XBA: Uma linguagem orientada a objetos com tipagem estrutural

Henrique Stagni
hstagni@linux.ime.usp.br

Orientador: Prof. Francisco Reverbel
reverbel@ime.usp.br

Sumário

1 Introdução **3**
- 1.1 Tema . . . 3
- 1.2 Objetivo . . . 4
- 1.3 Estrutura do texto . . . 4

2 Sistemas de tipos **5**
- 2.1 Conceitos . . . 5
  - 2.1.1 Sistemas de tipo . . . 5
  - 2.1.2 Validade . . . 6
  - 2.1.3 Folga . . . 6
  - 2.1.4 Tipos dinâmicos . . . 6
  - 2.1.5 Inferência de tipos . . . 7
  - 2.1.6 Subtipos, tipagem nominativa ou estrutural . . . 7
- 2.2 Sistema de tipos Hindley-Milner . . . 7
  - 2.2.1 A linguagem L . . . 8
  - 2.2.2 Sistema de Tipos para L . . . 8
  - 2.2.3 Inferência de tipos . . . 10
  - 2.2.4 Variáveis genéricas de tipo - **let** com polimorfismo . . . 12

3 Linguagem XBA **13**
- 3.1 Características . . . 13
  - 3.1.1 Suporte à orientação a objetos . . . 13
  - 3.1.2 Funções como objetos de primeira classe . . . 14
  - 3.1.3 Aplicação parcial de métodos (*currying*) . . . 14
  - 3.1.4 Sistema de tipos estrutural . . . 14
  - 3.1.5 Uso de seletores para chamar métodos em objetos . . . 15
  - 3.1.6 Uso de identação para definir os blocos da linguagem . . . 15
- 3.2 Exemplos . . . 15

3.2.1 HelloWorld 15
3.2.2 Construtores e métodos 16
3.2.3 Estruturas de controle 17
3.2.4 Definindo novas estruturas de controle 18
3.3 JVM como alvo 19

4 Especificação da linguagem XBA **21**

5 Sistemas de tipos da linguagem XBA **22**
5.1 Descrição do sistema de tipos 22
5.2 Descrição do algoritmo de inferência utilizado 22

6 Implementação **23**
6.1 Introdução 23
6.2 Análise léxica e sintática 23
6.2.1 Explicação... 23
6.2.2 Javacc 23
6.3 Geração de código 23
6.3.1 Explicação... 23
6.3.2 ASM 23

7 Conclusão **24**

A Parte Subjetiva **26**
A.1 Dificuldades e desafios encontrados 26
A.2 Disciplinas relevantes para este trabalho 26
A.3 Continuação deste trabalho 26

Capítulo 1

Introdução

1.1 Tema

Linguagens com *tipagem dinâmica* têm sido alvo de grande atenção nos últimos anos. Por não se basearem em sistemas estáticos nominativos[1] de tipagem – que geralmente são muito rígidos – esse tipo de linguagem facilita a produção de um código mais flexível, reusável e, portanto, de sistemas capazes de se adaptar melhor à mudanças de especificações. Por outro lado, a não checagem de tipos em tempo de compilação também traz algumas desvantagens. A principal delas é que, evidentemente, muitos erros que seriam detectados durante a compilação em linguagens estáticas só são detectados em tempo de execução em linguagens dinâmicas. Em muitos casos, estes erros só são acusados em um lugar do código que é executado bem depois do que o trecho onde a inconsistência de tipos é realmente feita, dificultando o processo de debug. Além disso, linguagens dinâmicas podem apresentar um desempenho inferior ao alcançado por linguagens estáticas, que muitas vezes possuem compiladores capazes de gerar códigos otimizados baseados nas informações providas pela tipagem estática.

Linguagens com *tipos estruturais* podem ser consideradas, de certa forma, como um meio termo entre linguagens com tipagem estática nominativas e linguagens dinâmicas. Essas linguagens possuem um sistema de tipos checado estaticamente. Mas ao contrário de linguagens estáticas nominativas (como C, Pascal ou Java) – onde tipos são definidos por nomes explicitamente declarados – os tipos são dados implicitamente pelo conjunto de operações suportado por cada objeto da linguagem. Esse sistema de tipos permite uma flexibilidade quase tão

[1] usados por exemplo em linguagens como C, Pascal, Java

grande quanto aquela fornecida por linguagens dinâmicas, mantendo, ainda assim, a possibilidade de detecção de muitos erros em tempo de compilação.

1.2 Objetivo

O objetivo do trabalho proposto consiste na especificação de uma linguagem de programação , para a qual um compilador será gerado, e pelo desenvolvimento do compilador propriamente dito. Este último deve ser capaz de inferir os tipos estruturais dos programas de entrada – detectando possíveis erros – além de gerar os *bytecodes* correspondentes – de modo a permitir que esses programas sejam executados pela JVM.

