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 tipos 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 Hindley-Milner de tipos 7
2.2.1 A linguagem L 8
2.2.2 Sistema de Tipos para L 9
2.2.3 Inferência de tipos 11
2.2.4 Variáveis genéricas de tipo (*let-polimorfismo*) 13

3 Linguagem XBA **15**
3.1 Características 15
3.1.1 Sistema de tipos estrutural 15
3.1.2 Suporte a orientação a objetos 16
3.1.3 Suporte a *currying* em todos os métodos 16
3.1.4 Funções como objetos de primeira classe 16
3.1.5 Uso de seletores para chamar métodos em objetos 16
3.1.6 Uso de indentação para definir os blocos da linguagem . . 17
3.2 Exemplos 17

3.2.1 HelloWorld 17
3.2.2 Construtores e métodos 18
3.2.3 `if` e instâncias padrões 19
3.3 JVM como alvo 20
3.4 Outras características 21
3.5 Gramática da linguagem 22

4 Sistema de tipos da linguagem XBA 24
4.1 Descrição do sistema de tipos 24
4.1.1 Objetivo 24
4.1.2 Os tipos 25
4.1.3 Exemplo de tipos 26
4.1.4 Regras de tipos 28
4.1.5 Variáveis mutáveis 28
4.2 Inferência de tipos na linguagem XBA 28
4.2.1 Unificação 29
4.2.2 A inferência em si 30

5 Implementação 31
5.1 Análise léxica e sintática 31
5.1.1 JavaCC 32
5.2 Análise semântica, inferência de tipos e geração de código . . . 32
5.2.1 ASM 33
5.2.2 Alguns detalhes de implementação 34
5.3 Deficiências do protótipo desenvolvido 35

6 Considerações finais 37

A Parte Subjetiva 41
A.1 Dificuldades e desafios encontrados 41
A.2 Disciplinas relevantes para este trabalho 42
A.3 Continuação na área 43

Capítulo 1

Introdução

1.1 Tema

Linguagens com *tipagem dinâmica* têm sido alvo de grande atenção nos últimos anos. Por não se basear 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 depuração. 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) — nas quais 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

[1]Sistemas estáticos nominativos são usados em linguagens como C, Pascal, Java, entre outras.

tão 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 protótipo de 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 conceitos sobre sistemas de tipos, apresentando, também, uma variação do sistema de tipos *Hindley-Milner* e um algoritmo de inferência para ele, usando uma linguagem simples como exemplo. No capítulo 3, é dada uma introdução a linguagem XBA, por meio de exemplos. No capítulo 4, apresentamos o sistema de tipos que usaremos na linguagem e o algoritmo de inferência utilizado. Por fim, discutimos alguns detalhes de implementação no capítulo 5.

[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 tipos

Programas de computador podem, muitas vezes, conter erros - ou introduzidos 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 irrecuperáveis, 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 são *sistemas de tipos*.

Um sistema de tipos[1] pode ser definido como um método que analisa sintaticamente um programa e prova a ausência de certos comportamentos indesejáveis, por meio da classificação dos *termos*[1] do programa 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

[1]Um termo é definido como uma construção sintática que, em tempo de execução, pode ser avaliada para um valor

que t é *bem-tipado*. Essa associação é feita respeitando uma série de *regras de tipagem* (*typing rules*) que, para cada construção da linguagem, determina o tipo da expressão associada, com base nos tipos de suas sub-expressões. 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 é avaliado para 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. A rigor, pela definição apresentada anteriormente, essas linguagens não têm um sistema de tipos (ou têm 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, vamos usá-la durante o texto.

2.1.5 Inferência de tipos

Dizemos que linguagens são *tipadas explicitamente* quando se exige 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 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 Hindley-Milner de tipos

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 *lambda calculus* foram inicialmente estudados por Hindley[3]. Milner [4, 5] estendeu esse trabalho, adicionando o conceito de variáveis genéricas, possibilitando que construções *let* fossem *polimórficas* — daí o nome Hindley-Milner. Extensões e variações desse sistema de tipos são usadas por diversas linguagens de caráter funcional, como OCaml, F# e Haskell.

2.2.1 A linguagem L

A linguagem L é dada pela seguinte gramática:

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

(parênteses podem ser usados para evitar ambiguidades)

Nessa gramática, x representa qualquer identificador válido, $\lambda x.e$ uma função anônima, (ee') uma aplicação de função. Há também uma estrutura de controle condicional (**if**) e 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 cujo valor não é uma função, usar como condição de **if** uma cujo valor não é **true** ou **false**, usar **fst** ou **snd** em uma expressão cujo valor não é um par ou, ainda, usar algum dos operadores **iszero** , **pred** , **succ** sobre expressões cujos valores não são números naturais.

Nosso sistema de tipos só aceitará programas que, durante a execução, certamente nunca alcancem 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 *type variables* [2] — uma variável que representa um tipo indeterminado.

3. Se τ_1 e τ_2 são tipos, então $\tau_1 \rightarrow \tau_2$ 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 *type variables*.

Tipos que não contêm *type variables* 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* $\rightarrow$ *Int*. Também gostaríamos de associar o tipo $\tau := (\mathit{Int} \rightarrow \alpha) \rightarrow \alpha$ à expressão $\lambda f.(f\ \mathbf{0})$, uma vez que essa expressão poderia ser associada a qualquer tipo τ' obtido pela *substituição* de α por um outro tipo qualquer.

[2] Não encontramos uma tradução apropriada — “variável de tipo” não parece ser uma boa opção — e portanto usaremos a expressão em inglês.

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

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 dessas 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*:

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

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

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

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

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

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

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

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

Vamos descrever, aqui, um processo algorítmico capaz de *inferir* os tipos de um programa escrito na linguagem L. O processo de *inferência* consiste em determinar o tipo para cada um dos termos de L, de tal forma que esses tipos satisfaçam as regras previamente descritas. Caso isso não seja possível, o processo de inferência deve acusar um erro.

Para sistemas de tipagem que tem como base o sistema Hindley-Milner, a inferência de tipos geralmente recai no conhecido problema da *unificação*.

Problema da unificação

O problema da unificação recebe como entrada dois *termos lógicos* e devolve uma *substituição* que torna iguais ambos os termos. Uma *substituição* é um função que mapeia variáveis para termos. Um *termo lógico* é definido recursivamente como:

- uma *constante* $(a, b, c, \ldots)$
- uma *variável* $(x, y, z, \ldots)$
- uma aplicação de *função* $(f, g, h, \ldots)$ sobre um termo.

.

Por exemplo, podemos unificar $f(a, y)$ com $f(x, g(b, x))$ usando uma substituição que mapeia y para $f(b, a)$ e x para 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 pode ser obtida pela composição de S^* com alguma outra substituição. O problema da unificação foi estudado originalmente em [7]; existem algoritmos que o resolvem em tempo linear no tamanho da entrada [14].

O problema da unificação no contexto de sistemas de tipos

O problema da unificação pode ser "traduzido" da seguinte forma para o contexto dos tipos definidos para a linguagem L:

- Estaremos interessados na unificação de dois *tipos*, ao invés da unificação de dois *termos lógicos*.
- Os tipos pré-definidos *Bool* e *Int* farão o papel das *constantes*.
- As *type variables* faram o papel das *variáveis*.
- Os construtores de tipos $\rightarrow$, $\times$ farão o papel das *funções* (infixas).

