]), a visão procedimental da unificação numa chamada para o procedimento *append* (figura 2.5).

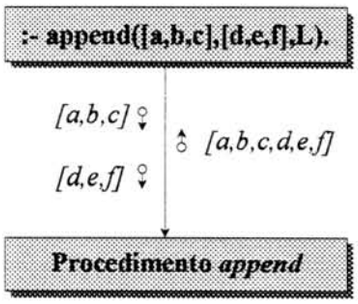


FIGURA 2.6 - Visão procedimental da unificação - Passagem de parâmetros

Outra característica interessante da visão procedimental da unificação é a direcionalidade dos argumentos. Conforme afirmações encontradas em [DEB89] e [LIN93], na programação em lógica os argumentos de um procedimento não são direcionados, ou seja, não existe nos programas em lógica o conceito de argumento de entrada e argumento de saída. Este fato deve-se à natureza bidirecional da variável lógica e à passagem de parâmetros através da unificação. No entanto, segundo Debray, freqüentemente os procedimentos podem ser analisados do ponto de vista direcional.

Na figura 2.6, pode-se verificar a interpretação direcional dos argumentos para uma chamada ao procedimento *append*. Neste exemplo, a chamada possui dois argumentos fechados e um argumento contendo uma variável livre. Os dois argumentos fechados atuam como entradas e o argumento livre atua como saída. Na figura 2.6, a

chamada do procedimento *append* possui uma determinada configuração para a direcionalidade dos argumentos, ou seja, os dois primeiros são argumentos de entrada e o último é argumento de saída. Torna-se claro pelo exemplo, que a direção dos argumentos pode variar de acordo com a forma da chamada. No entanto, afirma Lin ([LIN93]) que na prática um procedimento possui poucas configurações de direcionalidade.

As características da variável lógica e da unificação são importantes para compreensão das análises apresentadas a partir do capítulo 3. Dentre estas características destacam-se neste trabalho o escopo de cláusula da variável lógica e a analogia unificação/passagem de parâmetros.

2.2.6 Instanciação

Durante uma unificação as variáveis existentes nos termos a serem unificados podem sofrer atribuições e passar a referenciar algum termo. Este processo de atribuição recebe o nome de **instanciação**. Por exemplo, a unificação dos termos *data(D,M,ano(N))* e *data(X,fevereiro,A)* resulta nas seguintes instanciações:

❶ *D* é instanciado com *X*. Igualmente, *X* é instanciado com *D*;

❷ *M* é instanciado com *fevereiro*;

❸ *A* é instanciado com *ano(N)*.

Desta forma, a unificação resulta em três instanciações. A primeira instanciação faz com que a variável *D* referencie a variável *X* e vice-versa. A segunda instanciação resulta na variável *M* referenciando o **átomo** *fevereiro*. Na terceira instanciação, a variável *A* passa a referenciar a **estrutura** *ano(A)*. As definições de átomo e estrutura podem ser encontradas em [STE86]. Se uma variável não está instanciada com um átomo ou com uma estrutura, recebe o nome de **variável livre**. Portanto, uma variável livre pode estar instanciada com outra variável livre. Qualquer alteração no conteúdo de uma delas reflete na outra. No exemplo, as variáveis D, X e N estão livres.

Basicamente, existem dois estados para um termo, ou seja:

- **Aberto**: Um termo está aberto quando possui variáveis livres, ou seja, ainda pode receber instanciações;
- **Fechado**: Um termo está fechado quando não possui variáveis livres e conseqüentemente não aceita mais instanciações.

Por exemplo, o termo *ano(A)* está aberto. No entanto, numa unificação com *ano(1995)* resultará fechado, pois a variável *A* será instanciada com o átomo *1995* e o termo não poderá receber outras instanciações (não possui mais variáveis livres).

Uma variável livre é um termo aberto por excelência. Desta forma, pode-se compreender as instanciações como um progressivo fechamento dos termos. Durante a execução de um programa, um termo torna-se cada vez mais fechado. A variável livre é o termo mais aberto da programação em lógica.

O conceitos apresentados nesta subseção serão utilizados no restante no texto. Deve-se dedicar atenção especial para a noção de termo aberto e fechado, a qual será utilizada em grande parte deste trabalho.

2.2.7 Não-Determinismo

No ponto de vista dos autores de [STE86], o **não-determinismo** é um poderoso conceito teórico utilizado para definir de forma concisa um modelo computacional abstrato. Em complemento, o não-determinismo pode ser utilizado para definição e implementação de algoritmos. Afirmam ainda Sterling e Shapiro, que uma máquina não-determinista é aquela que pode escolher corretamente sua próxima operação quando existirem várias alternativas. Finalmente, esses autores destacam que verdadeiras máquinas não-deterministas não podem ser criadas, mas podem ser simuladas.

A seção 2.2.3 destaca a existência de duas ocasiões, durante a execução de um programa em lógica, em que o avaliador deve escolher uma entre várias alternativas. Na solução de uma resolvante, o avaliador deve escolher a próxima meta a ser reduzida (regra de computação). Na execução de um procedimento, o avaliador deve escolher a próxima cláusula a ser solucionada (regra de busca). Estas duas possibilidades de escolha geram dois tipos de não-determinismo, ou seja, **não-determinismo de meta (não-determinismo E)** e **não-determinismo de cláusula (não-determinismo OU).** O não-determinismo existente na escolha da cláusula pode ser subdividido em dois subtipos, ou seja, **não-determinismo *don't care*** e **não-determinismo *don't know*.**

No não-determinismo *don't care* apenas uma solução é gerada por meta. Desta forma, mesmo que uma meta possua várias soluções, apenas uma será utilizada durante a execução de um programa. Portanto, a falha de uma meta ocasiona a falha do programa inteiro. Neste caso, normalmente a regra de busca baseia-se numa escolha aleatória de cláusulas.

No não-determinismo *don't know* são geradas todas as soluções de uma meta. Desta forma, uma chamada poderá ser realizada várias vezes, dependendo do número de soluções geradas pelas metas anteriores. Portanto, a falha de uma meta poderá ocasionar uma nova tentativa até que não hajam mais possibilidades de execução. Normalmente, a regra de busca baseia-se na execução seqüencial de todas as cláusulas. A linguagem Prolog é baseada no não-determinismo *don't know* e utiliza o *backtracking* para implementar a geração de todas as soluções de uma meta.

A análise estática dos programas em lógica deve considerar suas características não-deterministas. Desta forma, os conceitos apresentados nesta subseção serão utilizados várias vezes no decorrer do texto. Destaca-se a importância do não-determinismo no estudo da complexidade na programação em lógica (capítulo 6).

2.3 Paralelismo na Programação em Lógica

Esta seção descreve a relação entre paralelismo e programação em lógica. Inicialmente é apresentada uma introdução às fontes de paralelismo. Logo após, aborda-se mais detalhadamente o paralelismo OU e o paralelismo E. Finalmente, é descrito o estado da arte da análise de granulosidade nos programas em lógica.

2.3.1 Fontes de Paralelismo

Segundo Hermenegildo ([HER86]), o relacionamento entre programação em lógica e paralelismo está baseado na liberdade (não-determinismo) que o avaliador possui na escolha dos caminhos de execução. Esta liberdade permite o desenvolvimento de

avaliadores que explorem em paralelo os diversos caminhos oriundos do não-determinismo. Desta forma, os dois principais tipos de não-determinismo originam as duas principais fontes de paralelismo, ou seja:

- **Paralelismo OU**: execução paralela das cláusulas de um predicado. Este tipo de paralelismo está baseado no não-determinismo OU, ou seja, na liberdade de escolha das cláusulas no predicado;
- **Paralelismo E**: execução em paralelo das metas do corpo de uma cláusula. Este paralelismo baseia-se no não-determinismo E, ou seja, na liberdade de escolha das metas na resolvante.

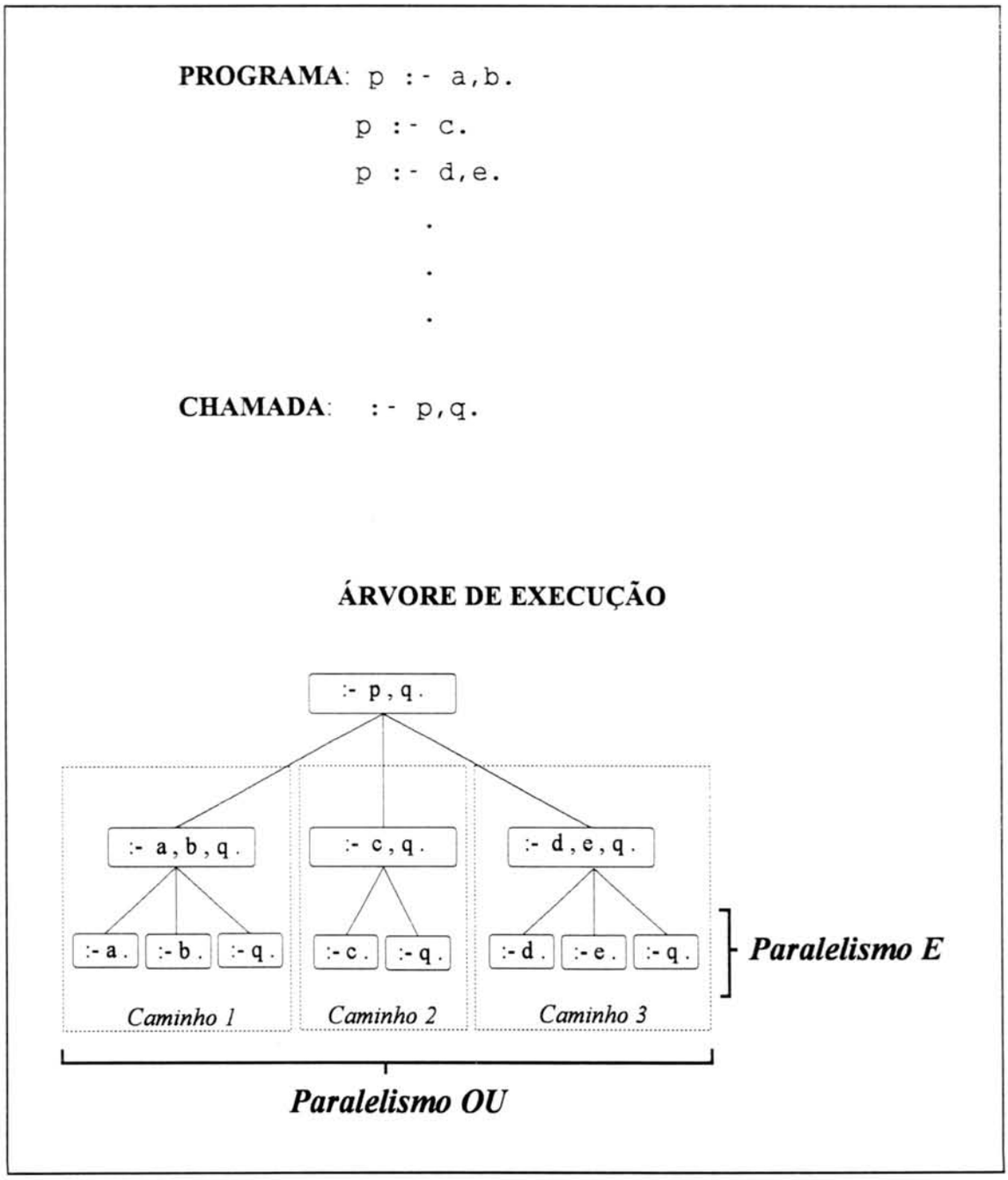


FIGURA 2.7 - Fontes de paralelismo na programação em lógica

A figura 2.7 exemplifica a exploração do paralelismo na programação em lógica. Para maior simplicidade são apresentados apenas os nomes de predicados. Conforme mostrado na figura 2.7, o paralelismo OU é explorado na aplicação da regra de busca e o paralelismo E na aplicação da regra de computação. Desta forma, o paralelismo OU explora a busca paralela de diferentes caminhos para o mesmo objetivo e o paralelismo E executa em paralelo passos de um caminho.

Durante a execução de um programa, normalmente surgem várias oportunidades para exploração do paralelismo OU e do paralelismo E. O avaliador gerencia estas oportunidades de acordo com sua política de paralelismo. Alguns sistemas exploram apenas um tipo de paralelismo e outros implementam os dois tipos (**paralelismo E/OU**).

Deve-se ressaltar ainda, que existem outras fontes de paralelismo na programação em lógica. Estas fontes não estão relacionadas com o não-determinismo e exploram o paralelismo em baixo nível. Destacam-se nesta categoria:

- **Paralelismo de unificação** ([VLA92]): A unificação entre uma meta e a cabeça de uma cláusula normalmente exige que diversos pares de termos (argumentos) sejam unificados. Estas unificações podem ser executadas em paralelo;
- **Paralelismo de fluxo** ([SHA89]): A utilização de vários processos pode acelerar o tratamento de estruturas de dados complexas. Um processo pode atuar como produtor das estruturas e outros processos podem atuar como consumidores (esquema produtor-consumidor). O processo produtor continua sua execução logo após enviar uma estrutura de dados para um processo consumidor. Por sua vez, o processo consumidor continua sua execução logo após receber a estrutura. Desta forma, explora-se o potencial de paralelismo existente no esquema produtor-consumidor. Conforme afirmam Gupta e Jayaraman ([GUP93]), o paralelismo de fluxo pode ser utilizado para explorar o paralelismo E dependente (discutido na subseção 2.3.3);
- **Paralelismo de busca** ([KAS83]): Neste tipo de paralelismo o programa é dividido em conjuntos disjuntos de cláusulas, os quais são avaliados por diferentes processos durante a busca de soluções.

2.3.2 Paralelismo OU

O paralelismo OU destaca-se como a primeira fonte de paralelismo pesquisada na programação em lógica. Este fato deve-se em grande parte à simplicidade para sua exploração quando comparado com o paralelismo E. A seguir são apresentadas algumas características que estimulam a exploração do paralelismo OU:

- **Generalidade**: a exploração do paralelismo OU não implica na restrição do poder da linguagem de programação em lógica, ou seja, o poder de expressão da lógica não é comprometido pela exploração do paralelismo;
- **Simplicidade**: o paralelismo OU pode ser explorado de forma implícita sem tornar a compilação muito complexa. Sendo assim, a exploração automática do paralelismo é estimulada;
- **Similaridade com os avaliadores seqüenciais**: pode-se explorar o paralelismo OU utilizando um modelo de avaliador semelhante às propostas de execução seqüencial. Esta característica resulta em duas vantagens significativas. Em

primeiro lugar, os avaliadores paralelos do sistema OU podem incluir a tecnologia disponível para aumento de desempenho na execução seqüencial de programas em lógica. Sendo assim, obtem-se a melhor velocidade absoluta por processador. Além disso, os avanços da tecnologia seqüencial poderão ser incorporados nos avaliadores do sistema OU sem alterações no modelo paralelo. Em segundo lugar, a similaridade entre a exploração do paralelismo OU e a execução seqüencial auxilia na preservação da semântica da programação em lógica. Sendo assim, mais uma vez é estimulada a exploração automática do paralelismo.

- **Granulosidade**: devido à paralelização de caminhos completos na busca de soluções (ramos da árvore de busca), normalmente o paralelismo OU implica uma elevada granulosidade, ou seja, uma elevada complexidade para execução dos caminhos (grãos). Quanto mais próximo da raíz da árvore de busca for explorado o paralelismo OU, maior será a complexidade dos grãos. A elevada granulosidade torna o paralelismo OU atraente, pois grãos grandes geram pouca comunicação entre processadores, facilitando assim a exploração eficiente do paralelismo;
- **Aplicações**: a exploração do paralelismo na regra de busca faz com que o paralelismo OU seja especialmente interessante em aplicações com alto nível de não-determinismo (não determinismo de cláusula / não-determinismo OU). Dentre estas aplicações destacam-se as relacionadas com inteligência artificial. A relação entre paralelismo OU e processo de busca é notória. Desta forma, o surgimento de oportunidades para exploração desta fonte de paralelismo está diretamente relacionado com a natureza do problema. As aplicações que envolvam muita busca geram muita oportunidade de paralelismo OU. São exemplo destas aplicações: verificação de regras em sistemas especialitas, prova de teoremas, interpretação de linguagem natural e consulta em base de dados.

Devido à independência das regras de busca e computação, o paralelismo OU pode ser combinado tanto com uma regra de computação seqüencial (**sistemas OU**) quanto com um regra de computação que explore o paralelismo E (**sistemas E/OU**).

Deve-se salientar ainda, dois problemas relacionados com a exploração do paralelismo OU, ou seja, a **explosão de processos** e a **gerência do ambiente de ligação de variáveis**.

- **Explosão de processos**: Conforme afirma Hermenegildo ([HER86]), a exploração paralela de **todos** os caminhos gerados pela regra de busca pode ocasionar ineficiência na execução. Este fato ocorre, quando acontece um rápido crescimento da árvore de busca e conseqüentemente do número de processos OU. Os programas com alto nível de não-determinismo OU, tais como a maioria das aplicações em inteligência artificial, podem criar um grande número de processos com baixa granulosidade. Considerando-se que os sistemas paralelos reais possuem um número limitado de processadores, uma explosão de processos cria problemas na gerência do paralelismo (alto consumo de memória e elevado custo de gerenciamento). Esta situação é agravada, quando apenas uma solução é necessária, fazendo com que grande parte da computação torne-se inútil. Dentre as soluções propostas para este problema destacam-se o uso de anotações para restringir a geração de caminhos paralelos

e o uso de heuristicas para eliminar, o mais rápido possível, o máximo de caminhos que não levam a uma solução;

- **Gerência do ambiente de ligação de variáveis**: Quando um avaliador seqüencial realiza a busca de soluções em determinado caminho da árvore de busca, ocorrem diversas instanciações de variáveis. Se o avaliador resolver percorrer outro caminho para obter outras soluções para a mesma pergunta, basta desfazer as instanciações realizadas pela primeira execução. No entanto, quando o paralelismo OU é explorado, vários caminhos são percorridos simultaneamente e portanto a mesma variável pode sofrer diferentes instanciações. Surge assim, a necessidade de gerência do ambiente de ligação de variáveis. Este problema inclui questões relacionadas com o armazenamento e tratamento das diversas instanciações da mesma variável.

2.3.3 Paralelismo E

A exploração do paralelismo E é mais complexa do que a exploração do paralelismo OU. Esta complexidade deve-se em grande parte ao **escopo léxico** das variáveis na programação em lógica. Uma variável na programação em lógica possui um escopo léxico de uma cláusula, ou seja, as variáveis estão confinadas em cláusulas e existem apenas localmente. Desta forma, a execução paralela de cláusulas (paralelismo OU) não envolve conflitos e dependências de variáveis. Por outro lado, o paralelismo E explora a execução paralela de objetivos da mesma cláusula, os quais podem compartilhar variáveis e consequentemente ocasionar conflitos e necessidade de gerência das dependências. A seguir são apresentadas algumas comparações do paralelismo OU e paralelismo E:

- quase todos os programas em lógica possuem paralelismo E. Este fato não ocorre com o paralelismo OU, o qual surge com menos freqüência;
- ao contrário do paralelismo OU, o paralelismo E pode ser vantajoso para programas fortemente determinísticos;
- todo o trabalho realizado pelo paralelismo E é útil, pois sempre é necessário explorar todos os objetivos de uma cláusula para obter sua solução. Conforme discutido na subseção 2.3.2, o paralelismo OU pode realizar trabalho inútil.

O principal problema para exploração do paralelismo E é o **conflito de ligações de variáveis**. A execução paralela de dois objetivos do corpo de uma mesma cláusula pode ocasionar problemas, quando esses objetivos compartilham uma ou mais variáveis. Cada uma das execuções independentes pode instanciar a variável compartilhada com valores distintos, o que é inadmissível na programação em lógica. Este conflito de ligação de variáveis deve ser solucionado, existindo várias propostas para implementação do paralelismo E.

Basicamente, as propostas para implementação do paralelismo E dedicam-se à solução do conflito de ligação de variáveis. Estas propostas podem ser classificadas em dois grupos:

- **paralelismo E independente**: Neste grupo enquadram-se as propostas que exploram o paralelismo somente entre literais independentes. Essa independência ocorre quando os literais não compartilham variáveis livres (paralelismo restrito) ou quando apesar dos literais compartilharem variáveis

livres, fica determinada através de análise estática a impossibilidade de conflito de ligação (paralelismo não restrito). A proposta apresentada em [DEG84] é um exemplo clássico de paralelismo E independente restrito. Em [HER89] encontra-se uma abordagem de ambos os tipos de paralelismo E independente;

- **paralelismo E dependente**: Neste grupo estão as propostas que exploram o paralelismo entre quaisquer literais, mesmo aqueles dependentes, ou seja, aqueles que compartilham variáveis e possibilitam conflitos de ligação. Desta forma, o tratamento do conflito de ligações deve ser realizado durante a execução paralela, através de algum mecanismo de gerência do paralelismo. Destaca-se como proposta deste tipo de paralelismo o sistema Parlog ([CLA86], [GRE87]). Mais detalhes sobre o paralelismo E dependente podem ser encontrados em [SHA89].

As propostas para implementação do paralelismo E independente enfocam basicamente a detecção da independência entre literais. Desta forma, pode-se classificar estas propostas em três grupos, de acordo com a forma de detecção:

- **detecção na compilação (análise estática)**: Nesta proposta as dependências entre literais são estabelecidas completamente durante a compilação. Sendo assim, não será introduzido custo computacional (*overhead*) na execução. No entanto, a análise de dependências é baseada no pior caso, pois não é conhecido tudo a respeito das possíveis ligações das variáveis. Esta abordagem conservativa pode ocasionar perda de paralelismo. Destacam-se como propostas de análise estática os textos [CHA85] e [CHA85a].
- **detecção na execução (análise dinâmica)**: Esta abordagem propõe a determinação da independência entre literais durante a execução. Desta forma, a detecção das dependências atinge alta precisão, pois são conhecidas todas as informações sobre as ligações das variáveis. No entanto, o custo introduzido na execução por esta abordagem, reduz significativamente a eficiência da exploração do paralelismo. O modelo E/OU de Conery ([CON85]) realiza a detecção da independência durante a execução.
- **detecção combinada**: Nesta proposta, a detecção da independência entre literais é realizada de forma combinada, ou seja, parte na compilação e parte na execução. Durante a compilação é realizado grande parte do trabalho, postergando-se para a execução a detecção definitiva. Desta forma, tenta-se equilibrar os benefícios das abordagens estática e dinâmica. Durante a compilação, obtem-se o máximo de informações sobre o programa, reduzindo o custo introduzido na execução. Durante a execução, quando todas as informações sobre as ligações estão disponíveis, detecta-se as independências com precisão. Como exemplo clássico desta abordagem, pode-se citar o paralelismo E restrito proposto por Degroot ([DEG84], [DEG87], [DEG89]). Além deste, destaca-se o modelo &-prolog de Hermenegildo ([HER90], [HER91], [BUE93]).

2.3.4 Análise de Granulosidade

Conforme ressaltado na subseção 2.1.2, as técnicas para realização da análise de granulosidade variam de acordo com o paradigma de programação. Na programação em lógica, os estudos para análise de granulosidade são recentes ([TIC88], [DEB90],

[KIN90], [TIC90], [KIN92], [ZHO92], [DEB93], [TIC93], [GAR94]) e possuem como base os trabalhos desenvolvidos nos paradigmas convencional e funcional (citados na subseção 2.1.2). Nota-se nos últimos anos, que as pesquisas sobre análise de granulosidade na programação em lógica tem alcançado resultados estimulantes, motivando assim, o aprofundamento dos estudos e o aprimoramento dos modelos propostos.

Deve-se ressaltar ainda, que não existe um consenso entre os pesquisadores, quanto à definição de análise de granulosidade no âmbito da programação em lógica. A definição utilizada neste trabalho foi apresentada nas subseções 2.2.1 e 2.2.2. Esta definição é ampla o bastante para ser aplicada em qualquer paradigma de programação. No escopo dessa dissertação, análise de granulosidade consiste na análise do tamanho dos grãos, ou seja, a determinação da complexidade adequada para os módulos que deverão ser executados seqüencialmente num único processador. No entanto, alguns pesquisadores definem análise de granulosidade de forma diferente. Por exemplo, Nai-Wei Lin ([LIN95]) afirma que análise de complexidade e análise de granulosidade são sinônimos. Por sua vez, Saumya Debray pensa que o termo análise de granulosidade é confuso e muito abrangente ([DEB95]). Debray afirma ainda, que no seu ponto de vista, análise de granulosidade significa intuitivamente "análise para controle da granulosidade". Por outro lado, Evan Tick concorda com a definição apresentada neste texto ([TIC95]) e complementa dizendo que análise de granulosidade não resolve o problema de escalonamento. O autor desta dissertação não concorda com a definição de Lin e acredita que análise de complexidade e análise de granulosidade, apesar de compartilharem diversos estudos, são tópicos de pesquisa diferentes. Além disso, o autor deste texto concorda com Debray quanto à possível confusão gerada pela abrangência do termo análise de granulosidade, mas acredita que uma clara definição no escopo de um trabalho resolve esse problema. O termo análise de granulosidade é utilizado, inclusive no título, em textos clássicos na pesquisa do paralelismo na programação em lógica, tais como [TIC88], [DEB90], [TIC90], [ZHO92], [TIC93].

Destaca-se como a primeira proposta para análise de granulosidade nos programas em lógica o trabalho de Tick ([TIC88], [TIC90]). Neste estudo, é proposto um algoritmo simples para estimar a granulosidade relativa dos objetivos de uma cláusula. As informações fornecidas por este algoritmo podem ser utilizadas para auxiliar nas decisões de escalonamento durante a execução paralela de programas em lógica. A análise proposta por Tick é estática e conseqüentemente fornece informações com baixa precisão. A principal fonte de imprecisão desta proposta reside no tratamento das chamadas recursivas. O modelo despreza esta importante fonte de complexidade dos programas em lógica. Este fato deve-se à natureza da análise estática, a qual não permite previsões do número de chamadas recursivas que serão realizadas durante a execução. O modelo de Tick foi implementado num trabalho de diplomação na UFRGS ([SCH93]) e está disponível para futuros estudos.

Em [KIN90] é apresentada uma proposta para análise de granulosidade em linguagens lógicas concorrentes ([SHA89]). Neste método, o particionamento dos programas é realizado completamente em tempo de compilação (análise estática), através da análise de complexidade dos objetivos. Em trabalho posterior, King e Soper aplicaram informações sobre dependência de dados para particionar os literais das cláusulas em *threads*, de forma que a execução possa ser totalmente ordenada em tempo de compilação ([KIN92]). Este particionamento estático de literais pode reduzir o custo

computacional associado com o escalonamento e sincronização de processos durante a execução.

No ano de 1990 surgiu uma abordagem que disponibilizou informações precisas de granulosidade através de complexas análises estáticas combinadas com ações durante a execução. A proposta de Debray, Hermenegildo e Lin ([DEB90]) realiza uma sofisticada análise do programa fonte e resulta em expressões que representam a complexidade absoluta dos objetivos (em segundos, por exemplo) em função de seus argumentos de entrada. As expressões de complexidade podem ser resolvidas durante a execução e as informações obtidas podem ser utilizadas para auxiliar no escalonamento de tarefas. Basicamente, a proposta consiste em realizar a maior parte do trabalho em tempo de compilação, deixando para a execução apenas o estritamente necessário para obtenção de precisão. Desta forma, este modelo visa o equilíbrio entre a análise estática (baixo custo adicional durante a execução) e a análise dinâmica (alta precisão nas informações de granulosidade). O modelo proposto em [DEB90] vem sendo pesquisado e aprimorado nos últimos anos ([DEB93], [LIN93], [DEB94a], [GAR94]). Em [TIC93] são apresentadas críticas a abordagem proposta em [DEB90], acompanhadas de uma comparação entre o modelo descrito em [DEB90] e o modelo proposto em [ZHO92] e [TIC93].

Em [ZHO92] e [TIC93] é apresentada uma evolução do modelo descrito em [TIC90]. O novo modelo propõe uma análise combinada como a descrita em [DEB90]. No entanto, nesta nova abordagem é realizada uma análise estática simplificada, a qual obtem expressões aproximadas da complexidade dos objetivos de uma cláusula. Estas expressões representam a complexidade relativa de um objetivo em relação à complexidade da cláusula, ou seja, não obtem-se a complexidade absoluta do objetivo (em segundos, por exemplo) mas sim sua influência na complexidade da cláusula (porcentagem da complexidade da cláusula). Assim como o modelo apresentado em [DEB90], a proposta descrita em [ZHO92] e [TIC93] visa o equilíbrio das análises estática e dinâmica. O principal argumento apresentado em [TIC93] para justificar o uso da complexidade relativa, consiste na afirmação de que durante a execução paralela o escalonador não necessita de informações precisas sobre a complexidade dos grãos. Torna-se importante ressaltar, que conforme afirma Evan Tick ([TIC94]), os resultados utilizados na análise de desempenho apresentada em [TIC93], foram obtidos através de simulações, ou seja, não existe um protótipo implementado para o modelo.

Deve-se destacar ainda uma afirmação encontrada em [TIC93]. Neste artigo, Zhong e Tick afirmam que na análise de granulosidade existe claramente uma negociação entre precisão e custo introduzido na execução. Afirmam ainda, que as propostas [DEB90] e [TIC90] representam os dois extremos no espectro da análise de granulosidade. Concluem argumentando que a proposta descrita em [ZHO92] e [TIC93] representa uma opção intermediária entre [TIC90] e [DEB90], obtendo-se precisão suficiente e diminuindo-se o custo introduzido na execução. No entanto, em [LIN93] encontra-se a afirmação de que o método defendido em [ZHO92] e [TIC93] não é preciso o bastante para geração de código paralelo no formato específico utilizado nas pesquisas para exploração de paralelismo desenvolvidas por Lin, Debray e Hermenegildo. Não existe um consenso da comunidade científica sobre qual das propostas é mais promissora e qual delas obtem resultados mais eficientes. Nos testes de desempenho apresentados em [LIN93] e [TIC93], as duas propostas alcançam bons resultados. Desta forma, não pode-se afirmar com certeza qual delas assumirá maior importância nas pesquisas futuras. No entanto, considerando-se as recentes pesquisas

sobre análise de granulosidade na programação em lógica, pode-se afirmar com convicção que a análise combinada é uma forte tendência nesta linha de estudo.

Na opinião do autor desta dissertação a proposta de Debray, Lin e Hermenegildo destaca-se das demais abordagens, assumindo maior importância para as pesquisas relacionadas com a programação em lógica. Esta convicção resulta basicamente das seguintes considerações sobre o modelo descrito em [DEB90] e [LIN93]:

- a alta precisão no cálculo da complexidade dos procedimentos e os sofisticados métodos utilizados durante a análise estática permitem a **aplicação do modelo em diversas áreas de pesquisa relacionadas com a programação em lógica**, tais como: exploração do paralelismo, simulação na execução de programas, otimização de código objeto, depuração de programas e *backtracking* inteligente;
- grande parte do **modelo está baseado na análise estática**, principalmente na interpretação abstrata de programas em lógica. Desta forma, o modelo investe nesta linha de pesquisa, evidenciando o ponto de vista dos seus criadores. Manuel Hermenegildo afirma que o futuro da exploração do paralelismo na programação em lógica reside no aprimoramento da análise estática de programas ([HER93]). Além disso, analisando-se a orientação dos trabalhos de pesquisa de Saumya Debray, constata-se sua continua dedicação à interpretação abstrata ([DEB86], [DEB89], [GIA92] e [DEB94]). O autor dessa dissertação concorda com os criadores do modelo apresentado em [DEB90] e acredita que o futuro das pesquisas sobre paralelismo na programação em lógica encontra-se no aprimoramento das metodologias para análise de programas em tempo de compilação, dentre as quais destacam-se a interpretação abstrata e a análise de complexidade;
- além das informações de complexidade utilizadas para realizar a análise de granulosidade, o **modelo disponibiliza diversas informações sobre os procedimentos do programa**, tais como, número de soluções e relações de tamanho entre entradas e saídas. Estas informações podem ser utilizadas em diversas aplicações;
- o **modelo possui uma forte base teórica**, ou seja, o desenvolvimento da proposta está baseado em diversos algoritmos, teoremas, proposições e provas. Além disso, o modelo realiza várias abordagens matemáticas, dedicando grande parte da proposta ao tratamento de equações diferenciais e solução de sistemas de equações;
- a **construção de um protótipo denominado CASLOG** (Complexity Analyses System for LOGic) valida a proposta e aumenta sua praticidade. Este protótipo está disponível à comunidade científica e pode ser utilizado em diversas pesquisas sobre programação em lógica (esta dissertação utiliza o protótipo CASLOG durante a análise de complexidade - capítulo 6);
- o **modelo deverá cada vez mais assumir uma posição de destaque junto a comunidade científica.** Esta convicção possui como base as seguintes observações: um dos criadores do modelo (Manuel Hermenegildo) é responsável na atualidade por um dos projetos mais promissores na exploração do paralelismo na programação em lógica (projeto &-Prolog [HER90], [HER91], [BUE93]), um dos criadores do CASLOG (Nai-Wei Lin) afirma que

continuará investindo no desenvolvimento deste trabalho ([LIN94]) e os três criadores do modelo (Debray, Lin e Hermenegildo) publicaram recentemente trabalhos relacionados com análise de complexidade e granulosidade na programação em lógica ([DEB93], [DEB94a] e [GAR94]).

2.4 Conclusões

O capítulo 2 apresentou diversos conceitos relacionados com paralelismo, programação em lógica e exploração do paralelismo na programação em lógica. Estes conceitos servirão de base para o restante do texto, facilitando ao leitor a compreensão dos próximos capítulos. Na seção 2.1, destaca-se a observação de que a análise de granulosidade encontra-se entre os principais problemas a serem solucionados para exploração eficiente do paralelismo (subseção 2.1.2). Destacam-se ainda e assumem grande importância para o restante do texto, os conceitos de grão, granulosidade e análise de granulosidade (subseção 2.1.2). Na subseção 2.1.3, assume importância especial, a opinião do autor desta dissertação quanto à tendência à aceitação cada vez maior do paralelismo implícito e a relevância da exploração do paralelismo nos paradigmas não convencionais de programação. Por sua vez, a subseção 2.1.4, permitiu o aprimoramento da discussão sobre paralelismo e paradigmas de programação. Em seguida, através das seções 2.3 e 2.4, foram apresentados importantes conceitos sobre programação em lógica e exploração de paralelismo neste paradigma. Destacando-se das demais, a subseção 2.3.4 discutiu a análise de granulosidade na programação em lógica.

Algumas conclusões deste capítulo são as seguintes:

- a análise de granulosidade é um problema básico para exploração do paralelismo;
- existe uma tendência à aceitação cada vez maior do paralelismo implícito e conseqüentemente da exploração automática do paralelismo;
- os paradigmas não convencionais de programação facilitam a exploração automática do paralelismo;
- na opinião do autor desta dissertação o trabalho sobre análise de granulosidade desenvolvido por Debray, Lin e Hermenegildo deverá assumir cada vez mais, nos próximos anos, uma posição de destaque;
- a análise combinada é uma forte tendência para realização da análise de granulosidade na programação em lógica.

O próximo capítulo inicia a apresentação do modelo GRANLOG. Vários conceitos discutidos no capítulo 2 serão utilizados e aperfeiçoados no decorrer do texto. O modelo GRANLOG permite a aplicação dos diversos conceitos sobre programação em lógica e paralelismo discutidos no decorrer deste capítulo.

3 Modelo GRANLOG

Este capítulo introduz o principal tópico da dissertação, ou seja, o modelo **GRANLOG** (**GR**anularity **AN**alyzer for **LOG**ic Programming). O GRANLOG é a espinha dorsal dessa pesquisa, servindo de suporte para organização das informações pesquisadas e concretização do aspecto criativo deste trabalho. A seção 3.1 aborda os princípios básicos que orientam e motivam o desenvolvimento do GRANLOG. A seção 3.2 apresenta uma visão geral da proposta. A seção 3.3 descreve a organização básica do modelo. Na seção 3.4, o GRANLOG é comparado com outros modelos. A seção 3.5 propõe duas aplicações para o GRANLOG, ou seja, auxílio a decisões de escalonamento e simulação da execução de programas. Finalmente, a seção 3.6 apresenta as conclusões do capítulo.

O principal objetivo deste capítulo é apresentar uma visão geral da proposta, antes da abordagem específica de cada uma das três etapas que compõem a análise de granulosidade. Estas etapas são discutidas em detalhes nos próximos três capítulos. A cada etapa equivale um módulo do modelo. O modelo GRANLOG foi apresentado à comunidade científica através das publicações [BAR94a], [BAR95] e [BAR95a].

3.1 Princípios Básicos

A grande variedade de sistemas paralelos (arquitetura paralela e plataforma para execução) atualmente disponíveis faz com que o desenvolvimento, a portabilidade e a manutenção de programas paralelos tornem-se tarefas difíceis e onerosas. Dentre as diversas características que variam entre sistemas paralelos destacam-se: tipo de memória (compartilhada ou distribuída), número de processadores, tipo de processadores (homogêneos ou heterogêneos), tipo de escalonamento (centralizado ou distribuído), velocidade dos canais de comunicação e existência de processadores de entrada/saída. Estas características influenciam diretamente na forma de exploração do paralelismo. Sendo assim, o **primeiro princípio** básico da proposta está relacionado com os sistemas paralelos.

1° **Princípio**: **Independência do sistema paralelo.** As informações de granulosidade, obtidas através da análise realizada pelo GRANLOG, são completamente independentes do sistema paralelo onde será executado o programa. Desta forma, obtem-se flexibilidade e portabilidade para as informações geradas pela análise.

Com relação ao próximo princípio torna-se importante ressaltar que na programação em lógica a exploração automática do paralelismo é estimulada, devido ao paralelismo implícito existente na avaliação das expressões lógicas e a clara distinção entre a semântica e o controle da linguagem. Estas características possibilitam a programação de computadores paralelos em nível de dificuldade semelhante à da programação seqüencial e ainda facilitam a migração de programas entre máquinas seqüenciais e paralelas. Desta forma, surge o **segundo princípio** do GRANLOG.

2° **Princípio**: **Detecção automática do paralelismo (paralelismo implícito).** O modelo propõe a exploração automática do paralelismo nos programas em lógica. Desta forma, o programador não participa da paralelização dos programas e a linguagem de programação mantém-se inalterada. Este princípio permite o aproveitamento de

programas em lógica já existentes, além de liberar o programador do encargo de gerenciar explicitamente o paralelismo do problema.

O **terceiro princípio** está baseado na constatação de que as principais fontes de paralelismo na programação em lógica são o paralelismo OU e o paralelismo E. Segundo Kalé ([KAL87]), qualquer modelo que pretenda explorar ao máximo o paralelismo existente nos programas em lógica deverá analisar estas duas fontes. Portanto, o último princípio do modelo proposto assume uma importância significativa.

3° Princípio: **Máxima detecção do paralelismo existente nos programas em lógica.** A análise realizada pelo GRANLOG gera informações de granulosidade relacionadas com o paralelismo OU e com o paralelismo E, visando desta forma, a máxima exploração do paralelismo existente nos programas em lógica.

3.2 Visão Geral

O GRANLOG é um modelo computacional que propõe uma apurada análise estática de programas em lógica (análise de granulosidade). Basicamente, o modelo determina os grãos existentes num programa e possibilita a obtenção de informações relacionadas com estes grãos, tais como complexidade do grão (granulosidade) e custo para transmissão de suas entradas e saídas. Desta forma, o principal objetivo do GRANLOG é a geração de **informações de granulosidade**, através da detecção do paralelismo existente num programa em lógica. Estas informações podem ser utilizadas em diversas aplicações, tais como auxílio a decisões de escalonamento e simulação da execução de programas (seção 3.5).

A figura 3.1 apresenta o mais alto nível de abstração do GRANLOG, destacando a entrada e a saída do modelo.

FIGURA 3.1 - Mais alto nível de abstração do GRANLOG

O GRANLOG possui como entrada o programa em lógica e como saída o **programa granulado**. O programa granulado é composto do programa em lógica acrescido de **anotações de granulosidade**. As anotações de granulosidade armazenam as informações resultantes da análise de granulosidade. No decorrer dos próximos três capítulos estas anotações serão descritas e comentadas.

Uma importante característica do modelo GRANLOG é o armazenamento das informações de granulosidade diretamente no programa fonte através de anotação. Destacam-se como vantagens desta forma de armazenamento:

- **Encapsulamento:** O programa em lógica e os resultados da análise de granulosidade ficam encapsulados numa única estrutura, ou seja, o programa granulado. Desta forma, diminui-se o número de arquivos a serem tratados e cria-se apenas uma via de comunicação entre o GRANLOG e suas aplicações. A existência de apenas um canal de comunicação simplifica a interface GRANLOG/aplicações;
- **Sintaxe Prolog:** A sintaxe das anotações é compatível com a sintaxe da linguagem Prolog. Desta forma, as aplicações podem realizar o mesmo tratamento léxico/sintático para o programa e as anotações. Sendo assim, as aplicações do GRANLOG podem desfrutar da tecnologia existente para manipulação de programas Prolog. Além disso, o uso de anotações simplifica a adaptação de programas existentes para que passem a utilizar as informações de granulosidade. Por exemplo, os compiladores Prolog que geram código seqüencial podem ser adaptados para geração de código paralelo, simplesmente considerando as novas informações introduzidas no programa fonte pelas anotações de granulosidade;
- **Inteligibilidade:** As anotações organizam as informações de granulosidade de forma clara e adaptada ao contexto do programa, facilitando a compreensão dos resultados da análise de granulosidade. O usuário pode comodamente estudar as anotações e compreender as decisões tomadas pelo GRANLOG. Desta forma, o sistema assume uma finalidade didática e contribui para o estudo da paralelização de programas em lógica. Além disso, através do estudo das anotações o usuário pode reorganizar o programa de forma a explorar mais eficientemente o paralelismo.

Merece destaque ainda, a forte tendência por parte da comunidade científica para utilização de anotações na exploração de paralelismo na programação em lógica. Dentre diversos trabalhos que utilizam anotações destacam-se a proposta de Degroot para o paralelismo E restrito ([DEG84], [DEG87], [DEG89]) e o modelo &-Prolog de Hermenegildo ([HER90], [HER91], [BUE93]).

Existem diversas aplicações que podem utilizar as informações fornecidas pelo GRANLOG. A seção 3.5 apresenta de forma resumida duas destas aplicações, ou seja, auxílio a decisões de escalonamento e simulação da execução de programas. Além destas, destacam-se aplicações tais como reorganização do programa fonte visando aprimorar a exploração do paralelismo e implementação de *backtracking* inteligente ([CHA85a]). O capítulo 8 apresenta em detalhes a integração entre o GRANLOG e um modelo para execução paralela de programas em lógica, denominado OPERA ([WER94a]). A integração OPERA/GRANLOG ([BAR95a]) demonstra a aplicação das informações de granulosidade na exploração do paralelismo na programação em lógica.

3.3 Organização Básica

O GRANLOG é composto de três módulos: **Analisador Global** (AGL), **Analisador de Grãos** (AGR) e **Analisador de Complexidade** (AC). A figura 3.2 apresenta a organização básica do modelo.

O módulo AGL realiza uma análise global do programa, visando determinar os modos, os tipos e as medidas de tamanho dos argumentos de cada procedimento (cláusula ou predicado). O modo de um argumento determina a direcionalidade de sua

instanciação (*entrada*, *saída* ou *entrada/saída*). O tipo de um argumento determina o padrão de sua instanciação (lista, inteiro, etc). A medida de tamanho de um argumento estabelece qual a medida que deve ser utilizada para determinar o tamanho de um argumento. Este tamanho influencia de forma direta na complexidade de um procedimento. O módulo AGL determina ainda as dependências de dados entre os literais de cada cláusula. A análise destas dependências determina a ordem obrigatória para execução dos literais, estabelecendo o fluxo de dados na execução da cláusula. Através de análises, pode-se inferir os modos ([MEL85], [DEB89]), os tipos e medidas de tamanho ([MIS84], [YAR87]) e as dependências de dados ([CHA85], [CHA85a], [DEB89]) para um programa em lógica. O módulo AGL recebe como entrada o programa em lógica e possui como saída os modos, tipos, medidas de tamanho e dependências.

FIGURA 3.2 - Organização básica do GRANLOG

O módulo AGR utiliza as informações de dependências geradas pelo AGL para determinar quais são os grãos existentes no corpo das cláusula. O módulo AGR realiza ainda a análise de entradas e saídas, ou seja, determina quais são as entradas e as saídas dos grãos existentes no programa. As informações inferidas pelo AGR são anotadas no programa fonte. O conjunto destas anotações recebe o nome de **anotação de grãos**. Conforme mostra a figura 3.2, o módulo AGR tem como entradas o programa em lógica e os modos, tipos, medidas de tamanho e dependências resultantes do módulo AGL. A saída do AGR é o **programa particionado**, o qual é composto pelo programa em lógica acrescido da anotação de grãos.

O módulo AC avalia a complexidade dos grãos (granulosidade) detectados pelo Analisador de Grãos e a complexidade das **resolventes locais** para cada literal. Uma resolvente local consiste de todos os literais que seguem um determinado literal numa cláusula. O Analisador de Complexidade determina ainda a relação entre o tamanho das entradas e saídas dos grãos (avaliação dos custos de comunicação). As informações inferidas pelo AC são anotadas no programa particionado. O conjunto de anotações feitas pelo módulo AC recebe o nome de **anotação de complexidade.** Desta forma, o módulo AC recebe como entrada o programa particionado e acrescenta a anotação de

complexidade. O resultado da integração do programa particionado e da anotação de complexidade é denominado **programa granulado**. Portanto, o programa granulado é o resultado final da análise de granulosidade feita pelo GRANLOG. Deve-se ressaltar ainda, que toda a anotação acrescida ao programa em lógica recebe o nome de **anotação de granulosidade**, ou seja, a anotação de granulosidade é composta da anotação de grãos (módulo AGR) e da anotação de complexidade (módulo AC).

3.4 Comparação com Outros Modelos

A subseção 2.3.4 apresenta e analisa diversos modelos para análise de granulosidade na programação em lógica. O estudo destes modelos serviu de base para o desenvolvimento do GRANLOG. Sendo assim, torna-se importante discutir as semelhanças e diferenças existentes entre os principais modelos pesquisados e a proposta apresentada neste trabalho. Através desta discussão, serão compreendidas as principais contribuições do GRANLOG para a comunidade científica.