A partir daqui, para facilitar a compreensão do texto, a linguagem desenvolvida será chamada de **XBA**[2]

1.3 Estrutura do texto

No capítulo 2, introduzimos informalmente conceitos sobre sistemas de tipos, apresentando, também, uma variação do sistema de tipos *Hindley-Milner*, usando uma linguagem simples como exemplo. No capítulo 3, introduzimos a linguagem XBA por meio de exemplos, fornecendo maiores detalhes sobre a mesma no capítulo seguinte. No capítulo 5, apresentamos o sistema de tipos que usaremos na linguagem. Por fim, discutimos alguns detalhes de implementação no capítulo 6.

[2] o nome XBA vem da nova instrução *invokedynamic* da JVM cujo opcode em hexadecimal é *0xBA*

Capítulo 2

Sistemas de tipos

2.1 Conceitos

2.1.1 Sistemas de tipo

Programas de computador podem, muitas vezes, conter erros - ou introduzido involuntariamente por programadores, ou decorrentes de uma especificação incompleta do problema que tentam resolver. Não é incomum, portanto, que a execução de um programa atinja estados indesejáveis e inrecuperáveis, como por exemplo uma divisão por 0, uma soma de um inteiro com uma *string* ou até mesmo um *loop* infinito.

Para que a introdução de erros em programas seja minimizada, muitas linguagens de programação oferecem *métodos formais* que checam, em tempo de compilação, se os programas escritos satisfazem certas propriedades, garantindo, assim, que certos estados indesejáveis nunca sejam atingíveis durante a execução. Dentre todos esses métodos formais, certamente o mais utilizado consiste em utilizar *sistemas de tipos*.

Um sistema de tipo pode ser definido [1] como um método que analisa sintaticamente um programa e prova a ausência de certos comportamentos indesejáveis, por meio da classificação de seus *termos*[1] de acordo com características dos valores para os quais esses termos são computados.

Essas características são chamadas de *tipos* e sistemas de tipos funcionam associando cada termo t de um programa a um certo tipo – nesse caso dizemos que t é *bem-tipado*. Essa associação é feita respeitando uma série de *regras de*

[1]qualquer construção sintática que, em tempo de execução, pode ser avaliada para um valor

tipagem (*typing rules*) que, para cada construção da linguagem, determina o tipo da expressão associada, com base nos tipos de suas subexpressões. Quando se associa um tipo a todos os termos de um programa, dizemos que o programa está *bem-tipado*

2.1.2 Validade

Como dito anteriormente, sistemas de tipos são concebidos com o intuito de assegurar que a execução de um programa não vai atingir um certo conjunto de estados indesejáveis.

Fixado um conjunto de estados indesejáveis S, é desejável que um sistema de tipos garanta que nenhum programa *bem-tipado* seja capaz de atingir um estado de S durante a execução. Essa propriedade é chamada de *validade*(*soundness*) de um sistema de tipos. A *validade* de um sistema é geralmente provada, mostrando que as seguintes propriedades são satisfeitas:

- **Progresso**: Durante a execução, um termo bem-tipado não está em nenhum estado de S.
- **Preservação**: Quando ocorre um passo da avaliação de um termo bem-tipado, o resultado é outro termo bem-tipado.

2.1.3 Folga

A *folga*(*slack*) de um sistema de tipos consiste no conjunto de programas válidos (aqueles que não atingem um estado indesejável durante a execução) que não são aceitos pelo sistema de tipo, isto é, que não podem ser tipados.

O problema de decidir se um programa de uma linguagem *Turing-completa* atinge um determinado conjunto de estados é indecidível. Esse é um corolário direto do problema da terminação (*halting problem*). Desta forma, como a checagem de tipos deve ser decidível, segue que todo sistema de tipos rejeita certos programas válidos, ou seja, possui alguma *folga*

2.1.4 Tipos dinâmicos

É comum o uso do termo *tipagem dinâmica* para linguagens que não fazem verificações estáticas (em tempo de compilação) de tipos, acusando erros apenas em tempo de execução. À rigor, pela definição apresentada anteriormente, essas

linguagens não apresentam um sistema de tipos (ou apresentam um sistema de tipos em que toda expressão é classificada como sendo de um tipo único, o que claramente não exige nenhuma verificação). Entretanto, por ser muito comum, usaremos essa expressão durante o texto.

2.1.5 Inferência de tipos

Dizemos que linguagens são *tipadas explicitamente* quando exige-se que o programador forneça *anotações de tipo* aos termos do programa, com o intuito de guiar a checagem de tipos. Linguagens que não exigem tais anotações possuem *tipagem implícita*. Nesse caso, a checagem de tipos deve ser capaz de *inferir* os tipos dos termos dos programas de entrada.

