**Adriano Bonat Alves**

# *Compilando código funcional para bytecodes Java*

Pelotas

Novembro de 2008

**Adriano Bonat Alves**

# *Compilando código funcional para bytecodes Java*

Orientador:
Prof. Dr. Cristiano Damiani Vasconcelos

UNIVERSIDADE FEDERAL DE PELOTAS - UFPEL

Pelotas

Novembro de 2008

# *Resumo*

ALVES, Adriano B. **Compilando código funcional para *bytecodes* Java**. 2008. 73f. Trabalho acadêmico (Graduação) – Bacharelado em Ciência da Computação. Universidade Federal de Pelotas, Pelotas.

Este trabalho propõe o estudo da interoperabilidade e dos problemas envolvidos na tradução entre linguagens de diferentes paradigmas, principalmente entre os paradigmas funcional e orientado a objeto. Para a realização deste estudo, foi desenvolvido um compilador de uma linguagem funcional que gera *bytecodes* Java. A interoperabilidade entre linguagens é de grande interesse, como mostra o sucesso das plataformas .NET e Java, respectivamente da Microsoft e da Sun Microsystems. Um dos motivos da baixa adoção de linguagens funcionais é a falta de uma maneira fácil destas interoperarem com outras linguagens. A máquina virtual Java executa programas binários na forma de *bytecodes* Java. Embora esta máquina virtual tenha sido desenvolvida para a linguagem de programação Java, outras linguagens de programação podem ser executadas sobre esta. Alguns dos motivos do grande interesse por máquinas virtuais são a explosão da web, grandes sistemas desenvolvidos com base em várias linguagens e o suporte a código legado. A máquina virtual resolve estes problemas, pois é uma plataforma onde várias linguagens podem interoperar. A linguagem funcional definida neste trabalho tem sintaxe similar a de Haskell, sua ordem de avaliação é estrita, é estaticamente tipada, possui inferência de tipos e não possui suporte à sobrecarga (*overload*).

Palavras-chave: Sistema de tipos. Linguagens funcionais. Interoperabilidade. Orientação a objeto.

# *Abstract*

ALVES, Adriano B. **Compilando código funcional para *bytecodes* Java**. 2008. 73f. Trabalho acadêmico (Graduação) – Bacharelado em Ciência da Computação. Universidade Federal de Pelotas, Pelotas.

This work proposes the study of interoperability and problems involved in the translation between languages from different paradigms, particularly the functional paradigm and the object oriented. For this study, was developed a compiler of a functional language that generates Java bytecodes. The interoperability between languages is of great interest, as demonstrated by the success of the platforms .NET and Java, respectively of Microsoft and Sun Microsystems. One of the reasons for the low adoption of functional languages is the lack of an easy way these interoperate with other languages. The Java virtual machine runs programs in a binary form called Java bytecodes. Although this virtual machine has been developed for the Java programming language, other programming languages can benefit themselves too. Some of the reasons for the great interest in virtual machines are the explosion of the web, the development of large systems based on several languages and the support for legacy code. The virtual machine solves these problems because it is a platform where different languages can interoperate. The functional language defined in this work is similar to the syntax of Haskell, is strict, statically typed, has type inference and does not have support to overload.

Keywords: Type system. Functional languages. Interoperability. Object Oriented.

# *Lista de Figuras*


2.1 Sistema de tipos de Damas-Milner . . . . . p. 28

2.2 Algoritmo *W* . . . . . p. 29

2.3 Implementação da classe de tipos *Monad* . . . . . p. 34

2.4 Implementação de um mônada para *logs* . . . . . p. 34

2.5 Exemplo de uso do mônada definido na Fig. 2.4 . . . . . p. 35

3.1 Tipos de dados da máquina virtual Java . . . . . p. 39

3.2 Pseudo-código do laço principal do interpretador da JVM . . . . . p. 43

4.1 Exemplo de tipo de dado de enumeração para cores . . . . . p. 57

4.2 Saída do analisador sintático para o exemplo da Fig. 4.1 . . . . . p. 57

4.3 Implementação da função *len* . . . . . p. 58

4.4 Exemplo de saída do analisador sintático para a função *len* . . . . . p. 58

4.5 Um exemplo de implementação da função *map* . . . . . p. 65

4.6 Tradução do tipo lista para classes Java . . . . . p. 66

4.7 Implementação da segunda classe para a tradução de *map* . . . . . p. 67

# *Lista de Tabelas*

3.1 Intervalo dos tipos de dados da máquina virtual Java . . . . . p. 40

3.2 Instruções que manipulam diretamente a pilha de operandos . . . . . p. 44

# *Lista de abreviaturas e siglas*

| | |
|---|---|
| API | *Application Programming Interface* - Interface de Programação de Aplicativos |
| BNF | Backus Naur *Form* - Forma de Backus Naur |
| FFI | *Foreign Function Interface* - Interface de funções externas |
| GC | *Garbage Collector* - Coletor de Lixo |
| GHC | Glasgow Haskell *Compiler* - Compilador Haskell de Glasgow |
| JAR | Java *ARchive* - Arquivo Java |
| JDK | Java *Development Kit* - Kit de desenvolvimento Java |
| JNI | *Java Native Interface* - Interface nativa para Java |
| JVM | *Java Virtual Machine* - Máquina virtual Java |
| LCF | *Logic for Computable Functions* - Lógica para funções computáveis |
| Lisp | *LISt Processor* - Processador de listas |
| ML | *Meta-language* - Meta linguagem |
| opcode | *Operation Code* - Código de instrução |
| PC | *Program Counter* - Contador de programa |
| Yacc | *Yet Another Compiler Compiler* - Ainda outro compilador de compiladores |

# *Sumário*


**1 Introdução** p. 10

1.1 Motivação . . . . . p. 11

1.2 Objetivos . . . . . p. 12

1.3 Organização do Trabalho . . . . . p. 12

**2 Programação Funcional** p. 14

2.1 Cálculo Lambda . . . . . p. 14

2.2 LISP . . . . . p. 16

2.3 ML . . . . . p. 18

2.3.1 Declarações e Tipos de Dados . . . . . p. 20

2.4 Sistema de tipos e Inferência . . . . . p. 26

2.4.1 Segurança e Checagem de Tipos . . . . . p. 26

2.4.2 Inferência de Tipos . . . . . p. 27

2.5 Haskell . . . . . p. 30

2.5.1 Avaliação Preguiçosa . . . . . p. 30

2.5.2 Classes de tipo . . . . . p. 31

2.5.3 Mônadas . . . . . p. 33

2.6 Síntese . . . . . p. 35

**3 Máquina Virtual Java** p. 37

3.1 Características Gerais . . . . . p. 37

3.2 Tipos de Dados . . . . . p. 38

3.3 Áreas de Memória . . . . . p. 39
3.3.1 Área de Métodos . . . . . p. 40
3.3.2 Heap . . . . . p. 41
3.3.3 Pilha da *Thread* . . . . . p. 41
3.3.4 Pilha de Métodos Nativos . . . . . p. 42
3.4 Conjunto de Instruções (*opcodes*) . . . . . p. 43
3.4.1 Instruções de Variáveis Locais e Pilha de Operandos . . . . . p. 43
3.4.2 Instruções Aritméticas . . . . . p. 45
3.4.3 Instruções de Conversão de Tipos . . . . . p. 45
3.4.4 Criação e Manipulação de Objetos . . . . . p. 46
3.4.5 Instruções de Transferência de Controle . . . . . p. 46
3.4.6 Instruções de Chamada e Retorno de Métodos . . . . . p. 47
3.4.7 Instruções de Manipulação de Exceções . . . . . p. 47
3.4.8 Instruções para Sincronização . . . . . p. 47
3.5 Execução . . . . . p. 47
3.6 Especificação dos Arquivos *.class* . . . . . p. 48

**4 Implementação** p. 49
4.1 Analisador Sintático . . . . . p. 49
4.1.1 Parsec . . . . . p. 49
4.1.2 Código Intermediário . . . . . p. 52
4.2 Inferidor de Tipos . . . . . p. 58
4.3 Saída do Compilador (*backend*) . . . . . p. 59
4.3.1 Jasmin . . . . . p. 60
4.3.2 Traduções . . . . . p. 62
4.4 Síntese . . . . . p. 67

**5 Conclusão** p. 68

5.1 Trabalhos Futuros . . . . . p. 69

**Referências Bibliográficas** p. 70

# 1 Introdução

As primeiras linguagens de programação foram desenvolvidas com o simples objetivo de poder controlar o comportamento dos computadores. Prova disto é que estas linguagens ofereciam diretamente, de uma forma não abstrata, as funcionalidades do *hardware* em que iriam ser executadas.

Desta forma, de linguagens de montagem primitivas surgiram várias linguagens de alto nível, começando com FORTRAN (BACKUS, 1981) em 1950. O número destas linguagens cresceu tão rapidamente que no início de 1980, para um melhor estudo, estas já eram agrupadas, devido às suas semelhanças, em famílias de linguagens (HUDAK, 1989).

A família de linguagens de programação funcionais, na qual a computação é o resultado da avaliação de expressões (principalmente a aplicação de funções), tem atraído, atualmente, a atenção de pesquisadores tanto da área acadêmica como da indústria de desenvolvimento de *software*.

Uma das razões dessa atração, principalmente pelas linguagens funcionais puras e de avaliação preguiçosa, é que estas podem usufruir do poder de processamento dos processadores com múltiplos núcleos (processadores *multicore*) (MEIJER, 2008). O fato de serem puras diz respeito à ausência de efeito colateral, ou seja, a avaliação de uma expressão não influi no resultado da avaliação de outra expressão. Juntamente com isso, o fato de ter uma avaliação preguiçosa permite que a avaliação dos parâmetros de uma expressão seja realizado somente quando necessário. Essas características fazem com que programas nestas linguagens sejam muito mais facilmente paralelizáveis do que em linguagens de paradigma imperativo (HUDAK, 1989).

Embora as linguagens funcionais resolvam vários problemas, estas muitas vezes necessitam utilizar a interface de funções externas (*Foreign Function Interface* - FFI) para acessar alguma funcionalidade de que não dispõem, como uma biblioteca de outra linguagem, ou ter acesso direto a serviços do sistema operacional. A importância de boas FFIs é largamente reconhecida atualmente, principalmente frente à tendência de desenvolvimento baseado em componentes escritos em várias linguagens (BENTON; KENNEDY, 1999).

Geralmente as linguagens funcionais possuem alguma maneira de chamar funções externas em C, porém, trabalhar diretamente com uma linguagem de mais baixo nível, que não tenha um sistema de tipos seguro e sem um coletor de lixo nunca irá ser trivial ou elegante (BENTON; KENNEDY, 1999). Um dos motivos da baixa adoção de linguagens funcionais é a falta de uma maneira fácil destas interoperarem com outras linguagens (BENTON; KENNEDY; RUSSELL, 1998).

A interoperabilidade entre linguagens é de grande interesse, como mostra o sucesso das plataformas .NET e Java, respectivamente da Microsoft e da Sun Microsystems. Os projetistas de linguagens de programação demonstram isto através de linguagens como SML♯, Mondrian e Scala (MICROSOFT, 2008b; MICROSOFT, 2008a; ODERSKY, 2008). Todas estas linguagens têm a interoperabilidade com outras linguagens como sua funcionalidade principal (MATTHEWS; FINDLER, 2007). Alguns dos motivos do grande interesse por máquinas virtuais são a explosão da web, grandes sistemas desenvolvidos com base em várias linguagens e o suporte a código legado. A máquina virtual resolve estes problemas, pois é uma plataforma onde várias linguagens podem interoperar.

A possibilidade de escrever algoritmos de forma concisa, ou então facilmente paralelizar a execução destes algoritmos, demonstra alguns dos maiores poderes das linguagens funcionais. Através da execução de uma linguagem funcional sobre a arquitetura de uma máquina virtual poderíamos obter os benefícios da portabilidade, de termos um sistema de tipos comum entre as linguagens, assim como também a segurança proporcionada pela *sandbox*, que protege o sistema hospedeiro durante a execução do programa.

## 1.1 Motivação

Como vimos na seção anterior, é crescente o número de linguagens de programação que baseiam a sua execução sobre máquinas virtuais. Sobre a máquina virtual Java temos como exemplo os projetos para rodar Python (Jython) (PEDRONI; RAPPIN, 2002), Ruby (JRuby) (BINI, 2007), recentemente Scala, e centenas de outras linguagens (TOLKSDORF, 2008).

A máquina virtual Java (LINDHOLM; YELLIN, 1999) executa programas binários na forma de *bytecodes* Java. Embora esta máquina virtual tenha sido desenvolvida para a linguagem de programação Java, outras linguagens de programação podem ser executadas sobre esta.

A plataforma Java foi escolhida por diversas razões, dentre elas:

– há pouco tempo teve seu código-fonte aberto, sendo possível para qualquer pessoa estudá-lo;

– a máquina virtual Java é uma das mais populares;

– a linguagem Java é a mais popular[1] segundo TIOBE (TIOBE, 2008), sendo mais popular que a linguagem C.

O propósito deste trabalho é o desenvolvimento de uma base que possibilite futuras pesquisas sobre a interoperabilidade entre linguagens de diferentes paradigmas, principalmente entre os paradigmas funcional e orientado a objeto. Como parte dessa base, foi desenvolvido um compilador de uma linguagem funcional que gera *bytecodes* Java.

## 1.2 Objetivos

O objetivo deste trabalho é implementar um compilador que permita o estudo dos problemas envolvidos na geração de código funcional para a máquina virtual Java, assim como a interoperabilidade entre as linguagens do paradigma funcional e orientado a objeto. Essa implementação envolve:

– a definição de uma linguagem funcional estrita e sem sobrecarga, baseada na sintaxe de Haskell;

– a construção de um compilador, onde o *frontend* é composto por um analisador sintático e um inferidor de tipos, e cujo *backend* gera *bytecodes* Java;

– o estudo dos possíveis problemas de interoperabilidade entre o paradigma funcional e o orientado a objeto.

Em relação a geração de código, serão abordadas apenas as representações de tipos de dados algébricos e um estudo inicial sobre a tradução de funções.

## 1.3 Organização do Trabalho

Este trabalho, dividido em cinco capítulos, apresenta primeiramente os conceitos estudados durante o seu desenvolvimento, seguido da descrição da ferramenta desenvolvida, os resultados obtidos e as conclusões.

[1] O índice da comunidade de programação TIOBE leva em conta o número de pessoas que utilizam a linguagem, assim como o número de cursos e de parceiros (*third party vendors*).

No capítulo dois, é apresentada uma introdução à programação funcional, suas características, como funciona a inferência de tipos, e também um pouco da evolução das linguagens deste paradigma de desenvolvimento de *software*.

O capítulo três trata da máquina virtual Java, descrevendo suas características, *opcodes*, tipos de dados, modelo de execução e também sobre seus arquivos de *bytecodes* (arquivos *.class*).

No quarto capítulo, é descrita a ferramenta desenvolvida, as bibliotecas utilizadas e as traduções feitas entre as expressões da linguagem funcional deste trabalho para construções na linguagem Java.

No quinto e último capítulo, são apresentadas as conclusões e as sugestões de trabalhos futuros.

# 2 *Programação Funcional*

Programação Funcional é assim chamada porque um programa consiste inteiramente em funções. A própria função principal do programa é definida em termos de outras funções, e assim sucessivamente, até o nível base das funções, que são as primitivas da linguagem (HUGHES, 1989). Ao contrário do paradigma imperativo, linguagens funcionais, em sua forma pura, não suportam mudanças de estado, dando ênfase na aplicação de funções.