Inferência de tipos em L

Uma vez definida a unificação de tipos, inferir o tipo de uma expressão é um processo simples que, essencialmente, infere os tipos das sub-expressões recursivamente, unificando-os com tipos que contém a estrutura esperada, de acordo com as regras de tipagem.

A seguir, descrevemos mais especificamente o processo que infere o tipo de uma expressão e, para algumas das principais construções da linguagem. O processo é similar para as demais construções de L.

- Se e é da forma $\lambda x.e'$, associe x a uma nova *type variable*, denominada α. Infira o tipo τ da expressão e' (usando a associação feita a x). Seja τ' o tipo associado a α (por possíveis substituições decorrentes de unificações) após a inferência de e'. Devolva $\tau' \rightarrow \tau$.
- Se e é da forma $(e_1 e_2)$, infira os tipos τ_1 e τ_2 de e_1 e e_2, respectivamente. Unifique τ_1 com o tipo $\tau_2 \rightarrow \alpha$, onde α é uma nova *type variable*. Se a unificação falhar, a expressão contém um erro de tipos. Caso contrário, devolva o tipo associado a α, após a unificação.
- Se e é da forma **let** $x = e_1$ **in** e_2, associe x a uma nova *type variable*, denominada α. Infira o tipo τ de e_1, e associe-o ao identificador x. Devolva o tipo inferido (usando a associação feita a x) de e_2.
- Se e é da forma **fst** e', infira o tipo de e' e unifique-o com o tipo $\alpha \times \beta$, no qual α e β são novas *type variables*. Devolva o tipo associado a α, após a unificação.

- Se e é da forma **if** e_1 **then** e_2 **else** e_3, infira os tipos τ_1, τ_2, τ_3 das expressões e_1, e_2 e e_3, respectivamente. Unifique τ_1 com *Bool* e τ_2 com τ_3. Caso alguma das unificações falhe, a expressão contém um erro de tipos. Caso contrário, devolva o tipo associado a τ_2 após a unificação.

- Constantes como 0, **true**, **false** são associadas aos tipos correspondentes.

Na prática, o processo de inferência guarda e manipula uma coleção de associações entre identificadores da linguagem L e tipos.

Exemplo

Vamos analisar um exemplo simples, de como a expressão $e := \lambda f.f0$ teria seus tipos inferidos:

- Primeiramente, associa-se f a uma nova *type variable* denominada α e infere-se o tipo da subexpressão $(f0)$.

- Infere-se recursivamente o tipo de f ($= \alpha$) e o tipo de 0 (= *Int*). Unifica-se α com *Int* $\rightarrow \beta$, onde β é uma nova *type variable.* (Essa unificação consiste simplesmente na substituição $\alpha \mapsto$ *Int* $\rightarrow \beta$). Finalmente, o tipo inferido da expressão $(f0)$ é β.

- Por fim, o tipo inferido de e deve ser $\tau \rightarrow \beta$, onde τ é o tipo associado a α. Ou seja, como esperado, o tipo inferido de e é dado por $(Int \rightarrow \beta) \rightarrow \beta$.

2.2.4 Variáveis genéricas de tipo (*let-polimorfismo*)

Considere a seguinte expressão:

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

Embora essa expressão seja perfeitamente válida, ela não pode ser tipada pelo algoritmo de inferência de tipos que descrevemos: após termos associado f o tipo $\alpha \rightarrow \alpha$, em algum momento, contudo, tentaríamos unificar α com *Int* e com *Bool*, o que não seria possível. O sistema *Hindley-Milner* de tipos resolve esse problema usando o chamado *let-polimorfismo*: em certas condições, *type variables* são associadas ao quantificador $\forall$; por exemplo o tipo da função f desse exemplo seria $\forall \alpha.\alpha \rightarrow \alpha$.

Quando uma *type variables* qualificada dessa forma estiver associada ao identificador de uma construção **let**, deve-se criar uma nova *type variables*, para *cada ocorrência* desse identificador no corpo do **let**. Esse procedimento exige cuidados

especiais para que a inferência de tipos não se torne muito ineficiente. Por simplicidade, o sistema de tipos da linguagem XBA não dará suporte a *let-polimorfismo*; assim sendo, não descrevemos essa característica do sistema *Hindley-Milner* com mais detalhes.

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, apresentando 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 ela.

Ainda nesse capítulo, discutimos em 3.3 o porquê da escolha da JVM como alvo do compilador desenvolvido. Finalmente, em 3.5, apresentamos uma listagem da gramática da linguagem.

3.1 Características

As principais características da linguagem desenvolvida são discutidas brevemente a seguir:

3.1.1 Sistema de tipos estrutural

A linguagem XBA infere *tipos estruturais* a 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.

3.1.2 Suporte a orientação a objetos

A linguagem XBA tem suporte a orientação a objetos, assim como várias linguagens existentes atualmente, como C++, Java, C#, Ruby, etc. Em XBA não existe o conceito (presente em Java e em C++) de tipo "básico" ou "primitivo", cujas instâncias não são objetos. Todo identificador guarda uma referência para um objeto que pode receber mensagens (chamadas de método). Da mesma forma, qualquer literal (um número, uma string, um caráter) representa um objeto que também pode receber mensagens.

O suporte a objetos em XBA 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 comportamento similar a da herança pode ser emulado através por meio de delegação de mensagens — e ao se delegar mensagens, a subtipagem vem de graça, uma vez que a linguagem possui tipagem estrutural.

3.1.3 Suporte a *currying* em todos os métodos

Assim como em diversas linguagens funcionais, todos os métodos/funções possuem *currying*. Isso significa que se uma função possui dois parâmetros e a chamamos fornecendo apenas um deles, o resultado será uma função que aceita o segundo parâmetro.

3.1.4 Funções como objetos de primeira classe

Em XBA, uma função é definida como um objeto que possui um método chamado `apply`. Dessa forma, assim como em linguagens com caráter funcional (tais como Haskel, ML, Scala, F#, etc), funções podem ser recebidas como parâmetros, devolvidas por outros métodos/funções, etc.

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

mais agradável a leitura de algumas chamadas de métodos com vários parâmetros, uma vez que cada argumento aparece logo depois do 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 indentação 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 import print from br/usp/ime/mac499/xba/runtime/XBAPrinter
2
3 class Hello
4     method init! =
5         print "Hello␣World"
```

Esse é um exemplo de um programa que apenas escreve o texto *Hello World* na tela. Em XBA, não existe um método especial que serve como ponto de entrada de um programa. Ao se executar `xba HelloWorld`, uma instância da classe `HelloWorld` é criada. Caso tenham sido passados argumentos pela linha de comando, esses argumentos seriam usados como parâmetros para o *construtor* da classe. (O próximo exemplo mostrará uma classe que possui um construtor com parâmetros).

O método com o nome especial `init!` é um método que é automaticamente chamado após um objeto ser construído. O ponto de exclamação indica que o método não recebe parâmetros.

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 , portanto, ser usado como uma função) capaz de imprimir uma string. A aplicação de funções se dá por justaposição e é associativa à esquerda. A primeira linha do programa importa essa função, de forma que seja possível utilizá-la.

3.2.2 Construtores e métodos