2.1.6 Subtipos, tipagem nominativa ou estrutural

Dizemos que um tipo τ é um *subtipo* de τ' quando não se perde a propriedade de *validade* ao se permitir que termos do tipo τ sejam usados em construções que esperam por termos do tipo τ'. Geralmente essa relação é denotada por $\tau <: \tau'$.

Existem linguagens em que a informação que identifica um tipo consiste no nome que é dado a ele durante sua definição. Este é o caso de linguagens como Java ou C++. Nessas linguagens, a relação de subtipagem deve ser declarada *explicitamente* pelo programador. Linguagens que possuem essas características tem um sistema de tipos *nominativo*.

Por outro lado, sistemas de tipos em que a estrutura dos tipos é usada para identificá-los e para estabelecer automaticamente a relação de subtipagem são ditos *estruturais*. Nada impede, entretanto, que um sistema de tipos estrutural permita que se dê nomes aos tipos, por praticidade; nesse caso, um nome não é mais do que uma forma conveniente de se referenciar um tipo, cuja identidade ainda é dada apenas por características estruturais.

2.2 Sistema de tipos Hindley-Milner

Para entender melhor os conceitos apresentados sobre sistemas de tipos e introduzir outros conceitos, vamos definir um sistema de tipos para uma linguagem puramente funcional L e mostrar um algoritmo de inferência de tipos para esse sistema. O sistema de tipos que será apresentado aqui é uma variação do chamado sistema *Hindley-Milner* de tipos.

Algoritmos de inferência de tipos para *lamda calculus* foram inicialmente estudados por Hindley[2]. Milner [3, 4] estendeu esse trabalho, adicionando o conceito de variáveis genéricas, possibilitando que construções *let* fossem *polimórficas* – daí o nome Hindley-Milner. Estensões e variações desse sistema de tipos são usadas por diversas linguagens de carácter funcional como OCaml, F# e Haskell.

O algoritmo de inferência de tipos que mostraremos aqui segue as idéias do algoritmo de Wand[5], que separa o processo em duas partes: a primeira gera equações contendo restrições sobre os tipos das expressões e a segunda resolve tais equações reduzindo-as para o problema da *unificação*

2.2.1 A linguagem L

A linguagem L é dada pela seguinte gramática:

$$\begin{aligned} e ::= & x | (ee') | \textbf{ if } e \textbf{ then } e' \textbf{ else } e'' | \lambda x.e | \\ & \textbf{pair}(e, e') | \textbf{ fst } e | \textbf{ snd } e | \\ & 0 | \textbf{ succ } e | \textbf{ pred } e | \textbf{ iszero } e | \textbf{ true } | \textbf{ false } | \\ & \textbf{let } x = e \textbf{ in } e' | \textbf{ fix } x.e \end{aligned}$$

(parênteses podem ser usados para evitar ambiguidades)

Nessa gramática, x representa qualquer identificador válido, $\lambda x.e$ uma função anônima, (ee') uma aplicação de funções. Há também operadores que lidam com pares, construindo-os (**pair**) e extraindo cada um de seus elementos (**fst** , **snd**). A construção **let** $x = e$ **in** e' é apenas uma abreviação para $((\lambda x.e')e)$. Por fim, a construção **fix** $x.e$ representa a aplicação do operador de *ponto fixo* sobre a função $\lambda x.e$ – essa construção só existe para possibilitar a definição de funções recursivas, tornando a linguagem Turing-completa. Não daremos muito atenção a essa construção ao definir as regras do sistema de tipos.

Não vamos definir formalmente as regras de avaliação de L pois variações de linguagens como essa são usadas e formalizadas exaustivamente em vários textos sobre o assunto.

2.2.2 Sistema de Tipos para L

Estados indesejáveis

Os estados indesejáveis S que queremos evitar em tempo de execução são aqueles em que não há como prosseguir usando alguma das regras de avaliação da

linguagem. Apesar de não termos definido essas regras, os estados S são óbvios: fazer uma aplicação em uma expressão que não avalia para uma função, usar como condição de **if** uma expressão que não avalia para **true** ou **false**, usar **fst** ou **snd** em uma expressão que não avalia para um par ou usar alguma dos operados **iszero** , **pred** , **succ** sobre expressões que não avaliam para números naturais.

Nosso sistemas de tipos só aceitará programas que, durante a execução, certamente nunca alcançam tais estados.

