UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Execução Distribuída de Programas Funcionais usando a Máquina Virtual Java

por

ANDRÉ RAUBER DU BOIS

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

Prof. Dr. Antônio Carlos da Rocha Costa
Orientador

Porto Alegre, março de 2001

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Du Bois, André Rauber

Execução Distribuída de Programas Funcionais usando a Máquina Virtual Java / por André Rauber Du Bois. — Porto Alegre: PPGC da UFRGS, 2001.

91 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2001. Orientador: Costa, Antônio Carlos da Rocha.

1. Implementação de Linguagens Funcionais. 2. Programação Funcional Paralela. 3. Java. I. Costa, Antônio Carlos da Rocha. II. Título.

Agradecimentos

Agradeço a todos que de alguma forma ajudaram na realização deste trabalho, seja direta ou indiretamente. Em especial:

- Ao Vanderlei M. Rodrigues por toda a ajuda e incentivo neste trabalho. Valeu Vandi!
- Ao meu orientador, o prof. Antônio Carlos da Rocha Costa, por ter acreditado neste trabalho.
- À Graçaliz Dimuro, Renata Reiser, Marilton Aguiar e ao pessoal do GMFC por me acompanharem durante toda a caminhada, desde a graduação.
- Aos Feldens, que me aguentaram nesses dois anos!
- Ao pessoal do Instituto de Informática da UFRGS, funcionários e professores, por me acolherem.
- Aos alunos: Mônica, Simone, Júlio e Daniela Bagatini que tornaram os dias no Instituto de Infomática muito mais divertidos!
- Aos professores Adenauer, Jorge Barbosa, Geyer e Tiarajú. Valeu!

Por último, agradeço a pessoa que mais me ajudou nesses dias de estudo, a minha namorada Juliana Vizzotto. *Jú, obrigado por me fazer tão feliz*!

Este trabalho é dedicado ao Pai Paulo,
à Mãe Lili e ao Irmão Mateus,
sempre presentes em todos
os momentos.

Sumário

Lista de Abreviaturas

CAF	Constant Application Form
FIFO	First In First Out
JNI	Java Native Interface
JVM	Máquina Virtual Java
NF	Forma Normal
PE	Processing Element
WHNF	Weak Head Normal Form

Lista de Figuras

Lista de Tabelas

Resumo

O objetivo deste trabalho é apresentar a implementação em Java de uma máquina abstrata para execução distribuída de programas funcionais. Mostra-se como as facilidades da linguagem Java foram utilizadas para a implementação de uma linguagem funcional paralela que roda os programas funcionais de forma distribuída em uma rede de computadores.

Linguagens Funcionais geralmente são implementadas usando uma máquina abstrata para a execução dos programas. Essas máquinas são usualmente máquinas de redução de grafos. Para se rodar os programas funcionais na máquina virtual Java implementou-se a máquina de redução de grafos G-Machine em Java.

Nesta dissertação, apresenta-se inicialmente a implementação da G-Machine em Java, realizada como primeira etapa do trabalho e discute-se a abordagem utilizada para essa implementação. Mostra-se em seguida, como os programas funcionais podem ser compilados para rodar nessa G-Machine. Na segunda etapa do trabalho, modifica-se o sistema implementado para permitir a execução distribuída dos programas funcionais. Finalmente apresenta-se uma avaliação de desempenho e mostra-se possíveis trabalhos futuros.

Palavras-chave: Implementação de Linguagens Funcionais, Programação Funcional Paralela, Java.

TITLE: "DISTRIBUTED EXECUTION OF FUNCTIONAL PROGRAMS USING THE JVM"

Abstract

The objective of this work is to present the implementation in the Java language of an abstract machine for distributed execution of functional programs. We show how the Java facilities were used to implement a parallel functional programming language with a distributed runtime system.

Functional Languages are usually implemented using an abstract machine to execute programs. These abstract machines are usually graph reduction machines. To run the functional programs on the Java Virtual Machine we have implemented the G-Machine graph reduction machine in Java.

In this text, we first present the implementation of the G-Machine in Java, and discuss its implementation. Then we show how this implementation was modified to allow distributed execution of functional programs. Finally some benchmarks, possible future works and conclusions are presented.

Keywords: Implementation of Functional Languages, Parallel Functional Programming, Java.

1 Introdução

Este trabalho trata sobre a execução de linguagens funcionais puras e não-estritas de forma distribuída utilizando a máquina Virtual Java - *Java Virtual Machine* (JVM). Mostra-se, com esse trabalho, que a linguagem Java possui várias características que facilitam a implementação de uma linguagem funcional que avalia os seus programas de forma paralela/distribuída.

1.1 Linguagens Funcionais e Programação Paralela

As linguagens de programação tradicionais, chamadas de imperativas, têm por característica de programação explorar a arquitetura das máquinas. Por isso um programa consiste basicamente de instruções para modificar a memória que são executadas pela unidade de processamento. Uma desvantagem de se usar esta estratégia para a resolução de problemas é que o programador acaba se distanciando do problema para se preocupar com as características da máquina.

Certamente existiam outras maneiras de se solucionar problemas antes da invenção do computador. Na matemática, seria o caso das funções. Uma função gera um resultado dependendo de seus parâmetros. Essa é a idéia da programação funcional: utilizar o conceito matemático de funções para construir programas. Um programa seria uma função que de acordo com seus parâmetros devolveria uma resposta. Este estilo de programação dispensa o programador de conhecimentos sobre a arquitetura da máquina. Devido ao fato de envolver conceitos familiares para qualquer pessoa que tenha um conhecimento básico de matemática, essa maneira de programar é de fácil compreensão por estudantes que não possuem contato com os conceitos de arquitetura de computadores.

Na programação funcional, cada função opera como uma *caixa preta* com um comportamento bem definido, e o programa funcional é baseado na composição dessas funções puras e sem estado. Consequentemente, uma função não gera *efeitos colaterais* e o resultado de uma aplicação de função depende apenas de seus argumentos e não do contexto em que a aplicação de função está sendo avaliada. A chamada de uma função com os mesmos argumentos sempre devolverá o mesmo valor, o que é chamado de *transparência referencial* [PLA93]. A transparência referencial permite que se substitua a chamada de uma função por seu corpo e vice-versa. Com isso é possível construir-se provas matemáticas em cima das definições do programa funcional [THO99].

Hoje em dia, os computadores além de serem usados para tarefas triviais como edição de textos e envio de e-mails, estão cada vez mais sendo usados para controlar tarefas críticas como trafego aéreo ou procedimentos médicos. Por isso é muito importante o controle de erros nos programas. Provar-se matematicamente propriedades dos programas ajuda a aumentar a confiabilidade dos mesmos.

A programação funcional ainda fornece uma nova e interessante maneira de modularizar programas, pois uma função pode ser construída através de outras funções, de modo que um problema pode ser dividido em vários problemas menores.

Hughes [HUG89] sugere que o poder das linguagens funcionais parte principalmente de duas novas maneiras que elas propiciam de se modularizar e agrupar programas: *funções de alta ordem* e *avaliação preguiçosa* ou *procrastinada* (*lazy*

evaluation).

Funções de alta ordem são funções em que um de seus argumentos, o seu resultado, ou ambos, são uma função. O exemplo clássico é a função `map` que recebe como argumento uma função e uma lista e aplica esta função a todos os elementos da lista.

Na estratégia de avaliação preguiçosa, os argumentos das funções são avaliados apenas uma vez e apenas quando necessários. Linguagens funcionais também lidam com dados de forma *lazy*, onde apenas as partes necessárias da estrutura de dados são geradas, o que permite se trabalhar com estruturas infinitas. Nas linguagens procedurais, geralmente os argumentos de um programa são avaliados em sua totalidade antes de serem passados para o mesmo, impossibilitanto dessa maneira a utilização de estruturas infinitas.

A avaliação preguiçosa também proporciona uma ferramenta importante para a modularização de programas. O programador pode definir uma abstração que usa um conjunto de dados grande ou infinito e ter certeza de que a função cliente apenas irá construir as partes necessárias para a sua computação. Por exemplo, pode-se definir uma árvore que gere todos os movimentos possíveis em um tabuleiro de xadrez e utilizar-se uma função com um algoritmo do tipo A* que, através de uma heurística, encontra a melhor sequência de jogadas para se vencer a partida.

A transparência referencial e a falta de efeitos colaterais permitem que as expressões no programa funcional não tenham uma ordem fixa para avaliação, pois uma mudança na ordem não pode mudar os valores ou resultados da avaliação. Por isso diz-se que as linguagens funcionais permitem a sua execução de maneira fácil em sistemas multi-processados, pois as tarefas podem ser associadas a diferentes processadores, sem o perigo dos efeitos colaterais.

Com a disseminação do uso de computadores em nossa sociedade, torna-se cada vez maior o nível de complexidade dos problemas a serem solucionados. Dessa maneira, acaba-se ultrapassando o limite da tecnologia disponível nas máquinas sequenciais existentes nos dias de hoje. Como alternativa para o aumento do poder computacional, surgem as arquiteturas paralelas. A idéia é de se utilizar um conjunto de processadores que cooperam entre si para resolver uma determinada tarefa.

Porém, junto com as arquiteturas paralelas surgem as dificuldades para programá-las. Por exemplo, em linguagens paralelas imperativas, interfaces de comunicação e sincronização devem ser definidas entre as tarefas paralelas para assegurar a sua correta interação. O programador também é responsável por forçar a proteção de certas áreas de memória para prevenir o *deadlock* [TAN97]. Para se atingir estes objetivos, novas estruturas de programação devem ser inseridas na linguagem de maneira a explicitar e administrar o paralelismo, o que torna a programação muito mais complexa, além dos programas ficarem mais difícies de se entender.

Uma vantagem de se programar arquiteturas paralelas usando linguagens funcionais puras, como dito anteriormente, é que a falta de efeitos colaterais permite que várias partes dos programas, como por exemplo os argumentos de uma função, possam ser avaliados ao mesmo tempo sem que uma avaliação interfira na outra.

Como exemplo simples, porém ilustrativo, do uso de linguagens funcionais não estritas para a obtenção do paralelismo, pode-se pensar na avaliação de uma simples soma: `e1 + e2`. É necessário que se avalie `e1` e `e2`. Em uma linguagem imperativa, a avaliação de uma sub-expressões pode causar um efeito colateral afetando valores na outra sub-expressão. Como resultado, as duas expressões devem

ser simplificadas seqüencialmente em uma ordem pré-definida. Por outro lado, uma linguagem funcional pura garante que nenhuma avaliação pode interferir em outra, permitindo avaliar os argumentos em paralelo e, quando os valores forem obtidos, pode-se computar a soma.

1.2 Conectando Linguagens Funcionais e Java

Java é uma linguagem de programação, baseada em C++, desenvolvida pela Sun Microsystems, tendo como primeiro objetivo a programação de dispositivos eletrônicos inteligentes [DEI2001], mas que tornou-se popular pelas suas facilidades na programação de conteúdo *Web* dinâmico e pelo grande número de bibliotecas de programação disponíveis. Existem vários motivos para se desejar de alguma maneira a conexão entre linguagens funcionais e a linguagem Java. Dentre elas podemos destacar:

Filosofia "Escreva uma Vez, Rode em Qualquer Lugar"

A linguagem Java se torna atraente para vários programadores pois a sua implementação é baseada em uma Máquina Virtual. O conceito de Máquina Virtual torna fácil a implementação da mesma linguagem para várias arquiteturas diferentes, tornando o código gerado pelo compilador independente da máquina. Dessa maneira pode-se por exemplo compilar um programa Java utilizando um compilador rodando em uma Sun e rodar o programa em uma Máquina Intel com Windows. Existem atualmente implementações da Máquina Virtual Java (ou interpretador Java) para várias arquiteturas e com grandes empresas no mercado dando suporte. Com um compilador que gere Java Byte-Code (código que é aceito pela máquina Java) a partir de programas funcionais, pode-se permitir que os desenvolvedores de programas funcionais usufruam da portabilidade e da independência de arquitetura que a linguagem Java proporciona.

Bibliotecas

Um problema das linguagens funcionais, hoje em dia, é que como elas normalmente são desenvolvidas dentro de grupos de pesquisa, geralmente não conseguem atender a necessidade dos programadores de possuir um conjunto forte de bibliotecas para os mais diversos fins [WAD98]. Conectar-se linguagens funcionais com a linguagem Java pode ser uma solução para esse problema. Java possui um grande número de bibliotecas de classes pré-definidas na linguagem, para os mais diversos propósitos. Além disso, existem vários grupos de pesquisa e empresas privadas trabalhando para aumentar ainda mais esse número. Como a linguagem Java também trabalha com componentes de software, existe uma série de componentes que poderiam ser utilizados pelos desenvolvedores de programas funcionais.

Comunicação com outras Linguagens de Programação

Uma maneira de aumentar o uso das linguagens funcionais seria empacotando programas funcionais em componentes de software, permitindo que eles possam interagir com componentes escritos em outras linguagens de programação. Dessa maneira o desenvolvedor de software poderia usar a programação funcional para

resolver algumas partes do problema, que seriam mais difíceis de serem resolvidas usando uma linguagem imperativa. A linguagem Java, além de possuir sua própria arquitetura de componentes, conhecida como *JavaBeans*, possui compiladores que podem, partindo de um programa Java, gerar componentes de padrões mais estabelecidos no mercado, como CORBA e COM. Possuindo-se então um compilador que gera código Java partindo de programas funcionais, pode-se usar a tecnologia de compiladores Java existentes no mercado para se criar componentes de software implementados usando programação funcional.

Programação de Aparelhos Eletrônicos

Sabe-se que uma característica importante das linguagens funcionais é facilidade de sua utilização para a construção de linguagens de domínio específico [HUD2000]. Hoje em dia existem projetos de se utilizar a linguagem Java para programação de aparelhos eletrônicos simples como aparelhos de microondas e videocassetes com processadores Java embutidos [OCO97]. Utilizando-se o alto nível de abstração das linguagens funcionais e a sua capacidade de criação de linguagens de domínio específico poder-se-ia usá-las para a programação desses dispositivos. Existem estudos de utilização de linguagens funcionais para a programação de dispositivos que não são "computadores", como é o caso da linguagem Erlang [ARM92] desenvolvida pela Ericsson para a programação de PABXs.

Juntamente com este trabalho sobre execução distribuída de programas funcionais na máquina Java, fez-se um trabalho sobre comunicação entre programas funcionais e programas escritos em Java [DUB2000]. Utilizando o compilador desenvolvido neste trabalho, fez-se um estudo sobre a criação de componentes de software partindo de programas funcionais compilados para Java. O artigo descrevendo este estudo encontra-se no Anexo 3.

1.3 Objetivos deste Trabalho

Devido aos vários motivos apresentados anteriormente, parece ser uma tendência de pesquisa, hoje em dia, a integração de linguagens funcionais com outros paradigmas de programação, principalmente com a linguagem Java, uma linguagem inovadora, que introduz várias características que modificam o paradigma de computação distribuída [WAD97](uma série de pesquisas relacionadas com este assunto serão apresentadas no próximo capítulo desta dissertação).

Neste trabalho, pretende-se dar um novo passo à essa pesquisa, trabalhando-se com execução distribuída de programas funcionais paralelos na arquitetura Java.

Como foi dito anteriormente, as linguagens funcionais possuem várias vantagens que facilitam a sua execução paralela. Uma das características da linguagem Java que ajudaram a popularizar o seu uso foram as suas bibliotecas para a programação distribuída. De todos os trabalhos que tratam da integração de programação funcional com a linguagem Java que conseguimos analisar, nenhum utiliza essas características da programação funcional e da linguagem Java para a execução dos programas funcionais de forma distribuída em uma rede de computadores.

A linguagem Java é conhecida pelas facilidades que fornece para a programação de aplicações distribuídas. Ela possui bibliotecas para programação Cliente/Servidor,

threads, *sockets*, *streams*, chamada remota de procedimento etc. E todas estas características são muito bem documentadas em vários livros existentes no mercado (por exemplo [FAR98], [HUG99], [OAK97], [HAR97], [DEI2001]). Parece que um dos próximos passos possíveis na pesquisa de integração de linguagens funcionais com Java seria o aproveitamento dessas características de Java para adicionar facilidades aos programas funcionais.

A idéia deste trabalho é usar máquinas Java distribuídas em uma rede de computadores de maneira que elas trabalhem juntas para a avaliação de programas funcionais paralelos. As máquinas Java distribuídas formariam uma única máquina para a execução de programas funcionais como pode ser visto na figura 1.1.

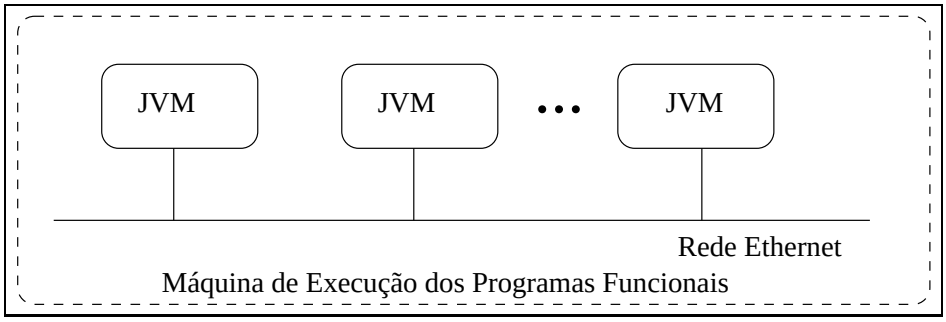

FIGURA 1.1 – Máquina Distribuída para Execução dos Programas Funcionais

O principal objetivo deste trabalho é:

Desenvolver uma Linguagem de Programação Funcional Paralela Independente de Arquitetura

Utilizando-se a arquitetura Java definiu-se uma linguagem funcional paralela que roda em redes de computadores comuns (Ethernet 10Mbps), disponíveis em qualquer universidade. A ideia é criar um sistema de tempo de execução que utilize máquinas Java espalhadas pela rede para formar uma máquina paralela que execute programas funcionais. Cada máquina na rede rodando a máquina virtual Java pode servir como um nodo para esta máquina virtual MIMD construída para a execução de programas funcionais. Pelo fato de existirem implementações da máquina Java para os mais diferentes tipos de processadores e sistemas operacionais, essas máquinas de execução de programas funcionais podem ser híbridas, utilizando qualquer máquina que esteja disponível na rede. Para adicionar um outro nodo a esta máquina distribuída é apenas necessário que o nodo esteja conectado à rede e que possua uma JVM instalada, dessa maneira não dependendo de nenhum software adicional como PVM [GEI94] ou MPI [MPI95], como na maioria das linguagens funcionais paralelas disponíveis hoje em dia (ex: [TRI96], [JUN98], [BRE98], [CAR2000] e [POI2000]).

Linguagens Funcionais geralmente são implementadas usando uma máquina abstrata para a execução dos programas. Essas máquinas são geralmente máquinas de redução de grafos [PJO87a]. Para se implementar esta linguagem funcional, escolheu-se implementar em cima da JVM uma máquina de redução de grafos baseada na G-Machine [AUG84]. Dessa maneira, os programas funcionais são compilados para essa máquina de redução de grafos e então podem rodar como qualquer programa Java na JVM.

Escolheu-se implementar uma máquina de redução de grafos ao invés de se fazer um mapeamento direto para a JVM pois:

- Mapear-se os programas funcionais para a máquina G-Machine é muito mais fácil do que fazer um mapeamento direto para a JVM;
- Implementando-se uma máquina do tipo G-Machine em cima da JVM fica muito mais fácil de reaproveitar a implementação com o objetivo de compilar outras linguagens funcionais para rodarem na JVM. O desenvolvedor da linguagem só precisa ter conhecimento sobre a G-Machine, o que é muito mais fácil do que se entender o funcionamento da JVM;

Escolheu-se a utilização da G-Machine pois a maioria das linguagens funcionais puras são implementadas usando a G-Machine ou uma de suas variações [WAK98a]. Além disso, existe uma série de livros que apresentam o funcionamento da G-Machine em detalhes, sendo muito mais fácil a compreensão de seu funcionamento do que o de outras máquinas de redução de grafos mais modernas.

Outro aspecto importante é que as máquinas de redução de grafos são um modelo atrativo para a implementação de linguagens paralelas [PJO87a]. Na implementação sequencial da redução de grafos, trabalha-se com uma tarefa de avaliação que recebe como argumento a raiz do programa principal e a avalia até a WHNF (*weak head normal form*)[PLA93]. Na execução distribuída dos programas funcionais, simplesmente trabalha-se com várias tarefas que fazem a avaliação simultânea de várias partes do grafo.

Na linguagem de programação paralela implementada neste trabalho escolheu-se fornecer ao usuário combinadores para que ele indique de forma explícita quais tarefas no programa funcional podem ser executadas de forma paralela. Apesar do programador indicar quais tarefas devem ser executadas em paralelo, todo o processo de criação das tarefas, distribuição para as unidades de processamento disponíveis e sincronização, é feita pelo sistema de tempo de execução. Esse esquema de o programador indicar apenas quais tarefas podem ser executadas em paralelo e deixar que o sistema de tempo de execução tome conta do resto, parece ser a tendência na implementação de linguagens funcionais paralelas, como pode ser visto nos trabalhos relacionados que são apresentados no próximo capítulo.

Escolheu-se para este trabalho implementar de modo completo uma linguagem funcional, ao invés de usar componentes já disponíveis, pois o processo de implementação da linguagem desde o início iria facilitar o aprendizado de todos os conceitos relacionados à implementação de linguagens funcionais necessários para a realização deste trabalho. Outro motivo é que possuir uma linguagem de programação funcional totalmente escrita na universidade facilita a criação de um grupo de estudos sobre o assunto. Além disso, fica-se livre para adicionar novas características na linguagem, pois se possui o domínio completo da tecnologia usada para implementá-la. Apesar de compiladores como o GHC [PJO93a] serem implementados de maneira modular para facilitar a sua expansão por outros pesquisadores, eles são geralmente muito grandes e usam um código bastante otimizado o que dificulta o seu entendimento para fins didáticos [MEI97].

É importante ressaltar que o objetivo deste trabalho não é implementar uma linguagem funcional paralela extremamente veloz, pois uma implementação em cima JVM perde bastante em desempenho pelo fato da linguagem Java ser interpretada.

O objetivo desta dissertação é contribuir com idéias de como poderia ser implementada a execução distribuída de programas funcionais usando as facilidades da linguagem Java.