```
1  class Vector with-x:px and-y:py
2      field x = px
3      field y = py
4
5      method sqnorm! = x*x + y*y
6
7      method +:v2 =
8          let nx = x + v2 x!, ny = y + v2 y! in
9              Vector new-with-x:nx and-y:ny
10
11     method -:v2 =
12         let tmp = Vector new-with-x: (x - v2 x!) in
13             tmp and-y: (y - v2 y!)
14
15     method x! = x
16
17     method y! = y
```

Nesse exemplo um pouco mais completo de uma classe (um vetor no plano) já é possível entender como podemos definir novas classes e métodos em XBA.

Na primeira linha do programa, é declarada uma classe, cujo construtor possui dois parâmetros (px e py). Isso significa que instâncias dessa classe podem ser construídas mandando a mensagem `new-with-x:and-y:` (note que um `new-` deve ser adicionado ao primeiro seletor) para a classe `Vector` — o nome de uma classe representa um objeto que aceita apenas a mensagem que cria instâncias daquela classe. Veja um exemplo de instanciação na linha 9.

Campos, que são as variáveis que guardam o estado de um objeto, são definidos pela palavra reservada `field` (linhas 2 e 3). Esses campos *sempre* devem ser

inicializados e são imutáveis.

O método `sqnorm!` é um método que não aceita parâmetros[1](indicado pelo ponto de exclamação) e devolve a norma ao quadrado do vetor em questão. O valor devolvido por um método consiste na expressão que vem após o sinal de igual. Uma expressão também pode ser um bloco de expressões. Blocos são definidos pela indentação e representam sequências de expressões; o valor de um bloco é o valor da última expressão nele contida.

Na linha 7, 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 associações de identificadores a expressões por meio da construção `let...in`, muito comum em diversas linguagens.

Por fim, o método '`-`' foi escrito de maneira diferente, para exemplificar que métodos possuem *currying*. Aplicamos 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), que 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 com dois parâmetros $p1$ e $p2$ consiste, na verdade, em fazer duas chamadas de método (uma com $p1$ e a outra com $p2$ no objeto resultante da primeira chamada).

3.2.3 `if` e instâncias padrões

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

```
1  import print from br/usp/ime/mac499/xba/runtime/XBAPrinter
2
3  class Mdc
4      method between:x and:y =
5          if (y = 0) then x
```

[1]Na realidade métodos com nome terminado por `!` aceitam um único parâmetro da classe especial `Unit`, que é um classe com apenas uma instância (`unit`) — essa classe desempenha papel análogo ao do tipo **void** das linguagem C,C++, Java.

```
6         else
7            self between:y and: (x%y)
8
9    export mdc = Mdc new!
```

Esse classe contém um método que calcula o mdc entre `x` e `y` usando o algoritmo de Euclides. Note que a linguagem possui a construção `if`, que funciona de maneira similar a construções similares em linguagens funcionais. Observe que o identificador especial `self` é uma referência para a instância que recebeu a chamada de método.

A linha 9 *exporta* uma instância padrão dessa classe, chamada `mdc`. Isso significa que outras classes podem *importar* essa definição, usando a instância `mdc` diretamente, sem a necessidade de chamar o construtor da classe *Mdc*. Desta forma, chamadas a esse objeto podem ser feitas de maneira bem conveniente, como por exemplo: `mdc between:5 and:3`. A função `print` do primeiro exemplo, é, também, uma instância exportada por uma classe `XBAPrint`.

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 para 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 plataformas.

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 a objetos. 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 possui um *garbage-collector* que libera a memória ocupada por 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. Nas implementações mais usadas da JVM, o compilador *just-in-time* (JIT) 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.

- A versão 7 da JDK possui uma nova instrução *invokedynamic*, acompanhada de novos mecanismos, que facilitam o desenvolvimento de linguagens com caráter dinâmico para a JVM.

Por último, já existe um número imenso de bibliotecas que rodam sobre a JVM. Isso possibilita que linguagens que são executadas nessa máquina virtual forneçam algum mecanismo de interoperabilidade, tornando possível o uso daquelas bibliotecas pelos usuários dessas linguagens.

3.4 Outras características

Uma especificação completa da linguagem XBA gastaria muito espaço e não teria muito propósito, devido a simplicidade da linguagem. Aqui listamos algumas características da linguagem XBA:

- Para se usar classes definidas em outros módulos, é necessário colocar uma diretiva do tipo `import`, similar a diretiva equivalente da linguagem Java.

- O escopo de um identificador associado a uma expressão por meio de um `let`, consiste no corpo dessa construção *e nas expressões associadas aos identificadores subsequentes deste `let`*, no caso do `let` possui mais do que uma associação. Por exemplo, a construção `let x = e, y = x in f` é semanticamente equivalente à expressão `let x = e in let y = x in f`.

- Mensagens unárias (sem parâmetros) têm precedência maior do que mensagens que possuem parâmetro fornecido por meio de um seletor que, por sua vez, têm precedência maior do que mensagens que utilizam operadores. Os operadores também têm uma ordem de precedência: operadores

relacionadas à multiplicação ("*", "/", "%") possuem maior precedência que operadores relacionados à adição ("+", "-") que, por sua vez, têm precedência sobre operadores de comparação (">", "<", "=", etc).

- Não é permitido o uso da variável `self` em construções do tipo `export` — não existe uma instância associada a essa construção.
- A linguagem suporta operadores *unicode* no lugar de operadores usualmente representados por dois caracteres ASCII (por exemplo, o caráter "≥" pode ser usado no lugar de ">=").

3.5 Gramática da linguagem

Segue, abaixo, a gramática (ISO-EBNF) da linguagem XBA. Observação: antes da análise sintática da linguagem existe um pré-processamento que gera caracteres do tipo *LINE_BREAK*, *BEGIN_BLOCK* e *END_BLOCK*, de acordo com a indentação.

```
(* Estrutura geral do Programa *)
program = {(import_class_statement | import_instance_statement}, class_declaration;

import_class_statement  = "import", internal_class_name,
                            [".", identifier], LINE_BREAK;
import_instance_statement  = "import", identifier, "from",
                            internal_class_name, LINE_BREAK;
export_statement  = "export", identifier, "=", expression, LINE_BREAK;
internal_class_name = identifier, {"/", identifier}

class_declaration = "class", identifier, [parameter_list],
                        BEGIN_BLOCK,
                          {field_declaration},
                          {method_declaration}
                        END_BLOCK,
                        {export_statement};

(* Seletores e lista de parâmetros *)
parameter_list  =   ({binary_selector identifier}-, [unary_selector])
                  | (unary_selector);
binary_selector = identifier, ":";
unary_selector  = identifier, "!";

(* Declaração de campos e métodos *)
field_declaration  = "field", identifier, "=", expression;
method_declaration = "method", parameter_list, "=", expression;


(* Definição de bloco e da ordem de precendência dos operadores  *)
block                = BEGIN_BLOCK,
                        expression, {LINE_BREAK, expression}, LINE_BREAK,
```

```
                       END_BLOCK;
expression           = conditional_expression;
compare_expression   = additive_expression, {op_cmp, additive_expression};
additive_expression  = product_expression, {op_add, product_expression};
product_expression   = message_expression, {op_mult, message_expression};


(* Mensagem e outras expressões *)
message_expression   = single_expression,
                        {binary_selector, single_expression} (* mensagem binária *)
                      | single_expression; (* aplicacao de funcao *)

single_expression    = ("(", expression, ")")  |  block | literal |
                           "self"  |  if_expression | let_expression;
literal              = string_literal | integer_literal | boolean_literal;