Tipos

Podemos definir, inicialmente, um **tipo** da seguinte maneira:

1. *Int* e *Bool* são tipos. (esses serão os tipos dos números naturais e das expressões booleanas, respectivamente)
2. α é um tipo, se $\alpha \in TV$, onde TV é um conjunto de *variáveis de tipo*.
3. Se τ_1 e τ_2 são tipos, então $\tau_1 \to \tau_2$ também é um tipo. (esse será o tipo de uma função)
4. Se τ_1 e τ_2 são tipos, então $\tau_1 \times \tau_2$ também é um tipo. (esse será o tipo de um par)

Usaremos os símbolos $\tau, \tau_i, \tau', \tau'', \ldots$ para designar tipos e as símbolos $\alpha, \beta, \gamma, \ldots$ para designar variáveis de tipos.

Tipos que não contêm variáveis são chamados de *tipos simples*, enquanto os que contêm são chamados de *politipos*. Variáveis de tipos representam tipos indefinidos, que podem ser substituídos por outros tipos. Como exemplo, gostaríamos de associar a expressão $\lambda x.$ **succ** (x) ao tipo *Int* $\to$ *Int*. Também gostaríamos de associar o tipo $\tau := (\textit{Int} \to \alpha) \to \alpha$ à expressão $\lambda f.(f0)$, uma vez que essa expressão poderia ser associada a qualquer tipo τ' obtido pela *substituição* de α por um outro tipo qualquer.

Quando obtemos um tipo τ' por meio da substituição de todas ocorrências de uma váriavel α de um tipo τ por algum outro tipo, dizemos que $\tau \geq \tau'$. Por exemplo, $\alpha \to \alpha \geq (\textit{Int} \to \beta) \to (\textit{Int} \to \beta) \geq (\textit{Int} \to \textit{Bool}) \to (\textit{Int} \to \textit{Bool})$.

Regras do sistema de tipos

Aqui, vamos definir o conjunto de regras que associações de termos de L a tipos devem satisfazer. Um *juízo de tipo* é uma asserção do tipo:

$\Gamma \vdash e : \tau$,

onde Γ é um conjunto de associações de identificadores à tipos, e é uma expressão de L e τ é um tipo. Uma asserção dessa simplesmente significa que: dadas as associações Γ, a expressão e tem tipo τ. As regras do sistema de tipos de L serão dadas por simples *inferências lógicas*, nas quais os predicados usados serão *juízos de tipos*:

$$\frac{}{\Gamma \vdash \textbf{true} : \textit{Bool}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textit{Bool}}$$

$$\frac{}{\Gamma \vdash 0 : \textit{Int}} \qquad \frac{\Gamma \vdash e : \textit{Int}}{\Gamma \vdash \textbf{succ}\ (e) : \textit{Int}}$$

$$\frac{\Gamma \vdash e : \textit{Int}}{\Gamma \vdash \textbf{pred}\ (e) : \textit{Int}} \qquad \frac{\Gamma \vdash e : \textit{Int}}{\Gamma \vdash \textbf{iszero}\ (e) : \textit{Bool}}$$

$$\frac{(\{x : \tau\} \in \Gamma)}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \Gamma_x \cup \{x : \tau_1\} \vdash e' : \tau_2}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash \textbf{pair}(e, e') : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \textbf{fst}\ (e) : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \textbf{snd}\ (e) : \tau_2} \qquad \frac{\Gamma \vdash e : \textit{Bool} \qquad \Gamma \vdash e' : \tau \qquad \Gamma \vdash e'' : \tau}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash ee' : \tau_2}$$

Nessas regras, Γ_x denota o conjunto de associações Γ, excluindo uma possível associação ao identificador x. A expressão $\Gamma \cup \{x : \tau\}$ denota Γ acrescido da associação de x ao tipo τ.

2.2.3 Inferência de tipos

Aqui, mostraremos um forma de se inferir automaticamente os tipos de expressões de L (de forma que as regras descritas anteriormente sejam satisfeitas) sem que seja necessário que o programa contenha alguma anotação de tipos.

Para cada subexpressão e do programa de entrada, introduziremos uma variável de tipo – denotada por $\tau(e)$ – que corresponde ao tipo, ainda desconhecido, associado a e. Diretamente a partir das regras do sistema de tipos, podemos construir uma série de *restrições* à esses tipos, sobre a forma de equações. Mostraremos abaixo algumas dessas restrições:

- A expressão 0 gera a restrição: $\tau(0) = \textit{Int}$.
- A expressão $\lambda x.e$ gera a restrição: $\tau(\lambda x.e) = \tau(x) \to \tau(e)$.
- A expressão **if** e **then** e' **else** e'' gera as restrições:
  $\tau(\textbf{if } e \textbf{ then } e' \textbf{ else } e'') = \tau(e'') = \tau(e'), \tau(e) = \textit{Bool}$.
- A expressão ee' gera as restrição: $\tau(e) = \tau(e') \to \tau(ee')$.
- etc.