Assim, a linguagem implementada neste trabalho é uma linguagem funcinal simples, baseada na linguagem Core apresentada por Peyton Jones em seus livros de implementação de linguagens funcionais [PJO87a] e [PJO92a]. Essa linguagem pode ser facilmente utilizada como código intermediário para implementações mais robustas.

1.4 Estrutura da Dissertação

Além deste capítulo inicial, esta dissertação esta dividida em mais cinco capítulos e quatro Anexos, que são apresentados a seguir:

Capítulo 2 - Estado da Arte

Nesta parte do trabalho apresentam-se algumas pesquisas relacionadas com esta dissertação e um pouco da história das linguagens funcionais. Primeiramente, descrevem-se o surgimento das linguagens funcionais e a sua motivação, desde a linguagem LISP até os dias de hoje, com a linguagem Haskell. Em seguida, descreve-se alguns dos trabalhos relacionados à programação funcional paralela. Por último, são apresentadas algumas pesquisas sobre a integração da programação funcional com a linguagem Java.

Capítulo 3 - A Implementação da G-Machine em Java

Neste capítulo, apresenta-se a G-Machine e a sua implementação utilizando a linguagem Java. Mostra-se também a linguagem funcional FUN que foi implementada para testar a G-Machine Java descrita neste trabalho. São apresentados também os esquemas de compilação utilizados para compilar os programas funcionais para a linguagem Java. No final, compara-se o desempenho da linguagem FUN com o de outra linguagem funcional compilada para Java.

Capítulo 4 - Execução Distribuída dos Programas Funcionais

Neste capítulo, estende-se a linguagem FUN com combinadores para se expressar o paralelismo nos programas funcionais. Apresenta-se então o funcionamento do sistema tempo de execução distribuído e a sua implementação na linguagem Java. Por último, mostra-se o *speedup* conseguido em alguns programas paralelos.

Capítulo 5 - Trabalhos Futuros

Este capítulo apresenta alguns trabalhos futuros a serem realizados e indicam-se trabalhos relacionados que podem dar continuidade ao que foi produzido neste trabalho.

Capítulo 6 - Conclusões e Considerações Finais

Apresentam-se conclusões obtidas com a realização deste trabalho.

Anexo 1 - Programas Funcionais Utilizados na Dissertação

Neste anexo estão os programas funcionais utilizados nos *benchmarks* apresentados nesta dissertação.

Anexo 2 - Código Java

Neste anexo estão algumas das classes Java implementadas durante a realização deste trabalho.

Anexo 3 - Artigo "*Functional Beans*"

Juntamente com este trabalho sobre execução distribuída de programas funcionais na máquina Java, fez-se um trabalho sobre comunicação entre programas funcionais e programas escritos em Java. Utilizando o compilador desenvolvido neste trabalho, fez-se um estudo sobre a criação de componentes de software partindo de programas funcionais compilados para Java. Este artigo foi apresentado no *Ninth International Workshop on Functional and Logic Programming* (WFLP'2000), realizado em Benicàssim, Espanha de 28 à 29 de Setembro de 2000.

Anexo 4 - Artigo "*Distributed Execution of Functional Programs on the JVM*"

Este artigo apresenta os principais conceitos desta dissertação. Ele foi aceito para apresentação no *Workshop: Functional Programming and λ-Calculus* que acontecerá no EUROCAST 2001 (*Eight International Conference on Computer Aided Systems Theory*). Este congresso será realizado de 19 à 23 de fevereiro de 2001 em Las Palmas de Gran Canaria, Ilhas Canárias - Espanha. O artigo será publicado na série *Lecture Notes in Computer Science* da Springer-Verlag.

2 Estado da Arte

Neste capítulo apresentam-se alguns trabalhos relacionados que foram estudados, além de mostrar-se um pouco da história do desenvolvimento das linguagens funcionais.

Na primeira parte, descreve-se o surgimento das linguagens funcionais e a sua motivação, desde a linguagem LISP até os dias de hoje, com a linguagem Haskell. Em seguida, descrevem-se alguns dos trabalhos relacionados à programação funcional paralela. Por último, são apresentadas algumas pesquisas sobre a integração da programação funcional com a linguagem Java.

2.1 Programação Funcional

Com o surgimento dos computadores, no final da década de 40, surgiram também as linguagens de programação. No princípio, os computadores eram programados diretamente em linguagem binária. Com o tempo, viu-se que este tipo de programação tornava o trabalho quase impossível, pois a programação era muito tediosa, propensa a erros e o código gerado era de leitura difícil, complicando a sua reutilização em outras aplicações. O próximo passo foi o surgimento das linguagens de montagem, ou *assembly* e das linguagens de alto nível. Nas linguagens de montagem as instruções são substituídas por mnemônicos e os endereços por símbolos definidos pelo programador. Uma instrução em linguagem de montagem corresponde exatamente a uma instrução da linguagem de máquina.

As linguagens de alto nível fornecem um maior nível de abstração que as linguagens de montagem. Nessas linguagens cada instrução equivale a várias instruções em linguagem de máquina. Uma das primeiras linguagens de alto nível a fazer sucesso foi o FORTRAN, devido ao compilador otimizado implementado por John Backus. A razão de seu sucesso foi o desenvolvimento de um compilador extremamente rápido, sendo esta a grande vantagem em cima das outras tentativas de implementação de linguagens de alto nível.

Nessas linguagens procedurais, cada instrução pode ser vista como uma macro de várias instruções da máquina e elas ainda possuem a abstração de procedimentos e definição de variáveis locais. Mesmo com o sucesso dessas linguagens de alto nível, os programadores ainda precisavam de uma maior abstração para poder expressar os seus algoritmos de uma maneira mais natural. A programação em linguagens procedurais do tipo FORTRAN, apesar de ser em mais alto nível do que a programação binária, ainda estava intimamente ligada à arquitetura da máquina.

Um dos primeiros a expressar a sua frustração com as linguagens de programação tradicionais foi o próprio John Backus, o mesmo do compilador FORTRAN, em 1978 na sua palestra de recepção do prêmio Turing [BAC78]: "*Can Programming be Liberated from the von Neumann Style?*".

A solução para as linguagens imperativas seriam as linguagens *declarativas*. A ênfase nesse tipo de linguagem é a de se prover uma notação de alto nível que seja conveniente ao programador ao invés de se trabalhar com uma máquina "von Neumman"abstrata. Nessas linguagens declarativas, o programador possui pouco ou nenhum controle sobre a ordem de avaliação dos programas e fica liberado de uma série de problemas relacionados a máquina, precisando apenas enfocar na solução do

problema. Existem dois tipos principais de linguagens de programação declarativa que são a programação em Lógica (cuja linguagem de trabalho é o Prolog) e a programação funcional (baseada no cálculo-λ de Church [CHU41]), que é o assunto deste trabalho.

A primeira linguagem a ser considerada do tipo funcional foi a linguagem LISP [WIN89], desenvolvida nos anos 60 por John MacCarty para o processamento de listas e que era utilizada na área de Inteligência Artificial. As primeiras linguagens funcionais incorporavam ainda uma série de características imperativas. A primeira linguagem a adotar uma semântica não estrita foi a linguagem SASL desenvolvida por David Turner. Turner trabalhou no projeto de várias outras linguagens funcionais, dentre elas, a linguagem Miranda [TUR85].

Finalmente, em 1987, na conferência FPCA'87, surgiu a linguagem Haskell [HUD2001] que foi definida para se tornar um padrão para as linguagens funcionais, de maneira a direcionar todas as pesquisas na área para uma única linguagem. A linguagem Haskell possui sintaxe derivada da linguagem Miranda e adota a maioria das características das linguagens funcionais puras, além de incorporar as classes de tipo [HAL94]. A versão atual da linguagem chama-se Haskell 98 [PJO99] e trabalha-se atualmente para o surgimento de um novo padrão, com novas características, que será chamado de Haskell 2.

Mais sobre o paradigma de programação funcional pode ser encontrado no livro escrito por Bird e Wadler [BIR88]. Sobre a linguagem Haskell existe uma série de livros como por exemplo: [BIR98], [DAV92], [THO99] e [HUD2000].

Sobre implementação de linguagens funcionais utilizando redução de grafos pode-se citar os livros do Peyton Jones [PJO87a] e [PJO92a], o livro do Davie [DAV92] e o livro sobre a implementação da linguagem Clean [PLA93].

2.2 Programação Funcional Paralela

Como mencionado anteriormente, as linguagens funcionais possuem várias características que facilitam a sua utilização na programação de arquiteturas multiprocessadas.

Com o avanço na tecnologia de compilação de linguagens funcionais e o aparecimento das máquinas abstratas para avaliação de programas funcionais, acreditou-se por um tempo que dever-se-ia criar novas arquiteturas de hardware para rodar programas funcionais [HAM94]. Dessas máquinas, várias eram multi-processadas. Como nas primeiras implementações de linguagens funcionais paralelas geralmente a granularidade das tarefas era muito fina, acreditava-se que a criação de hardware específico para a avaliação de programas funcionais paralelos iria resolver o problema do alto custo de comunicação e criação das tarefas [CAR2000].

ALICE (*Applicative Language Idealized Computing Engine*) [DAR81] foi a primeira e uma das mais famosas dessas máquinas paralelas, apesar dos resultados não terem sido encorajadores. Mesmo assim ela serviu como referência para outras máquinas com maior sucesso como por exemplo o ICL Flagship [WAT87] e GRIP (*Graph Reduction in Parallel*) [PJO87b].

Com o tempo viu-se que a construção de hardware proprietário era muito custoso e dificilmente iria substituir as máquinas atuais. Outro fator importante que inibiu a construção de hardware proprietário foram os avanços na área de compilação

eficiente de programas funcionais.

Hoje em dia as implementações de linguagens funcionais paralelas tentam utilizar uma arquitetura mais acessível. Por exemplo, o sistema GRIP foi implementado novamente em cima de PVM [HAM93]. Em seguida o mesmo grupo criou o GUM(*Graph reduction for a Unified Machine model*) [TRI96]. O GUM é um novo sistema de tempo de execução para o compilador GHC [PJO93a] . Este sistema de tempo de execução serve para implementar o GpH (*Glasgow Parallel Haskell*) [TRI93], que é uma extensão paralela da linguagem Haskell. O GUM é baseado em troca de mensagens e a sua portabilidade é facilitada pelo fato de ter sido implementado em cima de PVM. Ele esta disponível tanto para arquiteturas de memória compartilhada quanto distribuída.

Em GpH o programador utiliza um combinador par para indicar que uma tarefa deve ser avaliada em paralelo. Por exemplo:

```
x `par` e
```

significa que x pode ser avaliado por uma outra unidade de processamento enquanto o processador atual continua a avaliação de e. Geralmente a programação usando par utiliza expressões do tipo [TRI96]:

```
let x = f a in x `par` e
```

onde e menciona x. Apesar do programador indicar quais tarefas podem ser realizadas em paralelo, é o sistema de tempo de execução (o GUM) que decide quais tarefas devem ser criadas, para quais processadores elas serão alocadas e como elas devem ser sincronizadas. É por isso que se diz que a linguagem GpH utiliza uma mescla de paralelismo implícito e explícito. Esse tipo de combinador para expressar o paralelismo em linguagens funcionais aparece em vários trabalhos como por exemplo [KES96], [CHA94] e [JUN98]. A linguagem FUN, implementada neste trabalho, utiliza a mesma abordagem do GpH, o paralelismo é introduzido na linguagem através de combinadores (explícito), mas toda a gerência do comportamento das tarefas é feita pelo sistema de tempo de execução (implícito).

A GpH, pelo fato de ser uma extensão de Haskell, é uma linguagem que está em constante desenvolvimento, e é alvo de diversas pesquisas, por isso a inserção de paralelismo na linguagem FUN foi baseada nos conceitos dessa linguagem.

Atualmente os desenvolvedores da GpH estão trabalhando na linguagem GdH (*Glasgow Distributed Haskell*) [POI2000], que é a integração do *Concurrent Haskell* [PJO96a] com a GpH.

Outra extensão paralela da linguagem Haskell é o Haskell$_{\#}$ [CAR2000], desenvolvido na UFPE. Em Haskell$_{\#}$ cada módulo Haskell pode ser definido como um *processo* que pode se comunicar com outros processos através de *canais de comunicação*. Os canais de comunicação são guiados pelos tipos das funções no módulo Haskell, ou seja, uma função que recebe um valor do tipo Int em um processo pode se comunicar com uma função que gera um valor do tipo Int em outro processo. Um canal de comunicação é então formado pelo mapeamento de uma porta de entrada de um processo a uma porta de saída de outro. A criação, configuração e alocação dos processos nas máquinas é feita de forma explícita usando uma linguagem de configuração separada do código Haskell. Essa linguagem é a HCL (*Haskell*$_{\#}$

Configuration Language).

Outra linguagem que utiliza anotações para indicar o paralelismo e que continua sendo alvo de pesquisas é a linguagem *Clean* [PLA93]. Em Clean o programador pode expressar, através de anotações, tarefas como cópia do grafo, escalonamento e localização das tarefas.

Além da programação funcional paralela implícita e explícita existe a programação utilizando *skeletons* [COL89]. *Skeletons* são funções de alta ordem que capturam os algoritmos, ou padrões de comportamento, de vários tipos de paralelismo como por exemplo *divisão e conquista* ou *pipeline*. Essas funções de alta ordem devem ser então utilizadas pelo programador para expressar o paralelismo nos programas. Uma vantagem é que os *skeletons* podem ser usados em várias arquiteturas diferentes bastando modificar a implementação do *skeleton*, dessa maneira o programa final não precisa ser modificado. Um problema dos *skeletons* é que o conjunto dos *skeletons* necessários para expressar todos os tipos possíveis de paralelismo é muito grande, ou até infinito [HAM94].

Sobre programação funcional paralela e projetos relacionados ao assunto, recomenda-se a leitura de [HAM94] e [JUN98].

2.3 Programação Funcional e Java

Devido aos motivos listados na introdução desta dissertação, existem várias pesquisas que tentam de alguma forma fazer a conexão entre programação funcional e a linguagem Java. Dividimos aqui estes trabalhos em três tipos: compilação de linguagens funcionais para Java, acesso de programas Java a partir de programas funcionais (e vice-versa) e adição de características funcionais à linguagem Java.

Compilação de Linguagens Funcionais para Java

Existem vários trabalhos que descrevem a compilação de linguagens funcionais puras (não-estritas) para Java. Claus Reinke em seu artigo *Towards a Haskell/Java Connection* [REI99], aponta esta como sendo uma das mais promissoras maneiras de se conectar programação funcional com a linguagem Java. Primeiramente, destacam-se os artigos de David Wakeling ([WAK98a], [WAK98b] e [WAK98c]) sobre o seu compilador de Haskell para Java. No primeiro artigo [WAK98a], ele descreve uma implementação baseada na G-Machine, tentando fazer um mapeamento direto da G-Machine para a máquina Java. Analisando-se o artigo, sua implementação parece ser bem diferente da apresentada nesta dissertação. Ele usa um array Java para representar a pilha da G-Machine o que torna o programa ineficiente e dificulta a ação do coletor de lixo da linguagem Java. Infelizmente, o compilador não está disponível na internet, impossibilitando dessa maneira uma comparação detalhada. No artigo, Wakeling diz que o grande problema de sua implementação é a velocidade pois os programas são de 2 à 9 vezes mais lentos do que os mesmos programas rodando no interpretador Hugs [LOC2001]. No segundo artigo [WAK98b], ele tenta, sem sucesso, conseguir um maior desempenho usando a $< \nu$, G>-Machine [AUG89]. No artigo final da série [WAK98c], ele consegue um maior desempenho usando um compilador Java que gera diretamente código de máquina ao invés de usar a JVM.

Existe um outro compilador de Haskell para Java, baseado na *Spineless Tagless G-Machine* [PJO92b], desenvolvidos por Tullsen [TUL97]. Porém o seu artigo

não é muito claro e ele apresenta o tempo de execução de apenas dois programas. Outra implementação baseada na G-Machine é a da linguagem funcional Ginger, desenvolvida por Meehan e Joy na universidade de Warwick [JOY98]. No artigo, eles descrevem um processo de compilação diferente do apresentado nesta dissertação, evitando a geração de uma classe para cada função do programa funcional.

Todas estas implementações tem em comum o fato de não estarem disponíveis para download e de possuirem um tempo de execução inferior ao do interpretador Hugs.

Existem duas implementações em Java de linguagens funcionais que estão disponíveis na internet, são as linguagens Mondrian [MEI97] e Curry [HAN99]. A linguagem Mondrian é uma linguagem funcional que estende Haskell com algumas características de orientação a objetos. A sua implementação em Java utiliza um esquema de compilação parecido com o da linguagem FUN, porém sua implementação é baseada em outra máquina abstrata [MEI2001]. A linguagem curry é uma linguagem Lógica/Funcional que possui um compilador para Java que gera uma classe para cada função porém também usa uma outra máquina abstrata para a execução [HAN99].

Existem também implementações de linguagens funcionais estritas em Java, como por exemplo o compilador de ML, o MLJ [BEN98] e o compilador Kawa da linguagem Scheme [BOT98]. Na UFRGS, foi desenvolvido um trabalho de mestrado sobre a implementação de um construtor de funções Lisp em Java [GAV97].

Acesso de Programas Java a partir de Programas Funcionais (e vice-versa)

Claus Reiken em seu trabalho, citado anteriormente [REI99], além de apresentar uma série de idéias de como seria a conexão entre programação funcional e a linguagem Java, apresenta um trabalho no qual ele faz chamadas a classes Java através de programas funcionais rodando no interpretador Hugs. Para a implementação, ele usa a *Java Native Interface* (JNI).

No trabalho Lambada [MEI2000] Meijer e Finne apresentam uma série de módulos Haskell que servem para acessar programas Haskell partindo de classes Java e vice-versa. Este trabalho também usa a JNI para a implementação do sistema.

Juntamente com este trabalho sobre execução distribuída de programas funcionais, como dito anteriormente, foi desenvolvido um trabalho sobre criação de componentes de software contendo programas funcionais [DUB2000]. No artigo, explica-se como se criou um componente Java Beans partindo de um programa funcional e apresenta-se a implementação de um applet Java que utiliza este componente. Este artigo encontra-se no Anexo 3.

Adição de Características Funcionais na Linguagem Java

Odersky e Wadler [ODE97] implementaram uma versão de Java chamada de Pizza. Esta implementação estende Java com funções de alta ordem, polimorfismo de parâmetros e tipos algébricos de dados.

Outro trabalho interessante é o Juaskell [HAM99] no qual Hammond implementa na linguagem Java as estratégias de avaliação da linguagem GpH.

3 A Implementação da G-Machine em Java

Linguagens Funcionais geralmente são implementadas usando uma máquina abstrata para a execução dos programas. Essas máquinas são geralmente máquinas de redução de grafos. No caso, as expressões no programa funcional são vistas como grafos e cada função definida no programa serve como uma regra de redução para o grafo.

Nesta parte do trabalho, será apresentada a máquina de redução de grafos G-Machine e a sua implementação em Java.

3.1 Introdução à G-Machine

A G-Machine é a especificação de uma máquina abstrata de redução de grafos usada para a implementação de linguagens funcionais [PJO87a], baseada na compilação de expressões. A idéia principal é compilar o corpo de cada função para uma seqüência de instruções e essas instruções, quando executadas, irão construir o grafo representando o corpo da função e reduzir o programa para a *Weak Head Normal Form* (WHNF).

Então, para se rodar o programa funcional, são necessários dois estágios. Primeiramente, o programa é compilado para uma seqüência de instruções aceitas pela G-Machine, que são chamadas de *G-Code*. No segundo estágio, estas instruções são executadas, o que é chamado de *run-time* (tempo de execução). Os estágios podem ser vistos na figura 3.1.

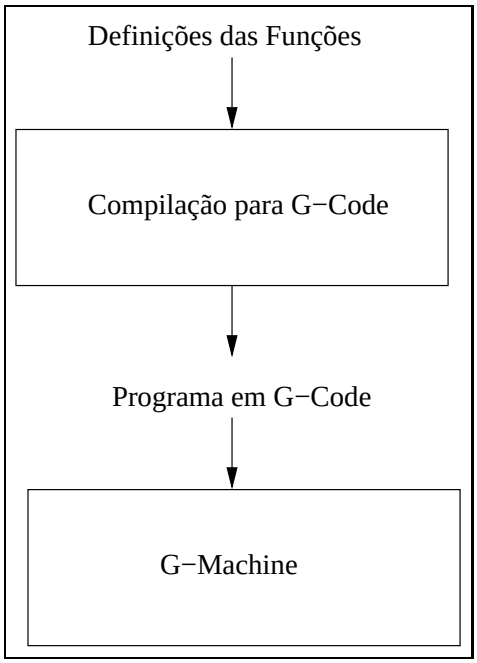

FIGURA 3.1 – Execução de Programas na G-Machine

A utilização de uma máquina abstrata, como a G-Machine, facilita a implementação do mesmo sistema para uma série de plataformas de hardware e sistemas operacionais diferentes. Outra vantagem é que é mais fácil compilar o programa

funcional para um código abstrato como o G-Code.

Segue um exemplo do funcionamento da G-Machine. Considere uma função

```
f x = id x
```

Essa função é compilada para a seguinte seqüência de G-Code:

```
PUSH 0
PUSHGLOBAL id
MKAP
UPDATE 1
POP 1
UNWIND
```

Na figura 3.2 pode-se ver como essa seqüência de instruções é executada na G-Machine para reduzir uma chamada de função. A instrução PUSH 0 é usada para colocar no topo da pilha um ponteiro para a expressão que está na posição 0 na pilha (o endereçamento é relativo ao topo da pilha começando com 0 e a pilha na figura 3.2 cresce para baixo). Em seguida a instrução PUSHGLOBAL id coloca na pilha um ponteiro para a função pré-definida id. A instrução MKAP retira os dois elementos que estão no topo da pilha deixando no topo um ponteiro para um nodo de aplicação de função entre eles. A instrução UPDATE 1 faz com que a raiz do redex original contenha um nodo que aponte para a aplicação de função que foi construída. A instrução POP 1 retira o elemento que esta no topo da pilha pois ele já foi utilizado. Por último, a instrução UNWIND faz com que a G-Machine continue a avaliação do programa.