(* if e let *)
let_expression       = "let", identifier, "=", expression,
                              (",", identifier, "=", expression)*,
                              "in", expression;
if_expression        = "if", expression, "then",
                       expression, "else", expression;

(* Operadores *)
op_add = ["+" | "-"];
op_mult = ["*" | "/" | "%"];
op_cmp = ["<" | ">" | ">=" | "<=" | "=" | "!="];


(* Outros *)
alpha = ('a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z');
digit = ('0' | '1' | ... | '9');

identifier = identifier_slice, {-, identifier_slice}
identifier_slice = (alpha), {alpha | digit}

string_literal = quotes  , { all characters - quotes } , quotes ;
boolean_literal = "true" | "false";
integer_literal = ["-"]{digit}-;
```

Capítulo 4

Sistema de tipos da linguagem XBA

4.1 Descrição do sistema de tipos

O sistema de tipos da linguagem desenvolvida se baseia em extensões do sistema *Hindley-Milner* de tipos para lidar com *records*[9, 10, 11], que são estruturas de dados contendo valores que são indexados por rótulos. (*records* são similares a *struct*s da linguagem C).

Records podem ser usados para modelar objetos. Os artigos mencionados anteriormente lidam com *extensões de records*, que são usados posteriormente para modelar a herança entre objetos, o que consiste em uma complicação teórica adicional. O sistema de tipos desenvolvido para a linguagem XBA, usa como base as ideias contidas nesses artigos, tentando simplificá-las, uma vez que a linguagem não suporta herança e é puramente orientada a objetos, isto é, toda a expressão é avaliada para um objeto.

4.1.1 Objetivo

O sistema de tipos tem o objetivo de evitar que, em tempo de execução, uma mensagem (chamada de método) seja enviada a um objeto que não possui o método correspondente. Ou seja, queremos evitar que os programas compilados sejam impedidos de prosseguir sua execução porque algum objeto não possui o método requisitado por uma chamada — esse tipo de situação caracteriza os *estados indesejáveis* do sistema de tipos.

Para que o sistema de tipos possa cumprir seu objetivo, os tipos devem descrever, portanto, a "estrutura externa" de um objeto, isto é, o conjunto de métodos que o objeto possui.

4.1.2 Os tipos

Um tipo τ é definido é definido como:

- Uma *type variable*.
- Um conjunto de triplas da forma (τ_P, L, τ_R) nas quais τ_P e τ_R são tipos e L é um rótulo (*label*).

Adicionalmente, um tipo possui um valor booleano que indica se o tipo é fechado ou aberto.

Quando associamos um termo a um tipo τ, cada uma das triplas (τ_P, L, τ_R) de τ descreve características dos *métodos* aceitos pelo objeto para o qual aquele termo será avaliado:

- τ_P descreve o tipo do parâmetro do método em questão.
- L é um rótulo associado ao seletor do método em questão.
- τ_R descreve o tipo do valor devolvido pelo método em questão.

É importante re-enfatizar aqui que, em virtude do *currying*, podemos assumir que todos os métodos possuem apenas um parâmetro.

Um tipo *fechado* descreve termos cujos valores são objetos que aceitam *exatamente* o conjunto de métodos descrito por suas triplas. Por outro lado, um tipo *aberto* descreve termos cujos valores são objetos que aceitam *ao menos* o conjunto de métodos descrito por suas triplas.

Vamos representar um tipo fechado simplesmente pelo seu conjunto de triplas $\{t_1, t_2, \ldots, t_n\}$ e um tipo aberto pelo seu conjunto de triplas acrescidos de reticências: $\{t_1, t_2, \ldots t_n, \ldots\}$. Vamos representar *type variables* por letras precedidas pelo caráter '.

Tipos genéricos

Tipos são ditos *genéricos* se possuem *type variables* em sua estrutura. Tipos genéricos são comuns em funções que não fazem muitas pressuposições sobre seus argumentos, como a função *identidade* ou a função *map*. Por exemplo, o tipo associado a uma expressão que é avaliada para uma instância da função *identidade* é $\{('a, \texttt{apply}, 'a)\}$ — lembrando que, em XBA, funções nada mais são que objetos que possuem um método `apply`.

Tipos pré-definidos

A linguagem possui tipos pré-definidos, associados a cadeias de caracteres (*String*), a valores inteiros (*Int*), a valores booleanos (*Bool*), e ao tipo *unit* (*Unit*). É importante ressaltar que esses tipos não são "especiais" — eles possuem uma estrutura como qualquer outro. Por exemplo o tipo *Int* possui, dentre outras, uma tripla da forma $(Int, *, Int)$.

4.1.3 Exemplo de tipos

Aqui, apresentamos alguns trechos de programas em XBA e os tipos correspondentes. Representamos os tipos com diagramas para facilitar o entendimento.

Exemplos simples

```
1 class Printer
2     method print-diameter: circle =
3         print (2 * circle radius!)
```

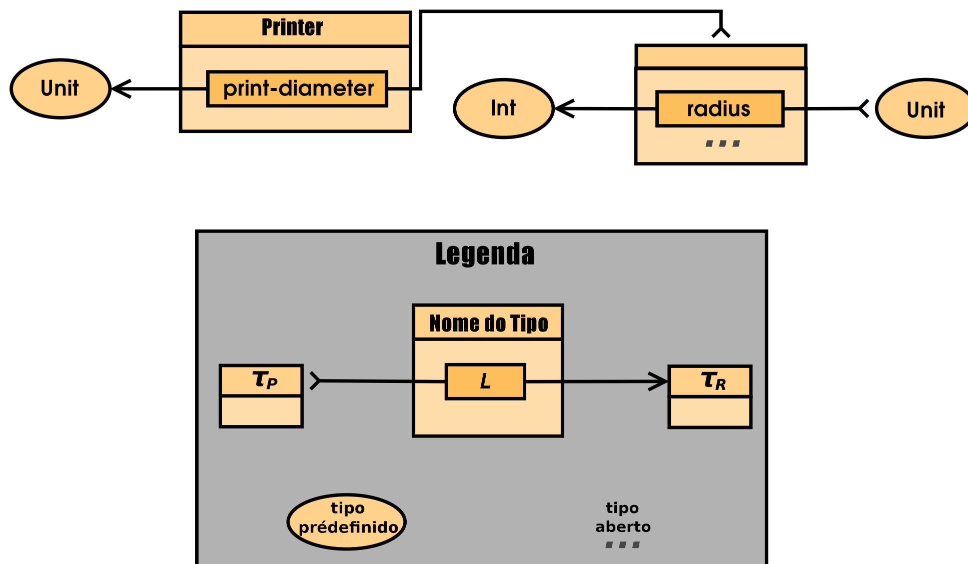

Figura 4.1: Diagrama representando o tipo de expressões que são avaliadas para instâncias de `Printer`

Note, aqui, que o tipo associado a expressão `circle radius!` deve ser, obrigatoriamente, um *Int* pois é o argumento do método ∗ sobre outro termo do tipo *Int*. Apenas por curiosidade, a função *print*, usada no exemplo, possui o tipo $\{(\{(Unit, \texttt{to-string}, String), \ldots\}, \texttt{apply}, Unit)\}$.

Exemplo simples modificado