Este capítulo mostra as origens da programação funcional com o Cálculo Lambda na seção 2.1, e nas seções seguintes explora as primeiras linguagens funcionais e a evolução delas até a linguagem estado da arte, Haskell, na seção 2.5.

## 2.1 Cálculo Lambda

O cálculo lambda é um modelo matemático, e pode ser pensado como uma linguagem de programação pura, baseada na definição e aplicação de funções, e o seu método de iteração é através da recursão. Esse modelo permite a representação de qualquer algoritmo, e é pura no sentido de que as funções recebem e retornam dados, que podem ser inclusive funções, e não podem ser alterados pela função.

Na área da matemática e da ciência da computação, funções de ordem superior (*high order functions*) são funções que podem receber funções como argumentos, assim como produzir funções como resultado de sua computação.

Por ser um modelo simples, o cálculo lambda permite demonstrar alguns conceitos importantes de linguagens de programação, como por exemplo ligação, escopo, ordem de avaliação, computabilidade, sistemas de tipos, etc.

Existem três tipos de expressões lambda:

– Variáveis: expressas, normalmente, através de identificadores alfanúmericos;

– Aplicações ($e_1\ e_2$): representam a aplicação da expressão $e_1$ para $e_2$;

– Abstrações ($\lambda x.e$): representam a função que retorna o valor $e$ quando recebe o parâmetro formal $x$.

Podemos, por exemplo, definir a função identidade através da seguinte *abstração-*$\lambda$:

$$(\lambda x.x)$$

Quando as abstrações-$\lambda$ encontram-se aninhadas, há a convenção sintática de deixar de usar o ponto para separá-las, assim como de eliminar os parênteses para facilitar o entendimento da expressão. Outras duas convenções dizem respeito quanto à ordem de aplicação das funções, que devem ocorrer da esquerda para a direita, e que o escopo de uma abstração-$\lambda$ se estende o máximo possível para a direita, por exemplo $\lambda x.xy$ deve ser lido como $\lambda x.(xy)$, e não como $(\lambda x.x)y$. Em uma expressão $\lambda x.E$, $E$ é chamado de escopo $x$.

As variáveis em expressões-$\lambda$ podem ser *ligadas* ou *livres*. Uma variável é ligada quando está associada a uma abstração-$\lambda$, e livre caso contrário. Expressões que diferem apenas no nome das variáveis ligadas (ou seja, não há uma diferença semântica) são chamadas de expressões $\alpha$-equivalentes, por exemplo a expressão $(\lambda y.y)$ é $\alpha$-equivalente à função identidade mostrada previamente, logo são expressões sinônimas.

A aplicação de abstrações-$\lambda$ é baseada em substituições, por exemplo:

$$(\lambda x.M)N = [N/x]M$$

significa que $N$ é o parâmetro da abstração-$\lambda$. Assim, todas as ocorrências de $x$ na expressão $M$ serão substituídas pela expressão $N$, e essa substituição pode ser representada por $[N/x]M$. Uma expressão $W$ é dita $\beta$-equivalente a $Y$ se a expressão $W$ for $\alpha$-equivalente a $Y$ após zero ou mais substituições. As seguintes expressões são $\beta$-equivalentes a $x$:

$$(\lambda y.y)x$$
$$(\lambda y.x)x$$
$$(\lambda f.fx)(\lambda y.y)$$
$$x$$

Em cálculo lambda há duas ordens de se avaliar uma expressão: avaliação por valor (*eager evaluation*), onde a redução ocorre no parâmetro antes deste ser aplicado à função, e avaliação

preguiçosa (*lazy evaluation*), onde o parâmetro da função é substituído inteiramente no corpo da função, havendo redução do parâmetro dentro do corpo da função somente se necessário.

A ordem de avaliação de uma expressão é importante, pois pode resultar na expressão na sua forma normal (quando nenhuma substituição pode mais ser aplicada a uma expressão), ou então no não término da computação. A expressão $(\lambda x \lambda y.y)((\lambda z.zz)(\lambda z.zz))(\lambda w.w)$ demonstra a importância da escolha da ordem de avaliação, uma vez que através da ordem de avaliação por valor a computação dessa expressão não terminaria.

> **Teorema de Church e Rosser**: Se *v* é o resultado da avaliação de uma expressão *M* aplicando a ordem de avaliação preguiçosa, então qualquer que seja a ordem aplicada, ou o resultado é *v* ou a avaliação falha (não termina). Se a avaliação de *M* não termina usando a ordem preguiçosa, a avaliação não termina usando qualquer ordem. (CHURCH; ROSSER, 1936)

## 2.2 LISP

LISP ou Lisp (MCCARTHY et al., 1965) (acrônimo para *LISt Processor*) foi a primeira linguagem de programação a possibilitar a definição de expressões sem efeitos colaterais (MITCHELL; APT, 2001). O projeto da linguagem foi publicado por McCarthy em um artigo em 1960 (MCCARTHY, 1960), onde mostrava que com apenas alguns simples operadores e uma notação para funções, era possível construir uma linguagem Turing-completa (LISP, 2008). Sua principal aplicação é na área de pesquisa em inteligência artificial e computação simbólica.

Muitas implementações de Lisp foram desenvolvidas, levando a diversos dialetos da linguagem, como o Maclisp desenvolvido no MIT na década de 1960, e o Scheme, desenvolvido também no MIT na década de 1970 por Guy Steele e Gerald Sussman. Atualmente a forma de Lisp mais usada é o *Common Lisp* (SYSTEMS et al., 1994), que é uma especificação da linguagem publicada pela ANSI, e contém algumas primitivas de orientação a objeto.

Embora Lisp tenha sua inspiração no cálculo lambda, elas possuem algumas diferenças importantes. Listas são o principal tipo de dado de Lisp, enquanto funções são o único tipo de dado no cálculo lambda puro (existe o cálculo lambda enriquecido, que possui algumas construções como de seleção, que tornam o uso de cálculo lambda mais agradável e fácil).

Na especificação original, existia apenas dois tipos de dados: átomos e listas. Átomos podem ser alfanúmericos ou apenas números, e sua diferença consiste em ser imutável e único. As listas em Lisp são simplesmente ligadas, sendo a função *car* usada para retornar o dado do nodo, e $cdr$ para retornar o próximo nodo. A função *cons* é utilizada para construir uma lista, e

existe a lista vazia, denotada por *nil*.

O artigo de McCarthy definia duas sintaxes possíveis para a linguagem: expressões-S (*s-expressions* ou *Symbolic expressions*) e expressões-M (*Meta expressions*), embora a segunda não seja muito usada. Expressões-S representam listas, e misturam códigos e dados em uma representação regular, baseada no uso intensivo de parênteses, que representam listas, e tem como o espaço o separador dos elementos. Desta forma, a linguagem é extremamente flexível, pois funções são declaradas como listas, e assim podem ser processadas uniformemente como se fossem dados, o que dá origem à idéia de *meta-programação*, onde programas podem se modificar.

Lisp, na verdade, não é uma linguagem totalmente pura. Apenas um sub-conjunto de suas funções é puro, o que forma o chamado Lisp puro. São elas:

| | |
|---|---|
| *cons* | constrói uma lista |
| *car* | retorna o dado do nodo |
| *cdr* | retorna o próximo nodo |
| *eq* | retorna *true* se as duas expressões têm o mesmo valor |
| *atom* | retorna *true* se o valor da expressão é atômico |

e algumas funções especiais:

| | |
|---|---|
| *cond* | retorna o valor da primeira expressão que tiver valor diferente de *nil* |
| *lambda* | similar ao cálculo lambda, retorna uma expressão para ser avaliada |
| *define* | associa um átomo a uma expressão |
| *quote* | retorna uma expressão cujo valor é o seu parâmetro |
| *eval* | avalia o seu parâmetro |

As expressões que começam com alguma dessas funções especiais são avaliadas de tal forma que determinadas partes de sua expressão não irão ser avaliadas naquele momento, por exemplo:

$$(cond\ \ (p_1\ \ e_1)\ \ (p_2\ \ e_2)\ \ (p_n\ \ e_n))$$

Na expressão exemplificada, a avaliação dos parâmetros de *cond* vai ocorrer da esquerda para a direita, achando o primeiro $p_i$ com valor diferente de *nil*, retornando então o valor da expressão $e_i$.

$$((lambda\ \ (x)\ \ (+\ \ x\ \ 10))\ \ 8)$$

Nesse exemplo da função *lambda*, os seus parâmetros só serão avaliados quando um valor for aplicado. Se executada, essa expressão possui o valor 18.

```
(define find (lambda (x y)
        (cond ((eq y nil) nil)
              ((eq x (car y)) x)
        (true (find x (cdr y)))
)))
```

Através desse exemplo é mostrado como ocorre a recursão em Lisp, onde associamos ao átomo *find* uma expressão lambda. A função *find* procura um elemento *x* em uma lista *y*, retornando o valor *x* caso o encontre, ou então retorna *nil* caso o elemento *x* não seja encontrado na lista.

## 2.3 ML

A família de linguagens baseada em Algol foi desenvolvida em paralelo com Lisp, e levou ao desenvolvimento de ML e Modula (WIRTH, 1977).

As principais características das linguagens derivadas de Algol são as expressões separadas por ponto e vírgula, a estrutura em blocos, funções e procedimentos, e tipagem estática.

ML foi a primeira linguagem a incluir inferência de tipos polimórficos.

ML é uma linguagem funcional com algumas possibilidades imperativas, sendo por isso considerada uma linguagem funcional impura. Ao mesmo tempo que é possível criar funções através de expressões lambda, passá-las para outras, e retornar funções como resultado de computações, ML também permite a escrita de algoritmos de forma imperativa, com uma sintaxe parecida com a de linguagens que descendem da família Algol. A versão de ML mais utilizada é a *Standard* ML (MILNER; TOFTE; HARPER, 1990).

Enquanto Lisp é uma linguagem dinamicamente tipada, ou seja, os tipos das expressões são resolvidos em tempo de execução, ML é uma linguagem estaticamente tipada, onde as expressões possuem seus tipos resolvidos em tempo de compilação. Seu sistema de tipos foi parte importante do projeto da linguagem, e é frequentemente considerado o mais limpo e expressivo (MITCHELL; APT, 2001).

Até o desenvolvimento do sistema de tipos de ML, linguagens com sistemas de tipos consis-

tentes (*sound type systems*) eram consideradas restritivas. Esse novo sistema é matematicamente preciso, no sentido de que se uma parte do compilador, chamado de verificador de tipos (*type checker*), determina que uma expressão possui um certo tipo, é então garantido que na avaliação daquela expressão o valor resultante terá como tipo o mesmo determinado pelo verificador. Logo, uma das vantagens que linguagens estaticamente tipadas possuem é essa segurança dada pelo verificador de tipos, que possibilita que vários problemas de programação sejam encontrados em tempo de compilação. Por exemplo, se o verificador determinou que uma expressão possui o tipo "ponteiro para string", então é garantido que quando a expressão for avaliada, o valor dessa expressão será um "ponteiro para string", e não um ponteiro para uma string que já foi desalocada da memória, ou que esteja apontando para outro valor que não uma string.

A linguagem foi desenvolvida por Robin Milner e seus colegas na década de 1970 na Universidade de Edinburgh, como parte de um projeto maior chamado de *Logic for Computable Functions* - LCF , cujo objetivo era desenvolver um sistema para provar propriedades de programas funcionais, de uma maneira automática ou semi-automática. Dessa maneira, ML foi desenvolvida como uma meta-linguagem (daí a origem de seu nome, *Meta-language*) para o projeto LCF.

Um conceito fundamental do LCF é o de táticas de prova (*proof tatics*), que consiste em uma função que recebe uma fórmula, e realizando algumas asserções tenta achar uma prova para essa fórmula. Essa função possui três possíveis resultados:

$$tática(fórmula) = \begin{cases} \text{tem sucesso e retorna a prova} \\ \text{procura infinitamente} \\ \text{falha} \end{cases}$$

Uma idéia adotada foi desenvolver um tipo *prova*, que seria retornado pela função *tática*, e distinguiria os resultados onde se consegue uma prova para a fórmula dos que não conseguem. Assim o tipo da função seria:

$$tática\text{: } fórmula \rightarrow prova$$

O problema que surgiu dessa solução foi como retornar as situações de falha pela função. Milner desenvolveu então o primeiro mecanismo de exceções com tipagem segura (*type-safe*). Desse modo, quando for detectado que não há uma prova para a fórmula, a função pode lançar uma exceção.

Com esse mecanismo de exceções em ML, a notação:

$$f\text{: } A \rightarrow B$$

significa $\forall x$ em $A$, se $f(x)$ termina normalmente sem lançar uma exceção, então $f(x)$ está em $B$.

### 2.3.1 Declarações e Tipos de Dados

Em ML as declarações que associam o valor de uma expressão a um identificador são escritas da seguinte maneira:

```
val <identificador> = <expressão>;
```

Assim, se quiséssemos associar o valor 3 ao identificador *y*, faríamos:

```
val y = 3;
```

É importante notar que uma vez associado um valor a um identificador, esse não muda, portanto as declarações de ML introduzem constantes, não variáveis.

Há duas maneiras de se declarar funções, uma com sintaxe uniforme com a declaração de identificadores, e que de certa forma lembra cálculo lambda:

```
val f = fun x => x+1;
```

e outra que não utiliza de atribuição:

```
fun f(x) = x + 1;
```

porém ambas as declarações se comportam da mesma maneira quando executadas. Na realidade o compilador transforma a segunda forma na primeira, sendo assim apenas um "açúcar sintático" [1].

[1] Açúcar sintático (*syntatic sugar*) é um termo introduzido por Peter Landin, que significa adições a sintaxe de uma linguagem de programação que não afetam a sua funcionalidade, apenas tornam o seu uso mais agradável (em inglês *sweeter*) ao programador

**Tipos Básicos**

Os seguintes tipos básicos fazem parte de ML:

– *Unit*

```
() : unit
```

Assim como o *void* da linguagem C (KERNIGHAN; RITCHIE, 1988), é o tipo de retorno de funções que realizam apenas efeitos colaterais.

– *Bool*

```
true : bool
false : bool
```

Possui apenas esses dois valores válidos, sendo que valores *booleanos* estão geralmente associados à estrutura de seleção $if$:

```
if e1 then e2 else e3
```

sendo que a expressão $e_1$ deve ser um *booleano*, e $e_2$ e $e_3$ devem possuir o mesmo tipo, já que o resultado de um $if$ é a execução de $e_2$ ou $e_3$, e este é o tipo da expressão $if$.

Existem também as operações *AND*, *OR* e *NOT*, porém o nome dos dois primeiros são *andalso* e *orelse*, nomes que reforçam qual é a ordem de avaliação. Por exemplo em uma expressão *a andalso b*, se a expressão *a* for $true$ então *b* será avaliado, caso contrário não. Já na expressão *a orelse b*, caso *a* seja $true$ a expressão *b* não será avaliada.

– Inteiros
Valores como $-2,-1,0,1,2$ são inteiros válidos em ML, e possuem as seguintes operações aritméticas triviais disponíveis: $+$, $-$, $*$ e $div$. Esses operadores são usados na forma infixa.

– Strings
São representadas através de uma sequência de caracteres entre aspas duplas.

```
"Olá mundo!"  :  string
```

– Real

O tipo para números de ponto-flutuante em ML é o *real*, e as operações aritméticas $+$, $-$ e $*$ são sobrecarregadas, porém operam apenas com operandos do mesmo tipo. Por exemplo, a soma de um número inteiro 7 e o número $5,2$ é inválida, pois são de tipos diferentes. Para essa soma ocorrer, o número 7 poderia ser convertido para ponto-flutuante através da função *real*, ou então utilizar as funções *ceil* (arredonda para cima) ou $floor$ (arredonda para baixo) para converter o número $5,2$.