3.2 Descrição da G-Machine e suas Instruções

A G-Machine é uma máquina de estado finita composta por 4 componentes [PJO87a]:

P A Pilha. Usada para armazenar os dados que estão sendo trabalhados.
G O Grafo. Contém todos os nodos do grafo que está sendo avaliado.
C O Código. A lista de instruções que devem ser executadas.
D O Dump. É utilizado para armazenar o estado de execução quando se deseja trabalhar com outras partes do grafo. Consiste de uma pilha de pares (P,C) onde P é uma pilha e C é uma seqüência de código.

Dessa maneira, o *estado* da G-Machine pode ser definido como uma tupla de 4 elementos $<$ P, G, C, D $>$. Nesta parte do trabalho, as instruções da G-Machine serão descritas como *transições de estado* (denotada pelo simbolo $\Rightarrow$).

É interessante explicar detalhadamente a notação que será utilizada na descrição de estados da G-Machine:

Uma pilha cujo elemento no topo é n será escrita como n:P, onde P é o restante da pilha. Uma pilha vazia é representada por []. A mesma descrição vale para a seqüência de código C, que tendo INST como a primeira instrução a ser executada

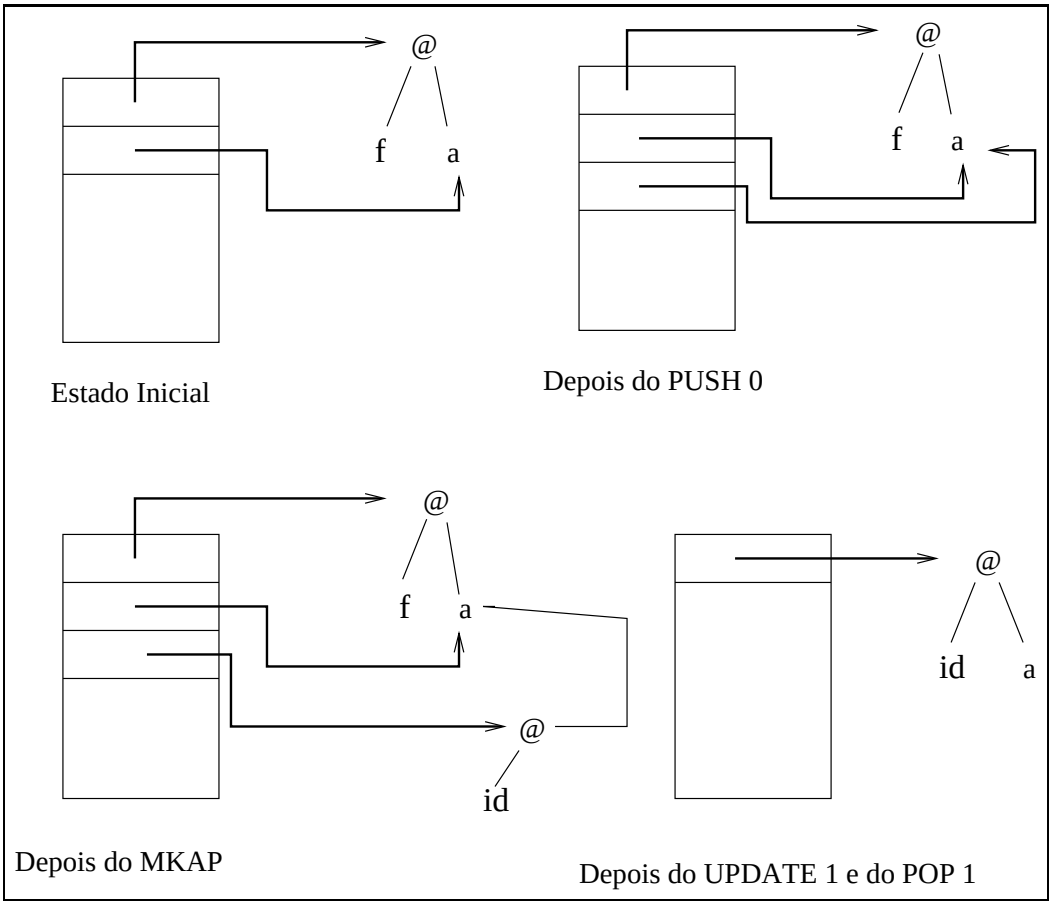

FIGURA 3.2 – Execução das Instruções na G-Machine

seria descrita por INST:C.

A pilha dump cujo primeiro par é (P,C) é descrita como (P,C):D.

Os tipos de nodos que podem aparecer no Grafo são:

- INT i Um inteiro (ou qualquer outro valor de um tipo básico da linguagem).
- CONS tag list Um nodo CONS que representa um construtor de tipo. Possui uma tag que o identifica e uma lista list de elementos que o compõe.
- AP n_1 n_2 Um nodo que representa aplicação de função.
- FUN K C Uma função com K argumentos e seqüência de instrucoes C.
- IND n Significa que o valor deste nodo encontra-se no nodo n do grafo. Os nodos IND são gerados pela instrução UPDATE.

Por exemplo, a notação G[n=AP n_1 n_2] significa que o nodo n no grafo é um nodo de aplicação de n_1 em n_2 (n é apenas um nome para este nodo).

3.2.1 As Instruções da G-Machine

A avaliação de um programa funcional na G-Machine irá começar através da seguinte seqüência de instruções:

```
PUSHGLOBAL main
EVAL
```

Estas instruções simplesmente dizem que deve-se colocar na pilha a função principal do programa funcional que é um CAF (*Constant Application Form*) geralmente chamado de main e em seguida começar a sua avaliação.

A instrução PUSHGLOBAL pode ser descrita por uma transição simples de estado, da seguinte maneira:

PUSHGLOBAL
< P, G, PUSHGLOBAL (FUN K I):C, D >
⇒ < n:P, G[n=FUN K I], C, D >

Isto significa que quando PUSHGLOBAL for a primeira instrução, a G-Machine irá fazer uma transição para um novo estado em que

- Um novo nodo n está no topo da pilha;
- O grafo foi atualizado com a informação de que o nodo n é FUN K C;
- O código a ser executado agora é a seqüência após PUSHGLOBAL (FUN K C).

Outras instruções mais complicadas podem ser explicadas, por uma análise de casos, usando *casamento de padrões*. Segue a definição da instrução EVAL:

EVAL
< n:P, G[n=AP n_1 n_2], EVAL:C, D >
⇒ < n:[], G, UNWIND:[], (P,C):D >

< n:P, G[n=FUN 0 C'], EVAL:C, D >
⇒ < n:[], G, C':[], (P,C):D >

< n:P, G[n=INT i], EVAL:C, D >
⇒ < n:P, G, C, D > (O mesmo para CONS e FUN que não são CAFs.)

A regra de transição para a instrução EVAL é escolhida dependendo do nodo que está no topo da pilha:

- A primeira regra diz que se o nodo no topo da pilha for um nodo de aplicação de função, a pilha atual e o código devem ser salvos na dump e uma nova pilha é formada contendo o nodo que estava no topo da pilha antiga. Deve-se executar então a instrução UNWIND.
- Se o nodo no topo da pilha for uma função sem argumentos, um CAF, a máquina deve salvar o seu estado na dump, deixando na pilha apenas o CAF e deve então executar o código associado ao supercombinador.
- Se o nodo no topo da pilha for um valor do tipo INT, CONS ou for uma função não CAF a máquina não deve fazer nada pois o valor já esta na WHNF.

A instrução UNWIND é outra instrução de controle que possui uma definição com vários casos:

UNWIND
< n:P, G[n=INT i], UNWIND:[], (P,C):D >
⇒ < n:P, G, C, D> (O mesmo para nodos CONS)

< n_1:P, G[n_1=IND n], UNWIND:C, D >
⇒ < n:P, G, UNWIND:C, D>

< v:P, G[v=AP v' n], UNWIND:[], D >
⇒ < v':v:P, G, UNWIND:[], D >

< v_0:v_1:...:v_k:P, G[v_0=FUN K C, v_i=AP v_{i-1} n_i $(1 \leq i \leq k)$], UNWIND:[], D >
⇒ < n_1:n_2:...:n_k:v_k:P, G, C, D >

< v_0:v_1:...:v_a:[], G[v_0=FUNC K C'], UNWIND:[], (P,C):D > (onde $a <$ K)
⇒ < v_a:P, G, C, D>

Para a instrução UNWIND existem cinco casos:

- Se o elemento no topo da pilha é do tipo CONS ou INT isso significa que ele já está avaliado e encontra-se na WHNF. Deve-se então restaurar o estado anterior da pilha usando-se os valores que estão no topo da dump e deixando no topo da pilha o valor que está na WHNF.
- Se o nodo no topo da pilha é do tipo IND deve-se colocar no topo da pilha o elemento apontado pelo IND e repetir-se a instrução UNWIND.
- Se o nodo no topo da pilha é uma aplicação de função, deve-se repetir o UNWIND para o próximo nodo.
- Quando houver uma função no topo da pilha e houverem argumentos suficiente na pilha, esta deve ser reorganizada e o código para a função deve ser executado.
- Se o ítem no topo da pilha é uma função, mas a pilha não contém todos os argumentos da função, significa que ela esta na WHNF. Dessa maneira deve-se retirar o primeiro elemento da dump e voltar ao estado anterior, deixando o resultado da avaliação no topo da pilha restaurada.

Como última instrução de controle temos a instrução CASEJUMP que é usada para implementar uma função condicional (do tipo if):

CASEJUMP
< n:P, G[n=True], (CASEJUMP [True:codT;False:codF]):C, D >
⇒< n:P, G, codT:C, D >

< n:P, G[n=False], (CASEJUMP [True:codT;False:codF]):C, D >
⇒< n:P, G, codF:C, D >

A instrução CASEJUMP possui em seu corpo duas seqüências de instruções. Ela espera encontrar no topo da pilha um valor booleano e de acordo com este valor uma das seqüências de instruções é escolhida.

As outras instruções da G-Machine são mais simples de se compreender e estão definidas sem explicações adicionais a seguir:

PUSH
$<$ n_0:n_1:...:n_k:P, G, PUSH k:C, D $>$
$\Rightarrow$ $<$ n_k:n_0:n_1:...:n_k:P, G, C, D $>$

PUSHINT
$<$ P, G, PUSHINT i:C, D $>$
$\Rightarrow$ $<$ n:P, G[n=INT i], C, D $>$

POP
$<$ n_1:n_2:...:n_k:P, G, POP k:C, D $>$
$\Rightarrow$ $<$ P, G, C, D $>$

SLIDE
$<$ n_0:n_1:...:n_k:P, G, SLIDE k:C, D $>$
$\Rightarrow$ $<$ n_0:P, G, C, D $>$

UPDATE
$<$ n_0:n_1:...:n_k:P, G, UPDATE k:C, D $>$
$\Rightarrow$ $<$ n_1:...:n_k:P, G[n_k=G n_0], C, D $>$

NEG
$<$ n:P, G[n=INT i], NEG:C, D $>$
$\Rightarrow$ $<$ n':P, G[n'=INT (-i)], C, D $>$

ADD
$<$ n_1:n_2:P, G[n_1=INT i_1, n_2=INT i_2], ADD:C, D$>$
$\Rightarrow$ $<$ n:P, G[n=INT (i_1 + i_2)], C, D $>$
(Da mesma maneira para os outros operadores aritméticos, lógicos e de comparação).

MKAP
$<$ n_1:n_2:P, G, MKAP:C, D$>$
$\Rightarrow$ $<$ n:P, G[n=AP n_1 n_2], C, D $>$

CONS
$<$ n_1:...:n_k:P, G, CONS t k:C, D$>$
$\Rightarrow$ $<$ n:P, G[n=CONS t [n_1, ..., n_k]], C, D $>$

SPLIT
$<$ n:P, G[n=CONS t [n_1, ..., n_k]], SPLIT k:C, D$>$
$\Rightarrow$ $<$ n_1:...:n_k:P, G, C, D $>$

3.3 Implementando a G-Machine em Java

A idéia deste trabalho é implementar uma G-Machine com uma interface simples, de maneira a permitir que outros pesquisadores que conheçam o funcionamento da G-Machine possam, sem muito esforço, reutilizar o código para implementar outras linguagens funcionais.

Escolheu-se mapear cada função do programa funcional para uma classe Java. Por exemplo, a função:

```
f x = id x
```

deve ser compilada para a seguinte classe Java:

```
class f extends Nsuperc{

public f (){
        narg = 1;
        name = new String ("f"); }

public void code () {
        GM.push(0);
        GM.pushglobal(new id());
        GM.mkap();
        GM.update(1);
        GM.pop(1);
        }
}
```

A classe f estende a classe-pai Nsuperc. A classe Nsuperc é uma classe abstrata que compreende todas as funções definidas pelos usuários. As classes que implementam as funções de um programa funcional sempre irão definir duas variáveis em seu construtor: a variável narg que indica o número de argumentos da função e a String name que possui o nome da função e é usada para *debug*.

Todas as classes que implementam a classe Nsuperc possuem um método público chamado de code que possui em seu corpo as instruções da G-Machine para instanciar a função que está sendo definida.

A G-Machine foi implementada como uma classe Java (a classe GM), que possui em seu corpo uma série de métodos estáticos que são as instruções da G-Machine, como pode ser visto na figura 3.3.

A classe GM possui um método reduce (Nsuperc main) que é responsável pela avaliação do programa funcional (uma ilustração simples do método reduce pode ser visto na figura 3.4). Ele começa colocando no topo da pilha a expressão inicial do programa, que é a função main e em seguida chama o seu método code. Então GM.head devolve uma referência para o valor que está no topo da pilha, que é o resultado da execução de code. O método GM.unwind() busca, na espinha do grafo,

```
class GM {

  private static Gstack stack = new Gstack();

  static void push (int n)
  { (...) }

  static void pop (int n)
  { (...) }

  static void mkap ()
  { (...) }

  (...)

}
```

FIGURA 3.3 – A Classe GM

a próxima expressão a ser avaliada e chama o seu método code. Este processo é repetido até que se encontre no topo da pilha um objeto pertencente a classe Nbasic ou seja, um valor na WHNF.

Para a implementação da G-Machine, escolheu-se não usar a estrutura de pilha já implementada na biblioteca Java e, sim, construir-se uma nova estrutura de dados. O motivo para isso é que existe uma série de operações que a G-Machine realiza em sua pilha que não estão implementadas na pilha da Java e a mesma também não dispõe das funcionalidades necessárias à implementação destas operações.

A pilha foi implementada como uma lista encadeada de elementos do tipo No. Cada objeto No possui um elemento do tipo Node, ou seja, um nodo do grafo da G-Machine e uma referência para o próximo elemento da pilha, ou seja, outro elemento No:

```
class No{
      public Node actual;
      public No next;

      No (Node p, No n)
      {
      this.actual = p;
      this.next = n;
      }
}
```

Usando este modelo fica fácil implementar a pilha e suas instruções mais complicadas.

```
 static String reduce (Nsuperc main) {

    Node value;

    GM.put(main);
    main.code();
    value = (Node) GM.head();

    while (!(value instanceof Nbasic))
    {
     GM.unwind();
     value = (Node) GM.head();
    }

    (...)
}
```

FIGURA 3.4 – O Loop de Execução da G-Machine

```
class Gstack {
public No first;
public int size;

public Gstack() {
first = null;
size = 0;}

(...)
```

A pilha implementada para a G-Machine é uma lista encadeada que possui elementos da classe Node. A classe Node é a classe base para todos os tipos de nodos que formam o grafo da G-Machine. Uma hierarquia de tipos de nodos pode ser vista na figura 3.5.

A classe Node é simplesmente uma classe abstrata que que não possui nenhum método definido. Ela apenas define que todos os nodos devem possuir uma string name. Esta string, como dito anteriormente, é usada para *debug* dos programas:

```
abstract public class Node {
public String name;
}
```

Diretamente abaixo da classe Node temos as classes Nbasic, Nsuperc e Nap. A classe Nsuperc é a classe que representa as funções definidas no programa funcional. A classe Nbasic são os valores básicos da linguagem como números, booleanos e tipos algébricos.

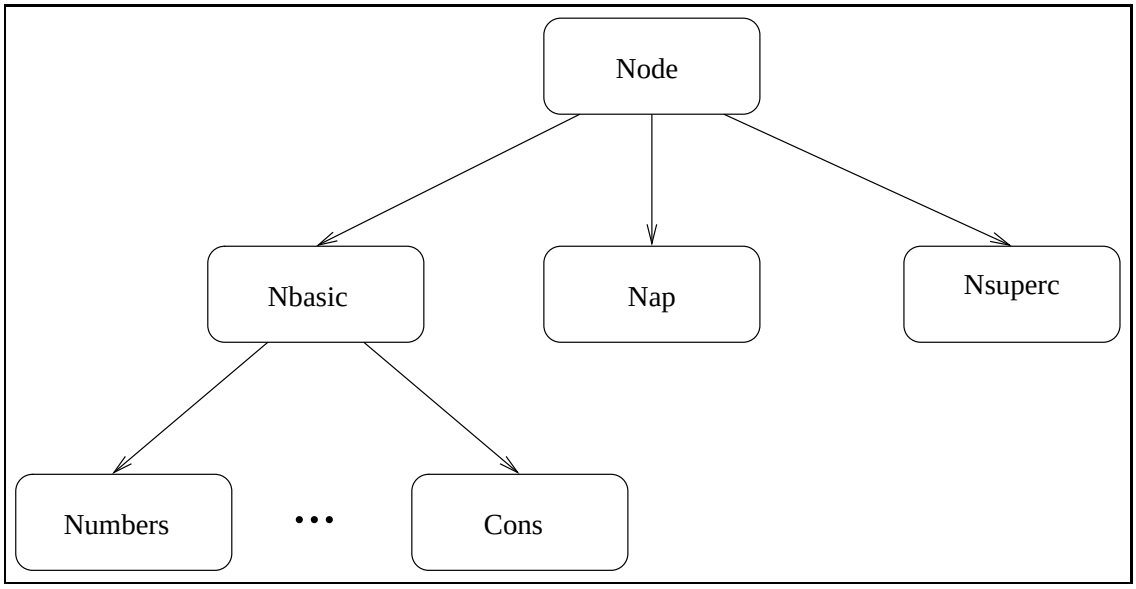

FIGURA 3.5 – Classes que implementam os nodos

A classe Nap é a classe que representa aplicação de função. Ela é definida da seguinte maneira:

```
public class Nap extends Node{
public Node func, arg, ind;
public Nap (Node f, Node a)
{
this.ind = null;
this.func = f;
this.arg = a;
name = new String ("Nap");
}
}
```

Após a avaliação de uma aplicação de função, deve-se atualizar o nodo de aplicação com o seu valor. Isso é feito utilizando a variável ind presente na classe Nap. O campo ind possui inicialmente o valor null e a instrução GM.update() atualiza o campo com o valor resultante da aplicação de função. Durante a redução do grafo, se o campo ind de uma aplicação de função não for null, o caminho apontado pela variável ind deve ser seguido.

Java possui coleta de lixo automática [JON96]. O coletor de lixo faz a coleta dos objetos que não estão mais sendo usados. Por isso, quando é feita a atualização de um nodo de aplicação de função, os campos func e arg devem ser modificados para null. Dessa maneira, se eles não estiverem sendo apontados por alguma outra parte do grafo, eles poderão ser coletados.

Da mesma maneira, quando a função sendo compilada for uma CAF, esta deve possuir um campo ind para ser atualizado com a instrução GM.update() da G-Machine, após a função ter sido avaliada. A diferença é que este campo ind deve ser acessado por todas as instâncias da classe que implementa a função. Nesse caso,

o campo ind é uma variável de classe. As variáveis de classe são compartilhadas por todas as instâncias daquela classe. Então, uma vez que a CAF foi avaliada, outras instâncias da mesma função serão beneficiadas.

3.4 Compilando uma Linguagem Funcional para a G-Machine

Para testar a G-Machine Java descrita neste trabalho, implementou-se um compilador que, partindo de uma pequena linguagem funcional, gera classes Java contendo G-Code. O objetivo desta parte do trabalho é descrever esta linguagem funcional e a sua implementação. Primeiramente, a linguagem é descrita através de exemplos simples. Em seguida, se apresentam os esquemas de compilação usados para transformar as funções do programa funcional em classes Java.

3.4.1 FUN: Uma Linguagem Funcional Simples

A linguagem FUN é uma pequena linguagem funcional parecida com Haskell e baseada na linguagem Core descrita por Peyton Jones em [PJO92a]. A linguagem FUN, apesar de ser simples apresenta todas as características importantes de uma linguagem funcional pura e pode ser usada como código intermediário para uma implementação mais robusta. A linguagem será apresentada através de alguns exemplos.

Um programa funcional consiste de um conjunto de definições de funções, incluindo uma expressão inicial que é chamada de main. Para se executar o programa funcional deve-se avaliar o resultado da função main. Ex:

```
twice f x = f (f x)

id x = x;

main = twice twice id 3
```

Na linguagem FUN pode-se fazer definições locais nas funções usando-se a construção let:

```
add x y = x + y;

succ x = let suc = add 1 in
     succ x;

main = succ (succ (succ 2))
```

Pode-se usar as expressões let apenas para nomear valores ou expressões. Não se pode fazer casamento de padrões.

Em todas as linguagens funcionais modernas, pode-se definir tipos de dados algébricos. Por exemplo em Haskell usa-se o comando data seguido do nome do tipo que esta sendo construído e seus componentes:

```
data Tempo = Frio | Calor
```

```
data Lista a = Nil | Cons a (List a)
```

Ao invés de permitir construtores definidos pelos usuários, como por exemplo Calor ou Nil, a linguagem FUN possui apens uma família de construtores:

Cons $\{tag, arity\}$

Onde tag é um número inteiro que identifica o construtor dentro do tipo que está sendo definido e arity especifica quantos argumentos o construtor precisa. Temos então:

```
frio = Cons {1,0}
calor = Cons {2,0}

nil = Cons{1,0}
cons x xs = Cons{2,2} x xs

folha a = Cons {1,1} a
ramo a b = Cons {2,2} a b

mkPair a b = Cons {1,2} a b
```

A tag é necessária para diferenciar construtores diferentes quando eles constróem um mesmo objeto. Por exemplo, uma lista é construída utilizando-se dois construtores, o cons e nil, por isso eles devem ter tags diferentes. Em um programa bem tipado, objetos de tipos diferentes não precisam ser distinguidos durante a execução do programa, por isso as tags só precisam ser únicas dentro de um tipo de dados.