Basicamente, merecem ser ressaltados os seguintes pontos:

- **Granulosidade E/OU**: De forma inédita, o GRANLOG propõe uma metodologia para análise de granulosidade na programação em lógica envolvendo tanto o paralelismo OU, quanto o paralelismo E (princípio 3, veja seção 3.1). As demais propostas preocupam-se apenas com o paralelismo E. Dentre os textos pesquisados, o único que discute o problema da granulosidade no paralelismo OU é [GAR94]. No entanto, este texto não apresenta uma proposta e discute de forma breve o problema;
- **Tipo de análise**: O GRANLOG propõe uma análise do tipo combinada, ou seja, a realização da análise de granulosidade em tempo de compilação e execução (veja último parágrafo do subseção 2.1.2). Este tipo de análise também é utilizada nos trabalhos descritos em [DEB90], [ZHO92] e [TIC93], e mais recentemente discutida em [GAR94]. Por outro lado, em [TIC90] e [KIN92] são utilizadas análises estáticas;
- **Anotação de granulosidade**: De forma inédita, o GRANLOG propõe uma anotação para registrar as informações de granulosidade obtidas pela análise. Nenhum dos textos pesquisados propõe uma anotação de granulosidade. No entanto, outros trabalhos propõem o controle da execução paralela de programas em lógica através de anotações ([DEG87], [HER91]) e o controle da granulosidade através da reorganização do programa ([DEB90], [TIC93], [GAR94]);
- **Análise dos custos de comunicação**: De forma inédita, o GRANLOG propõe uma metodologia e gera anotações para tratamento dos custos de comunicação em ambientes paralelos que envolvam troca de mensagens (memória distribuída) na execução paralela de programas em lógica. As demais propostas não consideram esta possibilidade pois, ou direcionam-se para ambientes com memória compartilhada ou simplesmente desprezam esta importante fonte de ineficiência na exploração do paralelismo em ambientes distribuídos;
- **Argumentos como canais de comunicação**: De forma inédita, o GRANLOG introduz o conceito de argumentos de procedimentos como canais de comunicação na execução paralela de programa em lógica. Este conceito

permite o tratamento dos custos de comunicação envolvidos na troca de mensagens em ambientes com memória distribuída;

- **Grão composto de diversas metas no paralelismo E**. De forma inédita, o GRANLOG expande o conceito de paralelismo E, permitindo a exploração do paralelismo no corpo das cláusulas, não somente através da paralelização de metas, mas também através da paralelização de partes do corpo da cláusula;
- **Novos modos e tipos**: O GRANLOG propõe um conjunto novo de modos e tipos para a programação em lógica, os quais são utilizados durante a análise de granulosidade e armazenados no programa granulado através da anotação de granulosidade;
- **Análise automática**: Da mesma forma que todas as propostas pesquisadas, o GRANLOG realiza uma análise automática de granulosidade;
- **Modelo genérico - Múltiplas aplicações**: A proposta de criação de um modelo independente do sistema paralelo (primeiro princípio, veja seção 3.1) é inédita. Da mesma forma é inédita, a proposta de criação de um modelo que tenha como resultado informações que possam ser utilizadas em múltiplas aplicações;
- **Amplitude da análise**: O modelo proposto difere das demais abordagens pela amplitude da análise. As demais propostas pesquisadas dedicam-se basicamente à obtenção de informações de complexidade e aplicação destas informações no controle da granulosidade. O GRANLOG envolve a análise global (modos, tipos, medidas e dependências) e a análise de grãos (determinação dos grãos em potencial) na análise de granulosidade, criando assim uma estrutura completa direcionada para obtenção de informações de granulosidade consideradas importantes para paralelização de programas em lógica.

Nos próximos capítulos as características apresentadas nesta seção serão discutidas durante a descrição das etapas da análise de granulosidade. Além disso, novas características do modelo serão analisadas e comparadas com outras abordagens.

3.5 Aplicações

Nesta seção são apresentadas duas propostas para aplicação das informações de granulosidade fornecidas pelo GRANLOG. A subseção 3.5.1 propõe a aplicação no auxílio a decisões de escalonamento. A subseção 3.5.2 descreve a aplicação na simulação da execução de programas.

3.5.1 Auxílio a Decisões de Escalonamento

Utilizando a anotação fornecida pelo GRANLOG, um compilador paralelizador poderá gerar, além do código objeto, informações de granulosidade que auxiliem o escalonamento das tarefas paralelas durante a execução de um programa. O próprio código objeto poderá ser adaptado para conter as decisões de escalonamento, utilizando como base para tomada de decisões as informações de granulosidade.

As informações geradas pelo GRANLOG estão limitadas pela análise estática, ou seja, não são fornecidas informações que dependam do sistema paralelo a ser utilizado. Esta característica mantém o GRANLOG num alto nível de abstração, permitindo portabilidade e flexibilidade para suas anotações (primeiro princípio, apresentado na seção 3.1). Por exemplo, o número de processadores e o custo de comunicação

dependem da arquitetura paralela a ser utilizada. Portanto, estas informações somente poderão ser fornecidas para o escalonador pelo próprio sistema paralelo. A figura 3.3 apresenta uma possível configuração para aplicação das informações geradas pelo GRANLOG no auxilio a decisões de escalonamento.

Pode-se identificar na figura 3.3 três níveis de abstração. Em alto nível de abstração (nível de anotação) encontra-se o GRANLOG, fornecendo informações de paralelismo completamente independentes do sistema paralelo. No nível médio de abstração (nível de compilação) encontra-se o compilador paralelizador, onde poderão ser consideradas algumas informações específicas do sistema paralelo, tal como o tipo de memória (compartilhada ou distribuída). Desta forma, o compilador poderá complementar as informações de granulosidade geradas pelo GRANLOG, adicionando por exemplo, expressões que permitam o cálculo do custo de comunicação dos grãos, baseado no tamanho de suas entradas e saídas. A compilação deverá ser dirigida para um sistema paralelo específico. No entanto, mantém-se ainda neste nível de abstração alguma flexibilidade, por exemplo, quanto ao número de processadores disponíveis e seus tipos. No baixo nível de abstração (nível de execução) encontra-se a arquitetura paralela e o sistema de escalonamento. Neste nível o conhecimento do sistema paralelo deverá ser completo, ou seja, deverão ser conhecidas informações tais como o número de processadores disponíveis e seus tipos.

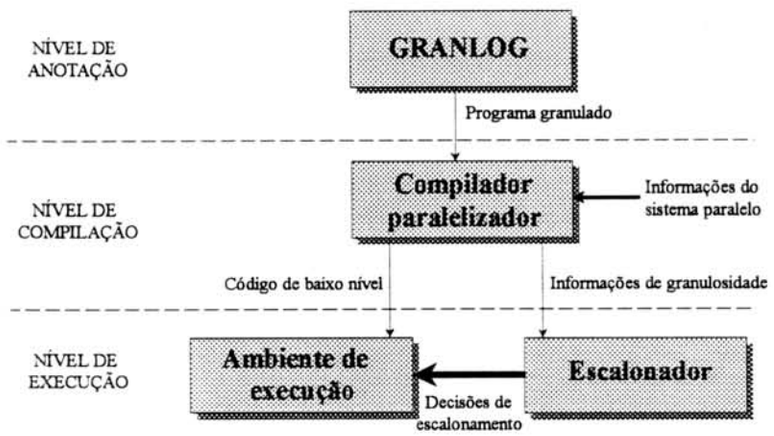


FIGURA 3.3 - Aplicação do GRANLOG no auxílio ao escalonamento

O capítulo 8 apresenta a integração OPERA-GRANLOG. O OPERA ([WER94a]) é um sistema para execução paralela de programas em lógica (nível de execução). Este sistema utiliza as informações fornecidas pelo GRANLOG para aumentar a eficiência na execução dos programas.

3.5.2 Simulação da Execução de Programas

Utilizando as informações geradas pelo GRANLOG pode-se simular a execução de programas em lógica. Dentre as possíveis simulações destacam-se:

- **Simulação da complexidade de programas em lógica.** Nesta simulação, pode-se analisar de forma genérica as características de um programa, tais como: complexidade de execução (metas, cláusulas, predicados e programa), tamanho dos argumentos e dinâmica de execução (iteração das chamadas recursivas);

- **Simulação da paralelização de programas em lógica.** Nesta simulação podem ser previstas as decisões de escalonamento e os custos para execução paralela de um programa. As características do sistema paralelo podem ser alteradas conforme a simulação a ser realizada, permitindo assim a modelagem de várias arquiteturas paralelas e distribuídas.

A figura 3.4 apresenta uma possível configuração para aplicação do GRANLOG na construção de um ambiente para simulação da execução de programas em lógica. Na figura 3.4 o GRANLOG fornece informações de granulosidade para um ambiente, onde será simulada a execução de programas em lógica. Este ambiente poderá fornecer várias opções de simulação e um suporte gráfico para visualização da execução. Além disso, o ambiente poderá proporcionar uma visualização gráfica da dinâmica de execução do programa em lógica, permitindo ao usuário analisar em grafos as alterações da complexidade dos grãos em função das iterações das chamadas recursivas.

FIGURA 3.4 - Aplicação do GRANLOG na simulação de programas em lógica

O ambiente possivelmente permitirá ao usuário a configuração do sistema paralelo a ser utilizado na simulação, determinando o número de processadores e seus tipos. O usuário poderá fornecer ainda informações do tipo de memória (compartilhada ou distribuída), velocidade dos canais de comunicação e existência de processadores de entrada/saída. Desta forma, pode-se simular a execução de programas em vários sistemas paralelos configurados de diversas maneiras.

O autor desta dissertação acredita que a simulação da execução de programas em lógica através da visualização gráfica, será em breve, uma das principais aplicações do GRANLOG. Aconselha-se o texto [SHU89] para aprofundamento dos estudos sobre programação visual no universo da computação. Em [BRO94], encontra-se um estudo da programação visual aplicada à computação paralela. Por sua vez, o texto [CAR93] aborda a visualização gráfica da execução paralela de programas em lógica.

3.6 Conclusões

Este capítulo apresentou uma visão genérica do modelo GRANLOG. Inicialmente, foram apresentados os princípios básicos do modelo. Logo após, descreveu-se de forma resumida a proposta, apresentando uma visão genérica do GRANLOG, acompanhada de uma descrição da organização básica do modelo e um comparação com outras abordagens. Finalmente, foram descritas duas aplicações, ilustrando assim a importância das informações de granulosidade fornecidas pelo modelo proposto.

Após a leitura deste capítulo pode-se concluir que:

- os princípios básicos (seção 3.1) estabelecem três condições indispensáveis para o desenvolvimento do modelo proposto;
- o GRANLOG é um modelo genérico o bastante para encontrar aplicações em diversas áreas de pesquisa da programação em lógica;
- o GRANLOG apresenta várias contribuições para o estudo da análise de granulosidade na programação em lógica;
- destaca-se a aplicação do modelo na exploração do paralelismo nos programas em lógica;
- a organização modular do GRANLOG facilita sua manutenção e adaptação para aplicações futuras;
- o modelo serve de suporte para organização da pesquisa sobre análise automática de granulosidade desenvolvida neste trabalho.

Nos próximos três capítulos são apresentadas as três etapas da análise de granulosidade proposta pelo GRANLOG, ou seja, análise global, análise de grãos e análise de complexidade.

4 Análise Global

Neste capítulo é apresentado o primeiro estágio do GRANLOG, ou seja, a análise global. Este estágio é implementado pelo primeiro módulo do modelo, denominado Analisador Global (AGL). A seção 4.1 apresenta uma introdução à análise global, com o intuito de contextualizar o leitor e destacar diversos conceitos e referências. Por sua vez, a seção 4.2 descreve o módulo Analisador Global, iniciando assim a discussão da análise global no contexto do GRANLOG. A seção 4.3 aborda o conceito de modos na programação em lógica e apresenta os modos propostos neste trabalho. Na seção 4.4 discute-se tipos na programação em lógica, descrevendo-se os tipos definidos e utilizados no GRANLOG. A seção 4.5 aborda o tópico medidas de tamanho de termos na programação em lógica, destacando as medidas usadas no modelo proposto. A seção 4.6 descreve a análise de dependências nos programas em lógica e apresenta a abordagem proposta neste trabalho. A seção 4.7 demonstra, através de um exemplo completo, como a análise global é realizada no GRANLOG. Finalmente, a seção 4.8 apresenta as conclusões deste capítulo.

4.1 Introdução

Um programa é uma descrição textual da solução de um problema. Conforme discutido na subseção 2.1.4, esta descrição pode assumir diversas formas (comandos, funções, lógica, etc) dependendo do paradigma de programação. No entanto, independentemente do paradigma, o programa é uma representação estática utilizada para controlar a dinâmica de execução. A análise desta representação estática permite a obtenção de informações que podem ser utilizadas para diversas finalidades, tais como, aprimoramento da execução seqüencial, depuração do programa e paralelização automática. Esta análise do texto do programa recebe o nome de **análise estática**. Na programação em lógica, a análise estática pode ser realizada a nível de cláusula ou a nível de programa. Quando realizada a nível de cláusula é denominada **análise local**. Desta forma, a análise local está restrita ao escopo de uma cláusula, o que simplifica a análise, mas limita a quantidade e a precisão das informações obtidas. Dentre as propostas de análise local destaca-se o modelo de Degroot para exploração do paralelismo E ([DEG84], [DEG87], [DEG89]). Quando realizada a nível de programa, a análise estática recebe o nome de **análise global**. A análise global é complexa, mas no entanto, atinge altos níveis de precisão e resulta numa grande quantidade de informações.

Uma técnica para realização da análise global é a **interpretação abstrata**. A intepretação abstrata em linguagens imperativas foi proposta inicialmente por Cousot e Cousot ([COU77]). Em estudos posteriores, foi demonstrado por Bruynooghe ([BRU87a]), Jones e Sondergaard ([JON87]) e Mellish ([MEL86]) que esta técnica pode ser estendida para a programação em lógica. A interpretação abstrata consiste na simulação da execução de um programa com a utilização de um domínio abstrato. Este domínio preserva de forma genérica os aspectos interessantes que serão compartilhados pelas execuções reais. Basicamente, a interpretação abstrata permite uma análise do fluxo de dados e a previsão do comportamento dinâmico do programa.

A complexidade da análise global depende das informações a serem inferidas. Por sua vez, estas informações dependem diretamente da aplicação. Dentre as principais informações que podem ser inferidas através da análise global de programas em lógica

destacam-se: modos, tipos, medidas de tamanho dos argumentos dos procedimentos e dependências de dados entre os literais das cláusulas.

Atualmente existem várias propostas de análise global na programação em lógica. Em [CHA85a] é apresentada uma proposta para determinação das dependências de dados com o objetivo de implementar *backtracking* inteligente. A mesma abordagem é utilizada em [CHA85] para paralelização de programas em lógica. Dentro do projeto &-Prolog ([HER90], [HER91], [BUE93]) foram desenvolvidos diversos trabalhos relacionados com interpretação abstrata. Estes trabalhos foram aplicados no aperfeiçoamento do sistema &-Prolog. Deve-se destacar ainda, a interessante proposta de Debray ([DEB89]) para inferência de modos e dependências de dados, a clássica abordagem de interpretação abstrata de Mellish ([MEL81]) aplicada à otimização de compiladores ([MEL85]) e o texto [BRU87] que propõe um modelo para inferência de modos e tipos através da interpretação abstrata.

4.2 Módulo Analisador Global

O primeiro módulo do GRANLOG realiza a análise global de programas em lógica. Esse módulo é denominado **Analisador Global (AGL)**. A figura 4.1 apresenta uma visão genérica do módulo AGL. Conforme mostra a figura, o AGL recebe o programa em lógica e infere os modos, tipos e medidas de tamanho dos argumentos dos procedimentos. Além disso, são determinadas as dependências de dados entre os literais de cada cláusula. Desta forma, o AGL produz informações sobre o programa, as quais serão utilizadas pelo demais módulos do GRANLOG.

FIGURA 4.1 - Módulo Analisador Global

Atualmente, o módulo AGL não implementa uma análise global. Esta implementação é complexa e será pesquisada e desenvolvida em trabalhos futuros. A análise global está sendo amplamente discutida pela comunidade científica. Segundo Nai-Wei Lin ([LIN93]), a ampliação do seu trabalho sobre análise automática de complexidade de programas em lógica incluirá a análise global. Além disso, Debray e Hermenegildo têm dedicado esforços no sentido de aprimorar técnicas de interpretação abstrata ([BUE93], [DEB94]).

O GRANLOG segue a estratégia proposta em [LIN93] e cria três declarações para o usuário introduzir nos programas os modos, tipos e medidas de tamanho dos argumentos dos procedimentos. Estas declarações são apresentadas e discutidas nas próximas seções deste capítulo. Por outro lado, a análise de dependências é realizada

automaticamente pelo módulo AGL (algoritmo apresentado na figura 4.7). Desta forma, o modelo propõe a análise global de programas, mas no entanto limita-se a estabelecer as informações de modos, tipos e medidas que deverão ser inferidas. Sendo assim, o GRANLOG cria uma estrutura para aplicação da análise global, definindo claramente as informações que devem ser inferidas e as interfaces entre os módulos do modelo. Na medida em que os modos, tipos e medidas forem obtidos através de trabalhos futuros sobre análise global, o usuário será poupado do uso de declarações.

O autor desta dissertação acredita que a realização de uma análise global via interpretação abstrata é indispensável para evolução do GRANLOG. A completa concretização do segundo princípio do modelo (análise automática, conforme descrito na seção 3.1) depende do aprimoramento do módulo AGL, através da inclusão da análise global. A liberação do uso de declarações por parte do usuário é vital para a principal tese defendida neste trabalho, ou seja, a exploração completamente automática do paralelismo.

4.3 Modos

Esta seção descreve o conceito de modos na programação em lógica e apresenta a utilização de modos no GRANLOG.

4.3.1 Modos na Programação em Lógica

A noção de **modos** na programação em lógica foi introduzida por Warren ([WAD77]) como uma forma de classificação das chamadas de procedimentos nos programas em lógica. Esta classificação é baseada na descrição do estado de instanciação dos argumentos no momento da chamada do procedimento. Desta forma, surgem os conceitos de **modo de argumento** e **modo de procedimento**. O modo de argumento descreve o estado de instanciação de um único argumento no momento da chamada. Por sua vez, o modo de procedimento define completamente uma chamada através da descrição do estado de instanciação de todos os argumentos.

Diferentes trabalhos de pesquisa tem considerado diferentes conjuntos de modos de argumentos. Por exemplo, Warren ([WAD77]) utiliza as seguintes descrições para os estados de instanciação dos argumentos no momento das chamadas: instanciado (+), não-instanciado (-) e desconhecido (?). O modo não-instanciado indica que o argumento é uma variável livre. Em [RED84] e [MAN87] são considerados os modos fechado (*ground*) e desconhecido (*unknown*). O modo fechado equivale a um argumento sem variáveis livres. Por sua vez, Debray ([DEB89]) define os modos: fechado (c), desconhecido (d), vazio (e), não-instanciado (f) e instanciado (nv). Mellish ([MEL81], [MEL85]) adota os modos: desconhecido (o), não-instanciado (-), instanciado com uma estrutura onde os argumentos estão todos não-instanciados (+-), fechado (++), instanciado (+) e com possibilidade de instanciação (?). Na tese de Nai-Wei Lin ([LIN93]) são utilizados apenas dois modos, ou seja, fechado (+) e aberto (-). No trabalho de Lin, o estado aberto abrange os modos instanciado (contendo variáveis livres) e não-instanciado (variável livre) das demais abordagens.

Destaca-se das demais abordagens a classificação de modos de argumentos utilizada em [DIE88] e [DIE88a]. Esta proposta não classifica os modos pelo estado de instanciação dos argumentos e sim pela direção do fluxo de dados vinculado com um

argumento em determinada chamada de procedimento. São utilizados os seguintes modos:

- **entrada (i)**: O argumento não sofre instanciações durante a execução do procedimento. Desta forma, o argumento transporta informações para o procedimento e não retorna resultados. Portanto, o fluxo de dados entra no procedimento. Neste caso, normalmente o argumento está fechado antes da chamada. O modo de entrada não possui equivalência com as demais classificações, mas aproxima-se do modo fechado;
- **saída (o)**: O argumento é uma variável livre antes da chamada. Desta forma, o argumento não leva informações, mas no entanto retorna resultados. Portanto, o fluxo de dados sai do procedimento. Este modo equivale ao modo não-instanciado das demais abordagens;
- **entrada/saída (io)**: O estado de instanciação do argumento é desconhecido. Desta forma, também é desconhecida a direção do fluxo de dados. O modo entrada/saída equivale ao modo desconhecido de algumas das abordagens que consideram os estados de instanciação.

A classificação de modos através da direção do fluxo de dados recebe o nome de **enfoque direcional**. O enfoque direcional é utilizado no restante do texto e especialmente na subseção 4.3.2, onde são definidos os modos do GRANLOG. Existe uma forte relação entre o enfoque direcional de modos e a visão procedimental da unificação (passagem de parâmetros) na chamada de procedimentos nos programas em lógica. Na subseção 2.2.6 é descrita esta visão, acompanhada de um exemplo (figura 2.6). Naquele exemplo, o procedimento *append* é chamado com os dois primeiros argumentos fechados e o último aberto. Desta forma, no enfoque direcional, o modo do procedimento *append* no exemplo da figura 2.6 é *entrada/entrada/saída*.

Em [LIN93] é utilizado um enfoque direcional diferente do proposto em [DIE88] e [DIE88a]. No trabalho de Nai-Wei Lin, o modo fechado (argumento sem variáveis livres) é referenciado como modo de entrada e o modo aberto (argumento contendo variáveis livres) é referenciado como modo de saída. Desta forma, as duas abordagens são incompatíveis, pois as definições de modo de entrada e saída são diferentes.

Conforme afirma Lin ([LIN93]), na programação em lógica um procedimento pode ser invocado por duas chamadas com diferentes características direcionais. Desta forma, pode-se associar um conjunto de modos a cada procedimento. Por outro lado, na programação convencional um procedimento possui apenas um único modo. No entanto, Nai-Wei Lin afirma ainda, que apesar da possibilidade de **múltiplos modos**, na prática, cada procedimento nos programas em lógica possui um pequeno número de modos.

No texto [LIN93], o modo de um procedimento é **definido** se todas as chamadas neste modo que resultam em sucesso, retornam argumentos de saída fechados. Caso contrário, o modo é dito **indefinido**. Deve-se notar que no trabalho de Nai-Wei Lin todos os argumentos que não estão fechados antes da chamada são considerados argumentos de saída. Seguindo a definição de Lin, na figura 2.6 o procedimento *append* possui um modo definido, pois o argumento de saída retorna fechado. Em [LIN93] é apresentado um exemplo de um procedimento com modo indefinido (procedimento *lookup*).

Existem basicamente dois métodos para tratamento de modos na programação em lógica, ou seja, **anotação pelo usuário** e **inferência automática**. No primeiro método, o usuário introduz no programa a descrição dos modos dos procedimentos através de anotações. Estas anotações recebem o nome de **declarações de modos** ([WAD77], [LIN93]). Por outro lado, no segundo método os modos são inferidos automaticamente através de uma análise global ([BRU87], [DEB89]). A principal vantagem da anotação é a simplificação do sistema computacional. A principal vantagem da inferência automática é a liberação do usuário do envolvimento com o tratamento dos modos.

Conforme afirma Debray ([DEB89]), as informações de modos encontram diferentes aplicações em sistemas de alto desempenho para programação em lógica. Os modos podem ser utilizados na geração de código especializado para unificação. Este código é mais eficiente do que rotinas de propósito geral porque reduz o número de casos a serem tratados ([WAD77], [VAN87]). Outra aplicação é a detecção de determinismo e computações funcionais, reduzindo desta forma, o esforço realizado na busca de soluções ([MEL85], [DEB89a]). Os modos são importantes ainda na integração de programas em lógica e linguagens de programação funcional ([RED84]). Além disso, os modos podem ser aplicados na determinação das dependências (seção 4.6), as quais são utilizadas em várias transformações para otimização de programas em lógica ([DEB88]), no aperfeiçoamento do *backtracking* ([CHA85a]) e na paralelização de programas em lógica ([CHA85], [WAR88]).

4.3.2 Modos no GRANLOG

O GRANLOG utiliza quatro modos de argumentos, ou seja: **entrada**, **saída**, **entrada/saída** e **indefinido**. Um argumento é considerado entrada se nunca sofre nenhuma instanciação durante a execução do procedimento, ou seja, o argumento é consumido pelo procedimento. Normalmente, um argumento de entrada está fechado antes da chamada. Um argumento de saída sempre será uma variável livre antes da chamada de um procedimento, ou seja, este argumento será produzido pelo procedimento. Um argumento será entrada/saída se estiver parcialmente instanciado antes da chamada (entrada) e sofrer instanciações durante a execução do procedimento (saída). Uma parte deste argumento é consumida pelo procedimento e outra é produzida (instanciações adicionais). Finalmente, um argumento será indefinido quando não for possível identificar qual será seu estado de instanciação no momento da chamada. Os modos dos argumentos são abreviados pelos caracteres ***i*** (*input*), ***o*** (*output*), ***io*** (*input/output*) e ***?*** (indefinido).

Os quatro modos definidos no parágrafo anterior permitem uma abordagem simplificada e completa do enfoque direcional. Este enfoque é utilizado pelo GRANLOG na busca de seu principal objetivo, ou seja, a paralelização eficiente de programas em lógica. O autor desta dissertação acredita que a classificação direcional de modos apresentada por Dietrich ([DIE88], [DIE88a]) destaca-se das demais propostas discutidas na subseção 4.3.1. Esta abordagem possui três modos simples, mas bem adaptados a análise direcional. A definição de modos do GRANLOG assemelha-se a proposta de Dietrich. A diferença encontra-se na criação do modo indefinido e na redefinição do modo de entrada/saída. O modo entrada/saída de Dietrich equivale ao modo indefinido do GRANLOG. O modo entrada/saída criado pelo GRANLOG não encontra equivalente na abordagem de Dietrich. A principal vantagem da nova classificação consiste na organização dos modos de forma a suportar a análise dos **três**

tipos possíveis de fluxos de dados a serem canalizados através de um argumento na execução de uma meta. A nova classificação permite a análise de argumentos de entrada, argumentos de saída e **argumentos de entrada/saída**. A classificação de Dietrich suporta a análise de apenas dois tipos de fluxo de dados, ou seja, entrada e saída. O modo indefinido criado pelo GRANLOG (modo entrada/saída de Dietrich) não permite nenhum tipo de análise de fluxo de dados. Para este modo, a denominação **indefinido** proposta pelo GRANLOG é mais adequada do que o nome entrada/saída apresentada por Dietrich.

Comparando-se a proposta de modos do GRANLOG com o trabalho [LIN93], constata-se que existem algumas incompatibilidades. Por exemplo, na abordagem de Lin o argumento que possui modo de entrada deve estar fechado antes da chamada. No GRANLOG e no trabalho de Dietrich, um argumento de entrada não deve sofrer instanciações adicionais durante a execução do procedimento, no entanto, não é necessário que esteja fechado antes da chamada. Além disso, o trabalho de Lin classifica como saída todos os argumentos que estejam abertos no momento da chamada. Esta definição é completamente incompatível com o GRANLOG e o trabalho de Dietrich, pois nestas abordagens, um argumento com modo de saída deve conter uma variável livre. Finalmente, os modos utilizados em [LIN93] restringem-se a modo de entrada e modo de saída, enquanto o GRANLOG possui quatro modos e Dietrich possui três classificações.

Do ponto de vista do paralelismo na programação em lógica, os modos determinam o fluxo de dados entre os grãos durante a exploração do paralelismo E. Por outro lado, a exploração do paralelismo OU envolve um fluxo de dados (pilhas da máquina abstrata) que não pode ser dimensionado através dos modos de argumentos. No caso do paralelismo E, a análise dos modos permite a identificação das entradas e saídas de cada um dos grãos. Se um procedimento for executado em determinado processador, seus argumentos de entrada e entrada/saída deverão estar disponíveis no início da execução. Em arquiteturas de memória distribuída esta exigência implica o envio de mensagens entre processadores. O processador origem (exportador de trabalho) deverá enviar os argumentos de entrada e entrada/saída para o processador destino (importador de trabalho).

Por outro lado, os resultados da execução de um procedimento devem ficar disponíveis no processador que executou a chamada para o procedimento (exportador). No caso de arquiteturas com memória distribuída, esta situação exige o envio de uma mensagem do processador importador para o processador exportador contendo os resultados, ou seja, os argumentos de saída e de entrada/saída alterados. Portanto, os argumentos dos procedimentos podem ser considerados como canais de comunicação unidirecionais (entrada e saída) ou bidirecionais (entrada/saída) entre os processadores durante a exploração do paralelismo E. A figura 4.2 mostra as possíveis configurações de modos como canais de comunicação entre processadores. Um argumento com modo indefinido gera fluxo de dados. No entanto, a direção desse fluxo é desconhecida. Este fato é demonstrado pela ausência do modo indefinido na figura 4.2.

No GRANLOG, o usuário deve especificar os modos dos procedimentos através de declarações de modos. Cada procedimento do programa em lógica deve obrigatoriamente ter apenas um modo associado. Desta forma, existe apenas um padrão de instanciação que pode ser utilizado para chamada de um procedimento. Seguindo os passos de Nai-Wei Lin ([LIN93]), quando um procedimento deve ser chamado com *n*

diferentes modos (múltiplos modos), o programador deve renomear a definição do procedimento em *n* diferentes versões, de forma que cada uma delas corresponda a um dos diferentes modos. No entanto, conforme citado na subseção 2.2.6 e na subseção 4.3.1, na prática os procedimentos possuem um pequeno número de modos. Na verdade, segundo Debray ([DEB89]), pode-se afirmar que é bastante freqüente o caso em que um procedimento em determinado programa possua apenas um modo.

Apesar da versão atual do GRANLOG não tratar múltiplos modos, conforme será descrito no capítulo 5, as anotações de grão criadas neste trabalho suportam a especificação de um modo para cada chamada de procedimento e desta forma, suportam a utilização de múltiplos modos. Sendo assim, o modelo atual é flexível o bastante para viabilizar a utilização de diversos modos por procedimento. Uma das principais características de um modelo é a flexiblidade para suportar futuras melhorias. Dentre os principais aperfeiçoamentos que deverão ser introduzidas futuramente no GRANLOG encontram-se a inferência automática de modos e o tratamento de múltiplos modos.

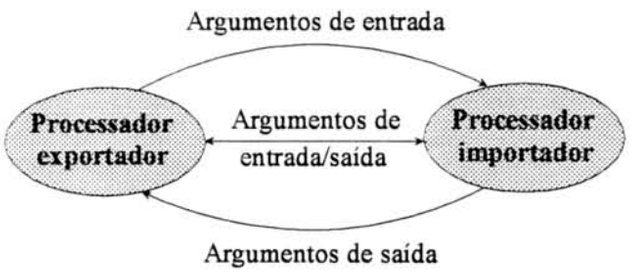


FIGURA 4.2 - Modos de argumentos como canais de comunicação

Deve-se destacar ainda, que o GRANLOG introduz o novo conceito de modo indefinido (simbolizado por *?*). Este conceito aumenta a flexibilidade no tratamento dos modos, pois permite ao usuário estabelecer que o modo de um argumento não pode ser definido e desta forma pode assumir qualquer estado de instanciação no momento da chamada do procedimento. Sendo assim, o sistema continua não suportando múltiplos modos, mas introduz um modo que é flexível o bastante para permitir vários padrões de instanciação para chamadas do mesmo procedimento. Por exemplo, o usuário pode estabeler que o procedimento *append* possui os três argumentos com modo indefinido. Portanto, este procedimento pode ter diversas chamadas com diferentes padrões de instanciação para os argumentos. O procedimento possui apenas um modo, mas amplo o bastante para suportar todos os demais. Deve-se ressaltar no entanto, que neste caso o modo é tão amplo que não introduz nenhuma informação no sistema e portanto não contribui com a análise de granulosidade.

A sintaxe da declaração de modos utilizada pelo GRANLOG é a seguinte:

```
:- mode(p/n,lista_de_modos).
```

Na declaração de modos, *p/n* é o identificador do procedimento, ou seja, nome/aridade e *lista_de_modos* é uma lista contendo em ordem os modos dos argumentos do procedimento. Por exemplo, o procedimento *append* da figura 2.6 deverá possuir a seguinte declaração de modos:

```
:- mode(append/3,[i,i,o]).
```

No caso de todos os argumentos de um procedimento possuirem o mesmo modo, o usuário poderá abreviar a *lista_de_modos*, colocando apenas um modo na lista. Por exemplo, se todos os argumentos do procedimento *append* forem de entrada, o usuário poderá utilizar a seguinte declaração:

```
:- mode(append/3,[i]).
```

Além disso, o usuário deve estar ciente de que a declaração pode ser introduzida em qualquer ponto do programa. No entanto, aconselha-se por questão de clareza, que as declarações sejam colocadas imediatamente antes do procedimento. Esta conduta é aconselhada em diversos trabalhos que utilizam declarações na programação em lógica.

Da mesma forma que o trabalho desenvolvido por Nai-Wei Lin ([LIN93]), o GRANLOG manipula apenas procedimentos com modos definidos e cláusulas **acíclicas com modos corretos** ([RED84]) em Prolog. A definição completa de cláusula acíclica com modos corretos é bastante complexa e não será apresentada neste trabalho. No entanto, intuitivamente uma cláusula enquadra-se neste padrão quando sua estrutura e os modos dos argumentos das suas metas componentes geram fluxos de dados contínuos e sem ciclos.

O GRANLOG adapta o conceito de modo definido apresentada por Lin (veja subseção 4.3.1) para a nova proposta de modos. No GRANLOG um procedimento possui modo definido se e somente se todas as chamadas neste modo que resultam em sucesso, retornam fechados os argumentos de saída, entrada/saída e indefinido. Na verdade, ambas as definições são bastante semelhantes, pois o modo de saída (modo aberto) de Lin equivale aos modos de saída, entrada/saída e indefinido do GRANLOG. Através desta mesma equivalência, adapta-se no GRANLOG a definição para cláusulas acíclicas com modos corretos apresentada em [DEB93] e [LIN93]. Em [LIN93] são apresentados exemplos de cláusulas acíclicas com modos corretos e de cláusulas que não enquadram-se neste padrão.

4.4 Tipos

Esta seção descreve de forma genérica os tipos na programação em lógica e apresenta especificamente os tipos definidos e utilizados pelo GRANLOG.

4.4.1 Tipos na Programação em Lógica

Nas linguagens de programação convencionais a noção de tipos é amplamente utilizada. Um **tipo** consiste numa especificação que limita o universo de valores que podem ser assumidos por um objeto da linguagem. Segundo Cardelli ([CAR89]), o uso de informações de tipos pode ser visto como uma especificação parcial do programa. A utilização de tipos possui como principais benefícios: permitir a alocação estática de memória, auxiliar na depuração de programas e aprimorar a documentação.

A programação em lógica é **atipada,** ou seja, não possui definições de tipos. Esta característica aumenta a flexibilidade dos programas e potencializa a utilização da variável lógica. No entanto, na prática os termos utilizados por um programa em lógica normalmente podem ser enquadrados num conjunto de tipos definidos. Vários trabalhos dedicam-se ao estudo de tipos na programação em lógica ([BRU87], [YAR87], [AZZ88], [DIE88a], [RIB92], [RIB92a], [PHU92], [MOR94]).

Do ponto de vista procedimental, a tipagem possibilita a detecção dos tipos dos argumentos de um procedimento num programa em lógica. Por exemplo, os argumentos do procedimento *append*, mostrado na figura 2.5, são necessariamente do tipo **lista**, pois os três argumentos sempre serão listas. A determinação dos tipos é realizada através da análise estática do programa. Por exemplo, na codificação do procedimento *append* está explícito que os argumentos devem ser listas (utilização de colchetes).

A figura 4.3 apresenta um procedimento para obtenção de números da série de *Fibonacci*. A análise deste procedimento determina que seus argumentos são do tipo **inteiro**. Este fato deve-se a duas constatações. Primeiro, a terceira cláusula utiliza os dois argumentos em operações aritméticas (*is*). Desta forma, os argumentos devem ser números, mas não necessariamente inteiros. Segundo, as duas primeiras cláusulas estabelecem o término da recursividade apenas para valores inteiros. Assim, fica estabelecido que o procedimento exige argumentos do tipo inteiro.

```
fib(0,0).
fib(1,1).
fib(M,N) :-
      M > 1, M1 is M - 1, M2 is M - 2,
      fib(M1,N1), fib(M2,N2), N is N1 + N2.
```

FIGURA 4.3 - Procedimento *fibonacci*

Dentre os principais tipos encontrados na programação em lógica destacam-se: lista, inteiro, ponto-flutuante, átomo e estrutura. Os tipos são parte fundamental da proposta apresentada neste texto.

Conforme descrito em [RIB92] e [RIB92a], pode-se classificar as propostas relacionadas com tipos na programação em lógica, através da forma como são obtidas as informações de tipos. Esta classificação resulta em dois grupos, ou seja:

- **Declarações de tipos**: Neste grupo de propostas, as informações de tipos são obtidas através de declarações explícitas, geralmente introduzidas no programa pelo usuário ([MYC84], [ESC89], [JAC90]);
- **Inferência de tipos**: Neste grupo, as informações são inferidas a partir das construções presentes no programa ([XUJ88], [AZZ89], [FRU89]).

Confirmando esta classificação, Yardeni e Shapiro ([YAR87]) utilizam estes dois grupos para apresentar sua proposta. Além disso, em [YAR87] pode-se encontrar um interessante histórico da evolução dos tipos na programação em lógica. Deve-se destacar ainda, os trabalhos [BRU87] e [PHU92]. Nestas duas abordagens os autores integram o estudo de modos e tipos, resultando em interessantes fontes de informações e referências. Em [BRU87] é apresentada uma proposta de interpretação abstrata e sua aplicação na inferência de modos e tipos. Por sua vez, o texto [PHU92] descreve um sistema declarativo de tipos para Prolog e apresenta de forma resumida a relação existente entre tipos e modos.

4.4.2 Tipos no GRANLOG

O GRANLOG utiliza os seguintes tipos para os argumentos dos procedimentos de um programa em lógica: inteiro (*int*), lista (*list*), estrutura (*struct*), ponto-flutuante (*float*), átomo (*atom*), variável (*var*), indefinido (*?*) e entrada/saída (*io*). A determinação de tipos possui como principal objetivo permitir a previsão dos custos de comunicação durante a execução paralela de um programa em lógica.

Em arquiteturas de memória distribuída a execução de tarefas em paralelo depende da troca de mensagens entre processadores. Estas mensagens representam um dos principais custos da paralelização de um programa. O custo para transmissão de uma mensagem é função do seu tamanho. Portanto, a previsão do tamanho da mensagem permitirá a previsão do custo. Nos programas em lógica, as mensagens entre processadores possuem como principal conteúdo as entradas e saídas dos grãos. Desta forma, conhecendo-se o tipo e o tamanho dos argumentos, pode-se prever o tamanho das mensagens e consequentemente o custo de comunicação envolvido na execução de um grão. Os tipos podem ser inferidos através da análise estática. O tamanho de um argumento de entrada e saída será obtido durante a execução.

A seguir é apresentada a notação de tipos utilizada no GRANLOG:

- Inteiro: **int**

Indica que o argumento é um inteiro. Não possui parâmetros. Por exemplo, o número 100 será representado simplesmente como *int*.

- Ponto-flutuante: **float**

Indica que o argumento é um número de ponto-flutuante. Não possui parâmetros. Por exemplo, o número 24.4 será representado simplesmente como *float*.

- Átomo: **atom(*TAMANHO*)**

Indica que o argumento é um átomo. O parâmetro *TAMANHO* contém o número de caracteres do identificador do átomo. O símbolo *?* será utilizado no caso de indefinição de *TAMANHO*. Por exemplo, o átomo *mae* será representado como *atom(3)*.

- Variável: **var**

Indica que o argumento é uma variável livre. Neste caso não são utilizados parâmetros.

- Lista: **list(*TAMANHO,TIPOS*)**

Indica que o argumento é uma lista. O parâmetro *TAMANHO* contém o número de elementos da lista e o parâmetro *TIPOS* é uma lista contendo em ordem o tipo de cada um dos elementos. Se o número de elementos de *TIPOS* for menor que o *TAMANHO*, então o conteúdo de *TIPOS* determina a proporção dos tipos dos elementos da lista representada. O símbolo *?* será utilizado no caso de indefinição dos parâmetros. Por

exemplo, a lista *[1,abc,123.45,2,def,543.21]* será representada como *list(6,[int,atom(3),float])*. Neste caso, *TAMANHO* é maior do que o número de elementos em *TIPOS*. A divisão de *TAMANHO* pelo tamanho de *TIPOS* resulta em 2, ou seja, existem 2 elementos de cada tipo na lista.

- Estrutura: **struct(*FUNCTOR,ELEMENTOS,TIPOS*)**

Indica que o argumento é uma estrutura. O parâmetro *FUNCTOR* contém o tamanho do functor. O parâmetro *ELEMENTOS* determina o número de argumentos da estrutura. O parâmetro *TIPOS* é uma lista contendo o tipo de cada um dos argumentos da estrutura. Se o número de elementos de *TIPOS* for menor do que *ELEMENTOS*, então o conteúdo de *TIPOS* determina a proporção dos tipos dos argumentos da estrutura representada. O símbolo *?* será utilizado em caso de indefinição dos parâmetros. O caractere ***r*** será utilizada no parâmetro *TIPOS* para definir uma estrutura recursiva (exemplificado na subseção 4.5.2 com o procedimento *traverse* mostrado na figura 4.6). Por exemplo, a estrutura teste(100,12.3) será representada como *struct(5,2,[int,float])*.

- Indefinido: **?**

Indica que o tipo do argumento não pode ser determinado.

- Entrada/saída: **io(*ENTRADA,SAÍDA*)**

Indica que o argumento possui duas notações de tipos, uma para a entrada e outra para a saída. O tipo *io* é utilizado em argumentos de entrada/saída para indicar o estado de instanciação antes da chamada (entrada) e após a chamada (saída). O parâmetro *ENTRADA* descreve a instanciação antes da chamada. O parâmetro *SAÍDA* indica o estado de instanciação após a chamada. O símbolo *?* será utilizado em caso de indefinição dos parâmetros. Um exemplo deste tipo poderia ser *io(list(3,[var]),list(3,[int]))*.

Conforme demonstrado em alguns dos exemplos, os tipos propostos pelo GRANLOG podem ser combinados, permitindo assim a representação de qualquer termo em Prolog. Além disso, o tipo indefinido pode ser utilizado quando não for possível determinar o tipo do argumento.

Na versão atual do GRANLOG o usuário deve introduzir os tipos dos argumentos dos procedimentos através de declarações. No entanto, os tipos utilizados pelo GRANLOG podem ser inferidos através de análises automáticas. Além disso, o modelo proposto é independente da fonte das informações de tipo, o que permite a introdução da inferência automática em versões futuras. Esta dissertação não aprofunda o estudo da inferência de tipos e nem propõe um modelo para realização desta tarefa.

Cada procedimento do programa em lógica deve obrigatoriamente ter apenas uma declaração de tipos. Desta forma, se um procedimento deve ser chamado com *n* diferentes tipos (múltiplos tipos), o programador deve renomear a definição do procedimento em *n* diferentes versões, de forma que cada uma delas corresponda a um dos tipos. Apesar da versão atual do GRANLOG não tratar múltiplos tipos, conforme será descrito no capítulo 5, as anotações de grãos criadas neste trabalho suportam a

especificação de um tipo para cada chamada de procedimento e desta forma, suportam a utilização de múltiplos tipos. Sendo assim, o modelo atual é flexível o bastante para viabilizar a utilização de diversos tipos por procedimento. Dentre os principais aperfeiçoamentos que deverão ser introduzidos futuramente no GRANLOG encontram-se o tratamento de múltiplos tipos. Este aperfeiçoamento surgirá naturalmente quando for introduzida a inferência automática.

A sintaxe da declaração de tipos no GRANLOG é a seguinte:

```
:- type(p/n,lista_de_tipos).
```

Na declaração de tipos, *p/n* é o identificador do procedimento, ou seja, nome/aridade e *lista_de_tipos* é uma lista contendo em ordem os tipos dos argumentos do procedimento. Por exemplo, o procedimento *append*, apresentado na figura 2.5, poderia possuir a seguinte declaração de tipos:

```
:- mode(append/3,[list(?,[int]),list(?,[int]),list(?,[int])]).
```

Esta declaração de tipos estabelece que os três argumentos do procedimento *append* são listas de inteiros. A colocação do caractere *?*, no primeito parâmetro da especificação de tipo, indica que as listas podem ter qualquer tamanho. Desta forma, o procedimento *append* deverá sempre ser chamado com listas de inteiros, mantendo-se no entanto, a flexibilidade quanto ao tamanho destas listas.

No caso de todos os argumentos de um procedimento possuírem o mesmo tipo, o usuário poderá abreviar a *lista_de_tipos*, colocando apenas um tipo na lista. Por exemplo, a declaração de tipos do procedimento *append* poderá ser simplificada para:

```
:- mode(append/3,[list(?,[int])]).
```

Assim como as declarações de modos, as declaração de tipos podem ser introduzidas em qualquer ponto do programa. No entanto, da mesma forma, aconselha-se por questão de clareza, que as declarações sejam colocadas imediatemente antes do procedimento.

Deve-se ressaltar ainda, que mesmo sem a inferência automática de tipos, diversos aperfeiçoamentos podem ser feitos para aumentar a precisão das informações tratadas pelo GRANLOG. Dentre estas melhorias destaca-se a **adaptação da declaração a chamadas**. Este aperfeiçoamento consiste na adaptação da declaração de tipos a cada uma das chamadas para o procedimento que surgirem no programa. Por exemplo, a declaração apresentada para o procedimento *append* pode ser adaptada à realidade de cada chamada, onde o tamanho das listas de entrada pode ser conhecido. Se o sistema encontrar uma chamada *append(Lista,[1,2,3,4,5],Lista_res)*, pode adaptar o argumento *lista_de_tipos* para *[list(?,[int]),list(5,[int]),list(?,[int])]*. A adaptação da declaração a chamadas ficará mais clara quando, no capítulo 5, for apresentada a anotação de grãos.

4.5 Medidas de Tamanho de Termos

Nesta seção é descrita a utilização de medidas de tamanho de termos na programação em lógica e sua aplicação no GRANLOG.

4.5.1 Medidas de Tamanho de Termos na Programação em Lógica

Segundo Nai-Wei Lin ([LIN93]), várias medidas podem ser utilizadas para determinar o tamanho de um termo na programação em lógica. Estas medidas variam de acordo com o tipo do termo e sua aplicação. Por exemplo, em determinadas situações é útil saber o número de elementos de uma lista. Neste caso, a medida para o tamanho do termo será o número de elementos da lista. Em outras situações será útil saber quantas constantes estão armazenadas em uma lista, independentemente de quantos elementos ela possui. Neste caso, a medida para o tamanho do termo será o número de constantes armazenados na lista. Sendo assim, é vital para determinação do tamanho de um termo, o estabelecimento da aplicação que será dada para esta informação.

Uma das principais aplicações para a medida do tamanho de termos na programação em lógica é a análise de complexidade. Conforme afirma Lin ([LIN93]), a análise de complexidade consiste na inferência do montante de recursos computacionais consumidos durante a execução de um programa, por exemplo, o tempo de ocupação do processador. A análise de complexidade no âmbito da presente proposta será discutida no capítulo 6. No caso desta aplicação, sabe-se que o tempo para execução de um procedimento na programação em lógica é função do tamanho dos argumentos de entrada (termos), pois estes determinam o número de chamadas recursivas que serão executadas pelo procedimento. Desta forma, a determinação do tamanho destes argumentos é vital para determinação da complexidade do procedimento.

No caso da análise de complexidade, a medida de tamanho adequada para um argumento de entrada pode ser determinada através da análise de tipos das posições dos argumentos ([VER91]). Basicamente, a idéia consiste em analisar os ciclos em **grafos de tipos** para determinar como o procedimento trata recursivamente os argumentos de entrada para construir os argumentos de saída. Através desta análise, pode-se determinar a medida de tamanho adequada dos argumentos de entrada para a análise de complexidade. Em [LIN93] é apresentado um exemplo de grafo de tipos e como a análise destes grafos pode ser utilizada para estabelecer a medida adequada para os argumentos de entrada. Além disso, encontra-se em [DEB90] e [DEB93] considerações a respeito de medidas de tamanho de termos aplicadas à análise de complexidade e análise de granulosidade.

4.5.2 Medidas de Tamanho de Termos no GRANLOG

O GRANLOG utiliza as mesmas medidas de tamanho apresentadas em [DEB93] e [LIN93]. Estas medidas são as seguintes: valor de um inteiro (*int*), tamanho de lista (*length*), número de constantes de um termo (*size*), profundidade de um estrutura (*depth*) e irrelevante (*void*). As medidas de tamanho no GRANLOG possuem duas aplicações, ou seja, **Determinação da Granulosidade (DGR)** e **Determinação do Tamanho das Saídas (DTS)**. A determinação da granulosidade permite a previsão do tamanho dos grãos (granulosidade) através da análise de complexidade. A segunda aplicação permite a previsão do tamanho das saídas de um procedimento em função do

tamanho de suas entradas. Esta informação pode ser utilizada para prever os custos de comunicação envolvidos na paralelização de um grão e para realizar simulações na execução de programas. As medidas de tamanho no GRANLOG sempre são determinadas através da consideração dessas duas aplicações.