– Tuplas

As tuplas podem ser de dois, três, ... *n* tipos distintos:

```
(256, "Maria")
('Nitrogênio, 14.00674, "Não metal")
```

Partes da tupla podem ser acessadas através de funções que iniciam com "#" e são seguidas pela posição da parte desejada, que começa em 1. Por exemplo: $\#1(256, "Maria")$ retornaria o número 256.

– Registros

Assim como os registros em Pascal (JENSEN; WIRTH, 1974), ou *structs* em C, em ML existem os chamados registros, que são parecidos com as tuplas, porém usam apóstrofos ao invés de parênteses e as partes possuem nomes:

```
val pessoa = { Nome = "Alfredo", Idade = 33 }
```

O método para acessar alguma parte do registro é também parecido com o de tuplas. Utiliza-se, por exemplo, $\#Nome(pessoa)$ para acessar o campo correspondente ao nome da *pessoa*.

– Listas

As listas em ML podem ter qualquer tamanho, com a restrição de que todos os elementos da lista possuam o mesmo tipo de dados. A lista vazia em ML, assim como em Lisp, é representada por *nil*, e a função *cons* por dois pontos.

$$3 :: nil$$
$$1 :: 2 :: [3,4,5]$$

O primeiro exemplo constrói a lista $[3]$, enquanto o segundo a lista $[1,2,3,4,5]$.

**Tipos Sinônimos e Algébricos**

Além dos tipos básicos, ML permite a declaração de tipos sinônimos e novos tipos de dados. Tipos sinônimos são construídos com base em tipos básicos, se associando um nome.

```
type Celsius = real;
```

Para declaração de novos tipos de dados, também chamados de tipos algébricos, é usada a seguinte sintaxe:

```
datatype <nome do tipo> = <construtor 1> | <construtor 2> | ...  |
                        | <construtor n>;
```

onde a sintaxe para os construtores tem a forma:

```
<construtor n> = <nome do construtor> |
               | <nome do construtor> of <tipos dos argumentos>
```

Os tipos algébricos podem ser classificados da seguinte forma:

– Enumeração
  São tipos em que os construtores não possuem argumentos, por exemplo:

```
datatype Temperatura = Fria | Quente | Morna;
datatype Estacoes = Inverno | Verao | Primavera | Outono;
```

– Produto
  É a definição de um construtor com parâmetros:

```
datatype Dupla = Par of float * float;
```

– União disjunta
  Utilizados quando não apenas desejamos distinguir entre possíveis valores, mas também armazenar algumas informações importantes ao construtor.

```
datatype Forma = Circulo of float | Retangulo of float * float;
```

  Podemos interpretar o exemplo acima como: *Circulo* constrói uma *Forma* com qualquer $float$.

– Recursivos
  São tipos que utilizam a sua própria definição dentro de seus construtores.

```
datatype ArvoreBin = Folha of int | Nodo of (ArvoreBin * ArvoreBin);
```

  O construtor *Folha* representa uma folha da árvore, enquanto o outro, *Nodo*, representa uma bifurcação, que leva a duas outras árvores.

– Polimórficos
  Quando deseja-se parametrizar o tipo a ser armazenado no tipo algébrico. Poderíamos declarar uma árvore binária que não apenas armazenasse inteiros, mas qualquer outro tipo de valor:

```
datatype 'a ArvoreBin = Folha of 'a | Nodo of (ArvoreBin * ArvoreBin);
```

  Desta forma, o tipo $'a$ poderá assumir qualquer tipo válido no programa.

Os construtores não executam nenhuma computação sobre seus argumentos, mas apenas armazenam os dados, que podem ser acessados posteriormente através de *casamento de padrão* (*pattern matching*).

```
fun valorNaArvore(x, Folha(y)) = x = y
| valorNaArvore(x, Nodo(y,w)) = valorNaArvore(x,y) orelse valorNaArvore(x,w);
```

O casamento de padrão ocorre na ordem de declaração das cláusulas, primeiramente tentando casar o segundo argumento de *valorNaArvore* com uma Folha, disponibilizando o inteiro através do identificador *y*. Caso não case, é tentado o segundo padrão, onde é testado se a *ArvoreBin* passada é uma instância de *Nodo*, e, caso seja, as duas sub-árvores estarão disponíveis em *y* e *w*.

**Referência**

Conforme dito na seção 2.3, ML é uma linguagem funcional impura. O motivo disto é que o sistema de tipos define tipos de dados que fazem referência a uma área na memória, chamada de célula (*reference cell*). Contudo, o sistema de tipos é construído de tal forma que não permite situações comuns em outras linguagens, como C, de ponteiros para áreas de memória que estão desalocadas, ou então ponteiros para um tipo, apontando para dados de outro tipo.

O mecanismo de referência possui 3 operações básicas:

- *Criação de uma referência*: $ref\ v$
  Cria uma célula de referência para o valor *v*, e retorna um tipo de dado $ref\ v$;
- *Acesso ao valor de uma célula de referência*: $!\ v$
  Retorna o valor armazenado na célula de referência *v*, cujo tipo é o de *v*;
- *Alteração do valor de uma célula de referência*: $r := v$
  Coloca o valor *v* na célula de referência *r*, e retorna *unit*.

Pode-se observar que a maneira utilizada por ML para evitar a criação de ponteiros não inicializados é que um ponteiro para ser criado deve apontar para uma célula, que é inicializada com um valor fornecido.

Como ML não fornece funções para descobrir o endereço de um valor, o mecanismo de referências é uma perfeita abstração, que fornece um lugar para se colocar um valor, de qualquer tamanho, e que também lida com as questões de gerenciamento de memória desse espaço. Um exemplo que demonstra essa abstração é:

```
val s1 = ref ‘‘string menor’’;
s1 := ‘‘uma string gigante’’;
```

É responsabilidade do compilador gerenciar o redimensionamento da célula que contém o valor de $s1$. O exemplo acima também demonstra que duas ou mais expressões imperativas podem ser combinadas, apenas utilizando-se ponto-e-vírgula para separá-las.

## 2.4 Sistema de tipos e Inferência

Tipos são uma coleção de entidades que possuem propriedades em comum. São geralmente utilizados para:

- dar nome e organizar conceitos;
- dar certeza de que sequências de bits na memória do computador serão interpretadas consistentemente;
- prover informação para o compilador sobre os dados que estão sendo manipulados pelo programa, possibilitando ao compilador fazer algumas otimizações.

O uso de tipos também tem papel importante na documentação do programa, pois facilita a leitura, o entendimento e a manutenção pelo programador, com a vantagem sobre comentários no programa, pois tipos são checados pelo compilador, enquanto comentários podem estar desatualizados ou escritos de forma errada.

Um sistema de tipos pode ser formalmente descrito através de um conjunto de regras de inferência de tipo para cada uma das expressões válidas na linguagem. É definido um algoritmo que, dados uma expressão e um contexto de tipos, infere o tipo principal dessa expressão. O contexto de tipos contém as variáveis livres da expressão (variáveis usadas, mas não definidas na expressão). O tipo principal de uma expressão é o mais geral possível que essa expressão poderá ter (PIERCE, 2002). Os tipos representados pelo tipo principal são chamados de instâncias desse tipo.

### 2.4.1 Segurança e Checagem de Tipos

Um erro de tipo acontece quando uma expressão utiliza uma entidade de forma errada, por exemplo somar um inteiro com uma string.

Uma linguagem é dita de tipagem segura (*type safe*) quando não há possibilidade de um programa violar o seu sistema de tipos, logo um inteiro possui um tipo, uma função possui um tipo diferente de um inteiro, então um inteiro não poderá ser usado como uma função.

A checagem dos tipos pode ocorrer em tempo de execução (*runtime check*) ou em tempo de compilação, sendo que a primeira introduz uma sobrecarga (*overhead*) no tempo de execução das operações, pois o compilador gera um código que faz algumas assertivas antes de executar a operação, estas assertivas poderiam ser resolvidas em tempo de compilação, evitando que problemas sejam descobertos quando o programa já estiver em ambiente de produção.

Uma combinação das duas maneiras de checagem é empregada em diversas linguagens, como na linguagem Java, que, por exemplo, checa se os tipos estão sendo corretamente usados na compilação, mas verifica problemas com acesso a posições acima do tamanho máximo nas matrizes (*array bound errors*), em tempo de execução.

### 2.4.2 Inferência de Tipos

Inferência de tipos é o processo de determinar os tipos das expressões, baseando-se nos tipos que são conhecidos dos símbolos que encontram-se nelas. Foi proposto para a linguagem ML por Robin Milner, porém sua idéia pode ser aplicada em outras linguagens de programação.

Conforme veremos no algoritmo de inferência, os tipos ainda não resolvidos são representados por variáveis de tipos. ML possui suporte a tipos polimórficos, que são representados por variáveis de tipos.

As expressões válidas no núcleo da linguagem ML, proposta por Damas-Milner (MILNER, 1978; DAMAS; MILNER, 1982), são definidas pela seguinte sintaxe, onde $x$ representa uma variável, elemento de um conjunto predefinido de variáveis:

$$e \ ::= \ x \ \mid \ e \ e' \ \mid \ \lambda x.e \ \mid \ \texttt{let} \ x \ \texttt{=} \ e \ \texttt{in} \ e'$$

Nessa mini-linguagem, uma expressão da forma $\texttt{let} \ x \ \texttt{=} \ e \ \texttt{in} \ e'$ pode introduzir uma variável $x$ de tipo polimórfico, de forma que $x$ possa ser usada na expressão $e'$ em contextos que requerem tipos distintos.

Sendo $\alpha$ uma meta-variável de tipo, as expressões de tipos nessa mini-linguagem são dadas pela seguinte sintaxe:

$$\tau \ ::= \ \alpha \ \mid \ \tau \rightarrow \tau$$
$$\sigma \ ::= \ \tau \ \mid \ \forall \alpha.\sigma$$

$$\Gamma \vdash x : \sigma \qquad (x : \sigma \in \Gamma) \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e' : \sigma'} \qquad (\sigma < \sigma') \qquad \text{(INST)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha.\sigma} \qquad (\alpha \text{ não é livre em } \Gamma) \qquad \text{(GEN)}$$

$$\frac{\Gamma \vdash e : \tau \to \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\ e') : \tau'} \qquad \text{(APPL)}$$

$$\frac{\Gamma_x \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash (\lambda x.e) : \tau' \to \tau} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma_x \cup \{x : \tau'\} \vdash e' : \tau}{\Gamma \vdash (\text{let } x = e \text{ in } e') : \tau} \qquad \text{(LET)}$$

Figura 2.1: Sistema de tipos de Damas-Milner

Os tipos são assim divididos em monomórficos (denotados por variáveis $\tau$, $\tau'$ etc., possivelmente subscritas) e polimórficos (denotados por $\sigma$, $\sigma'$, etc). Tipos polimórficos são definidos por meio do quantificador universal, sendo por isso também chamados de tipos quantificados.

O conjunto de expressões "bem tipadas" é definido pelo sistema de tipos apresentado na Figura 2.1. Fórmulas desse sistema têm a forma $\Gamma \vdash e : \sigma$, significando que a expressão $e$ tem tipo $\sigma$ no contexto de tipos $\Gamma$.

Um contexto de tipos no sistema de Damas-Milner contém apenas uma suposição de tipo para cada variável $x$. $\Gamma_x$ representa o contexto $\Gamma$, mas sem qualquer suposição de tipo para $x$. A relação $\sigma < \sigma'$ indica que o tipo polimórfico $\sigma$ é mais geral que o tipo $\sigma'$.

Um sistema de tipos "declarativo", como o da Figura 2.1, não provê diretamente um método para inferência de tipos, uma vez que pode existir mais de uma regra a ser usada em determinados casos (ou seja, pode existir mais de uma derivação para uma mesma fórmula $\Gamma \vdash e : \sigma$). Isso ocorre, no caso do sistema da Figura 2.1, devido à existência das regras (INST) e (GEN).

Para inferência de tipos neste sistema, é usado um algoritmo atualmente já bastante conhecido, chamado de *Algoritmo W*. Sua definição se baseia no uso de unificações (MITCHELL, 1996).

Uma substituição é uma função de variáveis de tipo em expressões de tipo. Estas funções são comumente representadas como $[\tau_1/\alpha_1 ... \tau_n/\alpha_n]$, ou $[\tau_i/\alpha_i]^{i=1..n}$. Substituições são estendidas de forma natural para homomorfismos sobre termos (VASCONCELLOS, 2004).

$W(\Gamma, x)$ =
  Se $\Gamma(x) = \forall\alpha_1 ... \alpha_2.\tau$ então $(Id, [\beta i/\alpha_i]\tau)$
  senão *Falha*

$W(\Gamma, e\,e')$ =
  let $(S_1, \tau) = W(\Gamma, e)$
    $(S_2, \tau') = W(S_1\Gamma, e')$
    $S$ = *unificar*$(S_2\tau, \tau' \rightarrow \beta)$ onde $\beta$ é livre
  in $(S \circ S_2 \circ S_1, S\beta)$

$W(\Gamma, \lambda x.e)$ =
  let $(S, \tau) = W(\Gamma_x \cup \{x : \beta\}, e)$
  in $(S, S(\beta \rightarrow \tau))$

$W(\Gamma, let\ \ x = e\ \ in\ \ e')$ =
  let $(S_1, \tau) = W(\Gamma, e)$
    $(S_2, \tau') = W(S_1\Gamma_x \cup \{x : fechamento\ \ (S_1\Gamma, \tau)\}, e')$
  in $(S_1 \circ S_2, \tau')$

Figura 2.2: Algoritmo $W$

Escrevemos simplesmente $S\tau_1 = S\tau_2$, em vez de $S(\tau_1) = S(\tau_2)$, e, em geral, adotamos a convenção (usual) de que a aplicação de substituições é associativa, escrevendo, por exemplo, $SS'S''\alpha$ em vez de $(S \circ (S' \circ S''))(\alpha)$, onde $\circ$ é o operador de composição de funções.

Dois tipos $\tau_1$ e $\tau_2$ são ditos unificáveis quando existe uma substituição $S$ tal que $S(\tau_1) = S(\tau_2)$. Nesse caso, a substituição $S$ é chamada de *unificador* dos tipos $\tau_1$ e $\tau_2$. Um unificador $S_g$ é chamado de *unificador mais geral* se, para qualquer outro unificador $S$ existe uma substituição $S'$ tal que $S' \circ S_g = S$. O *algoritmo W*, como apresentado por Damas e Milner (MILNER, 1978), tem como entrada um par com um contexto de tipos $\Gamma$ e uma expressão, e retorna uma substituição e o tipo principal da expressão. Caso a expressão não tenha tipo principal, é indicada a ocorrência de um erro. O algoritmo é apresentado na Figura 2.2, sendo que *unificar*$(\tau_1, \tau_2)$ representa o unificador mais geral para o par de expressões de tipo, e o fechamento de um tipo (quantificação de suas variáveis de tipo) é definido como:

$$fechamento(\Gamma, \tau) = \forall\alpha_1 ... \alpha_n.\tau$$

onde $\alpha_1 ... \alpha_n$ são variáveis de tipo que ocorrem em $\tau$, mas não em $\Gamma$.

Robinson (ROBINSON, 1965) apresentou pela primeira vez um algoritmo que obtém o uni-

ficador mais geral para dois tipos ou então retorna um erro, caso os tipos não sejam unificáveis.