Na linguagem FUN o casamento de padrões em cima dos tipos algébricos é feito utilizando-se expressões case:

```
isCold temp = case temp of
      <1> -> true;
      <2> -> false;

length xs = case xs of
      <1> -> 0;
      <2> a as -> 1+ length as;
```

É importante notar que nas expressões case cada alternativa possui apenas uma tag e um certo número de variáveis (o número de variáveis deve ser o mesmo do valor de arity no construtor).

A linguagem FUN possui também a função if para controle de fluxo condicional:

```
fac n = if (n==0) 1 (n * fac (n-1));
```

fib n = if (n<=1) 1 (fib (n-1) + fib (n-2) +1);

Como pode-se ver através dos exemplos, a linguagem FUN é uma linguagem simples porém possui algumas facilidades que permitem a escrita de programas funcionais ricos e expressivos sem muito trabalho do programador.

O objetivo da linguagem é apenas facilitar o uso das G-Machines implementadas neste trabalho. A idéia é utilizar esta linguagem no futuro como código intermediário para uma implementação mais completa, por isso a linguagem FUN não possui checagem de tipos.

3.4.2 Compilando a linguagem FUN para classes Java

Nesta parte do trabalho apresentam-se os esquemas de compilação utilizados para transformar os programas escritos na linguagem FUN para classes Java que utilizam a G-Machine.

Os esquemas de compilação aqui apresentados geram uma classe Java para cada função definida no programa funcional. Cada função no programa funcional é compilada usando o esquema de compilação $\mathcal{SC}$.

$\mathcal{SC}[\![f\ x_1 ... x_n = e]\!]$ =
"public class" ++ f ++ " extends Nsuperc{

public" ++ f ++ "(){
narg =" ++ n ++ ";
name = new String (" ++ f ++");}

public void code (){" ++
$\mathcal{R}[\![e]\!][x_1 \longmapsto 0, ..., x_n \longmapsto n-1]\ n$
++ " } }"

$\mathcal{R}[\![e]\!]\ p\ d\ =\ \mathcal{E}[\![e]\!]$ ++ "GM.update (d); GM.pop (d);"

FIGURA 3.6 – Esquemas $\mathcal{SC}$ e $\mathcal{R}$ de Compilação da linguagem FUN

O esquema de compilação $\mathcal{SC}$ simplesmente gera a estrutura da classe Java. Esse esquema gera a declaração da classe, que terá o mesmo nome da função sendo compilada, o construtor da classe e a estrutura do método code().

Para gerar as instruções da G-Machine que devem aparecer no método code(), o esquema $\mathcal{SC}$ utiliza o esquema $\mathcal{R}$. O esquema $\mathcal{R}$ recebe como argumento, além da expressão que está sendo compilada, uma função p que indica o *offset* na pilha para acessar os argumentos da função. O esquema $\mathcal{R}$ gera as instruções GM.update e GM.pop que finalizam a instanciação da função na G-Machine. As outras instruções são geradas pelos esquemas $\mathcal{E}$ e $\mathcal{C}$.

Mesmo trabalhando-se com linguagens funcionais não estritas, existem certos contextos em que a função que está sendo avaliada irá precisar de seus argumentos de forma estrita, como por exemplo em uma soma de dois valores. Muitas vezes, funções que são garantidamente estritas podem ser avaliadas em muito menos passos

do que se utilizando uma avaliação *lazy*. Deve-se então, de alguma forma, analisar o código que está sendo compilado e identificar possíveis ocorrências de um contexto estrito, para que se possa gerar um código otimizado. Porém, deve-se tomar cuidado nessa análise para não gerar um código demasiadamente estrito. Dessa maneira, o código irá avaliar partes das expressões que não seriam necessárias, podendo não chegar a um resultado final.

$\mathcal{E}[\![i]\!]\ p\ =$ "GM.pushint(i);"	
$\mathcal{E}[\![let\ x_1 = e_1; ...; x_n = e_n\ \mathbf{in}\ e]\!]\ p =$	$\mathcal{C}[\![e_1]\!]\ p^{+0} + +... + +$ $\mathcal{C}[\![e_n]\!]\ p^{+(n-1)} + +$ $\mathcal{E}[\![e]\!]\ p' + +$ "GM.slide(n);" onde $p' = p^{+n}[x_1 \longmapsto n-1, ..., x_n \longmapsto 0]$
$\mathcal{E}[\![if\ e_0\ e_1\ e_2]\!]\ p\ =$	$\mathcal{E}[\![e_0]\!]\ p\ + +$ "if (((Bool)GM.head()).value()) { GM.pop(1);" $+ +\ \mathcal{E}[\![e_2]\!]\ p\ + +$ "} else { GM.pop(1);" $+ +\ \mathcal{E}[\![e_2]\!]\ p\ + +$ "}"
$\mathcal{E}[\![case\ e\ \mathbf{of}\ alts]\!]\ p =$	$\mathcal{E}[\![e]\!]\ p + +$ "switch (((Nconstr)GM.head()).tag){" $+ +$ $\mathcal{D}[\![alts]\!]\ p + +$"}"
$\mathcal{E}[\![Cons\{t, a\}\ e_1\ ...\ e_a]\!]\ p =$	$\mathcal{C}[\![e_a]\!]\ p + +\ ...\ \mathcal{C}[\![e_1]\!]\ p + +$"GM.cons(t,a);"
$\mathcal{E}[\![negate\ e]\!]\ p =$	$\mathcal{E}[\![e]\!]\ p\ + +$ "GM.neg();"
$\mathcal{E}[\![e_0 + e_1]\!]\ p\ =$ (e de maneira similar para as outras operações aritméticas e de comparação)	$\mathcal{E}[\![e_1]\!]\ p\ + + \mathcal{E}[\![e_0]\!]\ p^{+1}\ + +$ "GM.add();"
$\mathcal{E}[\![e]\!]\ p\ =\ \mathcal{C}[\![e]\!]\ + +$ "GM.eval();" (último caso, default)	

FIGURA 3.7 – Esquema $\mathcal{E}$ de Compilação da linguagem FUN

Os esquemas de compilação da linguagem FUN identificam dois tipos de contexto:

- **Estrito**: O valor será necessário na WHNF
- **Lazy**: O valor talvez seja necessário na WHNF

Para cada um dos contextos, se utiliza um esquema de compilação diferente, que irá gerar um conjunto de instruções da G-Machine diferentes. Para o contexto estrito, utiliza-se o esquema $\mathcal{E}$ e para o contexto lazy usa-se o esquema $\mathcal{C}$.

A idéia é tentar encontrar o maior número de contextos estritos, pois assim é possível gerar um código mais rápido. A análise de contexto presente nesse trabalho é baseada em [PJO92a] e pode ser descrita recursivamente:

- A expressão no corpo da definição de uma função está sempre em um contexto estrito.
- Se $e_0 \odot e_1$ ocorrem em um contexto estrito e $\odot$ é um operador aritmético ou de comparação então e_0 e e_1 também estão em um contexto estrito.
- Se negate e ocorre em um contexto estrito, então e também está em um contexto estrito
- Se a expressão if e_0 e_1 e_2 ocorre em um contexto extrito as expressões e_0, e_1 e e_2 também ocorrem.
- Se let $\triangle$ in e ocorrer em um contexto estrito então a expressão e também está em um contexto estrito.

Como o corpo de uma função sempre é avaliado em um contexto estrito, o esquema $\mathcal{R}$ chama diretamente o esquema $\mathcal{E}$. O contexto estrito se propaga usando as regras definidas recursivamente e em caso contrário o esquema $\mathcal{C}$ é usado.

$\mathcal{C}[\![f]\!]\ p\ =$ "GM.pushglobal(new" $++\ f\ ++$ "());"
$\mathcal{C}[\![x]\!]\ p\ =$ "GM.push (" $++\ p\,x\ ++$ ");" onde x é uma variável local
$\mathcal{C}[\![i]\!]\ p\ =$ "GM.pushint(i);"
$\mathcal{C}[\![Cons\{t,a\}\ e_1\ ...\ e_a]\!]\ p = \mathcal{C}[\![e_a]\!]\ p + +\ ...\ \mathcal{C}[\![e_1]\!]\ p + +$ "GM.cons(t,a);"
$\mathcal{C}[\![e_0\ e_1]\!]\ p\ =\ \mathcal{C}[\![e_1]\!]\ p\ + +\mathcal{C}[\![e_0]\!]\ p^{+1}\ + +$ "GM.mkap();"

$\mathcal{C}[\![let\ x_1 = e_1; ...; x_n = e_n\ \mathbf{in}\ e]\!]\ p =$
$\mathcal{C}[\![e_1]\!]\ p^{+0} + +... + +$
$\mathcal{C}[\![e_n]\!]\ p^{+(n-1)} + +$
$\mathcal{C}[\![e]\!]\ p' + +$ "GM.slide(n);"
onde $p' = p^{+n}[x_1 \longmapsto n-1, ..., x_n \longmapsto 0]$

FIGURA 3.8 – Esquema $\mathcal{C}$ de Compilação da linguagem FUN

É interessante notar, nos esquemas de compilação, que algumas funções, como por exemplo if, são mapeadas diretamente para a sua correspondente em Java (ver figuras 3.7 e 3.8). Outro exemplo é a expressão case que é mapeada diretamente para a função switch do Java.

A transformação de expressões case para switch do Java utiliza outros dois esquemas de compilação que são o $\mathcal{D}$ e o $\mathcal{A}$ que estão na figura 3.9.

$\mathcal{D}[\![alt_1\ ...\ alt_n]\!]\ p = [\mathcal{A}[\![alt_1]\!]\ p,\ ...,\ \mathcal{A}[\![alt_n]\!]\ p]$

$\mathcal{A}[\![< t > \ x_1\ ...\ x_n\ ->\ body]\!]\ p =$"case "$+ + t + +$" :
GM.split(n);" $+ + \mathcal{E}[\![body]\!]p' + +$ "GM.slide(n); break;"
onde $p' = p^{+n}[x_1 \longmapsto n-1, ..., x_n \longmapsto 0]$

FIGURA 3.9 – Esquemas $\mathcal{D}$ e $\mathcal{A}$ de Compilação para a linguagem FUN

O esquema $\mathcal{D}$ é utilizado para compilar as alternativas em uma expressão case. Este por sua vez utiliza o esquema $\mathcal{A}$ para compilar cada uma das alternativas do case.

Importante notar que, devido à natureza recursiva da definição dos esquemas de compilação, fica fácil a implementação dos mesmos utilizando uma linguagem funcional. Como parte deste trabalho, implementou-se um compilador para a linguagem FUN basedo nos esquemas de compilação aqui apresentados, utilizando a linguagem Haskell [PJO99].

Na figura 3.10 pode-se observar o ambiente implementado para a programação usando a linguagem FUN.

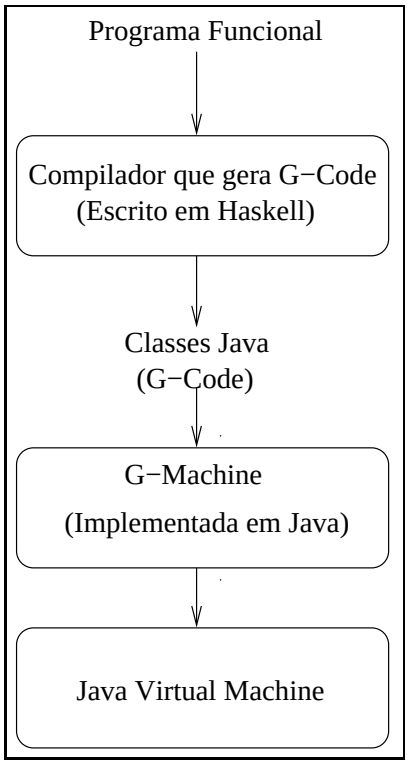

FIGURA 3.10 – Execução de Programas na G-Machine Java

Os programas escritos na linguagem FUN devem ser compilados usando o compilador implementado em Haskell. Este compilador gera um conjunto de classes Java que fazem chamadas à G-Machine, implementada em Java. O programa final é um programa na linguagem Java que deve rodar em uma JVM.

3.4.3 Benchmarks

Apresenta-se aqui os tempos de execução de alguns programas na linguagem FUN e Mondrian [MEI97].

O programa fib calcula os elementos na lista de números Fibonacci, sieve usa o algoritmo *Eratosthenes* para calcular uma lista de números primos e sumeuler soma a lista de valores gerados por euler. coins é uma aplicação que, dada uma certa coleção de moedas e um valor, calcula o número de maneiras possíveis de se pagar esse valor com essas moedas. As implementações dos programas encontram-se no Anexo 1 deste texto.

TABELA 3.1 – Tempo de Execução dos Programas

Programas	FUN	Mondrian
nfib 25	52s	1m47s
sumeuler 200	1m7s	1m11s
coins	27s	23s
sieve 300	36s	31s

Os resultados aqui apresentados servem apenas para dar uma idéia do funcionamento da linguagem FUN em comparação com outra linguagem funcional compilada para Java. Para resultados mais interessantes seria necessário programas que comparassem diferentes aspectos das máquinas abstratas em que são implementadas as duas linguagens além de ser necessário um conhecimento mais profundo da máquina abstrata em que se baseia a implementação da linguagem Mondrian [MEI2001].

4 Execução Distribuída dos Programas Funcionais

Nesta parte do trabalho, apresenta-se a implementação em Java de uma G-Machine distribuída para a execução de programas funcionais paralelos. A idéia é utilizar as facilidades de programação distribuída da linguagem Java para modificar a G-Machine da linguagem FUN, de maneira que se possa implementar programas funcionais que exploram o paralelismo. Para se rodar os programas de forma distribuída/paralela, usam-se JVMs rodando em computadores diferentes, conectadas por *sockets*, de maneira a formar uma única máquina virtual que executa programas funcionais.

Na parte anterior do trabalho, mostrou-se que redução de grafos é uma técnica que pode ser utilizada para a implementação de linguagens funcionais sequenciais. Nesta parte do trabalho, mostra-se que a redução de grafos também pode ser usada para a implementação de linguagens funcionais paralelas.

Para expressar o paralelismo nos programas funcionais escritos na linguagem FUN, optou-se por adicionar alguns combinadores à linguagem, para que o programador possa indicar quais tarefas podem ser avaliadas de forma concorrente. Apesar do programador indicar quais avaliações podem ser feitas em paralelo, todo o processo de criação, sincronização e finalização das tarefas é feito pelo sistema de tempo de execução.

Primeiramente, neste capítulo, apresentam-se os combinadores utilizados para se expressar o paralelismo nos programas funcionais. Em seguida, apresentam-se como estes combinadores foram implementados e a nova instrução da G-Machine usada para isso. Então explica-se como funciona o ambiente de execução distribuída e a sua implementação em Java. No final do capítulo, mostram-se alguns resultados obtidos.

4.1 Paralelismo na linguagem FUN

Para expressar o paralelismo na linguagem FUN, escolheu-se implementar dois combinadores parecidos, porém de comportamento ligeiramente diferente: par e parap. O combinador par funciona como o operador par da linguagem *Glasgow Parallel Haskell* (GpH) [TRI93]. Uma expressão par p e possui o mesmo valor de e:

```
par p e = e
```

Seu comportamento dinâmico é indicar que p pode ser avaliado em uma outra máquina (ou em outro PE - *Processing Element*), enquanto a máquina atual continua a avaliação do resto do programa. O combinador parap [PJO92a] é um sinônimo para aplicação de função:

```
parap p e = p e
```

Seu comportamento serve para indicar que e pode ser avaliado em outra máquina. Estes operadores são implementados usando uma nova instrução da G-

Machine, a instrução GM.par [PJO92a], que será explicada de forma detalhada na próxima seção deste texto. Esta instrução coloca o nodo que está no topo da pilha em uma *spark pool* (que é uma *pool* de nodos que podem ser avaliados em paralelo com o programa principal) e deixa no topo da pilha um nodo do tipo Npool que possui o endereço na *spark pool* em que o nodo antigo está agora localizado.

A idéia de se implementar dois combinadores para se expressar paralelismo na linguagem FUN serve simplesmente para dar mais opções para o programador. Este deve decidir em cada caso qual combinador irá expressar a sua idéia da melhor maneira. De qualquer forma, muitas vezes os dois combinadores podem ser usados, de modo intercambiável, para implementar o mesmo programa.

Por exemplo, pode-se considerar o clássico algoritmo recursivo para cálculo de números na seqüência fibonacci:

```
fib n = if (n<=1) 1 ( fib (n-1) + fib (n-2) +1);
```

Esta implementação que utiliza um algorítmo de *divisão e conquista* pode ser facilmente paralelizada da seguinte maneira utilizando parap:

```
pfib n = let pfib1 = pfib (n-1); pfib2 = pfib (n-2) +1
      in
      if (n<=1) 1 (parap (seq pfib1 ((+) pfib1)) pfib2);
```

ou utilizando par:

```
pfib n = let fib1 = pfib (n-1); fib2 = pfib (n-2)
      in
      if (n<=1) 1 (seq (par fib2 fib1) (fib1 + fib2 +1));
```

Nestes programas foi utilizado o combinador seq para controlar em que ordem as avaliações no programa funcional são feitas. Se e1 não é $\perp$ (onde $\perp$ significa um valor inválido), a expressão seq e1 e2 tem valor e2; senão possui valor $\perp$. O comportamento da função é avaliar e1 até sua WHNF antes de retornar e2.

A utilização do combinador seq é importante pois é muito difícil de se prever em que ordem as expressões em um programa funcional serão avaliadas [TRI93]. Por isso a utilização do seq sempre garante que as expressões serão avaliadas na ordem desejada.

Um aspecto interessante dos combinadores seq e par é a sua utilização na implementação de funções de alta ordem que expressam outros comportamentos de paralelismo. Por exemplo, pode-se expressar um paralelismo *orientado a dados*, no qual todos os elementos de uma estrutura de dados são avaliados em paralelo, com a implementação paralela da função map. A função map é uma função que recebe dois argumentos, uma função de tipo func :: a -> b, uma lista de tipo list::[a] e devolve como resposta uma lista de tipo list::[b], ou seja, a função map aplica sua função argumento a todos os elementos da lista. Essa função é definida da seguinte maneira na linguagem FUN:

```
map f list = case list of
      <1> -> nil;
```

```
    <2> a b -> cons (f a) (map f b);
```

A versão paralela da função map pode ser implementada da seguinte maneira:

```
parmap f l = case l of
    <1> -> nil;
    <2> a b -> let fa = f a;
    fb = parmap f b
    in seq (par fa fb) (cons fa fb);
```

Esta definição paralela da função map faz com que a função argumento seja aplicada em todos os elementos da lista em paralelo. A combinação da função par com seq na definição, faz com que primeiramente a função avalie a *espinha* da lista enviando os nodos, que representam a aplicação da função nos elementos da lista, para a *spark pool* e somente depois disso comece a avaliar seus valores.

Outro detalhe que deve-se observar quando se trabalha com estruturas de dados, é que a G-Machine sempre avalia os programas até a WHNF. Por isso, se uma expressão que representa uma estrutura de dados, por exemplo uma lista de valores, é enviada para a *spark pool* e posteriormente para uma máquina cliente, esta lista será avaliada apenas até sua WHNF, ou seja apenas até o primeiro cons da lista ter sido construído, gerando assim muito pouco paralelismo. Quando se deseja que a lista seja avaliada por completo na máquina cliente, deve-se utilizar uma função que force esta avaliação. Por exemplo, supondo que na expressão:

```
par lista computacao
```

lista seja uma expressão que represente uma lista, em que cada elemento exige um certo poder computacional para ser gerado e computacao seja uma expressão que utiliza lista. Neste exemplo lista sera enviada para uma máquina cliente para ser avaliada, porém pouco trabalho será realizado pois a lista é avaliada apenas até a WHNF e então é devolvida para a máquina principal. Para que lista seja avaliada por completo na máquina cliente antes de ser enviada de volta para a máquina principal, é necessário se utilizar uma função que force a avaliação da lista:

```
forceList xs = forcel xs xs;

forcel xs ys = case xs of
    <1> -> ys;
    <2> p ps -> seq p (forcel ps ys);
```

A função forcel simplesmente força a avaliação de todos os elementos da lista e devolve como resultado a própria lista avaliada. A expressão anterior deveria ter sido definida da seguinte maneira:

```
par (forceList lista) computacao
```

Da mesma maneira, se por exemplo a função argumento de parmap construir uma lista, esta deve ser composta com a função forceList para assegurar que a lista

```
public class par extends Nsuperc{

      public par(){
          nargs = 2;
          name= new String ("par");
      }

      public void code() {

        GM.push(0);
        GM.par();
        GM.update(0);
        GM.push(1);
        GM.update(2);
        GM.pop(2);
      }
}
```

FIGURA 4.1 – Implementação do combinador par

seja construída em paralelo.

Programação funcional paralela usando par e seq é explicada em detalhes em [TRI93].

4.2 Implementando os Combinadores Paralelos

Os combinadores par e parap são implementados usando uma nova instrução da G-Machine, a GM.par(). Esta instrução serve para enviar para a *spark pool* o nodo que está no topo da pilha. Esta instrução é implementada de maneira simples em Java:

```
public static void par () {
      Node node = GM.head();
      GM.pop(1);
      int add = pool.putnode(node);
      GM.push (new Npool(add));
      }
```

Primeiramente, o nodo é retirado da pilha e enviado para a pool. Isto é feito utilizando o método putnode(node) da *spark pool*. Este método retorna o endereço na pool em que o nodo agora está localizado. Em seguida é colocado na pilha um nodo do tipo Npool que contém o endereço do nodo antigo na pool. A classe pool é implementada como uma lista encadeada de nodos com um esquema de gerenciamento de endereços simples.

Usando a instrução GM.par() e as outras instruções da G-Machine, pode-se

```
public class parap extends Nsuperc{

      public parap(){
           nargs = 2;
           name = new String ("parap");
      }

      public void code(){

        GM.push(1);
        GM.par();
        GM.push(1);
        GM.mkap();
        GM.update (2);
        GM.pop(2);
      }
}
```

FIGURA 4.2 – Implementação do combinador parap

implementar o combinador par, como pode ser visto na figura 4.1.

O combinador par é implementado da seguinte maneira: Primeiramente, a instrução GM.push(0) coloca no topo da pilha o primeiro argumento de par, que é o argumento que se deseja enviar para a *spark pool*. Por isso, a próxima instrução a ser executada é a GM.par() que envia o nodo para a *pool* deixando em seu lugar um nodo do tipo Npool que contém o endereço da nova localização do nodo na *pool*. Em seguida a instrução GM.update é executada para que o argumento da função seja realmente substituído pelo nodo Npool que está no topo da pilha. Agora deve-se fazer com que a raiz do *redex* aponte para o segundo argumento da função e isto é feito com as instruções GM.push(1) e GM.update(2).