Na versão atual do GRANLOG o usuário deve introduzir as medidas de tamanho dos argumentos através de declarações. No entanto, as medidas utilizadas no GRANLOG podem ser inferidas através de análise automática, conforme descrito em [LIN93]. A análise automática de medidas utiliza as mesmas técnicas aplicadas na inferência automática de tipos. Além disso, cada procedimento do programa em lógica deve obrigatoriamente ter apenas uma declaração de medidas.

A sintaxe da declaração de medidas utilizada pelo GRANLOG é a seguinte:

```
:- measure(p/n,lista_de_medidas).
```

Na declaração de medidas, *p/n* é o identificador do procedimento, ou seja, nome/aridade e *lista_de_medidas* é uma lista contendo em ordem as medidas a serem utilizadas para mensurar o tamanho dos argumentos do procedimento.

A seguir são apresentadas as notações para medidas de tamanho de termos utilizadas no GRANLOG, acompanhadas de um conjunto de exemplos.

- Inteiro: **int**

A medida *int* é utilizada para argumentos do tipo inteiro que influenciam na complexidade e/ou no tamanho das saídas do procedimento. Por exemplo, considere o procedimento para o cálculo do fatorial, apresentado na figura 4.4.

```
fact(0,1).
fact(N,M) :-
      N > 0,
      N1 is N-1, fact(N1,M1), M is N * M1.
```

FIGURA 4.4 - Procedimento *Fatorial*

Considere ainda que as seguintes declarações de modos e tipos foram atribuídas ao procedimento:

```
:- mode(fact/2,[i,o]).
:- type(fact/2,[int]).
```

Neste caso, a complexidade do procedimento depende do tamanho do primeiro argumento (entrada), o qual determina o número de chamadas recursivas que serão realizadas. Além disso, o primeiro argumento determina o tamanho do segundo (saída). A relação entre o tamanho das entradas e o tamanho das saídas depende das medidas a serem utilizadas na representação das saídas. Sendo assim, deve-se estabelecer qual

medida será utilizada para representar o segundo argumento do procedimento *Fact*. Deve-se considerar ainda que os dois argumentos são do tipo inteiro. Portanto, os dois argumentos devem ser mensurados através da medida *int*. Resumindo, os dois argumentos são relevantes para a Determinação da Granulosidade (DGR) e para a Determinação do Tamanho das Saídas (DTS), são do tipo inteiro e portanto a seguinte declaração de medida deve ser utilizada:

```
:- measure(fact/2,[int]).
```

O mesmo resultado seria obtido através da análise do procedimento *fibonacci* apresentado na figura 4.3, considerando-se o primeiro argumento como entrada e o segundo como saída.

- Número de elementos de uma lista: **length**

A medida *length* é utilizada para argumentos do tipo lista que influenciam na complexidade e/ou no tamanho das saídas do procedimento. Além disso, esta medida somente deverá ser utilizada quando a característica relevante da lista for o número de elementos. Por exemplo, considere o procedimento para a concatenação de listas (procedimento *append*), apresentado na figura 2.5. Considere ainda que as seguintes declarações de modos e tipos foram atribuídas a esse procedimento:

```
:- mode(append/3,[i,i,o]).
:- type(append/3,[list(?,[int])]).
```

Neste caso, a complexidade do procedimento depende do tamanho do primeiro argumento (entrada), o qual determina o número de chamadas recursivas que serão realizadas. Além disso, os dois primeiros argumentos determinam o tamanho do terceiro (saída). O terceiro argumento é uma saída, portanto deve-se estabelecer qual a medida a ser utilizada para determinar seu tamanho. Deve-se considerar ainda que os três argumentos são do tipo lista e, neste caso, é relevante o número de elementos destas listas. Portanto, os três argumentos devem ser mensurados através da medida *length*. Resumindo, os três argumentos são relevantes para as análises do GRANLOG (DGR e DTS), são do tipo lista e sua característica interessante é o número de elementos. Portanto, a seguinte declaração de medidas deve ser utilizada:

```
:- measure(append/3,[length]).
```

A mesma análise poderia ser realizada para o procedimento *Nrev* apresentado na figura 2.2, considerando-se o primeiro argumento como entrada e o segundo como saída. Neste caso, deve-se considerar ainda a influência do procedimento *append*, localizado no corpo da segunda cláusula do procedimento *Nrev*, o qual participa tanto da composição da complexidade quanto da saída do *Nrev*.

• Número de constantes e símbolos de função num termo: **size**

A medida *size* será utilizada onde for relevante para a análise, o número de constantes e símbolos de função que aparecem no argumento. Por exemplo, em [LIN93] é apresentado um procedimento para a transformação de listas aninhadas em listas lineares (procedimento *flatten*). Este procedimento está reproduzido na figura 4.5.

```
flatten([],[]).
flatten(X,[X]) :- atomic(X), X \== [].
flatten([XXs],Ys) :-
      flatten(X,Ys1), flatten(Xs,Ys2), append(Ys1,Ys2,Ys).
```

FIGURA 4.5 - Procedimento *flatten*

Considere ainda que as seguintes declarações de modo e tipo foram atribuídas a este procedimento:

```
:- mode(flatten/2,[i,o]).
:- type(flatten/2,[list(?,[?])]).
```

Analisando o procedimento *flatten*, constata-se que sua complexidade depende do tamanho do primeiro argumento (entrada), pois este estabelece o número de chamadas recursivas que serão executadas (veja as chamadas de *flatten* no corpo da terceira cláusula). Em complemento, o segundo argumento é uma saída, tornando-se assim relevante sua medida para análise do procedimento. Finalmente, deve-se considerar a característica dos argumentos que será medida, ou seja, qual o aspecto dos argumentos que influencia na análise. O primeiro argumento do procedimento é uma lista. No entanto, neste caso, o número de elementos da lista não é relevante para a análise e sim o número de constantes que estão armazenadas nesta lista. Por exemplo, a lista pode ter três elementos que são listas contendo diversas constantes. No caso do primeiro argumento de *flatten*, não é relevante que sejam três elementos e sim o número de constantes que as três listas armazenam. Portanto, deve-se utilizar a medida *size*. Por sua vez, o segundo argumento (saída) armazena todas as constantes existentes na lista de entrada, alterando a distribuição de forma a não existirem mais listas como elementos na lista de saída (linearização da lista de entrada). Desta forma, a medida relevante para o argumento de saída será o número de elementos, ou seja, a medida *length*. Resumindo, os dois argumentos são relevantes para as análises do GRANLOG (DGR e DTS), são do tipo lista e suas características relevantes são o número de constantes para o primeiro argumento (medida *size*) e o número de elementos da lista no segundo argumento (medida *length*). A seguinte declaração de medidas deve ser utilizada:

```
:- measure(flatten/2,[size,length]).
```

• Profundidade de uma estrutura: **depth(*LISTA_DE_POSIÇÕES*)**

A medida *depth* será usada quando o argumento for uma estrutura e a característica relevante para a análise for a profundidade dos ramos da árvore que representa a estrutura. O parâmetro *LISTA_DE_POSIÇÕES* contém as posições na estrutura, dos argumentos que são relevantes para a análise. A utilização de um exemplo facilita a compreensão desta medida. A figura 4.6 apresenta o procedimento *traverse*.

```
traverse(nil).
traverse(t(X,L,R)) :- traverse(L), traverse(R).
```

FIGURA 4.6 - Procedimento *traverse*

O procedimento *traverse* percorre uma estrutura que representa uma árvore binária. Claramente na figura 4.6, nota-se que o argumento do procedimento influencia na complexidade da execução. Além disso, constata-se que o argumento é do tipo estrutura. Verifica-se ainda, que desta estrutura, os dois últimos argumentos deteminam a complexidade das chamadas no corpo da segunda cláusula (chamadas recursivas). Sendo assim, a análise deste procedimento depende destes dois argumentos. Através desta análise pode-se gerar as seguintes anotações de modos, tipos e medidas para o procedimento *traverse*:

```
:- mode(traverse/1,[i]).
:- type(traverse/1,[struct(1,3,[?,r,r])]).
:- measure(traverse/1,[depth([2,3])]).
```

A declaração *measure* pode ser lida da seguinte forma: são relevantes para a análise, a profundidade do segundo e do terceiro argumento da estrutura que será enviada para o procedimento *traverse*.

• Irrelevante para a análise: **void**

A medida *void* determina que o argumento não é relevante para a análise, ou seja, o argumento não participa na complexidade de execução do procedimento e nem influencia no tamanho de suas saídas. A medida *void* pode ser utilizada em argumentos de qualquer tipo. O exemplo completo apresentado na seção 4.7 demonstra o uso dessa medida.

Deve-se ressaltar ainda, que de forma semelhante ao tratamento de tipos, pode-se realizar a **adaptação da declaração a chamadas** nas medidas de tamanho de argumentos. Por exemplo, a declaração apresentada para o procedimento *append* pode ser adaptada à realidade de cada chamada, onde o tamanho das listas de entrada pode ser conhecido. Se o sistema encontrar uma chamada *append(Lista,[1,2,3,4,5],Lista_res)*, pode adaptar o argumento *lista_de_medidas* para *[length,void,length]*, pois sendo o segundo argumento uma constante, torna-se supérflua sua medida e o tamanho deste argumento pode ser incorporado na anotação durante a análise estática.

4.6 Análise de Dependências entre Literais

Esta seção discute a análise de dependências na programação em lógica (subseção 4.6.1) e descreve sua realização no modelo GRANLOG (subseção 4.6.2).

4.6.1 Análise de Dependências na Programação em Lógica

No decorrer dos últimos anos, vários trabalhos de pesquisa têm investigado a análise de dependências nos diversos paradigmas de programação. Esta análise determina as dependências existentes entre partes do programa. Desta forma, a análise de dependências estabelece a ordem obrigatória de execução destas partes. Basicamente, existem dois tipos de dependências ([PAD86], [GAL91], [BAR93], [BAR94]), ou seja, **dependência de dados** e **dependência de controle**. A dependência de dados está relacionada com a seqüência obrigatória na execução do programa para obtenção de um fluxo de dados correto, ou seja, representa a relação entre os dados manipulados pelo programa. Por sua vez, a dependência de controle está relacionada com a seqüência obrigatória na execução do fluxo de controle, ou seja, descreve a ordem correta de execução de um programa em função de suas estruturas de controle.

A análise de dependências possui diversas aplicações no universo da computação. No entanto, dentre as suas principais aplicações encontra-se a paralelização de programas. Uma importante etapa em diversas metodologias para exploração do paralelismo consiste na obtenção de grafos que representem as dependências ([KRU88], [MCC89], [SHI90], [GIR92], [GER93], [BAR94]). Normalmente, esta representação recebe o nome de **grafo de tarefas**. Conforme pode ser constatado em [BAR94], os grafos de tarefas são a base para o particionamento dos programas.

Nas pesquisas relacionadas com a programação em lógica, a análise de dependências vêm ocupando uma posição de destaque. Dentre os principais trabalhos nesta área destacam-se os textos [CHA85], [CHA85a] e [DEB89]. Estes trabalhos demonstram a utilização da análise de dependências na paralelização de programas em lógica e no aperfeiçoamento do *backtracking*. Em [DEB89] encontra-se um histórico da evolução dos estudos sobre análise de dependências na programação em lógica. Merecem destaque ainda, os trabalhos sobre análise de dependências desenvolvidos no âmbito do projeto &-Prolog ([HER90], [HER91], [BUE93]). Em [LIN93] a análise de dependências é utilizada para implementação da análise de complexidade nos programas em lógica.

Deve-se ressaltar ainda, que a análise de dependências na programação em lógica difere da análise realizada nas linguagens imperativas. Na programação em lógica, devido ao escopo de cláusula da variável lógica (veja subseção 2.2.6), a análise de dependências sempre é realizada individualmente em cada cláusula, ou seja, não existem dependências entre cláusulas do mesmo procedimento. Esta característica facilita a análise de dependências e estimula a exploração do paralelismo na programação em lógica. Apesar do escopo das variáveis lógicas ser restrito a uma cláusula, a análise de dependências deve ser baseada numa análise global do programa. Este fato está relacionado com a propagação do compartilhamento de variáveis durante a execução do programa em lógica, conforme demonstrado em [CHA85] e [DEB89].

Outra diferença interessante entre a programação imperativa e a programação em lógica consiste no tipo de dependências que podem ser criadas. As linguagens de

programação em lógica puras não contém estruturas de controle, não permitindo assim a criação de dependências de controle. Desta forma, na programação em lógica existe apenas dependência de dados. Mais uma vez, as características da programação em lógica estimulam a exploração do paralelismo. Por outro lado, a base das linguagens imperativas é o controle explícito da execução através de estruturas de programação. O uso destas estruturas de controle cria diversas dependências de controle. Desta forma, a programação imperativa possui dependências de dados (algumas bastante complicadas, tais como as criadas pelas variáveis globais) e estimula a criação de dependências de controle. Mais uma vez, é dificultada a exploração do paralelismo no paradigma de programação convencional.

4.6.2 Análise de Dependências no GRANLOG

O GRANLOG analisa as dependências de dados existentes entre os literais de cada uma das cláusulas de um programa em lógica. Como resultado desta análise, o GRANLOG cria para cada cláusula do programa um **grafo de dependências** onde estão explicitadas as dependências de dados. Estes grafos podem ser utilizados em qualquer aplicação onde as dependências sejam relevantes. No GRANLOG, as dependências de dados são utilizadas para determinar os grãos existentes nas cláusulas. Esta tarefa é realizada pelo módulo Analisador de Grãos apresentado no próximo capítulo.

A análise de dependências realizada pelo GRANLOG é baseada em duas fontes de informação, ou seja, na **estrutura das cláusulas** e nos **modos dos procedimentos.** A primeira fonte considera os literais no corpo das cláusulas, seus argumentos e as variáveis existentes em cada argumento. Estas informações são obtidas através da análise do programa. A segunda fonte considera os modos do procedimento ao qual a cláusula pertence e os modos de todos os procedimentos chamados no corpo da cláusula. Estas informações podem se obtidas através de interpretação abstrata ou através de declarações de modos (veja seção 4.3.1).

A figura 4.7 apresenta o **Algoritmo para Análise de Dependências (AAD)** proposto pelo modelo GRANLOG. Este algoritmo possui duas entradas, ou seja, a cláusula do programa e os modos dos procedimentos envolvidos na codificação da cláusula. Além disso, o algoritmo possui apenas uma saída, ou seja, o grafo de dependências. Basicamente, pode-se dividir o algoritmo em três partes: tratamento das entradas na cabeça da cláusula (TE), tratamento dos literais do corpo da cláusula (TL) e tratamento das saídas na cabeça da cláusula (TS). Estes tratamentos são realizados por três laços consecutivos, os quais podem ser visualizados na figura 4.7. Na verdade, pode-se considerar o AAD como uma linha de produção de grafos de dependências. Esta linha de produção possui três estágios, onde cada estágio compõe uma parte do grafo.

O algoritmo apresentado na figura 4.7 utiliza uma **Tabela de Variáveis** (TABV) para criação do grafo de dependências. Esta tabela contém três campos onde são armazenadas informações sobre as variáveis encontradas na cláusula em análise. A figura 4.8 mostra a organização da TABV. O primeiro campo armazena o nome da variável. No segundo campo é armazenado um número equivalente ao literal onde a variável foi fechada, ou seja, completamente instanciada. A cabeça possui número 0 e os literais do corpo da cláusula são numerados seqüencialmente da esquerda para a direita. O terceiro e último campo de TABV armazena um valor booleano que indica se a variável possui potencial de compartilhamento (PC). O potencial de compartilhamento é utilizado para

introduzir na análise de dependências, considerações a respeito de possíveis conflitos de ligação. Uma variável com PC permite o conflito de ligação com todas as demais variáveis que também possuam PC.

ENTRADAS: Cláusula e Modos dos procedimentos
SAÍDA: Grafo de dependências
ABREVIATURAS: GD = Grafo de Dependências
TABV = Tabela de Variáveis
PC = Potencial de Compartilhamento
MÉTODO:

```
início
   Adicionar em GD um vértice para cada literal;
TE
   para cada argumento da cabeça faça
     se modo do argumento então
       i     : Adicionar em TABV variáveis do argumento;
       io, ? : Adicionar em TABV variáveis do argumento com PC = TRUE;
     fimse
   fimpara
TL
   para cada literal L do corpo da cláusula faça
     para cada argumento faça
       se modo do argumento então
         i : para cada variável V no argumento faça
               Adicionar arco em GD de TABV[V].literal para L;
             fimpara
         o : Adicionar em TABV variáveis do argumento;
         io,? :
           para cada variável V no argumento faça
             se V existe na tabela então
               Adicionar arco em GD de TABV[V].literal para L;
               se ((TABV[V].literal = 0) e TABV[V].PC então
                 para todos elementos de TABV faça
                   se ((literal <> 0) e PC então
                     Alterar TABV.literal para L;
                   fimse
                 fimpara
               fimse
             senão
               Adicionar V em TABV;
             fimse
           fimpara
       fimse
     fimpara
   fimpara
TS
   para cada argumento da cabeça faça
     se modo do argumento = i então
       para cada variavel V do argumento faça
         se TABV[V].literal <> 0 então
           Adicionar arco em GD de TABV[V].literal para cabeça;
         fimse
       fimpara
     fimse
   fimpara
fim
```

FIGURA 4.7 - Algoritmo para análise de dependências

	Variável	Literal	PC
1			
	.	.	.
	.	.	.
	.	.	.
n			

FIGURA 4.8 - Tabela de variáveis

Os seguintes pontos devem ser considerados para compreensão do algoritmo de análise de dependências:

- **Modos definidos**: O GRANLOG considera apenas procedimentos com modos definidos, ou seja, após uma chamada, todos os argumentos, com excessão dos que possuam modo de entrada (modo *i*), devem retornar fechados;
- **Tratamento dos *Builtins***: Durante a análise de dependências os *builtins* são tratados da mesma forma que os demais literais. No entanto, cada *builtin* possui uma especificação de modos que permite sua análise. Por exemplo, o *builtin IS* é tratado como possuindo dois argumentos, uma entrada (expressão) e uma saída (resultado). Os *builtins* para operações relacionais (<,>,>=, etc) são tratados como possuindo duas entradas. O exemplo apresentado na seção 4.7 demonstra o resultado da análise de dependências envolvendo *builtins*;
- **Análise local**: O algoritmo realiza uma avaliação local de dependências, ou seja, a análise restringe-se ao escopo da cláusula. Desta forma, as únicas informações sobre o estado das variáveis na cabeça antes da execução são obtidas através dos modos. A utilização do conceito de potencial de compartilhamento (PC) permite uma análise conservativa para evitar possíveis conflitos de ligação durante a execução. O uso da interpretação abstrata soluciona este problema, permitindo através da análise global uma apurada determinação das dependências;
- **Corretude nos modos**: O algoritmo considera que os modos dos procedimentos estão corretos, ou seja, não existem incompatibilidades entre os modos dos diversos procedimentos envolvidos na análise. Por exemplo, se uma variável aparece na cabeça num argumento que possua modo *i*, não poderá aparecer numa chamada no corpo em um argumento que seja *o*. A tabela apresentada na figura 4.9 mostra as regras básicas que evitam incompatibilidade entre modos da cabeça e do corpo da cláusula. A tabela deve ser lida da seguinte forma: se uma variável aparece na cabeça da cláusula em um argumento que possua o modo apresentado na primeira coluna, somente poderá aparecer no corpo da cláusula em argumentos que possuam os modos apresentados na segunda coluna. Além disso, devem ser evitados erros de modos no corpo da cláusula, por exemplo, uma variável não pode aparecer em argumentos de saída de chamadas diferentes, pois obviamente, na segunda chamada a variável estará fechada (modos definidos) e o modo necessariamente será entrada.

A análise de dependências proposta pelo GRANLOG é conservativa, ou seja, sempre que for detectada a **possibilidade de dependência** será considerada sua existência. Esta conduta, quando aplicada a uma análise local, gera um excesso de dependências, mas no entanto garante a corretude das informações obtidas. Um aperfeiçoamento indispensável para a evolução desta proposta é a utilização da interpretação abstrata. Esta técnica permite a obtenção de maior precisão, melhorando substancialmente os resultados da análise. Em [LIN93] é apresentado um algoritmo semelhante ao proposto pelo GRANLOG. O algoritmo de Nai-Wei Lin é mais simples devido a utilização de apenas dois modos na sua proposta de análise de complexidade. Outros textos apresentam análises de dependências mais complexas e completas do que a algoritmo proposto neste trabalho. Em [DEB89] a análise de dependências é realizada através de uma sofisticada interpretação abstrata. Em [CHA85] e [CHA85a] a análise de

dependências baseia-se numa análise global do programa através da propagação de informações de instanciação.

CABEÇA	CORPO
i	i, io, ?
o	o, io, ?
io	io, ?
?	?

FIGURA 4.9 - Regras de modos

Finalmente, deve-se ressaltar que o restante do modelo está baseado nos grafos de dependências obtidos pelo algoritmo apresentado na figura 4.7. Sendo assim, quanto maior a precisão destes grafos, maior a precisão do restante da análise de granulosidade. Além disso, é relevante a modularidade implícita no modelo proposto, ou seja, o restante do modelo não considera como os grafos de dependência são obtidos. O aperfeiçoamento destes grafos é transparente para os demais módulos e desta forma a melhoria da análise de dependências é automaticamente propagada para o restante da análise.

4.7 Exemplo de Análise Global: Programa *Torre de Hanói*

O programa apresentado na figura figura 4.10 será utilizado para exemplificar o processo de análise global. O programa implementa o jogo *Torre de Hanói* e possui dois procedimentos, ou seja, procedimento *hanoi* e procedimento *append*. O procedimento *hanoi* possui cinco argumentos. O primeiro argumento determina o número de peças que deverão ser utilizadas no jogo. Os próximos três argumentos contém os nomes dos três pinos que deverão ser utilizados na apresentação dos resultados. O quinto e último argumento conterá o resultado do jogo. Este resultado é armazenado numa lista onde estão explicitados os movimentos das peças.

```
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
      N > 1, N1 is N - 1,
      hanoi(N1,A,C,B,M1), hanoi(N1,B,A,C,M2),
      append(M1,[mv(A,C)],T), append(T,M2,M).

append([],L,L).
append([HL],L1,[HR]) :- append(L,L1,R).
```

FIGURA 4.10 - Programa *Torre de Hanói*

As **anotações de modos** para os dois procedimentos poderiam ser:

```
:- mode(hanoi/5,[i,i,i,i,o]).

:- mode(append/3,[i,i,o]).
```

A primeira anotação indica que os quatro primeiros argumentos do procedimento *hanoi* serão entradas e o quinto e último argumento será saída. A segunda anotação indica que os dois primeiros argumentos do procedimento *append* atuarão como entrada e o último servirá como saída. Desta forma, os canais de comunicação relacionados com a execução paralela do procedimento *hanoi* ficam configurados conforme mostra a figura 4.11. Os canais de comunicação para o procedimento *append* são mostrados na figura 4.12.

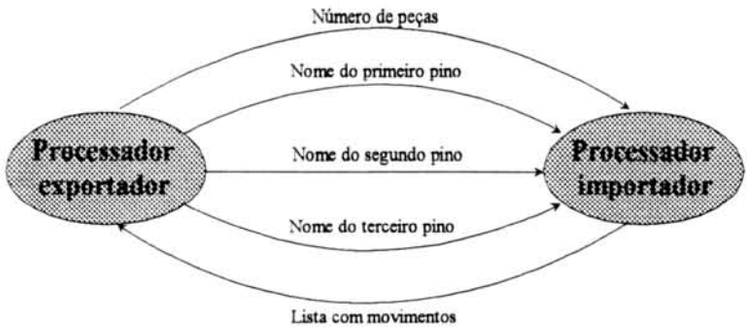


FIGURA 4.11 - Canais de comunicação para procedimento *hanoi*

FIGURA 4.12 - Canais de comunicação para o procedimento *append*

As **anotações de tipos** para os dois procedimentos poderiam ser as seguintes:

```
:- type(hanoi/5,[int,atom(?),atom(?),atom(?),list(?,[struct(2,2,[atom(?)])])]).

:- type(append/3,[list(?,[struct(2,2,[atom(?)])])]).
```

A primeira anotação contém os tipos dos argumentos de *hanoi*. Conforme indica a anotação, o primeiro argumento é um inteiro, os próximos três são átomos de tamanho desconhecido e o último (saída) é uma lista de estruturas. O tamanho da lista é desconhecido antes da execução. No entanto, sabe-se que o nome das estruturas armazenadas na lista possui tamanho dois e existem dois argumentos em cada estrutura. Sabe-se ainda, que estes argumentos são átomos de tamanho desconhecido. A segunda anotação indica que todos os argumentos do procedimento *append* são listas de tamanho desconhecido contendo estruturas com nomes de tamanho dois e com dois argumentos que são átomos de tamanho desconhecido. Se o usuário estabelecer um tamanho fixo para o nome dos pinos, a anotação de tipos torna-se mais esclarecedora. Por exemplo, o usuário pode determinar que os pinos terão nomes com três caracteres. Sendo assim, as anotações de tipo seriam as seguintes:

```
:- type(hanoi/5,[int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).

:- type(append/3,[list(?,[struct(2,2,[atom(3)])])]).
```

Nesta anotação, o símbolo de indefinição (?) é utilizado apenas onde realmente não pode haver previsão de valores, como é o caso do tamanho de saída da lista contendo os resultados de *hanoi* e o tamanho das listas do procedimento *append*. A figura 4.13 apresenta como ficaria a configuração dos canais de comunicação para a execução paralela do procedimento *hanoi* com o conhecimento dos tipos dos argumentos. A figura 4.14 mostra os canais de comunicação do procedimento *append*.

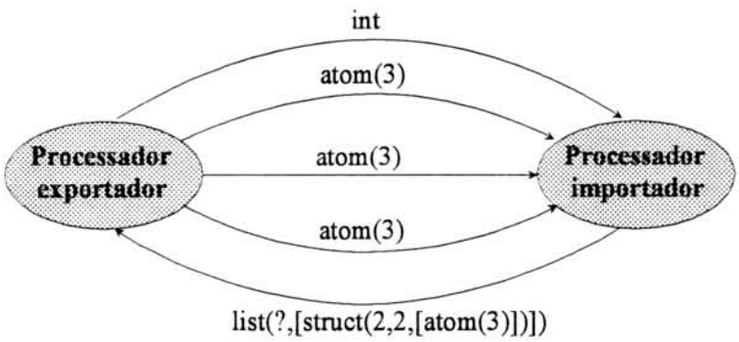


FIGURA 4.13 - Canais de comunicação com tipos para procedimento *hanoi*

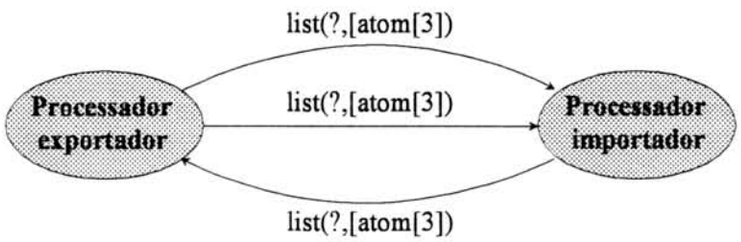


FIGURA 4.14 - Canais de comunicação com tipos para procedimento *append*

As **anotações de medidas** para os procedimentos do exemplo seriam as seguintes:

```
:- measure(hanoi/5,[int,void,void,void,length]).
:- measure(append/3,[length]).
```

A primeira anotação indica que o primeiro e o último argumento de *hanoi* são relevantes para a análise do GRANLOG (aplicações DGR e DTS, veja subseção 4.4.2). Além disso, está indicado na anotação que o segundo, terceiro e quarto argumentos não interferem nesta análise. Intuitivamente, percebe-se que o primeiro argumento (número de peças) interfere na complexidade da execução do procedimento (aplicação DGR, veja subseção 4.4.2). Analisando-se o programa, constata-se que o número de peças determina o número de chamadas recursivas de *hanoi* no corpo da segunda cláusula. Desta forma, a medida do primeiro argumento é indispensável. Em complemento, o

último argumento é uma saída e deve-se saber sua medida para determinar sua relação com a entrada (aplicação DTS, veja subseção 4.4.2). Os nomes dos pinos (segundo, terceiro e quarto argumentos) não interferem na complexidade do procedimento e não influenciam no número de elementos da lista de saída. Por sua vez, a segunda anotação indica que os três argumentos de *append* são relevantes para a análise realizada pelo GRANLOG, pois possuem uma medida. Conforme mostrado na anotação, os três argumentos de *append* possuem a medida *length*, ou seja, o número de elementos de uma lista. O primeiro argumento influencia na complexidade do procedimento *append*, pois determina o número de chamadas recursivas que serão executadas no corpo da segunda cláusula. O segundo argumento influencia no tamanho da saída e portanto também deve ser levado em consideração. Finalmente, o terceiro argumento é a saída e portanto deve ser estabelecido qual sua medida para obtenção da relação entre entradas e saída (aplicação DTS, veja subseção 4.4.2).

A análise de dependências dos procedimentos da figura 4.10, realizado pelo algoritmo apresentado na figura 4.7, resultaria nos grafos de dependência mostrados nas figuras 4.15 e 4.16. A figura 4.15 mostra os grafos de dependência do procedimento *hanoi*. Na figura 4.16 são apresentados os grafos de dependência das cláusulas do procedimento *append*. Os nodos I (*input*) e O (*output*) representam as entradas e saídas da cláusula, mostrando as dependências entre a cabeça e o corpo. Os demais literais são representados por números de acordo com sua posição no corpo da cláusula.

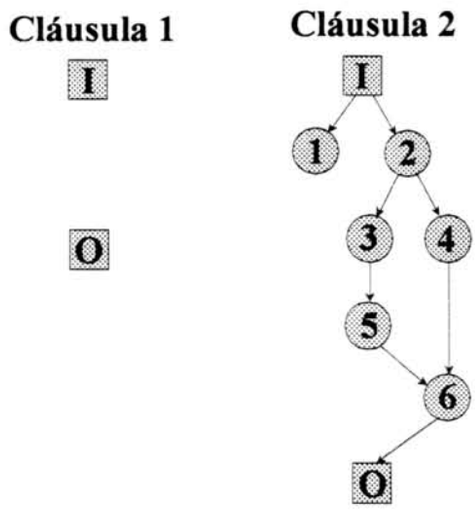


FIGURA 4.15 - Grafos de dependência do procedimento *hanoi*

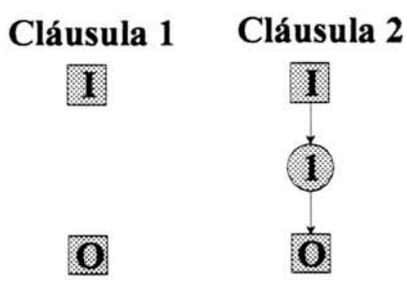


FIGURA 4.16 - Grafos de dependência do procedimento *append*

4.8 Conclusões

Neste capítulo foi apresentado o primeiro módulo do modelo GRANLOG, ou seja, o Analisador Global. Durante sua apresentação, foram introduzidos diversos conceitos e citadas diversas referências sobre vários tópicos de pesquisa da programação em lógica, tais como: análise de modos, análise de tipos, análise de medidas, interpretação abstrata e análise de dependências. Além disso, foram descritos os modos, tipos e medidas utilizadas no GRANLOG, bem como o algoritmo para análise de dependências proposto neste trabalho.

Destacam-se como principais conclusões deste capítulo:

- a inclusão da interpretação abstrata no AGL é um passo importante para evolução do modelo GRANLOG;
- apesar das declarações atuais não suportarem múltiplos modos e múltiplos tipos, a anotação proposta é flexível o bastante para suportar esta evolucão;
- o algoritmo para análise de dependências realiza uma análise local e portanto assume uma posição conservativa para o compartilhamento de variáveis;
- o aperfeiçoamento da análise de dependências através da interpretação abstrata refletirá na precisão do restante da análise de granulosidade.

O próximo capítulo apresenta o módulo Analisador de Grãos (AGR). Este módulo utiliza as informações obtidas no módulo AGL para introduzir a anotação de grãos no programa em lógica, gerando assim o programa particionado.

5 Análise de Grãos

Este capítulo apresenta o segundo estágio do modelo proposto, ou seja, a análise de grãos. No GRANLOG, a análise de grãos é implementada pelo módulo Analisador de Grãos (AGR). Na seção 5.1 encontra-se uma introdução à análise de grãos, onde são discutidos diversos conceitos e trabalhos sobre este assunto. A seção 5.2 descreve o módulo Analisador de Grãos. A seção 5.3 apresenta os tipos de grãos propostos pelo GRANLOG. Por sua vez, a seção 5.4 descreve como é realizada a determinação de grãos no modelo proposto. Nessa seção, destaca-se o algoritmo para inferência de grãos. Na seção 5.5 é apresentada a anotação de grãos gerada pelo módulo AGR. A seção 5.4 demonstra a análise de grãos para o programa *Torre de Hanói*, dando continuidade ao exemplo iniciado no capítulo anterior (seção 4.7). Finalmente, a seção 5.5 apresenta as conclusões deste capítulo.

5.1 Introdução

A **análise de grãos** consiste na identificação dos grãos, ou seja, na determinação dos módulos que deverão ser executados seqüencialmente num único processador. Deve-se diferenciar a análise de grãos da análise de granulosidade. Todo particionamento determina os grãos, portanto todo particionamento é uma análise de grãos. Por outro lado, conforme discutido na subseção 2.1.2, a análise de granulosidade enfatiza a máxima eficiência no particionamento, devendo para isso, considerar aspectos sobre complexidade dos grãos e custos de paralelização. Sendo assim, pode-se afirmar que a análise de granulosidade é uma análise de grãos visando a máxima eficiência. Portanto, toda análise de granulosidade é uma análise de grãos, mas nem toda análise de grãos é uma análise de granulosidade.

Normalmente, as metodologias para exploração de paralelismo no paradigma de programação convencional consideram aspectos relacionados com os custos de paralelização e complexidade dos módulos ([KRU88], [SAR89], [GIR92]). Sendo assim, estas metodologias enquadram-se na definição de análise de granulosidade. No entanto, várias metodologias para exploração do paralelismo na programação em lógica não consideram os aspectos **custo** e **complexidade** e portanto, realizam apenas análise de grãos. Esta interessante característica do paralelismo na programação em lógica deve-se em grande parte à facilidade para determinação dos possíveis grãos num programa em lógica. Basicamente, todas as cláusulas podem ser executadas em paralelo (paralelismo OU). Além disso, as metas do corpo de uma cláusula também podem ser executadas em paralelo (paralelismo E), devendo-se para isso apenas evitar os conflitos de ligação. Por outro lado, nos programas imperativos, o particionamento é mais complexo e normalmente realiza a agregação de partes do programa para formação dos grãos. A agregação exige considerações sobre custo e complexidade. No entanto, recentemente, os pesquisadores do paralelismo na programação em lógica descobriram a importância das informações de custo e complexidade para obtenção da máxima eficiência na execução paralela de programas em lógica. Desta forma, surgiram propostas para dimensionamento do tamanho adequado dos grãos através da análise de granulosidade. Como exemplo, pode-se citar a evolução do projeto &-Prolog. Inicialmente, os esforços de pesquisa deste projeto foram direcionados para construção de um sistema para exploração do paralelismo na programação em lógica, sem preocupações com os aspectos custo e complexidade. Obtiveram assim, um sistema que explora paralelismo E,

particionando o corpo das cláusulas através de análises de dependências. Portanto, o sistema realizava apenas análise de grãos. Recentemente, parte dos esforços de pesquisa deste grupo tem sido direcionados para obtenção da máxima eficiência na execução paralela de programas em lógica. Para alcançar este objetivo, vários estudos estão sendo realizados, dentre os quais destacam-se os relacionados com análise de complexidade ([DEB93], [DEB94a]), análise de granulosidade ([DEB90], [GAR94]) e interpretação abstrata ([DEB94]).

Conforme descrito no último parágrafo da subseção 2.1.2, a análise de granulosidade pode ser realizada de forma estática, dinâmica ou combinada. Da mesma forma, a análise de grãos na programação em lógica (análise sem considerações sobre custo e complexidade) pode ser realizada em tempo de compilação, execução ou em ambos. Basicamente, na **análise estática** determina-se os grãos em tempo de compilação através da análise de dependências entre os literais do corpo de cada uma das cláusulas. Esta opção não introduz custo computacional (*overhead*) na execução, mas impõe uma tomada de decisões conservativa, pois ainda não são conhecidos os verdadeiros estados de instanciação das variáveis. Um exemplo clássico deste tipo de metodologia é a proposta de Chang ([CHA85]). Por sua vez, a **análise dinâmica** determina os grãos durante a execução. Esta opção é precisa, pois considera os reais aspectos da execução, mas no entanto, introduz um alto custo na execução. A proposta de Conery ([CON85] realiza uma análise dinâmica. A combinação da análise estática e dinâmica, resulta na **análise combinada**. Nesta opção, são realizadas análises durante a compilação, mas a decisão de particionamento é postergada para a execução. Destaca-se como metodologia para análise combinada o paralelismo E restrito proposto por Degroot ([DEB84], [DEG87]).

No escopo do projeto &-Prolog foram desenvolvidos três algoritmos para análise de grãos na programação em lógica, ou seja, os algoritmos CDG, UDG e MEL ([MUT90]). Basicamente, estes algoritmos consideram as dependências existentes entre os literais das cláusulas e anotam no programa as possibilidades de particionamento. Os algoritmos CDG e MEL realizam uma análise combinada, anotando no programa um conjunto de condições que podem ser testadas em tempo de execução para tomada de decisões sobre o particionamento do programa. Por sua vez, o algoritmo UDG realiza uma análise estática e anota no programa todas as possibilidades de particionamento, sem introduzir condições a serem testadas durante a execução. As condições anotadas pelos algoritmos CDG e MEL postergam para execução as decisões de particionamento, visando explorar ao máximo o paralelismo. Durante a execução, o estado de instanciação das variáveis é conhecido e portanto, o particionamento pode ser mais preciso. Por outro lado, o algoritmo UDG não utiliza condições, devendo assim, realizar uma anotação conservativa de particionamento. Esta anotação conservativa pode ocasionar perda de paralelismo. Deve-se destacar ainda, que a base do particionamento realizado pelos três algoritmos é a análise de dependências, a qual determina quais objetivos podem ser executados em paralelo. Quanto mais precisas forem as informações de dependências, mais precisas serão as anotações feitas pelos três algoritmos.

Conforme discutido no capítulo 4 (seção 4.1), a análise de um programa em lógica pode ser local ou global. A análise local considera apenas informações obtidas no escopo de uma cláusula. A análise global considera informações obtidas através da análise de todo o programa. O capítulo 4 ressalta ainda, que uma das principais informações que podem ser obtidas através da análise de programas em lógica são as dependências existentes entre os diversos objetivos de uma cláusula. Além disso, sabe-se que a

precisão das informações de dependência é função do tipo de análise realizada para sua obtenção e que através da análise global obtem-se informações mais precisas. Sendo assim, o tipo de análise influencia diretamente na precisão das informações de dependências e conseqüentemente na conduta dos algoritmos CDG, MEL e UDG. Conforme constata-se no texto [BUE94], quando for realizada uma análise global apurada, as anotações geradas pelo algoritmo UDG são tão eficazes quanto as anotações geradas pelos algoritmos CDG e MEL. Sendo assim, elimina-se a única desvantagem do algoritmo UDG, ou seja, a anotação conservativa de paralelismo (perda de paralelismo). Esta conclusão torna-se importante na medida em que o algoritmo UDG não anota condições a serem testadas em tempo de execução, evitando-se assim custo adicional na execução e diminuindo-se a complexidade da anotação. O algoritmo UDG é a base da determinação de grãos descrita na seção 5.4.

5.2 Módulo Analisador de Grãos

A figura 5.1 mostra uma visão genérica do módulo Analisador de Grãos. Nesta figura constata-se que o AGR possui como entradas as informações inferidas no módulo AGL e o programa em lógica. Baseado nas informações de dependências, o AGR realiza uma análise de grãos, onde são determinados os possíveis grãos existentes no programa e suas entradas e saídas. Além disso, o AGR determina os tipos e as medidas das entradas e saídas de cada grão. Desta análise, resulta uma **anotação de grãos** onde estão explicitados os **grãos em potencial** do programa e as informações a respeito destes grãos. A anotação de grãos será apresentada em detalhes no restante deste capítulo. Um grão em potencial é uma parte do programa que possui potencial para tornar-se um grão durante a execução. Um grão em potencial somente será grão, quando em tempo de execução for decidido pela sua paralelização. Sendo assim, a existência de grãos em potencial somente tem sentido na análise combinada de granulosidade, pois na análise estática os grãos são completamente definidos antes da execução. A figura 5.1 mostra ainda, que a saída do módulo AGR é o **programa particionado**. O programa particionado consiste do programa em lógica acrescido da anotação de grãos.

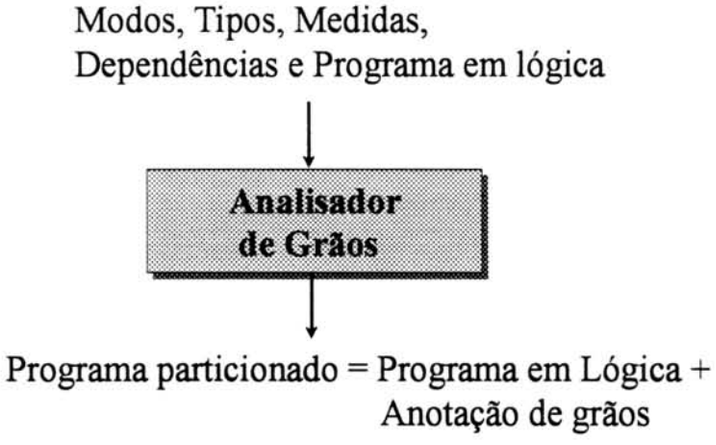


FIGURA 5.1 - Módulo Analisador de Grãos

A figura 5.2 apresenta a estrutura interna do módulo AGR. Esta figura demonstra que o AGR possui internamente dois submódulos, ou seja, o **Determinador de Grãos** (DG) e o **Gerador de Anotação de Grãos** (GAG). O submódulo DG determina os

grãos em potencial existentes em cada uma das cláusulas do programa. Este submódulo será discutido na seção 5.4. O módulo GAG utiliza as informações fornecidas pelo DG e as demais informações fornecidas pelo AGL para gerar a anotação de grãos. A anotação de grãos gerada pelo submódulo GAG será apresentada em detalhes na seção 5.5. Conforme mostra a figura 5.2, a determinação de grãos possui como base as informações de dependências inferidas pelo módulo AGL. Utilizando estas informações, o DG determina as partes do corpo das cláusulas que são independentes e conseqüentemente podem ser executadas em paralelo (grãos em potencial). Para realizar esta tarefa, o submódulo DG utiliza um algoritmo semelhante ao algoritmo UDG desenvolvido no projeto &-Prolog. Sendo assim, o AGR realiza uma análise de grãos estática, ou seja, não são introduzidas condições a serem testadas em tempo de execução para determinar a possibilidade de paralelismo. O potencial de paralelismo fica completamente definido em tempo de compilação. Esta importante decisão de projeto resulta de uma série de observações a respeito da tendências de pesquisa na exploração do paralelismo na programação em lógica. Dentre estas observações, destaca-se a constatação do crescente interesse pela análise global de programas em lógica, a qual permite a determinação precisa de dependências em tempo de compilação.

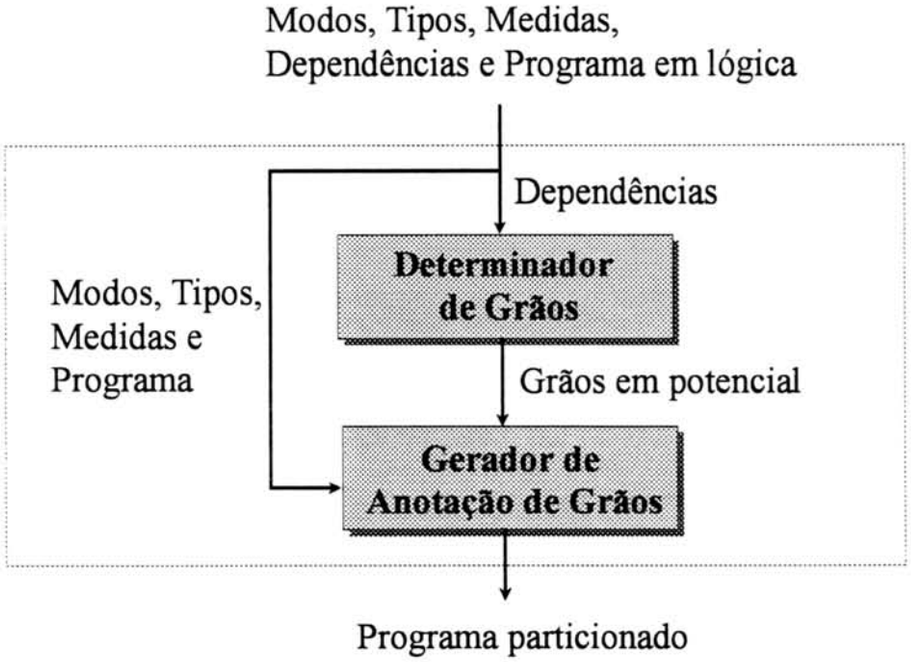


FIGURA 5.2 - Estrutura interna do módulo AGR

Deve-se ressaltar ainda, que o programa particionado, resultante da análise de grãos, contém um conjunto interessante de informações que podem ser utilizadas em diversas aplicações. Esta constatação assume relevância na medida em que o programa particionado torna-se um produto do GRANLOG, o qual pode ser utilizado independentemente do restante da análise. Outra interessante observação, consiste na evolução da análise realizada pelo GRANLOG. O programa particionado resultante do AGR contém todas informações necessárias para execução paralela de programa em lógica. No entanto, não contém informações que permitam a consideração de custos de paralelização e complexidade dos grãos, as quais serão introduzidas pelo próximo módulo do GRANLOG (veja a organização básica do modelo na figura 3.2). Desta

forma, o programa particionado pode ser utilizado para execução paralela de programas em lógica, mas não permite a busca da máxima eficiência, a qual é característica básica da análise de granulosidade.

5.3 Tipos de Grãos Propostos pelo GRANLOG

Conforme discutido na subseção 2.3.1, existem basicamente duas fontes de paralelismo na programação em lógica, ou seja, o paralelismo OU e o paralelismo E. Além disso, o terceiro princípio do GRANLOG (discutido na seção 3.1) estabelece a busca da máxima exploração do paralelismo através de considerações sobre ambas as fontes. Desta forma, considerando as duas fontes de paralelismo e visando a satisfação do terceiro princípio, o GRANLOG propõe a criação de dois tipos de grãos para a programação em lógica, ou seja, o **grão-OU** e o **grão-E**. O grão-OU é utilizado para explorar o paralelismo OU e o grão-E permite a exploração do paralelismo E. Estes dois tipos de grãos possuem características distintas que devem ser discutidas para compreensão da proposta do GRANLOG.