## 2.5 Haskell

Durante muito tempo pesquisadores de linguagens funcionais projetavam suas próprias linguagens para explorar novas idéias, tornando muito difícil a interação com outros pesquisadores. Diante da necessidade de uma linguagem em comum, esses pesquisadores formaram um comitê para a definição de uma linguagem de experimentação/investigação, criando assim a linguagem Haskell (JONES, 2002; HUDAK et al., 2007).

Haskell é uma linguagem com avaliação preguiçosa e pura, ou seja, diferentemente de ML, não possui efeitos colaterais ou a presença de funcionalidades imperativas. O processo de definição da linguagem foi altamente baseado na linguagem Miranda (TURNER, 1987), por isso ambas possuem muitas similaridades.

A primeira versão de Haskell ("Haskell 1.0") foi definida em 1990. Os esforços do comitê resultaram em uma série de definições da linguagem, que no final de 1997 deram origem ao Haskell 98, uma especificação com a intenção de ser uma versão estável, mínima e portável da linguagem, que acompanhasse uma biblioteca padrão (chamada de *Prelude*) para ensino, e também fosse uma base para futuras extensões. Haskell 98 foi revisada em 2002, e no início de 2006 começou o processo para definição da próxima versão de Haskell, chamada de Haskell′ (Haskell *Prime*).

Um programa em Haskell é um *conjunto* de equações que definem funções e tipos algébricos de dados. É importante frisar a palavra *conjunto*, pois a ordem das equações nos programas é, em geral, irrelevante, e não há a necessidade de se declarar a entidade antes de seu uso.

### 2.5.1 Avaliação Preguiçosa

Sendo Haskell uma linguagem funcional com avaliação preguiçosa, é possível declarar uma função que retorna uma lista, como no exemplo, infinita de números pares:

```
numerosPares :: [Integer]
numerosPares = filter (\n → n `mod` 2 == 0) [2..]
```

Nesse exemplo, a primeira linha é chamada de assinatura da função. Ela explicita o tipo da função ao compilador, e este depois compara o tipo que ele inferiu com o explicitado, alertando

o programador caso o tipo explicitado seja mais geral que o inferido pelo compilador. O tipo da função *numerosPares* é uma lista de inteiros. A segunda linha define a computação da função. É utilizada a função $filter$, que tem assinatura $(a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$. A letra $a$ representa um tipo polimórfico, já que em Haskell estes são representados por letras a,b,..., etc. Dessa forma, a função $filter$ recebe, como primeiro parâmetro, uma função que filtra os elementos passados no segundo parâmetro, retornando todos os elementos em que a função de filtragem retornou $True$. Como função de filtragem foi utilizada a sintaxe para expressões lambda, que testa através de *mod* se um número $n$ é par. No segundo parâmetro para $filter$, a lista de elementos a serem filtrados é uma lista infinita de elementos que começa em 2 ([2..]). Esse exemplo não poderia ocorrer em uma linguagem estrita, pois ficaria para sempre gerando os números da sequência do segundo parâmetro.

### 2.5.2 Classes de tipo

Uma das inovações que foram propostas no projeto de Haskell que a torna uma linguagem distinta em relação a outras linguagens funcionais é o conceito de classes de tipo (*type classes*). Este foi proposto por Wadler e Blott (WADLER; BLOTT, 1989) como solução para um problema relativamente pequeno, qual seja, o de tratar a igualdade e a sobrecarga de operadores númericos. A solução com o tempo foi generalizada para diversas formas, como é apresentado no artigo "Type classes: exploring the design space" (JONES; MEIJER, 1997).

Classes de tipo permitem sobrecarga *ad-hoc* através da adição de restrições nas variáveis de tipos em tipos parametricamente polimórficos. Essa restrição tipicamente envolve uma classe de tipo $T$ e uma variável de tipo $a$, significando que $a$ somente poderá ser instanciada para um tipo cujos membros suportem as operações sobrecarregadas associadas com $T$. Graças a classe de tipos há uma maneira uniforme de se tratar sobrecarga em Haskell.

O padrão Haskell 98 define diversos tipos de classe, dentre eles: igualdade ($Eq$), conversão de/para string (*Read* e *Show* respectivamente), enumerações ($Enum$), operações númericas ($Num$, $Real$, $Integral$, $Fractional$, $Floating$, $RealFrac$ e $RealFloat$), e indexação de matrizes ($Ix$). Analisando a classe de igualdade:

```
class Eq a where
      (==) ::  a → a →Bool
      (/=) ::  a → a →Bool
```

Para um tipo de dado ser considerado uma instância de $Eq$, deve implementar todos os

métodos definidos na classe. Considerando a implementação padrão de igualdade definida no *Prelude*:

```
class Eq a where
      (==), (/=):: a → a → Bool
      x == y = not (x /= y)
      x /= y = not (x == y)
```

Assim, o programador poderá implementar apenas uma das funções, pois a definição de uma está relacionada com a da outra, por exemplo:

```
data Pessoa = Pessoa { nome :: String, rg :: Int }

instance Eq Pessoa where
        p1 == p2 = rg p1 == rg p2
```

No exemplo acima, sobrecarregamos o operador de igualdade (==) para o tipo de dados *Pessoa*, declarando que duas pessoas são iguais se possuírem o mesmo número de identidade. Esse exemplo também mostra a declaração de tipos algébricos com declaração de campos, chamados de registros (*records*) em outras linguagens. Para cada nome de campo (*nome*, *rg*) no registro o compilador automaticamente gera uma função que extrai a informação armazenada nesse campo.

A linguagem oferece uma alternativa à igualdade declarativa, disponibilizando assim uma igualdade estrutural automática para qualquer tipo de dado declarado. Essa igualdade estrutural é ativada adicionando-se a palavra-chave “deriving” na declaração do tipo algébrico:

```
data Pessoa = Pessoa { nome :: String, rg :: Int }
    deriving Eq
```

Dessa forma, duas pessoas serão iguais apenas se o conteúdo de suas estruturas forem iguais. Esta funcionalidade não está disponível para classes declaradas pelo programador.

### 2.5.3 Mônadas

Operações de entrada e saída sempre foram um ponto delicado em linguagens funcionais, pois o objetivo destas operações é produzir efeitos colaterais. Tomando como exemplo a operação de entrada de dados pelo teclado, cada chamada a essa operação poderá retornar valores diferentes, o que não se encaixa com o conceito de função em uma linguagem funcional pura, onde o resultado de uma função, com os mesmos parâmetros, deverá ser sempre o mesmo. Porém, por outro lado, não há sentido em existir um programa sem que o mesmo se comunique com o "mundo externo".

Enquanto esse problema continuava sem solução, em 1989, Eugenio Moggi publicou um artigo em que utilizava um conceito da Teoria das Categorias, chamado de Mônadas, para descrever funcionalidades de linguagens de programação (MOGGI, 1989). Moggi utilizava mônadas para modularizar a estrutura da semântica denotacional, sistematizando o tratamento de diversas funcionalidades, como estados e exceções. Philip Wadler reconheceu que a técnica utilizada por Moggi poderia ser utilizada para estruturar outros programas funcionais (WADLER, 1990), e assim sugeriu a sua introdução em Haskell, sendo então, a primeira linguagem funcional pura com uma maneira de simular computações imperativas.

Um mônada é uma tripla ($M$, *unit*, *bind*) contendo um construtor de tipo $M$ e duas funções polimórficas.

$$\begin{aligned} &\mathit{unit} \;::\; a \rightarrow M\ a \\ &\mathit{bind} \;::\; M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b \end{aligned}$$

Lê-se $a \rightarrow M\ b$ como uma função que recebe um parâmetro de tipo $a$ e tem como resultado um valor de tipo $b$, podendo adicionalmente causar algum efeito capturado por $M$. Este efeito pode ser: agir sobre um estado, gerar alguma saída, lançar alguma exceção, ou qualquer outra coisa que o programador desejar.

A função *unit* serve para transformar um valor de tipo $a$ em uma computação que retorne esse valor. Para aplicarmos uma função $a \rightarrow M\ b$ em uma computação $M\ a$ usamos a função *bind*.

$$m\ `\mathit{bind}`\ \lambda a.\quad n$$

O exemplo acima pode ser lido como: realize a computação *m*, associe (*bind*) o seu resultado a variável *a*, e então realize a computação *n*. Com esta função, torna-se possível o encadeamento de ações imperativas.

```
class Monad m where
      (>>=) :: m a → (a → m b) → m b
      (>>) :: m a → m b → m b
      return :: a → m a
      fail :: String → m a

      m >> k = m >>= \_ → k
      fail s = error s
```

Figura 2.3: Implementação da classe de tipos *Monad*

Vistos os conceitos fundamentais, em Haskell as funções *unit* e *bind* são conhecidas por *return* e pela função infixa >>=, e para poderem ser sobrecarregadas estão declaradas na classe de tipos *Monad* na biblioteca padrão *Prelude*. Sua implementação é mostrada na Fig. 2.3.

```
type Logs = [String]
data Logger a = Logger (a, Logs)

instance Monad Logger where
         return x = Logger (x, logVazio)
         m >= f = Logger (valorFinal, log1 ++ log2)
           where Logger (valorIntermediario, log1) = m
                 Logger (valorFinal, log2) = f valorIntermediario

logVazio :: [a]
logVazio = []

record :: String → Logger ()
record s = Logger ((), [s])

runLogger :: Logger a → (a, Logs)
runLogger (Logger x) = x
```

Figura 2.4: Implementação de um mônada para *logs*

Para se declarar um construtor de tipo *M* um mônada, é necessário apenas se sobrescrever as funções *return* e >>=, visto que as duas outras funções possuem implementações padrão. A

função $>>$ é definida em termo da função $>>=$, e sua diferença é que o resultado da primeira computação não é utilizado pela segunda.

Algo muito comum em programas imperativos é a utilização de funções que registram as operações executadas, os chamados *logs*. No paradigma imperativo basta colocar as chamadas às funções de registro nos locais em que se deseja obter mais informações em tempo de execução, porém no paradigma funcional, isso é visto como um efeito colateral. Na Fig. 2.4 é implementado um mônada para fazer *logs*, e o seu uso é exemplificado na Fig. 2.5 em um programa que calcula expressões aritméticas.

```
data Termo = Const Int | Soma Termo Termo

calc ::  Termo -> Logger Int
calc (Const x) =
    do record ("valor de x eh "++ (show x))
       return x

calc (Soma x y) =
    do vx <- calc x
       vy <- calc y
       record ("A soma eh:  "++ (show (vx+vy)))
       return (vx+vy)
```

Figura 2.5: Exemplo de uso do mônada definido na Fig. 2.4

Notamos que o uso de mônadas é muito simples, e Haskell nos provê a sintaxe com "*do*" para encadear ações, o que não passa de um açúcar sintático que depois é transformado para o operador $>>=$. Um exemplo de uso e do resultado obtido no *console* interativo do compilador é:

```
Main> runLogger (calc (Soma (Const 37) (Const 5)))
(42,["valor de x eh 37","valor de x eh 5","A soma eh:  42"])
```

## 2.6 Síntese

A fim de facilitar o entendimento deste trabalho, este capítulo introduziu a história e a base teórica do mundo das linguagens funcionais. Começamos o capítulo explicando a origem das

linguagens funcionais através do estudo do Cálculo Lambda. Na seção sobre Lisp, falamos sobre a primeira linguagem funcional desenvolvida. Em seguida vimos a linguagem ML, que foi a primeira linguagem com inferência de tipos e cujo sistema de tipos é até hoje considerado um dos mais limpos e expressivos. Falando em inferência de tipos, a seção após ML foi inteiramente sobre sistemas de tipos e como a inferência de tipos acontece. Ao final deste capítulo, sobre programação funcional, não poderíamos deixar de falar sobre o estado da arte e abordar a linguagem Haskell, que tem sido a principal ferramenta no estudo do paradigma funcional.

Desta forma, podemos seguir para o próximo capítulo, que é dedicado à máquina virtual Java.

# 3 *Máquina Virtual Java*

A máquina virtual Java (*Java Virtual Machine* - JVM ) é o principal componente do ambiente Java, sendo o responsável pelo pequeno tamanho de seu código compilado, e também da independência de sistema operacional e de hardware (LINDHOLM; YELLIN, 1999). Existem versões da JVM para diversas arquiteturas e sistemas operacionais.

Assim como um computador real, a JVM possui um conjunto de instruções (*opcodes*) e manipula áreas na memória durante o tempo de execução. Portanto, ela não tem conhecimento algum sobre a linguagem de programação Java (ARNOLD; GOSLING, 2000) (embora tenha sido projetada para ela), conhece apenas o formato binário dos seus arquivos de entrada, os arquivos *.class*. Desta forma, qualquer linguagem que tenha funcionalidades capazes de serem expressas no formato dos arquivos *.class* poderá ser executada na JVM. Várias linguagens de diversos paradigmas podem ser encontrados na lista de Robert Tolksdorf (TOLKSDORF, 2008).

## 3.1 Características Gerais

A arquitetura da JVM é baseada em pilhas (*stacks*), onde cada linha de execução (*thread*) possui a sua.

A segurança na máquina virtual é garantida pela compilação onde os tipos corretos são determinados, como também em tempo de execução pelo gerenciador de segurança.

A JVM restringe as operações utilizando quatro mecanismos interligados: o carregador de classes, o verificador de *bytecode*, as verificações realizadas em tempo de execução e o gerenciador de segurança. As três primeiras são relacionadas com a garantia dos tipos estarem corretos (*type correctness*) como também com a integridade do sistema de *runtime*. O gerenciador de segurança é utilizado para controlar o acesso a determinadas funções no sistema. Essas quatro técnicas utilizadas para restringir a execução de um programa na JVM são coletivamente chamadas de Java *sandbox*.

Para possibilitar que programas acessem as funcionalidades do sistema operacional hospe-

deiro, existem métodos que podem ser marcados como nativos. Métodos não-Java ou nativos são métodos implementados em linguagens como C, C++, ou linguagem de montagem, e então compilados para um determinado processador alvo. A desvantagem é que o programa com métodos nativos perde a independência do sistema hospedeiro que a JVM proporciona, assim como do *hardware* da máquina. Uma das maneiras mais conhecidas para se implementar métodos nativos é através da *Java Native Interface* - JNI (VENNERS, 1996).

O ambiente Java é composto não apenas pela máquina virtual e pela linguagem de programação Java, mas também pelo formato binário dos arquivos *.class* e por um conjunto de bibliotecas padrão que sempre acompanham a máquina virtual. Essas bibliotecas compõem a chamada *Application Programming Interface* - API Java e, devido ao fato de oferecerem diversas facilidades como classes para acesso à internet, conteúdo multimídia, componentes para interfaces gráficas, etc., são um fator importante na grande adoção da plataforma Java.

Outra razão que contribui para a segurança dos programas na JVM é que não há uma maneira de diretamente se manipular dados na memória, tampouco de se liberar memória alocada explicitamente. O mecanismo de *garbage collector* - GC oferecido pela máquina virtual é responsável pela liberação da memória de objetos que não são mais referenciados no programa, evitando vários problemas que ocorrem em linguagens de programação como C e C++.

## 3.2 Tipos de Dados

A computação na JVM ocorre através de operações sobre tipos de dados, ambos definidos na especificação. Os tipos de dados podem ser divididos em tipos primitivos e tipos referência (*reference*). Variáveis de tipos primitivos guardam valores primitivos. Contudo, variáveis de tipo referência têm como valor uma referência para objetos, mas não são objetos por si só. Os tipos disponíveis na JVM podem ser vistos na Figura 3.1.