O combinador parap também é simples de ser implementado usando a instrução GM.par. Sua implementação não é muito diferente da do par como pode ser visto na figura 4.2.

A primeira instrução a ser executada no combinador parap é GM.push(1) que coloca no topo da pilha o segundo argumento do combinador. Em seguida chama-se a instrução GM.par enviando então o elemento no topo da pilha para a *spark pool*. Em seguida é construído um nodo de aplicação de função (instrução GM.mkap) entre o primeiro argumento do parap e o nodo Npool. Esse nodo de aplicação de função então é devolvido como resposta da função (GM.update(2)).

4.3 Execução Distribuída dos Programas Funcionais

O objetivo deste trabalho é utilizar JVMs rodando em máquinas diferentes em uma rede para formar uma única máquina distribuída que execute programas

funcionais paralelos.

O primeiro passo para rodar um programa funcional paralelo nesta máquina (depois de sua compilação) é a criação do *Ambiente Funcional Distribuído* (ver figura 4.3). O ambiente sempre terá um PE principal, que irá construir a máquina distribuída, iniciar a avaliação do programa funcional e coordenar a distribuição das tarefas entre as máquinas. Este PE roda a *G-Machine Principal*. Então, para criar o ambiente de execução distribuída, primeiramente inicia-se a execução da G-Machine principal. Esta irá esperar pela conexão das JVMs rodando a *G-Machine cliente*. Uma vez que todos os clientes se conectaram à máquina principal, o ambiente de execução estará completo e o programa funcional pode começar a ser avaliado.

Os combinadores implementados para expressar o paralelismo na linguagem FUN servem para indicar avaliações que não precisam de uma ordem exata para ocorrer. Os combinadores apenas apontam valores no programa que poderiam ser avaliados de forma concorrente. Toda a vez que a G-Machine principal encontra uma chamada para par ou parap ela envia uma parte do grafo que está sendo avaliado para a *spark pool* e deixa no grafo um nodo (do tipo Npool) que possui o endereço na *spark pool* onde o sub-grafo está localizado agora. Sempre que, durante a execução do programa, a G-Machine encontrar um nodo do tipo Npool e precisar de seu valor para continuar a avaliação do programa, ela sabe que o valor do mesmo está na *spark pool* no endereço contido no nodo Npool.

Quando um valor é enviado para a *spark pool*, o sistema de tempo de execução verifica se existe alguma máquina cliente que não esteja executando alguma tarefa. Em caso afirmativo este envia o nodo para a máquina inativa usando um escalonamento do tipo FIFO. Esta abordagem de enviar nodos para as máquinas apenas quando elas estiverem inativas é chamada de *criação preguiçosa de tarefas* (*lazy task creation*)[HAM94].

Quando a G-Machine cliente recebe um nodo ela o avalia até a WHNF e então o envia de volta para a *spark pool* que o coloca em seu endereço antigo.

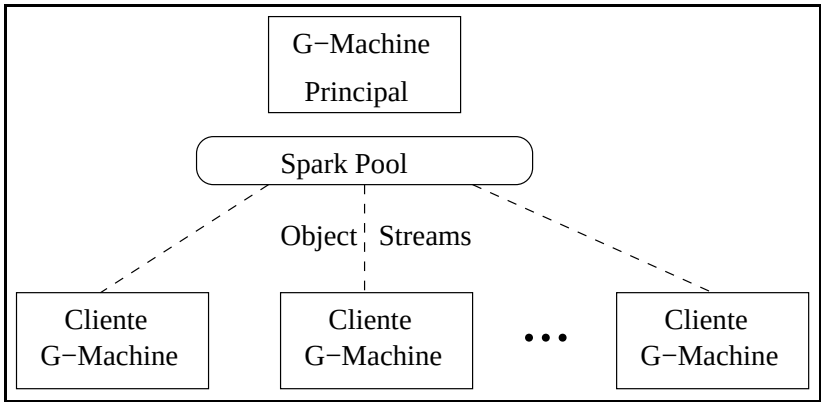

FIGURA 4.3 – Ambiente Funcional Distribuído

Quando a G-Machine principal encontra um nodo do tipo Npool durante a execução do programa, três coisas podem acontecer:

- Se o nodo na *spark pool* está em sua WHNF (isso significa que alguma máquina cliente já o avaliou), ela coloca o nodo em sua posição inicial e continua a avaliação do programa;

- Se o nodo não está na WHNF e não está sendo avaliado por nenhum cliente, ela retira o nodo da *spark pool* e continua a avaliação do programa;

- Se o nodo está sendo avaliado por algum cliente, a G-Machine principal salva seu estado, pega qualquer nodo disponível na *spark pool* e começa a avaliá-lo. Quando o valor estiver na WHNF ele é retornado para o seu endereço na *spark pool*. A G-Machine então verifica se o primeiro nodo está na WHNF (ou seja, foi devolvido pela máquina cliente). Nesse caso ela recupera seu estado original e continua a avaliação do programa principal. Caso contrário, o processo é repetido.

4.3.1 O que fazer com chamadas par e parap nos clientes?

A solução implementada atualmente no sistema é ignorar as chamadas par e parap quando elas ocorrerem nas máquinas clientes. Dessa maneira as máquinas clientes possuem uma implementação diferente dos combinadores par e parap na qual parap significa apenas aplicação de função e par é uma função que devolve seu segundo argumento. Optou-se por essa solução para evitar um excesso de comunicação na rede no acesso à *spark pool* que fica localizada em apenas uma máquina. Se todas as máquinas conectadas à G-Machine principal começassem a enviar nodos para a *spark pool* ao mesmo tempo, além do excesso de tráfego na rede, iria ocorrer uma sobrecarga de trabalho na *thread* que coordena as conexões.

Ignorando as chamadas par e parap nas máquinas clientes, o sistema trata com eficiência os seguintes casos:

- **O cliente recebe uma tarefa de granularidade grossa que gera tarefas de granularidade fina** Neste caso o *overhead* de comunicação na rede com o envio das tarefas de granularidade fina para a *spark pool* seria muito grande comparado com o ganho com a avaliação paralela. Da mesma maneira essas tarefas seriam logo necessárias na máquina cliente gerando mais comunicação.

- **O cliente recebe uma tarefa grande que não gera outras tarefas** Neste caso a tarefa que o cliente recebe não possui chamadas as funções par e parap. Desse modo o cliente recebe o sub-grafo, faz a avaliação, envia o resultado para a máquina principal e requisita um outro grafo para avaliar.

A principal deficiência da abordagem implementada aparece quando a tarefa que está na máquina cliente gera outras tarefas que consomem um grande tempo computacional. Se todas as máquinas estão ocupadas trabalhando não há problemas. Pode acontecer porém que a *spark pool* na máquina principal fique sem tarefas e consequentemente várias máquinas clientes fiquem sem tarefas enquanto uma das clientes poderia estar gerando trabalho para elas. Neste caso seria interessante que a máquina cliente enviasse tarefas para a *spark pool*. O problema é como saber se essa tarefa que está sendo criada na máquina cliente é uma tarefa que vale a pena ser enviada para a *spark pool* ou se o tempo de envio, recepção e sincronização dessa tarefa não é maior que o tempo de sua execução.

Por esses motivos escolheu-se a abordagem de ignorar as chamadas par e parap nas máquinas clientes, mesmo que isso possa algumas vezes gerar um balanceamento de carga ruim.

4.4 Implementação da Execução Distribuída

Java é uma das linguagens de programação cujo uso mais vem crescendo nos últimos anos [DEI2001]. Um dos grandes motivos para a sua popularização é a grande quantidade de classes pré-definidas da linguagem que faciltam o desenvolvimento de aplicativos para diversas áreas [FLA97].

Para a implementação do ambiente de execução distribuída da linguagem FUN foram usadas uma série de características implementadas na linguagem Java como *sockets*, *streams* de objetos, serialização de objetos e *threads*.

O envio de nodos da máquina principal para as máquinas clientes foi implementado utilizando a abstração de *sockets*. A classe Socket da linguagem Java realiza comunicação em uma rede usando uma conexão baseada em *streams*.

Para se conectar duas máquinas usando a abstração de *sockets*, deve-se apenas instanciar dois objetos do tipo Socket um na máquina principal e outro na máquina cliente e usar uma *stream* para conectá-los. Então tudo o que for escrito no *socket* na máquina principal pode ser lido na máquina do cliente.

Em Java existe um tipo especial de *stream* que se chama *stream de objetos*. Se os *sockets* do cliente e do servidor estão conectados por uma *stream* de objetos, pode-se com facilidade enviar objetos Java de uma máquina para outra.

Para objetos poderem ser enviados através de *streams* de objetos, estes devem implementar a interface Java Serializable. Um aspecto interessante do package Java.io de Java é a habilidade de *serializar* objetos, o que significa converter um objeto em uma *stream* de bytes de maneira que este objeto possa ser recuperado ao seu estado original mais tarde [FLA97].

Segue agora a explicação de como essas características da linguagem Java foram usadas para implementar a G-Machine distribuída.

4.4.1 Montando o Ambiente de Execução Distribuída

Quando a G-Machine principal é iniciada, ela fica primeiramente esperando por conexões das máquinas clientes. Após todas as máquinas estarem conectadas o usuário pode iniciar a avaliação do programa funcional.

No momento em que a G-Machine principal é iniciada, o programa se divide em duas *threads*, a thread *principal* que possui a G-Machine e que faz a avaliação dos programas funcionais e a *thread server* que lida com as conexões das máquinas clientes e os acessos à *spark pool*.

A *thread* principal é o método GM.reduce() explicado no capítulo anterior. Para a G-Machine distribuída, este método é modificado apenas para criar e iniciar a thread server:

```
(...)
server s = new server();
Thread th = new Thread (s);
Thread principal = Thread.currentThread();
th.setPriority((principal.getPriority()-1));
th.start();
(...)
```

```
public class server implements Runnable {

     public final static int port = 1515;
     ObjectOutputStream p;
     ServerSocket ss;

     public void run(){

           try{
            ss = new ServerSocket(port);
            while (true)
            {
               connect fs = new connect (ss.accept());
               fs.start();
            }
        }catch (IOException e)
        { (...) }
     }
}
```

FIGURA 4.4 – Classe server

É interessante notar que as *threads* em sistemas Unix e em sistemas Windows possuem um comportamento diferente [OAK97]. Em sistemas do tipo Unix o escalonamento das *threads* é feito de forma determinística pois depois que uma *thread* começa a ser executada, ela só libera o processador quando acaba a sua execução, mesmo que hajam outras *threads* querendo executar. Em sistemas do tipo Windows é feito um *timesharing* [TAN97] do processador entre as várias *threads* criadas no sistema. Como o objetivo deste trabalho era criar um sistema que fosse multiplataforma, resolveu-se forçar o escalonamento das *threads* de maneira que elas se comportem da mesma forma em qualquer sistema operacional. Este resultado foi obtido manipulando-se as *prioridades* das *threads*.

A linguagem Java implementa um escalonamento de tarefas baseado em prioridades. Isso significa que cada *thread* em um programa Java recebe uma certa prioridade que é um valor positivo definido dentro de um intervalo pré-definido (entre Thread.MIN_PRIORITY e Thread.MAX_PRIORITY). A Máquina Virtual Java nunca muda a prioridade da *thread*, mesmo que esta mude de estado ou que já esteja rodando a algum tempo. A prioridade de uma *thread* só pode ser mudada pelo programador. O valor da prioridade é importante pois a JVM garante que a *thread* rodando é sempre a com maior prioridade entre as outras. Se as *threads* possuem a mesma prioridade o escalonamento depende do sistema operacional, como descrito anteriormente.

Primeiramente, a *thread* server é iniciada com uma prioridade menor que a *thread* principal (th.setPriority (principal.getPriority()-1)), mas como a *thread* principal está bloqueada (esperando que o usuário inicie o programa), a *thread server* pode manipular a conexão das máquinas clientes.

A *thread* server cria um *socket* na porta 1515 que fica esperando pela conexão das máquinas clientes. Cada vez que uma máquina se conecta à essa porta, é criada uma outra *thread* (do tipo connect) que irá manipular o envio e o recebimento de dados entre uma máquina e outra, assim como os acessos à *spark pool* localizada na máquina principal (ver figura 4.4)

Quando o usuário iniciar o programa, significa que nenhuma outra máquina cliente irá se conectar à máquina principal. A *thread* principal assume então o processador, pois ela fica desbloqueada e possui a maior prioridade. As outras *threads* ficam bloqueadas, o que não é problema pois ainda não existe nenhum valor na *spark pool* para ser processado.

A partir do momento que os valores são inseridos na *spark pool* as *threads* connect ganham uma prioridade mais alta que a *thread* principal e permanecem com a prioridade alta até o final da execução do programa funcional. Esse fato não interfere no desempenho da *thread* principal, as *threads* connect passam a maior parte do tempo bloqueadas, pois elas simplesmente retiram um valor da *spark pool*, enviam para a máquina cliente e ficam bloqueadas até a resposta do cliente (como será visto em seguida).

4.4.2 A *spark pool*

A *spark pool* é implementada como uma lista encadeada de objetos com um controle de endereçamento para acesso desses objetos. A *spark pool* provê uma interface de acesso através dos seguintes métodos:

- public synchronized int putnode (Node a) Este método é utilizado pela instrução GM.par para enviar os nodos para a *spark pool*. Ele retorna o endereço no qual o nodo está agora localizado.

- public synchronized Node getnode (int add) É usado pela G-Machine para retirar o nodo da *spark pool* quando esta, durante a avaliação do programa, encontra uma referência a um nodo que está na *spark pool* (um nodo do tipo Npool). Se o nodo está sendo avaliado por uma máquina cliente este método retorna um objeto null.

- public synchronized packet fgetnode () É usado pela *thread* connect para retirar da *spark pool* os nodos que ainda não foram avaliados. Este método retira os nodos da *pool* na mesma ordem em que eles foram colocados (FIFO). Ele retorna um valor do tipo packet (ver figura 4.5). Este valor contém um nodo para ser avaliado pela máquina cliente e o seu endereço na *spark pool*. Caso não existam valores na *spark pool* o endereço retornado no packet é -1, o que significa um erro pois este endereço não é válido.

- public synchronized void updatenode (int add, Node node) É usado para se substituir o nodo na *spark pool* por seu valor na WHNF. Esta função é utilizada tanto pela G-Machine principal, quanto pelas *threads* connect quando recebem o nodo avaliado de uma máquina cliente. Este método recebe como argumento um objeto do tipo Node e o endereço na *spark pool* em que este valor deve ser colocado.

Como sempre haverá mais de uma *thread* acessando a *spark pool* ao mesmo tempo é preciso garantir a consistência dos dados. Esta consistência é adquirida

```
public class packet implements Serializable{

  public Node value;
  public int add;

  public packet (Node value, int add)
  {
     this.value = value;
     this.add = add;
  }
}
```

FIGURA 4.5 – A classepacket

usando a palavra reservada synchronized na declaração dos métodos. A palavra reservada synchronized provê um *mutex lock* garantindo que duas *threads* não possam chamar os métodos da *pool* ao mesmo tempo [OAK97].

A sincronização funciona da seguinte maneira: quando um método é declarado synchronized ele precisa possuir um *token* chamado de *lock* para poder executar. Uma vez o método adquirindo o *token*, ele pode executar, e o *token* só será liberado quando o método terminar a execução. Só existe um *lock* para cada objeto, por isso se duas *threads* tentarem chamar métodos synchronized ao mesmo tempo em um mesmo objeto, somente uma irá executar o método imediadamente; a outra terá que esperar até que a primeira *thread* libere o *lock*.

4.4.3 Enviando e recebendo Nodos do Grafo

Para se enviar Nodos da G-Machine na máquina principal para uma máquina cliente, estes nodos necessitam ser serializáveis. A classe Node deve então implementar a interface Serializable. Isto é feito através de uma pequena modificação na declaração da classe:

```
import java.io.*;

abstract public class Node implements Serializable {
     public String name;
}
```

Para se implementar a serialização, deve-se sempre importar a package Java.io.

Todos os outros tipos de nodos da G-Machine Java herdam a capacidade de serialização da classe Node, já que todos derivam dessa classe (como pode ser visto na figura 3.5).

Quando um nodo da G-Machine for enviado de uma máquina para outra, este deve ser enviado juntamente com o seu endereço na *spark pool*. Isto é feito *empacotando* o nodo juntamente com o seu endereço dentro de um objeto da classe packet (figura 4.5).

É importante observar que a classe packet também deve ser serializável e que todos os tipos básicos da linguagem Java são serializáveis.

Quando a *spark pool* possuir nodos, as *threads* connect irão enviá-los para as respectivas máquinas cliente utilizando o Socket criado na classe server. O nodo é retirado da *spark pool* e enviado dentro de um packet, utilizando uma stream de objetos.

Quando a máquina principal recebe uma conexão de uma G-Machine cliente que deseja receber um nodo para avaliação, esta deve retirá-lo da *pool* e enviá-lo através da *stream* de conexão dentro de um packet :

```
class connect extends Thread{

      Socket connection;

public connect (Socket s){
      connection = s;
      }

public void run(){
      try{
      ObjectOutputStream str;
      str = new ObjectOutputStream(connection.getOutputStream());

      (...)
      packet pack = GM.pool.fgetnode();
      if (pack.add != -1) {
      str.writeObject(pack);
      str.flush();
      ObjectInputStream in;
      in= new ObjectInputStream(connection.getInputStream());
      packet resp = (packet) in.readObject();
      (...)
```

O método fgetnode() da *spark pool*, como dito anteriormente, devolve o primeiro nodo disponível para ser avaliado já dentro de um objeto packet. Então, o objeto é enviado para a máquina cliente através do método writeObject() da stream (no exemplo str). Para se certificar do envio do objeto deve-se usar o método flush().

Após o envio do nodo para a máquina cliente, é criada uma *stream* de entrada (in) na qual a *thread* connect irá receber a resposta do cliente (packet resp = (packet) in.readObject();). Enquanto a resposta não vem, a *thread* fica bloqueada, liberando o processador para executar outras tarefas.

A máquina cliente deve receber o pacote, extrair o nodo, avaliá-lo até a WHNF e enviá-lo de volta à G-Machine juntamente com o seu endereço na *spark pool*:

```
      (...)
packet clientpacket = (packet) p.readObject();
Node resp = (Node) GM.reduce (clientpacket.node);
```

```
public static Node reduce (Node main)
{
  GM.push(main);
  GM.eval();
  return (GM.head());
}
```

FIGURA 4.6 – método reduce da G-Machine Cliente

```
out = new ObjectOutputStream (theSocket.getOutputStream());
out.writeObject (new packet (resp, clientpacket.add));
      (...)
```

O método readObjetct() lê o objeto que chega pela stream (nesse caso p). O nodo é retirado do pacote e enviado para o método reduce da máquina cliente, que o avalia até a WHNF (ver figura 4.6) usando a instrução GM.eval() explicada anteriormente.

O nodo na WHNF é então empacotado novamente junto com seu endereço na *spark pool*, e é enviado de volta para a máquina principal (método writeObject()) usando uma *stream* de saída conectada ao *socket*.

Quando a *thread* connect recebe a resposta do cliente, ela coloca o nodo de volta em seu endereço na *spark pool* usando o método updatenode:

```
      (...)
packet resp = (packet) in.readObject();
GM.pool.updatenode(resp.add ,resp.value);

      (...)
```

Este processo é repetido até que a G-Machine principal acabe de avaliar o programa principal. Então as *threads* são finalizadas e o ambiente de execução é terminado.

4.4.4 Modificação na instrução UNWIND

A G-Machine principal sempre encontrará os nodos do tipo Npool durante a execução da instrução GM.unwind(). Caso encontre um nodo desse tipo, ela deve simplesmente retirar o nodo correspondente da *spark pool* usando o método getnode. Caso o nodo esteja sendo avaliado em uma máquina cliente o método getonde irá devolver um objeto null. Nesse caso a G-Machine deve tentar avaliar um outro nodo que esteja disponível na *spark pool*. Isso é feito com a instrução GM.tryother:

```
boolean neednode = true;
while(neednode) {
      newval = pool.getnode (((Npool)node).add);
```

```
        if (newval == null)
        {GM.tryother(); }
        else
        {
        neednode=false;
        GM.push(newval);
        }
}
```

A instrução GM.tryother tenta retirar um nodo disponível usando o método fgetnode da *spark pool*. Se existe um nodo disponível, este é avaliado e devolvido para o seu lugar na *spark pool*, caso contrário a *thread* principal é colocada para *dormir* por algum tempo, pois não há nenhum trabalho a ser feito.

4.5 Exemplos de Programas Paralelos e Seus Tempos de Execução

Apresentam-se aqui sete programas paralelos implementados na linguagem FUN e seus tempos de execução em uma rede Ethernet, com máquinas Pentium 200 MMX rodando o sistema operacional Linux:

TABELA 4.1 – Tempo de Execução (t(s)) e *Speedup* (SU) dos Programas Paralelos

Programas	1 Proc.		2 Proc.		3 Proc.		4 Proc.	
	t(s)	SU	t(s)	SU	t(s)	SU	t(s)	SU
pfib 25	55s	1.0	34s	1.6	20s	2.75	20s	2.75
dsum (fatorial)	25s	1.0	14s	1.78	9s	2.77	9s	2.77
pcoins	28s	1.0	20s	1.4	17s	1.64	16s	1.75
peuler 200	1m10s	1.0	48s	1.45	33s	2.12	24s	2.91
minmax	20s	1.0	11s	1.81	9s	2.22	7s	2.85
ptak	42s	1.0	19s	2.21	19s	2.21	19s	2.21
listOfFibs	34s	1.0	20s	1.7	20s	1.7	20s	1.7

- pfib É um dos programas mais utilizados em *benchmarks* de programas funcionais paralelos [JUN98]. A versão utilizada aqui é otimizada para a G-Machine implementada.

- dsum Programa que aparece como exemplo no livro do Peyton Jones [PJO92a]. Calcula em paralelo a soma dos números de 1 até n. Este programa é também um dos programas exemplo que acompanham o Haskell Paralelo [TRI96].