Conforme mostra a figura 2.7, o paralelismo OU explora a execução paralela de caminhos durante a busca de soluções. Sendo assim, define-se **grão-OU** como um grão criado para executar caminhos da árvore de busca. Este tipo de grão surge quando durante a execução de um programa decide-se executar em paralelo cláusulas do mesmo procedimento. A figura 2.7 mostra que um caminho é composto pela cláusula e pelas resolventes que ainda não foram executadas até o momento da criação do caminho, ou seja, até o momento da execução do procedimento que origina o caminho. Na figura 2.7, esta situação é demonstrada pela existência da resolvente **q** nos três caminhos. Para executar em paralelo um grão-OU, é necessário que o processador importador (processador que assume a execução do grão-OU) tenha acesso a todo o contexto de execução do programa até o momento da criação do grão. Em sistemas com memória compartilhada, este acesso pode ser realizado diretamente na memória comum. Neste caso, o custo para paralelização consiste apenas na criação do ambiente para que o grão-OU possa ser executado (criação de processos, inicialização da execução, etc). Em sistemas com memória distribuída, o contexto deve ser enviado através de mensagens. Neste caso, o custo para paralelização envolve ainda o custo para envio destas mensagens. Deve-se ressaltar, que o grão-OU não retorna resultados, ou seja, após a inicialização, o grão-OU torna-se independente e não retorna soluções para seu ponto de origem. Portanto, não existem custos de comunicação com o retorno de soluções.

Atualmente, os sistemas que exploram o paralelismo existente no corpo de uma cláusula (paralelismo E) são baseados na paralelização individual de metas, ou seja, utilizam como grãos apenas metas isoladas. Esta tendência faz com que o paralelismo E seja definido como a **execução paralela de metas do corpo de uma cláusula**. A figura 2.7 mostra o paralelismo E sendo explorado desta forma. No entanto, a máxima exploração do paralelismo existente no corpo de uma cláusula somente poderá ser realizada se os grãos não ficarem limitados a metas isoladas. Através da análise das dependências entre os literais das cláusulas, pode-se determinar quais partes do corpo de uma cláusula poderão ser executadas em paralelo, definindo-se assim vários níveis de grãos. Desta forma, os grãos não serão necessariamente metas isoladas, mas sim partes da cláusula, que poderão coincidentemente ser uma meta. Portanto, no escopo do GRANLOG é necessária uma redefinição do paralelismo E, permitindo assim uma visão mais genérica da exploração de paralelismo no corpo das cláusulas. Para o GRANLOG o

paralelismo E consiste na **execução paralela de partes do corpo de uma cláusula**. Sendo assim, define-se **grão-E** como um grão criado para executar uma parte do corpo de uma cláusula. Para executar em paralelo um grão-E, é necessário que o processador importador (processador que assume a execução do grão-E) tenha acesso apenas aos argumentos de entrada do grão (o restante do contexto não é relevante). Em sistemas com memória compartilhada, estes argumentos podem ser acessados na memória comum. Neste caso, o custo de paralelização consiste da criação do ambiente para execução do grão-E. Em sistemas com memória distribuída, os argumentos de entrada devem ser enviados através de mensagens. Portanto, o custo de paralelização envolve ainda o custo de envio das mensagens. Diferentemente do grão-OU, o grão-E retorna resultados para o processador exportador e portanto, em sistemas distribuídos, existe um custo adicional para envio dos resultados através de mensagens.

Atualmente, o GRANLOG propõe a criação de apenas uma espécie de grão-OU denominada **grão-cláusula**. O grão-cláusula consiste de uma cláusula considerada como um grão (apenas um caminho por grão-OU). Conforme discutido em [GAR94], existe a possibilidade do agrupamento de cláusulas para criação de apenas um grão-OU (no GRANLOG este grão receberia o nome de **grão-cláusulas**). No entanto, esta dissertação não aborda o agrupamento de cláusulas, apesar do autor considerar este tema indispensável para obtenção da máxima eficiência na execução paralela de programas em lógica. O GRANLOG propõe ainda, a criação de duas espécies de grão-E denominadas **grão-meta** e **grão-metas**. O grão-meta consiste de uma meta considerada como grão. Por sua vez, o grão-metas consiste de um conjunto de metas agrupadas para formação de um único grão. O grão-metas permite a concretização da nova definição de paralelismo E proposto pelo GRANLOG. Estes três tipos de grãos originam três anotações (***grain_clause***, ***grain_goal*** e ***grain_goals***), as quais descrevem as características dos grãos em potencial existentes no programa. Estas anotações serão discutidas na seção 5.5.

5.4 Determinação de Grãos

A determinação de grãos identifica os grãos em potencial existentes num programa em lógica. Conforme discutido na seção 5.3, o GRANLOG propõe a criação de dois tipos de grãos, ou seja, grãos-OU e grãos-E. A determinação dos grãos-OU é simples, pois cada cláusula é um grão em potencial (grão-cláusula) que poderá ser executado em paralelo para exploração do paralelismo OU. Portanto, não é necessária nenhuma análise para identificar este tipo de grão. Por outro lado, a determinação dos grãos-E é complexa. Atualmente, o GRANLOG explora o paralelismo E independente (veja subseção 2.3.3). Neste tipo de paralelismo, devem ser consideradas as dependências existentes entre os literais do corpo de cada uma das cláusulas do programa. Esta análise é necessária para evitar conflitos de ligação das variáveis compartilhadas pelos literais a serem executados em paralelo. Portanto, o principal problema a ser resolvido pela identificação de grãos consiste em determinar os possíveis grãos existentes no corpo das cláusulas (paralelismo E). Para solucionar este problema, o GRANLOG utiliza o **Algoritmo para Determinação de Grãos** (ADG) apresentado na figura 5.3. Considerando-se a estrutura interna do módulo AGR apresentado na figura 5.2, o ADG encontra-se no submódulo Determinador de Grãos (DG). Na verdade, o algoritmo ADG é o núcleo do módulo AGR e implementa a maior parte do submódulo DG.

Conforme mostra a figura 5.2, o submódulo DG recebe as informações de dependências e identifica os grãos em potencial existentes no programa. Na subseção 4.6.2 foi apresentado o algoritmo para análise de dependências utilizado no GRANLOG. Além disso, naquela subseção encontra-se a afirmação de que o algoritmo de análise de dependências poderia ser considerado como uma linha de produção de grafos de dependência, onde para cada cláusula do programa gera-se uma grafo que representa as dependências entre seus literais. Da mesma forma, o algoritmo ADG pode ser considerado como um produtor de UGEs (Unconditional Graph Expression). Uma UGE consiste numa representação linear dos grãos existentes no corpo de uma cláusula. Basicamente, uma UGE é uma CGE (Conditional Graph Expression) ([DEG84], [DEG87], [HER90]) sem anotações de condições a serem testadas em tempo de execução. Este tipo de representação linear é apropriada para geração de código para máquinas abstratas (Warren Abstract Machine [AIT91], por exemplo), mas pode ocasionar perda de paralelismo ([DEG84], [MUT90]). Desta forma, a entrada do ADG é um grafo de dependências e sua saída é uma UGE. Resumindo, cada cláusula gera um grafo de dependências (algoritmo AAD) e cada grafo de dependências gera uma UGE (algoritmo ADG). Portanto, cada cláusula possui uma UGE que representa seus grãos em potencial.

ENTRADA: Grafo de Dependências (GD)
SAÍDA: Unconditional Graph Expression (UGE)

MÉTODO:

```
início

❶   para cada Operação relacional livre ou Builtin IS livre faça
      Insere literal em seqüência na UGE;
      Retira literal do GD;
    fim-para

❷   Obter conjuntos P, Q, PP e QQ;

❸   Faça UGE = UGE + Elementos de P em paralelo;

❹   para cada subconjunto SQQ do conjunto QQ faça
      Faça GD_SQQ = Grafo de dependências de SQQ;
      Faça UGE_QQ = Resultado da aplicação do ADG em GD_SQQ;
      Faça UGE = Resultado da integração de UGE com UGE_QQ;
    fim-para

fim
```

FIGURA 5.3 - Algoritmo para determinação dos grãos (ADG)

A lógica utilizada no algoritmo ADG exige que os grafos de dependência a serem analisados tenham certas características que não condizem com os grafos de dependência gerados pelo algoritmo AAD. Portanto, é necessário um tratamento dos grafos produzidos pelo AAD antes da determinação de grãos pelo algoritmo ADG. Duas ações devem ser executadas para adaptação dos grafos, ou seja:

❶ Retirar do grafo de dependência o nodo que representa a cabeça da cláusula;

❷ Aplicar uma Regra de Transitividade para as dependências entre os nodos do grafo.

Baseado nestas considerações, a organização interna do módulo Determinador de Grãos é apresentada na figura 5.4. Nesta figura, encontram-se dois submódulos, ou seja, o submódulo **Adaptador de Grafos** (AG) e o submódulo ADG. O submódulo AG é responsável pela adaptação dos grafos de dependência gerados pelo AAD. Este submódulo efetua as duas ações de adaptação e cria um grafo de dependência pronto para aplicação do ADG. Por sua vez, o submódulo ADG é constituido pelo algoritmo ADG mostrado na figura 5.3. Conforme constata-se na figura 5.3, a análise realizada pelo ADG pode ser dividida em quatro passos. Visando facilitar a compreensão do funcionamento do módulo Determinador de Grãos, será realizada uma análise completa para o procedimento *fibonacci* apresentado na figura 4.3.

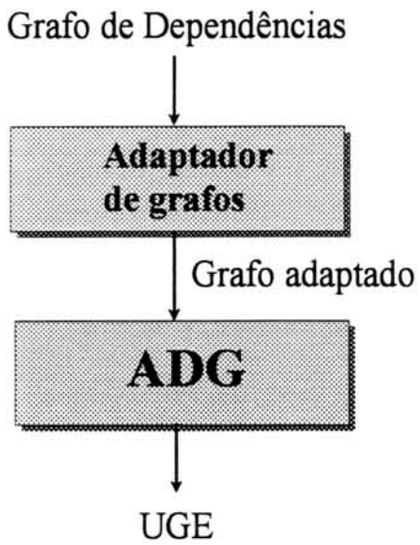


FIGURA 5.4 - Organização do submódulo Deteminador de Grãos

Inicialmente, deve-se realizar a análise global. Sendo assim, podem ser utilizadas as seguintes anotações para o procedimento *fibonacci*:

```
:- mode(fibo/2,[i,o]).
:- type(fibo/2,[int]).
:- measure(fibo/2,[int]).
```

Logo após, o programa é submetido à análise de dependências realizada pelo algoritmo AAD (figura 4.7). Desta análise, resultam os grafos de dependência apresentados na figura 5.5. Seguindo a mesma notação utilizada no capítulo 4, os nodos **I** e **O** representam as entradas e saídas da cláusula, mostrando as dependências entre a cabeça e o corpo. Os demais literais são representados por números de acordo com sua posição no corpo da cláusula.

Terminada a análise global, as informações de modos, tipos, medidas e dependências são entregues ao segundo módulo do GRANLOG, ou seja, o Analisador de Grãos. A figura 5.2 mostra que apenas as dependências são relevantes para

determinação dos grãos. Portanto, o módulo DG recebe apenas os grafos de dependência, enquanto as demais informações são desviadas para o módulo de geração de anotação.

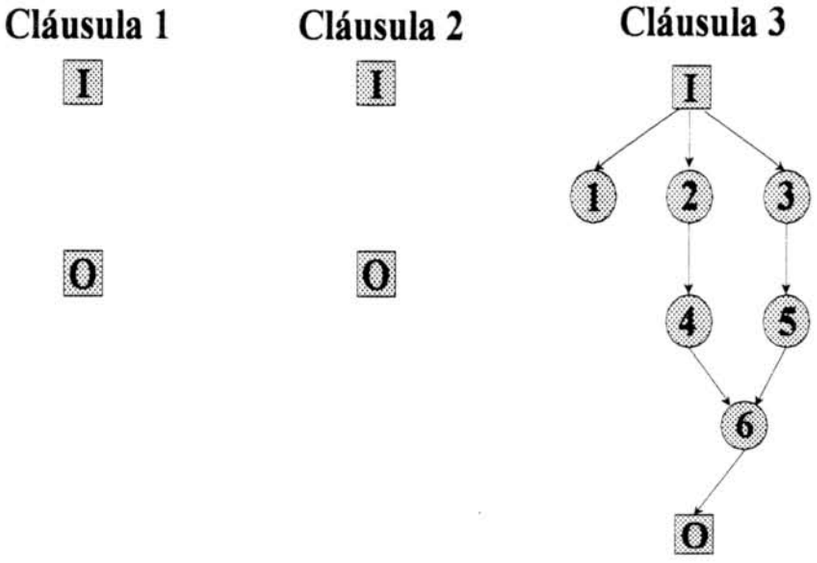


FIGURA 5.5 - Grafos de dependência do procedimento *fibonacci*

Conforme mostra a figura 5.4, o grafo de dependências inicialmente é tratado pelo módulo Adaptador de Grafos, que realiza duas adaptações em sua estrutura. A primeira adaptação retira do grafo o nodo que representa a cabeça da cláusula. Este nodo não é relevante para a determinação dos grãos. Apesar da representação gráfica utilizar dois nodos para representar a cabeça (nodos I e O), na verdade o algoritmo de análise de dependências gera apenas um nodo para a cabeça. A segunda adaptação consiste em aplicar uma Regra de Transitividade as dependências existentes no corpo da cláusula. Esta regra estabelece que se um literal B depende de um literal A (A → B) e um literal C depende de um literal B (B → C), necessariamente o literal C depende do literal A (A → C) e esta **dependência deve estar explícita** no grafo. A figura 5.6 mostra o grafo de dependências adaptado. Nesta figura encontra-se apenas o grafo adaptado da terceira cláusula. As linhas pontilhadas representam as dependências criadas pela aplicação da Regra de Transitividade. As duas primeiras cláusulas do procedimento *fibonacci* possuem apenas cabeça e portanto não *sobrevivem* à primeira adaptação feita pelo módulo AG. Sendo assim, o módulo Determinador de Grãos não gera UGEs para as duas primeiras cláusulas, pois essas não possuem grãos em potencial no seu corpo (não possuem nem mesmo corpo).

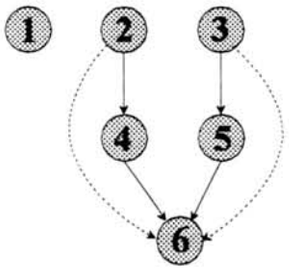


FIGURA 5.6 - Grafo adaptado pelo submódulo Adaptador de Grafos (AG)

O grafo adaptado pode ser analisado pelo ADG. O **primeiro passo** do ADG realiza uma análise do grafo de dependências para detectar literais que não dependam de nenhum outro literal (**literais livres**) e tenham baixa granulosidade. Por decisão de projeto, estes tipos de literais devem ser executados seqüencialmente, pois seu tempo de execução é baixo e sua paralelização pode causar perda de desempenho. Atualmente, dois tipos de literais são detectados e tratados pelo primeiro passo, ou seja, operações relacionais e o *builtin* IS. O tratamento consiste em retirá-los do grafo de dependência antes do particionamento da cláusula e introduzi-los na UGE de forma a serem executados seqüencialmente. A aplicação do primeiro passo do ADG, no grafo apresentado na figura 5.6, resulta na UGE e no grafo apresentados na figura 5.7.

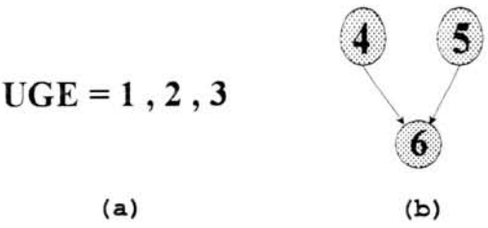


FIGURA 5.7 - Resultado do primeiro passo do ADG no procedimento *fibonacci*

Os três primeiros literais do corpo da terceira cláusula do procedimento *fibonacci* possuem baixa granulosidade e estão livres (não dependem de nenhum outro literal). Desta forma, devem ser executados seqüencialmente. A UGE representa esta situação indicando através de **vírgulas** a obrigatoriedade da execução seqüencial (figura 5.7a). Portanto, a UGE mostrada na figura 5.7a pode ser lida da seguinte forma: **os literais 1, 2 e 3 devem ser executados em seqüência**. O grafo mostrado na figura 5.7b representa as dependências entre os literais da terceira cláusula após a retirada dos literais livres de baixa granulosidade.

O **segundo passo** cria quatro conjuntos que serão utilizados no restante da análise. Estes conjuntos são utilizados para gerenciar o particionamento da cláusula. O conjunto **P** contém todos os literais livres encontrados no grafo. O conjunto **Q** contém os demais literais, ou seja, aqueles que dependem de algum outro literal. Por sua vez, o conjunto **QQ** contém subconjuntos formados por elementos de **Q**. Estes subconjuntos agrupam os literais que dependem dos mesmos elementos armazenados em **P**. Finalmente, o conjunto **PP** contém subconjuntos formados por elementos de **P**. Estes subconjuntos agrupam os literais dos quais dependem os elementos agrupados nos subconjuntos de **QQ**. Portanto, existe uma relação entre os elementos de **PP** e **QQ**. Para cada subconjunto de **QQ** existe um subconjunto em **PP**. Por exemplo, se P = {A,B}, Q = {C,D,E}, A → C, B → C, A → D, B → D, A → E então PP = {{A,B},{A}} e QQ = {{C,D},{E}}. Neste exemplo, o primeiro subconjunto de **QQ** contém os elementos de **Q** que dependem de **A** e **B**. Por sua vez, o segundo subconjunto de **QQ** contém o elemento de **Q** que depende somente de **A**. Conforme demonstra o exemplo, os subconjuntos de **QQ** nunca compartilham elementos.

A aplicação do segundo passo do algoritmo **ADG** no grafo apresentado na figura 5.7b gera os seguintes conjuntos:

```
P  = {4,5}          PP = { {4,5} }
Q  = {6}            QQ = ( {6} }
```

Estes conjuntos representam o paralelismo existente no grafo. O conjunto **P** contém os literais livres, os quais podem ser executados em paralelo. No exemplo, os literais 4 e 5 não dependem de nenhum outro literal e portanto são livres. O conjunto **Q** armazena o literal 6, indicando que ele não é livre e portanto não pode ser ainda paralelizado. O conjunto **QQ** possui apenas um subconjunto, o qual possui relação direta com o único subconjunto de **PP**. O elemento do subconjunto de **QQ** (literal 6) depende de todos os elementos armazenados no subconjunto de **PP** (literais 4 e 5). Neste exemplo, existe apenas um subconjunto em **PP** e um subconjunto em **QQ**. No entanto, em procedimentos mais complexos poderão existir vários subconjuntos em **PP** e **QQ**, conforme demonstrado em [MUT90] e [BUE93]. Além disso, o exemplo apresentado na seção 5.6 gera um grafo de dependências mais complexo, permitindo assim uma maior esclarecimento sobre a geração e aplicação dos conjuntos **P**, **Q**, **PP** e **QQ**.

O **terceiro passo** do algoritmo inicia o particionamento do grafo. Neste passo todos os elementos de **P** são considerados com potencial de paralelismo e portanto paralelizados. Esta paralelização é representada pela colocação do símbolo **&** entre os literais. Este símbolo indica a independência entre as partes da **UGE** que ele delimita (paralelismo E independente [HER89]). Além disso, este passo concatena os literais de **P** paralelizados com a **UGE** já existente. Esta concatenação é realizada através de uma vírgula indicando a obrigatoriedade da execução seqüencial da **UGE** existente com os novos literais paralelizados. Portanto, a **UGE** apresentada na figura 5.7a se transforma em:

$$\mathbf{UGE = 1 , 2 , 3 , \textit{4 \& 5}}$$

Esta UGE pode ser lida assim: **os literais 1, 2 e 3 devem ser executados em seqüência, logo após, os literais 4 e 5 podem ser executados em paralelo.**

O **quarto e último passo** do **ADG** analisa o restante do grafo e complementa a **UGE**. Este passo consiste de um laço que analisa, a cada iteração, um dos subconjuntos **SQQ** armazenados no conjunto **QQ**. O laço possui internamente três ações. A primeira ação cria para cada **SQQ** um grafo de dependências (**GD_SQQ**) composto dos elementos de **SQQ** e das dependências existentes entre estes elementos. Estas informações são facilmente obtidas no grafo de dependências original. A segunda ação aplica recursivamente o algoritmo **ADG** no **GD_SQQ**, obtendo assim, uma **UGE** (**UGE_QQ**) para o conjunto **SQQ** em análise. A terceira ação integra a **UGE_QQ** com a **UGE** existente. Esta integração consiste na inserção da **UGE_QQ**, criada na segunda ação, num ponto da **UGE** onde mantenha-se a corretude das dependências. Este processo insere a **UGE_QQ** na **UGE**, de forma que todos os elementos do subconjunto de **PP**, que tenha relação direta com **QQ**, sejam obrigatoriamente executados antes da **UGE_QQ**. Necessariamente, todos os elementos deste subconjunto de **PP** devem estar codificados na **UGE**, pois no terceiro passo todos os elementos de **P** foram inseridos na **UGE**. Resumindo, o quarto passo cria para cada subconjunto de **QQ** uma **UGE**, valendo-se para isso de chamadas recursivas ao algoritmo **ADG**. Cada **UGE_QQ** obtida é inserida na **UGE** corrente, a qual torna-se a nova **UGE** corrente onde serão realizadas

as próximas inserções (próximas iterações da laço). Portanto, a **UGE** obtida no terceiro passo serve de base para a inserção sucessiva das **UGE_QQs**.

Dando continuidade ao exemplo, constata-se que o conjunto **QQ** possui apenas um subconjunto **SQQ**, ou seja, SQQ = {6}. Sendo assim, será realizada apenas uma iteração do laço. A primeira ação cria o **GD_SQQ** para o subconjunto **SQQ**. O **GD_SQQ** é composto de um único nodo (literal 6) e não possui informações de dependências. A segunda ação aplica o algoritmo **ADG** ao **GD_SQQ**. Desta chamada, resulta a seguinte **UGE_QQ**:

UGE_QQ = 6

Analisando-se o fluxo de execução da chamada recursiva do **ADG**, constata-se que, sendo o literal 6 um *builtin* IS, o primeiro passo do algoritmo cria a **UGE_QQ** e retira do **GD_SQQ** seu único nodo. Desta forma, os próximos passos não serão executados, pois não existe grafo a ser analisado.

A terceira ação integra a **UGE_QQ** com a **UGE** corrente (**UGE** criada no terceiro passo). Esta integração consiste na inserção da **UGE_QQ** num determinado ponto da **UGE**. Este ponto é determinado pela análise da **UGE**, devendo-se inserir a **UGE_QQ** logo após os elementos do subconjunto de **PP** relacionado com o **SQQ** em análise. O subconjunto de **PP** relacionado com **SQQ** é {4,5} e portanto, a **UGE_QQ** deve ser inserida seqüencialmente, logo após os literais 4 e 5. Esta inserção resulta na **UGE** final apresentada a seguir:

UGE = 1 , 2 , 3 , 4 & 5, 6

A leitura desta UGE é a seguinte: **os literais 1, 2 e 3 devem ser executados em seqüência, logo após, os literais 4 e 5 podem ser executados em paralelo e após o término da execução dos literais 4 e 5, pode-se executar o literal 6.**

A UGE final contém a representação do paralelismo existente no corpo da cláusula. Esta representação explicita os grãos em potencial que poderão ser utilizados para exploração do paralelismo E na execução paralela de um programa em lógica. Devido à simplicidade das dependências existentes entre os literais da terceira cláusula do procedimento *fibonacci*, foram obtidos apenas grãos-meta na UGE final. No entanto, no exemplo apresentado na seção 5.6 serão demonstrados grãos-metas, ou seja, grãos formados por conjunto de metas. A próxima seção descreve a anotação de grãos, completando a análise de grãos do procedimento *fibonacci* e tecendo comentários a respeito de todos os grãos existentes neste procedimento (inclusive os grãos-cláusula).

5.5 Anotação de Grãos

A anotação de grãos consiste de um conjunto de anotações inseridas no programa em lógica pelo módulo AGR. Esta anotação destaca os grãos existentes no programa e disponibiliza um conjunto de informações relacionadas com estes grãos. A figura 5.2

mostra que o submódulo responsável pela geração da anotação de grãos (submódulo GAG) recebe como entradas o programa em lógica, os modos, tipos e medidas gerados pelo módulo AGL e os grãos em potencial (UGEs) gerados pelo submódulo DG. De posse destas entradas, o submódulo GAG introduz no programa em lógica a anotação de grãos, gerando o programa particionado (saída do módulo AGR, veja figura 5.1).

Uma importante decisão tomada durante o projeto do GRANLOG foi a utilização de anotações para armazenamento das informações inferidas pelo modelo. Na seção 3.2 encontra-se uma discussão sobre as motivações que levaram a esta decisão. As anotações geradas pelo módulo Analisador de Grãos (anotação de grãos) e pelo módulo Analisador de Complexidade (anotação de complexidade, discutida no próximo capítulo) compõem a anotação final gerada pelo GRANLOG, ou seja, compõem a anotação de granulosidade.

A anotação de grãos é composta de dois tipos de anotações, ou seja, **anotação de grãos em potencial** (AGP) e **anotação de informações de grãos** (AIG). A **AGP** é utilizada para anotar os grãos em potencial existentes no programa em lógica. Esta anotação consiste basicamente da inserção do símbolo **&** no corpo das cláusulas, indicando quais partes da cláusula são independentes (potencial de paralelismo). Além disso, são utilizados parênteses para organizar os vários níveis de grãos. Conforme mostra a figura 5.2, o submódulo Gerador de Anotação de Grãos (GAG) recebe os grãos em potencial gerados pelo submódulo Determinador de Grãos (DG). Estes grãos em potencial são representados pelas UGEs discutidas na seção 5.4. As UGEs geradas pelo submódulo DG são uma representação das verdadeiras UGEs que serão construídas pelo submódulo GAG. A anotação de grãos em potencial é gerada com a substituição de cada número nas UGEs fornecidas pelo DG pelos verdadeiros literais existentes na cláusula.

Por sua vez, a **AIG** consiste de três anotações que armazenam informações sobre os grãos em potencial existentes no programa. Cada grão em potencial possui uma anotação de informações. Conforme discutido na seção 5.3, o GRANLOG propõe a criação de dois tipos de grãos, ou seja: grãos-OU e grãos-E. Além disso, na seção 5.3 são descritas três espécies de grãos propostas pelo GRANLOG, uma do tipo grão-OU (grão-cláusula) e duas do tipo grão-E (grão-meta e grão-metas). Cada uma destas espécies possui uma anotação, dando origem assim a três anotações, ou seja, ***grain_clause***, ***grain_goal*** e ***grain_goals***. A seguir será apresentada a sintaxe destas anotações e a descrição de cada um dos seus parâmetros. Logo após, a geração das anotações será exemplificada para o procedimento *fibonacci*. A sintaxe das anotações é a seguinte:

```
:- grain_clause(<identificador do procedimento>, <nome do grão>,
                <lista de modos>, <lista de medidas>).

:- grain_goal(<identificador do procedimento>, <nome do grão>,
              <lista de modos>,<lista de medidas>,<lista de tipos>).

:- grain_goals(<identificador do procedimento>, <nome do grão>,
               <lista de argumentos>,<lista de modos>,<lista de medidas>,
              <lista de tipos>).
```

As três **anotações *grain*** compartilham os seguintes parâmetros:

- **<identificador do procedimento>**: Identifica o procedimento ao qual o grão pertence. O identificador contém o nome do procedimento seguido da aridade. Por exemplo, o identificador *fibo/2* indica que o grão pertence ao procedimento *fibonacci*.
- **<nome do grão>**: Identifica o grão no procedimento. Esta identificação é realizada atraves de um nome para o grão que segue uma regra de construção. A regra estabelece que o nome é constituido da letra *g* (oriunda de *grain*) seguida de inteiros que indicam a posição do grão no procedimento. O primeiro inteiro indica a cláusula no qual o grão está localizado. Os próximos inteiros (separados por *underscore*) indicam a posição do grão na cláusula. Considerando-se que podem existir diversos níveis de grãos (grãos dentro de grãos), podem existir diversos inteiros separado por *underscore*. Por exemplo, o nome de grão *g2_3_1* indica que o grão está localizado na segunda cláusula (*g2*), dentro do terceiro grão do primeiro nível (*g2_3*) e é o primeiro grão interno a *g2_3* (*g2_3_1*).
- **<lista de modos>**: Este parâmetro contém uma lista com os modos dos argumentos do grão. Cada argumento possui um modo. Se todos argumentos possuirem o mesmo, a lista conterá apenas um elemento representando o modo único.
- **<lista de medidas>**: Este parâmetro contém uma lista com as medidas a serem utilizadas para dimensionar os argumentos dos grãos. Cada argumento possui uma medida. Quando todos argumentos compartilharem a mesma medida, a lista conterá apenas um elemento representando a medida compartilhada.

As anotações *grain_goal* e *grain_goals* compartilham o seguinte parâmetro:

- **<lista de tipos>**: Este parâmetro consiste de uma lista contendo os tipos dos argumentos do grão. Cada argumento possui um tipo. Se todos os argumentos compartilharem o mesmo tipo, a lista armazenará apenas o tipo compartilhado. A anotação *grain_clause* não possui este parâmetro, pois os tipos são utilizados para dimensionamento das mensagens (custos de comunicação) necessárias apenas para execução paralela de grãos-E em sistemas distribuídos. Estas mensagens conduzem as entradas (argumentos de entrada) do grão e os resultados do processamento (argumentos de saída). Conforme discutido na seção 5.3, o dimensionamento dos custos de comunicação gerados pela execução paralela de grãos-OU em sistemas distribuídos, deve considerar outras características do programa (envio do contexto completo da execução do programa).

A anotação *grain_goals* possui o seguinte parâmetro adicional:

- **<lista de argumentos>**: O parâmetro `<lista de argumentos>` contém o nome dos argumentos do grão-metas. Este parâmetro é necessário devido à inexistência desta informação na codificação da cláusula.

Dando prosseguimento ao exemplo iniciado na seção 5.4, será apresentada a seguir a geração de anotação de grãos para o procedimento *fibonacci*. Conforme apresentado

na seção 5.4, o submódulo Determinador de Grãos gera para a terceira cláusula do procedimento *fibonacci* a seguinte UGE:

$$\mathbf{UGE = 1 , 2 , 3 , 4 \ \& \ 5, 6}$$

De posse desta UGE e do programa em lógica, o submódulo Gerador de Anotações pode realizar a AGP, gerando a versão do procedimento *fibonacci* apresentado na figura 5.8.

```
fib(0,0).
fib(1,1).
fib(M,N) :-
      M > 1, M1 is M - 1, M2 is M - 2,
      fib(M1,N1) & fib(M2,N2), N is N1 + N2.
```

FIGURA 5.8 - Anotação de grãos em potencial para procedimento *fibonacci*

As duas primeiras cláusulas não foram alteradas, pois não possuem corpo e nem UGEs. A terceira cláusula foi anotada com o símbolo &, indicando o potencial de paralelismo. Basicamente, o processo de anotação consiste em transformar a cláusula numa UGE definitiva. Esta UGE é obtida com a substituição dos números na UGE gerada pelo submódulo Deteminador de Grãos pelos literais verdadeiros existentes na cláusula. Conforme constata-se na versão alterada do procedimento *fibonacci*, o único potencial de paralelismo encontra-se na execução paralela dos literais *fib*. Este potencial representa o paralelismo E existente no procedimento. O paralelismo OU é representado pela codificação do próprio procedimento, onde as cláusulas são representadas em separado. Portanto, a AGP não introduz nenhuma anotação para o paralelismo OU.

A segunda etapa de anotação consiste na geração da AIG. A AIG criada pelo submódulo GAG para o procedimento *fibonacci* é apresentada na figura 5.9. Foram geradas para o procedimento *fibonacci* três anotações *grain_clause* e duas anotações *grain_goal*. Conforme mostra a figura, não são geradas anotações *grain_goal* para os literais prédefinidos (*builtins*).

```
:- grain_clause(fibo/2,g1,[i,o],[void,int]).
:- grain_clause(fibo/2,g2,[i,o],[void,int]).
:- grain_clause(fibo/2,g3,[i,o],[int]).

:- grain_goal(fibo/2,g3_4,[i,o],[int],[int]).
:- grain_goal(fibo/2,g3_5,[i,o],[int],[int]).
```

FIGURA 5.9 - AIG gerada para o procedimento *fibonacci*

Cada anotação *grain_clause* equivale a uma cláusula do procedimento. O segundo parâmetro (nome do grão) indica a posição da cláusula no procedimento. O terceiro parâmetro das anotações *grain_clause* indica que o primeiro argumento é uma entrada e o segundo argumento atua como saída. Esta informação é fornecida pelo módulo AGL e deve ser considerada do ponto de vista da chamada e não do ponto de vista do grão-cláusula, ou seja, a indicação da direcionalidade dos argumentos não considera as

características individuais de cada cláusula. Portanto, todos grãos-cláusula de um procedimento sempre terão a mesma lista de modos.

O quarto parâmetro dos grãos *g1* e *g2* merece uma análise especial. Conforme descreve a subseção 4.5.2, as medidas de tamanho possuem duas aplicações, ou seja, **determinação da granulosidade** e **determinação do tamanho das saídas**. A subseção 4.5.2 cita ainda, que a medida *void* é utilizada quando o argumento não influencia na complexidade de execução do grão e nem influencia no tamanho de suas saídas. Portanto, neste caso, a medida *void* para o primeiro argumento dos grãos *g1* e *g2* indica em primeiro lugar, que seu tamanho não influencia na complexidade de execução da cláusula. Isto ocorre porque os grãos são fatos e portanto a complexidade de execução é constante. Em segundo lugar, a medida *void* indica que o tamanho do primeiro argumento não é relevante para determinação do tamanho da saída. Isto ocorre porque, neste caso, o tamanho da saída é constante, conforme pode-se constatar na figura 5.8. Durante a geração das anotações *grain_clause g1* e *g2*, o submódulo GAG ajusta as medidas de tamanho do quarto parâmetro para refletirem a realidade da cláusula. Apesar do usuário introduzir apenas uma declaração de medidas por procedimento (veja início do exemplo na seção 5.4), o módulo GAG infere através da análise da cláusula diversos ajustes que permitem uma adaptação local das informações, aumentando assim a precisão da anotação de grãos gerada pelo AGR. Por outro lado, o quarto argumento do grão-cláusula *g3* indica que o tamanho do primeiro argumento é relevante e deve ser medido como inteiro (medida *int*). Esta informação é idêntica a introduzida pela usuário através das declarações.

As anotações *grain_goal* para o procedimento *fibonacci* são semelhantes, pois os dois grãos-meta consistem da mesma meta (meta *fibo*). O nome dos grãos indica que o primeiro grão-meta encontra-se na terceira cláusula, localizado na quarta posição (quarto literal) e que o segundo grão-meta encontra-se na terceira cláusula, localizado na quinta posição (quinto literal). Os demais parâmetros dos grãos-meta são idênticos e indicam que os argumentos são entrada e saída, que os argumentos são relevantes para a análise de granulosidade (medidos como inteiros) e que os dois argumentos são do tipo inteiro.

As anotações *grain* poderiam ser colocadas em qualquer parte do programa, pois o identificador do procedimento (primeiro parâmetro) e o nome do grão (segundo parâmetro) **indicam com exatidão apenas um grão no programa**. No entanto, as anotações *grain* são introduzidas exatamente antes do procedimento onde estão localizados os grãos descritos por elas. O programa particionado (saída do módulo AGR) para o procedimento *fibonacci* é mostrado na figura 5.10.

```
:- grain_clause(fibo/2,g1,[i,o],[void,int]).
:- grain_clause(fibo/2,g2,[i,o],[void,int]).
:- grain_clause(fibo/2,g3,[i,o],[int]).
:- grain_goal(fibo/2,g3_4,[i,o],[int],[int]).
:- grain_goal(fibo/2,g3_5,[i,o],[int],[int]).


fib(0,0).
fib(1,1).
fib(M,N) :-
            M > 1, M1 is M - 1, M2 is M - 2,
            fib(M1,N1) & fib(M2,N2), N is N1 + N2.
```

FIGURA 5.10 - Programa particionado para procedimento *fibonacci*

O programa particionado gerado para o procedimento *fibonacci* não permite uma compreensão de todo o potencial da anotação de grãos. O exemplo apresentado na seção 5.6 demonstra a representação de vários níveis de grãos no corpo das cláusulas, exemplifica a anotação *grain_goals* e permite ao leitor constatar uma série de características interessantes do módulo AGR.

5.6 Exemplo de Análise de Grãos: Programa *Torre de Hanói*

A seção 4.7 exemplifica a análise global para o programa *Torre de Hanói* mostrado na figura 4.10. Esta análise gera um conjunto de informações sobre o programa, as quais serão utilizadas pela análise de grãos. Conforme mostra a figura 5.1, o módulo AGR recebe as informações geradas pelo módulo AGL e introduz no programa em lógica as anotações de grãos, criando assim o programa particionado.

Como mostra a figura 5.2, as informações de dependências são entreguês inicialmente ao módulo Determinador de Grãos (DG). Estas informações estão organizadas em grafos de dependências, os quais constituem uma das saídas do AGL. As figuras 4.15 e 4.16 mostram os grafos de dependências gerados pelo AGL para os dois procedimentos do programa *Torre de Hanói*. Estes grafos são utilizados pelo DG para gerar as UGEs que representam os grãos em potencial no corpo das cláusulas. Conforme descrito na seção 5.4, cláusulas que não possuam corpo (fatos), não produzem UGEs (não possuem paralelismo E). Portanto, as primeiras claúsulas de ambos os procedimentos do programa *Torre de Hanói* não geram UGEs. Os próximos parágrafos demonstram a geração de UGEs para as demais cláusulas do programa.

A segunda cláusula do procedimento *hanoi* produz uma UGE com várias caraterísticas interessantes. Conforme mostra a figura 5.4, o primeiro passo executado pelo módulo DG é a adaptação de grafos. A figura 5.11 apresenta o grafo de dependências da segunda cláusula do procedimento *hanoi*, após a adaptação realizada pelo submódulo Adaptador de Grafos.

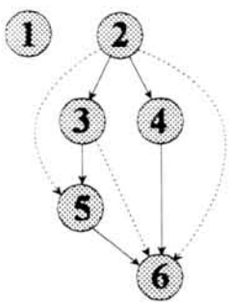
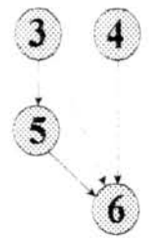


FIGURA 5.11 - Grafo adaptado da segunda cláusula do procedimento *hanoi*

A figura 5.4 mostra que o grafo adaptado serve de entrada para a segunda fase do DG, ou seja, para o submódulo ADG. Conforme citado na seção 5.4, o ADG é implementado pelo algoritmo apresentado na figura 5.3, o qual é constituido de quatro passos.

O **primeiro passo** detecta os literais livres de baixa granulosidade. O literal 1 é uma operação relacional e o literal 2 é um *builtin* IS. Conforme mostra a figura 5.11, ambos literais são livres. Portanto, os dois literais são inseridos na UGE para execução

seqüencial e retirados do grafo. A figura 5.12 apresenta a UGE e o grafo de dependências resultantes da aplicação do primeiro passo do algoritmo.



UGE = 1 , 2

(a) (b)

FIGURA 5.12 - Primeiro passo do ADG na segunda cláusula de *hanoi*

A UGE apresentada na figura 5.12a pode ser lida da seguinte forma: **os literais 1 e 2 devem ser executados em seqüência**. O grafo mostrado na figura 5.12b representa as dependências entre os demais literais da cláusula.

O **segundo passo** do algoritmo ADG gera os conjunto P, Q, PP e QQ, os quais serão utilizados no restante da análise. Para o grafo apresentado na figura 5.12b são criados os seguintes conjuntos:

```
P  = {3,4}
Q  = {5,6}
PP = { {3} , {3,4} }
QQ = { {5} ,  {6}  }
```

O conjunto **P** contém os literais livres, ou seja, os literais 3 e 4. O conjunto **Q** contém os demais literais, ou seja, os que dependem de algum outro literal (não livres). Os conjuntos **PP** e **QQ** contém subconjuntos que representam as dependências entre os literais dos conjuntos **P** e **Q**. Os subconjuntos de **QQ** contém elementos de **Q** que dependem dos mesmos elementos de **P**. Por sua vez, os subconjuntos de **PP** contém os elementos de **P** dos quais os elementos de um subconjunto em **QQ** dependem. Por exemplo, nos conjuntos gerados para o grafo da figura 5.12b, o segundo subconjunto de **QQ** contém o literal 6 e o segundo subconjunto de **PP** contém os literais 3 e 4. Sendo assim, pode-se afirmar que o literal 6 depende dos literais 3 e 4. Se houvessem mais literais no segundo subconjunto de **QQ**, todos dependeriam dos literais 3 e 4. Deve-se ressaltar, que para cada subconjunto em **PP** existe um subconjunto em **QQ**.

O **terceiro passo** inicia o particionamento. Neste passo são paralelizados todos os elementos do conjunto **P**. Conforme discutido na seção 5.4, o símbolo **&** é utilizado para representar a independência entre partes da cláusula, representando portanto, potencial de paralelismo. O terceiro passo gera a seguinte **UGE**

UGE = 1 , 2 , *3 & 4*

Esta UGE pode ser lida da seguinte forma: **execute seqüencialmente os literais 1 e 2, logo após, os literais 3 e 4 podem ser executados em paralelo**.

O **quarto passo** complementa a **UGE** atraves da análise dos subconjuntos de **PP** e **QQ**, os quais representam o potencial de paralelismo existentes no restante do grafo. Conforme pode-se verificar na figura 5.3, o quarto passo é constituido de um laço onde são executadas três ações. O laço analisa a cada iteração um elemento de **QQ**, executando assim uma iteração para cada subconjunto armazenado no conjunto **QQ**. O conjunto **QQ** contém dois elementos e portanto serão executadas duas iterações.

A **primeira iteração** analisa o subconjunto {5}. A primeira ação do laço gera o grafo **GD_SQQ**. Considerando-se que existe apenas um elemento no subconjunto, o grafo contém apenas um vértice e nenhuma aresta. A segunda ação aplica recursivamente o algoritmo **ADG** no grafo **GD_SQQ**, gerando a seguinte **UGE_QQ**:

UGE_QQ = 5

A terceira ação integra a **UGE_QQ** a **UGE** corrente. Esta integração insere a **UGE_QQ** na **UGE** de forma que as dependências sejam respeitadas. Basicamente, esta ação analisa o subconjunto **PP** relacionado com o subconjunto **QQ** e verifica onde deve ser realizada a inserção. A **UGE** corrente é a seguinte:

UGE = 1 , 2 , *3* & *4*

A **UGE_QQ** deve ser inserida em seqüência, logo após os elementos do subconjunto de **PP** relacionados com o subconjunto **QQ** que a gerou, respeitando assim as dependências. Portanto, a **UGE_QQ** deve ser inserida logo após o literal 3, pois este é o elemento do subconjunto de **PP**. Sendo assim, a **UGE** resultante é a seguinte:

UGE = 1 , 2 , (3 , 5) & 4

Neste caso, surge a necessidade da utilização de parênteses para explicitar os grãos, pois apesar do literal 5 depender do literal 3, o literal 4 é independente de ambos. Esta UGE pode ser lida da seguinte forma: **execute em seqüência os literais 1 e 2, logo após, existe a possibilidade de executar em paralelo o grupo de literais 3 e 5 com o literal 4. Por sua vez, os literais 3 e 5 devem ser executados em seqüência**. Neste exemplo, destaca-se o agrupamento dos literais 3 e 5 (grão-metas), os quais devem ser executados em seqüência e podem ser executados em paralelo com o literal 4. Esta representação dos grãos em potencial explora ao máximo o paralelismo existente no grafo.

A **segunda iteração** analisa o subconjunto {6}. Para este subconjunto é gerada a seguinte UGE_QQ:

UGE_QQ = 6

A integração UGE corrente e UGE_QQ resulta na seguinte UGE:

UGE = 1 , 2 , (3 , 5) & 4 , 6

Neste caso, a **UGE_QQ** deve ser inserida após os literais 3 e 4, os quais fazem parte do subconjunto de **PP** que estabelece as dependências do subconjunto **QQ** em análise. Desta forma, o literal 6 é inserido no final da **UGE**, respeitando-se assim as dependências. Esta **UGE** é o resultado final da análise realizada pelo módulo Determinador de Grãos para a segunda cláusula do procedimento *hanoi*. Nesta **UGE** está representado todo o potencial de paralelismo existente no corpo da cláusula (paralelismo E).

A segunda cláusula do procedimento *append* possui apenas um literal e portanto gera a seguinte UGE:

UGE = 1

Neste caso, não existe potencial de paralelismo. O submódulo Adaptador de Grafos realiza a adaptação do grafo mostrado na figura 4.16, gerando um grafo adaptado com apenas um vértice. Por sua vez, o submódulo ADG verifica que o literal apesar de estar livre não possui baixa granulosidade (**passo 1**) e cria os seguintes subconjuntos P, Q, PP e QQ (**passo 2**):

```
P  = {1}          PP = {}
Q  = {}           QQ = {}
```

No **terceiro passo**, o ADG gera a UGE e o **quarto passo** não executa nenhuma iteração (conjunto **QQ** vazio).

Conforma mostra a figura 5.2, as UGEs resultantes do submódulo Deteminador de Grãos são entreguês ao submódulo Gerador de Anotações, o qual gera a anotação de grãos. Além disso, a seção 5.5 afirma que a anotações de grãos é composta de dois tipos de anotações, ou seja, **anotação de grãos em potencial** (AGP) e **anotação de informações de grãos** (AIG).

Os próximos parágrafos demonstram a geração da AGP para o programa *Torre de Hanói*. As primeiras cláusulas dos procedimentos *hanoi* e *append* não geram AGP, pois não possuem UGEs e conseqüentemente, não possuem potencial de paralelismo E. A segunda cláusula do procedimento *append* mantém-se inalterada, pois sua UGE contém apenas o próprio literal *append* existente no seu corpo, não possuindo assim potencial de paralelismo. No entanto, a segunda cláusula do procedimento *hanoi* sofre interessantes mudanças, as quais representam o potencial de paralelismo existente no seu corpo. A AGP gerada para a segunda cláusula de *hanoi* é apresentada na figura 5.13.