Esse conjunto de restrições pode ser visto como uma instância do conhecido *problema da unificação*

Problema da unificação

O problema da unificação recebe como entrada dois *termos* e devolve uma *substituição* que torna iguais ambos os termos. Um *termo* é definido como uma constante ($a, b, c, \dots$) ou uma variável ($x, y, z, \dots$) ou uma aplicação de função ($f, g, h, \dots$) sobre um termo. Uma *substituição* é um função que mapeia variáveis para termos.

Por exemplo, podemos unificar $f(a, y)$ com $f(x, g(b, x))$ usando uma substituição que associa y a $f(b, a)$ e x a a. Uma substituição não precisa, necessariamente, associar um termo a cada variável – só é necessário que ambos os termos sejam idênticos após a substituição. Existem unificações que não podem ser feitas, como por exemplo entre os termos $f(a)$ e $g(a)$ (pois f e g são funções distintas) ou entre x e $f(x, a)$.

Podem existir diversas substituições distintas que unificam dois termos. O problema da unificação consiste, na verdade, em encontrar a substituição *mais geral possível* S, que tem a propriedade de que qualquer outra substituição S' que resolve o problema é o resultado da composição de S com alguma outra substituição. O problema da unificação pode ser resolvido em tempo linear no tamanho da entrada [7].

O problema da unificação (e suas soluções) pode ser facilmente estendido para trabalhar com unificações de vários termos, dois a dois, num sistema de equações.

Redução para o problema da unificação

Podemos ver o problema de resolver as restrições de tipos como um problema da unificação da seguinte forma: cada variável de tipo é uma variável do problema da unificação. Cada tipo primitivo (*Int* e *Bool*) é uma constante do problema da unificação. Por fim, os operadores $\rightarrow$ e $\times$ podem ser vistos como funções binárias entre termos.

Uma vez obtida a substituição solução do problema da unificação, basta aplicá-la a cada um dos termos, obtendo, assim, os tipos de cada uma das expressões. Caso não exista solução para o problema da unificação, então o programa de entrada não pode ser tipado, isto é, deve ser rejeitado pelo sistema de tipos.

Exemplo

Vamos analisar um exemplo simples, de como a expressão $\lambda f.f0$ teria seus tipos inferidos. Essa expressão gera o seguinte conjunto de restrições:

- $\tau(\lambda f.f0) = \tau(f) \rightarrow \tau(f0)$
- $\tau(f) = \tau(0) \rightarrow \tau(f0)$
- $\tau(0) = \textit{Int}$

A unificação desse sistema de restrições, dará como resultado a substituição $\tau(0) \mapsto \textit{Int}$, $\tau(f) \mapsto \textit{Int} \Rightarrow \tau(f0)$, $\tau(\lambda f.f0) \mapsto (\textit{Int} \rightarrow \tau(f0)) \rightarrow \tau(f0)$. Ou seja, a expressão toda possui o tipo $(\textit{Int} \rightarrow \alpha) \rightarrow \alpha$, como desejado.

2.2.4 Variáveis genéricas de tipo - let com polimorfismo

Considere a seguinte expressão:

let $f = \lambda x.x$ **in pair**$(f(0), f(true))$

Apesar de ser perfeitamente válida, essa expressão não pode ser tipada usando o algoritmo de inferência de tipos que descrevemos. Isso acontece porque o conjunto de restrições gerado exigirá que f aceite ao mesmo tempo um argumento do tipo primitivo *Int* e *Bool*.

(continua versão final)

Capítulo 3

Linguagem XBA

A linguagem XBA foi desenvolvida com o intuito de colocar em prática o estudo feito sobre sistemas de tipos, inferência de tipos e desenvolvimento de linguagens de programação.

Nessa seção, descrevemos a linguagem de maneira informal, citando suas principais características (seção 3.1) e dando alguns exemplos de programas válidos da linguagem (seção 3.2). Essa seção tem o objetivo de familiarizar o leitor com a linguagem, para que este possa entender o sistema de tipos utilizado, assim como as dificuldades envolvidas no processo de desenvolvimento de um compilador para essa linguagem, sem seja necessária a leitura completa da especificação da linguagem (seção 4), que deve servir apenas como referência.

Ainda nesse capítulo, discutimos em 3.3 o porquê da escolha da JVM como alvo do compilador desenvolvido.

3.1 Características

As principais características da linguagem desenvolvida são listadas a seguir:

3.1.1 Suporte à orientação a objetos

A linguagem XBA tem suporte *parcial* à orientação a objetos, assim como várias linguagens existentes atualmente, como C++, Java, C#, Ruby, etc. Ao contrário de Java, por exemplo, não existe o conceito de primitivas. Toda variável guarda uma referência para um objeto que pode receber mensagens (chamadas de método). Da mesma forma, qualquer literal (um número, uma string, um caracter)

representa um objeto que também pode receber mensagens.