- pcoins Este é um programa mais prático que usa funções muito utilizadas em programação funcional (ex. map e filter). Dadas uma coleção de moedas e um valor, calcula todas as maneiras de se pagar esse valor usando a coleção de moedas. O paralelismo é introduzido usando a função parmap. Este programa pertence ao *Glasgow nofib benchmark Suite* [PAR92], conjunto de programas para testar linguagens funcionais. Este programa aparece também na dissertação

de Doutorado de Junaidu [JUN98], na qual ele descreve a implementação da linguagem funcional paralela *Naira*.

- peuler Programa retirado da dissertação de doutorado de Mattson [MAT93], na qual ele apresenta algumas modificacoes no sistema GRIP, para realizar avaliação especulativa. Este programa faz a avaliação de uma lista de valores em paralelo tendo como resultado um bom *speedup*.

- minmax Algoritmo clássico da IA para implementação de jogos de tabuleiro. Cria tarefas que buscam em uma árvore, que representa os movimentos possíveis no jogo, o caminho que possui maior chance de levar a uma vitória.

- ptak Programa *divisão e conquista* do mesmo tipo do pfib que gera tarefas para calcular as chamadas recursivas. Retirado de [TRI96] e [JUN98].

- listOfFibs Gera a lista Fibonacci de uma maneira não muito eficiente. Programa serve para analisar como o sistema se comporta com a avaliação de uma lista de valores em paralelo, sendo que cada elemento leva um tempo computacional diferente para ser calculado. Programa retirado de [TRI96].

O tempo dos programas foi obtido usando System.currentTimeMillis() e não foi levado em conta o tempo de inicialização da máquina paralela.

A implementação desses programas encontra-se no Anexo 1.

Apesar dos programas conseguirem um bom *speedup* em relação a sua execução em apenas uma máquina, pode-se notar na tabela que alguns programas que poderiam ter um tempo de execução menor devido ao número de processadores disponíveis, possuem seu tempo de execução inalterado com mais de três processadores. Em alguns casos isso acontece pois o balanceamento de carga não é eficiente quando tarefas são geradas nas máquinas clientes. Deve-se notar também que mesmo os programas sequenciais são lentos, facilitando assim um bom *speedup* com a execução paralela.

Os programas e resultados apresentados aqui servem apenas como ilustração de como a máquina distribuída implementada se comporta. Apesar dos programas apresentados serem geralmente utilizados para testar linguagens funcionais paralelas, para uma análise mais qualificada da implementação, seria necessário um estudo mais profundo sobre quais programas utilizar para se analisar cada uma das características que se pode considerar importante em uma linguagem (funcional) paralela.

5 Trabalhos Futuros

5.1 Avaliação Especulativa

Analisando os tempos de execução dos programas paralelos, nota-se que muitas vezes, mesmo que se aumente o número de processadores, o tempo de execução dos programas não diminui. Isso acontece pois o número de tarefas geradas no programa é menor que o número de processadores disponíveis (ou por causa do balanceamento de carga). Uma maneira de se aumentar o número de tarefas seria trabalhar-se com *avaliação especulativa*.

A avaliação especulativa é uma estratégia para aumentar o paralelismo nos programas avaliando computações disponíveis mesmo antes de se saber se elas serão necessárias. Se o programa necessitar os resultados dessas computações mais tarde, o tempo gasto na avaliação desses valores tornará o programa muito mais rápido. Mesmo que os resultados sejam irrelevantes no futuro, o tempo gasto não irá gerar impacto na performance desde que sempre haja recursos disponíveis para as computações realmente necessárias.

Na avaliação especulativa de valores, criam-se tarefas não apenas quando o valor da tarefa vai ser necessário, mas também quando não se possui certeza da necessidade desse valor. A grande vantagem dessa abordagem é o aumento das oportunidades de paralelismo.

A transparência referencial da programação funcional permite que as computações irrelevantes geradas na avaliação especulativa possam ser descartadas com um *overhead* mínimo, pois as computações irrelevantes não afetam a avaliação das outras computações. Dessa maneira, seus efeitos não precisam ser *desfeitos* quando ela for descartada.

O grande perigo dessa abordagem é que os recursos da máquina podem ser consumidos avaliando partes do grafo que serão descartadas enquanto computações que serão realmente necessárias são adiadas.

Por exemplo pode-se considerar a seguinte expressão:

```
if e val1 val2
```

Apenas um valor, val1 ou val2, será usado na avaliação do if por isso a avaliação dos dois ao mesmo tempo irá desperdiçar os recursos da máquina. Por outro lado se existem recursos disponíveis pode-se avaliar val1 e val2 em paralelo com e, e quando acabar a avaliação de e o resultado selecionado para o if já possui a sua avaliação em estado adiantado [PJO87a].

5.1.1 Estendendo o Modelo para Tratar Avaliações Especulativas

A administração de tarefas especulativas é mais complicada que a de tarefas normais [MAT93]. Primeiramente, nenhuma tarefa especulativa deve ser executada enquanto existem tarefas normais esperando na *pool*. As tarefas normais realizam computações que realmente serão necessárias para o programa e as tarefas especulativas podem realizar tarefas que não serão úteis, podendo interferir na performance do programa. Deve-se observar também que se as tarefas normais começarem a ser adiadas, a execução do programa pode nunca terminar.

Quando se trabalha com avaliação especulativa, o sistema de administração das tarefas deve se preocupar com os seguintes casos[HUD83]:

- Como as tarefas especulativas competem por recursos com as tarefas normais, o sistema deve trabalhar com algum tipo de prioridade, dando sempre prioridade mais alta para as tarefas normais.
- Uma tarefa especulativa pode tornar-se irrelevante. Dessa maneira, ela deve ser terminada de forma prematura.
- Uma tarefa especulativa pode tornar-se realmente necessária. Neste caso ela deve possuir a mesma prioridade da tarefa normal.

Para modificar a G-Machine paralela implementada neste trabalho de maneira que suporte avaliação especulativa, pensou-se em adicionar uma outra *pool* à máquina principal, que seria a *pool especulativa*. Toda a tarefa especulativa gerada durante a avaliação do programa seria colocada nessa *pool* (ver figura 5.1).

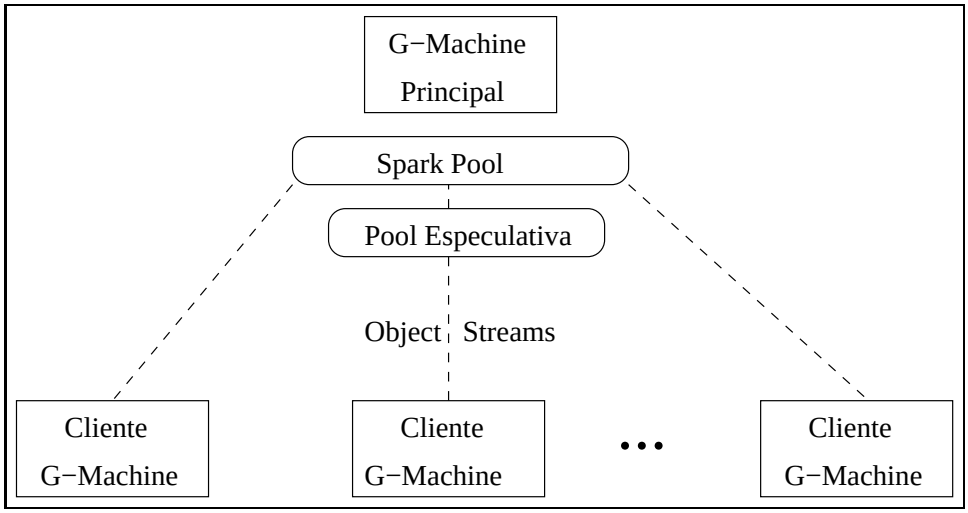

FIGURA 5.1 – Ambiente Funcional Distribuído com Avaliação Especulativa

Quando uma máquina cliente pedir uma tarefa para a máquina principal e a *spark pool* estiver vazia, deve-se buscar uma tarefa na *pool especulativa*. Com essa abordagem mantem-se as tarefas especulativas sempre com uma prioridade inferior.

Da mesma maneira a instrução GM.tryother() deve ser modificada para quando a máquina principal ficar bloqueda durante avaliação do programa e a *spark pool* estiver vazia, esta deve buscar tarefas na *pool* especulativa.

Quando uma tarefa especulativa está sendo avaliada por uma máquina cliente, e esta tarefa torna-se desnecessária, deve-se enviar uma mensagem à máquina cliente para que ela pare a avaliação. Quem deve iniciar esta mensagem é o próprio combinador que gerou a avaliação especulativa. Deve-se então, nessa abordagem, implementar-se funções que gerem a avaliação especulativa, da mesma maneira que par e parap começam a avaliação paralela normal.

Por exemplo, quando um if especulativo começar a avaliação do valor booleano, este deve enviar as duas outras tarefas para a *pool* especulativa. Quando souber o resultado do valor booleano, deve analizar os nodos na *pool* especulativa para tomar

a decisão do que deve ser feito: se o valor que não é necessário estiver sendo avaliado, a avaliação deve ser terminada.

Outra função que pode gerar avaliações especulativas é o or. O or especulativo pode gerar tarefas para avaliar os seus argumentos em paralelo e deve terminar a avaliação de um deles assim que o outro gerar um resultado true.

Pode-se também pensar em funções não-determinísticas para gerar as tarefas especulativas. Por exemplo, quando se trabalha com estruturas de dados do tipo árvore, muitas vezes criam-se várias tarefas para avaliar os ramos, mas deseja-se terminar as outras tarefas assim que uma encontrar uma folha, ou seja, um resultado dentre os possíveis.

O que apresentou-se aqui são apenas algumas idéias sobre o assunto. Existe uma série de outros problemas que devem ser considerados em uma análise mais profunda sobre o assunto. Por exemplo:

- O que fazer quando uma tarefa especulativa gera outras tarefas especulativas? Se uma tarefa especulativa possui 50% de chance de ser necessária, uma tarefa especulativa gerada por outra tarefa especulativa possui apenas 25%. No caso de se trabalhar com prioridades menores para essas tarefas, o que acontece quando a tarefa inicial não for mais especulativa e sim necessária? Modificar a prioridade de todas as sub-tarefas pode exigir muito esforço computacional.

- Uma tarefa especulativa que no final não é necessária, pode acabar consumindo todos os recursos disponíveis na computação de um valor inválido.

A avaliação especulativa de programas funcionais é mais adequada para máquinas superescalares.

5.2 Balanceamento de Carga

Na G-Machine paralela implementada, as chamadas par e parap são ignoradas quando acontecem nas máquinas clientes. Isso, algumas vezes, pode afetar o desempenho, quando se tem processadores ociosos enquanto um cliente avalia um nodo que poderia estar gerando tarefas para essas máquinas. Por outro lado, como dito anteriormente, o envio para a *pool* de várias tarefas que podem não ser utilizadas ou de grão fino, pode gerar um *overhead* muito maior do que o ganho com a paralelização.

Para reverter este problema, pensou-se em uma série de modificações possíveis no sistema de tempo de execução, de modo a melhorar o balanceamento de carga atualmente implementado.

Como o principal problema é a sobrecarga da rede e da máquina principal com o envio de tarefas para a *spark pool*, nesse novo modelo pensou-se em adicionar uma *spark pool* para cada máquina cliente (ver figura 5.2). Dessa maneira, sempre que ocorrer uma chamada par ou parap na máquina cliente, os nodos do grafo são enviados para a *spark pool* local, e não para a principal.

No novo modelo, a avaliação dos programas inicia da mesma maneira: a máquina principal começa a avaliação do programa, e depois de um tempo, passa a colocar partes do grafo na sua *spark pool*. Esses nodos são então enviados para os clientes que estiverem sem trabalho. O cliente recebe o nodo, faz a avaliação até a WHNF, envia a resposta para a máquina principal e espera por mais trabalho.

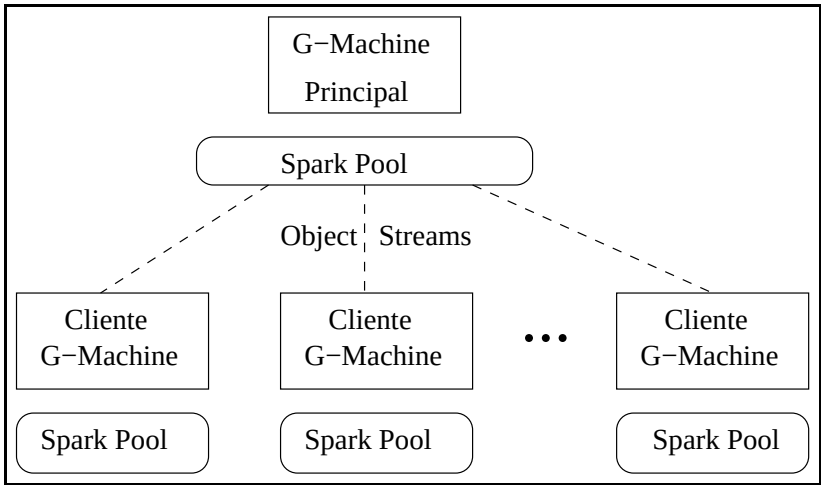

FIGURA 5.2 – Ambiente Funcional Distribuído com Balanceamento de Carga

As diferenças começam quando a máquina principal recebe um pedido de trabalho e sua *spark pool* esta vazia. Nessa situação, ela deve fazer um *multicast* do pedido (mensagem BUSCA) para todas as máquinas clientes. As máquinas clientes, quando recebem uma mensagem BUSCA, verificam sua *pool* local. Se existe algum nodo não avaliado, enviam uma mensagem em resposta (BUSCA_OK) que contém uma *cópia* do nodo e seu endereço na *pool* local.

A máquina principal armazena as respostas BUSCA_OK recebidas em uma outra pool (*pool de trabalhos possíveis*), juntamente com o endereço da máquina cliente que a enviou. A primeira mensagem BUSCA_OK recebida é marcada na *pool* de trabalhos possíveis como *sob-consideração*. A máquina principal então envia uma mensagem ASK_FOR_ACK para a cliente que enviou o nodo que está sendo considerado. Quando esta recebe a mensagem ASK_FOR_ACK ela verifica na sua *pool* se o nodo ainda não esta sendo avaliado. Em caso afirmativo, marca o nodo como *sendo avaliado* e envia um ACK como resposta senão uma mensagem NOPE.

Ao receber um ACK, a máquina principal pode enviar o nodo que estava sob-consideração para ser avaliado. Se receber um NOPE, elimina o nodo e coloca outro nodo da pool de trabalhos possíveis sob-consideração, recomeçando o processo.

Da mesma maneira a instrução GM.tryother() da máquina principal deve ser modificada. Primeiramente ela procura um nodo para avaliar na *spark pool*. Se não encontrar trabalho, procura um valor na *pool* de trabalhos possíveis. Se houver um valor na *pool* de trabalhos possíveis, este valor deve ser colocado sob-consideração, e deve ser enviada uma mensagens ASK_FOR_ACK, caso contrário deve-se enviar uma mensagem BUSCA.

É importante observar que entre uma mensagem BUSCA e uma mensagem ASK_FOR_ACK, o cliente pode avaliar o nodo que foi enviado em resposta a mensagem BUSCA. O nodo só fica bloqueado após o envio da mensagem ACK.

Este algoritmo traz algumas vantagens em relação ao anterior:

- Os nodos só são deslocados para a máquina principal quando eles forem realmente necessários;
- Apesar de ser preciso um número considerável de mensagens para poder começar a avaliação do valor, essas mensagens só são enviadas quando for necessário

o balanceamento de carga. Mesmo assim o tráfego na rede é bem menor do que se todos os nodos gerados pelas chamadas par e parap nos clientes fossem enviados para a máquina principal.

- O *multicast* da mensagem BUSCA evita várias mensagens perguntando para cada cliente se existem nodos disponíveis e a *pool* de trabalhos possíveis diminui a necessidade de mensagens BUSCA.

Este sistema também possui alguns problemas que devem ser considerados, por exemplo:

- O aumento do número de mensagens quando, durante a avaliação de um nodo recebido de uma máquina cliente, encontra-se uma referência à *spark pool* do cliente.
- A implementação é muito mais difícil devido ao número de mensagens diferentes, os problemas de sincronização e localidade.

Deve-se observar que este mesmo sistema de balanceamento de carga pode ser implementado no sistema de avaliação especulativa.

5.3 Outros Trabalhos

Vários outros trabalhos podem ser desenvolvidos em cima do que se começou com esta dissertação de mestrado. Aqui são listados alguns, com uma breve descrição:

Portar Características da Linguagem Java para os Programas Funcionais

Java possui várias bibliotecas de programação que poderiam ser utilizadas nos programas funcionais, como por exemplo, as de interface gráfica. Uma pesquisa interessante seria encontrar-se meios de utilizar essas bibliotecas nos programas funcionais. Uma abordagem interessante, seria a implementação de uma instrução do tipo ccall[PJO93b], utilizada no compilador Haskell GHC, para se chamar funções da linguagem C nos programas funcionais.

Compilação dos Programas

O compilador implementado neste trabalho é muito simples. Existem várias modificações, otimizações e adições que poderiam ser feitas no processo de compilação, como por exemplo:

- Adição de checagem de tipo;
- Implementar recursos sintáticos existentes usualmente nas linguagens de programação funcionais como facilidades para manipulação de listas, classes de tipo, *lambda lifting*, etc;
- Um trabalho interessante que pode ser desenvolvido, depois da adição das características citadas, é uma nova implementação do compilador na própria linguagem FUN. Dessa maneira fica-se livre da dependência do interpretador Hugs.

Otimização da Máquina Abstrata

Apesar de a maioria das linguagens funcionais implementadas serem baseadas na G-Machine ou em uma de suas variações [WAK98a], existem uma série de otimizações que podem ser feitas nesta máquina abastrata. Pode-se citar por exemplo a *Spineless Tagless G-Machine* [PJO92b] que é usada na implementação da linguagem Haskell no compilador GHC.

Integração com o Compilador GHC

O compilador Haskell GHC possui, dentre as suas várias fases de compilação [PJO93a], uma fase que simplifica os programas Haskell para uma linguagem mais simples, chamada de STG. Um trabalho interessante seria a utilização do compilador Haskell GHC como *front-end* para a G-Machine implementada, permitindo assim a compilação de programas Haskell para Java. Para isso, seria necessário implementar um compilador que transforme STG em FUN.

Ferramentas para a Análise da Execução Paralela

Uma área de pesquisa ativa em programação funcional paralela é o desenvolvimento de ferramentas que ajudam a compreender a execução dos programas. Como os programas funcionais paralelos são geralmente de alto nível, deixando grande parte do trabalho para o sistema de tempo de execução, fica difícil saber o motivo pelo qual um programa não apresenta o comportamento desejado. Por isso, é interessante o desenvolvimento de ferramentas que apresentem estatísticas do uso da *spark pool*, o número de tarefas que foram criadas, quantas tarefas foram realizadas nos clientes, qual o tempo de execução de cada máquina cliente, etc. Quando se tem um sistema totalmente desenvolvido na universidade, como é o caso da linguagem FUN, fica mais fácil de se desenvolver esse tipo de ferramenta.

Pré-Processador que Insere Automaticamente os Combinadores Paralelos

Apesar de, neste trabalho, os combinadores que expressam o paralelismo serem inseridos pelo programador, existem pesquisas que indicam que esses combinadores poderiam ser inseridos automaticamente no código [HAM94]. Nesse caso é necessário que se faça uma análise no fonte do programa para se descobrir quais expressões são realmente estritas, nesse caso podendo ser avaliadas em paralelo com o resto do programa.

6 Conclusões

Neste trabalho apresentou-se a implementação de uma linguagem funcional paralela que é compilada para Java. A idéia dessa linguagem funcional é utilizar máquinas conectadas em uma rede simples Ethernet, rodando a máquina virtual Java, para construir uma máquina paralela MIMD para a avaliação de programas funcionais paralelos.

Philip Wadler em seu artigo *Why no one uses functional languages* [WAD98], cita vários motivos pelos quais as linguagens funcionais não são tão populares. Dentre eles, destacam-se a pouca compatibilidade com outras linguagens de programação, o pequeno número de bibliotecas de programação de propósito geral e a dificuldade de se portar programas funcionais para várias arquiteturas. No artigo ele também cita pesquisas que estão sendo realizadas para se contornar este quadro.

Encontrar meios para que as linguagens funcionais conectem-se com outras linguagens de programação parece ser uma forma interessante de tornar as linguagens funcionais mais populares pois, dessa maneira, o programador poderia adaptar-se melhor ao estilo funcional de programação usando linguagens funcionais apenas em alguns trechos do programa. Por outro lado, o usuário de linguagens funcionais poderia beneficiar-se de bibliotecas de programação de outras linguagens. Existem vários trabalhos sobre a conexão de linguagens funcionais com outras linguagens de programação, como por exemplo [MEI2000], [PJO98], [SIB99], [FIN98], [WAL2001], [HUE96] e [DUB2000]. Dentre estes trabalhos, encontra-se várias pesquisas sobre a conexão de programas funcionais com programas escritos na linguagem Java.

Java é uma linguagem de programação baseada em objetos e com uma sintaxe similar a C++, desenvolvida pela Sun Microsystems, tendo como primeiro objetivo a programação de dispositivos eletrônicos inteligentes [DEI2001], mas que se tornou popular pelas suas facilidades na programação de conteúdo *Web* dinâmico e pelo grande número de bibliotecas de programação disponíveis. Um aspecto da linguagem Java que têm atraído programadores é o fato de os programas serem compilados para uma Máquina Virtual, a JVM (*Java Virtual Machine*). Dessa maneira os programas Java rodam em várias plataformas pois existem implementações da Máquina Virtual Java para a maioria das arquiteturas e sistemas operacionais.

Claus Reinke em seu artigo *Towards a Haskell/Java Connection* [REI99], aponta a compilação de programas funcionais para Java-Byte Code como sendo uma das mais promissoras formas de conexão do mundo das linguagens funcionais com o mundo Java. Existem vários trabalhos, citados anteriormente, utilizando essa abordagem.

As linguagens funcionais possuem uma série de vantagens que facilitam a sua execução paralela. Uma das características da linguagem Java que ajudaram a popularizar o seu uso foram as suas bibliotecas para a programação distribuída. De todos os trabalhos que tratam da integração de programação funcional com a linguagem Java que conseguimos analisar, nenhum utiliza essas características da programação funcional e da linguagem Java para a execução dos programas funcionais de forma distribuída em uma rede de computadores.