Comparando-se o programa mostrado na figura 4.10 com a anotação apresentada na figura 5.13, constata-se que houve uma alteração na ordem dos literais no corpo da segunda cláusula do procedimento *hanoi*. Esta alteração permite a máxima exploração do paralelismo existente no corpo da cláusula. Desta forma, a ordem estabelecida pelo programador nem sempre será a mais interessante para a exploração do paralelismo. O GRANLOG analisa o programa e realiza as alterações necessárias para paralelização eficiente da cláusula. Além disso, deve-se ressaltar o agrupamento num único grão (grão-metas) dos primeiros literais *hanoi* e *append* do corpo da segunda cláusula. Esta é uma inovação proposta pelo GRANLOG, visando a máxima exploração do paralelismo E. A seção 5.3 discute os tipos de grãos propostos pelo GRANLOG, abordando a criação do grão-metas e a redefinição do paralelismo E decorrente desta inovação.

```
hanoi(N,A,B,C,M) :-
      N > 1, N1 is N - 1,
      (hanoi(N1,A,C,B,M1),append(M1,[mv(A,C)],T)) &
       hanoi(N1,B,A,C,M2) , append(T,M2,M).
```

FIGURA 5.13 - AGP para segunda cláusula do procedimento *hanoi*

O resultado completo da AGP para o programa *Torre de Hanói* é apresentado na figura 5.14. Nesta figura constata-se que foram realizadas alterações apenas na segunda cláusula do procedimento *hanoi*. Após a geração da ADG, o submódulo Gerador de Anotações cria a anotação de informações de grãos (AIG). A figura 5.15 mostra a AIG completa, gerada para o programa *Torre de Hanói*.

```
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
      N > 1, N1 is N - 1,
      (hanoi(N1,A,C,B,M1),append(M1,[mv(A,C)],T)) &
      hanoi(N1,B,A,C,M2), append(T,M2,M).

append([],L,L).
append([HL],L1,[HR]) :- append(L,L1,R).
```

FIGURA 5.14 - Anotação de grãos em potencial para programa *Torre de Hanói*

Para o procedimento *hanoi* foram geradas sete anotações: duas *grain_clause*, uma *grain_goals* e quatro *grain_goal*. As anotações *grain_clause* armazenam as informações relacionadas com os grãos-cláusula existentes no procedimento. Cada cláusula gera um grão-cláusula. A primeira anotação *grain_clause* descreve o fato (grão *g1*). O terceiro parâmetro desta anotação indica a direcionalidade dos argumentos, ou seja, os quatro primeiros são entradas e o último é saída. O quarto parâmetro indica que o tamanho dos quatro primeiros argumentos não é relevante para a análise de granulosidade (medida *void*). Esse parâmetro indica ainda, que a medida associada a saída é *length*. Conforme discutido na seção 5.5, essas informações são obtidas localmente pelo submódulo GAG através de uma análise do fato. A segunda anotação *grain_clause* descreve a segunda claúsula do procedimento *hanoi*. O último parâmetro desta anotação indica que o primeiro e o último argumento são relevantes para a análise de granulosidade. O primeiro deve ser medido como um inteiro (medida *int*) e o segundo medido como o número de elementos de uma lista (medida *length*).

```
:- grain_clause(hanoi/5,g1,[i,i,i,i,o],[void,void,void,void,length]).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],[int,void,void,void,length]).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],[i,i,i,i,o],[int,void,void,void,length],
                [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])].
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],[int,void,void,void,length]).
                [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length]).
                [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
                 list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],[int,void,void,void,length]).
                [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],[list(?,[struct(2,2,[atom(3)])])]).

:- grain_clause(append/3,g1,[i,i,o],[void,length,length]).
:- grain_clause(append/3,g2,[i,i,o],[length]).
:- grain_goal(append/3,g2_1,[i,i,o],[length],[list(?,[struct(2,2,[atom(3)])])]).
```

FIGURA 5.15 - AIG para programa *Torre de Hanói*

A anotação *grain_goals* descreve o grão-metas existente no corpo da segunda cláusula. O segundo parâmetro contém o nome do grão seguindo as regras de construção descritas na seção 5.5. Este parâmetro (*g2_3*) indica que a anotação descreve o grão localizado na segunda cláusula e na terceira posição. O terceiro parâmetro contém o nome dos argumentos do grão-metas, os quais são obtidos pelo GAG através da análise das metas que compõem o grão. Este parâmetro é necessário devido a inexistência destas informações na codificação do programa, pois o grão-metas não é implementado por nenhum procedimento. Os argumentos do grão-cláusula e do grão-meta estão explícitos no código do programa e portanto não necessitam deste parâmetro. Os três últimos parâmetros da anotação para o grão-metas *g2_3* contém os modos, medidas e tipos para os argumentos do grão. Todas estas informações são inferidas pelo módulo GAG através da análise das metas componentes do grão e de suas respectivas informações de modos, medidas e tipos.

As anotações *grain_goal* descrevem as quatro metas existentes no corpo da segunda cláusula. Os dois primeiros grãos-meta são internos ao grão-metas *g2_3*. Seus nomes indicam esta situação (*g2_3_1* e *g2_3_2*). Neste exemplo, torna-se claro os vários níveis de grãos que podem existir num procedimento. Portanto, o grão-metas *g2_3* é formado pelos grãos-meta *g2_3_1* e *g2_3_2*. Os demais parâmetros armazenam as informações obtidas na análise global (modos, tipos e medidas). Conforme discutido no capítulo 4, atualmente estas informações são introduzidas no programa por declarações do usuário. No entanto, ainda o capítulo 4 cita que é possível realizar adaptações das informações introduzidas pelo usuário para a realidade dos grãos existentes no programa. O grão *g2_3_2* demonstra como pode-se realizar este aperfeiçoamento das anotações. Conforme constata-se na seção 4.7, o usuário introduziu as seguintes anotações para o procedimento *append*:

```
:- mode(append/3,[i,i,o]).
:- measure(append/3,[length].
:- type(append/3,[list(?,[struct(2,2,[atom(3)])])].
```

No entanto, no grão *g2_3_2* estas informações foram adaptadas para a realidade local da chamada do grão, gerando a seguinte anotação:

```
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length],
              [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
               list(?,[struct(2,2,[atom(3)])])]).
```

Analisando-se a anotação, verifica-se que o quarto parâmetro indica, através da medida *void*, que o segundo argumento não é relevante para determinação dinâmica (tempo de execução) da complexidade do grão. Além disso, o quinto parâmetro indica que a lista que servirá de entrada no segundo argumento possui, para qualquer chamada, apenas um elemento. Portanto, o tamanho da lista é constante. Estas duas adaptações refletem a situação local da chamada do grão-meta *g2_3_2*, aperfeiçoando assim, as informações introduzidas pelo usuário através das declarações. Deve-se ressaltar ainda, que o segundo argumento não é relevante para determinação dinâmica da complexidade do grão porque seu tamanho é constante (sempre uma lista com um elemento) e portanto a influencia deste argumento na complexidade pode ser avaliado em tempo de compilação. Outra característica interessante neste exemplo, consiste na possibilidade de dimensionamento exato, em tempo de compilação, do tamanho do segundo argumento do grão. Sabe-se que o segundo argumento sempre será uma lista de um elemento contendo uma estrutura com um nome de dois caracteres e com dois argumentos que são átomos com tamanho 3. Portanto, analisando-se as características do sistema paralelo a ser utilizado, pode-se descobrir exatamente o tamanho deste argumento. Sendo assim, pode-se determinar o custo de comunicação gerado pelo envio do argumento de um processador para outro. Em sistemas distribuídos este custo será o tempo necessário para envio de uma mensagem contendo o número de bytes que codificam o argumento.

A adaptação realizada pelo GRANLOG nas informações do grão *g2_3_2* é simples e considera apenas as informações explicitadas na codificação da cláusula. No entanto, através da análise global do programa, pode-se obter informações sobre o estado de instanciação das variáveis no momento das chamadas de grande parte dos grãos e portanto, pode-se introduzir nas anotações *grain* informações precisas sobre as características de cada grão no momento de sua execução. A utilização de declarações pelo usuário, mesmo com as adaptações locais realizadas pelo submódulo GAG, limita as informações introduzidas nas anotações *grain*. Esta situação pode ser verificada através da análise da AIG mostrada na figura 5.15. Nesta AIG, a maioria das anotações contém as mesmas informações introduzidas pelo usuário. A realização de uma análise global permitirá uma melhoria substancial na precisão das informações geradas pelo GRANLOG.

Para o procedimento *append* foram geradas três anotações: duas *grain_clause* e uma *grain_goal*. A primeira anotação *grain_clause* armazena informações sobre a primeira cláusula. O último parâmetro indica que o primeiro argumento não é relevante para a análise de granulosidade. Além disso, esse parâmetro indica que os dois últimos argumentos devem ser mensurados pela medida *length*. Nota-se nesse caso, que apesar da segunda entrada não influenciar na complexidade da cláusula (fato), influencia no tamanho da saída. A segunda anotação *grain_clause* indica que todos os argumentos influenciam na análise de granulosidade e devem ser medidos com *length*. A anotação *grain_goal* armazena as informações da única meta existente no corpo da segunda cláusula e contém as mesmas informaçõe introduzidas pelo usuário através das declarações *mode*, *type* e *measure*.

A figura 5.2 mostra que a saída do módulo AGR é o programa particionado, o qual é composto pelo programa em lógica acrescido da anotação de grãos. O programa particionado para o programa *Torre de Hanói* é mostrado na figura 5.16. Nesta figura verifica-se que as anotações *grain* são inseridas, exatamente antes do procedimento onde estão localizados os grãos que descrevem.

```
:- grain_clause(hanoi/5,g1,[i,i,i,i,o],[void,void,void,void,length]).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],[int,void,void,void,void]).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],[i,i,i,i,o],[int,void,void,void,length],
               [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],[int,void,void,void,length]).
               [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length]).
               [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
                list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],[int,void,void,void,length]).
               [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],[list(?,[struct(2,2,[atom(3)])])]).

hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
      N > 1, N1 is N - 1,
     (hanoi(N1,A,C,B,M1),append(M1,[mv(A,C)],T)) &
      hanoi(N1,B,A,C,M2), append(T,M2,M).

:- grain_clause(append/3,g1,[i,i,o],[void,length,length]).
:- grain_clause(append/3,g2,[i,i,o],[length,length,void]).
:- grain_goal(append/3,g2_1,[i,i,o],[length],[list(?,[struct(2,2,[atom(3)])])]).

append([],L,L).
append([HL],L1,[HR]) :- append(L,L1,R).
```

FIGURA 5.16 - Programa particionado para programa *Torre de Hanói*

5.7 Conclusões

Este capítulo discutiu o conceito análise de grãos e descreveu o segundo módulo do GRANLOG, ou seja, o Analisador de Grãos. Foram apresentados os tipos de grãos propostos pelo modelo e as duas etapas da análise de grãos, ou seja, a determinação de grãos e a geração de anotações. Destacou-se neste capítiulo a apresentação do algoritmo para determinação de grãos (ADG). Além disso, foram apresentados dois exemplos completos, demonstrando a análise de grãos para o procedimento *fibonacci* e para o programa *Torre de Hanói*.

As principais conclusões alcançadas neste capítulo são as seguintes:

- uma apurada determinação dos grãos em potencial existentes no programa pode ser realizada em tempo de compilação com o auxílio da análise global;
- a definição tradicional de paralelismo E como execução paralela de metas limita a exploração do paralelismo na programação em lógica;
- a nova definição de paralelismo E proposta pelo GRANLOG aumenta a exploração de paralelismo nos programas em lógica;
- a anotação de grãos gerada pelo módulo AGR terá uma melhoria significativa na precisão, quando o módulo AGL implementar uma análise global de programa;
- apesar da limitação imposta pelas poucas informações introduzidas pelas declarações do usuário, o GRANLOG realiza sempre que possível uma adaptação local das informações.

O próximo capítulo discute a análise de complexidade e descreve o último módulo do GRANLOG, denominado Analisador de Complexidade (AC). O módulo AC recebe o programa particionado produzido pelo AGR e gera o programa granulado, resultado final da análise de granulosidade.

6 Análise de Complexidade

Este capítulo discute a análise de complexidade e apresenta o último módulo do GRANLOG, ou seja, o Analisador de Complexidade (AC). A seção 6.1 introduz o conceito de análise de complexidade, discutindo conceitos básicos, aplicações e trabalhos sobre o assunto. Na seção 6.2 é apresentado o módulo Analisador de Complexidade, descrevendo-se suas interfaces e organização interna. A seção 6.3 descreve os tipos de complexidade utilizados pelo GRANLOG. Destaca-se nesta seção, uma proposta para criação de uma taxonomia para complexidade de programas em lógica. Esta taxonomia cria os termos complexidade OU e complexidade E. A seção 6.4 discute a proposta do GRANLOG para análise de complexidade OU. Por sua vez, a seção 6.5 descreve a proposta para análise de complexidade E. Finalizando o exemplo iniciado na seção 4.7 e complementado na seção 5.4, a seção 6.6 apresenta a análise de complexidade para o programa *Torre de Hanói*. Nesta seção exemplifica-se a geração da anotação de complexidade. Esta anotação complementa a anotação de grãos, formando a anotação final do GRANLOG, ou seja, a anotação de granulosidade. Finalmente, na seção 6.7 são apresentadas as conclusões deste capítulo.

6.1 Introdução

Conforme afirma Nai-Wei Lin ([LIN93]), a **complexidade computacional de um programa** consiste do montante de recursos computacionais, tais como tempo e espaço de memória, consumidos durante a execução do programa. Afirma ainda Lin, que a **análise de complexidade** consiste na inferência de informações a respeito da complexidade computacional de um programa através do exame do seu texto. Normalmente, a análise de complexidade dedica-se à previsão do tempo necessário para execução de um programa. Finalizando sua introdução sobre análise de complexidade, Lin cita vários trabalhos sobre o assunto e afirma que diversos pesquisadores têm dedicado seus estudos a automatização desta análise. Surge assim, a **análise automática de complexidade**, ou seja, a análise de complexidade realizada de forma automática pelo computador (sem participação do usuário).

A metodologia a ser empregada na análise de complexidade depende do paradigma de programação no qual foram desenvolvidos os programas a serem analisados. Os primeiros trabalhos sobre este tema dedicaram-se ao estudo da complexidade no paradigma tradicional (imperativo). Em trabalhos posteriores, foram pesquisados os paradigmas de programação funcional e lógica. Em [LIN93] encontra-se uma interessante retrospectiva (com diversas referências) sobre a análise de complexidade nos paradigmas de programação imperativa, funcional e lógica. Além disso, os textos [DEB93], [LIN93], [DEB94a] e [GAR94] apresentam estudos específicos sobre a análise de complexidade na programação em lógica.

Normalmente, a análise de complexidade não consegue prever com **exatidão** o montante de recursos que serão consumidos na execução de um programa. Por exemplo, raramente pode-se prever com exatidão quanto tempo demorará um programa para ser executado. No caso específico da programação em lógica, suas características não-determinísticas tornam-se um forte empecilho para a previsão exata da complexidade. No entanto, a análise de complexidade pode determinar com exatidão os limites (máximo e mínimo) da complexidade. Além disso, pode determinar a complexidade média do programa. Por exemplo, a análise de complexidade pode prever com exatidão que um

determinado programa **nunca** demorará mais do que 5 minutos (limite máximo) para ser executado. Pode determinar que o mesmo programa **nunca** executará em menos do que 1 minuto (limite mínimo). Pode prever ainda, que em média o programa executa em 3 minutos (complexidade média). Sendo assim, foram criados três tipos básicos de análise de complexidade, ou seja: **análise do pior caso**, **análise do melhor caso** e **análise do caso médio**. A análise do pior caso prevê a complexidade máxima de um programa, ou seja, o máximo de recursos que podem ser consumidos durante a execução. A análise do melhor caso dedica-se à previsão da complexidade mínima de um programa, ou seja, o mínimo de recursos que podem ser consumidos pela execução. Por sua vez, a análise do caso médio prevê a complexidade média na execução de um programa, ou seja, a média de recursos consumidos durante a execução. Tanto em [LIN93] quanto em [DEB94], encontra-se a afirmação de que dependendo da aplicação, um dos tipos de análise pode ser mais apropriada. Por exemplo, o modelo descrito em [LIN93] explora o pior caso para programas em lógica. No capítulo 12 do texto [LIN93], Nai-Wei Lin apresenta o motivo dessa escolha do ponto de vista da aplicação das informações de complexidade na análise de granulosidade. Os textos [DEB94a] e [GAR94] discutem os três tipos de análise de complexidade no âmbito da programação em lógica.

A análise de complexidade encontra diversas aplicações no universo dos computadores. No texto [LIN93] são discutidas genericamente várias aplicações e especificamente duas delas para a programação em lógica, ou seja, a análise de granulosidade e a otimização de programas. Conforme constata-se em [DEB93] e [LIN93], a análise de granulosidade é uma das principais aplicações para a análise de complexidade. Nai-Wei Lin dedica um capítulo de sua tese de doutorado à discussão desta aplicação na programação em lógica, apresentando diversas definições e descrevendo os resultados obtidos com a utilização da análise de complexidade no controle de granulosidade em dois modelos de exploração do paralelismo nos programas em lógica, ou seja, no modelo &-Prolog ([HER90], [HER91], [BUE93]]) e no modelo ROLOG ([KAL85]). No artigo [DEB93], Debray e Lin resumem sua proposta para análise de complexidade e mais uma vez, demonstram a aplicação no controle de granulosidade.

Conforme discutido na subseção 2.1.3, o autor dessa dissertação acredita que o paralelismo deve ser explorado de forma automática (paralelismo implícito). Além disso, o segundo princípio do GRANLOG, apresentado na seção 3.1, destaca a importância da exploração automática do paralelismo para o modelo proposto neste trabalho. A aplicação de técnicas apuradas para análise estática de programas é a base para construção de compiladores paralelizadores eficientes, pois apenas com o conhecimento de informações precisas sobre o comportamento dos programas, podem ser tomadas decisões apropriadas sobre paralelização. O GRANLOG utiliza duas técnicas apuradas para análise estática de programas em lógica, ou seja, a análise global e a análise de complexidade. Estas técnicas são genéricas e fornecem informações que podem ser utilizadas em diversas aplicações. A abordagem modular do GRANLOG, permite que tanto o módulo Analisador Global, quanto o módulo Analisador de Complexidade, possam ser utilizados de forma independente. A análise global foi discutida no capítulo 4. As próximas seções deste capítulo apresentam a proposta para análise de complexidade no modelo GRANLOG.

6.2 Módulo Analisador de Complexidade

O módulo Analisador de Complexidade (AC) realiza uma análise de complexidade do programa em lógica, visando a obtenção de informações a serem utilizadas no controle da granulosidade. Basicamente, o módulo AC analisa o programa e gera uma anotação contendo informações de complexidade. Esta anotação recebe o nome de **anotação de complexidade**. A figura 6.1 mostra uma visão genérica do módulo AC. Nesta figura, verifica-se que o módulo possui como entrada o programa particionado e como saída o programa granulado. O programa particionado é gerado pelo módulo AGR e constitui-se do programa em lógica acrescido da anotação de grãos. Por sua vez, o programa granulado constitui-se do programa particionado, acrescido da anotação de complexidade gerada pelo módulo AC. O programa granulado é o produto final do GRANLOG. Deve-se relembrar ainda, que a anotação de grãos combinada com a anotação de complexidade compõe a anotação de granulosidade. Esta anotação armazena todas as informações geradas pelo GRANLOG.

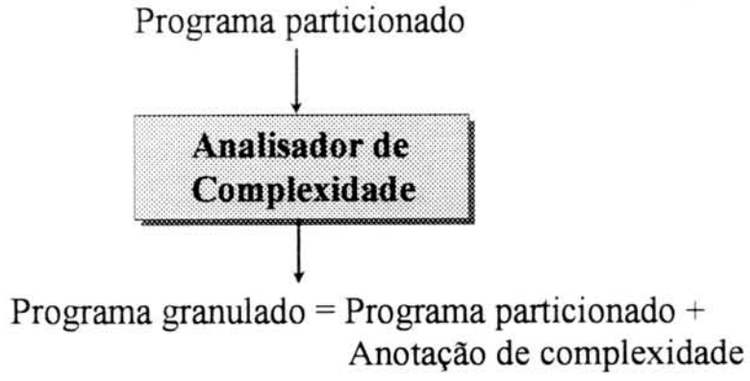


FIGURA 6.1 - Visão geral do módulo Analisador de Complexidade

Na figura 6.2 é apresentada a estrutura interna do módulo AC. Internamente, o módulo AC possui três submódulos, ou seja: **Analisador de Complexidade E** (ACE), **Analisador de Complexidade OU** (ACO) e **Gerador de Anotação de Complexidade** (GAC). Conforme será discutido na próxima seção, o GRANLOG utiliza dois tipos de complexidade para a programação em lógica, ou seja, a complexidade E e a complexidade OU. O submódulo ACE realiza a análise de complexidade E, gerando informações para o submódulo GAC. Por sua vez, o submódulo ACO realiza a análise de complexidade OU, fornecendo informações deste tipo de complexidade para o submódulo GAC. Finalmente, o submódulo GAC cria a anotação de complexidade, gerando assim o programa granulado. As próximas seções deste capítulo discutem os tipos de complexidade utilizados pelo modelo (seção 6.3), a análise de complexidade E (seção 6.4) e OU (seção 6.5) no âmbito do GRANLOG e a anotação de complexidade (seção 6.6) gerada pelo módulo AC.

6.3 Tipos de Complexidade Propostas pelo GRANLOG

Conforme discutido na subseção 2.1.4 existem diversos paradigmas de programação para desenvolvimento de sistemas computacionais. Além disso, a seção 6.1 ressalta que dependendo do paradigma a ser utilizado, adota-se diferentes metodologias para realização da análise de complexidade. Cada paradigma possui características

próprias, as quais devem ser consideradas para realização da análise e determinação do tipo de informação relevante para o usuário. No caso da programação em lógica, constata-se que o estudo da análise de complexidade deve considerar uma série de características específicas desse paradigma. Por exemplo, o não-determinismo é uma característica básica da programação em lógica e deve ser considerado na análise. Conforme discutido na subseção 2.2.1, a execução de programas em lógica é constituída por dois elementos, ou seja, **programa** e **avaliador**. Basicamente, a previsão do montante de recursos que será consumido pela execução de um programa depende da **estratégia de controle** implementada pelo avaliador. A subseção 2.2.3 discute as políticas de controle para programação em lógica e mais especificamente para a linguagem Prolog.

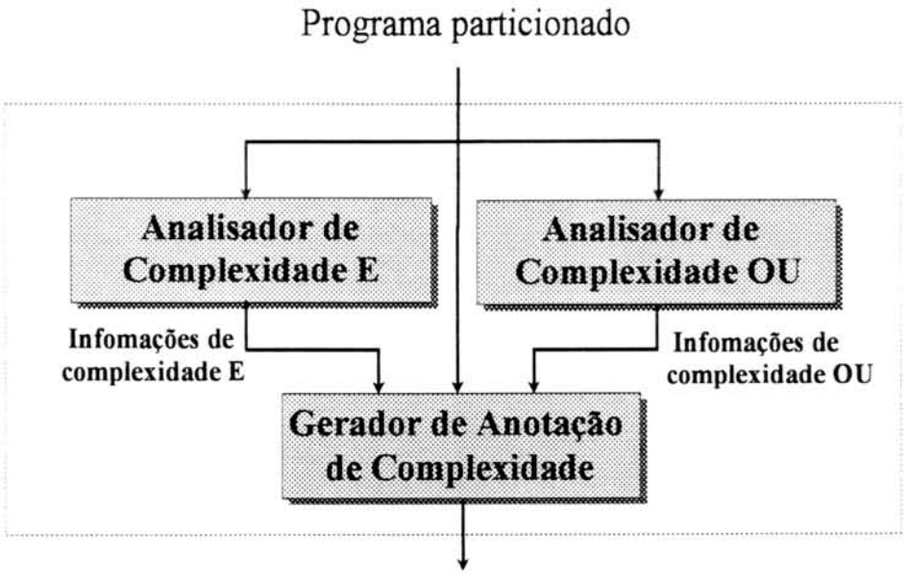


FIGURA 6.2 - Organização interna do módulo Analisador de Complexidade

Visando explorar ao máximo as características da programação em lógica, o GRANLOG utiliza dois tipos de complexidade, ou seja, **complexidade E** e **complexidade OU**. Essa taxonomia para complexidade de programas em lógica é proposta pelo GRANLOG e permite a classificação dos atuais trabalhos sobre complexidade em dois grupos, ou seja, **pesquisa sobre complexidade E** e **pesquisa sobre complexidade OU**.

Conforme discutido na subseção 2.2.3, a execução de programas em lógica pode ser visualizada como uma árvore de busca, gerada pela procura das diversas soluções. Além disso, a figura 2.7 mostra que para a execução de cada cláusula de um procedimento é gerado um caminho na árvore de busca (ramo da árvore), onde são solucionadas a cláusula e toda a **resolvente pendente**. A resolvente pendente é constituída pelas metas das chamadas anteriores que ainda não foram solucionadas. Na figura 2.7, a resolvente pendente, no momento da chamada para o procedimento **p**, é formada apenas pela meta **q**. No entanto, essa resolvente pode conter diversas metas acumuladas (ainda não executadas) durante as chamadas anteriores. Define-se **complexidade OU** como o montante de recursos computacionais consumidos durante a execução de um caminho da árvore de busca. Portanto, **análise de complexidade OU**

consiste na inferência de informações a respeito da complexidade OU, através da análise do texto do programa.

Deve-se destacar ainda, a relação existente entre complexidade OU, não-determinismo OU (subseção 2.2.8), grãos OU (seção 5.3) e paralelismo OU (subseção 2.3.1). Conforme discutido na subseção 2.3.1, o não-determinismo OU permite a exploração do paralelismo OU. Por sua vez, a seção 5.3 afirma que o paralelismo OU é explorado no GRANLOG através da criação dos grãos-OU. O conceito de complexidade não possui relação direta com o conceito de paralelismo, ou seja, o estudo da complexidade é completamente independente das pesquisas sobre paralelismo. No entanto, ambas linhas de pesquisa exploram as características da programação em lógica. Neste caso, o não-determinismo OU é uma característica da programação em lógica explorada tanto pelos estudos sobre paralelismo (paralelismo OU) quanto pelos estudos sobre complexidade (complexidade OU). No entanto, conforme destaca a seção 6.1, as informações de complexidade são de grande valia para exploração do paralelismo, o qual torna-se cada vez mais uma das suas principais aplicações. Sendo assim, pode-se criar uma relação entre os termos não-determinismo OU, grão-OU, paralelismo OU e complexidade OU. Esta relação permite que defina-se **granulosidade OU** como a complexidade de um grão-OU e **análise de granulosidade OU** como a análise de granulosidade (definida na subseção 2.1.2) relacionada com a exploração do paralelismo OU.

Considerando as características da programação em lógica, nota-se que além de informações sobre os recursos consumidos na execução de um caminho completo da árvore de busca (complexidade OU), torna-se relevante a obtenção de informações sobre a complexidade para execução de partes do corpo das cláusulas, em especial a complexidade para execução de metas (procedimentos). Sendo assim, o GRANLOG propõe a criação do segundo tipo de complexidade, ou seja, a complexidade E. Define-se **complexidade E** como o montante de recursos computacionais consumidos durante a execução de uma parte do corpo de uma cláusula. Desta forma, **análise de complexidade E** consiste na inferência de informações sobre a complexidade E, através da análise do programa fonte. Conforme discutido na subseção 2.2.5, grande parte do trabalho realizado pelos programas em lógica resulta da exploração da recursividade. Desta forma, grande parte da análise de complexidade E consiste em dimensionar o número de chamadas recursivas realizadas durante a execução de um procedimento.

Na figura 2.7 pode-se visualizar a complexidade E como a complexidade para execução de apenas uma das metas de um dos caminhos da árvore de busca. Por exemplo, no caminho 1 pode-se determinar a complexidade para execução da meta ***a***. A análise de complexidade E não considera resolventes pendentes, determinando apenas a complexidade para execução do procedimento ***a***. Conforme a definição de complexidade E, esta análise pode considerar a complexidade para execução de partes do corpo da cláusula (mais de uma meta). Por exemplo, na figura 2.7 a análise de complexidade pode deteminar a complexidade para execução das metas **a** e **b** do caminho 1.

Seguindo o mesmo raciocínio apresentado na discussão sobre a complexidade OU, pode-se criar uma relação entre os termos não-determinismo E (subseção 2.2.8), grão-E (seção 5.3), paralelismo E (subseção 2.3.1) e complexidade E. O não-determinismo-E é explorado em estudos independentes sobre complexidade-E e paralelismo E. Apesar desta independência, as informações sobre complexidade E podem ser aplicadas na busca da máxima eficiência na exploração do paralelismo E. Sendo assim, torna-se interessante

definir **granulosidade E** como a complexidade de um grão-E e **análise de granulosidade E** como a análise de granulosidade (subseção 2.1.2) vinculada com a exploração do paralelismo E.

A utilização de dois tipos de complexidade permite a geração de diferentes informações, ou seja, **informações de complexidade E** e **informações de complexidade OU**. No GRANLOG, esta situação pode ser visualizada na figura 6.2. Estes dois tipos de informações de complexidade podem ser utilizadas em diferentes aplicações ou ainda, para exploração de diferentes aspectos da mesma aplicação. No texto [LIN93] são apresentadas duas aplicações para as informações obtidas com a análise de complexidade E, ou seja, análise de granulosidade E e otimização de programas. Por sua vez, a análise de complexidade OU pode ser aplicada na análise de granulosidade OU. Ambos os tipos de informações podem ser utilizadas para simulação de programas em lógica (veja subseção 3.5.2).

Atualmente, existem diversos trabalhos relacionados com a análise de complexidade E. Dentre esses trabalhos destaca-se o estudo desenvolvido por Lin e Debray ([DEB93], [LIN93]). Por outro lado, a análise de complexidade OU é um tema bastante recente, sobre o qual as pesquisas estão apenas iniciando. No contexto da exploração do paralelismo na programação em lógica, o texto [GAR94] apresenta uma discussão sobre a análise de complexidade OU e sua aplicação na análise de granulosidade. Analisando-se os trabalhos desenvolvidos sobre complexidade na programação em lógica, constata-se que os esforços iniciais dos grupos de pesquisa foram direcionados para o estudo da complexidade E. Aparentemente, esta situação possui três causas principais, as quais permitem uma interessante discussão sobre caraterísticas da complexidade OU e complexidade E.

Em primeiro lugar, existe uma **grande semelhança entre as metodologias para análise de complexidade E e as metodologias para análise de complexidade em programas imperativos**. A análise de complexidade de um procedimento, tanto na programação em lógica (semântica procedimental, veja subseção 2.2.2) quanto na programação imperativa, consiste basicamente na análise do código fonte, considerando-se principalmente as chamadas de procedimentos (na programação em lógica as chamadas recursivas assumem grande importância, veja subseção 2.2.5). Por outro lado, a análise de complexidade OU deve considerar uma característica específica da programação em lógica, ou seja, as resolventes pendentes que devem ser solucionadas num caminho da árvore de busca. Esta característica deve ser abordada nos estudos específicos sobre complexidade na programação em lógica. Considerando-se ainda que os resultados obtidos nas pesquisas sobre complexidade em linguagens imperativas serviram de base para os estudos sobre complexidade na programação em lógica, pode-se concluir que a adoção inicial da complexidade E segue o caminho natural de aproveitamento de pesquisas já existentes.

Em segundo lugar, a análise de complexidade E é um problema simples quando comparado com a análise de complexidade OU. Basicamente, a complexidade de um procedimento é função da sua codificação e de suas entradas. A codificação pode ser analisada em tempo de compilação e as entradas devem ser obtidas durante a execução. Neste caso, pode-se prever com precisão através da análise estática o comportamento do procedimento em função de suas entradas, resolvendo-se de forma estática, grande parte da análise de complexidade. Portanto, **o objeto de estudo (procedimento) da análise de complexidade E é conhecido em tempo de compilação**. Por outro lado, a principal

componente da complexidade OU são as resolventes pendentes. Estas resolventes variam dinamicamente de acordo com a execução do programa. Cada execução pode gerar uma árvore de busca diferente. Sendo assim, cada execução pode gerar uma árvore com diferentes ramos. Portanto, o **objeto de estudo (ramo da árvore de busca) da análise de complexidade OU é criado em tempo de execução**. A dificuldade para obtenção estática de informações sobre os ramos da árvore de busca faz com que a análise de complexidade OU torne-se bastante complexa, quando comparada com a análise de complexidade E.

Em terceiro lugar, a complexidade para execução de um ramo da árvore de busca é constituída basicamente das complexidades de cada uma das metas existentes na cláusula que origina o ramo e nas resolventes pendentes. Deve-se considerar ainda, que a complexidade de metas é objeto de estudo da análise de complexidade E. Sendo assim, pode-se concluir que **as pesquisas sobre análise de complexidade OU dependem em grande parte dos resultados obtidos nas pesquisas sobre complexidade E**. Como exemplo desse fato, pode-se citar a evolução das pesquisas desenvolvidas por Debray e Hermenegildo. Após os resultados animadores obtidos nos estudos sobre a complexidade E ([LIN93], [DEB93]), surgiram esforços direcionados para o estudo da complexidade OU ([GAR94]).

6.4 Análise de Complexidade OU

Conforme definido na seção anterior, a **análise de complexidade OU** consiste na inferência do montante de recursos computacionais consumidos durante a execução de um caminho da árvore de busca. Esta inferência é realizada através da análise do texto do programa. A análise de complexidade OU é um tema de pesquisa recente, abordado atualmente por poucos pesquisadores. A publicação [GAR94] é um dos raros textos que abordam a complexidade OU. Nesta publicação são apresentadas algumas considerações sobre uma proposta para realização deste tipo de análise de complexidade. Deve-se ressaltar ainda, que a análise de complexidade OU serve de base para realização da análise de granulosidade OU. A máxima eficiência na exploração do paralelismo OU pode ser obtida através de considerações sobre complexidade e custo de paralelização.

A complexidade OU possui duas componentes, ou seja, a complexidade da cláusula que cria o ramo da árvore de busca e a complexidade da resolvente pendente no momento da criação do ramo. A complexidade da cláusula pode ser obtida com precisão aplicando-se técnicas já pesquisadas para inferência de complexidade E. Estas técnicas permitem a determinação da complexidade de execução de procedimentos. Considerando-se uma cláusula como um procedimento único (mesmo que na realidade não seja), pode-se inferir sua complexidade. O trabalho apresentado em [LIN93] pode ser adaptado para obtenção precisa desta informação. No entanto, a determinação precisa da complexidade da resolvente pendente é um problema de difícil solução. Conforme discutido na seção 6.3, a resolvente pendente somente será conhecida durante a execução do programa em lógica e portanto a determinação estática de sua complexidade torna-se bastante difícil. Atualmente, os pesquisadores buscam soluções para o **problema complexidade da resolvente pendente**. O autor dessa dissertação acredita que a principal dificuldade para analisar a complexidade OU consiste em obter informações **precisas** sobre a complexidade da resolvente pendente sem introduzir considerável custo computacional (*overhead*) na execução dos programas. Além disso, na medida em que são direcionados esforços para prever estaticamente a complexidade

OU, aumenta a complexidade das metodologias, pois é bastante difícil prever em tempo de compilação um comportamento tão dinâmico quanto a criação da árvore de busca durante a execução de programas em lógica.

O GRANLOG propõe uma nova abordagem para análise de complexidade OU. Nessa abordagem o problema complexidade da resolvente pendente é tratado através de uma negociação entre precisão, complexidade da metodologia e custo introduzido na execução. Basicamente, a proposta prioriza o baixo custo adicional na execução e a simplicidade na metodologia em detrimento da precisão das informações obtidas na análise. O GRANLOG utiliza como base para análise de complexidade OU a adaptação de uma proposta de Evan Tick ([TIC88], [TIC90]) para análise de complexidade E. Essa proposta determina **completamente** em tempo de compilação a complexidade dos procedimentos de um programa em lógica. Esta análise estática definitiva é baseada em simplificações que abstraem as possíveis variações dinâmicas da complexidade dos procedimentos. A principal simplificação consiste em desprezar as chamadas recursivas, as quais fazem com que a complexidade dos procedimentos dependam das suas entradas (conhecidas apenas em tempo de execução). Essas simplificações sacrificam a precisão das informações geradas pela análise, mas no entanto, criam uma metodologia simples e que após a adaptação proposta pelo GRANLOG, viabiliza a análise de complexidade OU. O texto [BAR94] descreve e analisa o método proposto por Tick. Destaca-se em [BAR94], uma comparação da proposta de Tick com outras metodologias. Além disso, em [SCH93] encontra-se a descrição de uma implementação da proposta de Tick. Essa implementação é utilizada na construção do submódulo Analisador de Complexidade OU (figura 6.2).

A proposta de Tick pode ser descrita pelas seguintes regras:

❶ a complexidade de uma cláusula é a soma da complexidade dos procedimentos chamados no seu corpo;

❷ a complexidade de um procedimento é a média da complexidade de suas cláusulas;

❸ a complexidade de uma chamada recursiva possui valor 1 por definição.

A terceira regra simplifica a análise de complexidade, permitindo a obtenção dos resultados definitivos em tempo de compilação. No entanto, conforma mostra a subseção 2.2.5, a recursividade é a principal fonte de complexidade dos programas em lógica. Portanto, a simplificação introduzida pela terceira regra gera imprecisão nos resultados da análise. Os textos [TIC88], [SCH93] e [BAR94] exemplificam a aplicação das três regras na determinação da complexidade de procedimentos num programa em lógica.

O GRANLOG adapta o modelo de Tick visando três objetivos básicos, ou seja: direcionar para a complexidade OU as informações geradas pela análise, compatibilizar essas informações com o modelo para análise de complexidade E utilizado no GRANLOG (seção 6.5) e aumentar a precisão da análise. As seguintes regras são utilizadas para análise de complexidade OU no GRANLOG:

❶ a unidade básica de medida de complexidade é o número de resoluções (chamadas de procedimentos) executadas numa chamada de procedimento. Na proposta de Tick, a unidade de medida é o número de chamadas recursivas executadas por um procedimento (terceira regra). Esta adaptação visa a compatibilidade com a análise de complexidade E descrita na seção 6.5, a qual também utiliza como unidade de medida o número de resoluções;

❷ a complexidade de uma regra é a soma da complexidade dos procedimentos chamados no seu corpo, acrescido de 1 (resolução da cabeça da regra). Portanto, a complexidade de um fato é 1. Na abordagem de Tick, um fato possui complexidade 0. Esta adaptação compatibiliza a análise de complexidade OU com a análise de complexidade E descrita na seção 6.5, a qual segue as mesmas regras. Além disso, computar os fatos na composição da complexidade aumenta a precisão das informações;

❸ a complexidade de um procedimento é a soma da complexidade das cláusulas que o compõem. Conforme discutido na seção 6.1, existem três tipos de análise de complexidade, ou seja, análise do pior caso, análise do melhor caso e análise do caso médio. A segunda regra do modelo de Tick estabelece a utilização de uma análise simplificada do caso médio. Visando compatibilizar as análises de complexidade OU e E, o GRANLOG adapta o modelo para realização da análise do pior caso. A complexidade máxima é obtida com a soma das complexidades das cláusulas componentes do procedimento;

❹ a complexidade de uma chamada recursiva é obtida pela soma da complexidade das cláusulas do procedimento, atribuindo-se valor 1 para as chamadas recursivas. A complexidade de um procedimento que possui chamadas recursivas é obtida após a determinação da complexidade dessas chamadas. Esta abordagem simplificada, determina completamente em tempo de compilação a complexidade dos procedimentos, pois a complexidade da chamadas recursivas fica resolvida de forma estática. As chamadas recursivas constituem a principal fonte de complexidade dos programas em lógica (subseção 2.2.5) e sofrem forte influência dos dados processados pelo programa. Portanto, a abordagem proposta pelo GRANLOG introduz imprecisão na análise. Na proposta de Tick, uma chamada recursiva possui valor 1 (terceira regra). No GRANLOG, um fato possui valor 1 (uma resolução para sua solução) e uma chamada recursiva possui valores mais significativos, compatíveis com a complexidade do procedimento no qual estão inseridos;

❺ conforme discutido em [TIC93], os procedimentos mutuamente recursivos são combinados em *clusters*. Todos os procedimentos pertencentes a um *cluster* possuem a mesma complexidade. Esta complexidade é obtida com a soma da complexidade dos procedimentos do *cluster*. As chamadas mutuamente recursivas recebem valor zero.

A figura 6.3 exemplifica a aplicação das regras utilizadas pelo GRANLOG. Esta figura mostra uma programa em lógica simplificado, contendo apenas os *functores* dos procedimentos. A figura mostra ainda, o cálculo da complexidade de cada cláusula e de cada procedimento. O mesmo programa é utilizado em [TIC88], [SCH93] e [BAR94] para exemplificar o modelo proposto por Evan Tick. Sendo assim, aconselha-se para maior esclarecimento sobre ambas as propostas, uma comparação entre os resultados mostrados nos três textos e os resultados apresentados na figura 6.3.

Os resultados apresentados na figura 6.3 foram obtidos com a aplicação das regras propostas pelo GRANLOG. Por exempo, a obtenção da complexidade do procedimento ***d*** (***Cd***) depende da determinação da complexidade de suas duas cláusulas através da regra 3. A primeira cláusula é um fato. Portanto, segundo a regra 2, sua complexidade (***Cd1***) é 1. A segunda cláusula é uma regra. Sendo assim, conforme determina a regra 2, sua complexidade (***Cd2***) é a soma das complexidades das chamadas do seu corpo

acrescida da complexidade da cabeça. A complexidade da cabeça vale 1 (equivalente a um fato). Além disso, encontra-se no corpo da regra uma chamada recursiva. Neste caso, aplica-se a regra 4 e determina-se que a complexidade da chamada recursiva (***Cdr***) é 3. Portanto, a complexidade da segunda cláusula (***Cd2***) é 4. Finalmente, somando-se a complexidade da primeira cláusula (***Cd1***) com a complexidade da segunda (***Cd2***), obtém-se a complexidade do procedimento ***d*** (***Cd***), ou seja, 5 resoluções. As complexidades dos procedimentos ***c*** e ***e*** são obtidas da mesma forma. Por outro lado, a complexidade dos procedimentos ***a*** e ***b*** depende da aplicação da regra 5 para tratamento de chamadas mutuamente recursivas. A figura 6.3 mostra que as complexidades das duas chamadas mutuamente recursivas (***Cam*** e ***Cbm***) recebem valor 0. A complexidade da chamada recursiva na segunda cláusula do procedimento ***a*** (***Car***) recebe o tratamento descrito na regra 4. Após a obtenção da complexidade de cada cláusula dos procedimentos ***a*** e ***b***, pode-se determinar a complexidade parcial destes procedimentos (***Cap*** e ***Cbp***). O procedimento ***a*** possui complexidade parcial igual a 195 e o procedimento ***b*** igual a 7. Finalmente, aplicando-se a regra 5, obtém-se o valor final da complexidade dos procedimentos mutuamente recursivos (***Ca*** e ***Cb***). Somando-se os dois valores parciais, obtém-se o valor de 202 resoluções como complexidade final de ambos procedimentos.

```
a.                Ca1=1
a :- b,a,c.       Ca2=1+Cbm+Car+Cc=1+0+98+73=172     Cap=Ca1+Ca2+Ca3+Ca4=195
a :- d,e.         Ca3=1+Cd+Ce=1+5+15=21
a.                Ca4=1

b :- d,a.         Cb1=1+Cd+Cam=1+5+0=6               Cbp=Cb1+Cb2=7
b.                Cb2=1
                                                     Ca=Cb=Cap+Cbp=195+7=202
c :- d,d,d,c.     Cc1=1+Cd+Cd+Cd+Ccr=1+5+5+5+25=41
c :- d,c.         Cc2=1+Cd+Ccr=1+5+25=31             Cc=Cc1+Cc2+Cc3=73
c.                Cc3=1

d.                Cd1=1                              Cd=Cd1+Cd2=5
d :- d.           Cd2=1+Cdr=1+3=4

e :- d.           Ce1=1+Cd=1+5=6                     Ce=Ce1+Ce2=15
e :- e.           Ce2=1+Cer=1+8=9
```

FIGURA 6.3 - Resultados do modelo de Tick após adaptações do GRANLOG

O exemplo demonstra que o método resulta em valores constantes para a complexidade dos procedimentos. Esta simplificação viabiliza a análise de complexidade OU, conforme será discutido nos próximos parágrafos. No entanto, a abordagem simplificada introduz imprecisão nos resultados, pois a verdadeira complexidade dos procedimentos somente poderá ser determinada em tempo de execução. Esta complexidade depende de valores disponíveis apenas durante a execução e varia de acordo com os dados processados pelo programa. A principal fonte de imprecisão encontra-se no tratamento das chamadas recursivas. Por exemplo, a complexidade obtida para o procedimento ***d*** é 5 resoluções. No entanto, a complexidade real desse procedimento certamente dependerá dos seus argumentos de entrada, os quais determinarão o número de chamadas recursivas a serem realizadas. Apesar da imprecisão, o método utilizado pelo GRANLOG permite o controle da granulosidade OU e aumenta a precisão da abordagem proposta por Evan Tick. Naquela abordagem, todas as chamadas recursivas possuem valor 1. Na adaptação proposta pelo GRANLOG,

a complexidade das chamadas recursivas depende das complexidades do restante do procedimento, o que representa uma realidade. Além disso, a proposta de Evan Tick despreza a influência dos fatos na complexidade dos procedimentos. O GRANLOG utiliza como unidade de medida a resolução. Portanto, um fato possui complexidade 1.

Conforme discutido na seção 6.3, a complexidade OU mede a complexidade de um ramo da árvore de busca. Durante a resolução de um procedimento é criado um ramo para cada cláusula. Além disso, um ramo é composto da cláusula e da resolvente pendente. Sendo assim, o GRANLOG gera duas informações a serem utilizadas na análise de complexidade OU, ou seja, **complexidade das cláusulas** (CC) e **complexidade das resolventes pendentes locais** (CRPL). A figura 6.2 mostra estas informações sendo geradas pelo submódulo Analisador de Complexidade OU e consumidas pelo submódulo Gerador de Anotação de Complexidade (GAC). O submódulo GAC introduz na anotação de complexidade ambas informações, as quais ficam disponíveis no programa granulado e podem ser utilizadas para realização da análise de granulosidade OU. A informação CC consiste da complexidade de cada cláusula do programa. No exemplo apresentado na figura 6.3, a informação CC consiste da complexidade de todas as cláusulas do programa. Por exemplo, a complexidade da primeira cláusula do procedimento ***c*** (***Cc1***) é 41. Define-se **resolvente pendente local** (RPL) como todas as chamadas que sucedem uma determinada chamada no corpo de uma regra. Portanto, RPLs somente existem em regras e estão ligadas a uma determinada chamada. Sendo assim, a CRPL consiste da complexidade de cada resolvente pendente local existente no programa. Por exemplo, na figura 6.3 a CRPL da primeira chamada do corpo da segunda cláusula do procedimento ***a*** (chamada para o procedimento ***b***) é 171, ou seja, a soma de ***Car*** e ***Cc***.

A seção 6.3 destaca que o objeto de estudo (ramo da árvore de busca) da análise de complexidade OU é criado em tempo de execução. Sendo assim, a complexidade OU somente pode ser determinada durante a execução. A principal dificuldade consiste na determinação da complexidade da resolvente pendente no momento da chamada de um procedimento, pois esta resolvente depende dos caminhos já percorridos durante a execução do programa. Estes caminhos variam consideravelmente entre diferentes execuções do mesmo programa em lógica. Visando solucionar este problema, o GRANLOG propõe uma abordagem combinando análise estática e dinâmica, ou seja, em tempo de compilação são obtidas informações de complexidade OU e em tempo de execução estas informações são utilizadas para determinação da complexidade dos ramos da árvore de busca.

A complexidade OU é obtida pela soma da cláusula que origina o ramo e da resolvente pendente no momento da chamada do procedimento. Esta soma deve ser realizada em tempo de execução no momento em que houver interesse pela complexidade de um ramo da árvore. A complexidade das cláusulas do programa em lógica é disponibilizada através da informação CC gerada pela análise descrita nos parágrafos anteriores. Por outro lado, a complexidade da resolvente pendente depende dos caminhos percorridos durante a execução do programa. Cada chamada durante a execução, torna pendente sua RPL. Sendo assim, deve-se criar um **acumulador de CRPLs** (ACRPL) que será atualizado a cada chamada. Quando for realizada uma chamada, o ACRPL deve ser acrescido da CRPL da chamada. Portanto, o ACRPL mantém a cada instante da execução o valor da resolvente pendente. A qualquer momento durante a execução, a complexidade de um ramo da árvore de busca pode ser determinada pela soma da complexidade da cláusula com o ACRPL. Deve-se

compreender ainda, que o ACRPL está relacionado com uma execução completa de um procedimento, ou seja, cada chamada no procedimento deve utilizar o mesmo valor do ACRPL. Qualquer chamada no procedimento deve atualizar o ACRPL, somando o valor do ACRPL do procedimento com o valor da CRPL da chamada. Resumindo, cada execução de procedimento possui um valor próprio para o ACRPL. Por exemplo, na figura 6.3 o procedimento ***a*** é composto por quatro cláusulas. Neste caso, devem ser criados quatro ramos na árvore de busca para resolver o programa. Se esse procedimento for chamado sem resolventes pendentes, o ACRPL para execução de **todo** o procedimento ***a*** possui valor 0. Sendo assim, a complexidade do primeiro ramo é 1 (ACRPL+Ca1), a complexidade do segundo ramo é 172 (ACRPL+Ca2), a complexidade do terceiro ramo é 21 (ACRPL+Ca3) e do último 1 (ACRPL+Ca4). Dando prosseguimento ao exemplo, a solução da segunda cláusula necessita chamar três procedimentos. No momento da chamada do primeiro (procedimento ***b***), deve-se somar o valor atual do ACRPL (zero) com a CRPL relacionada com a chamada de ***b***. Esta CRPL possui valor 171 (soma de ***Car*** e ***Cc***). Portanto, o ACRPL possui valor 171. No momento da chamada do segundo procedimento (procedimento ***a***), deve-se somar o mesmo valor do ACRPL (zero) com a CRPL relacionada com a chamada de ***a***. Esta CRPL possui valor 73 (valor de ***Cc***). Portanto, o ACRPL possui valor 73. Nota-se que o mesmo ACRPL é utilizado para ambas as chamadas. Sendo assim, deve-se implementar um controle para gerenciar os valores dos ACRPLs relacionados com cada procedimento em execução. Por exemplo, no retorno do procedimento ***b*** o valor do ACRPL local deve estar intacto para ser utilizado pela chamada de ***a***.