O suporte é *parcial* pois não usaremos o conceito de herança, considerada muitas vezes como uma condição necessária para que uma linguagem seja orientada a objetos. Um comportamente similar à herança pode ser emulado através da delegação de mensagens.

3.1.2 Funções como objetos de primeira classe

Assim como em diversas linguagens que possuem caracter funcional, tais como Haskel, ML, Scala, F#, etc funções são *objetos de primeira classe*. Isso significa que:

- funcões podem ser passadas como argumento de outras funções (ou métodos).
- O valor de retorno de uma função ou método pode ser uma função.
- É possível criar *funções anônimas*.

Na verdade, para a linguagem XBA, uma função é definida como um objeto que possui um método chamado `apply`.

3.1.3 Aplicação parcial de métodos (*currying*)

Assim como em diversas linguagens funcionais, é possível fazer a aplicação parcial de argumentos em qualquer função ou método. Isso significa que se uma função possui dois parâmetros e a chamarmos enviando apenas um deles, o resultado será uma função que aceita o segundo parâmetro.

3.1.4 Sistema de tipos estrutural

A linguagem XBA infere *tipos estruturais* à todas as expressões, acusando inconsistências de tipo. O tipo de cada objeto, consiste, basicamente, no conjunto de mensagens aceitas por este. Isso significa que erros comuns, como passar um argumento 'errado' à um método, são muitas vezes acusados pela linguagem, que, ainda assim, mantém boa parte da flexibilidade existente em linguagens dinâmicas.

3.1.5 Uso de seletores para chamar métodos em objetos

Assim como em Smalltalk, as mensagens são passadas a objetos por meio de seletores. Apesar de ser apenas um detalhe sintático, essa característica torna algumas chamadas de métodos com vários parâmetros mais agradáveis de se ler, uma vez que cada argumento aparece entre um seletor correspondente. Por exemplo, uma chamada do tipo:

- `cliente comprar-produto:p usando-desconto:d`

é possivelmente mais legível do que:

- `cliente.comprar-produto-usando-desconto(p, d).`

3.1.6 Uso de identação para definir os blocos da linguagem

Assim como Python, a linguagem XBA usa a própria indentação do programa para definir blocos de código, ao invés de usar chaves (como, por exemplo, Java e C/C++ fazem) ou palavras reservadas (como Pascal faz, usando **begin** e **end**). Esse também é um detalhe meramente sintático, mas que pode melhorar a legibilidade de um programa pela diminuição de ruído.

3.2 Exemplos

A seguir listamos alguns exemplos de programas para essa linguagem. Os exemplos são analisados para mostrar o funcionamento da linguagem mais a fundo:

3.2.1 HelloWorld

```
1 class Hello
2     method entry-with-arguments: args
3         print "Hello␣World"
```

Esse é um exemplo de um programa que apenas escreve o texto *Hello World* na tela. O método com o nome especial **entry-with-arguments** marca que esse é um ponto de entrada válido de um programa. Outro ponto importante a ser notado é que **print** não é uma palavra reservada da linguagem, mas sim uma função (um objeto com um método apply que pode ser, portanto, usado como uma função) capaz de imprimir uma string. A aplicação de funções se dá por justaposição e é associativa à esquerda.

3.2.2 Construtores e métodos

```
1  class Vector
2      field x
3      field y
4
5      constructor with-x:px with-y:py
6          x ← px
7          y ← py
8
9      method sqnorm!
10         return x*x + y*y
11
12     method +:v2
13         var nx, ny
14         nx ← x + v2 x!
15         ny ← y + v2 y!
16         return Vector new-with-x:px with-y:py
17
18     method -:v2
19         var tmp
20         tmp ← Vector new-with-x: (x - v2 x!)
21         return tmp with-y: (y - v2 y!)
22
23     method x!
24         return x
25     method y!
26         return y
```

Nesse exemplo um pouco mais completo de um classe (um vetor bidimensional) já é possível entender como podemos definir novas classes e métodos em XBA.

Campos, que são as variáveis que guardam o estado de um objeto, são definidos pela palavra reservada **field** (linhas 2 e 3). Membros são imutáveis por padrão, mas esse comportamente pode ser alterado usando o modificador **mutable**.

A palavra reservada **construtor** (linha 5) denota o início da descrição de um método *da classe*[1], capaz de instanciar objetos do tipo Vector. O nome do mé-

[1]O nome de uma classe representa um objeto que aceita apenas mensagens que chamam construtores, ou seja, que instanciam objetos daquela classe

todo que será gerado será precedido por **new-** para indicar que se trata de um construtor – a linha 16 mostra um exemplo de chamada a esse construtor.