Neste trabalho dá-se continuidade ás pesquisas de integração de linguagens funcionais e Java, buscando utilizar as características de programação distribuída presentes em Java, para implementar uma linguagem funcional paralela. Essa lin-

guagem possui um sistema de tempo de execução que utiliza máquinas Java, espalhadas em uma rede, para formar uma máquina paralela que rode programas funcionais.

Espera-se que, com esta dissertação, tenha-se dado os primeiros passos para uma implementação eficiente de linguagens funcionais paralelas na aquitetura Java além de abrir caminhos para que outros possam no futuro continuar a estudar implementação de linguagens funcionais nesta universidade.

6.1 Problemas Encontrados

Um problema encontrado na implementação da máquina de redução de grafos com memória distribuída ocorre quando duas tarefas necessitam avaliar o mesmo grafo. Por exemplo:

```
let x = 4 * 5 in
parap (add (f x)) (y x);
```

Neste caso, se a tarefa que avalia f x for enviada para a máquina cliente antes de x ser avaliado na máquina principal, o x será avaliado mais de uma vez.

Este é um problema de difícil solução que pode deixar alguns programas paralelos menos eficientes.

Uma solução para esse problema talvez fosse a implementação de uma memória global acessada por todos os PEs, mas isso também gera uma série de outras preocupações e *overheads*, como por exemplo, manter a consistência dos dados que estão em cada máquina, localidade, comunicação etc. O aumento do número de mensagens na rede poderia acabar afetando drasticamente o sistema.

Outro problema da implementação é a *spark pool* que atualmente apresenta um comportamento aceitável para até 400 nodos, o que significa por volta de 800 acessos à *spark pool*. Já para programas divisão e conquista como o fib na versão apresentada na seção 4.1, o desempenho pode ser péssimo, dependendo do argumento da função. Por exemplo, um fib 25 gera mais de 15.000 acessos à *spark pool*.

6.2 Considerações Finais

Este trabalho contribuiu com idéias de como a arquitetura Java poderia ser utilizada para a implementação de uma linguagem funcional paralela. Primeiramente apresentou-se a implementação de uma linguagem funcional em Java usando a G-Machine. Em seguida, mostrou-se como foram utilizadas as facilidades para programação distribuída da linguagem Java para a implementação de um sistema de tempo de execução distribuído para uma linguagem funcional paralela. Por último, apresentou-se uma série de modificações e otimizações que poderiam ser implementadas no sistema.

Espera-se que o que se começou com este trabalho, sirva de inspiração para que outros pesquisadores interessados em linguagens funcionais possam dar continuidade às idéias propostas aqui.

Anexo 1 Fonte dos Programas Funcionais

A.1 Prelúdio da Linguagem FUN

```
cons a b = Cons {2,2} a b;

nil = Cons {1,0};


head xs = case xs of
        <1> -> 0;
        <2> a b -> a;

tail xs = case xs of
        <1> -> nil;
        <2> a b -> b;

dropWhile p xs = case xs of
  <1> -> nil;
  <2> y ys -> if (p y) (dropWhile p ys) (xs);

mkpair a b =  Cons {1,2} a b;

zip xs ys = case xs of
 <1> -> nil;
 <2> p ps -> cons (mkpair p (head ys)) (zip ps (tail ys));

map f l = case l of
        <1> -> nil;
        <2> a b ->  cons (f a) (map f b);

filter val xs = case xs of
   <1>          -> nil;
   <2> p ps -> let rest = filter val ps
               in
               if (val p) (cons p rest) rest;


append xs ys = case xs of
  <1> -> ys;
  <2> p ps -> cons p (append ps ys);

length xs = case xs of
  <1> -> 0;
  <2> p ps -> 1 + length ps;

from n = cons n (from (n+1));


concat = foldr append nil;

foldr f z xs = case xs of
  <1> -> z;
  <2> p ps -> f p (foldr f z ps);

sumList xs = case xs of
  <1> -> 0;
  <2> p ps -> p + (sumList ps);
```

A.2 Programa nfib em FUN

```
fib n = if (n<=1) 1 ( fib (n-1) + fib (n-2) +1);

main = fib 25
```

A.3 Programa euler em FUN

```
sumEuler n = (sumList (map euler (fromupto 1 n)));

fromupto x y = if (x==y) nil (cons x (fromupto (x+1) y));

euler n = length (filter (teste n) (fromupto 1 n));

teste a b = (hcf a b) == 1;

rem x y = x - (y * (x/y));

hcf x y = if (y==0) x (hcf y (rem x y));

main = sumEuler 200
```

A.4 Programa coins em FUN

```
mynub xs = case xs of
  <1> -> nil;
  <2> p ps -> cons p (mynub (removeDuplicates p ps));


removeDuplicates d xs = case xs of
  <1> -> nil;
  <2> p ps -> if (p == d) (removeDuplicates d ps) (cons p (removeDuplicates d ps));


del xs y = case xs of
  <1> -> nil;
  <2> p ps -> if (y == p) (ps) (cons p (del ps y));

pay_num pri val coins = if (val ==0) 1 (paynum pri val coins);

paynum pri val coins = case coins of
  <1> -> 0;
  <2> p ps -> let bar_coins = (dropWhile (maior val) coins) in
        sumList (map (aux pri val bar_coins) (mynub bar_coins));

maior a b = b>a;

aux pri val coins1 c = pay_num (pri-1) (val-c) (del (dropWhile (maior c) coins1) c);

vals = cons 250 (cons 100 (cons 25 (cons 10 (cons 5 (cons 1 nil)))));

quants = cons 5 (cons 8 (cons 8 (cons 9 (cons 12 (cons 17 nil)))));

coinsz = (zip vals quants) ;

coins = concat (map separa coinsz);

separa tupla = case tupla of
  <1> a b -> fromupto a 1 b;

fromupto a x y = if (x==y) (cons a nil) (cons a (fromupto a (x+1) y));


main = pay_num 100 137 coins
```

A.5 Programa sieve em FUN

```
find n xs = case xs of
        <1>      -> (negate 1);
        <2> x xs -> (if (n==0) x (find (n-1) xs));
```

```
sieve xs = case xs of
   <1>      -> nil;
   <2> p ps -> cons p (sieve (filter (nonMultiple p) ps));


nonMultiple p n = ((n/p) *p) ~= n;

main = find 300 (sieve (from 2))
```

A.6 Programa nfib em Mondrian

```
package testes.nfib;

import mondrian.prelude.*;

nfib = n ->
        if (n < 2)
                  1
          else
                  1 + nfib (n-1) + nfib (n-2);

main =
        putStr (nfib 25);
```

A.7 Programa euler em Mondrian

```
package testes.euler;

import mondrian.prelude.*;

main:IO;
main = putStr (sumEuler 200);

sumEuler = n -> sumList (map euler (fromupto 1 n));

euler = n -> length (filter (teste n) (fromupto 1 n));

teste = a -> b->  (hcf a b) == 1;

rem = x -> y -> x - (y * (x/y));

hcf = x -> y -> if (y==0) x else (hcf y (rem x y));

fromupto  = x -> y ->
                if (x==y)
                        []
                else (x :: (fromupto (x+1) y));

sumList = xs ->
        switch (xs) {
        case []: 0;
        case (y::ys): y + (sumList ys);

};
```

A.8 Programa coins em Mondrian

```
package testes.coins;

import mondrian.prelude.*;

main : IO;
main =         putStr (pay_num 100 137 coins);
```

```
sumList = xs ->
        switch (xs) {
        case []: 0;
        case (y::ys): y + (sumList ys);

};

mynub = xs ->
        switch(xs) {
        case []: [];
        case (y::ys): y:: (mynub (removeDuplicates y ys));
};

removeDuplicates = d -> xs ->
        switch(xs) {
        case []: [];
        case (p::ps):
                if (p==d)
                        (removeDuplicates d ps)
                else
                        (p:: (removeDuplicates d ps));

};


del = xs -> y ->
        switch(xs){
        case []: [];
        case (p::ps):
                if (y==p)
                ps
                else
                (p::(del ps y));

};

pay_num = pri -> val -> coins ->
                                if (val ==0)
                                 1
                                 else
                                 (paynum pri val coins);

paynum = pri -> val -> coins ->
        switch(coins){
        case []: 0;
        case (p::ps):let {bar_coins = (dropWhile (maior val) coins);} in
                sumList (map (aux pri val bar_coins) (mynub bar_coins));
};

maior = a -> b -> (b>a);

dropWhile = p -> xs ->
        switch (xs){
        case []: [];
        case (y::ys): if (p y)
                        (dropWhile p ys)
                      else
                              xs;
};

aux = pri -> val -> coins1 -> c ->
        pay_num (pri-1) (val-c) (del (dropWhile (maior c) coins1) c);


vals = 250 :: (100 :: (25 :: (10 :: (5 :: (1 ::[])))));

quants = 5 :: (8 ::( 8::( 9 :: (12 :: (17 :: [])))));


class Tuple;
class Pair extends Tuple {first:Code; second:Code;};
```

```
mkPair =  a-> b-> new Pair {first = a; second = b;};

zip = xs -> ys ->
        switch(xs){
        case []: [];
        case (p::ps): (mkPair p (head ys)) :: (zip ps (tail ys));
};

coinsz = (zip vals quants) ;

coins = concat (map separa coinsz);

separa = tupla -> switch (tupla) {
                case Pair { a=first; b=second; }: fromupto a 1 b;

};

head = xs ->
        switch(xs){
        case (p::ps):p;

        };

tail = xs ->
        switch(xs){
        case (p::ps):ps;

        };

fromupto = a -> x -> y ->
         if (x==y) (a :: [])
        else (a::(fromupto a (x+1) y));
```

A.9 Programa sieve em Mondrian

```
package testes.sieve;

import mondrian.prelude.*;

main : IO;
main =
        putStr (find 100 (sieve (from 2)));

sieve : List -> List;
sieve = xs ->
        switch (xs) {
                case (y::ys): y :: sieve (filter (notMultiple y) ys);
        };

find : Int -> List -> Int;
find = n -> xs ->
        switch (xs){
                case (y::ys):
                        if (n==0) y
                        else
                        (find (n-1) ys);
        };
```

```
notMultiple : Int -> Int -> Bool;
notMultiple = x -> y ->
        not ((y / x) * x == y);
```

A.10 Programa coins paralelo em FUN

```
parmap f l = case l of
        <1> -> nil;
        <2> a b -> let fa = f a;
                       fb = parmap f b
                       in seq (par fa fb) (cons fa fb);


mynub xs = case xs of
  <1> -> nil;
  <2> p ps -> cons p (mynub (removeDuplicates p ps));


removeDuplicates d xs = case xs of
  <1> -> nil;
  <2> p ps -> if (p == d) (removeDuplicates d ps) (cons p (removeDuplicates d ps));


del xs y = case xs of
  <1> -> nil;
  <2> p ps -> if (y == p) (ps) (cons p (del ps y));

pay_num pri val coins = if (val ==0) 1 (paynum pri val coins);

pay_num2 pri val coins = if (val ==0) 1 (paynum2 pri val coins);

paynum2 pri val coins = case coins of
  <1> -> 0;
  <2> p ps -> let bar_coins = (dropWhile (maior val) coins) in
        sumList (map (aux pri val bar_coins) (mynub bar_coins));
paynum pri val coins = case coins of
  <1> -> 0;
  <2> p ps -> let bar_coins = (dropWhile (maior val) coins) in
                  let list = (parmap (aux pri val bar_coins) (mynub bar_coins)) in
              seq list (sumList list);


maior a b = b>a;

aux pri val coins1 c = pay_num2 (pri-1) (val-c) (del (dropWhile (maior c) coins1) c);

vals = cons 250 (cons 100 (cons 25 (cons 10 (cons 5 (cons 1 nil)))));

quants = cons 5 (cons 8 (cons 8 (cons 9 (cons 12 (cons 17 nil)))));

coinsz = (zip vals quants) ;


coins = concat (map separa coinsz);

separa tupla = case tupla of
  <1> a b -> fromupto a 1 b;

fromupto a x y = if (x==y) (cons a nil) (cons a (fromupto a (x+1) y));

main = pay_num 100 137 coins
```

A.11 Programa euler paralelo em FUN

```
parmap f l = case l of
        <1> -> nil;
```

```
        <2> a b -> let fa = f a;
                       fb = parmap f b
                       in seq (par fa fb) (cons fa fb);


sumEuler n = let list = (parmap euler (fromupto 1 n)) in
               (seq list (sumList list));

euler n = length (filter (teste n) (fromupto 1 n));

teste a b = (hcf a b) == 1;

rem x y = x - (y * (x/y));

hcf x y = if (y==0) x (hcf y (rem x y));

fromupto x y = if (x==y) nil (cons x (fromupto (x+1) y));

main = sumEuler 200
```

A.12 Programa fib paralelo em FUN

```
pfib n = let pfib1 = pfib (n-1)
        in
        if (n<=1) 1 (parap (seq pfib1 (add pfib1)) (fib (n-2) +1));

fib n = if (n<=1) 1 ( fib (n-1) + fib (n-2) +1);

add a b = a + b;

main = pfib 25
```

A.13 Programa listoffibs paralelo em FUN

```
forcel xs = force4 xs xs;

force4 xs ys = case xs of
  <1> -> ys;
  <2> p ps -> seq p (force4 ps ys);

fib n = if (n<=1) 1 ( fib (n-1) + fib (n-2) +1);

parmap f l = case l of
        <1> -> nil;
        <2> a b -> let fa = f a;
                       fb = parmap f b
                       in seq (par fa fb) (cons fa fb);

fromupto x y = if (x==y) (cons y nil) (cons x (fromupto (x+1) y));

main = seq (forcel (parmap fib (fromupto 10 22))) 1
```

A.14 Programa dsum paralelo em FUN

```
dsum lo hi = let sum2 = dsum lo (mid lo hi)
        in
         if (lo == hi) hi (parap (seq sum2 (add sum2))
                                 (sum ((mid lo hi)+1) hi));


sum lo hi = let sum2 = sum lo (mid lo hi)
        in
        if  (lo == hi) hi (add sum2 (sum ((mid lo hi)+1) hi));
```

```
mid lo hi = (lo+hi)/2;

psum n = dsum 1 n;

add a b = a+ b;

main = psum 1000
```

A.15 Programa tak paralelo em FUN

```
tak x y z = if (x<=y) z (let x1 = tak (x-1) y z;
                             y1 = tak (y-1) z x;
                             z1 = takseq (z-1) x y in
                         seq (par x1 (par y1 z1)) (takseq x1 y1 z1));

takseq x y z = if (x<=y) z (let x1 = takseq (x-1) y z;
                                    y1 = takseq (y-1) z x;
                                    z1 = takseq (z-1) x y in
                                takseq x1 y1 z1);


main = tak 18 12 6
```

A.16 Programa minmax paralelo em FUN

```
tree a b =  Cons {1,2} a b;

getbranch tree = case tree of
        <1> a b -> b;

getnode tree = case tree of
        <1> a b -> a;


parmap f l = case l of
        <1> -> nil;
        <2> a b -> let fa = f a;
                       fb = parmap f b
                       in seq (par fa fb) (cons fa fb);

minimo list = case list of
        <1> -> 0;
        <2> x xs -> search x xs;

search head tail = case tail of
        <1> -> head;
        <2> x xs -> search (min head x) xs;

min head x = if (head < x) head x;

game new p = tree p (map (game new) (new p));

stat f node = static2 f (getbranch node) node;

static2 f l node = case l of
        <1>-> tree (f (getnode node)) l;
        <2> x xs -> tree (f (getnode node)) (map (stat f) l);

minmax tree = pminmax tree (getbranch tree);

minmax2 tree = seqminmax tree (getbranch tree);

pminmax tree l = case l of
        <1> -> getnode tree;
        <2> x xs -> let plist = parmap minmax2 l
                        in
                        seq plist (negate (minimo plist));
```

```
seqminmax tree l = case l of
        <1> -> getnode tree;
        <2> x xs -> negate (minimo (map minmax2 l));


cut int tree = if (int == 0) (cut2 tree) (cut3 int tree);

cut2 arv = case arv of
        <1> x xs -> tree x nil;

cut3 int arv = case arv of
        <1> x xs -> tree x (map (cut (int-1)) xs);

staticev a = 1;

nova a = (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 nil)))));

main = minmax (stat staticev (cut 6 (game nova 1)))
```

Anexo 2 Fontes das Classes Principais

B.1 Arquivo GM.java da máquina paralela principal

```
//////////////////////////////////////////////////////////////////////
// © 2001 André Rauber Du Bois                                      //
// File: GM.java                                                    //
//                                                                  //
// This is the implementation of the main parallel G-Machine. This  //
// class stays on the main PE. It starts the evaluation of the      //
// program and sends the nodes to the spark pool. It also starts    //
// the threads that handles the connections with the clients.       //
//                                                                  //
//////////////////////////////////////////////////////////////////////
package gmachine.gm;

import gmachine.lang.*;
import java.net.*;
import java.io.*;

// The GM.class is the implementation of the Main G-Machine which
// starts the threads and the spark pool.
// It has some static methods that are the G-Machine
// instructions.

public class GM {

 // The Stack
 private static Gstack stack = new Gstack();


 //The Spark Pool

 static Gpool pool = new Gpool();


 //Instruction GM.par used to send nodes to the Spark Pool
 public static void par ()
 {
  Node node = GM.head();
  stack.pop(1);
  int add = pool.putnode(node);
  GM.put (new Npool(add));
 }

 // GM.push -- pushes a node that is already in the stack
 //onto the top of the stack
 public static void push (int n)
 {
   stack.push (n);
 }

 //GM.pop -- pops a node out of the stack
 public static void pop (int n)
 {
   stack.pop (n);
 }


 //GM.mkap -- creates an application node
 public static void mkap ()
 {
   stack.mkap();
 }

 //GM.slide -- discard pointers on the stack
 public static void slide (int n)
 {
   stack.slide(n);
 }
```

```
//GM.update -- updates the root of the redex
public static void update (int n)
{
  stack.update (n);
}

//GM.head -- returns a reference to the first element of the stack
public static Node head ()
{
  return stack.head();
}

//GM.put -- Pushes a node onto the stack
public static void put (Node a)
{
  stack.put (a);
}

//showstk -- prints the state of the stack for debuging
public static void showstk ()
{
  stack.showstk ();
}

//rearrange -- Rearranges the stack to call the instructions of a
//supercombinator
public static void rearrange (int n)
{
  stack.rearrange(n);
}

//negates an integer
public static void negate()
{
  Node a = GM.head();
  GM.pop(1);
  GM.put (new Int (-((Int) a).value()));
}

//strict == --- equals an basic type
public static void eq()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put ( ( ((Nbasic) a).equal(((Nbasic) a),((Nbasic) b))) );
}


//GM.or -- strict or
public static void or()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put ( new Bool ((((Bool) a).value()) || (((Bool) b).value()))  );
}


//GM.and -- strict and
public static void and()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put ( new Bool ((((Bool) a).value()) && (((Bool) b).value()))  );
}

//GM.gt -- strict >
public static void gt()
```

```
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Bool ((((Int) a).value()) > (((Int) b).value())));
}

//GM.lt -- strict <
public static void lt()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Bool ((((Int) a).value()) < (((Int) b).value())));
}

// GM.ge --    >=
public static void ge()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Bool ((((Int) a).value()) >= (((Int) b).value())));
}

//GM.le -- <=
public static void le()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Bool ((((Int) a).value()) <= (((Int) b).value())));
}

// GM.ne -- ~=
public static void ne()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Bool ((((Int) a).value()) != (((Int) b).value())));
}


//GM.add -- strict sum
public static void add()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Int ((((Int) a).value()) + (((Int) b).value())));
}

//GM.add -- multiplication of integers
public static void mul()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);

  GM.put (new Int ((((Int) a).value()) * (((Int) b).value())));
}

//GM.sub -- subtraction
public static void sub()
{
```

```
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Int ((((Int) a).value()) - (((Int) b).value())));
}

// Integer Division
public static void div()
{
  Node a = GM.head();
  GM.pop(1);
  Node b = GM.head();
  GM.pop(1);
  GM.put (new Int ((((Int) a).value()) / (((Int) b).value())));
}

//GM.pack -- Creates a Cons
public static void pack (int t, int a)
{
 Node args[] = new Node [a];
 int i;

 for (i = 0; i < a; i++)
 {
  args[i] = (Node) GM.head();
  GM.pop(1);
 }
 GM.put (new Nconstr (t,a,args));
}


// GM.split -- pushes onto the stack the arguments of the Cons
public static void split (int n)
{
 Nconstr c = (Nconstr) GM.head();
 GM.pop(1);
 int i;
 int count =(n-1);

 for (i=0;i<n;i++)
 {
     GM.put ((Node) c.args[count]);
   count--;
 }
}

//GM.eval -- evaluates to WHNF the top element of the stack
public static void eval()
{
  int n=0;

  if (!((stack.head() instanceof Nbasic)
       || (GM.head() instanceof Nconstr)))
   {

     while (!((GM.head() instanceof Nbasic)
       || (GM.head() instanceof Nconstr)))
     {
      n++;
      GM.unwind();
     }

   }
}


//GM.unwind -- unwinds the spine of the graph
public static void unwind ()
{
  Node node = (Node) GM.head();

  if (node.ind !=null)
```

```
   {
          Node ind = (node).ind;
          GM.pop (1);
          GM.put (ind);
   }
  else
   {
    if (node instanceof Nap)
    {
      GM.put (((Nap)node).func);
    }
    else
     {
       if (node instanceof Nsuperc)
       {
        if (((Nsuperc)node).nargs != 0)
        {
        GM.pop (1);
        }
        GM.rearrange(((Nsuperc)node).nargs);
        ((Nsuperc)node).code();
       }
       else {
         if (node instanceof Npool)
             {
          boolean neednode = true;
          Node newval;
          GM.pop(1);
          while(neednode)
         {
          newval = pool.getnode (((Npool)node).add);
          if (newval == null)
          {
            GM.tryother();
          }
          else
          {
           neednode=false;
           if (newval instanceof Nap){
                 if (((Nap)newval).ind instanceof Npool)
                        { ((Nap)newval).ind = null;} }
                       GM.put (newval);
                       GM.eval();
                       pool.updatenode(((Npool)node).add ,GM.head());

             }
         }
        }
       }
     }

  }
}


 //GM.tryother -- tries to evaluate an available node from the spark
 //pool
 public static void tryother()
 {
    packet mypacket = pool.fgetnode();
   if (mypacket.message != -1)
   {
   if (mypacket.value instanceof Nap)
    {
      if (mypacket.value.ind instanceof Npool)
       {
           mypacket.value.ind = null;
       }
    }
    GM.put (mypacket.value);
    GM.eval();
    Node resp = GM.head();
    GM.pop(1);
```

```
    pool.updatenode(mypacket.add ,resp);
       }
    else
    {  try{
          Thread.currentThread().sleep (100);
          }catch (Exception e){System.out.println(e);}

    }
 }



 // Prints a list

 public static void printlist ()
{
  Node val = GM.head();

  System.out.print("[");
  GM.put(((Nconstr)val).args[0]);
  GM.eval();
  System.out.print((((Int)GM.head()).value()));
  GM.pop(2);
  GM.put(((Nconstr)val).args[1]);
  GM.eval();
  val = GM.head();
           while (!(((Nconstr)val).nargs == 0))
            {
             System.out.print(",");
             GM.put(((Nconstr)val).args[0]);
             GM.eval();
             System.out.print((((Int)GM.head()).value()));
             GM.pop(2);
             GM.put(((Nconstr)val).args[1]);
             GM.eval();
             val = GM.head();
            }

  System.out.print("]");

}




 //This is the main method of the Parallel G-Machine
 // It receives the main CAF and retuns a value in the WHNF

  public static void reduce (Nsuperc main)
 {
        Node val;
        String resul;
        ThreadGroup my_server = new ThreadGroup("server");

        //Creates the server thread
        server s = new server();
        Thread th = new Thread (my_server,s);
        Thread princ = Thread.currentThread();

        th.setPriority((princ.getPriority()-1));
        pool.setthread(th);
        th.start();


        //Now waits for connections from the Clients
        System.out.println ("Hit <enter> when all clients are connected");
        try{
        System.in.read();
        } catch (IOException e){}
        //After the user hits ENTER it starts the avaluation of the program

        GM.put(main);
```

```
        main.code();
        val = (Node) GM.head();
        while (!(val instanceof Nbasic))
        {
          GM.unwind();
          val = (Node) GM.head();
        }


        if (val instanceof Int)
        {
         resul=new String("" + (((Int)val).value())+"");
         System.out.println(resul);
        }
        else
        {
          if (val instanceof Bool)
          {
           resul=new String("" + (((Bool)val).value())+"");
           System.out.println(resul);
          }
          else
          {

            if (((Nconstr)val).nargs == 0)
            {
              System.out.println("[]");
            }
            else
            {
            GM.put(((Nconstr)val).args[0]);
            //GM.showstk();
            GM.eval();
           // GM.showstk();
            if (GM.head() instanceof Nconstr)
            {
              System.out.print("[");

              GM.printlist();
              GM.pop(2);
              GM.put(((Nconstr)val).args[1]);
              GM.eval();
              val = GM.head();

             while (!(((Nconstr)val).nargs == 0))
             {
              System.out.print(",");
              GM.put(((Nconstr)val).args[0]);

              GM.eval();
              GM.printlist();
              GM.pop(2);
              GM.put(((Nconstr)val).args[1]);
              GM.eval();
              val = GM.head();

             }
             System.out.print("]");

            }else {GM.pop(1); GM.printlist(); }
            }

            }

        }

 }

}
```