Todos os tipos primitivos da linguagem Java são tipos primitivos na JVM. Embora o tipo *boolean* seja dito primitivo, o conjunto de instruções tem um suporte limitado a ele. Quando um compilador traduz código Java para *bytecodes*, ele utiliza *int*s e *byte*s para representar dados *boolean*. Na máquina virtual $false$ é representado pelo inteiro zero, e $true$ por qualquer inteiro diferente de zero. Inteiros também são utilizados em operações envolvendo o tipo *boolean*. Matrizes de *boolean* são representadas como matrizes de *byte*.

Assim como na linguagem Java, os tipos primitivos da JVM possuem o mesmo intervalo de valores, independentemente de plataforma operacional ou do *hardware* sendo utilizado. Embora os intervalos de valores sejam especificados, os tamanhos de cada tipo de dado não o são,

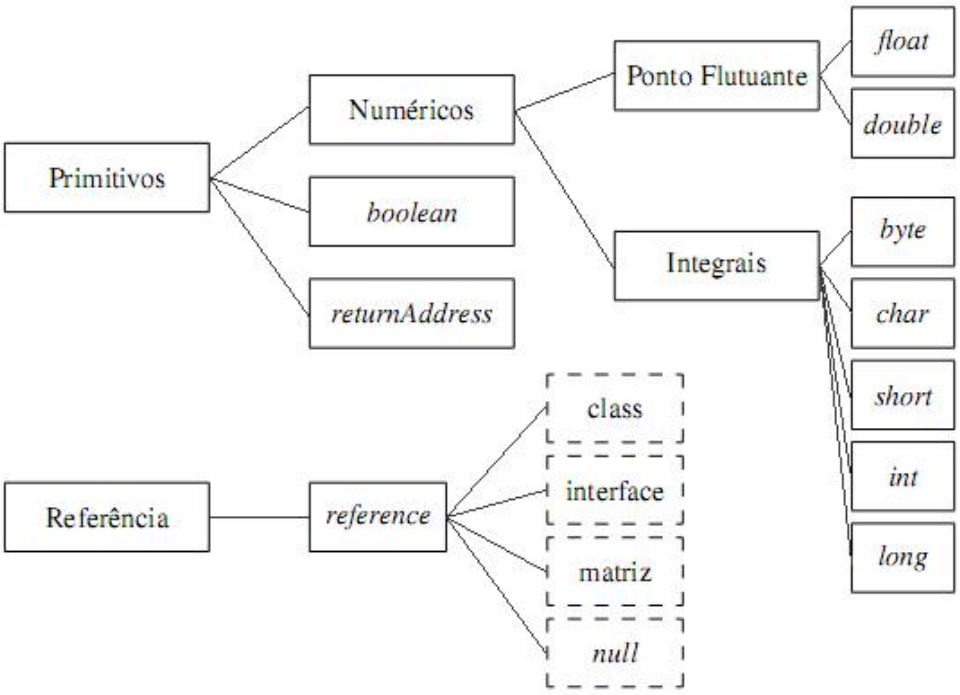


Figura 3.1: Tipos de dados da máquina virtual Java

ficando a critério de quem implementa a máquina virtual.

A JVM trabalha com um tipo primitivo que não é disponibilizado ao programador da linguagem Java, o tipo *returnAddress*. Este tipo é utilizado na linguagem, na implementação de cláusulas $finally$.

O tipo referência da JVM é chamado de $reference$. Valores deste tipo podem ser: de classe (*class type*), de interface (*interface type*) e de matriz (*array type*). Todos possuem como valores referências para objetos criados dinamicamente. Um outro valor de $reference$ é *null*, o qual indica que a variável não referencia nenhum objeto.

Os intervalos dos tipos de dado da JVM são mostrados na tab. 3.1.

## 3.3 Áreas de Memória

Na especificação da JVM são definidas várias áreas de memória. Algumas destas são alocadas quando a máquina virtual é inicializada e desalocadas quando a JVM termina a sua execução. Outras são alocadas juntamente na criação da *thread*, e desalocadas com o término de sua execução.

A JVM suporta a execução de várias *threads*, onde cada uma possui um registrador *program counter* - PC . Este registrador contém o endereço da instrução no método sendo executado no

Tabela 3.1: Intervalo dos tipos de dados da máquina virtual Java

| **Tipo** | **Intervalo** |
|---|---|
| *byte* | inteiro de 8-bits com sinal ($-2^7$ até $2^7-1$, inclusive) |
| *short* | inteiro de 16-bits com sinal ($-2^{15}$ até $2^{15}-1$, inclusive) |
| *int* | inteiro de 32-bits com sinal ($-2^{31}$ até $2^{31}-1$, inclusive) |
| *long* | inteiro de 64-bits com sinal ($-2^{63}$ até $2^{63}-1$, inclusive) |
| *char* | caractere Unicode sem sinal de 16-bits (0 até $2^{16}-1$, inclusive) |
| *float* | ponto flutuante com precisão simples de 32-bits no padrão IEEE 754 |
| *double* | ponto flutuante com precisão dupla de 64-bits no padrão IEEE 754 |
| *returnAddress* | endereço de um *opcode* dentro do mesmo método |
| *reference* | referência para um objeto na *heap*, ou *null* |

momento pela *thread*, porém, se o método for nativo, o conteúdo do registrador é indefinido.

Veremos em mais detalhes, nas próximas subseções, as principais áreas de memória da JVM.

### 3.3.1 Área de Métodos

Dentro de uma instância da JVM, as informações sobre os tipos carregados ficam armazenados em um local na memória chamado de área de métodos. Quando a máquina virtual carrega algum tipo, um carregador de classes (*class loader*) localiza e carrega a classe, para que então algumas informações sejam extraídas. Estas informações extraídas são guardadas na área de métodos, mas a forma em que são armazenadas fica a critério da implementação.

Durante a execução da aplicação, a JVM irá fazer buscas nessas informações de tipos, e, por isso, essas informações devem estar armazenadas de uma forma otimizada para que as buscas sejam rápidas.

Como esta área de memória é compartilhada entre todas as *threads* em execução pela JVM, o acesso às estruturas de dados da área de métodos deve ser projetado para ser *thread-safe*. Se duas *threads* tentam utilizar uma classe que não está carregada, então a primeira que realizou a solicitação terá a prioridade de carregá-la, enquanto a outra espera.

O tamanho da área de memória não precisa ser fixo. A máquina virtual pode expandir e contrair a área de métodos da maneira que a aplicação necessitar. Ainda com relação à memória da área de métodos, esta pode ser não continua (LINDHOLM; YELLIN, 1999).

Durante a execução do programa, classes podem não ser mais referenciadas. Assim, as informações armazenadas na área de métodos podem ser liberadas por um *garbage collector*,

mantendo no mínimo possível o tamanho ocupado.

Para cada tipo que a JVM carrega, uma tabela de símbolos (*constant pool*) deve ser armazenada. Esta tabela é um conjunto de constantes que são usados pelo tipo, incluindo literais e referências simbólicas para tipos, campos e métodos. As entradas nessa tabela são acessadas através de um índice, assim como os elementos de uma matriz (VENNERS, 1996).

### 3.3.2 Heap

A *heap* é a área de dados onde as instâncias de classes e as matrizes são alocadas.

Assim como a área de métodos, a *heap*: é compartilhada entre todas as *threads* da JVM, bem como é criada na inicialização da máquina virtual, podendo ter um tamanho fixo ou ser expandida/contraída sob demanda, não precisando a sua área de memória ser continua.

Embora a JVM tenha uma instrução para alocar memória na *heap* para um novo objeto, não há uma instrução que libera a memória alocada por este objeto. Fica sob responsabilidade da implementação da JVM a liberação da memória ocupada por objetos que não são mais referenciados na aplicação, sendo que, normalmente, a máquina virtual utiliza algum algoritmo de *garbage collection* para a liberação da memória.

### 3.3.3 Pilha da *Thread*

Em conjunto com a criação de uma *thread* é criada uma pilha (JVM *stack*), que contém páginas (*frames*). As páginas dessa pilha atuam de maneira similar aos registros de ativação de linguagens, como por exemplo a linguagem C. A função da página é armazenar os parâmetros, as variáveis locais, a pilha de operandos e alguns dados próprios.

As variáveis locais na página estão organizadas em uma matriz de palavras (*words*), onde o primeiro índice é zero. Palavra é a unidade básica de tamanho para os tipos de dados na JVM. Pela especificação da JVM, uma palavra deve ter o tamanho necessário para armazenar um valor de tipo *byte*, *short*, *int*, *char*, $float$, *returnAddress*, ou $reference$. Duas palavras devem comportar um *long* ou $double$. O tamanho de uma palavra geralmente é definido como sendo o tamanho de um ponteiro nativo na plataforma do hospedeiro. Valores de tipo *byte*, *short*, e *char* são convertidos para *int* antes de serem colocadas nas variáveis locais. Valores de tipo *long* e $double$ ocupam dois lugares consecutivos na matriz.

Assim como as variáveis locais, a pilha de operandos na página é uma matriz de palavras. Porém, a pilha de operandos não é acessada através de índices, mas, através de operações de

empilhar e desempilhar.

Alguns dados próprios na página da pilha da JVM são utilizados para ajudar no acesso a dados que estão na tabela de símbolos, bem como para retornar o resultado do método a quem o chamou, ou, em casos de término por exceção, guardar informações sobre a exceção que ocorreu.

A máquina virtual executa apenas duas operações na pilha: empilha e desempilha páginas. Quando um método é chamado, é criada e empilhada uma nova página na pilha, e esta torna-se a página atual. Um método pode terminar de forma normal, quando termina a sua computação, ou de forma repentina, com o lançamento de uma exceção. De ambas maneiras, a máquina virtual desempilha a página do método sendo executado, e descarta-o. Então, a página do método anterior, que está no topo da pilha, passa a ser a página atual.

Todos os dados na pilha da *thread* são privados àquela *thread*. Desta forma, o acesso a variáveis locais nos métodos não precisa ser sincronizado, pois as variáveis locais estarão alocadas em uma página na pilha da *thread* que chamou o método.

Assim como a área de métodos e a *heap*, a área de memória em que está a pilha ou as páginas não precisa ser continua.

### 3.3.4 Pilha de Métodos Nativos

Quando uma *thread* chama um método nativo, surge um novo mundo no qual as estruturas e as restrições de segurança da JVM não limitam mais o programador. Um método nativo pode acessar as áreas de memória discutidas nas seções anteriores (dependendo da interface de métodos nativos que o implementador oferecer). O programador poderá utilizar os registradores nativos do processador, alocar memória diretamente do sistema hospedeiro, etc.

Qualquer interface de métodos nativos irá usar algum tipo de pilha desses métodos. Quando uma *thread* chama um método, a máquina virtual cria uma nova página e empilha na pilha da JVM. Entretanto, quando uma *thread* chama um método nativo, a pilha da JVM não é utilizada. Ao invés de empilhar uma nova página, a JVM irá simplesmente ligar dinamicamente o módulo que oferece o método, e chamá-lo.

Assim como em outras áreas de memória, a memória ocupada pelas pilhas de métodos nativos não precisa ser de um tamanho fixo, podendo expandir/contrair de acordo com a necessidade da aplicação.

## 3.4 Conjunto de Instruções (*opcodes*)

Cada instrução da JVM consiste em um código de operação de um byte (justificando o nome de *bytecode*), podendo necessitar de zero ou mais operandos. Quando operandos são necessários, após o byte da instrução seguem um ou mais bytes que podem representar o índice de uma variável na matriz de variáveis locais, um valor imediato ou uma referência à tabela de símbolos da classe (SILVA, 2003). Um pseudo-código do laço principal do interpretador da JVM, ignorando exceções, é mostrado na Figura 3.2.

```
do {
    busca um opcode;
    if (opcode exige operandos) busca operandos;
    executa a ação do opcode
} while (tem mais a fazer);
```

Figura 3.2: Pseudo-código do laço principal do interpretador da JVM

Na maioria das instruções com tipos, o tipo ao qual a instrução se aplica é explicito no mnemônico do *opcode* pelas letras i, l, s, b, c, f, d, a, que correspondem, respectivamente, aos tipos *int*, *long*, *short*, *byte*, *char*, *float*, *double* e *reference*. Estas letras são utilizadas por toda a JVM, e são chamadas de descritores de tipos. Instruções onde o tipo não é ambíguo não possuem uma letra descrevendo o tipo no seu mnemônico. Por exemplo, *arraylength* sempre tem como operando um objeto que é uma matriz. Algumas instruções, como *goto*, possuem seus operandos sem tipo.

Nas subseções a seguir veremos as principais categorias de instruções que a JVM provê.

### 3.4.1 Instruções de Variáveis Locais e Pilha de Operandos

Como a JVM é uma máquina baseada em pilhas, quase todas as suas instruções são relacionadas à pilha de operandos. A maioria das instruções empilham e desempilham valores, ou executam ambas operações.

As instruções que transferem valores entre a pilha de operandos e as variáveis locais são:

– Carregam uma variável local na pilha de operandos: *iload*, *iload_<n>*, *lload*, *lload_<n>*, *fload*, *fload_<n>*, *dload*, *dload_<n>*, *aload*, *aload_<n>*;

– Armazenam um valor da pilha de operandos em uma variável local: *istore*, *istore_<n>*, *lstore*, *lstore_<n>*, *fstore*, *fstore_<n>*, *dstore*, *dstore_<n>*, *astore*, *astore_<n>*;

– Carregam uma constante na pilha de operandos: *bipush*, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *iconst_<i>*, *lconst_<l>*, *fconst_<f>*, *dconst_<d>*;

– Ganha acesso a mais variáveis locais usando um índice maior: *wide*.

As instruções acima que terminam com "<n>" indicam instruções que oferecem operandos implícitos, não sendo necessário que o operando seja buscado ou armazenado. Por exemplo, *iload_0* empilha a variável local que está no índice zero na pilha de operandos, o que é equivalente a instrução *iload 0*. Da mesma forma, istore_0 é equivalente a *store 0*, e ambos armazenam na variável local de índice zero o valor que está no topo da pilha de operandos.

Existem três maneiras de se empilhar constantes: o valor da constante estar implícito no *opcode* (*iconst_1* empilha a constante um), o valor da constante ser o operando do *opcode* (*iconst 1*), ou então a constante vir da tabela de símbolos (*ldc*). Como em alguns algoritmos é comum inicializarmos alguma variável com o valor $-1$, os projetistas da JVM incluíram o *opcode iconst_m1*.

O índice para endereçar variáveis locais é de 8-bits, o que limita o número máximo dessas para apenas 256. A instrução *wide* pode estender o índice com outro de 8-bits, elevando o número máximo de variáveis locais para 65536. Este *opcode* precede à instrução que deseja acessar alguma variável local acima do limite de 256.

Tabela 3.2: Instruções que manipulam diretamente a pilha de operandos

| **Instrução** | **Descrição** |
|---|---|
| *nop* | não executa nada |
| *pop* | desempilha a palavra do topo da pilha de operandos |
| *pop2* | desempilha as duas palavras do topo da pilha de operandos |
| *swap* | troca as duas palavras no topo da pilha |
| *dup* | duplica o operando no topo da pilha |
| *dup2* | duplica os dois operandos no topo da pilha |

Embora a maioria das instruções na JVM opere sobre um tipo, algumas instruções manipulam a pilha de operandos de forma independente de tipo. Estas instruções são mostradas na tab. 3.2.