Nas linhas 6 e 7 vemos que a atribuição à variáveis se dá usando o caracter unicode ⟵. O uso de caracteres unicode facilita a leitura do código e não introduzem uma grande dificuldade na escrita de código, desde que se use um editor configurado apropriadamente.

O método **sqnorm!** é um método que não aceita parâmetros (indicado pelo ponto de exclamação) e devolve a norma ao quadrado do vetor em questão[2].

Na linha 10, vemos que a linguagem aceita o uso de operadores. Operadores nada mais são do que métodos especiais que possuem uma sintaxe infixa (a precedência de operadores é a precedência que se espera em operações matemáticas). A própria classe Vector define dois operadores (+ e -).

No corpo dessas definições, é possível notar que variáveis locais de métodos são declaradas por meio da palavra reservada **var**. (varíaveis, assim como membros, também são imutáveis, a menos que se adicione o modificador **mutable**).

Por fim, o método '-' foi escrito de maneira diferente, para exemplificar a aplicação parcial de métodos. Aplicamos parcialmente o método **new-with-x:with-y:**, usando apenas o primeiro seletor. O resultado dessa aplicação é um objeto que possui o método **with-y:** (e somente esse método) e termina a chamada. Apesar de exemplificado com um construtor, a aplicação parcial de métodos pode ser feita em qualquer método que possui mais de um parâmetro. A rigor, todos os métodos da linguagem possuem exatamente um parâmetro. Fazer uma chamada de método de dois parâmetros $p1$ e $p2$ consiste, na verdade, em fazer duas chamadas de método (uma com $p1$ e a outra com $p2$).

3.2.3 Estruturas de controle

O exemplo a seguir mostra uma classe contendo um método que calcula o máximo divisor comum entre dois inteiros.

```
1 class Mdc {
2     method between:x and:y
3         mutable var dividendo, divisor, resto
4         dividendo ← x
5         divisor ← y
```

[2]Na realidade métodos desse tipo aceitam um único parâmetro da classe especial **Unit** que é um classe com apenas uma instância (**unit**) – essa classe é análoga ao **void** das linguagem C,C++, Java.

```
6           while (dividendo % divisor  0) do:
7               resto ← dividendo % divisor
8               dividendo ← divisor
9               divisor ← resto
10          return divisor
11
12      export mdc ← Mdc new
13  }
```

Esse classe contém um método que calcula o mdc entre **x** e **y** usando o algoritmo de Euclides. A linha 12 exporta uma instância padrão dessa classe, chamada **mdc**, de tal forma que, uma vez que essa definição é importado, é possível fazer uma chamada do tipo **mdc between:5 and:3**.

O importante a ser notado aqui é que **while** não é uma construção da linguagem XBA – a linguagem não possui nenhuma construção para fazer condicionais (**if**) ou laços (**while**, **for**, etc).

A identificador **while** que aparece na linha 6 desse exemplo é um objeto (automaticamente importado) implementado na 'biblioteca padrão' da linguagem. Esse objeto possui um método **apply:do:** que recebe uma condição e um bloco de código, executando esse último enquanto a condição permanecer verdadeira.

Em uma chamada de método comum, a expressão E :=**dividendo % divisor ≠ 0** seria avaliada e seu resultado seria enviado ao **while**. Entretanto, isso não ocorre aqui – o que é passado para o **while** é uma *closure* – uma função que captura as variáveis **dividendo** e **divisor** e, sempre que avaliada, devolve o resultado da expressão E. É esse o processo que torna possível a existência de um objeto como **while** que imita o comportamento de uma construção que geralmente está incorporada às linguagens de programação usuais.

No próximo exemplo, construiremos uma nova estrutura de controle similar ao **while**, com o objetivo de enunciar qual é a condição necessária para que um método avalie seus argumentos dessa maneira *especial* descrita acima.

3.2.4 Definindo novas estruturas de controle

Nesse exemplo, construiremos uma nova estrutura de controle chamada **until** que funcionará de maneira similar ao **while**, com a diferença que o bloco de código será executado enquanto a condição for **falsa**.

```
1   class Until
2       method apply:condition do:code
```

```
3           while (condition execute! negated!) do:
4               code execute!
5
6       export until ← Until new
```

O primeiro seletor do método dessa classe se chama **apply:**, fazendo com que o primeiro argumento (a condição) dessa estrutura de controle possa ser passado por justaposição. O segundo argumento, que é o bloco que será executado a menos que a condição seja verdadeira, é obtido a partir do seletor **do:**. A linha X é responsável por instanciar um objeto dessa classe, com o nome **until**, e exportá-lo. Dessa forma, qualquer classe que o importe poderá usar a construção **until** da mesma forma que a construção **while** é usada.