A execução de ***b*** cria dois novos ramos na árvore. O primeiro ramo possui complexidade 177, resultante da soma do ACRPL atual com a complexidade 6 da primeira cláusula (***Cb1***). O segundo ramo possui complexidade 172, resultante da soma ACRPL+***Cb2***. O valor do ACRPL deve ser atualizado continuamente, permitindo assim, a determinação da complexidade de um ramo da árvore de busca a qualquer momento durante a execução do programa em lógica.

A seção 6.6 discute a anotação de complexidade gerada pelo módulo Analisador de Complexidade. Nesta anotação constam as informações de complexidade OU (CC e CRPL) geradas pelo submódulo ACO. Sendo assim, a discussão sobre como as informações de complexidade OU são anotadas para posterior utilização será postergada até a seção 6.6.

6.5 Análise de Complexidade E

A seção 6.3 ressalta que a análise de complexidade E dedica-se à previsão do montante de recursos computacionais que serão consumidos durante a execução de **parte do corpo de uma cláusula**. Além disso, a mesma seção destaca que a comunidade científica tem dedicado nos últimos anos consideráveis esforços de pesquisa para desenvolvimento e aplicação da análise de complexidade E. Devido a esses esforços, atualmente existem diversos trabalhos concluidos que alcançaram resultados satisfatórios. É o caso da tese de doutorado de Nai-Wei Lin ([LIN93]). O texto [LIN93] cita e discute vários trabalhos e pode ser utilizado como fonte de referências para aprofundamento dos estudos sobre análise de complexidade E. Em [DEB93] encontra-se um resumo do trabalho de Lin. Alguns textos mais recentes abordam especificamente a análise de complexidade E ([DEB94a]) e sua relação com a análise de granulosidade ([GAR94]).

Conforme mostra a figura 6.2, no GRANLOG a análise de complexidade E é realizada pelo submódulo Analisador de Complexidade E, o qual recebe o programa particionado e gera informações de complexidade. O submódulo ACE possui como base o sistema proposto em [DEB93] e [LIN93], o qual é denominado CASLOG (Complexity Analysis System for LOGic). O CASLOG realiza uma análise estática de um programa em lógica e produz para cada procedimento uma expressão de complexidade. Esta expressão representa a complexidade do procedimento em função do tamanho de suas entradas. Essas expressões podem ser resolvidas durante a execução para determinar com precisão a complexidade de cada procedimento do programa. Além disso, deve-se ressaltar que o CASLOG realiza uma análise de complexidade do pior caso, considerando o não-determinismo dos programas em lógica e o número de possíveis soluções que podem ser geradas durante a execução. Aconselha-se a leitura dos textos [DEB93] e [LIN93] para aprofundamento dos estudos sobre o CASLOG.

A figura 6.4 apresenta as expressões de complexidade geradas pelo CASLOG para os procedimentos *append* (figura 2.5), *nrev* (figura 2.2), *fibonacci* (figura 4.3) e *hanoi* (figura 4.10). No CASLOG, a geração de expressões de complexidade depende dos modos e medidas atribuidos aos argumentos dos procedimentos. No apêndice C do texto [LIN93], encontra-se os modos e tipos utilizados para obtenção das expressões mostradas na figura 6.4. O apêndice apresenta ainda vários outros exemplos de análise de complexidade na programação em lógica.

$$C_{append} = \$1 + 1$$
$$C_{nrev} = 0.5 * exp(\$1,2) + 1.5 * \$1 + 1$$
$$C_{fibonacci} = 1.45 * exp(1.62,\$1) + 0.55 * exp(-0.62,\$1) - 1$$
$$C_{hanoi} = \$1 * exp(2,\$1) + exp(2,\$1\text{-}1) - 2$$

FIGURA 6.4 - Expressões de Complexidade

A notação das expressões utiliza operadores matemáticos (*,/,+,-), funções matemáticas (por exemplo, *exp(a,b)* significa a elevado na potência b) e o símbolo *$* significando o tamanho de determinado argumento do procedimento (*$1* por exemplo, significa o tamanho do primeiro argumento). As complexidades dos quatro procedimentos utilizados no exemplo da figura 6.4 dependem apenas do tamanho primeiro argumento (*$1*). O tamanho de um argumento de entrada depende da medida a ser utilizada na sua mensuração. A subseção 4.5.2 discute as medidas de tamanho de argumentos utilizadas no GRANLOG. Além disso, a seção 5.5 mostra que a anotação de grãos gerada pelo submódulo Analisador de Grãos contém as medidas a serem utilizadas para mensurar o tamanho dos argumentos. Desta forma, existe uma relação direta entre as medidas de tamanho de argumentos geradas pela análise de grãos e as expressões de complexidade geradas pela análise de complexidade E. O tamanho dos argumentos de entrada pode ser deteminado com precisão em tempo de execução. No entanto, a mensuração do tamanho dos argumentos durante a execução ocasiona custos que contribuem para perda de desempenho. Soluções para esse problema vêm sendo pesquisadas. Em [HER94] é apresentada uma interessante discussão desse problema e uma proposta para sua solução.

Atualmente, as expressões de complexidade geradas pelo CASLOG utilizam como medida de complexidade o número de resoluções, ou seja, a solução de uma expressão

resulta no número de resoluções que serão realizadas durante a execução de um procedimento. A mesma medida foi utilizada pelo GRANLOG na análise de complexidade OU. Sendo assim, existe uma compatibilidade entre os resultados fornecidos pelas análises de complexidade E e OU. Esta compatibilidade é vital para futuras pesquisadas envolvendo a análise E e a análise OU na exploração do paralelismo E/OU na programação em lógica.

Além das expressões de complexidade, o CASLOG gera relações para determinação do tamanho das saídas de um procedimento em função do tamanho de suas entradas. O GRANLOG propõe a utilização dessas relações para previsão dos custos de comunicação na execução paralela de programas em lógica em sistemas com memória distribuída. A previsão do tamanho das saídas permite o dimensionamento do tamanho das mensagens necessárias para comunicação dos resultados da execução paralela dos procedimentos. Saumya Debray ([DEB95a]) concorda com a proposta do GRANLOG e acredita que as relações geradas pelo CASLOG podem ser utilizadas para avaliação de custos de comunicação em sistemas distribuídos. Argumentando nesse sentido, Debray apoia outra proposta do GRANLOG, ou seja, a criação de expressões para determinação direta dos custos de comunicação na execução paralela de procedimentos em ambientes distribuídos. Estas expressões seriam semelhantes às expressões de complexidade e permitiriam a determinação dos custos de comunicação em função do tamanho dos argumentos de entrada. A utilização conjunta dessas expressões e das expressões de complexidade permitirá uma análise de granulosidade em ambientes distribuídos, permitindo uma negociação entre complexidade e custo. A previsão do tamanho das saídas dos procedimentos pode ser utilizada ainda na simulação da execução de programas em lógica. A subseção 3.5.2 discute esta aplicação do GRANLOG.

A figura 6.5 mostra as relações de tamanho entre entradas e saídas, geradas pelo CASLOG para os procedimentos *append*, *nrev*, *fibonacci* e *hanoi*. O apêndice C de [LIN93] apresenta uma série de exemplos de relações, dentre as quais encontram-se as quatro mostradas na figura 6.5.

$$R_{append} = [\$1,\$2,\$1+\$2]$$
$$R_{nrev} = [\$1,\$1]$$
$$R_{fibonacci} = [\$1,0.45*\exp(1.62,\$1)-0.45*\exp(-0.62,\$1)]$$
$$R_{hanoi} = [\$1,0,0,0,\exp(2,\$1)-1]$$

FIGURA 6.5 - Relações de tamanho entre entradas e saídas

As relações utilizam uma notação semelhante às expressões de complexidade. O símbolo $ possui a mesma função. Além disso, também são utilizados operadores e funções matemáticas. No entanto, a relação é construída na forma de uma lista, onde cada elemento corresponde a um argumento do procedimento. Se o argumento é de entrada e influencia no tamanho das saídas, o CASLOG coloca o símbolo *$* acompanhado da posição do argumento. Se o argumento é de entrada e não influencia no tamanho das saídas, o CASLOG coloca um símbolo nulo (0). Se o argumento for de saída, o CASLOG coloca a expressão utilizada para calcular seu tamanho em função do tamanho dos argumentos de entrada. Nas relações apresentadas na figura 6.5, os dois primeiros argumentos do procedimento *append* são entrada e o último saída. Portanto, o tamanho do argumento de saída é calculado pela soma dos dois argumentos de entrada. No caso do procedimento *nrev*, o primeiro argumento é entrada e o segundo saída.

Desta forma, o tamanho do argumento de saída é igual ao tamanho do argumento de entrada. As relações para os procedimentos *fibonacci* e *hanoi* seguem o mesmo raciocínio. Da mesma forma que as expressões de complexidade, as relações de tamanho entre entradas e saídas possuem uma relação direta com as medidas de tamanho de argumentos discutidas na subseção 4.5.2. Naquela subseção, destaca-se que as medidas de tamanho dos argumentos possuem duas aplicações, ou seja, **determinação da granulosidade (DGR)** e **determinação do tamanho das saídas (DTS)**. A DGR é concretizada através da expressões de complexidade e a DTS é implementada pelas relações de tamanho entre entradas e saídas dos procedimentos. Além disso, a combinação das informações de tipos obtidas pela análise global (subseção 4.5.1) com as relações obtidas pela análise de complexidade permitem um análise completa do fluxo de dados entre procedimentos. O exemplo apresentado na seção 6.7 demonstra a combinação das informações de modos, medidas e tipos (análise global) com as expressões de complexidade e relações de tamanho entre entradas e saídas (análise de complexidade).

O CASLOG dedica-se à determinação da complexidade dos procedimentos existentes num programa em lógica. No entanto, conforme discutido na seção 6.3, o GRANLOG define complexidade E como o montante de recursos computacionais consumidos durante a execução de uma **parte do corpo de uma cláusula**. Portanto, a definição proposta pelo GRANLOG amplia a abordagem do CASLOG e permite a análise da complexidade de execução de grupos de metas do corpo de uma cláusula, ou seja, grupos de procedimentos. Esta característica é vital para exploração do paralelismo E, conforme definido na seção 5.3. Em complemento, sabe-se que uma cláusula é um grupo de metas e portanto, utilizando a nova definição proposta pelo GRANLOG pode-se utilizar o CASLOG para inferir a complexidade de uma única cláusula do procedimento. Sendo assim, o CASLOG fornecerá ainda informações sobre a complexidade de cada cláusula do programa. Estas informações poderão ser utilizadas para aumentar a precisão da análise de complexidade OU (seção 6.4) e para comparação entre as abordagens propostas por Evan Tick ([TIC88]) e Nai-Wei Lin ([LIN93]). Conclui-se assim que o submódulo ACE, mostrado na figura 6.2, não é composto apenas pelo CASLOG. Este submódulo contém ainda filtros que adaptam o programa em lógica, fazendo com que o CASLOG realize a análise de complexidade de acordo com a nova definição proposta pelo GRANLOG. Esses filtros são responsáveis pela adaptação tanto das entradas quanto das saídas do CASLOG. A organização interna do submódulo ACE é discutida durante a apresentação do protótipo GRANLOG (capítulo 8). A seção 6.7 demonstra os novos resultados obtidos com a ampliação do CASLOG. A demonstração é baseada na análise de complexidade do programa *Torre de Hanói* realizada pelo GRANLOG.

Durante os estudos do CASLOG foram realizados diversos testes para avaliação do sistema e para validação dos seus resultados. O restante desta seção apresenta e discute os resultados obtidos no principal teste, realizado com o intuito de avaliar a precisão das informações de complexidade geradas pelo CASLOG. O teste foi realizado numa estação Sun SLC no laboratório do Instituto de Informática da UFRGS num horário de baixa utilização da rede (rede vazia). O teste consistiu nos seguintes passos:

❶ Submissão dos procedimentos *append*, *nrev*, *fibonacci* e *hanoi* à análise de complexidade do CASLOG. Esse passo gerou as expressões de complexidade apresentadas na figura 6.4;

❷ Execução dos quatro procedimentos em três ambientes diferentes, ou seja, C-Prolog versão 1.5+ ([PER87]), Sicstus Prolog versão 2.1 ([CAR92]) e emulador WAM seqüencial utilizado no projeto OPERA ([WER94], [YAM94]). Cada procedimento foi executado com vários tamanhos para o argumento de entrada, gerando assim, diversas previsões de complexidade e diversos tempos de execução. As tabelas 6.1, 6.2, 6.3 e 6.4 mostram os resultados obtidos nos passos ❷, ❸ e ❹. Cada tabela contém os resultados do teste de um procedimento. A primeira coluna contém o tamanho da entrada utilizada em cada execução (valor de *SI*). A terceira coluna contém o tempo da execução em milisegundos para o Sicstus, a quinta coluna para o C-Prolog e a sétima para o emulador WAM. As demais colunas são preenchidas pelos próximos passos. Os programas Prolog utilizados nesse passo são apresentados no anexo A;

❸ Resolução das expressões de complexidade para cada entrada (coluna 1) utilizada no passo ❷, ou seja, obter o número de resoluções previstas pelo CASLOG para cada execução. Este passo resulta na segunda coluna das quatro tabelas;

❹ Obtenção do tempo necessário para uma resolução na execução dos procedimentos em cada ambiente. Este valor é obtido com a divisão do tempo total de execução (coluna 3, 5 e 7) pelo número de resoluções previstas pelo CASLOG (coluna 2). Os resultados são apresentados nas colunas 4, 6 e 8;

❺ Obtenção do tempo médio de uma resolução para execução de um procedimento num determinado ambiente, ou seja, obtenção da média para cada uma das colunas 3, 5 e 7. As médias são apresentadas na última linha das tabelas 6.1, 6.2, 6.3 e 6.4.

TABELA 6.1 - Teste do CASLOG com procedimento *append*

APPEND							
TAM.	**RES.**	**SICSTUS**		**C-PROLOG**		**WAM**	
		Tempo (ms)	**Tempo/Res.**	**Tempo (ms)**	**Tempo/Res.**	**Tempo (ms)**	**Tempo/Res.**
10	11	0.95	0.09	0.83	0.08	4.00	0.36
20	21	1.45	0.07	2.50	0.12	7.00	0.33
30	31	3.00	0.10	3.33	0.11	10.00	0.32
40	41	3.45	0.08	5.00	0.12	16.00	0.39
50	51	3.95	0.08	6.66	0.13	19.00	0.37
60	61	4.95	0.08	7.50	0.12	22.00	0.36
70	71	5.45	0.08	8.33	0.12	25.00	0.35
80	81	6.50	0.08	9.17	0.11	29.00	0.36
90	91	6.95	0.08	10.00	0.11	33.00	0.36
100	101	8.45	0.08	11.67	0.12	36.00	0.36
1000	1001	82.00	0.08	121.67	0.12	349.00	0.35
Média Tempo/Resoluções			0.08		0.12		0.35

TABELA 6.2 - Teste do CASLOG com procedimento *nrev*

NREV							
TAM.	RES.	SICSTUS		C-PROLOG		WAM	
		Tempo (ms)	Tempo/Res.	Tempo (ms)	Tempo/Res.	Tempo (ms)	Tempo/Res.
10	66	6.50	0.10	7.50	0.11	27.00	0.41
20	231	20.50	0.09	24.17	0.10	87.00	0.38
30	496	41.50	0.08	56.67	0.11	185.00	0.38
40	861	71.00	0.08	101.67	0.12	313.00	0.36
50	1326	108.50	0.08	155.00	0.12	482.00	0.36
60	1891	155.95	0.08	220.83	0.12	685.00	0.36
70	2556	209.45	0.08	295.83	0.12	919.00	0.36
80	3321	271.95	0.08	391.67	0.12	1201.00	0.36
90	4186	341.50	0.08	493.33	0.12	1513.00	0.36
100	5151	420.95	0.08	605.00	0.12	1864.00	0.36
Média Tempo/Resoluções			0.08		0.12		0.36

TABELA 6.3 - Teste do CASLOG com procedimento *fibonacci*

FIBONACCI							
TAM.	RES.	SICSTUS		C-PROLOG		WAM	
		Tempo (ms)	Tempo/Res.	Tempo (ms)	Tempo/Res.	Tempo (ms)	Tempo/Res.
1	2	0.00	0.00	0.00	0.00	1.00	0.50
2	3	1.00	0.33	0.83	0.28	3.00	1.00
3	5	1.45	0.29	1.67	0.33	6.00	1.20
4	9	2.45	0.27	2.45	0.27	12.00	1.33
5	15	4.50	0.30	5.00	0.33	19.00	1.27
6	25	7.45	0.30	7.50	0.30	32.00	1.28
7	41	11.95	0.29	13.33	0.32	51.00	1.24
8	67	18.95	0.28	22.50	0.30	83.00	1.24
9	109	31.45	0.29	35.00	0.32	138.00	1.27
10	177	51.50	0.29	58.33	0.33	225.00	1.27
11	287	85.95	0.30	94.17	0.33	357.00	1.24
12	465	140.95	0.30	152.50	0.33	584.00	1.26
13	753	225.00	0.30	248.33	0.33	949.00	1.26
14	1218	364.95	0.30	402.50	0.33	1534.00	1.26
15	1972	593.50	0.30	649.17	0.33	2479.00	1.26
Média Tempo/Resoluções			0.30		0.33		1.26

TABELA 6.4 - Teste do CASLOG com procedimento *hanoi*

HANOI							
TAM.	RES.	SICSTUS		C-PROLOG		WAM	
		Tempo (ms)	Tempo/Res.	Tempo (ms)	Tempo/Res.	Tempo (ms)	Tempo/Res.
1	1	0.00	0.00	0.00	0.00	0.00	0.00
2	8	1.00	0.12	0.83	0.10	4.44	0.56
3	26	3.00	0.11	3.33	0.13	15.56	0.60
4	70	8.45	0.12	10.00	0.14	35.56	0.51
5	174	18.50	0.11	22.50	0.13	120.00	0.69
6	414	43.00	0.10	54.17	0.13	217.78	0.53
7	958	99.50	0.10	125.00	0.13	468.89	0.49
8	2174	220.00	0.10	279.17	0.13	1024.44	0.47
9	4862	486.00	0.10	618.33	0.13	2253.33	0.46
10	10750	1054.45	0.10	1395.00	0.13	4887.50	0.45
Média Tempo/Resoluções			0.10		0.13		0.53

A análise das informações apresentadas nas quatro tabelas permite as seguintes considerações:

❶ **Durante a execução de um procedimento num determinado ambiente, a complexidade de resolução mantém-se praticamente constante**. Por exemplo, na tabela 6.2, a quarta coluna demonstra que a complexidade de uma resolução durante a execução do *nrev* no Sicstus possui praticamente o valor 0.08 milisegundos. **A estabilidade da complexidade de resolução de um procedimento num ambiente permite previsões precisas do tempo envolvido na execução**. Por exemplo, se o procedimento *nrev* fosse chamado com uma lista de tamanho 1000, através da sua expressão de complexidade (figura 6.4), pode-se prever que seriam necessárias 501.501 resoluções. Como cada resolução possui uma complexidade de 0.08 milisegundos, o tempo total de execução do procedimento seria aproximadamente 40 segundos. Do ponto de vista da análise de granulosidade, a previsibilidade do tempo de execução de um procedimento é fundamental, pois permite uma comparação entre complexidade dos grãos e custos de paralelização. Essa comparação é a base da exploração com máxima eficiência do paralelismo na programação em lógica;

❷ **A complexidade de uma resolução para um procedimento varia de acordo com o ambiente de execução (*software*)**. Por exemplo, conforme mostra a última linha da tabela 6.1, uma resolução do procedimento *append* possui complexidade 0.08 ms no Sicstus, 0.12 ms no C-Prolog e 0.35 ms na WAM. Esta instabilidade da complexidade de resolução entre ambientes é inevitável, pois cada ambiente executa o procedimento em um deteminado tempo, apesar do número de resoluções manter-se constante. Sendo assim, **não é possível utilizar para todos os ambientes a mesma complexidade para uma resolução num procedimento**. Do ponto de vista da análise de granulosidade, essa instabilidade não cria problemas, pois apenas um ambiente é utilizado na construção de uma determinada plataforma paralela;

❸ **A complexidade de uma resolução para um procedimento varia de acordo com o poder computacional do processador (*hardware*)**. Por exemplo, a execução do *nrev* no C-Prolog numa Sun IPC gera uma complexidade de resolução de 0.10 ms. A última linha da sexta coluna da tabela 6.2 mostra que uma resolução do *nrev* no C-Prolog com uma estação Sun SLC possui complexidade 0.12 ms. Desta forma, **não é possível utilizar em arquiteturas que possuam processadores com poder computacional diferente, ainda que com o uso do mesmo ambiente, uma mesma complexidade para uma resolução num procedimento**. Do ponto de vista da análise de granulosidade, essa instabilidade torna-se um problema complicado em arquiteturas que possuam processadores heterogêneos, pois o tamanho do argumento de entrada não será o único fator a ser considerado no dimensionamento da complexidade de um grão. Deve-se considerar ainda, qual será o processador que executará o grão. Destaca-se nesse caso, o uso de redes de computadores para execução paralela de programas em lógica;

❹ **A complexidade de uma resolução para diferentes procedimentos executados no mesmo ambiente pode ser diferente**. Por exemplo, a última linha da quarta coluna das quatro tabelas mostra que uma resolução no ambiente Sicstus pode possuir diferentes valores dependendo do procedimento.

Por exemplo, uma resolução do *append* possui complexidade 0.08 ms. Para o *nrev* a complexidade é igual, ou seja, 0.08 ms. No entanto, para o procedimento *fibonacci* a complexidade é de 0.30 ms e para o *hanoi* é de 0.10ms. Portanto, **não é possível utilizar para todos os procedimentos um único valor para complexidade de uma resolução num ambiente**. Esse fato está vinculado à medida de complexidade utilizada pelo CASLOG, ou seja, a resolução. Durante a análise de complexidade o CASLOG despreza operações relacionais e aritméticas, pois essas não precisam de resoluções para sua execução. No entanto, essas operações são uma importante fonte de complexidade. Os programas utilizados nos testes (anexo A) possuem diversas operações aritméticas e relacionas. Por exemplo, o procedimento *fibonacci* possui uma operação relacional (>) e três operações aritméticas (*builtin* IS). Esse procedimento possui a maior complexidade de resolução. Por outro lado, os procedimentos *append* e *nrev*, não possuem nenhuma dessas operações. Essa característica faz com que uma resolução possua a mesma complexidade em ambos os procedimentos (veja tabelas 6.1 e 6.2). Conclui-se assim, que é natural que haja variação da complexidade de uma resolução para um procedimento no mesmo ambiente, pois cada procedimento possui diferentes operações aritméticas e/ou relacionais. Além disso, deve-se considerar ainda, que a complexidade para executar uma resolução varia de acordo com o número e a complexidade das instanciações realizadas. Sendo assim, mesmo que não existam operações relacionais e aritméticas, podem haver variações na complexidade de resolução entre procedimentos no mesmo ambiente. A solução para o problema encontra-se na mudança da medida de complexidade, ou seja, no uso de outra medida em substituição a resolução. Em [DEB93] são propostas como opções para métricas de complexidade o número de unificações e o número de instruções executadas. Saumya Debray ([DEB95a]) concorda com a necessidade de exploração de novas métricas para uso na análise de complexidade. Além disso, Debray salienta que atualmente a resolução está sendo utilizada por diversos grupos de pesquisa (por exemplo, projeto &-Prolog [BUE93]) para realização da análise de complexidade. Do ponto de vista da análise de granulosidade, essa imprecisão é um problema grave, pois cada procedimento do programa deverá possuir uma complexidade de resolução diferente. Essa abordagem é impraticável. Sendo assim, acredita-se que enquanto não for possível a utilização de outra métrica para complexidade, deve-se optar por soluções simplificadas. Uma solução bastante simples é o uso da complexidade de resolução sem operações aritméticas e/ou relacionais. Por exemplo, seguindo os resultados das tabelas 6.1 e 6.2 (procedimentos sem operações aritméticas e/ou relacionais), pode-se adotar a complexidade 0.08 ms para uma resolução no Sicstus. Outra solução é o uso da maior complexidade de resolução dos procedimentos de um programa. Por exemplo, o programa *Torre de Hanói* é composto pelos procedimentos *hanoi* e *append* (figura 4.10). De acordo com as tabelas 6.1 e 6.4, no Sicstus a complexidade de resolução do procedimento *hanoi* (0.10 ms) é maior do que a complexidade de resolução do procedimento *append* (0.08 ms). Sendo assim, adota-se a complexidade 0.10 ms para uma resolução durante a execução do programa *Torre de Hanói* no Sicstus. Neste caso, cada programa possuirá uma complexidade de resolução. Além disso, esta solução é compatível com a análise de complexidade do pior caso proposta pelo CASLOG.

6.6 Anotação de Complexidade

Conforme mostram as figuras 6.1 e 6.2, o módulo Analisador de Complexidade (AC) possui como entrada o programa particionado fornecido pelo módulo Analisador de Grãos. O programa particionado consiste do programa em lógica acrescido da anotação de grãos. Além disso, a figura 6.1 mostra que o módulo AC fornece como saída o programa granulado, produto final do GRANLOG. O programa granulado é constituído do programa particionado acrescido da anotação de complexidade. A **anotação de complexidade** é uma anotação que armazena no programa particionado as informações de complexidade obtidas durante a análise de complexidade. Esta anotação contém informações de complexidade E e informações de complexidade OU. Na figura 6.2, pode-se visualizar o fluxo de informações dentro do módulo AC. Essa figura mostra que o submódulo Gerador de Anotação de Complexidade (GAC) recebe como entradas o programa particionado e as informações geradas pelos submódulos de análise. De posse dessas entradas, o GAC pode realizar a anotação de complexidade, gerando assim, o programa granulado.

Existem dois tipos de anotação de complexidade, ou seja, **anotação de complexidade E** e **anotação de complexidade OU**. A anotação de complexidade E armazena as informações fornecidas pelo submódulo ACE. Por sua vez, a anotação de complexidade OU contém as informações geradas pelo submódulo ACO.

Conforme descrito na seção 6.4, a análise de complexidade OU gera duas informações, ou seja, **complexidade das cláusulas** (CC) e **complexidade das resolventes pendentes locais** (CRPL). Na figura 6.2, essas informações são representadas pelo fluxo "informações de complexidade OU" entre os submódulos ACO e GAC. Além disso, a seção 5.5 descreve as três anotações geradas pelo módulo Analisador de Grãos (AGR), ou seja, *grain_clause*, *grain_goal* e *grain_goals*. Essas anotações estão armazenadas no programa particionado. O submódulo GAC utiliza como base para as anotações de complexidade OU, as três anotações geradas pelo módulo AGR. Na anotação *grain_clause* é introduzida a informação CC e nas anotações *grain_goal* e *grain_goals* é introduzida a informação CRPL. Essas informações são inseridas como novos parâmetros, acrescidos no final das anotações *grain*.

A figura 5.10 mostra o programa particionado resultante da análise de grãos do procedimento *fibonacci*. Na figura 6.6 é apresentado o mesmo programa particionado acrescido da anotação de complexidade OU.

```
:- grain_clause(fibo/2,g1,[i,o],[void,int],1).
:- grain_clause(fibo/2,g2,[i,o],[void,int],1).
:- grain_clause(fibo/2,g3,[i,o],[int],11).
:- grain_goal(fibo/2,g3_4,[i,o],[int],[int],5).
:- grain_goal(fibo/2,g3_5,[i,o],[int],[int],0).


fib(0,0).
fib(1,1).
fib(M,N) :-
            M > 1, M1 is M - 1, M2 is M - 2,
            fib(M1,N1) & fib(M2,N2), N is N1 + N2.
```

FIGURA 6.6 - Programa particionado para procedimento *fibonacci* acrescido da anotação de complexidade OU

A seção 6.4 descreve a metodologia proposta pelo GRANLOG para obtenção das informações de complexidade OU. O procedimento *fibonacci* é composto por três cláusulas. As duas primeiras são fatos. Sendo assim, a complexidade dessas cláusulas é *1* (uma resolução). Este valor é inserido como último parâmetro das duas primeiras anotações *grain_clause*. Por outro lado, a terceira cláusula é um regra. Neste caso, devem ser aplicadas as regras descritas na seção 6.4 para determinação de sua complexidade. A aplicação dessas regras determina a complexidade *11* para a terceira cláusula. Esse valor é anotado na terceira *grain_clause*. Por último, são inseridas as informações CRPL. Para a primeira chamada de *fibonacci*, a CRPL é *5*, ou seja, a complexidade da segunda chamada de *fibonacci*. Esse valor é inserido na anotação *grain_goal* relacionada com a primeira chamada *fib*. Por fim, a segunda chamada de *fibonacci* possui uma CRPL com valor 0, ou seja, essa chamada não possui resolvente pendente local. Nota-se nesse exemplo, que as operações aritméticas (*builtin IS*) e relacionais não são consideradas durante a análise de complexidade OU e portanto não participam da composição das complexidades anotadas no procedimento *fibonacci*. Esta conduta compatibiliza a análise de complexidade OU com a análise de complexidade E, descrita na seção 6.5. No entanto, o autor dessa dissertação acredita que uma análise de complexidade precisa, deve considerar as operações aritméticas e relacionais. No entanto, essas operações não podem ser mensuradas com o uso da medida **resolução**. Saumya Debray e Nai-Wei Lin concordam com essa posição. No entanto, a utilização de outras medidas, tal como o número de instruções (WAM ou máquina real) executadas, permitirá o tratamento de operações durante a análise de complexidade.

A anotação de complexidade OU apresentada na figura 6.6 pode ser utilizada para orientar a paralelização do programa. Por exemplo, o procedimento *fibonacci* pode ser chamado com o acumulador de CRPLs (ACRPL) contendo valor 0, ou seja, não existem resolventes pendentes. No momento da execução do procedimento *fib*, o ambiente constata a possibilidade de criação de grãos-OU, ou seja, constata a possibilidade de execução paralela de ramos da árvore de busca. Neste caso, o sistema deve avaliar a complexidade dos ramos para verificar se existem grãos-OU com granulosidade satisfatória. Neste caso, o primeiro ramo possui complexidade 1 (complexidade do primeiro fato + ACRPL), o segundo ramo possui complexidade 1 (complexidade do segundo fato + ACRPL) e o terceiro ramo possui complexidade 11 (complexidade de regra + ACRPL). Certamente, nessa situação o sistema não executará em paralelo nenhum grão-OU, pois o custo para realizar a paralelização em qualquer ambiente paralelo supera o tempo para execução de uma resolução. Nesse exemplo, o sistema paralelo decidirá pela execução local (sem paralelização) de todo o procedimento *fibonacci*, apesar da **possibilidade** de paralelismo. Sem o controle da granulosidade, o sistema executaria em paralelo um dos ramos. Se fosse um dos dois primeiros (complexidade 1), o custo para paralelização certamente superaria a complexidade do ramo, ocasionado um perda de desempenho. Se fosse o último (complexidade 11), o custo poderia ser menor do que a complexidade, mas no entanto, o processador exportador executaria localmente com rapidez os dois fatos e ficaria sem trabalho. Nesse caso, a paralelização também ocasionaria perda de desempenho. Os custos para paralelização das tarefas são altos, principalmente em sistemas com memória distribuída. Considerando complexidades e custos, a análise de granulosidade controla a exploração do paralelismo, visando a máxima eficiência.

Conforme descrito na seção 6.5, a anotação de complexidade E gera duas informações, ou seja, as expressões de complexidade e as relações de tamanho entre

entradas e saídas. Na figura 6.2, essas informações são representadas pelo fluxo "informações de complexidade E" entre os módulos ACE e GAC. O GRANLOG propõe a criação de duas novas anotações para registro das expressões e relações, ou seja, são criadas as anotações ***granularity*** e ***out_size***. A primeira anotação armazena uma expressão de complexidade que será utilizada para determinar a granulosidade dos grãos-E. Por sua vez, a segunda registra uma relação que será utilizada para deteminar o tamanho das saídas de um grão-E em função do tamanho de suas entradas. A sintaxe dessas anotações é a seguinte:

```
:- granularity(<identificador da expressão>, <expressão>).

:- out_size(<identificador da relação>, <relação>).
```

A anotação *granularity* possui dois parâmetros. O primeiro contém um identificador para a expressão. Esse identificador é composto pela letra ***e*** seguida de um número natural. O segundo parâmetro é a própria expressão que será armazenada no formato apresentado na figura 6.4. A anotação *out_size* também possui dois parâmetros. O primeiro contém um identificador para a relação. Esse identificador é composto pela letra ***r*** seguida de um número natural. O segundo parâmetro é a própria relação que será armazenada no formato mostrado na figura 6.5. Em complemento, o submódulo GAC introduz dois novos parâmetos nas anotações *grain*. Esses parâmetros são colocados no final dessas anotações e contém os identificadores da expressão e relação vinculados com o grão. Desta forma, cria-se uma ligação entre as anotações *grain*, *granularity* e *out_size*.

Na figura 5.10 é apresentado o programa particionado resultante da análise de grãos do procedimento *fibonacci*. A figura 6.6 mostra o programa particionado acrescido da anotação de complexidade OU. Por sua vez, a figura 6.7 mostra o programa particionado acrescido de ambas as anotações de complexidade, ou seja, a figura 6.7 apresenta o programa granulado para o procedimento *fibonacci*. Esse programa é o produto final do GRANLOG.

```
:- granularity(e1,1.45*exp(1.62,$1)+0.55*exp(-0.62,$1)-1)
:- out_size(r1,[0,0]).
:- out_size(r2,[0,1]).
:- out_size(r3,[$1,0.45*exp(1.62,$1)-0.45*exp(-0.62,$1)]).

:- grain_clause(fibo/2,g1,[i,o],[void,int],1,1,r1).
:- grain_clause(fibo/2,g2,[i,o],[void,int],1,1,r2).
:- grain_clause(fibo/2,g3,[i,o],[int],11,e1,r3).
:- grain_goal(fibo/2,g3_4,[i,o],[int],[int],5,e1,r3).
:- grain_goal(fibo/2,g3_5,[i,o],[int],[int],0,e1,r3).


fib(0,0).
fib(1,1).
fib(M,N) :-
          M > 1, M1 is M - 1, M2 is M - 2,
          fib(M1,N1) & fib(M2,N2), N is N1 + N2.
```

FIGURA 6.7 - Programa granulado para programa *fibonacci*

A seção 6.5 ressalta que a análise de complexidade E determina a complexidade para execução de partes do corpo de uma cláusula. Além disso, destaca que através dessa análise pode-se deteminar a complexidade para execução de uma cláusula completa, basta analisar integralmente a cláusula. Sendo assim, o submódulo ACE realiza a análise de complexidade E de cada cláusula do programa em lógica. Por sua vez, o submódulo GAC gera anotações *granularity* e *out_size* para cada uma dessas cláusulas. Essas anotações são vinculadas as anotações *grain_clause* que identificam as cláusulas. Por exemplo, a terceira anotação *grain_clause* na figura 6.7 possui dois novos parâmetros, ou seja, ***e1*** e ***r3***. O parâmetro ***e1*** identifica a expressão de complexidade para a cláusula e o parâmetro ***r3*** identifica sua relação de tamanho entre entradas e saídas. Torna-se interessante verificar que o penúltimo parâmetro das anotações *grain_clause* contém a mesma informação que o antepenúltimo. A diferença encontra-se na precisão da informação. Por exemplo, no *grain_clause* ***g3***, a complexidade 11 é obtida de forma simples e estática utilizando a metodologia descrita na seção 6.4. Por sua vez, a expressão de complexidade indicada por ***e1*** é obtida através de um sofisticada análise e permite a determinação precisa da complexidade. Nas duas primeiras anotações *grain_clause* ambos os parâmetros são idênticos, pois o valor de execução dos fatos é constante (uma resolução). Destaca-se nessas duas anotações *grain_clause* um detalhe interessante. Quando a complexidade for constante, o submódulo GAC anota o valor diretamente nas anotações *grain*, ou seja, não é necessária a criação de uma anotação *granularity*.

As duas informações de complexidade das cláusulas podem ser utilizadas na comparação entre as abordagens descritas nas seções 6.4 e 6.5. Além disso, durante a análise de complexidade OU, o sistema poderá optar pelo uso da informação simplificada ou pelo uso das expressões de complexidade. Essas expressões são mais precisas, mas no entanto, devem ser resolvidas em tempo de execução (*overhead*). Por sua vez, as relações de tamanho entre entradas e saídas de cada cláusula podem ser utilizadas para diversas análises, dentre as quais destaca-se a influência de cada cláusula na composição da relação gerada para o procedimento.

Na figura 6.7 existem outros detalhes que merecem ser ressaltados. Por exemplo, as relações ***r1*** e ***r2*** contém apenas valores constantes. Esse fato decorre dessas relações descreverem fatos que possuem apenas valores constantes nos seus argumentos. O primeiro argumento é uma entrada e não influencia na análise (*void*), portanto recebe um valor nulo. O segundo argumento é uma saída que possui valor constante, portanto na relação, é inserido esse valor. Quando esses fatos forem resolvidos com sucesso, as saídas sempre possuirão o mesmo tamanho e poderão ser obtidas facilmente na relação. Outro detalhe interessante consiste no uso da mesma expressão e/ou relação por várias anotações *grain*. Por exemplo, a expressão ***e1*** e a relação ***r3*** são utilizadas pela última anotação *grain_clause* e pelas duas anotações *grain_goal*. Além disso, deve-se ressaltar que se um grão não possui saídas, não existe sentido em possuir uma anotação *out_size*. Sendo assim, quando isso ocorrer, o submódulo GAC coloca o símbolo "_" no último parâmetro da anotação *grain*.

A análise de apenas um procedimento simples como o *fibonacci*, permite uma série de considerações didáticas. No entanto, não permite a visualização de todo o potencial das informações que podem ser introduzidas pelo GRANLOG num programa em lógica. Em programas com diversos procedimentos, são criadas diversas expressões de complexidade e várias relações de tamanho entre entradas e saídas. Além disso, a existência de *grain_goals* e diferentes *grain_goal* no corpo das cláusulas, gera uma

grande variedade de anotações de complexidade. A seção 6.7 exemplifica a análise de complexidade para o programa *Torre de Hanói*. Esta seção descreve o último passo do exemplo iniciado na seção 4.7. A seção 6.7 mostra e discute o programa granulado criado para o programa *Torre de Hanói*. No capítulo 7 é apresentada a proposta de integração OPERA/GRANLOG. Durante essa apresentação é discutida a aplicação da anotação de complexidade no controle da granulosidade.

6.7 Exemplo de Análise de Complexidade: Programa *Torre de Hanói*

As seções 4.7 e 5.4 exemplificam a análise global e a análise de grãos para o programa *Torre de Hanói*. Dando continuidade ao exemplo, nesta seção é apresentada a análise de complexidade. A figura 6.8 mostra o programa granulado para o programa *Torre de Hanói*.

```
:- granularity(e1,$1*exp(2,$1)+exp(2,$1-1)-2).
:- granularity(e2,$1*exp(2,$1)+3*exp(2,$1-1)-2).
:- granularity(e3,$1-1).

:- out_size(r1,[0,0,0,0,1]).
:- out_size(r2,[$1,0,0,0,exp(2,$1)-1]).
:- out_size(r3,[$1,0,0,0,exp(2,$1)]).
:- out_size(r4,[$1,0,$1+1]).
:- out_size(r5,[$1,$2,$1+$2]).
:- out_size(r6,[0,$2,$2]).

:- grain_clause(hanoi/5,g1,[i,i,i,i,o],[void,void,void,void,length],1,1,r1).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],[int,void,void,void,length],39,e1,r2).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],[i,i,i,i,o],[int,void,void,void,length],
          [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])],19,e2,r3).
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],[int,void,void,void,length],
         [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])],24,e1,r2).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length]),
              [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
               list(?,[struct(2,2,[atom(3)])])],19,e1,r4).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],[int,void,void,void,length],
         [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])],5,e1,r2).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],[list(?,[struct(2,2,[atom(3)])])],0,
              e3,r5).

hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
      N > 1, N1 is N - 1,
     (hanoi(N1,A,C,B,M1),append(M1,[mv(A,C)],T)) &
      hanoi(N1,B,A,C,M2), append(T,M2,M).


:- grain_clause(append/3,g1,[i,i,o],[void,length,length],1,1,r6).
:- grain_clause(append/3,g2,[i,i,o],[length],4,e3,r5).
:- grain_goal(append/3,g2_1,[i,i,o],[length],[list(?,[struct(2,2,[atom(3)])])],0,
              e3,r5).

append([],L,L).
append([HL],L1,[HR]) :- append(L,L1,R).
```

FIGURA 6.8 - Programa granulado para programa *Torre de Hanói*

Na figura 6.8 encontra-se um conjunto de anotações de complexidade, onde destacam-se três expressões de complexidade, seis relações de tamanho e vários detalhes interessantes sobre a anotação final produzida pelo GRANLOG. Dentre esses detalhes, três merecem ser ressaltados. Em primeiro lugar, destacam-se as anotações de complexidade vinculadas com o único grão-metas do programa. Conforme mostra a figura 6.8, o grão-metas *g2_3* possui uma resolvente pendente local com valor 19. Esse valor resulta da soma da complexidade das metas *hanoi* e *append* que o sucedem na

cláusula. Sendo assim, a CRPL de um grão-metas sempre será a soma das metas que o sucedem. Obviamente, cada uma das metas que compõem o grão-metas possui sua própria CRPL, calculada da forma tradicional. Além disso, o grão-metas possui sua própria expressão de complexidade (*e2*) e sua própria relação de tamanho entre entradas e saídas (*r3*).

Em segundo lugar, o programa *Torre de Hanói* possui dois fatos, um no procedimento *hanoi* e outro no procedimento *append*. O fato do procedimento *hanoi* possui uma saída constante com tamanho 1, conforme mostra a relação *r1*. Este valor é devido à constatação de que a saída do fato sempre será uma lista de tamanho 1 e a medida utilizada para o argumento de saída é *length*. Por sua vez, o fato do procedimento *append* não possui uma saída constante, conforme mostra a relação *r6*. Nessa relação constata-se que o tamanho da saída será igual ao tamanho da segunda entrada. Esta constatação pode ser facilmente verificada pela análise do fato. Em terceiro lugar, deve-se ressaltar a relação *r4* vinculada com o grão-meta *g2_3_2*. Essa relação é uma simplificação da relação vinculada com uma chamada genérica para o procedimento *append* (relação *r5*). Essa simplificação deve-se ao fato de que a chamada para o grão-meta *g2_3_2* (primeira chamada para *append*) possui o tamanho do segundo parâmetro constante, ou seja, uma lista de tamanho 1. Neste caso, a relação *r4* pode ser utilizada para simplificar a análise do tamanho das saídas. A relação *r5* poderia ser utilizada, mas necessitaria mais processamento, pois envolveria a deteminação do tamanho do segundo argumento (entrada), mesmo esse sendo sempre constante. A mesma situação pode ocorrer com as expressões de complexidade, ou seja, se em deteminada chamada um argumento de entrada for sempre constante, será criada uma expressão simplificada que não envolva esse argumento. O restante da anotação mostrada na figura 6.8 segue as regras descritas nas seções 6.4, 6.5 e 6.6.

6.8 Conclusões

Neste capítulo foram apresentados conceitos e referências sobre análise de complexidade e especificamente o último módulo do GRANLOG, ou seja, o Analisador de Complexidade. No decorrer do capítulo, foram criados diversos conceitos novos, tais como análise de complexidade E, análise de complexidade OU, resolvente pendente, CRPL, ACRPL e outros. Além disso, foi apresentada uma proposta para análise de complexidade OU e uma abordagem para análise de complexidade E no âmbito do GRANLOG. Finalmente, foi discutida a anotação de complexidade criada pelo modelo proposto, a qual complementa as demais anotações descritas nos capítulos anteriores.

Destacam-se como principais conclusões e constatações desse capítulo:

- o estudo da análise de complexidade pode ser subdividido em duas áreas, ou seja, análise de complexidade E e análise de complexidade OU;
- atualmente a pesquisa sobre análise de complexidade E já possui resultados animadores, por outro lado, a análise de complexidade OU ainda é um tópico de estudo pouco pesquisado;
- a principal dificuldade da análise de complexidade OU é o tratamento das resolventes pendentes;
- a utilização da medida de complexidade **resolução** introduz imprecisão, por exemplo, no tratamento de operações aritméticas e relacionais;

- o estudo do CASLOG demonstrou que a complexidade de uma resolução para um procedimento sofre diversas instabilidades que devem ser dimensionadas para avaliação precisa da complexidade nos programas em lógica;
- um dos próximos passos de pesquisa no âmbito da análise de complexidade na programação em lógica deve ser o desenvolvimento de novas medidas de complexidade;
- a análise de complexidade é indispensável para realização da análise de granulosidade, pois essa análise depende de comparações de custos de paralelização e complexidade dos grãos (granulosidade);
- as anotações de complexidade podem ser utilizadas em várias análises, tanto para pesquisa do comportamento de programas, quanto para exploração do paralelismo na programação em lógica.

O próximo capítulo apresenta o protótipo GRANLOG. Esse protótipo concretiza grande parte do modelo e permite a discussão de vários aspectos práticos, tais como estruturas de dados utilizadas na implementação e simplificações adotadas para rápida viabilização de resultados. Além disso, o protótipo GRANLOG serve de suporte para integração OPERA-GRANLOG discutida no capítulo oito.

7 Protótipo GRANLOG

Este capítulo apresenta o protótipo GRANLOG. Esse protótipo concretiza grande parte das idéias discutidas nos capítulos anteriores, permitindo assim, a avaliação do modelo e sua aplicação prática. A seção 7.1 apresenta os princípios básicos do protótipo. Na seção 7.2 é descrita sua organização básica. Por sua vez, a seção 7.3 descreve especificamente as características de implementação do módulo Analisador Global. A seção 7.4 descreve a implementação do módulo Analisador de Grãos. A implementação do módulo Analisador de Complexidade é discutida na seção 7.5. Finalmente, a seção 7.6 apresenta as conclusões deste capítulo. Encontra-se em [KAY95] uma descrição detalhada do protótipo GRANLOG. Em [KAY95a] o protótipo foi apresentado à comunidade científica.

7.1 Princípios Básicos

O protótipo GRANLOG possui como base alguns princípios que nortearam seu desenvolvimento. Esses princípios foram utilizados para delimitar a abrangência da primeira versão do protótipo, bem como para determinar as prioridades e caminhos da implementação.