```
1 class Printer2
2     method print-diameter: circle =
3         print (circle radius! * 2)
```

Aqui, invertemos a ordem dos operandos na multiplicação do raio do círculo por 2. Isso afeta o tipo associado às instâncias da classe `Printer2` em relação ao primeiro exemplo. Agora, o tipo da expressão `c radius!` não precisa ser necessariamente um *Int*, mas qualquer classe que possua um método com seletor ∗ que aceite um *Int* como argumento.

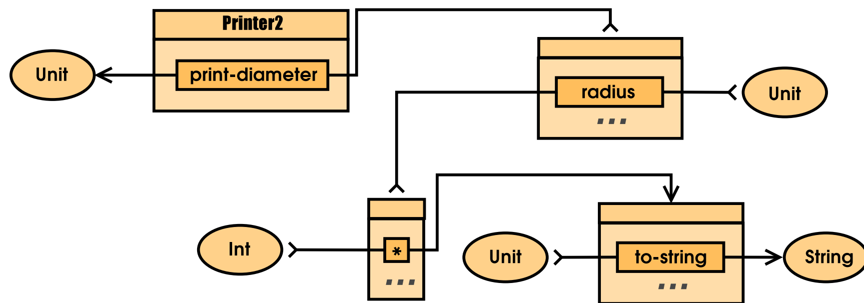

Figura 4.2: Diagrama representando o tipo de expressões que são avaliadas para instâncias da classe `Printer2`

Exemplo recursivo com *type variables*