B.2 Arquivo server.java

```
////////////////////////////////////////////////////////////////////
// © 2001 André Rauber Du Bois                                    //
// File: server.java                                              //
//                                                                //
// The server class handles the connections with the clients.     //
// It creates one Thread (thread connect) to each client machine  //
// connected with the main PE. The connect thread send the node   //
// that are in the spark pool to be evaluated in the client       //
// machines                                                       //
////////////////////////////////////////////////////////////////////
package gmachine.gm;

import gmachine.lang.*;
import java.net.*;
import java.io.*;

//The server class creates one Thread for each client connection. These
//Threads hadles the connection with the clients.

public class server implements Runnable {

 public final static int port = 1515;
 ObjectOutputStream p;
 ServerSocket ss;


 public void run()
 {
   try{
        ss = new ServerSocket(port);
        System.out.println ("Accepting Connections");
        while (true) {

                //Creates the connect Thread
                connect fs = new connect (ss.accept());
                fs.start(); // starts the Thread
                System.out.println ("New connection!!");
                }
      }catch (IOException e){}
 }
}

//The connect Thread handles the connections with the clients. It
//takes the nodes out of the spark pool and send them to the client machines.
//Then it waits for the node in the WHNF

class connect extends Thread{

     Socket connection;

        public connect (Socket s){
         connection = s;
        }

       public void run(){

         try{
           ObjectOutputStream p
             = new ObjectOutputStream(connection.getOutputStream());
           boolean empty=true;

        try{
           while (empty){
```

```
         if (GM.pool.getsize()!=0)
                {
                //Takes the node out of the spark pool
                packet pack = GM.pool.fgetnode();
                if (pack.message != -1)
                {
                 // Sends it to the client Machine
                 p.writeObject(pack);
                 p.flush();

                 ObjectInputStream in
                       = new ObjectInputStream(connection.getInputStream());

                 // Waits for the response of the Client
                 //The Threads remains blocked until the client response.
                 packet resp = (packet) in.readObject();
                 // Updates the node in the spark pool with its
                 // new value in the WHNF
                 GM.pool.updatenode(resp.add ,resp.value);
                }
                else
                {
                // if there is no node in the spark pool (message = -1)
                // stop the cliente (message = 1)
                  pack.message = 1;
                  p.writeObject(pack);
                }
              }
          }
         }catch (Exception f) {
         System.out.println ("Problemas na volta!" + f);
         }
        }catch (IOException e) {
         System.out.println ("Problemas");
        }

    }
}
```

B.3 Arquivo client.java

```
//////////////////////////////////////////////////////////////////////
// © 2001 André Rauber Du Bois                                      //
// File: client.java                                                //
//                                                                  //
// This is the main class of the Client G-Machine.  It receives     //
// the nodes from the main PE and evaluates them to WHNF using      //
// the local G-Machine.                                             //
//                                                                  //
//                                                                  //
//////////////////////////////////////////////////////////////////////
import gmachine.gm.*;
import java.net.*;
import java.io.*;

import gmachine.lang.*;

import gmachine.prelude.*;

//The Client Class handles the connection with the main PE.
//It asks for nodes to the main PE. Whe it recieves the node, it sends
//it to the local G-Machine. The the node is in WHNF the client sends
//it back to the main PE.

public class client{


        public static void main (String[] args) {

        Socket theSocket;
```

```
        ObjectInputStream p;
        ObjectOutputStream out;
        try {
        //Creats a socket connection with the main PE, which listens to
        // the port 1515
        theSocket = new Socket ("europa.inf.ufrgs.br",1515);
        System.out.println ("Connected!!");

        //creates the object stream
        p = new ObjectInputStream  (theSocket.getInputStream());
        try{
        while (true){
        //reads the packet
        packet packet = (packet) p.readObject();
        if (packet.message != 1){
        if (packet.value instanceof Nap)
        {
          if (packet.value.ind instanceof Npool)
          {
                packet.value.ind = null;
          }
        }
        //sends the node to the local G-Machine
        Node resp = (Node) GM.reduce (packet.value);

        //sends the node in WHNF back to the main PE
        out = new ObjectOutputStream  (theSocket.getOutputStream());
        out.writeObject (new packet (resp, packet.add));
        out.flush();

        }else
         {
         //  if message = STOP (message = 1)
          break;
        }


        }

        } catch (Exception e) {
        System.out.println (e);}
        } catch (IOException e) {
        System.out.println (e);}


 }
}
```

Anexo 3 Artigo *Functional Beans*

Anexo 4 Artigo *Distributed Execution of Functional Programs on the JVM*

Bibliografia

[AUG84] AUGUSTSSON, L. A Compiler for Lazy ML. In: ACM SYMPOSIUM ON LISP AND FUNCTIONAL PROGRAMMING, 1984, Austin. **Proceedings ...** [S.l.:s.n.], 1984. p. 218-227.

[AUG89] AUGUSTSSON, L.; JOHNSSON, T. Parallel Graph Reduction with the $\langle \nu, G \rangle$-Machine. In: CONFERENCE ON FUNCTIONAL PROGRAMMING AND COMPUTER ARCHITECTURE, 1989. **Proceedings ...** [S.l.]: ACM Press, 1989. p. 201-213.

[ARM92] ARMSTRONG, J. L. et al. Implemtenting a Functional Language for Highly Parallel Real Time Applications. In: SETSS, 1992, Florença. **Proceedings ...** [S.l.:s.n.], 1992.

[ARN97] ARNOLD, K.; GOSLING, J. **Programando em Java**. São Paulo: Makron Books, 1997.

[BAC78] 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.

[BEN98] BENTON, N.; KENNEDY, A.; RUSSELL, G. Compiling Standard ML to Java bytecodes. **SIGPLAN Notices**, [S.l.], v.34, n.1, Jan. 1999. Trabalho apresentado na International Conference on Functional Programming, 1999.

[BIR88] BIRD, R.; WADLER, P. **Introduction to Functional Programming**. [S.l.]:Prentice Hall, 1988.

[BIR98] BIRD, R. **Introduction to Functional Programming using Haskell**. 2nd ed. [S.l.]: Prentice Hall, 1998.

[BOT98] BOTHNER, P. Kawa: Compiling Scheme to Java. In: LISP USER'S CONFERENCE, 1998. **Proceedings ...** [S.l.:s.n.], 1998.

[BRE98] BREITINGER, S. et al. DREAM: the DistRibuted Eden Abstract Machine. In: IMPLEMENTATION OF FUNCTIONAL LANGUAGES, 1997. **Proceedings ...** Berlin: Springer-Verlag, 1998. p. 250-269.

[CAR2000] CARVALHO Jr., F. H. **Haskell$_{\#}$**: uma Extensão Paralela para Haskell. Recife: DI da UFPE, 2000. Dissertação de mestrado.

[CHA94] CHAKRAVARTY, M. M. T. A Self-Scheduling, Non-blocking, Parallel Abstract Machine for Non-strict Functional Languages. In: IFL, 1994. **Proceedings ...** England: University of East Anglia, 1994. p 1-34.

[CHU41] CHURCH, A. **The Calculi of Lambda Conversion**. [S.l.]: Princeton University Press, 1941.

[COL89] COLE, M. I. **Algorthmic Skeletons**: structured management of parallel Computation. [S.l.]: Pitman, 1989.

[DAR81] DARLINGTON, J.; REEVE, M. J. ALICE: a Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. In: FPCA, 1981. **Proceedings ...** [S.l.:s.n.], 1981. p 65-76.

[DAV92] DAVIE, A. J. T. **An Introduction to Functional Programming Systems Using Haskell**. [S.l.]: Cambridge University Press, 1992.

[DEI2001] DEITEL, H. M.; DEITEL, P. J. **Java, como Programar**. 3rd ed. Porto Alegre: Bookman, 2001.

[DUB2000] DU BOIS, A. R.; COSTA, A. C. R. Functional Beans. In: INTERNATIONAL WORKSHOP ON FUNCTIONAL AND LOGIC PROGRAMMING, WFLP, 9., 2000, Benicàsim. **Proceedings ...** Valencia: Univ. Politecnica de Valencia, 2000.

[FAR98] FARLEY, J. **Java Distributed Computing**. [S.l.]: O'Relly & Associates, 1997.

[FIN98] FINNE, S. H/Direct: a Binary Foreign Language Interface for Haskell. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, ICFP, 1998, Balitmore. **Proceedings ...** [S.l.]: ACM Press, 1998. p 153-162.

[FLA97] FLANAGAN, D. **Java in a Nutshell**: a Desktop Quick Reference. 2nd ed. [S.l.]: O'Relley & Associate, 1997.

[GAV97] GAVIDIA, Z. J. J.; AZEREDO, P. A. **Programação Funcional Usando Java**. Porto Alegre: CPGCC da UFRGS, 1997. Dissertação de Mestrado.

[GEI94] GEIST, G. A. **PVM** : Parallel Virtual Machine - a user's guide and tutorial for network parallel computing. Cambridge: MIT Press, 1994.

[HAM93] HAMMOND, K. Getting a GRIP. In: INTERNATIONAL WORKSHOP ON PARALLEL FUNCTIONAL PROGRAMMING, 1993. **Proceedings ...** Nijmegen:[s.n.], 1993.

[HAL94] HALL, C. et al. Type Classes in Haskell. In: ESOP, 1994. **Proceedings ...** [S.l.:s.n.], 1994.

[HAM94] HAMMOND, K.; Parallel Functional Programming: An Introduction. In: INTERNATIONAL SYMPOSIUM ON PARALLEL SYMBOLIC COMPUTATION, PASCO, 1., 1994, Hagenberg/Linz, Austria. **Proceedings ...** Hagenberg/Linz:[s.n.], 1994.

[HAM99] HAMMOND, K.; WALSH-KEMISH, D. Juaskell: implementing Evaluation Strategies in Java. In: LATIN-AMERICAN CONFERENCE ON FUNCTIONAL PROGRAMMING, CLaPF, 3., 1999, Recife. **Proceedings ...** Recife: UFPE, 1999.

[HAN99] HANUS, M.; SADRE, R. An Abstract Machine for Curry and Its Concurrent Implementation in Java. **Journal of Functional and Logic Programming**, [S.l.], n.6, Mar. 1999.

[HAR97] HAROLD, E. R. **Java Network Porgramming**. [S.l.]: O'Relly & Associates, 1997.

[HUD83] HUDAK, P. Distributed Task and Memory Management. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 1983. **Proceedings ...** [S.l.:s.n.], 1983. p. 277-89.

[HUD2000] HUDAK, P. **The Haskell School of Expression**: learning Functional Programming through multimedia. Cambridge: Cambridge University Press, 2000.

[HUD2001] HUDAK, P.; PETERSON, J.; FASEL, J. H. **A Gentle Introduction to Haskell**: Version 1.4. Disponível em: (htttp://www.haskell.org). Acesso em: jan. 2001.

[HUE96] HUELSBERGEN, L. **A Portabel C Interface for Standard ML of New Jersey**. [S.l.]: AT&T Bell Laboratories, 1996.

[HUG89] HUGHES, J. Why Functional Programming Matters. **The Computer Journal**, [S.l.], v.32, n. 2, feb. 1989.

[HUG99] HUGHES, M.; SHOFFNER, M.; HAMNER, D. **Java Network Programming**. 2nd ed. [S.l.]: Manning Publications, 1999.

[JON96] JONES, R.; LINS, R. **Garbage Collection**: algorithms for automatic dynamic memory management. [S.l.]: John Wiley and Sons, 1996.

[JOY98] JOY, M.; MEEHAN, G. **Compiling Lazy Functional Programs to Java byte-code**. Coventry: Deparment of Computer Science, University of Warwick, 1998.

[JUN98] JUNAIDU, S. B. **A Parallel Functional Language Compiler for Message-Passing Multicomputers**. St. Andrews: School of Mathematical and Computational Sciences - University of St Andrews, 1998. Tese de Doutorado.

[KES96] KESSELER, M. **The Implementation of Functional Languages on Parallel Machines with Dstributed Memory**. Nijmegen:University of Nijmegen, 1996. Tese de Doutorado.

[LEA97] LEA, D. **Concurrent Programming in Java**: design principles and patterns. [S.l.]: Addison-Wesley, 1997.

[LOC2001] LOCKYER, V. **Hugs OnLine**. Disponível em: (http://www.haskell.org/hugs/). Acesso em: jan. 2001.

[MAT93] MATTSON Jr., J. S. **An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction**. San Diego: University of California, San Diego, 1993. Tese de Doutorado.

[MEI97] MEIJER, E.; CLAESSEN, K. The Design and Implementation of Mondrian. In: HASKELL WORKSHOP, 1997 **Proceedings ...** [S.l.:s.n.], 1997. Disponível em: http://www.cs.uu.nl/∼erik/) Acesso em: jan. 2001

[MEI2000] MEIJER, E.; FINNE, S. Lambada: Haskell as a Better Java. In: HASKELL WORKSHOP, PLI, 2000. **Proceedings ...** Montreal: [s.n.], 2000.

[MEI2001] MEIJER, E.; PATERSON, R. **Down with λ-lifting**. Disponível em: (http://www.cs.uu.nl/∼erik/). Acesso: jan. 2001.

[MPI95] MPI: A Message-Passing Interface Standard. Version 1.1. Tenessee: University of Tennesee, 1995.

[OAK97] OAKS, S.; WONG, H. **Java Threads**. [S.l.]: O'Reilly & Associates, 1997.

[OCO97] O'CONNOR, M. J.; TREMBLAY, M. PicoJavaI: The Java Virtual Machine in Hardware. **IEEE Micro**, [S.l.], v. 17, n. 2, p. 45-53, 1997.

[ODE97] ODERSKY M.; WADLER, P. Pizza into Java: translating theory into practice. In: ACM SYMPOSIUM ON THE PRINCIPLES OF PROGRAMMING LANGUAGES, 1997. **Proceedings ...** [S.l.:s.n.], 1997. p 146-149.

[PAR92] PARTAIN, W. The nofib Benchmark suite of Haskell Programs. In: GLASGOW WORKSHOP ON FUNCTIONAL PROGRAMMING, 1992. **Proceedings ...** Berlin: Springer-Verlag, 1992. p. 195-202.

[PJO87a] PEYTON JONES, S. L. **The Implementation of Functional Programming Languagens**. New York: Prentice Hall, 1987.

[PJO87b] PEYTON JONES, S. L. et al. GRIP - a High-Performance Architecture for Parallel Graph Reduction. In: FPCA, 1987. **Proceedings ...** Berlin: Springer-Verlag, 1987. p. 98-112. (LNCS, 274).

[PJO92a] PEYTON JONES, S. L.; LESTER, D. **Implementing Functional Languages**: a tutorial. New York: Prentice Hall, 1992.

[PJO92b] PEYTON JONES, S. L. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. **Journal of Functional Programming**, [S.l.], v. 2, n. 2, 1992.

[PJO93a] PEYTON JONES, S. L. et al. The Glasgow Haskell Compiler: a technical overview. In: JOINT FRAMAWORK FOR INFORMATION TECNOLOGY TECHNICAL CONFERENCE, 1993. **Proceedings ...** Keele: [s.n.], 1993. p. 249-257.

[PJO93b] PEYTON JONES, S. L. ; WADLER, P. Imperative Functional Programming. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 20., 1993, Charleston. **Proceedings ...** [S.l.:s.n.], 1993.

[PJO96a] PEYTON JONES, S. L.; GORDON, A. D.; FINNE, S. O. Concurrent Haskell. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 23., 1996, St Petersburg Beach.**Proceedings ...** [S.l.]: ACM Press, 1996. p 295-308.

[PJO98] PEYTON JONES, S. L.; MEIJER, E.; LEIJEN, D. Scripting COM Components in Haskell. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 5., 1998. **Proceedings ...** [S.l.]: IEEE, 1998.

[PJO99] PEYTON JONES, S. L. et al. (Eds.). **Haskell 98**: a Non-strict, Purely Functional Language. Cambridge: Microsoft Research, 1999. Disponível em: (http://www.haskell.org/onlinereport/). Acesso em: jan. 2001.

[PLA93] PLASMEIJER, R.; EEKELEN, M. V. **Functional Programming and Parallel Graph Rewriting**. [S.l.]: Addison-Wesley, 1993.

[POI2000] POITON, R. F.; TRINDER, P. W.; LOIDL, H-W. GdH Design and Implementation In: IFL, 2000. **Proceedings ...** Aachen: [s.n.], 2000.

[REI99] REINKE, C. Towards a Haskell/Java Connection.In: IFL, 1998, London. **Proceedings ...** Berlin: Springer-Verlag 1999. p. 200-215.

[SIB99] SIBJORN, F.; LEIJEN, D.; MEIJER, E. Calling Hell from Heaven and Heaven from Hell. In: ICFP, 1999. **Proceedings ...** [S.l.:s.n.], 1999.

[TAN97] TANENBAUM, A. S.; WOODHULL, A. S. **Operating Systems, Design and Implementation**. 2nd ed. [S.l.]: Prentice Hall, 1997.

[THO99] THOMPSON, S. **Haskell, The Craft of Functional Programming**. 2nd ed. [S.l.]: Addison-Wesley, 1999.

[TRI93] TRINDER, P. W. et al. Algorithm + Strategy = Parallelism. **Journal of Functional Programming**, Cambridge, v. 1, n.1, Jan. 1993.

[TRI96] TRINDER, P. W. et al. GUM: a portable implementation of Haskell. In: Programming Language Design and Implementation. PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 1996, Philadelphia. **Proceedings ...** [S.l.:s.n.], 1996.

[TUL97] TULLSEN, M. **Compiling Haskell to Java**. [S.l.]: Yale University, 1997. (Technical Report. YALEU/DCS/RR-1204).

[TUR85] TURNER, D. A. **Miranda**: a Non-Strict Functional Language with Polymorphic Types. Berlin: Springer-Verlag, 1985. p. 1-16 (LNCS 201)

[WAD97] WADLER, P. A HOT opportunity. **Journal of Functional Programming**, Cambridge, v.7, n.2, p.127-128, Mar. 1997.

[WAD98] WADLER, P. Why no one uses functional languages. **SIGPLAN Notices**, [S.l.], v.33, n.8, p. 23-27, Aug. 1998.

[WAK98a] WAKELING, D. A Haskell to Java Virtual Machine Code Compiler. In: IFL 1997, St Andrews. **Proceedings ...** Berlin: Springer-Verlag, 1998. p. 39-52.

[WAK98b] WAKELING, D. Mobile Haskell: compiling Lazy Functional Programs for the Java Virtual Machine. In: PRINCIPLES OF DECLARATIVE PROGRAMMING, PLPI, 1998, Pisa **Proceedings ...** Berlin: Springer-Verlag, 1998. p. 335-352.

[WAK98c] WAKELING, D. Compiling Lazy Functional Programs for the Java Virtual Machine. **Journal of Functional Programming**, Cambridge, v.1, n.1, Jan.1998.

[WAL2001] WALLACE, M. **Calling Haskell from C using GreenCard**. Disponível em: (http://www.cs.york.ac.uk/fpnhc/CcallingHaskell.html) Acesso em: mar. 2001.

[WAT87] WATSON, P; WATSON, I. Evaluating Functional Programs on the FLAGSHIP Machine. In: FPCA, 1987. **Proceedings ...** Berlin: Springer-Verlag, 1987. p. 80-97. (LNCS 274).

[WIN89] WINSTON, P. H.; HORN, B. K. P. **Lisp**. 3rd ed. [S.l.]: Addison-Wesley, 1989.