### 3.4.2 Instruções Aritméticas

As instruções aritméticas computam um resultado que é tipicamente uma função de dois valores que estão na pilha de operandos, empilhando o resultado nesta. Não há suporte direto para operações aritméticas envolvendo valores de tipos *byte*, *short*, *char* e *boolean*. Estes tipos são convertidos para *int* antes de serem colocados na pilha de operandos. As instruções aritméticas são as seguintes:

- Adição: Add: iadd, ladd, fadd, dadd;
- Subtração: isub, lsub, fsub, dsub;
- Multiplicação: imul, lmul, fmul, dmul;
- Divisão: idiv, ldiv, fdiv, ddiv;
- Resto: irem, lrem, frem, drem;
- Troca de sinal: ineg, lneg, fneg, dneg;
- Deslocamento: ishl, ishr, iushr, lshl, lshr, lushr;
- OR bit a bit: ior, lor;
- AND bit a bit: iand, land;
- XOR bit a bit: ixor, lxor;
- Incremento de variável local: iinc;
- Comparação: dcmpg, dcmpl, fcmpg, fcmpl, lcmp.

A JVM não indica a existência de *overflow* durante operações com inteiros. As únicas operações com inteiros que podem lançar uma exceção são as operações de divisão entre inteiros (*idiv* e *ldiv*) e as instruções de resto inteiro (*irem* e *lrem*), as quais lançam uma exceção *ArithmeticException* se o divisor for zero.

### 3.4.3 Instruções de Conversão de Tipos

A JVM suporta as seguintes conversões númericas:

- *int* para *long* (*i2l*), *float* (*i2f*) ou *double* (*i2d*);

– *long* para *float* (*l2f*) ou *double* (*l2d*);

– *float* para *double* (*f2d*).

Estas conversões são utilizadas pois o programador explicitamente faz uma conversão de tipo no seu programa, ou então ocorre implicitamente através da máquina virtual, de modo a suprir a falta de algumas operações com outros tipos númericos.

### 3.4.4 Criação e Manipulação de Objetos

Embora tanto instâncias de classes como matrizes sejam objetos, a JVM as cria e manipula de formas distintas:

– Cria uma nova instância de uma classe: *new*;

– Cria uma nova matriz: *newarray*, *anewarray*, *multianewarray*;

– Acessa campos de classes (*static fields*) e atributos de instâncias: *getfield*, *putfield*, *getstatic*, *putstatic*;

– Carrega um item de uma matriz na pilha de operandos: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*;

– Armazenam um valor da pilha de operandos como um item de uma matriz: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*;

– Coloca na pilha de operandos o tamanho da matriz: *arraylength*;

– Verifica propriedades de instâncias de classes e matrizes: *instanceof*, *checkcast*.

### 3.4.5 Instruções de Transferência de Controle

As instruções de transferência de controle, condicional ou incondicionalmente, fazem com que a JVM pule para outras posições no método, continuando com a sua execução. São elas:

– Condicional: *ifeq*, *iflt*, *ifle*, *ifne*, *ifgt*, *ifge*, *ifnull*, *ifnonnull*, *if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmpgt*, *if_icmple*, *if_icmpge*, *if_acmpeq*, *if_acmpne*;

– Incondicional: *goto*, *goto_w*, *jsr*, *jsr_w*, *ret*;

– Para suportar a construção *switch*: *tableswitch*, *lookupswitch*.

### 3.4.6 Instruções de Chamada e Retorno de Métodos

As seguintes instruções executam chamadas a métodos:

- *invokevirtual* chama um método em um objeto;
- *invokeinterface* chama um método que é implementado por uma interface, buscando a implementação correta do objeto a ser chamada em tempo de execução;
- *invokespecial* chama um método de um objeto que necessite de um tratamento especial, seja um método de inicialização de uma instância, um método privado, ou um método da classe pai (*super*);
- *invokestatic* chama um método estático em uma classe.

As instruções que retornam os resultados nos métodos são distinguidas pelo tipo de retorno. São elas: *ireturn* (usado em tipos *boolean*, *byte*, *char*, *short*, ou *int*), *lreturn*, *freturn*, *dreturn* e *areturn*. Para métodos *void*, que não retornam nenhum resultado, métodos inicializadores de instâncias, classes ou interfaces, é utilizada a instrução *return*.

### 3.4.7 Instruções de Manipulação de Exceções

Uma exceção é lançada programaticamente usando a instrução *athrow*. Exceções também podem ser lançadas por várias instruções da JVM se uma condição anormal for detectada.

Blocos *try* ... *catch* podem ter um bloco *finally*, cujas instruções são executadas indiferentemente de haver uma exceção ou não. A implementação de *finally* pela JVM utiliza as instruções *jsr*, *jsr_w*, e *ret*.

### 3.4.8 Instruções para Sincronização

A sincronização de sequências de instruções é tipicamente utilizada nos métodos com o modificador *synchronized* da linguagem Java. A JVM oferece as instruções *monitorenter* e *monitorexit* para suportar tal construção da linguagem.

## 3.5 Execução

A JVM inicia a execução de uma aplicação chamando o método *main* de uma classe, cuja assinatura é mostrada abaixo, onde *args* são os parâmetros passados para a aplicação:

```
public static void main(String[] args)
```

A execução do método *main* ocorre na *thread* principal, porém, antes da execução, a JVM realiza a ligação e inicialização da classe.

A ligação envolve o processo de verificação, preparação e, opcionalmente, resolução.

O passo de verificação assegura que a representação carregada da classe a ser executada está semanticamente correta. Se algum problema for detectado durante esta fase, um erro é lançado.

A preparação envolve a alocação de espaço para as estruturas de dados que são utilizadas internamente pela máquina virtual, como tabelas de métodos.

Resolução é o processo de verificar as referências simbólicas para as outras classes e interfaces, carregando-as e verificando se as referências estão corretas.

A inicialização da classe consiste na execução dos inicializados estáticos da classe, de acordo com a ordem no programa (*textual order*). Porém, antes da inicialização da classe, sua classe pai deve ser inicializada, assim como todas as outras classes que estão acima na hierarquia de classes. No caso mais simples, a classe pai será Object (*java.lang.Object*).

Após todos esses passos, o método *main* é executado.

## 3.6 Especificação dos Arquivos *.class*

O arquivo *.class* é um arquivo binário, em um formato definido para que programas possam ser executados na JVM, de forma independente de plataforma e da implementação da máquina virtual. Cada arquivo *.class* contém a descrição completa de uma, e somente uma, classe ou interface Java.

Os dados são armazenados sequencialmente no arquivo, sem espaçamento (*padding*) ou alinhamento (*alignment*) entre eles. Esta falta de espaçamento reforça o cuidado dos projetistas da JVM em manter o *bytecode* compacto. Os dados dos itens das estruturas de dados que representam a classe/interface que ocuparem mais de um *byte* são divididos e organizados na ordem *big-endian* [1].

O presente trabalho não irá descrever a estrutura dos arquivos *.class*, visto que um extenso estudo sobre o tema pode ser encontrado em Silva (2003).

[1] Neste formato, os bytes de maior ordem ficam nas primeiras posições, o caso contrário é chamado de *little-endian*.

# 4 Implementação

Neste trabalho foi implementado um compilador para uma linguagem funcional que tem *bytecodes* Java como o seu código objeto. A linguagem funcional definida tem sintaxe similar a de Haskell, sua ordem de avaliação é estrita, é estaticamente tipada, possui inferência de tipos e não possui suporte à sobrecarga (*overload*). O compilador possui no seu *frontend* um analisador sintático e um inferidor de tipos, e seu *backend* gera os *bytecodes* Java.

As seções seguintes demonstram os recursos do compilador desenvolvido, bem como alguns aspectos de sua implementação e utilização.

## 4.1 Analisador Sintático

O analisador sintático é uma parte de um compilador ou interpretador, cuja função é verificar se o texto de entrada (código fonte) está de acordo com a especificação léxica e sintática da gramática especificada. Os analisadores léxicos e sintáticos podem ser implementados ou gerados semi-automaticamente por uma ferramenta que tem como entrada uma gramática escrita em BNF .

Neste trabalho, o analisador sintático foi programado sem o auxílio de uma ferramenta, utilizando uma biblioteca de combinadores monádicos em Haskell chamada de Parsec, o qual é visto em maiores detalhes na próxima seção.

### 4.1.1 Parsec

Parsec é uma biblioteca de combinadores monádicos em Haskell. Combinadores são funções de ordem superior que são utilizadas de forma infixa, e ditas monádicas porque lidam com mônadas.

Ao contrário de ferramentas que geram analisadores sintáticos, como o clássico Yacc (JOHNSON; SETHI, 1990), que oferecem um conjunto fixo de combinadores para definir gramáticas,

os combinadores monádicos são manipulados como valores de primeira classe[1] e podem ser combinados de forma a definirem analisadores mais específicos (LEIJEN; MEIJER, 2001). Outra vantagem é que o programador utiliza apenas uma linguagem, evitando a integração de diferentes linguagens e ferramentas (HUGHES, 2000).

Uma restrição do Parsec, e da maioria dos analisadores baseados em combinadores, é que não há uma maneira de lidar com recursão à esquerda. Caso uma produção seja recursiva à esquerda, o analisador entrará em um *loop* infinito. Uma solução é a reescrita da gramática, uma vez que toda gramática recursiva à esquerda pode ser reescrita como recursiva à direita (AHO; SETHI; ULLMAN, 1986). Outra solução é a utilização de um combinador *chainl* (FOKKER, 1995), que implementa o padrão de reescrita da gramática.

O Parsec já vem incluso na instalação do Glasgow Haskell Compiler - GHC , e pode ser importado através do módulo *Text.ParserCombinators.Parsec*.

Para construir um exemplo de uso do Parsec, vamos definir uma simples gramática para blocos de linguagens de programação:

```
<sentenca> ::= letras ou números
<bloco> =  [ <sentenca> ; ]+
```

Um bloco contém uma lista de sentenças, e este é o resultado da avaliação da função *bloco* do exemplo abaixo:

```
import Text.ParserCombinators.Parsec

separador = char ';'
sentenca  = many alphaNum
bloco     = do char '{';
                sentencas <- sentenca `endBy1` separador
                char '}';
                return sentencas
```

As funções *many*, *alphaNum*, *endBy1* e *char* são combinadores monádicos fornecidos pelo Parsec. Para testar o analisador, podemos utilizar a função *parseTest*:

[1] Valor de primeira classe (*first-class value* ou também as variantes *first-class object*, *first-class citizen* e *first-class object*) é uma entidade que pode ser usada sem restrições em programas (sem restrições quando comparado com outros objetos na mesma linguagem). O termo foi cunhado por Christopher Strachey na década de 1960.

```
*Main> parseTest bloco "{sentenca1;sentenca2;}"
["sentenca1","sentenca2"]
```

Vamos testar o caso onde uma sentença fica sem o ponto e vírgula no seu final:

```
*Main> parseTest body "{sentenca1}"
parse error at (line 1, column 11):
unexpected "}"
expecting letter or digit or ";"
```

Como pode-se ver no exemplo acima, a biblioteca não apenas informa em qual linha do texto de entrada o erro ocorreu, como também lista as possibilidades de produção (como o conjunto PRIMEIROS de um não-terminal). Uma das vantagens é que podemos associar uma mensagem de erro com os combinadores utilizando-se o combinador de erro <?>. Reescrevendo o exemplo anterior com mensagens de erro personalizadas:

```
separador = char ';' <?> "separador de sentenca"
sentenca  = many (alphaNum <?> "sentenca")
bloco     = do char '{';
               sentencas <- sentenca `endBy1` separador
               char '}';
               return sentencas
```

Testando esta nova definição obtemos:

```
*Main> parseTest body "{sentenca1}"
parse error at (line 1, column 11):
unexpected "}"
expecting sentenca or separador de sentenca
```

Para conhecer outros combinadores e obter a documentação dos utilizados no exemplo desta seção consulte a documentação do Parsec (LEIJEN, 2001).

### 4.1.2 Código Intermediário

De acordo com o texto de entrada, o analisador sintático armazena em suas estruturas de dados uma representação intermediária, chamada de código intermediário. Nesta subseção veremos as principais estruturas utilizadas, bem como os seus significados.

A representação adotada foi inspirada nos tipos algébricos usados por Mark P. Jones (JONES, 1999) para descrever informalmente o sistema de inferência de tipos da linguagem Haskell.

#### Identificadores

Os identificadores da linguagem são representados pelo tipo *Id*, que armazena uma *string* com o nome do identificador e também um inteiro representando o nível da sua declaração. Um nível de valor zero indica um identificador declarado no escopo global, e maiores que zero, indicam variáveis declaradas através do comando *let*.

```
data Id = Id String Int
```

#### Tipos

Os tipos são representados através do tipo algébrico *Type*. As expressões de tipos podem ser: variáveis de tipo (*TVar*), construtores de tipos (*TCon*), aplicações de um tipo para outro (*TAp*) e tipos polimórficos (também chamados de variáveis de tipo quantificadas).

```
data Type = TVar Tyvar
          | TCon Tycon
          | TAp Type Type
          | TGen Int

data Tyvar = Tyvar String [Int]
data Tycon = Tycon Id
```

Apresentamos abaixo alguns exemplos de como representar tipos primitivos da linguagem utilizando os construtores de tipos:

```
tChar = TCon (Tycon (Tycon (Id "Char"0)))
tArrow= TCon (Tycon (Tycon (Id "(->)"0)))
tList = TCon (Tycon (Tycon (Id "[]"0)))
```

Uma função auxiliar *fn* é utilizada para facilitar a criação de tipos para funções:

```
infixr 4 `fn`
fn ::  Type -> Type -> Type
a `fn` b = TAp (TAp tArrow a) b
```

**Substituições**

As substituições têm um papel importante na inferência de tipos, pois mapeiam variáveis de tipo para tipos.

```
type Subst = [(Tyvar, Type)]
```

Como as substituições podem ser aplicadas a tipos ou a qualquer estrutura que os contenha, então a função de aplicação (*apply*) deve ser sobrecarregada para diferentes tipos. Para isso implementamos uma classe:

```
class Subs t where
   apply ::  Subst -> t -> t
   tv ::  t -> [Tyvar]
```

A função *tv* retorna um conjunto de variáveis de tipo que estão contidas em seu argumento, ordenadas por ordem de ocorrência.

**Esquemas de Tipo (*Type Scheme*)**

*Type schemes* são utilizados na representação de tipos polimórficos, também chamados de tipos quantificados.

```
data Scheme = Forall Type
```

### Suposições de Tipo

Suposições sobre o tipo de uma variável são representadas por valores de *Assump*. Uma dupla com o tipo de definição (*Kind_of_defining_occurrence*) e com o esquema de tipo (*Scheme*) é associado a um identificador:

```
data Assump = Id :>:  (Kind_of_defining_occurrence, Scheme)
data Kind_of_defining_occurrence = CW | LB
```

O tipo de definição (*Kind_of_defining_occurrence*) indica se a variável é ligada a uma $\lambda$-abstração.

### Literais

Valores literais são representados por *Literal*:

```
data Literal = LitInt Integer
             | LitChar Char
             | LitStr String
             | LitFloat Double
```

### Padrões

Padrões (do inglês *patterns*) são utilizados para inspecionar e desconstruir valores de tipos algébricos. Os padrões serão representados através de instâncias do tipo *Pat*:

```
data Pat = PVar Id
         | PLit Literal
         | PCon Assump [Pat]
         | PWildcard
```

O construtor *PVar* representa variáveis que estão no padrão, da mesma forma literais por *PLit*, construções de padrões por *PCon*, e um padrão coringa por *PWildcard* (representado em programas como "_").