```
1 class ExampleRec
2     method visit-self-with: visitor
3         visitor visit: self
```

Esse exemplo demonstra que tipos podem conter *type variables*, indicando que se tratam de tipos *genéricos*. Além disso, tipos podem ser recursivos, isto é, podem conter referências cíclicas (ver figura 4.3).

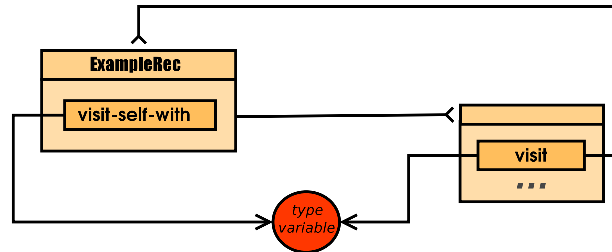

Figura 4.3: Diagrama representando o tipo de expressões que são avaliadas para instâncias da classe `ExampleRec`

4.1.4 Regras de tipos

Não vamos apresentar aqui as regras de tipos para todas as possíveis construções da linguagem: essas regras foram definidas de forma intuitiva, para se garantir que chamadas de métodos nunca falharão em tempo de execução. Como exemplo, colocamos abaixo a regra para a construção mais comum na linguagem XBA, uma chamada de método:

$$\frac{\Gamma \vdash e : \{(\tau_1, L, \tau_2), \ldots\} \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash (e \;\; L : e') : \tau_2}$$

4.1.5 Variáveis mutáveis

A razão de não termos variáveis mutáveis em XBA é que, se não forem tomados cuidados especiais, elas podem subverter o sistema de tipos (isto é, permitir programas que atingem estados indesejáveis). Existem artigos que estendem o sistema *Hindley-Milner* de tipos para comportar variáveis mutáveis [13], mas a solução mais simples e mais usada para esse problema é a de simplesmente limitar o *let-polimorfismo* [12]. Apesar de não termos *let-polimorfismo* na linguagem, preferimos não introduzir variáveis mutáveis na linguagem, por significar um trabalho extra (e por ainda não termos a segurança completa de que o sistema de tipos não seria subvertido).

4.2 Inferência de tipos na linguagem XBA

O algoritmo de inferência de tipos se baseia nas ideias do algoritmo visto em 2.2.3.

4.2.1 Unificação

Para aplicar as ideias do algoritmo visto em 2.2.3, precisamos definir o processo de unificação entre dois tipos da linguagem XBA.

O principal problema aqui é relacionado com os tipos abertos. Como a unificação tem o objetivo de tornar dois tipos idênticos, gostaríamos que tipos abertos pudessem ser "completados" durante a unificação. Por exemplo, gostaríamos que a unificação entre $\{(Unit, \texttt{sel1}, Unit), \dots\}$ e $\{(Unit, \texttt{sel1}, Unit), (Unit, \texttt{sel2}, Unit)\}$ fosse possível. Entretanto, o conceito de unificação identifica dois tipos por meio da substituição de variáveis por tipos.

Para deixar as coisas coerentes do ponto de vista *teórico*, os artigos [9, 10, 11] usam o conceito de *extension variables* (ou *row variables*) para os tipos que chamamos aqui de "abertos" — seria como se esses tipos contivessem todas as infinitas triplas da forma (τ_P, L, τ_R) para todos os infinitos rótulos L existentes. Do ponto de visto prático, contudo, podemos fazer algumas simplificações (principalmente por não trabalharmos com herança) e supor que a unificação pode "completar" tipos abertos, adicionando triplas da forma (τ_P, L, τ_R) a esses tipos.

Outra questão importante é que o algoritmo de unificação usado deve dar suporte a substituições circulares pois, como visto anteriormente, os tipos da linguagem XBA podem ser recursivos.

O Algoritmo

O algoritmo usado para unificação foi baseado em um algoritmo descrito em [2], que permite substituições circulares. Este algoritmo faz uso do conhecido *union-find*, que forma classes de equivalência entre elementos (que nesse caso serão tipos). Tipos que estão numa *mesma classe de equivalência* são tipos que foram unificados; assim, um tipo presente numa classe de equivalência é:

- uma *type variable*.
- um tipo τ que possui uma estrutura compatível com o tipo τ^*, representante da classe de equivalência. Ter uma estrutura compatível significa que, para cada tripla (τ_P, L, τ_R) de τ, existe uma tripla (τ_P^*, L, τ_R^*) em τ^*, tal que τ_P^* e τ_P estão na mesma classe de equivalência e que τ_R^* e τ_R também estão na mesma classe de equivalência. Além disso, se τ é fechado, a recíproca deve ser verdadeira.

O algoritmo consiste em uma rotina recursiva que tenta colocar os dois tipos de entrada numa mesma classe de equivalência, respeitando as condições acima.

Note que a rotina *union* deve ser modificada para atender algumas condições, como por exemplo a de não deixar que uma *type variable* seja a representante da classe de equivalência. Se a unificação for bem sucedida, a operação *find* sobre uma *type variable* encontra o tipo pela qual a *type variable* deve ser substituída.

4.2.2 A inferência em si

Uma vez definida a unificação de dois tipos, a inferência em si segue exatamente a mesma ideia da apresentada na seção 2.2.3: a inferência de uma expressão é um processo recursivo que infere os tipos das sub-expressões e os unifica com tipos contendo *type variables* que representam a estrutura *esperada* desses tipos, de forma que a expressão como um todo seja válida.

A figura 4.4 que ilustra como a inferência de tipos é feita para a expressão mais comum encontrada em programas em XBA, uma chamada de método.

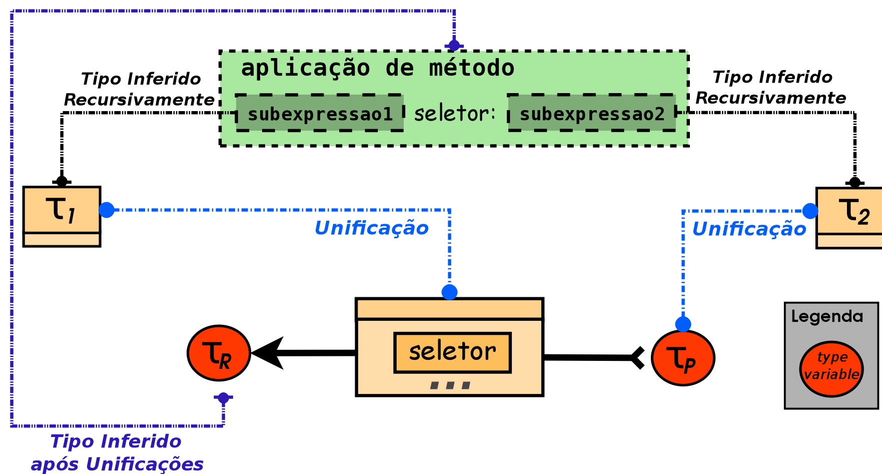

Figura 4.4: Diagrama representando inferência de tipos para uma chamada de método

Capítulo 5

Implementação

A parte prática desse trabalho consistiu na implementação de um protótipo de um compilador para a linguagem XBA. Aqui, descrevemos as principais etapas que compõe o processo de compilação de um programa XBA.

5.1 Análise léxica e sintática

O primeiro processamento que um compilador faz consiste na *análise léxica* do programa fonte. A *análise léxica* tem o objetivo de agrupar os caracteres do programa fonte nos chamados símbolos léxicos (*tokens*). Um *símbolo léxico* é uma sequência de caracteres que deve ser processado como uma única unidade. Por exemplo, na linguagem XBA, uma sequência de caracteres alfanuméricos seguidos pelo caráter ":" correspondem a um símbolo léxico denominado *seletor*. Esse símbolo léxico é tratado como uma única unidade pela fase seguinte do compilador, a *análise sintática*.

A *análise sintática* é responsável por verificar que o programa é aceito pela *gramática* da linguagem, que é descrita usando símbolos léxicos como unidade. A análise léxica geralmente gera uma representação intermediária do programa, chamada *árvore sintática*, na qual cada nó representa algum tipo de construção e os sub-nós representam as sub-construções que fazem parte daquela construção.

Por exemplo, o resultado da análise sintática sobre a expressão `3 + 5*2` (após ter sido passada pela análise léxica) é uma árvore cuja raiz é um nó — representando uma chamada de método relacionada ao operador `+` — que possui dois nós filhos: o primeiro representa o objeto no qual a chamada será realizada — o literal `3` — e o segundo representa o argumento da chamada: uma outra chamada

de método relacionada ao operador $*$ — que possui, por sua vez, sub-nós associados aos literais 5 e 2. Observe que a análise sintática captura características estruturais do programa, como a precedência das operações.

5.1.1 JavaCC

JavaCC (Java Compiler Compiler) foi a ferramenta utilizada para gerar os analisadores léxicos e sintáticos da linguagem XBA:

- Um analisador léxico é gerado a partir de uma descrição dos símbolos léxicos da linguagem; o JavaCC permite o uso de expressões regulares para a descrição dos *símbolos léxicos*.

- Um analisador sintático recursivo descendente é gerado a partir de uma descrição da gramática da linguagem. O JavaCC exige que a gramática seja da forma *LL(k)*. Isso significa, basicamente, que deve ser possível decidir qual regra de produção a gramática deve seguir a partir da leitura (da esquerda para a direita) dos k primeiros caracteres da expressão sendo analisada. Em particular, não é permitido que a gramática possua recursão a esquerda. Geralmente, uma gramática para uma linguagem de programação não está na forma *LL(k)*, mas pode ser modificada para que fique em tal forma. A referência [2] descreve algumas transformações nas regras de produção de uma gramática que podem transformá-la para a forma *LL(k)*.

Também foi usada a ferramenta *jjtree*, que é uma extensão do JavaCC que facilita a construção da árvore sintática do programa-alvo. Essa ferramenta permite que os nós sejam construídos automaticamente para certas regras de produção da gramática, facilitando muito a construção da árvore sintática — o trabalho do programador se reduz, basicamente, a inicializar os atributos dos nós construídos. Além disso, os nós gerados são objetos que implementam uma mesma interface, de tal forma que a árvore gerada segue o padrão de projeto *Composite*.

5.2 Análise semântica, inferência de tipos e geração de código

A etapa de geração do código parte da árvore sintática construída nas etapas anteriores e gera arquivos *.class*, executáveis pela JVM. Os arquivos *.class* contêm não