Os princípios básicos do protótipo GRANLOG são os seguintes:

1. **Dedicação ao paralelismo E independente tradicional.** A primeira versão do protótipo está orientada especificamente para exploração do paralelismo E. Sendo assim, não são geradas informações de granulosidade OU. Portanto, a anotação de granulosidade não contém anotações *grain_clause* e nem informações sobre complexidade das resolventes pendentes locais. Além disso, são geradas apenas informações de granulosidade para metas isoladas, ou seja, não são criadas anotações *grain_goals*. Portanto, a primeira versão do GRANLOG **dedica-se ao paralelismo E independente tradicional**. Para exploração desse tipo de paralelismo são necessárias apenas informações sobre o particionamento do corpo das cláusulas (símbolo &) e informações sobre suas metas componentes (anotação *grain_goal*). Esse princípio simplifica a implementação do protótipo, delimitando seu escopo ao tipo de paralelismo suportado atualmente pelo OPERA ([GEY92], [YAM93], [WER94], [WER94a], [YAM94]). Sendo assim, as informações geradas pela primeira versão do protótipo suportam a primeira aplicação do GRANLOG, ou seja, o auxílio a paralelização no OPERA. Esta aplicação é concretizada através da integração OPERA-GRANLOG descrita no próximo capítulo;

2. **Estruturas de dados genéricas e de ampla utilização.** Durante a análise de granulosidade, o protótipo GRANLOG cria diversas estruturas de dados contendo informações sobre o programa. Essas informações são utilizadas durante o processo de análise para geração da anotação de granulosidade. No entanto, essa anotação não contém todas as informações obtidas e organizadas pelo protótipo. Por exemplo, o módulo Analisador Global investiga o programa Prolog e armazena informações sintáticas em estruturas de dados a serem utilizadas na análise de dependências. Além disso, as dependências entre literais também são armazenadas em estruturas de dados que serão utilizadas pelo módulo Analisador de Grãos. As informações sintáticas e dependências não são

armazenadas na anotação de granulosidade. No entanto, essas informações são interessantes e podem ser utilizadas em diversas aplicações. Sendo assim, o protótipo GRANLOG possui como princípio básico organizar **todas** as informações obtidas durante a análise em **estruturas de dados genéricas e de ampla utilização**. Esse princípio reflete diretamente na organização das estruturas de dados e na lógica utilizada pelos algoritmos;

❸ **Utilizar na implementação, ferramentas amplamente difundidas e compatíveis com o treinamento do grupo de desenvolvimento**. A determinação das ferramentas a serem utilizadas na implementação do protótipo GRANLOG foi guiada por esse princípio básico. Com base nesse princípio, optou-se pela utilização da linguagem ANSI C, uma **ferramenta amplamente difundida e compatível com o treinamento do grupo de desenvolvimento**. Além disso, foram utilizados no módulo AGL, geradores de analisadores léxicos e sintáticos, mais especificamente os geradores LEX e YACC ([MAS91]). O texto [KAY95] apresenta a gramática regular (LEX) e a gramática livre de contexto (YACC) utilizadas para implementar o protótipo GRANLOG.

7.2 Organização Básica

O protótipo GRANLOG possui a mesma organização do modelo, ou seja, é composto de três módulos. Esses módulos recebem os mesmos nomes utilizados no modelo, ou seja, Analisador Global (AGL), Analisador de Grãos (AGR) e Analisador de Complexidade (AC). A figura 7.1 mostra a organização do protótipo destacando as estruturas de dados utilizadas para suportar o fluxo de informações.

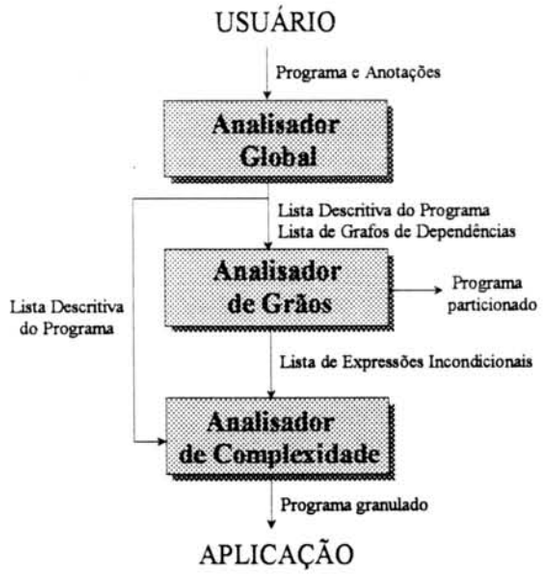


FIGURA 7.1 - Organização do protótipo GRANLOG

O módulo AGL recebe o programa Prolog, complementado pelas anotações de modos, tipos e medidas descritas no capítulo 4. Através da análise do texto, esse módulo produz duas estruturas de dados, ou seja, a **Lista Descritiva do Programa** (LDP) e a **Lista de Grafos de Dependência** (LGD). Ambas estruturas de dados seguem o segundo princípio descrito na seção 7.1. Por sua vez, o módulo AGR recebe ambas as listas e

realiza a análise de grãos. Desta análise, resulta um arquivo texto contendo o programa particionado. Além disso, o módulo AGR cria uma nova estrutura de dados denominada **Lista de Expressões Incondicionais** (LEI). A LEI e a LDP são entregues ao terceiro módulo do sistema. Finalmente, o módulo AC realiza a análise de complexidade, gerando um arquivo texto contendo o programa granulado. As próximas três seções deste capítulo descrevem com mais detalhes os módulos do protótipo GRANLOG, ressaltando as estruturas de dados e os algoritmos.

7.3 Módulo Analisador Global

O módulo Analisador Global é composto por dois submódulos (figura 7.2). O primeiro submódulo analisa o programa e cria a Lista Descritiva do Programa (LDP). Por sua vez, o segundo submódulo utiliza as informações armazenadas na LDP para criar a Lista de Grafos de Dependência (LGD). A criação da LDP é realizada em dois passos. O primeiro passo dedica-se à análise das anotações de modos, tipos e medidas. Esta análise realiza duas tarefas, ou seja, verifica se a sintaxe das anotações está correta e introduz na LDP as informações de modos, tipos e medidas. O segundo passo analisa o código Prolog e introduz na LDP os literais e variáveis componentes de cada cláusula do programa. Portanto, a LDP armazena de forma organizada todo o programa em análise. O primeiro passo foi implementado diretamente na linguagem C e o segundo gerado pelo LEX e YACC. O segundo submódulo do AGL cria a LGD. Esse submódulo implementa o algoritmo de análise de dependências apresentado na figura 4.7. O módulo AGL possui ainda uma terceira estrutura de dados responsável pelo armazenamento das informações sobre os predicados pré-definidos do Prolog. Essa estrutura mantém informações constantes que são utilizadas durante a análise de dependências. Deve-se ressaltar ainda, que as três estruturas de dados utilizadas no AGL seguem o segundo princípio do protótipo. O texto [KAY95] descreve em detalhes a organização dessas estruturas.

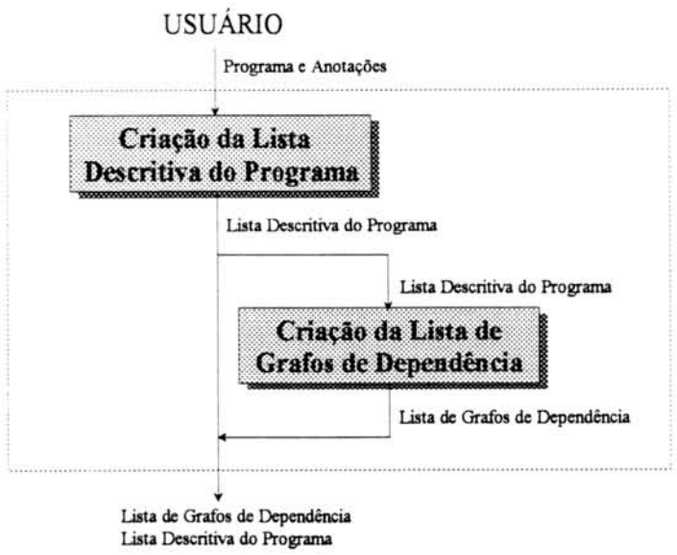


FIGURA 7.2 - Organização do Módulo Analisador Global no protótipo

7.4 Módulo Analisador de Grãos

O módulo Analisador de Grãos (AGR) do protótipo possui a mesma organização interna proposta pelo modelo (figura 5.2). A figura 7.3 mostra essa organização do ponto de vista do protótipo, ressaltando as estruturas de dados utilizadas na implementação. O módulo AGR recebe as duas listas produzidas pelo módulo AGL. O submódulo Determinador de Grãos (DG) utiliza a LGD para inferir os grãos existentes no programa. Basicamente, o DG implementa o algoritmo apresentado na figura 5.3. Esse algoritmo gera para cada grafo de dependência uma UGE. As UGEs geradas para todas as cláusulas do programa são armazenadas numa nova estrutura de dados denominada **Lista de Expressões Incondicionais** (LEI). Essa lista contém uma descrição dos grãos em potencial existentes no corpo de cada cláusula do programa. Além disso, conforme mostra a figura 5.4, o modelo GRANLOG prevê uma subdivisão para o submódulo DG. Essa subdivisão está implementada no protótipo. O texto [KAY95] descreve a implementação do algoritmo para determinação de grãos e apresenta com mais detalhes a organização da Lista de Expressões Incondicionais.

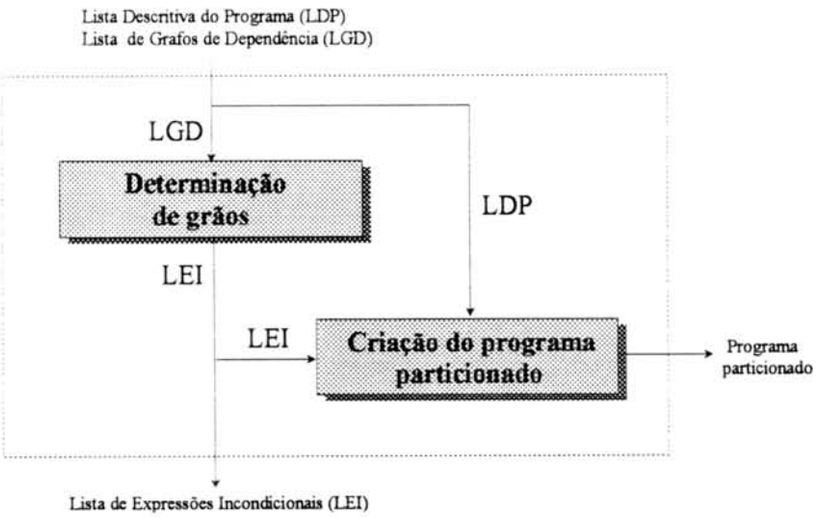


FIGURA 7.3 - Organização do módulo Analisador de Grãos no protótipo

O segundo submódulo do AGR gera a anotação de grãos. Esse submódulo possui como entrada a LEI e a LDP. De posse dessas estruturas, o Gerador de Anotações (GA) produz o Programa Particionado. Deve-se notar que na descrição do modelo (figura 5.2) o AGR possui como entrada o programa em lógica. No protótipo o programa em lógica está armazenado na LDP. Esse armazenamento é realizado no segundo passo da criação da LDP no módulo AGL. Deve-se ressaltar ainda, que seguindo o primeiro princípio descrito na seção 7.1, a anotação de grãos gerada atualmente pelo AGR não contém anotações *grain_clause* a *grain_goals*, ou seja, a anotação é direcionada para exploração do paralelismo E independente tradicional. Em [KAY95] é apresentado um exemplo dessa anotação para o procedimento *fibonacci*.

7.5 Módulo Analisador de Complexidade

Seguindo o primeiro princípio do protótipo, o módulo Analisador de Complexidade (AC) dedica-se apenas à análise de complexidade E, ou seja, o protótipo

não realiza a análise de complexidade OU. Essa situação implica uma simplificação da estrutura apresentada na figura 6.2, levando à organização mostrada na figura 7.4. A base do módulo AC é um sistema de Análise de Complexidade E para programação em lógica, denominado CASLOG ([DEB93], [LIN93]). Esse sistema analisa programas em lógica e gera expressões de complexidade e relações de tamanho entre entradas e saídas dos procedimentos. A seção 6.5 discute e exemplifica as expressões e relações geradas pelo CASLOG. Conforme mostra a figura 7.4, o módulo AC recebe como entradas a Lista Descritiva do Programa (LDP) e a Lista de Expressões Incondicionais (LEI). Utilizando como base a LDP, o módulo AC cria um programa em lógica no padrão CASLOG. O CASLOG exige como entrada um programa com dois tipos de anotações para todos os procedimentos, ou seja, anotações de modos e anotações de medidas. Essas anotações são bastante semelhantes às propostas pelo GRANLOG. No entanto, existem algumas diferenças.

FIGURA 7.4 - Organização do módulo Analisador de Complexidade no protótipo

O CASLOG trabalha apenas com dois modos, ou seja, entrada e saída. Por sua vez, o GRANLOG utiliza quatro modos, conforme descrito na subseção 4.3.2. Além disso, o CASLOG representa seus modos através dos símbolos "+" (entrada) e "-" (saída). O GRANLOG utiliza letras para representar seus modos. Sendo assim, é necessária uma adaptação dos modos do GRANLOG para os modos do CASLOG. Basicamente, a adaptação consiste em substituir o modo *i* pelo modo + e os demais modos do GRANLOG pelo modo - do CASLOG. Em [KAY95] encontra-se mais detalhes a respeito dessa adaptação. O programa adaptado é entregue ao CASLOG, o qual gera num arquivo texto as expressões e relações para cada procedimento.

Com bases na expressões e relações geradas pelo CASLOG e nas listas LDP e LEI, pode-se criar o programa granulado. Na figura 6.2, essa tarefa é realizada pelo submódulo GAC. O programa granulado é o produto final do GRANLOG e contém todas as informações necessárias para exploração eficiente do paralelismo E independente tradicional. Conforme descrito na seção 7.1, o primeiro princípio do

protótipo estabelece a geração apenas de informações relacionadas com metas isoladas. Sendo assim, o programa granulado contém apenas anotações *grain_goal*, *granularity*, *out_size* e o símbolo *&*. O texto [KAY95] exemplifica a geração do programa granulado para os programas *Fibonacci* e *Torre de Hanói*.

7.6 Conclusões

Este capítulo apresentou de forma resumida o protótipo GRANLOG. Durante a apresentação foram descritas diversas estruturas de dados utilizadas para implementar o sistema. Além disso, foram mostradas diversas relações entre o modelo e o protótipo, permitindo assim, a compreensão do estado atual da aplicação das idéias desenvolvidas nos capítulos anteriores. Inicialmente, discutiu-se os princípios básicos que nortearam o desenvolvimento do protótipo. Logo após, foi mostrada a organização geral do sistema. Nas últimas três seções foram detalhados os três módulos componentes do protótipo GRANLOG.

Destacam-se como algumas das principais conclusões desse capítulo:

- o protótipo possui uma estrutura semelhante ao modelo (três módulos);
- a organização de cada módulo foi simplificada de acordo com o primeiro princípio do protótipo;
- as estruturas de dados foram generalizadas ao máximo visando futuras aplicações (segundo princípio do protótipo);
- os algoritmos descritos nos capítulos 4 e 5 foram utilizados para implementar os módulos AGL e AGR

O protótipo atual do GRANLOG deve ser aperfeiçoado. Dentre os próximos passos de desenvolvimento do sistema merecem destaque: substituição das anotações de usuário por uma análise global automática, tratamento de grão-metas e grão-cláusulas no módulo de análise de grãos e realização da análise de complexidade OU.

O próximo capítulo discute a integração OPERA-GRANLOG. Durante essa discussão são abordados diversos aspectos sobre a aplicação das informações geradas pelo GRANLOG na paralelização de programas em lógica. A integração com o OPERA serve de alicerce para o modelo GRANLOG, pois permite sua avaliação e justifica em grande parte sua existência. A possibilidade de um aumento significativo na eficiência do OPERA foi um dos principais estímulos para criação do GRANLOG. Além disso, o anseio pela integração OPERA-GRANLOG influenciou de forma significativa nas decisões de projeto do protótipo. O primeiro princípio do protótipo é um claro reflexo da expectativa de integração dos dois sistemas.

8 Integração OPERA-GRANLOG

Este capítulo apresenta a integração OPERA-GRANLOG. A possibilidade dessa integração foi um dos principais estímulos para desenvolvimento do modelo GRANLOG. Além disso, a integração serve para validar as idéias discutidas no capítulos anteriores e estimular o aperfeiçoamento do modelo proposto nesse texto. A seção 8.1 apresenta de forma resumida o modelo OPERA. A seção 8.2 mostra uma visão geral da proposta de integração visando introduzir o assunto. Por sua vez, a seção 8.3 discute a utilização das informações fornecidas pelo GRANLOG no controle da granulosidade no OPERA. Esse item aborda de forma detalhada o tratamento das informações de granulosidade. Neste sentido, discute as estruturas de dados e os conceitos envolvidos na proposta de integração. Na seção 8.4 é apresentada a proposta de inclusão do GRANLOG na interface gráfica do OPERA, denominada XOPERA. Finalmente, a seção 8.5 apresenta as conclusões deste capítulo. A proposta de integração OPERA-GRANLOG foi apresentada à comunidade científica através do artigo [BAR95a].

8.1 Modelo OPERA

O OPERA é um modelo para exploração do paralelismo na execução de programas em lógica. Destacam-se como características básicas deste modelo:

- O modelo OPERA é direcionado para máquinas que utilizam memória distribuída;
- A MAP (Máquina Abstrata Prolog) utilizada no OPERA é baseada na WAM (*Warrem Abstract Machine*) ([AÏT91]);
- O OPERA propõe-se a explorar o paralelismo E/OU na programação em lógica;
- Atualmente, o OPERA explora apenas o paralelismo RAP (*Restricted AND Parallelism*) ([DEG84],[DEG87]), o qual combina uma análise de dependência e geração de CGE's (*Conditional Graph Expressions*) em tempo de compilação com a avaliação dessas expressões em tempo de execução ([GEY92],[YAM93]).

A MAP utilizada no OPERA é uma extensão da máquina abstrata proposta em [HER86]. No entanto, em [HER86] é considerada memória compartilhada, o que não condiz com o modelo OPERA onde utiliza-se memória distribuída. Em relação à arquitetura da máquina abstrata, o OPERA manteve as seguintes estruturas: ***Área de Código*** (armazena o código compilado), ***Local*** (armazena os ambientes e os nodos OU), ***Global*** (armazena as estruturas e listas criadas dinamicamente), ***Trail*** (armazena as ligações que precisam ser desfeitas quanto ocorrer um retrocesso) e ***PDL*** (utilizada para operações de unificação). Baseadas nas estruturas ***Goal Stack*** e ***Parcall Frame*** propostas em [HER86], o modelo OPERA cria a **Pilha de Objetivos Paralelos** (POP) e a **Estrutura de Chamadas Paralelas** (ECP). A primeira armazena os objetivos que podem ser executados em paralelo. A segunda controla a execução paralela dos diversos objetivos da POP. Além disso, algumas instruções da WAM foram modificadas. A instrução *proceed* foi alterada para detecção do sucesso na execução das metas e a instrução *allocate* foi alterada para alocar memória para as novas estruturas criadas visando suportar o paralelismo E.

A figura 8.1 apresenta a arquitetura de processos original ([WER94], [YAM94]) proposta pelo modelo OPERA para execução paralela de programas em lógica em ambientes distribuídos. Conforme mostra essa figura, a arquitetura de processos empregada pelo modelo OPERA possui dois tipos básicos de agentes: *mestre* e *trabalhador*. O *mestre* é o processo responsável pela gerência da arquitetura. Uma vez instalado, o *mestre* controla a execução inicializando processos e escalonando serviços. A figura 8.1, mostra que cada *trabalhador* é formado por três processos, ou seja, ***Solver*** (responsável pela execução dos programas Prolog,), ***Spy*** (responsável por manter o processo *mestre* informado sobre o estado dos trabalhadores) e ***Communicator*** (realiza o tratamento das comunicações).

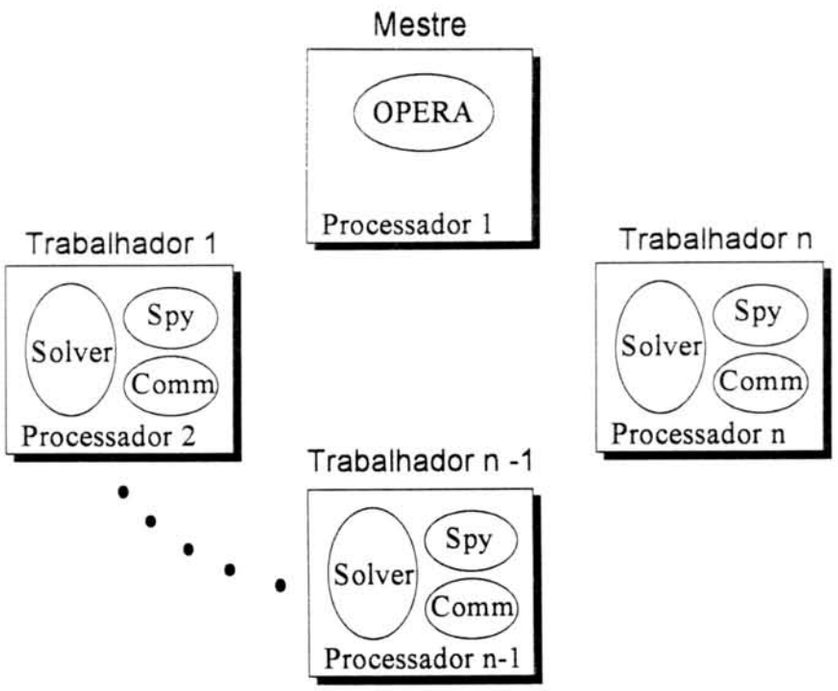


FIGURA 8.1 - Arquitetura de processos do modelo OPERA

A avaliação da carga de trabalho em um *trabalhador* é realizada dinamicamente. Com base nessa avaliação, o *mestre* toma as decisões de escalonamento. A carga do trabalhador é medida pelo número de literais armazenados na POP. Dependendo dessa carga, o trabalhador encontra-se em um dos seguintes estados (os limites entre os estados são especificados por cada aplicação):

- ***idle***: trabalhador não possui trabalho a ser realizado. Portanto, está esperando por uma autorização para importar trabalho de outro trabalhador;
- ***quiet***: trabalhador está ativo, entretanto, não possui carga suficiente para justificar uma transferência de trabalho para outro trabalhador (exportação);
- ***overloaded***: trabalhador está ativo e possui carga suficiente para justificar exportação.

O protótipo OPERA foi implementado numa rede de estações de trabalho SUN gerenciadas pelo sistema operacional UNIX (SunOS versão 4.1.1). Além disso, o padrão de interconexão é *Ethernet* com uma taxa de transferência de 10 Mbits por segundo. A determinação de quais estações serão utilizadas na execução de uma aplicação é feita

pelo usuário na inicialização do sistema ([YAM94]). Originalmente, a comunicação entre processos residentes em diferentes estações foi realizada com a utilização de primitivas construídas com *sockets* BSD (*Berkeley Software Distribution*), enquanto processos na mesma estação comunicavam-se através de memória compartilhada controlada por semáforos. Atualmente, o protótipo utiliza PVM (*Parallel Virtual Machine*) para comunicação entre estações. A migração de *sockets* para PVM alterou a arquitetura de processos proposta originalmente em [WER94] e [YAM94]. A versão PVM do OPERA não possui o processo *Communicator*.

A comunicação entre usuário e protótipo pode ocorrer de duas formas, ou seja, através da linha de comando ou através de uma interface gráfica denominada XOPERA. A XOPERA original é mostrada na figura 8.2. Essa interface foi construída com a ferramenta *Open Windows Developer's Guide* versão 1.1. Basicamente, a interface permite: a execução dos módulos componentes do OPERA (*compiler*, *assembler* e *emulator*), a visualização das entradas/saídas e a visualização no modo texto (*Trace*: *Text*) ou no modo gráfico (*Trace*: *Graphic*) de determinadas características da execução de programas. Atualmente, a interface XOPERA contém ainda suporte para execução do GRANLOG e para controle de granulosidade na execução de programas em lógica. Estes aperfeiçoamentos foram propostos e implementados durante o desenvolvimento do GRANLOG. A seção 8.4 discute essas alterações da interface.

Os textos [GEY92], [YAM93], [WER94], [WER94a] e [YAM94] podem ser utilizados para aprofundamento dos estudos do modelo OPERA. Além disso, em [WER94] e [YAM94] são apresentados diversos resultados obtidos com testes do protótipo OPERA original. Ainda nesses textos, encontram-se mais informações sobre a interface XOPERA original.

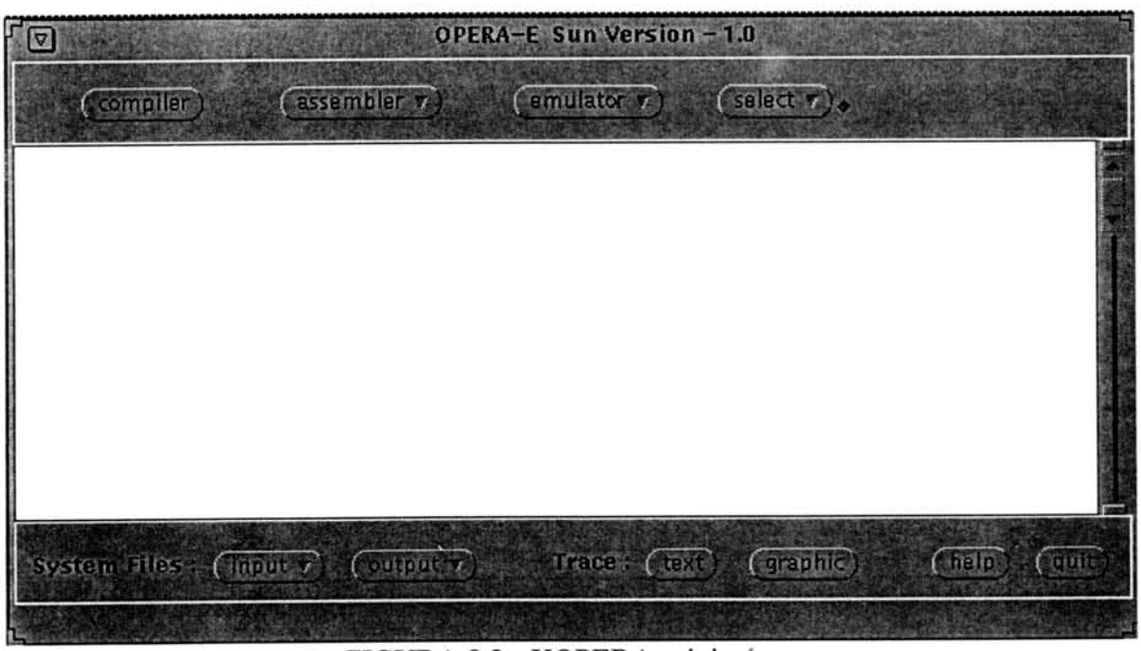


FIGURA 8.2 - XOPERA original

8.2 Visão Geral da Integração

A subseção 3.5.1 apresenta uma proposta de aplicação do GRANLOG no auxílio a decisões de escalonamento. Por sua vez, a figura 3.3 mostra os três níveis de abstração (anotação, compilação e execução) envolvidos nessa aplicação. A integração OPERA-GRANLOG pode ser compreendida através dessa proposta. Conforme apresentado na figura 3.3, o GRANLOG mantém-se num alto nível de abstração (anotação). Por outro lado, conforme discutido na seção 8.1, o OPERA é um modelo para execução de programas em lógica em ambientes distribuídos. Sendo assim, o OPERA encontra-se em baixo nível de abstração (execução). Com base nessas informações, a figura 3.3 pode ser adaptada para a realidade da integração OPERA-GRANLOG, conforme mostra a figura 8.3.

A figura 8.3 mostra que o usuário fornece o programa para o GRANLOG, onde será realizada a análise de granulosidade. Dessa análise, resulta o programa granulado. Esse programa é entregue ao compilador que possui como principal responsabilidade a compilação das cláusulas para geração de código WAM. Além disso, o compilador processa a anotação de granulosidade, gerando as informações que são utilizadas no ambiente de execução OPERA para controle da granulosidade. De posse do código WAM e das informações de granulosidade, o OPERA pode realizar a execução do programa.

FIGURA 8.3 - Visão geral da integração OPERA-GRANLOG

8.3 Utilização das Informações de Granulosidade

Conforme citado na seção 8.1, a extensão da WAM proposta pelo OPERA contém uma nova estrutura denominada Pilha de Objetivos Paralelos (POP). Considerando-se a arquitetura de processos do OPERA, a POP está localizada no *Solver* (figura 8.2). Portanto, cada trabalhador possui uma POP.

Durante a execução de um programa, o processo *Solver* coloca na POP todos os objetivos que **podem** ser executados em paralelo. Entretanto, a **possibilidade** de execução paralela de um procedimento não significa que a paralelização **deverá** ser

realizada. A paralelização de um procedimento implica vários custos. Estes custos podem ocasionar perda de desempenho da execução paralela em relação à execução seqüencial. Um procedimento somente deverá ser executado em paralelo quando sua granulosidade for maior do que os custos envolvidos na paralelização. Esta situação recebe o nome de **problema granulosidade versus custo**. Para solucionar este problema, pode-se simplesmente comparar a granulosidade com os custos de execução paralela e decidir se o objetivo deve ser paralelizado.

Basicamente, a proposta de integração OPERA-GRANLOG consiste na utilização pelo OPERA das informações fornecidas pelo GRANLOG, com o intuito de solucionar o problema granulosidade versus custo. A solução deste problema reflete diretamente no aumento da eficiência na execução paralela dos programas em lógica.

Conforme apresentado na figura 8.3, o OPERA recebe como entrada o código WAM executável e informações de granulosidade, as quais são utilizadas no decorrer da execução para resolver o problema granulosidade versus custo. O código WAM é obtido através da compilação das cláusulas e contém instruções seqüenciais e paralelas que descrevem o fluxo de execução. Dentre as instruções paralelas, destaca-se a instrução *push_call*, a qual é responsável pelo empilhamento na POP dos objetivos que podem ser executados em paralelo. As informações de granulosidade são obtidas pelo compilador através da análise das anotações (*granularity*, *out_size* e *grain_goal*) geradas pelo GRANLOG. Estas informações são organizadas num arquivo conforme mostra a figura 8.4, onde cada **entrada** equivale à descrição de uma chamada paralela realizada através de uma instrução *push_call*. Portanto, para cada *push_call* no código WAM existirá uma entrada no arquivo.

FIGURA 8.4 - Formato do arquivo de informações de granulosidade

Existem dois tipos de entradas, ou seja, **Entrada de Tamanho Limite** (ETL) e **Entrada de Custo Limite** (ECL). A figura 8.5 mostra a estrutura de cada uma destas entradas. A ETL armazena a descrição de uma chamada paralela (*push_call*) na qual a granulosidade depende apenas de um argumento. A ECL armazena a descrição de uma chamada paralela na qual a granulosidade depende de mais de um argumento.

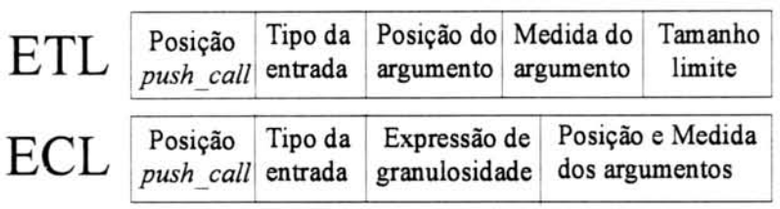


FIGURA 8.5 - Tipos de entradas do arquivo de informações de granulosidade

Constata-se na figura 8.5, que os dois tipos de entrada possuem os mesmos primeiros dois campos. O primeiro campo armazena a posição no código WAM da instrução *push_call* que realiza a chamada paralela descrita pela entrada. O conteúdo deste campo identifica de forma unívoca a chamada paralela, pois cada *push_call* possui um único lugar no código WAM. O segundo campo identifica o tipo da entrada, ou seja, ETL ou ECL. O terceiro campo da ETL indica a posição no literal do único argumento que influencia na granulosidade. O quarto campo indica a medida que deverá ser utilizada para obter o tamanho do argumento. O quinto e último campo possui o **Tamanho Limite** do argumento para realização da paralelização. Este campo contém o tamanho do argumento, a partir do qual a paralelização deve ser realizada. Este valor é obtido pelo compilador através da manipulação da expressão de granulosidade e do **Custo Limite**, ou seja, o custo para paralelização de um objetivo no ambiente OPERA. Portanto, para resolver o problema granulosidade versus custo, basta obter o tamanho do argumento indicado pelo terceiro campo utilizando a medida indicada pelo quarto campo e compará-lo com o tamanho limite armazenado no quinto campo. Se o tamanho do argumento for maior do que o tamanho limite, o objetivo deve ser colocado na POP.

O terceiro campo da ECL contém a expressão de granulosidade da chamada descrita pela entrada. Neste caso, para solucionar o problema granulosidade versus custo, basta resolver a expressão de granulosidade e comparar com o custo limite. Se a granulosidade for maior do que o custo limite, o objetivo deve ser colocado na POP. O quarto e último campo da ECL contém, em seqüência, as posições e medidas de tamanho dos argumentos que influenciam na granulosidade.

Analisando-se a descrição da ECL e da ETL, constata-se que a ETL é uma especialização da ECL para casos onde a granulosidade sofre influência de apenas um argumento. Esta simplificação permite que em tempo de execução, as entradas ETL sejam resolvidas com apenas uma comparação, sem a necessidade de resolver expressões.

Atualmente, o protótipo OPERA possui suporte para controle de granulosidade. A introdução desse controle ocasionou duas alterações no protótipo, ambas no código do processo *Solver*. Em primeiro lugar, sempre que uma nova aplicação é carregada para execução, o processo *Solver* carrega também as informações de granulosidade da aplicação, as quais estão armazenadas num arquivo. Este arquivo segue a descrição apresentada nas figuras 8.4 e 8.5. Em segundo lugar, o código que implementa a instrução *push_call* foi alterado para solução do problema granulosidade versus custo. Na versão original do protótipo, a instrução *push_call* simplesmente empilha um objetivo na POP, sem considerações quanto a granulosidade e custo. A nova versão implementa o algoritmo apresentado na figura 8.6.

```
se Tamanho do Argumento > Tamanho Limite então
   Empilha objetivo na POP (paralelização)
senão
   Executa o objetivo localmente
```

FIGURA 8.6 - Algoritmo para controle de granulosidade no OPERA

O algoritmo da figura 8.6 demonstra que atualmente o protótipo OPERA manipula apenas entradas ETL. Sendo assim, o controle de granulosidade somente pode ser realizado para procedimentos que tenham sua complexidade dependente de apenas um argumento de entrada. Diversos procedimentos enquadram-se nesse padrão. Por

exemplo, analisando-se as expressões de complexidade mostradas na figura 6.4, constata-se que a complexidade dos procedimentos *append*, *nrev*, *fibonacci* e *hanoi* depende de apenas um argumento de entrada.

O **Tamanho do Argumento** somente poderá ser obtido em tempo de execução. A dificuldade para obtenção desse tamanho depende da medida empregada. Por exemplo, se o argumento for um inteiro, o tamanho é o valor numérico do argumento e pode ser obtido diretamente nos registradores da WAM. Por outro lado, se o argumento for uma lista, a obtenção do tamanho envolverá uma análise mais complexa. Em [HER94] é proposta uma metodologia para realização dessa análise. Atualmente, o protótipo OPERA somente executa em paralelo procedimentos com argumentos do tipo inteiro.

O **Tamanho Limite** para um procedimento depende do custo para paralelização de um objetivo no ambiente OPERA (**Custo Limite**). Basicamente, esse custo consiste do tempo necessário para troca de mensagens entre os trabalhadores durante o processos de paralelização de um procedimento. Para realização desse processo, o OPERA utiliza quatro mensagens, conforme mostra a figura 8.7

FIGURA 8.7 - Troca de mensagens para paralelização no OPERA

A primeira mensagem consiste da **exportação do trabalho**, ou seja, o exportador envia para o importador os dados necessários para execução do procedimento. A segunda mensagem contém um **aviso de sucesso ou falha** na execução do procedimento. A terceira mensagem é uma **requisição dos resultados**, ou seja, o exportador solicita ao importador os resultados do processamento. Finalmente, o importador envia os **resultados**. Obviamente, o custo envolvido na troca de mensagens depende do montante de informações a serem trocadas entre trabalhadores, ou seja, depende do tamanho das mensagens. As únicas mensagens que variam de tamanho são a **exportação de trabalho** e os **resultados**. O tamanho da mensagem **exportação de trabalho** está relacionado com o tamanho dos argumentos de entrada do procedimento. Esse tamanho pode ser obtido em tempo de execução. O tamanho da mensagem **resultados** está relacionada com o tamanho dos argumentos de saída do procedimento. Esse tamanho pode ser obtido, antes da paralelização do procedimento, através das relações armazenadas nas anotações *out_size* fornecidas pelo GRANLOG. Resumindo, o custo para paralelização de um procedimento pode variar de acordo com o tamanho dos seus argumentos de entrada e saída, os quais somente podem ser determinados em tempo de execução. Além disso, esse custo pode ser previsto, antes da execução paralela, através das relações de tamanho entre entradas e saídas. Saumya Debray ([DEB95a])

concorda com essa abordagem e acrescenta que essas relações podem ser utilizadas para geração de **expressões de cálculo de custo de paralelização em ambientes distribuídos**.

Atualmente o protótipo OPERA somente exporta procedimentos com argumentos do tipo inteiro. Portanto, o volume de informações trocadas entre trabalhadores não varia de forma significativa. Sendo assim, o custo para paralelização de um procedimento pode ser considerado constante. Em horário de baixa utilização da rede na UFRGS e utilizando duas estações SUN SLC, foram realizados diversos testes que permitiram o dimensionamento desse custo. **O protótipo OPERA possui um Custo Limite de 0.013455 segundos**. Com base nesse custo, pode-se determinar o Tamanho Limite para os procedimentos a serem executados no OPERA. Por exemplo, para o procedimento *fibonacci* pode ser realizada a seguinte análise.

Na tabela 6.3 obtem-se que na WAM seqüencial utilizada no OPERA uma resolução do *fibonacci* possui uma complexidade de 1.26 milisegundos. Além disso, conforme demonstrado em [YAM94], a degradação de desempenho da execução do *fibonacci* na WAM paralela em relação a execução na WAM seqüencial é aproximadamente 20 %. Desta forma, a complexidade de uma resolução do *fibonacci* na WAM paralela é aproximadamente 1.512 ms. Portanto, o custo limite em resoluções pode ser obtido pelo seguinte cálculo:

Custo Limite em Resoluções* = *Custo Limite em Segundos* / *Complexidade de uma Resolução em Segundos

***Custo Limite em Resoluções* = 0.013455 / 0.001512 = 8.8988 resoluções**

Além disso, na figura 6.4 obtem-se a expressão de complexidade para o procedimento *fibonacci*.

$$C_{fibonacci} = 1.45 * \exp(1.62,\$1) + 0.55 * \exp(-0.62,\$1) - 1$$

Essa expressão determina a complexidade em resoluções. Portanto, com base no **Custo Limite em Resoluções** e na **Expressão de Complexidade**, determina-se o **Tamanho Limite** para o argumento de entrada que influencia na complexidade (*$1*).

$$C_{fibonacci} > \text{Custo Limite em Resoluções}$$

$$1.45 * \exp(1.62,\$1) + 0.55 * \exp(-0.62,\$1) - 1 > 8.8988$$

$$\text{Se } \$1 = 3 \text{ então } C_{fibonacci} = 5 \text{ resoluções}$$

$$\text{Se } \$1 = 4 \text{ então } C_{fibonacci} = 9 \text{ resoluções}$$

Portanto, o **Tamanho Limite** para o procedimento *fibonacci* é 3, ou seja, o procedimento *fibonacci* somente deverá ser exportado (colocado na POP) quando o tamanho do seu argumento de entrada for maior do que 3. Quanto isso ocorrer, a complexidade para execução do procedimento supera o custo para sua paralelização. O texto [LIN93] apresenta uma metodologia para determinação automática do **Tamanho Limite**.

8.4 Inclusão do GRANLOG na Interface XOPERA

Durante o desenvolvimento deste trabalho foram modeladas e implementadas alterações na interface XOPERA para gerenciar o GRANLOG e controlar a granulosidade no OPERA. A figura 8.2 mostra a interface XOPERA original. Conforme discutido na seção 8.1, esta interface foi projetada originalmente para gerenciar a execução dos módulos componentes do OPERA. A figura 8.8 apresenta a nova interface XOPERA. Esta interface inclui dois novos botões, ou seja: botão ***granlog*** e botão ***config***.

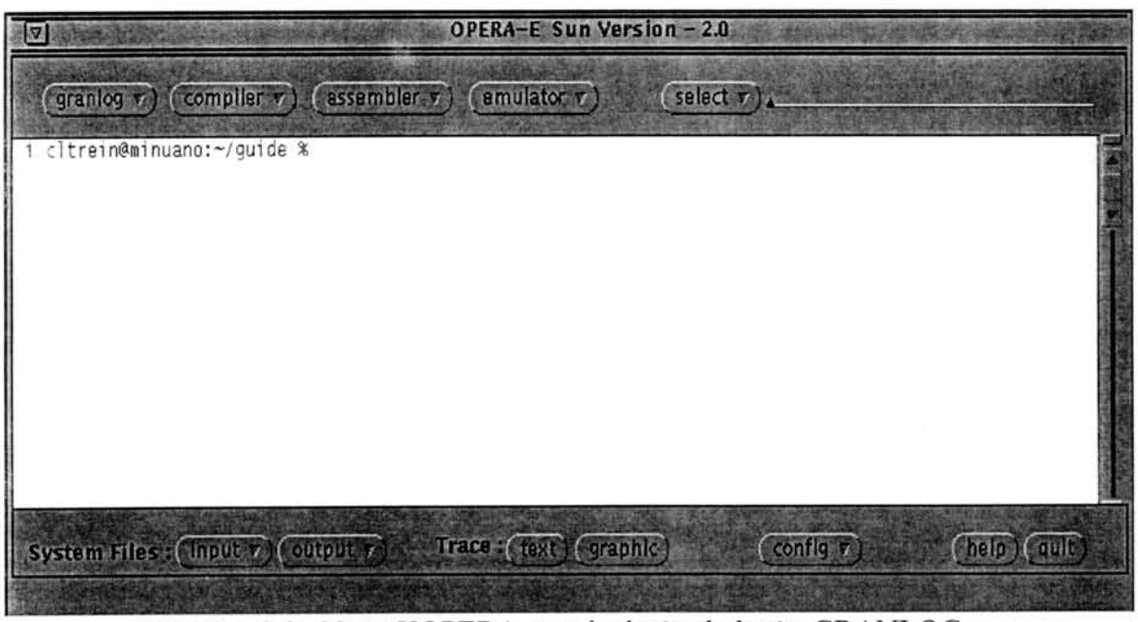


FIGURA 8.8 - Nova XOPERA com inclusão do botão GRANLOG

O botão ***granlog*** executa o protótipo GRANLOG. Desta execução resulta o programa granulado. Conforme mostra a figura 8.3, o programa granulado pode ser analisado pelo compilador para geração de código WAM e informações de granulosidade. O compilador é executado pelo botão ***compiler***. Por sua vez, o botão ***config*** permite a configuração do GRANLOG e dos três módulos componentes do OPERA. Através deste botão, o usuário acessa o menu de configuração. Conforme mostra a figura 8.9, este menu possui quatro opções, ou seja: *granlog*, *compiler*, *assembler* e *emulator*. Cada uma destas opções acessa uma janela onde o usuário pode configurar o módulo escolhido.

Atualmente, apenas o emulador possui uma janela de configuração. Conforme mostra a figura 8.10, esta janela permite dois tipos de configuração, ou seja, ambiente de execução (*execution environment*) e controle de granulosidade (*granularity control*). Através do botão *machines* o usuário configura o ambiente de execução, ou seja,

determina quais estações de trabalho farão parte da arquitetura OPERA. Esta tarefa é realizada através de diversas telas de seleção, as quais não serão discutidas neste trabalho. Por sua vez, o controle de granulosidade é configurado através da opção *granularity control*. O usuário pode optar por ligar ou desligar esse controle. Se a opção estiver marcada, o controle de granulosidade será realizado durante a execução dos programas.

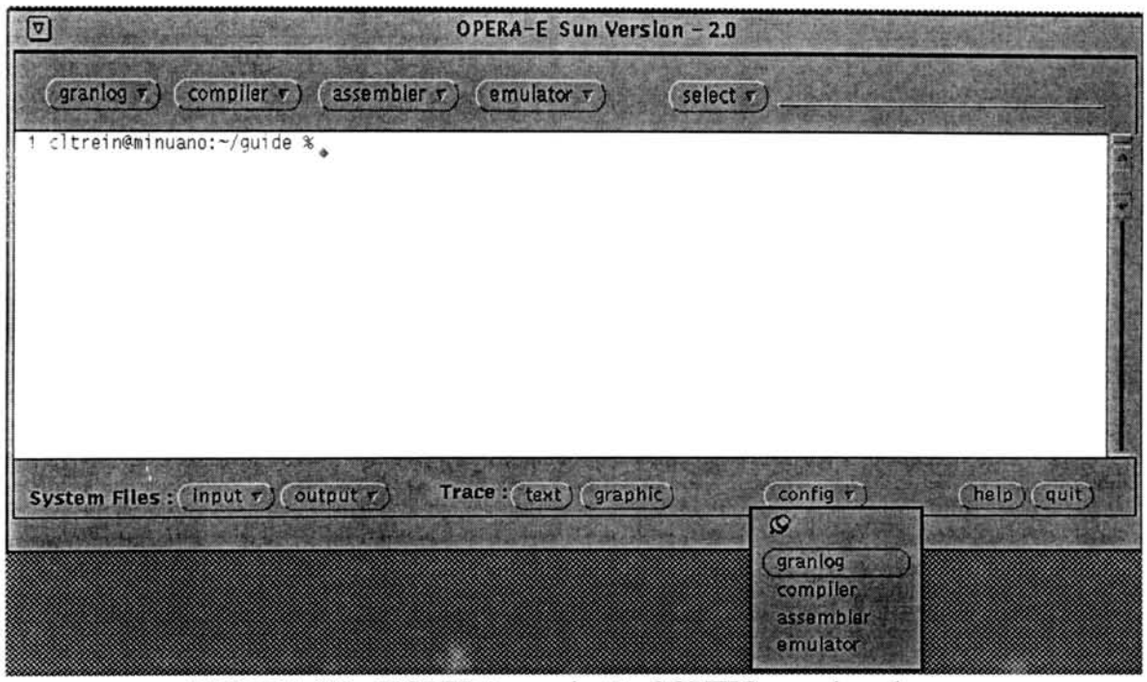


FIGURA 8.9 - XOPERA com botão CONFIG pressionado

As alterações da XOPERA propostas neste trabalho permitem a gerência integrada dos protótipos OPERA e GRANLOG. A nova interface permite a execução da análise de granulosidade (botão ***granlog*** no menu principal) e a gerência do controle de granulosidade no OPERA (opção *granularity control* na janela de configuração do emulador). Além disso, faz parte da nova proposta de interface, a filosofia de configuração individual dos módulos gerenciados pela XOPERA. A inclusão do botão ***config*** concretiza esta nova filosofia.