Como que a linguagem decide que os argumentos passados para esses métodos não devem ser avaliados no momento da aplicação, isto é, devem ser passados como uma *closure*? Isso ocorre pois o sistema de tipos estruturais infere que os argumentos passados para o método devem aceitar a mensagem especial **execute!**. Neste caso, no momento da chamada, as expressões que seriam avaliadas e passadas como argumento são transformadas em *closures* que são objetos que capturam o ambiente atual e possuem um método **execute!**.

3.3 JVM como alvo

A Java Virtual Machine (JVM) é a máquina virtual escolhida como alvo do compilador da linguagem XBA, isto é, o compilador compila os programas escritos em XBA para instruções aceitas pela JVM. Foram três os motivos principais que motivaram essa escolha.

Primeiro, a JVM é uma máquina virtual que possui implementações para diversos sistemas e diversas arquiteturas e é amplamente usada por diversas aplicações. Isso significa que, ao se fazer um compilador que tem a JVM como alvo, obtêm-se programas que podem ser usados nas mais diversas plataforma.

Segundo, é mais fácil fazer um compilador que tem a JVM como alvo, do que um que tem uma máquina real como alvo, principalmente quando a linguagem em questão deve fornecer suporte à orientação objeto. Isso se deve a diversas características presentes na JVM, dentre elas:

- Essa máquina virtual já trabalha com o conceito de objetos naturalmente. Em particular, existem instruções que instanciam objetos, chamam construtores, fazem chamadas de métodos, etc.

- A própria JVM já possui um *garbage-collector* que libera a memória usada na alocação de objetos e arrays que não estão mais sendo usados. Dessa forma, o compilador não precisa oferecer um *runtime* que gerencie a memória usada pelos programas, uma vez que esse gerenciamento já é feito pela máquina virtual.
- Não é tão importante que se gere um código muito otimizado para a JVM. Na implementações mais usadas da JVM, o compilador *just-in-time* é responsável por efetuar diversas otimizações, muitas delas agressivas, uma vez que, em tempo de execução, se tem acesso a maiores informações sobre o programa.

Por último, já existe um número imenso de bibliotecas que rodam em cima da JVM. Isso possibilita que linguagens que rodem em cima dessa máquina virtual forneçam algum mecanismo de interoperabilidade, tornando possível o uso dessas bibliotecas pelos usuários dessas linguagens.

Capítulo 4

Especificação da linguagem XBA

Talvez aqui não colocaremos uma especificação propriamente dita, apenas a gramática da linguagem e maiores detalhes sobre ela. Uma especificação completa gastaria muito espaço.

Capítulo 5

Sistemas de tipos da linguagem XBA

5.1 Descrição do sistema de tipos

5.2 Descrição do algoritmo de inferência utilizado

Capítulo 6

Implementação

6.1 Introdução

6.2 Análise léxica e sintática

6.2.1 Explicação...

6.2.2 Javacc

6.3 Geração de código

6.3.1 Explicação...

6.3.2 ASM

Capítulo 7

Conclusão

Referências Bibliográficas

[1] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2nd Edition, 2002.

[2] J.R. Hindley. *The principal type-scheme of an object in combinatory logic*. Transactions of the American Mathematical Society 146 pp. 29-60, 1969.

[3] Robin Milner. *A theory of polymorphism in programming*. JCSS 17,3, pp.348-375, 1978

[4] Luis Damas, Robin Milner. *Principal type-schemes for functional programs*. Proceedings of the 9th Annual Symposium on Principles of Programming Languages (Albuquerque, N. Mex., Jan. 25-27). ACM, New York, pp. 207-212, 1982

[5] Mitchell Wand. *A Simple Algorithm and Proof for Type Inference*. Fundamenta Infomaticae ,10: pp.115-122, 1987

[6] Michael I. Schwartzbach. *Polymorphic type inference*. Technical Report BRICS-LS-95-3, BRICS, June 1995

[7] M. S. Paterson, M. N. Wegman. *Linear unification*, Proceedings of the eighth annual ACM symposium on Theory of computing, p.181-186, May 03-05, 1976, Hershey, Pennsylvania, United States.

[8] Tim Lindholm, Frank Yellin. *The Java Virtual Machine Specification*, Second Edition. Prentice Hall, April 1999.

[9] Eric Bruneton. *ASM 3.0: A Java bytecode engineering library*. `http://download.forge.objectweb.org/asm/asm-guide.pdf`, February 2007.

Apêndice A

Parte Subjetiva

A.1 Dificuldades e desafios encontrados

A.2 Disciplinas relevantes para este trabalho

A.3 Continuação deste trabalho