só os bytecodes dos métodos da classe que está sendo gerada, como também contêm alguns metadados, uma seção contendo as constantes usadas no programa, os descritores de algumas chamadas de método, etc.

A etapa de geração de código também é responsável por fazer a *análise semântica* do programa, que consiste em determinar se o programa, apesar de válido semanticamente, possui erros semânticos — que não podem ser detectados pela análise sintática.

É nessa etapa, também, que ocorre a fase de *inferência de tipos*, em que os tipos dos programas são inferidos, com base na árvore sintática, e erros de tipagem são acusados, quando apropriado. A inferência de tipos é feita em conjunto com a geração de código, pois ambas seguem a árvore sintática do programa da mesma forma, mantendo o controle sobre o escopo dos identificadores.

5.2.1 ASM

ASM é uma biblioteca usada para manipulação de bytecodes, auxiliando tanto a geração de classes completas, como também a transformação do código de classes já existentes. A biblioteca usa exaustivamente o padrão de projeto *Visitor*, que propicia uma maneira elegante de se compor classes que leem e transformam informações sobre as classes.

Com o uso dessa ferramenta, o compilador pode se ater apenas à geração do código em si, sem ter que lidar com detalhes da especificação dos arquivos *.class*. Seguem abaixo algumas facilidades providas pela ferramenta:

- Toda a estrutura do arquivo *.class* é construída - informações sobre a versão da JVM, o *pool* de constantes, os contadores de número de métodos e campos, flags de acesso, entre outros são determinados e escritos pela biblioteca, seguindo a especificação correta.

- O tamanho máximo apropriado para a pilha de cada método é automaticamente computado pela biblioteca, com base nos bytecodes do método.

- São também computados pela ferramenta os chamados *stack map frames*, que são informações adicionais que podem existir opcionalmente nos arquivos *.class* para acelerar o processo de verificação da classe.

- É possível empilhar constantes (como *strings*) diretamente; a própria biblioteca se encarrega de colocar a constante no *pool* da classe e gerar a instrução que a carrega a partir de uma referência a sua entrada nesse pool.

- É possível usar *labels* para instruções de desvio de fluxo (como *gotos*) — no nível dos bytecodes essas instruções na verdade desviam o fluxo de acordo com um *offset* correspondente ao número de instruções (positivo ou negativo) entre o local de origem e o alvo desse desvio.

5.2.2 Alguns detalhes de implementação

As informações de tipo usadas durante a compilação são usadas apenas para barrar programas que podem apresentar um comportamento ruim — o de chamar métodos em objetos que não o possuem. O modelo de execução da JVM não entende o conceito de "tipos estruturais" e, portanto, a linguagem XBA se comporta como uma linguagem dinâmica para a JVM. Isso corresponde a uma dificuldade extra, uma vez que a JVM possui tipos nominais no próprio nível dos bytecodes o que significa que é um desafio implementar linguagens dinâmicas para a plataforma.

Entretanto, na versão mais recente da plataforma, foi adicionada uma nova instrução *invokedynamic* com a finalidade de facilitar a implementação de linguagens dinâmicas para a JVM. Como cada linguagem possui um mecanismo de *dispatch* diferente e se portam de forma diferente quando um método não pode ser encontrado num objeto, essa nova instrução não "engessa" o mecanismo de busca de métodos. Ela simplesmente passa esse trabalho para uma rotina que deve ser implementada pelo *runtime* da linguagem e que deve — com o auxílio de certos mecanismos disponibilizados pela JVM — se encarregar de encontrar o método correto a ser executado e tomar as devidas providências caso ele não seja encontrado.

As classes geradas pelo compilador desenvolvido são subclasses de `XBAObject`. Essas classes possuem, para cada método do programa fonte, um campo estático do tipo `XBAFunction` — que, por sua vez, também é uma subclasse de `XBAObject`. Uma `XBAFunction` é um *wrapper* para um `MethodHandle` que é uma classe introduzida na JDK7 que funciona como uma referência "leve" (sem informações de reflexão) para um método. Uma `XBAFunction` também guarda os seletores associados ao método-alvo e é responsável pela característica de *currying* da linguagem: uma chamada a uma `XBAFunction` associada a um `MethodHandle` com mais de um parâmetro devolve uma outra `XBAFunction`, já com os seletores atualizados, que aceita os demais parâmetros. Esses campos estáticos contendo `XBAFunctions` são inicializados pelo *inicializador estático* da classe; cada uma dessas `XBAFunctions` está associada a um método que está implementado na classe.

Cada chamada de método da linguagem XBA é traduzida para uma instrução *invokedynamic* que chama *a rotina de dispatch* da linguagem, passando o objeto que recebe a chamada (receptor), o seletor. Essa rotina simplesmente:

- Chama o método `function` no receptor, obtendo a `XBAFunction` associada ao seletor. Para `XBAObjects` usuais, o método `function` busca a `XBAFunction` apropriada dentre aquelas guardadas nos campos estáticos. Para `XBAFunctions` esse método simplesmente verifica que o seletor usado é o esperado e devolve uma referência para si mesmo.

- Faz o *caching* dessa busca para não ter que repeti-la no caso dessa mesma instrução ser executada novamente.

A figura 5.1 mostra um diagrama com a hierarquia de classes descrita acima:

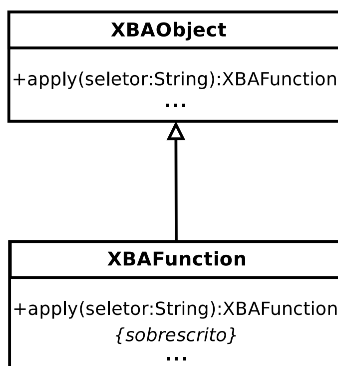

Figura 5.1: Diagrama de classes que fazem parte do runtime da linguagem

Por fim, a rotina de *dispatch* da linguagem não precisa se preocupar em não encontrar o método apropriado de um objeto — o sistema de tipos garante que os programas compilados nunca farão chamadas a métodos inexistentes no objeto receptor.

5.3 Deficiências do protótipo desenvolvido

O protótipo desenvolvido tinha como objetivo o estudo da inferência em sistemas estruturais de tipos. Embora esse objetivo principal tenha sido alcançado, o protótipo possui algumas deficiências que o distingue de um produto completo. Dentre tais deficiências, podemos ressaltar:

- Não se tomou um cuidado excessivo em mostrar mensagens de erro precisas em programas incorretos. O protótipo não aponta exatamente o local de possíveis erros semânticos ou de tipo. Esse tipo de comportamento é, evidentemente, inaceitável em um produto finalizado.

- Não é possível compilar mais de um módulo de uma única vez. Desta forma, não é possível compilar módulos que possuam dependências circulares. (Na verdade, o programador pode criar classes *dummy* para contornar esse problema, mas essa não é a solução ideal.)

- Não se tomou cuidado com a produção de um código eficiente. Apesar da JVM fazer otimizações agressivas em tempo de execução, a qualidade do código gerado não é compatível com a de um produto finalizado.

- A inferência de tipos ainda não foi integrada totalmente a geração de código. Por enquanto ainda se trata de um processo separado e que não é capaz de carregar módulos externos.

- Existem alguumas deficiências na arquitetura do código, que estão devidamente documentadas.

- Não são colocadas informações para depuração no código gerado.

Capítulo 6

Considerações finais

Este trabalho permitiu o estudo de alguns aspectos teóricos sobre sistemas de tipos, assim como um maior entendimento das dificuldades por trás do problema da *inferência de tipos*, que é uma característica cada vez mais comum nas linguagens atuais.

A complexidade do problema de se inferir tipos estruturais em uma linguagem orientada a objetos se mostrou muito maior do que a imaginada inicialmente. Isso fez com que fosse necessária a leitura de vários textos teóricos sobre assuntos relacionado. Foi também necessário simplificar boa parte das ideias vistas para que fosse possível chegar a resultados concretos no final do trabalho. Em particular, foi preciso abrir mão de diversas características idealizadas originalmente (como herança e variáveis), para que o sistema de tipos não se tornasse demasiadamente complexo.

Se por um ponto de vista essas simplificações possuem um lado negativo evidente, por outro, o desenvolvimento de uma linguagem simples com as características propostas também possui seu lado positivo, principalmente em termos *didáticos*. Os artigos que propunham sistemas estruturais geralmente não se baseavam em linguagens de verdade, apenas em modelos teóricos. Em contrapartida, linguagens reais (e que possuem tipagem estrutural e suporte à orientação a objetos — como OCaml) possuem um sistema de tipos *extremamente complexo*, o que dificulta seu estudo.

Ademais, o desenvolvimento de um protótipo de um compilador consistiu em uma oportunidade de colocar em prática outros tópicos teóricos de computação, como linguagens regulares, gramáticas livre de contexto, etc.

Finalmente, vale ressaltar que o sistema de tipos desenvolvido nesse trabalho também possui, apesar das vantagens citadas anteriormente, suas desvantagens.