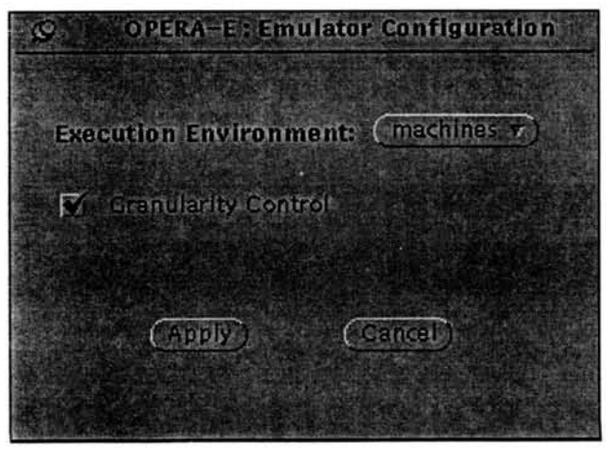


FIGURA 8.10 - Janela de configuração do Emulador

8.5 Conclusões

A integração OPERA-GRANLOG permite a união de dois modelos independentes, visando a exploração eficiente do paralelismo na programação em lógica. Além disso, a aplicação no OPERA das informações geradas pelo GRANLOG, sedimenta os dois modelos. Para o GRANLOG, fica demonstrada a importância de parte das informações disponíveis na anotação de granulosidade. Quanto ao OPERA, fica demonstrado que seu modelo suporta controle de granulosidade, o qual tornou-se nos últimos anos um centro de atenções da comunidade científica.

As principais conclusões desse capítulo são as seguintes:

- o modelo OPERA suporta controle de granulosidade;
- as informações de granulosidade fornecidas pelo GRANLOG podem ser utilizadas no OPERA para controlar a granulosidade;
- as alterações para inclusão do controle de granulosidade no protótipo OPERA restringem-se ao código do processo *Solver*;
- o ponto de controle da granulosidade localiza-se no código da instrução *push_call* onde ocorre o empilhamento de objetivos paralelos na POP;
- o custo de paralelização no protótipo OPERA atual é constante;
- a manipulação das expressões de complexidade em conjunto com o custo de paralelização permite a determinação do **Tamanho Limite** dos argumentos.

Futuros trabalhos poderão aprimorar e expandir a integração OPERA-GRANLOG. As informações armazenadas na anotação *out_size* não foram consideradas na proposta apresentada neste capítulo. Quando o protótipo OPERA suportar o envio de listas entre trabalhadores, essas informações poderão ser utilizadas para dimensionar o custo de comunicação. Esse custo será variável de acordo com o tamanho das mensagens. O dimensionamento antecipado (antes da paralelização) do tamanho da mensagem **Resultados** dependerá das informações disponíveis na anotação *out_size*. Além disso, pode-se explorar as informações de granulosidade para aprimorar as regras de escalonamento do OPERA. Conforme discutido na seção 8.1, estas regras utilizam como métrica para decisões de exportação/importação o número de objetivos armazenados na POP. Uma nova abordagem poderá considerar como métrica a granulosidade dos objetivos armazenados na pilha, o que permitirá maior precisão nas decisões de escalonamento. Outro trabalho futuro, será a modelagem e implementação do controle de granulosidade OU no OPERA. Esse controle utilizará as informações de granulosidade OU disponíveis no programa granulado. A modelagem do controle de granulosidade OU depende da criação de uma metodologia para exploração do paralelismo OU no OPERA. Destaca-se ainda como tarefa futura, uma avaliação da eficiência da integração OPERA-GRANLOG. Essa tarefa será realizada através da execução de programas no protótipo OPERA, com e sem controle de granulosidade. Sendo assim, o funcionamento estável desse protótipo é indispensável. Atualmente o protótipo OPERA apresenta problemas que inviabilizam essa avaliação. O próximo capítulo apresenta as conclusões finais deste trabalho.

9 Conclusões

O tema proposto neste trabalho possibilitou o estudo da análise automática de granulosidade na programação em lógica. Neste escopo foram pesquisados diversos tópicos dentre os quais destacam-se: paralelismo em geral, análise de granulosidade, programação em lógica, paralelismo na programação em lógica, análise de granulosidade nos programas em lógica, análise de modos, análise de tipos, análise de medidas de tamanho de termos, análise de dependências, análise global, interpretação abstrata, análise de grãos e análise de complexidade. Como decorrência desse estudo, surge o modelo denominado GRANLOG (**GR**anulartiy **AN**alisys for **LOG**ic Programming). Este modelo serviu e continuará servindo de suporte para organização do estudo e concretização do aspecto criativo da pesquisa. Com base no modelo, criou-se o protótipo GRANLOG. Esse protótipo coloca em prática grande parte dos conceitos teóricos criados durante a modelagem. O principal objetivo da prototipação foi a validação do modelo. No entanto, a existência do protótipo servirá como fonte de estímulo para desenvolvimento de trabalhos futuros. Como última tarefa, foi criada e implementada a proposta de integração OPERA-GRANLOG. A criação dessa proposta exigiu um estudo profundo do modelo OPERA e do seu protótipo. Com base neste estudo, foram identificados no OPERA os aspectos a serem considerados na integração. Além disso, foram projetadas e implementadas as alterações na interface XOPERA. Essas alterações introduziram na interface o suporte para execução do GRANLOG e controle da granulosidade no OPERA.

No decorrer do texto foram apresentadas conclusões ao final de cada capítulo. Esta metodologia aumentou a coesão dos demais capítulos e diminuiu a complexidade desse capítulo. Além disso, as conclusões ficaram contextualizadas, ou seja, cada conclusão está localizada no escopo onde foi obtida. Como complemento, nos próximos parágrafos são apresentadas as principais conclusões inferidas nesse trabalho.

O capítulo 2 abordou genericamente a programação em lógica e o paralelismo. Neste capítulo, concluiu-se que, atualmente, a **análise de granulosidade** é um dos principais tópicos de pesquisa na área do processamento paralelo. O autor dessa dissertação está convicto de que a exploração *eficiente* do paralelismo nos computadores passa necessariamente pelo aprofundamento dos estudos sobre análise de granulosidade. Além disso, no capítulo 2 surge a convicção de que existe uma forte tendência para adoção do **paralelismo implícito** e conseqüentemente para **exploração automática do paralelismo**. Finalmente; concluiu-se que a **análise combinada** (compilação e execução) vem sendo considerada pela comunidade científica como a melhor opção para modelagem da análise de granulosidade na programação em lógica.

No capítulo 3 foi apresentada uma visão geral do modelo GRANLOG. Neste capítulo afirmou-se que o **modelo proposto é genérico** o bastante para ser aplicado em várias áreas de pesquisa da programação em lógica. Além disso, concluiu-se que o GRANLOG trouxe diversas **contribuições** para o estudo da análise de granulosidade nos programas em lógica. Como conclusão final, destacou-se que a **organização modular** do GRANLOG facilita sua manutenção e adaptação para futuras aplicações.

Por sua vez, o capítulo 4 discutiu a análise global. Destaca-se como principal conclusão deste capítulo a importância da inclusão da **interpretação abstrata no AGL**. Neste sentido, concluiu-se ainda que o **aperfeiçoamento da análise de dependências**,

através da interpretação abstrata, refletirá na precisão do restante da análise de granulosidade.

No capítulo 5 foi abordada a análise de grãos. Durante este capítulo concluiu-se que a **análise de grãos pode ser realizada com precisão em tempo de compilação** desde que seja conciliada com uma análise global apurada. Além disso, no capítulo 5 surge a **nova definição de paralelismo E** proposta pelo GRANLOG. Finalmente, em consonância com as conclusões do capítulo anterior, concluiu-se que a análise global tornará a **anotação de grãos mais precisa**.

O capítulo 6 apresentou a análise de complexidade. Dentre diversas conclusões, salientou-se que os estudos sobre **análise de complexidade E** já alcançaram resultados animadores, mas a **análise de complexidade OU** ainda é um tópico de estudo pouco pesquisado. Ainda neste capítulo, concluiu-se que a **análise de complexidade é importante para a análise de granulosidade**, pois a identificação dos grãos visando a máxima eficiência possui como base a comparação entre custos de paralelização e complexidade dos grãos (granulosidade). Outra importante constatação deste capítulo, reside no fato de que as **anotações de complexidade podem ser utilizadas em diversas análises**, dentre as quais destacam-se a pesquisa do comportamento de programas e a exploração do paralelismo na programação em lógica.

No sétimo capítulo discutiu-se o protótipo GRANLOG. Inicialmente, esse capítulo concluiu que o **protótipo possui uma estrutura semelhante ao modelo**, ou seja, um organização em três módulos. Além disso, ressaltou-se que o protótipo implementa apenas os aspectos do modelo que poderiam ser utilizados para **controlar a granulosidade no modelo OPERA atual**. Ainda no capítulo 7, encontra-se a afirmação de que as **estruturas de dados foram generalizadas** ao máximo visando futuras aplicações.

A integração OPERA-GRANLOG foi discutida no capítulo 8. Neste capítulo concluiu-se que o **modelo OPERA suporta controle de granulosidade**. Concluiu-se ainda que **parte das informações de granulosidade fornecidas pelo GRANLOG encontram atualmente aplicação no OPERA**. No oitavo capítulo surge a constatação de que o **custo de paralelização no protótipo OPERA atual é constante**. Sendo assim, diversas simplificações podem ser feitas no processo de controle da granulosidade.

Futuros trabalhos poderão explorar novas capacidades e aplicações para o modelo proposto. Em primeiro lugar, devem ser direcionados esforços para **criação de uma análise global automática**. Nessa análise, devem ser inferidos modos, tipos, medidas e dependências. Desta forma, as anotações de usuário discutidas no capítulo 4 poderão ser eliminadas. Espera-se assim, aumentar a precisão e a simplicidade de uso do GRANLOG. Pretende-se ainda, **aprimorar a análise de complexidade**. Neste sentido, devem ser acompanhadas as evoluções do sistema CASLOG (análise de complexidade E) e aprofundados os estudos sobre análise de complexidade OU. Segundo Nai-Wei Lin ([LIN93]), dentre os próximos passos da evolução do CASLOG encontram-se o tratamento de argumentos de saída abertos, o aperfeiçoamento das medidas de tamanho de termos e a análise de complexidade do melhor caso e caso médio. No âmbito da complexidade OU, deve ser criado maior intercâmbio com outros grupos de pesquisa dedicados à pesquisa nessa área. Outro interessante trabalho futuro, será uma **avaliação da influência do tamanho das mensagens nos custos de paralelização em sistemas distribuídos**. Esta avaliação envolverá as informações disponíveis na anotação *out_size*

e servirá de base para expansão da proposta de integração OPERA-GRANLOG. Além disso, devem ser realizados **testes para avaliação da influência do controle de granulosidade na eficiência dos ambientes de execução paralela de programas em lógica**. Esta tarefa consistirá basicamente na comparação dos resultados obtidos na execução paralela de diversos programas no ambiente OPERA, com e sem utilização do controle de granulosidade.

Quanto à evolução do protótipo, estão previstas duas importantes tarefas. Em primeiro lugar, deve ser **introduzida na análise de grãos o tratamento dos grãos-metas**. Nesta fase, a anotação de grãos será acrescida da anotação *grain_goals*. Em segundo lugar, deve ser **implementada a análise de granulosidade OU**. Nesta fase, a anotação de grãos será acrescida da anotação *grain_clause* e a anotação de complexidade será acrescida das informações sobre complexidade OU. Quanto ao estudo de aplicações, devem ser direcionados esforços em dois sentidos, ou seja, aplicação do GRANLOG na **simulação da execução de programas** (subseção 3.5.2) e aplicação na **reorganização do código fonte de programas** visando aprimorar a exploração do paralelismo.

Atualmente, o processamento paralelo é considerado a melhor solução para aumento do desempenho dos sistemas computacionais. Sendo assim, essa nova tecnologia está sendo adotada como caminho para evolução dos computadores. No entanto, diversas dificuldades ainda devem ser vencidas para adoção definitiva do paralelismo. Grande parte dessas dificuldades reside na criação dos *softwares* paralelos. A solução da atual *crise do software paralelo* depende do surgimento de novas metodologias para manipulação de programas em sistemas paralelos. Cumprindo sua função natural, a comunidade científica da área de computação vem dedicando contínuos esforços para desenvolvimento destas novas metodologias. O GRANLOG faz parte desses esforços, contribuindo para criação de sistemas paralelos eficientes e confortáveis.

Anexo 1 Programas Utilizados no Teste do CASLOG

1.1 Programas em *Sicstus*

1.1.1 Programa para Concatenação de Listas

```
main :-
        nl,write('APPEND PARA SICSTUS'), nl,nl,
        write('Tamanho da lista: '), read(T),
        lista(T,L),
        statistics(runtime,[_,_]),
        append(L,[1],R1),
        statistics(runtime,[_,Dt]),
        nl, write('TEMPO = '), write(Dt), nl.


append([],L,L).
append([HL],L1,[HR]) :-
        append(L,L1,R).


lista(0,[]) :- !.
lista(N,[1C]) :-
        M is N - 1,
        lista(M,C).
```

1.1.2 Programa para Reversão de Listas

```
main :-
        nl,write('NREV PARA SICSTUS'), nl,nl,
        write('Tamanho da lista: '), read(T),
        lista(T,L),
        statistics(runtime,[_,_]),
        nrev(L,R),
        statistics(runtime,[_,Dt]),
        nl, write('TEMPO = '), write(Dt), nl.


nrev([],[]).
nrev([HL],R) :-
        nrev(L,R1),
        append(R1,[H],R).


append([],L,L).
append([HL],L1,[HR]) :-
        append(L,L1,R).


lista(0,[]) :- !.
lista(N,[1C]) :-
        M is N - 1,
        lista(M,C).
```

1.1.3 Programa para Cálculo de Números de Fibonacci

```
main :-
        nl, write('FIBONACCI PARA SICSTUS'), nl, nl,
        write('Numero de fibonacci: '), read(F),
        statistics(runtime,[_,_]),
        fib(F,R),
        statistics(runtime,[_,Dt]),
        nl, write('TEMPO = '), write(Dt), nl.


fib(0,0).
fib(1,1).
fib(M,N) :- M > 1,
        M1 is M-1,
        M2 is M-2,
        fib(M1,N1),
        fib(M2,N2),
        N is N1+N2.
```

1.1.4 Programa para solução do Problema Torre de Hanói

```
main :-
        nl,write('HANOI PARA SICSTUS'), nl, nl,
        write('Numero de pecas: '), read(P),
        statistics(runtime,[_,_]),
        hanoi(P,p1,p2,p3,R),
        statistics(runtime,[_,Dt]),
        nl, write('TEMPO = '), write(Dt), nl.


hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
        N > 1, N1 is N - 1,
        hanoi(N1,A,C,B,M1),
        hanoi(N1,B,A,C,M2),
        append(M1,[mv(A,C)],T),
        append(T,M2,M).


append([],L,L).
append([HL],L1,[HR]) :-
        append(L,L1,R).
```

1.2 Programas em *C-Prolog*

1.2.1 Programa para Concatenação de Listas

```
main :-
       nl,write('APPEND PARA C-PROLOG'), nl,nl,
       write('Tamanho da lista: '), read(T),
       lista(T,L),
       Ti is cputime,
       append(L,[1],R),
       To is cputime,
       Dt is To - Ti,
       nl, write('TEMPO = '), write(Dt), nl.


append([],L,L).
append([HL],L1,[HR]) :-
      append(L,L1,R).


lista(0,[]) :- !.
lista(N,[1C]) :-
                M is N - 1,
                lista(M,C).
```

1.2.2 Programa para Reversão de Listas

```
main :-
        nl,write('NREV PARA C-PROLOG'), nl,nl,
        write('Tamanho da lista: '), read(T),
        lista(T,L),
        Ti is cputime,
        nrev(L,R1),
        To is cputime,
        Dt is To- Ti,
        nl, write('TEMPO = '), write(Dt), nl.


nrev([],[]).
nrev([HL],R) :-
        nrev(L,R1),
        append(R1,[H],R).


append([],L,L).
append([HL],L1,[HR]) :-
        append(L,L1,R).


lista(0,[]) :- !.
lista(N,[1C]) :-
        M is N - 1,
        lista(M,C).
```

1.2.3 Programa para Cálculo de Números de Fibonacci

```
main :-
        nl, write('FIBONACCI PARA C-PROLOG'), nl, nl,
        write('Numero de fibonacci: '), read(F),
        Ti is cputime,
        fib(F,R),
        To is cputime,
        Dt is To - Ti,
        nl, write('TEMPO = '), write(Dt), nl.


fibo(0,0).
fibo(1,1).
fibo(M,N) :- M > 1,
        M1 is M-1,
        M2 is M-2,
        fib(M1,N1),
        fib(M2,N2),
        N is N1+N2.
```

1.2.4 Programa para solução do Problema Torre de Hanói

```
main :-
        nl,write('HANOI PARA C-PROLOG'), nl, nl,
        write('Numero de pecas: '), read(P),
        Ti is cputime,
        hanoi(P,p1,p2,p3,R),
        To is cputime,
        Dt is To - Ti,
        nl, write('TEMPO = '), write(Dt), nl.


hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
        N > 1, N1 is N - 1,
        hanoi(N1,A,C,B,M1),
        hanoi(N1,B,A,C,M2),
        append(M1,[mv(A,C)],T),
        append(T,M2,M).


append([],L,L).
append([HL],L1,[HR]) :-
        append(L,L1,R)
```

1.3 Programas em *PROLOG/OPERA*

1.3.1 Programa para Concatenação de Listas

```
main :-
        nl,write('APPEND PARA OPERA'), nl,nl,
        lista(100,L),
        nl, statistics, nl,
        append(L,[1],R),
        nl, statistics, nl.


append([],L,L).
append([HL],L1,[HR]) :-
        append(L,L1,R).


lista(0,[]) :- !.
lista(N,[1C]) :-
        is(M,N,-,1),
        lista(M,C).
```

1.3.2 Programa para Reversão de Listas

```
main :-
        nl,write('NREV PARA OPERA'), nl,nl,
        lista(10,L),
        nl, statistics, nl,
        nrev(L,R),
        nl, statistics, nl.


nrev([],[]).
nrev([HL],R) :-
        nrev(L,R1),
        append(R1,[H],R).

append([],L,L).
append([HL],L1,[HR]) :-
        append(L,L1,R).


lista(0,[]) :- !.
lista(N,[1C]) :-
        is(M,N,-,1),
        lista(M,C).
```

1.3.3 Programa para Cálculo de Números de Fibonacci

```
main :-
        nl, write('FIBONACCI PARA OPERA'), nl, nl,
        F = 10,
        nl, statistics, nl,
        fib(F,R),
        nl, statistics, nl.


fib(0,0).
fib(1,1).
fib(M,N) :- M > 1,
        is(M1,M,-,1),
        is(M2,M,-,2),
        fib(M1,N1),
        fib(M2,N2),
        is(N,N1,+,N2).
```

1.3.4 Programa para solução do Problema Torre de Hanói

```
main :-
       nl,write('HANOI PARA OPERA'), nl, nl,
       P = 5,
       nl,statistics, nl,
       hanoi(P,p1,p2,p3,R),
       nl, statistics, nl.


hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
       N > 1, is(N1,N,-,1),
       hanoi(N1,A,C,B,M1),
       hanoi(N1,B,A,C,M2),
       append(M1,[mv(A,C)],T),
       append(T,M2,M).


append([],L,L).
append([HL],L1,[HR]) :-
       append(L,L1,R).
```

Bibliografia

[AIT 91] AÏT-KACI, Hassan. **Warren's Abstract Machine - A Tutorial Reconstruction**. Cambridge: MIT Press, 1991. 114p.

[AZZ 88] AZZOUNE, H. Type Inference in Prolog. In: INTERNATIONAL CONFERENCE ON AUTOMATED DEDUCTION, 9., 1988, **Proceedings...** Berlin: Springer-Verlag, 1988. 775p. p.258-277. (Lecture Notes in Computer Science).

[AZZ 89] AZZOUNE, H. **Les types en Prolog. Un sisteme d'inference de type et es ses applications**. Grenoble:[s.n], 1989. Tese de Doutorado.

[BAC 78] BACKUS, J. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. **Communications of the ACM**, New York, v.21, n.8, p.613-641, Aug. 1978.

[BAL 89] BAL, H.E.; STEINER, J. G.; TANENBAUM, A. S. Programming Languages for Distributed Computing Systems. **ACM Computing Surveys**, New York, v.21, n.3, p.261-322, Sept. 1989.

[BAR 93] BARBOSA, J.L.V. **Construção de Compiladores para Máquinas de Processamento Paralelo**. Pelotas: UCPel, 1993. 73p. Monografia de Especialização.

[BAR 94] BARBOSA, J.L.V. **Análise da Granulosidade de Tarefas para Processamento Paralelo**. Porto Alegre: CPGCC-UFRGS, 1994. 93p. (Trabalho Individual, 376).

[BAR 94a] BARBOSA, J.L.V.; WERNER, O.; GEYER, C. F. R. Automatic Granularity Analysis in Logic Programming. In: LOGIC PROGRAMMING WORKSHOP (WLP 94), 10., 1994, Zurich. **Proceedings...** Zurich: Institut für Informatik der Universität Zürich, 1994. 185p. p.85-88.

[BAR 95] BARBOSA, J.L.V.; GEYER, C. F. R. GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 10., 1995, Canela. **Anais...** Porto Alegre: SBC, 1995. 639p. p.61-75.

[BAR 95a] BARBOSA, J.L.V.; GEYER, C. F. R. Integração OPERA-GRANLOG: Aplicação da Análise de Granulosidade na Execução Paralela de Programas em Lógica. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 22., 1995, Canela. **Anais...** Porto Alegre: SBC, 1995. 1482p. p.189-200.

[BIA 90] BIANCHINI, R. Introdução à Execução Paralela de Programas Lógicos. **Revista Brasileira de Computação**, Rio de Janeiro, v.6, n.1, p.15-32, jul./set. 1990.

[BRI 90] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Opera: a Parallel Prolog System and its Implementation on Supernode**. Grenoble: Laboratoire de Genie Informatique de Grenoble/CAP-Gemini-Innovation, 1990.19p. (Technical report).

[BRI 90a] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Scheduling of OR-parallel Prolog on Scalable, Reconfigurable, Distributed-Memory Multiprocessor**. Grenoble: Laboratoire de Genie Informatique de Grenoble/CAP-Gemini-innovation, 1990. 18p. (Technical Report).

[BRO 94] BROWNE, J.C. et al. **Visual Programming and Parallel Computing**. [S.l.]: University of Tennessee, April 1994. (Technical Report CS-94-229).

[BRU 87] BRUYNOOGHE, M. et al. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In: SYMPOSIUM ON LOGIC PROGRAMMING, 1987, San Francisco. **Proceedings...** New York: IEEE Press, 1987. p.192-204.

[BRU 87a] BRUYNOOGHE, M. **A Framework for the Abstract Interpretation of Logic Programs**. [S.l.]: Katholieke Universiteit Leuven - Department of Computer Science, 1987. (Technical report).

[BUE 93] BUENO, F. et al. **The AND-Prolog Compiler System - Automatic Parallelization Tools for LP**. Madrid: Universidad Politécnica de Madrid, 1993. 40p. (Tecnical Report - DIA/CLIP5/93.0).

[BUE 94] BUENO, F.; LA BANDA, M G.; HERMENEGILDO, M. A Comparative Study of Methods for Automatic Compile-Time Parallelization of Logic Programs. In: PASCO, 1994. **Proceedings...** [S.l.]: World Publishing Company, 1994.

[CAR 89] CARDELLI, L. Typeful Programming. In: IFIP ADVANCED SEMINAR ON FORMAL DESCRIPTION OF PROGRAMMING CONCEPTS, 1989. **Proceedings...** [S.l.:s.n.],1989.

[CAR 92] CARLSSON, M.; WIDEN, J. et al. **SICStus Prolog User's Manual**. Kista: Swedish Institute of Computer Science, August 1992. 170p.

[CAR 93] CARRO, M.; GÓMEZ, L.; HERMENEGILDO, M. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 10., 1993. **Proceedings...** Cambridge: MIT Press, 1993.

[CAS 86] CASANOVA, M.A.; GIORNO, F.A.C.; FURTADO, A.L. **Programação em Lógica**. Belo Horizonte: UFMG, 1986. 295p. Escola de Computação, 5., 1986, Belo Horizonte.

[CHA 85] CHANG, J.; DESPAIN, A.M.; DEGROOT, D. AND-parallelism of logic programs based on a static data dependency analysis. In: SPRING IEEE COMPUTER SOCIETY INTERNATIONAL CONFERENCE (COMPCON SPRING), 1985, San Francisco. **Proceedings...** New York: IEEE, 1985. p.218-225.

[CHA 85a] CHANG, J.; DESPAIN, A.M. Semi-intelligent backtracking of Prolog based on static data dependency analysis. **IEEE Software**, New York, v.1, n.6, p.10-21, 1985.

[CHU 41] CHURCH, A. The Calculi of Lambda Conversion. In: ANNALS OF MATHEMATICS STUDIES, 1941. **Proceedings...** Princeton: Princeton University Press, 1941.

[CLA 86] CLARK, K.; GREGORY, S. PARLOG: Parallel Programming in Logic. **ACM TOPLAS**, New York, v.8, n.1, p.76-91, Jan. 1986.

[CLO 81] CLOCKSIN, W.F.; MELLISH, C.S. **Programming in Prolog**. Berlin: Springer-Verlag, 1981. 189p.

[CLO 87] CLOCKSIN, W. A Prolog Primer. **BYTE**, Peterborough, NH, v.12, n.9, p.147-158, Aug. 1987.

[CON 85] CONERY, J.S.; KIBLER, D. F. AND Parallelism and Nondeterminism in Logic Programs. **New Generation Computing**, Berlin, v.3, n.1, p.43-70, 1985.

[COU 77] COUSOT, P.; COUSOT, R. Abstract Interpretation: A Unified Lattice Model for Static of Programs by Construction or Approximation of Fixpoints. In: SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 4., 1977. **Proceedings...** New York: ACM Press, 1977. p.238-252.

[DEB 88] DEBRAY S. K. Unfold/fold Transformations and Loop Optimization of Logic Programs. **SIGPLAN Notices**, New York, v.23, n.7, p.297-307, 1988. Trabalho apresentado na ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, 1988, Atlanta.

[DEB 89] DEBRAY, S. K. Static Inference of Modes and Data Dependencies in Logic Programs. **ACM Transactions on Programming Languages and Systems**, New York, v.11, n.3, p.418-450, July 1989.

[DEB 89a] DEBRAY, S. K.; WARREN, D. S. Functional Computations in Logic Programs. **ACM Transactions on Programming Languages and Systems**, New York, v.11, n.3, p.418-450, July 1989.

[DEB 90] DEBRAY, Saumya K.; HERMENEGILDO, Manuel; LIN, Nai-Wei. Task Granularity Analysis in Logic Programs. **SIGPLAN Notices**, New York, v.25, n.6, p.174-188, 1990. Trabalho apresentado na ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 1990, New York.

[DEB 93] DEBRAY, S. K.; LIN, N. Cost Analysis of Logic Programs. **ACM Transactions on Programming Languages and Systems**, New York, v.15, n.5, p.826-875, Nov. 1993.

[DEB 94] DEBRAY, S. K.; RAMAKRISHNAN, R. Abstract Interpretation of Logic Program Using Magic Transformations. **Journal of Logic Programming**, New York, v.18, n.2, p.149-176, Feb. 1994.

[DEB 94a] DEBRAY, S. K. et al. Estimating the Computacional Cost of Logic Programs. In: INTERNATIONAL STATIC ANALYSIS SYMPOSIUM, 1994, Namur, Belgium. **Proceedings...** Berlin: Springer-Verlag, 1994. (Lecture Notes in Computer Science, v.864).

[DEB 95] DEBRAY, S. K. **What's Granularity Analysis?** Tucson: University of Arizona, fevereiro 1995. (debray@cs.arizona.edu). Comunicação pessoal através da Internet.

[DEB 95a] DEBRAY, S. K. **Avaliação de Custos de Comunicação em Sistemas Distribuídos.** Porto Alegre: Universidade Federal do Rio Grande do Sul, julho 1995. (debray@cs.arizona.edu). Comunicação pessoal numa visita ao II/UFRGS.

[DEG 84] DEGROOT, Doug. Restricted And-Parallelism. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, 1984, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1984. p.471-478.

[DEG 87] DEGROOT, Doug. A Technique for compiling Execution Graph Expressions for Restricted And-Parallelism in Logic Programs. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 1987, Athens. **Proceedings...** Berlin: Springer-Verlag, 1987. p.1074-1093. (Lecture Notes in Computer Science, v.297)

[DEG 89] DEGROOT, Doug. Restricted AND-Parallelism and Side Effects in Logic Programming. In: HWANG, K.; DEGROOT, Doug (Eds.). **Parallel Processing for Supercomputers and Artificial Intelligence**. New York: McGraw-Hill, 1989. p.487-522

[DIE 88] DIETRICH, R. **Modes and Types for Prolog**. [S.l.]: Arbeitspapiere der GMD, January 1988. (Tecnical Report, 285).

[DIE 88a] DIETRICH, R. A Polymorphic Type System with Subtypes for Prolog. In: EUROPEAN SYMPOSIUM ON PROGRAMMING, 2., 1988. **Proceedings...** Berlin: Springer-Verlag, 1988. p.79-93. (Lecture Notes in Computer Science, n.300)

[ESC 89] ESCARDO, M.H. **Um Sistema de Tipos para Prolog**. Porto Alegre: CIC da UFRGS, 1989. Trabalho de Diplomação.

[FRÜ 89] FRÜHWIRTH, T. W. Type inference by program transformations and partial evaluation. In: INTERNATIONAL CONFERENCE AND SYMPOSIUM ON LOGIC PROGRAMMING, 1989, London. **Proceedings...** Cambridge: MIT Press, 1989. p.263-282.

[GAL 91] GARCIA, L. C. **Técnicas para Compilação em Paralelo**. Porto Alegre: CPGCC da UFRGS, Janeiro 1991. Trabalho de Disciplina.

[GAR 94] GARCÍA, Pedro L.; HERMENEGILDO, Manuel; DEBRAY, S. K. Towards Granularity Based Control of Parallelism in Logic Programs. In: INTERNATIONAL SYMPOSIUM ON PARALLEL COMPUTATION, 1994, Linz, Austria. **Proceedings...** [S.l.:s.n.], 1994.

[GEL 86] GELERNTER, David. Domesticating Parallelism. **Computer**, New York, v.19, n.8, Aug. 1986.

[GER 93] GERASOULIS, A.; YANG, T. On the Granularity and Clustering of Directed Acyclic Task Graphs. **IEEE Transactions on Parallel and Distributed Systems,** New York, v.4, n.6, p.686-701, June 1993.

[GEY 91] GEYER, Cláudio F. R. **Une Contribution a L'Etude du Parallelisme OU en Prolog sur des Machines sans Mémoire Commune**. Grenoble: Université Joseph Fourier, 1991. PhD Thesis.

[GEY 92] GEYER, Cláudio; YAMIN, Adenauer; WERNER, Otilia. Projeto OPERA: Um Modelo E/OU para Prolog. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 1992, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 1992. 390p. p.269-281.

[GIA 92] GIACOBAZZI, R.; DEBRAY, S. K.; LEVI, G. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, 1992, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1992.

[GIR 92] GIRKAR, M.; POLYCHRONOPOULOS, C. D. Automatic Extraction of Functional Parallelism from Ordinary Programs. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.3, n.2, Mar. 1992.

[GRE 87] GREGORY, S. **Parallel Logic Programming in Parlog - The Language and its Implementation**. New York: Addison-Wesley, 1987. 267p.

[GUP 93] GUPTA, G.; JAYARAMAN, B. AND-OR Parallelism on Shared-Memory Multiprocessors. **Journal of Logic Programming**, New York, v.17, n.1, p.59-89, Oct. 1993.

[HER 86] HERMENEGILDO, M. **An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation on Logic Programs in Parallel**. Austin: University of Texas, August 1986. 244p. PhD Thesis.

[HER 89] HERMENEGILDO, M.; ROSSI, F. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In: NORTH AMERICAN CONFERENCE ON LOGIC PROGRAMS, 1989. **Proceedings...** Cambridge: MIT Press, 1989. p.369-390.

[HER 90] HERMENEGILDO, M.; ROSSI, F. &-Prolog and its Performance: Exploiting Independente And-Parallelism. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 7., 1990. **Proceedings...** Cambridge: MIT Press, 1990. p.325-339.

[HER 91] HERMENEGILDO, M. V.; GREENE, K. J. The &-Prolog System: Exploiting Independent And-Parallelism. **New Generating Computing**, Berlin, v.9, n.3-4, p.233-256, 1991.

[HER 93] HERMENEGILDO, M. **Futuro da Exploração do Paralelismo na Programação em Lógica.** Porto Alegre: Universidade Federal do Rio Grande do Sul, março 1993. (herme@fi.upm.es). Comunicação pessoal numa visita ao II/UFRGS.

[HER 94] HERMENEGILDO, M.; GARCÍA P. L. **A Technique for Dynamic Term Size Computation via Program Transformation**. Madrid: Universidad Politécnica de Madrid, 1994. 20p. (Technical Report - CLIP 8/93.1).

[HUD 85] HUDAK, P.; GOLDBERG, B. Serial Combinators: Optimal Grains of Parallelism. In: FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE, 1985, Nancy, France. **Proceedings...** Berlin: Springer-Verlag, 1985. p.382-399. (Lecture Notes in Computer Science, v.201).

[HWA 84] HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing**. New York: McGraw-Hill, 1984. 846p.

[JAC 90] JACOBS, D. Type Declaration as Subtype Constraints in Logic Programs. **SIGPLAN Notices**, New York, v.25, n.6, p.166-173, 1990. Trabalho apresentado na ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 1990, New York.

[JAJ 92] JAJA, Joseph. **An Introduction to Parallel Algorithms**. Reading: Addison-Wesley, 1992. 566p.

[JON 87] JONES, N.; SONDERGAARD, H. A semantics-based framework for the abstract interpretation of prolog. In: ABSTRACT INTERPRETATION OF DECLARATIVE LANGUAGES, 1987. **Proceedings...** [S.l.:s.n.], 1987. p.124-142

[JON 88] JONES-PAGE, M. **Projeto Estruturado de Sistemas**. São Paulo: McGraw-Hill, 1988. 396p.

[KAL 85] KALÉ, L. V. **Parallel Architectures for Problem Solving**. Stony Brook: SUNY, 1985. PhD Thesis.

[KAL 87] KALÉ, L. V. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In: SYMPOSIUM ON LOGIC PROGRAMMING, 4., 1987. **Proceedings...** San Francisco: IEEE, 1987. p.125-133.

[KAR 88] KARP, A. H.; BABB II, R. G. A Comparison of 12 Parallel Fortran Dialects. **IEEE Software**, New York, p.52-67, Sept. 1988.

[KAS 83] KASIF, S.; KOHLI, M.; MINKER, J. **PRISM: a Parallel Inference System for Prolog Solving**. Lisboa: Universidade de Nova Lisboa, 1983. 23p. (Relatório Técnico).

[KAY 95] KAYSER, Patrícia; GEYER, Cláudio. **Implementação de um Analisador de Granulosidade para Prolog**. Porto Alegre: CPGCC da UFRGS, 1995. 71p. Projeto de Diplomação.

[KAY 95a] KAYSER, Patrícia; BARBOSA, Jorge; GEYER, Cláudio. Protótipo GRANLOG: Implementação de um Analisador de Granulosidade para Prolog no Projeto OPERA. In: SALÃO DE INICIAÇÃO CIENTÍFICA, 1995, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 1995. p.50.

[KIN 90] KING, A.; SOPER, P. Granularity Control for Concurrent Logic Programs. In: INTERNATIONAL SYMPOSIUM ON COMPUTER AND INFORMATION SCIENCES, 5., 1990, Nevsehir, Turkey. **Proceedings...** [S.l.:s.n.], 1990.

[KIN 92] KING, A.; SOPER, P. Schedule Analysis of Concurrent Logic Programs. INTERNATIONAL CONFERENCE AND SYMPOSIUM ON LOGIC PROGRAMMING, 1992, Washington. **Proceedings...** [S.l.:s.n.], 1992. p.478-492.

[KOW 74] KOWALSKI, R. A. Predicate Logic as a Programming Language. In: IFIP WORLD CONGRESS, 1974. **Proceedings...** [S.l.]:North-Holland, 1974. p.589-674

[KOW 79] KOWALSKI, Robert. **Logic for Problem Solving**. New York: Elsevier, 1979.

[KRU 88] KRUATRACHUE, B.; LEWIS, T. Grain Size Determination for Parallel Processing. **IEEE Software**, New York, p.23-32, Jan. 1988.

[LEI 92] LEIGHTON, Frank T. **Introduction to Parallel and Architectures:** Arrays, Trees, Hypercubes. [S.l.]: Morgan Kanfmann, 1992. 831p.

[LIN 93] LIN, N. **Automatic Complexity Analysis of Logic Programs**. Tucson: University of Arizona, 1993. 244p. Ph.D. Thesis.

[LIN 94] LIN, N. **Granularity Analysis**. Tucson: University of Arizona, maio 1994. (naiwei@cs.arizona.edu). Comunicação pessoal através da Internet.

[LIN 95] LIN, N. **What's Granularity Analysis?** Tucson: University of Arizona, fevereiro 1995. (naiwei@cs.arizona.edu). Comunicação pessoal através da Internet.

[MAN 87] MANNILA, H.; UKKONEN, E. Flow Analsys of Prolog programs. In: SYMPOSIUM ON LOGIC PROGRAMMING, 4., 1987, San Francisco. **Proceedings...** New York: IEEE Press, 1987.

[MAS 91] MASON, Tony; BROWN, Doug. **Lex & Yacc**. [S.l.]: O'Reilly & Associates, 1991. 216p.

[MCC 89] MCCREARY, C.; GILL, H. Automatic Determination of Grain Size for Efficient Parallel Processing. **Communications of the ACM,** New York, v.32, n.9, p.1073-1078, Sept. 1989.

[MCJ 65] MCCARTHY, J. et al. **LISP 1.5 Programmer's Manual**. Cambridge: MIT Press, 1965.

[MEL81] MELLISH, C. S. **The Automatic Generation of Mode Declarations for Prolog Programs**. Edinburgh: University of Edinburgh, August 1981. (DAI Research Paper 163).

[MEL 85] MELLISH, C. S. Some global otimizations for a Prolog compiler. **Journal of Logic Programming**, New York, v.2, n.1, p.43-66, Apr. 1985.

[MEL 86] MELLISH, C. S. Abstract Interpretation of Prolog Programs. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 3., 1986. **Proceedings...** Berlin: Springer-Verlag, 1986. p.463-475. (Lecture Notes in Computer Science, n.225).

[MIS 84] MISHRA, P. Toward a Theory of Types in Prolog. In: INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, 1984, San Francisco. **Proceedings...** New York: IEEE Press, 1984. p.289-298.

[MOL 93] MOLDOVAN, D. I. **Parallel Processing from Applications to Systems**. San Mateo: Morgan Kaufmann, 1993.

[MOR 94] MORA, Michael C. **Correção de Erros em um Ambiente de Depuração Inteligente para Prolog**. Porto Alegre: CPGCC da UFRGS, 1994. 78p. Dissertação de Mestrado.

[MUT 90] MUTHUKUMAR, K.; HERMENEGILDO, M. The CDG, UDG, and MEL Methods for Automatic Compile-Time Parallelization of Logic Programs for Independent And-parallelism. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 1990. **Proceedings...** Cambridge: MIT Press, 1990. p.221-237.

[MYC 84] MYCROFT, A.; O'KEEDE, A. A Polymorphic Type System for Prolog. **Artificial Intelligence**, Amsterdam, v.23, n.3, p.295-307, 1984.

[PAA 88] PAAKI, J. A Note on the Speed of Prolog. **SIGPLAN Notices**, New York, v.23, n.8, 1988.

[PAD 86] PADUA, D. A.; WOLFE, M. J. Advanced Compiler Optimizations for Supercomputers. **Communications of the ACM**, New York, v.29, n.12, p.1184-1201, Dec. 1986.

[PAN 91] PANCAKE, C. M. Where are we headed?. **Communications of the ACM,** New York, v.34, n.11, p.53-64, Nov. 1991.

[PER 78] PEREIRA, L. M.; PEREIRA, F. C. N.; WARREN, D. H. D. **User's Guide to DECsystem-10 Prolog**. Edinburgh: University of Edinburgh- Departament of Artificial Intelligence. 1978. 197p.

[PER 87] PEREIRA, Fernando; TWEED Christopher. **C-Prolog User's Manual Versions 1.5 and 1.5+**. Edinburgh: University of Edinburgh-Departament of Architecture, 1987. 64p.

[PHU 92] PHUONG, L. N. **Un Systeme Declaratif de Types Pour Prolog**. Grenoble: Institut National Polytechnique de Grenoble, 1988. 113p. PHD Thesis.

[QUI 87] QUINN, M. J. **Designing Efficient Algorithms for Parallel Computers**. New York: McGraw-Hill, 1983. 288p.

[RAB 90] RABHI, F. A.; MANSON, G. A. **Using Complexity Functions to Control Parallelism in Functional Programs**. [S.l.]: University of Sheffield, 1990. (Research Report CS-90-1)

[RED 84] REDDY, U. S. Transformation of Logic Programs to Functional Programs. In: INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, 1984, Atlantic City. **Proceedings...** New York: IEEE Press, 1984. p.187-196.

[REI 88] REINTJES, P. B. A VLSI Design Environment in Prolog. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 5., 1988, Seattle. **Proceedings...** [S.l.:s.n.], 1988. p.70-81.

[RIB92] RIBEIRO, A. M.; MÓRA, M. C. Utilização de Tipos na Depuração de Programas Prolog. In: SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 9., 1992, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 1992. p.38-51.

[RIB 92a] RIBEIRO, A. M. **Técnicas de Depuração Inteligente Aplicadas à Linguagem Prolog**. Porto Alegre: CPGCC da UFRGS, 1994. 145p. Dissertação de mestrado.

[ROB 92] ROBINSON, J. A. Logic and Logic Programming. **Communications of the ACM**, New York, v.35, n.3, p.40-65, Mar. 1992.

[ROU 69] ROUSSEL, P. **Prolog:** Manuel de Reference et d'Utilisation. [S.l.]: Univ. d'Aix-Marseille, Groupe de IA, 1975. (Technical report).

[SAR 89] SARKAR,V. **Partitioning and Scheduling Parallel Programs for Multiprocessors**. Cambridge: MIT Press, 1989.

[SCH 93] SCHLABITZ, A. **Análise de Grão e Analisador Léxico-Sintático da Linguagem Prolog**. Porto Alegre: CIC-UFRGS, 1993. 99p. Trabalho de Diplomação.

[SHA 86] SHAPIRO, E. Concurrent Prolog: A Progress Report. **Computer**, New York, v.19, n.8, p.44-58, Aug. 1986.

[SHA 89] SHAPIRO, E. The Family of Concurrent Logic Programming Languages. **ACM Computing Surveys**, New York, v.21, n.3, p.413-510, Sept. 1989.

[SHA 93] SHAPIRO, E.; WARREN, D. H. D. The Fifth Generation Project. **Communications of the ACM,** New York, v.36, n.3, Mar. 1993.

[SHU 89] SHU, N. C. Visual programming: Perspectives and Approaches. **IBM Systems Journal,** New York, v.28, n.4, p.525-547, 1989.

[SHI 90] SHIN, K. G.; CHEN, M. On the Number of Acceptable Task Assignments in Distributed Computing Systems. **IEEE Transactions on Computers,** New York, v.39, n.1, p.99-110, Jan. 1990.

[STE 86] STERLING, L.; SHAPIRO, E. **The Art of Prolog**. Cambridge: MIT Press, 1986.

[TIC 88] TICK, E. Compile-Time Granularity Analysis for Parallel Logic Programming Languages. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, 1988, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1988.

[TIC 90] TICK, E. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. **New Generation Computing,** Berlin, v.7, n.2-3, p.325-337, 1990.

[TIC 93] TICK, E; ZHONG, X. A Compile-Time Granularity Analysis Algorithm and Its Performance Evaluation. **New Generating Computing,** Berlin, v.11, n.3-4, p.271-295, 1993.

[TIC 94] TICK, E. **Granularity Analysis**. Eugene: University of Oregon, março 1994. (tick@asterix.cs.uoregon.edu). Comunicação pessoal através da Internet.

[TIC 95] TICK, E. **What's Granularity Analysis.** Eugene: University of Oregon, fevereiro 1995. (tick@asterix.cs.uoregon.edu). Comunicação pessoal através da Internet.

[VAN 87] VAN ROY, P.; DEMOEN, B.; WILLEMS, Y. D. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In: TAPSOFT 1987, Pisa. **Proceedings...** [S.l.:s.n.], 1987.

[VLA 92] VLAHAVAS, I; KEFALAS, P. A Parallel Prolog Resolution Based on Multiple Unifications. **Parallel Computing**, New York, v.18, n.11, p.1275-1283, Nov. 1992.

[VER 91] VERSCHAETSE, K.; DE SCHREYE, D. Deriving Termination Proofs for Logic Programs Using Abstract Procedures. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 8., 1991, Paris. **Proceedings...** Cambridge: MIT Press, 1991. p.301-315.

[XUJ 88] XU, J.; WARREN, D. S. A type inference system for PROLOG. In: INTERNATIONAL CONFERENCE AND SYMPOSIUM ON LOGIC PROGRAMMING, 5., 1988. **Proceedings...** Cambridge: MIT Press, 1988. p.605-619.

[WAD 77] WARREN, D. **Implementing Prolog - Compiling Predicate Logic Programs**. Edinburgh: University of Edinburgh, 1977. (DAI Research Report 39).

[WAR 88] WARREM, R.; HERMENEGILDO, M.; DEBRAY, S. K. On the practicality of global flow analysis of logic programs. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 5., Aug. 1988. **Proceedings...** Cambridge: MIT Press, 1988.

[WER 94] WERNER, O. **Uma Máquina Abstrata Estendida para o Paralelismo E na Programação em Lógica**. Porto Alegre: CPGCC-UFRGS, 1994. 145p. Dissertação de Mestrado.

[WER 94a] WERNER, O. et al. OPERA Project: an Approach Towards Parallelism Exploitation on Logic Programming. In: LOGIC PROGRAMMING WORKSHOP (WLP 94), 10., 1994, Zurich. **Proceedings...** Zurich: Institut für Informatik der Universität Zürich, 1994. 185p. p.29-32.

[WIL 93] WILSON, G. V. A Glossary of Parallel Computing Terminology. **IEEE Parallel & Distributed Technology**, New York, p.52-67, Feb. 1993.

[YAM 92] YAMIN, Adenauer C. **Modelos de Implementação do Paralelismo OU na Programação em Lógica**. Porto Alegre: CPGCC da UFRGS, 1992. 114p. (Trabalho Individual, 280).

[YAM 93] YAMIN, Adenauer C.; WERNER, Otilia; GEYER, Cláudio F. R. Prolog Paralelo em Rede de Computadores. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 5., 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. 775p. p.290-303.

[YAM 94] YAMIN, Adenauer C. **Um Ambiente Para Exploração de Paralelismo na Programação em Lógica**. Porto Alegre: CPGCC da UFRGS, 1994. 204p. Dissertação de Mestrado.

[YAR 87] YARDENI, E.; SHAPIRO, E. A Type System for Logic Programs. **Concurrent Prolog - Collected Papers.** Cambridge: MIT Press, 1987. v.2, p.211-244.

[ZHO 92] ZHONG, X. et al. Towards an efficient compile-time granularity analysis algorithm. In: GENERATION COMPUTER SYSTEMS, 5., 1992, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1992. p.809-816.

[ZIM 91] ZIMA, H.; CHAPMAN, B. **Supercompilers for Parallel and Vector Computers**. New York: ACM Press, 1991. 376p.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica

por

Jorge Luis Victória Barbosa

Dissertação apresentada aos Senhores:

Profa. Dra. Ana Maria de Alencar Price

Prof. Dr. Celso Maciel da Costa

Prof. Dr. Flávio Moreira de Oliveira (PUC-RS)

Prof. Dr. Antônio Carlos da Rocha Costa

Vista e permitida a impressão.
Porto Alegre, 12 / 09 / 97 .

Prof. Dr. Cláudio Fernando Resin Geyer,
Orientador.

Prof. Flávio Rech Wagner
Coordenador do Curso de Pós-Graduação
em Ciência da Computação - UFRGS
Instituto de Informática - UFRGS