A seguinte função retorna o primeiro elemento de uma lista:

```
head ::  [a] -> a
head (x:_) = x
```

Para fazer isto, a lista é desconstruída através do construtor de listas ":" em dois padrões: um padrão para a cabeça da lista ($x$) e outro que ignora o resto da lista. O retorno da função é o valor da variável $x$, que é associada ao padrão.

O código intermediário gerado pelo analisador sintático para o padrão da função *head* é:

```
PCon (Assump para ":")  [PVar (Id "x" 0),PWildcard]
```

### Expressões

As expressões são representadas por instâncias de *Expr*:

```
data Expr = Var Id
          | Lit Literal
          | Const Assump
          | Ap Expr Expr
          | Let BindGroup Expr
```

Variáveis e literais são representados, respectivamente, por *Var* e *Lit*. O construtor *Const* é utilizado para constantes com nomes, como, por exemplo, construtores de tipos. A aplicação de funções e a expressão *let* são representadas, respectivamente, por *Ap* e *Let*.

### Alternativas

Para explicar alternativas, vamos analisar o seguinte exemplo:

```
len ::  [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

Neste exemplo, se diz que a função *len* possui duas alternativas: a primeira retorna zero se a lista passada é vazia, e a segunda retorna um somado ao tamanho da lista *xs*.

As alternativas para as funções são representadas através do tipo *Alt*.

```
type Alt = ([Pat], Expr)
```

Este tipo é uma tupla, cujo primeiro componente é uma lista de padrões para cada um dos argumentos da função, e o segundo componente é a expressão associada àquela alternativa.

***Binding Groups***

Os *binding groups* são utilizados para agrupar todas as alternativas para um identificador, e podem ser explícitos e implícitos:

– Explícitos: este tipo de *binding group* representa funções em que o programador fez uma declaração explícita do seu tipo. Estes tipos serão verificados na fase de inferência.

```
type Expl = (Id, Scheme, [Alt])
```

– Implícitos: representam funções sem tipo explícito, que terão seus tipos inferidos durante o processo de inferência.

```
type Impl = (Id, [Alt])
```

Desta forma, um programa em Haskell pode ser representado por um tipo *BindGroup*, que armazena a lista de *binding groups* explícitos e implícitos:

```
type BindGroup = ([Expl], [Impl])
```

```
{ data Cores = Azul | Verde | Vermelho; }
```

Figura 4.1: Exemplo de tipo de dado de enumeração para cores

**Exemplos de saídas**

Vejamos dois exemplos de programas escritos na sintaxe da linguagem definida neste trabalho, analisando também a saída gerada pelo analisador sintático.

Neste primeiro exemplo, vamos declarar um tipo de dado de enumeração para algumas cores:

Uma vez que o analisador sintático ainda não fornece suporte a *layouts*, o programa deve ser declarado em um bloco entre chaves e cada declaração do programa deve ser terminada em ponto e vírgula, sendo que a última opcional.

A saída do analisador sintático para o exemplo da Fig. 4.1 é mostrada na Fig. 4.2.

```
([], -- Não há nenhum binding group explícito (Expl)
 [ -- Lista de binding groups implícitos (Impl)
  (Id "Azul" 0, -- Identificador ‘‘Azul’’ é definido em escopo global
    [ -- Lista de alternativas (Alt)
      ([], -- Nenhum padrão (Pat)
       -- Para um construtor de tipo, utilizamos o construtor Const de Expr
       -- :>:  é um construtor de suposições, assim, para todo ‘‘Azul’’ temos
       -- a suposição de tipo que ‘‘Azul’’ é do tipo ‘‘Cor’’
       Const (Id "Azul" 0 :>:  (CW,Forall (TCon (Tycon (Id "Cor" 0)))))
      )
    ]
  ), -- A explicação acima vale para os demais construtores de Cor
  (Id "Verde" 0,
      [([],Const (Id "Verde" 0 :>:  (CW,Forall (TCon (Tycon (Id "Cor" 0))))))]),
  (Id "Vermelho" 0,
      [([],Const (Id "Vermelho" 0 :>:
            (CW,Forall (TCon (Tycon (Id "Cor" 0))))))])
 ])
```

Figura 4.2: Saída do analisador sintático para o exemplo da Fig. 4.1

Neste segundo exemplo, mostrado na Fig. 4.3, vamos definir uma função polimórfica que calcula o tamanho de uma lista.

```
{
  len [] = 0
  len (x:xs) = 1 + len xs
}
```

Figura 4.3: Implementação da função *len*

Como não especificamos o tipo da função *len*, a lista de *binding groups* explícitos, assim como no exemplo anterior, é vazia. Por este motivo, vamos omitir algumas informações para simplificar a saída, mostrada na Fig. 4.4.

```
[  -- Lista de Impl
  (Id "len" 0,
    [ -- Lista de Alt
     ([PCon {- Suposição de tipo para [] -} []], -- Padrão
      Lit (LitInt 0) -- Expressão
     ),
     ( -- Padrão
      [PCon {- Suposição de tipo para (:)  -} [PVar (Id "x" 0),
                                               PVar (Id "xs" 0)]],
       -- Expressão
       -- Podemos notar que ‘‘1 + len xs’’ é o mesmo que a aplicação
       -- da função (+1) com a função (len xs)
       Ap (Ap (Var (Id "+" 0)) (Lit (LitInt 1)))
          (Ap (Var (Id "len" 0)) (Var (Id "xs"0)))
     )
    ] -- Final da lista de Alt
  )
]
```

Figura 4.4: Exemplo de saída do analisador sintático para a função *len*

## 4.2 Inferidor de Tipos

O inferidor de tipos possui como principal estrutura de dados o tipo algébrico *Assump*, que já foi descrito na seção 4.1.2. Um contexto de tipos é uma lista de suposições de tipo,

representado pelo tipo sinônimo *TypCtx*:

```
type TypCtx = [Assump]
```

No começo do processo de inferência de tipos é passado um contexto de tipo inicial. Algumas das suposições que compõem o contexto inicial são mostradas abaixo:

```
"+" :>:  (CW, Forall (tInt `fn` tInt `fn` tInt))
"−" :>:  (CW, Forall (tInt `fn` tInt `fn` tInt))
"*" :>:  (CW, Forall (tInt `fn` tInt `fn` tInt))
"/" :>:  (CW, Forall (tInt `fn` tInt `fn` tFloat))
"&&" :>:  (CW, Forall (tBool `fn` tBool `fn` tBool))
"||" :>:  (CW, Forall (tBool `fn` tBool `fn` tBool))
"not" :>:  (CW, Forall ((tBool `fn` tBool)))
"True" :>:  (CW, Forall tBool)
"False" :>:  (CW, Forall tBool)
"[]" :>:  (CW, (Forall (TAp tList (TGen 0))))
":" :>:  (CW, (Forall (TGen 0 `fn`
                    TAp tList (TGen 0) `fn` TAp tList (TGen 0))))
```

Baseado nesse contexto inicial, o tipo de cada declaração de função é inferido por um algoritmo similar ao algoritmo W, mostrado na Fig. 2.2.

Em uma segunda etapa, os tipos inferidos para as declarações são unificados com os tipos requeridos. O tipo requerido é o tipo que foi inferido no momento da aplicação de uma função. Para que não exista a necessidade de uma ordenação topológica baseada na ordem de aplicação de funções, foi utilizado para a unificação dos tipos, inferidos e requeridos, o algoritmo proposto por Vasconcellos, Figueiredo e Camarao (2003).

## 4.3 Saída do Compilador (*backend*)

Tendo como entrada o código intermediário e os tipos inferidos do programa, o objetivo da última parte do compilador é gerar *bytecodes* Java. Nas subseções a seguir, veremos como são gerados os arquivos *.class* e quais as traduções que ocorrem para se obter um código orientado a objeto a partir de um código funcional.

### 4.3.1 Jasmin

Um dos objetivos deste trabalho é gerar *bytecodes* Java para programas funcionais. Estes arquivos *.class*, como já foi visto na seção 3.6, são arquivos binários. Geralmente os compiladores de linguagens nativas, por exemplo de C, geram um arquivo texto que contém a representação do programa em uma linguagem de montagem (*assembly*). Este arquivo é posteriormente compilado por um montador para um formato binário. Infelizmente, não há uma linguagem de montagem e um respectivo montador no *kit* de desenvolvimento Java (Java Development Kit - JDK).

O padrão *de facto* de uma linguagem de montagem Java é o Jasmin (REYNAUD; MEYER, 2006). Este montador recebe na sua entrada um arquivo ASCII com descrições das classes Java, em um formato simples e utilizando os mesmos nomes de *opcodes* da JVM. O montador Jasmin então converte estas descrições para o formato binário dos arquivos *.class*, que então podem ser carregados normalmente pela JVM.

O montador Jasmin é distribuído através de um pacote Java (Java *ARchive* - JAR), e o funcionamento de como gerar os arquivos *.class* é mostrado abaixo:

```
$ java -jar jasmin.jar arquivo1.j arquivo2.j ...
```

#### Formato do Arquivo Jasmin

O formato do arquivo de montagem do Jasmin é mostrado abaixo:

```
<arquivo_jasmin> {
    <cabecalho>
    [<atributos>]*
    [<metodos>]*
}
```

Apenas uma classe Java pode ser especificada por arquivo de montagem. Desta forma, informações sobre a classe ficam na parte de cabeçalho, seguidas, opcionalmente, pelos atributos e métodos da classe.

```
.class public NomeDaClasse
.super java/lang/Object
.implements nome/qualificado/da/Interface
```

O código de montagem acima é equivalente a seguinte classe Java:

```
public class NomeDaClasse
       extends java.lang.Object
       implements nome.qualificado.da.Interface {}
```

Como podemos observar, a classe "NomeDaClasse" estende a classe *Object* do pacote *java.lang* e implementa a interface declarada por *nome.qualificado.da.Interface*. Para declararmos um atributo inteiro, público e com nome "nomeDoAtributo", usamos a seguinte linha:

```
.field public nomeDoAtributo I
```

A especificação dos tipos segue o padrão dos descritores de tipos (vistos na seção 3.4).

Resta-nos saber agora como descrever métodos utilizando a linguagem Jasmin. Um exemplo que declara um método chamado "umaString", sem parâmetros, público, e que retorna uma *string*, é mostrado abaixo:

```
.method public umaString()Ljava/lang/String;
  ldc ''isto eh uma string''
  areturn
.end method
```

A partir do exposto até agora, podemos escrever o clássico exemplo de "Alô mundo!" em Jasmin:

```
.class public Hello
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
  .limit stack 2

  getstatic java/lang/System/out Ljava/io/PrintStream;

  ldc "Alô Mundo!"

  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

  return
.end method
```

Com a primeira linha dentro do método, especificamos o limite de valores que podem ser empilhados na pilha de operandos aritméticos. No exemplo acima, a pilha tem tamanho de duas posições e, como podemos notar, apenas as instruções do montador começam com um ponto. As instruções da JVM não são prefixadas por este. A instrução *getstatic* pega um campo estático da classe do primeiro parâmetro. Assim, no exemplo, se esta fazendo uma referência para *java.lang.System.out*, cujo tipo é *java.io.PrintStream*. Logo após, a instrução *ldc* empilha um elemento que está na tabela de símbolos (*constant pool*), no caso do exemplo, a *string* "Alô Mundo!". Após a preparação para a chamada do método que irá imprimir na saída padrão do processo Java, é executado o método *println* através da instrução *invokevirtual*. No final do exemplo, temos a instrução *return*, fazendo com que a máquina virtual restabeleça o controle para o método que efetuou a chamada.

O resultado da execução, após a montagem, do exemplo é a frase "Alô Mundo!" na tela.

Para maiores informações e detalhes de cada uma das declarações possíveis, basta consultar a documentação do Jasmin (REYNAUD; MEYER, 2006).

### 4.3.2 Traduções

A linguagem funcional definida neste trabalho possui dois tipos de declarações: tipos de dados algébricos e funções. É com base nestes que iremos definir as traduções do paradigma funcional para o orientado a objeto da JVM. Embora o código seja gerado na linguagem de

montagem Jasmin, para facilitar a leitura, os exemplos serão apresentados na linguagem Java.

Primeiramente, os tipos de dados algébricos, como já vimos na seção 2.3.1, são divididos em: enumeração, compostos, recursivos e polimórficos. A tradução do tipo de dado algébrico de enumeração pode ser feito através de um novo tipo de dado na versão 5 da linguagem Java, o tipo enumerado (*enum type*) (MICROSYSTEMS, 2004), ou através do padrão *Typesafe Enum* (BLOCH, 2008), este compatível com versões anteriores da JVM.

Entretanto, este trabalho optou pela tradução desses tipos algébricos de uma maneira uniforme, de forma que cada construtor do tipo algébrico seja uma subclasse de uma classe abstrata que leva o nome do tipo. A tradução para a linguagem Java do exemplo de tipo algébrico de enumeração da Fig. 4.1 é mostrada abaixo:

```
public abstract class Cores {}
public Azul extends Cores {}
public Verde extends Cores {}
public Vermelho extends Cores {}
```

Diferentemente do tipo algébrico de enumeração, os demais tipos possuem construtores com parâmetros. Embora sejam tipos de dados, se passarmos apenas alguns parâmetros para um construtor, estaremos fazendo uma aplicação parcial (currificação) sobre o tipo de dado. Por exemplo:

```
{
data Animal = Gato String Int
}
```

O tipo do construtor *Gato* é *String -> Int -> Gato*, logo, só teremos uma instância de *Gato* se passarmos uma *string* e um inteiro. Como funções e tipos algébricos podem sofrer aplicações parciais, podemos pensar em uma maneira uniforme de representá-los.

A representação escolhida é o uso de *closures*, que consiste em um par de ponteiros, um para o código da função e outro para o registro de ativação (MITCHELL; APT, 2001). Quando o *closure* vai ser executado é feito um salto para o endereço do código da função. Em Java, isto será feito através da chamada a um método.

O presente trabalho definiu uma classe genérica e abstrata, chamada de *Fun*, a qual possui um método abstrato *apply* que mapeia elementos de um tipo para outro. Esta classe representa

a idéia do *closure*, e o método *apply* a da execução do *closure*. A implementação da classe *Fun* é mostrada abaixo:

```
public abstract class Fun<A, B> {
  public abstract B apply(A x);
}
```

Isso significa que em tempo de compilação as variáveis de tipo *A* e *B* serão substituídas pelos tipos especificados, garantindo que o programa esteja protegido pelo sistema de tipos contra erros de tipo.

Com base nisso, implementamos a tradução dos outros tipos algébricos como mostrado no exemplo abaixo, utilizando como exemplo o tipo *Animal*:

```
public abstract class Animal {}
public class Gato extends Animal {
  public String f1;
  public Integer f2;
  public Gato(String f1, Integer f2) {
    this.f1 = f1;
    this.f2 = f2;
  }
}
public class GatoF1 extends Fun<String, GatoF1F2> {
  public GatoF1F2 apply(String x) {
    return new GatoF1F2(x);
  }
}
public class GatoF1F2 extends Fun<Integer, Gato> {
  public String f1;
  public GatoF1F2(String f1) {
    this.f1 = f1;
  }
  public Gato apply(Integer x) {
    return new Gato(this.f1, x);
  }
}
```

O tipo algébrico *Animal* é traduzido em uma classe abstrata, pois não pode ser instanciada. Cada um dos construtores de *Animal* são subclasses deste. Se um construtor possui $n$ parâmetros então serão geradas $n+1$ classes intermediárias. Cada uma delas recebe o valor de um parâmetro, e retorna a instância da próxima classe, que recebe o próximo parâmetro e retorna outra classe, etc. A classe que receber o último parâmetro retorna uma instância da que possui o nome do construtor.

Como o construtor de tipo *Gato* possui dois parâmetros, serão necessárias três classes: *GatoF1* que recebe em *apply* o primeiro parâmetro (*String*) e retorna uma instância de *GatoF1F2*, *GatoF1F2* que recebe o segundo parâmetro do construtor (*Integer*), e *Gato*, cuja instância é retornada por *GatoF1F2* em seu método *apply*.

Uma vez que já definimos uma maneira de se traduzir tipos algébricos, e também uma classe abstrata *Fun* que força suas subclasses a implementarem o método *apply*, podemos utilizar esta mesma estrutura para implementar de uma maneira uniforme a tradução de funções.

Vamos utilizar como exemplo a função *map*, que através de uma função, mapeia uma lista de elementos de tipo *a*, para uma lista de elementos de tipo *b*. Uma implementação desta é mostrada na Fig. 4.5.