A principal delas é, justamente, que apenas o aspecto *estrutural* é levado em conta na checagem de tipos. Entretanto, existem muitas situações onde lidamos com dados que são compatíveis estruturalmente, mas não o são *semanticamente*. Por exemplo, um *vetor* e um *ponto* no plano, representam dados que são estruturalmente muito similares, mas que possuem uma diferença semântica importante. Gostaríamos que o sistema de tipos fosse capaz de captar tal diferença. Outra desvantagem é que a ausência completa de *anotações de tipos* pode dificultar o entendimento do código, uma vez que essas anotações servem como uma espécie de documentação de um programa.

Linguagens modernas, como *Scala*, possuem sistemas de tipos complexos; neles, é possível misturar tipos nominais com estruturais e inserir algumas anotações de tipos no programa, quando conveniente. Com isso, obtém-se as as principais vantagens de sistemas de tipos nominais e estruturais.

Referências Bibliográficas

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

[2] Aho, A. V., Sethi, R., Ullman, J. D. *Compilers: Principles, Techniques and Tools*. Addision-Wesley, Reading, 1998.

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

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

[5] 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

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

[7] J. Robinson. *A machine oriented logic on the resolution principle*, J. ACM 12, pp.25-41, 1965

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

[9] Didier Rémy. *Type inference for records in a natural extension of ML*, Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press, 1993

[10] Mitchell Wand. *Complete type inference for simple objects*, Proceedings of the IEEE Symposium on Logic in Computer Science. Ithaca, NY, 1987

[11] Mitchell Wand. *Type inference for objects with instance variables and inheritance*, Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design, C. A. Gunter and J. C. Mitchell, Eds. The MIT Press, 97–120, 1994

[12] Andrew K Wright. *Typing References by Assignment*, Euopean Symposium on Programming, February LNCS 582, pp 473-491, 1992

[13] Kung Chen, Martin Odersky. *A type system for a Lambda Calculus with Assignments*, 1993

[14] 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.

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

[16] Eric Bruneton. *ASM 4.0: A Java bytecode engineering library.* `http://download.forge.objectweb.org/asm/asm4-guide.pdf`, February 2007.

Apêndice A

Parte Subjetiva

A.1 Dificuldades e desafios encontrados

De um ponto de vista mais **objetivo** as duas principais dificuldades encontradas foram:

- Inicialmente, subestimei a dificuldade do problema de inferir os tipos estruturais do programa. A ideia original seria fazer um compilador para uma linguagem "usável" para a JVM (o que seria a parte difícil) e considerei que a parte relacionada ao sistema de tipos seria mais fácil. Entretanto, o *problema se mostrou muito difícil* e os artigos que tratavam sobre inferência de tipos estruturais eram geralmente *mais densos e exigiam um conhecimento prévio* sobre o assunto que eu não possuía — e ainda não possui em grande parte.

- Além disso, a parte relacionada a implementação foi prejudicada, em parte, por usar a nova instrução *invokedynamic* da JDK7 (e os mecanismos correspondentes). O problema é que essa versão da JDK só foi lançada oficialmente no meio do ano; nesse meio tempo tive que trabalhar com versões (beta, RC) "instáveis", cuja especificação mudava com certa frequência.

Do ponto de vista mais **subjetivo**:

- Definir o escopo de um trabalho foi algo que raramente tivemos que fazer durante a graduação. Apenas as disciplinas de laboratório exigiam algo do tipo e, ainda assim, os trabalhos que tínhamos que propor já tinham uma direção a ser seguida. Nesse sentido, senti uma dificuldade em ter que

definir o trabalho a ser feito. Se por um lado existe o desejo de se fazer algo complexo — que justifique o trabalho realizado durante dois semestres — por outro, existe o receio de não ser viável cumprir com o prometido.

- A **principal dificuldade**, de longe, encontrada nesse trabalho foi ter que lidar com *prazos tão grandes*. Com toda certeza, faltou para mim a *disciplina* para trabalhar no projeto durante todo o ano de maneira uniforme; acabei me dedicando ao trabalho de forma mais significativa apenas nos períodos que antecederam as datas limites da disciplina.

A.2 Disciplinas relevantes para este trabalho

Segue, abaixo, as disciplinas mais relevantes, em ordem, para a realização desse trabalho:

- **MAC0316 - Conceitos Fundamentais de Linguagens de Programação**. Certamente, a disciplina mais próxima desse trabalho por tratar sobre conceitos importantes de linguagens de programação, sobre o paradigma de programação funcional e por dar uma introdução a sistemas de tipos em linguagens de programação. De maneira complementar, a disciplina **Programação Funcional Contemporânea (oferecida na sigla MAC0434)** também contribuiu por se aprofundar no estudo do paradigma de programação funcional — que teve influência na escolha de algumas características da linguagem desenvolvida.

- **MAC0414 - Linguagens Formais e Autômatos**. A disciplina faz uma introdução a alguns conceitos teóricos tais como linguagens regulares, gramáticas livres de contexto, gramáticas LL(k), etc; o entendimento desses conceitos foi essencial para fazer a parte prática desse trabalho. Ferramentas como *JavaCC* pressupõem um conhecimento prévio nesse assunto.

- **MAC0239 - Métodos Formais em Programação**. Essa disciplina fornece uma introdução a conceitos de lógica para alunos do BCC. Sistemas de tipos consistem métodos formais capazes de provar a existência/inexistência de certos comportamentos em um programa. Dessa forma, muitos dos artigos lidos sobre inferência de tipos continham conceitos (e usavam notações) que foram estudados nesta disciplina.

- **MAC0242 - Laboratório de Programação II.** Na época em que cursei, a disciplina tinha o objetivo (dentre outros) de introduzir conceitos sobre linguagens de montagem, analisadores léxicos e sintáticos, (esses assuntos fazem parte , atualmente, do programa de MAC0211). Nesse curso, tivemos que implementar uma máquina virtual e um compilador para ela, algo que está intrinsecamente relacionado com as atividades desenvolvidas na parte prática desse trabalho. Foi nessa disciplina, também, que tive a oportunidade de trabalhar com o *JavaCC*, que facilitou, em muito, a produção do analisador sintático da linguagem.

- **MAC0441 - Programação Orientada a Objetos.** Por introduzir conceitos de orientação a objetos tais como classes, métodos, herança, etc que foram usados nesse trabalho. Além disso a própria implementação de um compilador em uma linguagem orientada a objetos (*Java*) geralmente faz uso de diversos padrões de projeto ensinados na disciplina , tais como *Visitor*, *Composite*, etc.

- **MAC0122/MAC0323** Estas disciplinas introdutórias do curso também tiveram grande importância para a realização desse trabalho. Nesse trabalho, foram usados algoritmos de caráter recursivo, algoritmos que lidam com árvores e uma variação do algoritmo **union-find**, etc. O estudo de todos esses temas está incluso no programa destas disciplinas.

A.3 Continuação na área

O estudo de linguagens de programação é, certamente, um assunto muito interessante. Acredito que esse assunto deva despertar o interesse de qualquer pessoa que programe com frequência. Ao programar, é comum encontrar uma situação na qual uma determinada linguagem parece não ter a expressividade necessária para lidar com um problema específico, de forma elegante. Ou, ainda, situações em que algum tipo de verificação estática poderia ser muito útil para evitar certos erros frequentes em um problema específico.

Se eu fosse continuar atuando na área que realizei este trabalho, eu provavelmente estudaria sistemas de tipos mais complexos, capazes de evitar um conjunto maior de estados indesejáveis, dando, assim, uma maior “segurança” em tempo de compilação. Também seria interessante estudar outros métodos formais capazes de garantir a ausência de comportamentos indesejados nos programas. Por exemplo, em algumas linguagens, como *SPARK*, é possível garantir que o valor de

uma certa variável é alterado usando apenas um dado conjunto de outras variáveis do programa. Garantias como esta podem ser muito úteis em determinadas situações, mas a verificação dessas condições é uma tarefa complexa.

O estudo de todos esses métodos de verificação estática de programas obviamente requer um conhecimento prévio em *lógica*. Textos sobre tais assuntos geralmente fazem referência a diversos conceitos e notações de lógica. Assim, para continuar um estudo nessa direção, também seria necessário adquirir um embasamento maior nessa área.