```
{
map ::  (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x :  map f xs
}
```

Figura 4.5: Um exemplo de implementação da função *map*

A tradução da função *map* depende de uma implementação de lista. Como o tipo lista é comumente utilizado, optamos por definí-lo de uma forma simplificada, fora da maneira descrita para tipos algébricos, visando a facilidade na interoperabilidade com outras linguagens. A tradução do tipo lista é mostrada na Fig. 4.6.

Na tradução dos tipos algébricos, se um construtor possui $n$ parâmetros, será necessária uma classe a mais $(n+1)$, pois esta representa a instância do construtor, que deve conter todos os parâmetros. Entretanto, na tradução de funções, se uma função possui $n$ argumentos, serão necessárias $n$ classes, porque a última apenas retorna o resultado da computação da expressão.

A primeira classe da tradução da função *map* será chamada de *MapF* devido à junção dos

```
abstract public class Lista<A> {}
public class Nil<A> extends Lista<A> {}
public class Cons<A> extends Lista<A> {
  public A x;
  public Lista<A> xs;

  public Cons(A x, Lista<A> xs) {
    this.x = x;
    this.xs = xs;
  }
}
```

Figura 4.6: Tradução do tipo lista para classes Java

nomes da função e de seu primeiro parâmetro [2]. A classe *MapF* recebe em *apply* a instância de uma classe que estenda *Fun<A,B>* e retorna uma instância da classe *MapFX*.

```
public class MapF<A, B> extends Fun< Fun<A, B>, MapFX<A, B> > {
  public MapFX<A, B> apply(Fun<A, B> x) {
    return new MapFX<A, B>(x);
  }
}
```

A segunda classe da tradução será chamada de *MapFX*, que recebe através de *apply* uma instância de lista com tipo genérico “A”, e retorna uma instância de lista com tipo genérico “B”. A tradução para esta classe é mostrada na Fig. 4.7.

Cumpre salientar que as instâncias diretas de *Lista* não existem por esta se tratar de uma classe abstrata. Existem instâncias apenas de suas subclasses *Nil* e *Cons*. Como o método *apply* recebe como parâmetro *Lista*, precisaremos verificar de qual classe é a instância, e então chamar o método para a subclasse de Lista. O método *applyFor* é um método sobrecarregado para cada construtor do tipo algébrico, enquanto o método *apply* apenas verifica a classe da instância e executa a chamada ao método *applyFor*, passando o parâmetro que recebeu.

[2] A convenção de nomenclatura de identificadores chamada de *camel case* é utilizada, onde “nome de uma classe” vira o identificador “nomeDeUmaClasse”. No caso do identificador ser o nome de uma classe, utiliza-se a primeira letra em maiúscula.

```
public class MapFX<A, B> extends Fun<Lista<A>, Lista<B> > {
  public Fun<A, B> f;

  public MapFX(Fun<A, B> x) {
    this.f = x;
  }

  public Lista<B> applyFor(Nil<A> x) {
    return new Nil<B>();
  }

  public Lista<B> applyFor(Cons<A> lista) {
    return new Cons<B>(f.apply(lista.x),
                (new MapF<A, B>()).apply(f).apply(lista.xs));
  }

  public Lista<B> apply(Lista<A> x) {
    if(x instanceof Nil)
      return applyFor((Nil<A>) x);
    else if (x instanceof Cons)
      return applyFor((Cons<A>) x);

    throw new RuntimeException("Unknown subtype for Lista");
  }
}
```

Figura 4.7: Implementação da segunda classe para a tradução de *map*

## 4.4 Síntese

Neste capítulo descrevemos todas as ferramentas e linguagens utilizadas durante a fase de implementação do compilador fruto deste trabalho. Também descrevemos através de exemplos e explicações como resolvemos alguns dos problemas da tradução de um programa escrito em uma linguagem funcional para outro paradigma, que no caso do presente trabalho, é orientado a objeto.

# 5 Conclusão

Este trabalho teve como objetivo o desenvolvimento de uma base que possibilite futuras pesquisas sobre a interoperabilidade entre linguagens de diferentes paradigmas e dos problemas envolvidos na tradução, principalmente entre os paradigmas funcional e orientado a objeto. Como parte dessa base, foi desenvolvido um compilador de uma linguagem funcional que gera *bytecodes* Java.

A implementação de um analisador sintático através de uma linguagem funcional com o uso de operadores monádicos é elegante, fácil e extremamente flexível. O código final é muito semelhante à gramática da linguagem, facilitando a adição de novas funcionalidades e a correção de problemas na implementação.

Durante o desenvolvimento das traduções, foram notadas semelhanças entre os tipos básicos da linguagem funcional e os de Java, dentre elas: os tipos numéricos são muito próximos, *strings* são vetores imutáveis de caracteres e nenhuma das linguagens possui tipo explícito de ponteiro.

O suporte a polimorfismo paramétrico (*generics*) presente na JVM a partir da versão 5 facilitou bastante a tradução das declarações, além de evitar a geração de várias classes que tenham o mesmo código, apenas com tipos diferentes.

Apesar da tradução ainda estar em uma fase inicial, não foram identificados grandes problemas em traduzir uma linguagem com um sistema de tipos similar a ML para uma representação orientada a objeto. O maior desafio, provavelmente, será a introdução de suporte a classes de tipos.

Por ser um trabalho muito extenso e que demanda um estudo aprofundado, especialmente sobre inferência de tipos, não foi possível focar em otimizações na tradução das declarações. Da mesma forma, não possuímos dados ou testes que verifiquem a eficiência dessas traduções, ficando como sugestão para trabalho futuro o estudo da atual performance em comparação com outras implementações de compiladores de linguagem funcional, assim como efetuar modificações para melhorar a performance atual.

Como já foi dito, este trabalho é muito extenso, e, por isso, não seria suficiente para explorar todos os tópicos envolvidos com a tradução de linguagens funcionais para a sua execução em plataformas de máquina virtual. Assim, tem-se também o objetivo de constituir uma base para trabalhos futuros.

## 5.1 Trabalhos Futuros

As seguintes metas podem ser destacadas para o aperfeiçoamento deste trabalho:

- Implementar a tradução de todas as declarações;
- Efetuar testes de performance e estudar possíveis otimizações, por exemplo, a otimização de chamadas recursivas de calda (*tail calls*);
- Avaliar a integração, sob a mesma plataforma, entre código funcional e orientação a objeto;
- Estudar a inclusão do suporte a mônadas;
- Implementar a ordem de avaliação preguiçosa;
- Desenvolvimento de um ambiente de execução (*runtime system*);
- Adicionar o tratamento de sobrecarga através de classes de tipo.

# *Referências Bibliográficas*

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-10088-6.

ARNOLD, K.; GOSLING, J. *The Java Programming Language*. third. [S.l.]: Addison-Wesley, 2000.

BACKUS, J. The history of Fortran I, II, and III. ACM, New York, NY, USA, p. 25–74, 1981.

BENTON, N.; KENNEDY, A. Interlanguage working without tears: blending SML with Java. In: *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 1999. p. 126–137. ISBN 1-58113-111-9.

BENTON, N.; KENNEDY, A.; RUSSELL, G. Compiling Standard ML to Java bytecodes. In: *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 1998. p. 129–140. ISBN 1-58113-024-4.

BINI, O. *Practical JRuby on Rails Web 2.0 Projects: Bringing Ruby on Rails to Java*. Berkeley, CA, USA: Apress, 2007. ISBN 1590598814.

BLOCH, J. *Effective Java (2nd Edition) (The Java Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN 0321356683, 9780321356680.

CHURCH, A.; ROSSER, J. Some properties of conversion. *Transactions of the American Mathematical Society*, v. 3, p. 472–482, 1936.

DAMAS, L.; MILNER, R. Principal Type-Schemes for Functional Programs. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. [S.l.]: ACM Press, 1982. p. 207–212. ISBN 0-89791-065-6.

FOKKER, J. Functional parsers. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK: Springer-Verlag, 1995. p. 1–23. ISBN 3-540-59451-5. Disponível em: <http://people.cs.uu.nl-/jeroen/article/parsers/parsers.ps>.

HUDAK, P. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 21, n. 3, p. 359–411, 1989. ISSN 0360-0300.

HUDAK, P. et al. A history of haskell: being lazy with class. In: *HOPL*. ACM, 2007. p. 1–55. Disponível em: <http://research.microsoft.com/~simonpj/papers/history-of-haskell/history-.pdf>.

HUGHES, J. Why functional programming matters. *Computer Journal*, Oxford University Press, Oxford, UK, v. 32, n. 2, p. 98–107, 1989. ISSN 0010-4620.

HUGHES, J. Generalising monads to arrows. *Science of Computer Programming*, v. 37, p. 67–111, 2000. Disponível em: <http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps>.

JENSEN, K.; WIRTH, N. *PASCAL - User Manual and Report*. [S.l.]: Springer Verlag, 1974.

JOHNSON, S. C.; SETHI, R. Yacc: a parser generator. W. B. Saunders Company, Philadelphia, PA, USA, p. 347–374, 1990.

JONES, M. J. S. P.; MEIJER, E. Type classes: exploring the design space. In: *Procedings of the Haskell Workshop 1997*. [s.n.], 1997. Disponível em: <http://research.microsoft.com-/Users/simonpj/Papers/type-class-design-space/multi.ps.gz>.

JONES, M. P. Typing Haskell in Haskell. In: *Proceedings of the 1999 Haskell Workshop*. [s.n.], 1999. Disponível em: <http://web.cecs.pdx.edu/~mpj/thih/>.

JONES, S. P. (Ed.). Language Definition. *Haskell 98 Language and Libraries: The Revised Report*. [s.n.], 2002. 277 p. PDF. Disponível em: <http://haskell.org/definition/haskell98-report.pdf>.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. second. [S.l.]: Prentice Hall, 1988.

LEIJEN, D. *Parsec, a fast parser combinator*. [S.l.], 2001. Disponível em: <http://legacy.cs-.uu.nl/daan/download/parsec/parsec.pdf>.

LEIJEN, D.; MEIJER, E. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. [S.l.], 2001. Disponível em: <http://www.cs.uu.nl/~daan/papers/parsec-paper.pdf>.

LINDHOLM, T.; YELLIN, F. *The Java Virtual Machine Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201432943.

LISP. 2008. Disponível em: <http://en.wikipedia.org/wiki/LISP>. Acesso em: 16 out. 2008.

MATTHEWS, J.; FINDLER, R. B. Operational semantics for multi-language programs. In: *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2007. p. 3–10. ISBN 1-59593-575-4.

MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, ACM, New York, NY, USA, v. 3, n. 4, p. 184–195, 1960. ISSN 0001-0782.

MCCARTHY, J. et al. *Lisp 1.5 Programmer's Manual*. second. [S.l.]: MIT Press, 1965.

MEIJER, E. Fundamentalist functional programming. In: *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*. New York, NY, USA: ACM, 2008. p. 99–100. ISBN 978-1-60558-267-2.

MICROSOFT. *Mondrian*. 2008. Disponível em: <http://research.microsoft.com/~emeijer-/Papers/ECOOP.pdf>. Acesso em: 12 nov. 2008.

MICROSOFT. *The SML.NET compiler*. 2008. Disponível em: <http://research.microsoft.com-/projects/sml.net/>. Acesso em: 12 nov. 2008.

MICROSYSTEMS, S. *Enums*. 2004. Disponível em: <http://java.sun.com/j2se/1.5.0/docs%-/guide/language/enums.html>.

MILNER, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, n. 17, p. 384–375, 1978.

MILNER, R.; TOFTE, M.; HARPER, R. *The Definition of Standard ML*. [S.l.]: MIT Press, 1990.

MITCHELL, J. *Foundations for Programming Languages*. [S.l.]: MIT Press, 1996.

MITCHELL, J. C.; APT, K. *Concepts in Programming Languages*. New York, NY, USA: Cambridge University Press, 2001. ISBN 0521780985.

MOGGI, E. Computational lambda-calculus and monads. In: *LICS*. [S.l.]: IEEE Computer Society, 1989. p. 14–23.

ODERSKY, M. *The Scala Programming Language*. 2008. Disponível em: <http://www.scala-lang.org>. Acesso em: 12 nov. 2008.

PEDRONI, S.; RAPPIN, N. *Jython Essentials: Rapid Scripting in Java*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002. ISBN 0596002475.

PIERCE, B. C. *Types and Programming Languages*. [S.l.]: MIT Press, 2002.

REYNAUD, D.; MEYER, J. *Jasmin*. March 2006. Disponível em: <http://jasmin.sourceforge-.net/>.

ROBINSON, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, ACM Press, v. 12, n. 1, p. 23–41, 1965. ISSN 0004-5411.

SILVA, J. C. da. *Migração do Ambiente de Programação da Linguagem LS para a Máquina Virtual Java*. Monografia (Bacharelado) — Curso de Ciência da Computação, Universidade Federal de Pelotas, Pelotas, 2003.

SYSTEMS, I. P. et al. *Draft proposed American National Standard Programming Language Common LISP*. Washington, DC, USA, jan. 1994.

TIOBE. *TIOBE Programming Community Index*. November 2008. Disponível em: <http://www.tiobe.com/tpci.htm>. Acesso em: 9 nov. 2008.

TOLKSDORF, R. *Programming Languages for the Java Virtual Machine*. 2008. Disponível em: <http://www.is-research.de/info/vmlanguages/>. Acesso em: 12 set. 2008.

TURNER, D. An overview of miranda. *Bulletin of the EATCS*, v. 33, p. 103–114, 1987. Disponível em: <http://dblp.uni-trier.de/db/journals/eatcs/eatcs33.html>.

VASCONCELLOS, C.; FIGUEIREDO, L.; CAMARAO, C. Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell. *j-jucs*, v. 9, n. 8, p. 873–890, 2003. Disponível em: <http://www.jucs.org/jucs_9_8/practical_type_inference_for>.

VASCONCELLOS, C. D. *Inferência de Tipos com Suporte para Sobrecarga Baseada no Sistema CT*. Tese (Doutorado) — Universidade Federal de Minas Gerais, 2004.

VENNERS, B. *Inside the Java Virtual Machine*. New York, NY, USA: McGraw-Hill, Inc., 1996. ISBN 0079132480.

WADLER, P. Comprehending monads. In: *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*. New York, NY, USA: ACM, 1990. p. 61–78. ISBN 0-89791-368-X.

WADLER, P.; BLOTT, S. How to make ad-hoc Polymorphism less ad-hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. [S.l.]: ACM Press, 1989. p. 60–76. ISBN 0-89791-294-2.

WIRTH, N. Design and Implementation of Modula. *Software – Practice and Experience*, v. 7, p. 67–84, 1977.