UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THALES DE FREITAS MAGALHÃES

UMA INTRODUÇÃO À INTERPRETAÇÃO ABSTRATA DE LINGUAGENS FUNCIONAIS

RIO DE JANEIRO
2024

THALES DE FREITAS MAGALHÃES

UMA INTRODUÇÃO À INTERPRETAÇÃO ABSTRATA DE LINGUAGENS FUNCIONAIS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Hugo de Holanda Cunha Nobrega

RIO DE JANEIRO
2024

CIP - Catalogação na Publicação

Magalhães, Thales de Freitas
M188i Uma introdução à interpretação abstrata de linguagens funcionais / Thales de Freitas Magalhães. -- Rio de Janeiro, 2024.
43 f.

Orientador: Hugo de Holanda Cunha Nobrega.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. programação funcional. 2. análise estática de programas. 3. interpretação abstrata. 4. avaliação parcial. I. Nobrega, Hugo de Holanda Cunha, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

THALES DE FREITAS MAGALHÃES

UMA INTRODUÇÃO À INTERPRETAÇÃO ABSTRATA DE LINGUAGENS FUNCIONAIS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 02 de abril de 2024

BANCA EXAMINADORA:

Documento assinado digitalmente
gov.br
HUGO DE HOLANDA CUNHA NOBREGA
Data: 08/04/2024 22:55:49-0300
Verifique em https://validar.iti.gov.br

Hugo de Holanda Cunha Nobrega
Ph.D. (UFRJ)

Documento assinado digitalmente
gov.br
JOAO ANTONIO RECIO DA PAIXAO
Data: 09/04/2024 06:52:38-0300
Verifique em https://validar.iti.gov.br

João Antonio Recio da Paixão
D.Sc. (UFRJ)

Documento assinado digitalmente
gov.br
HUGO MUSSO GUALANDI
Data: 08/04/2024 23:42:16-0300
Verifique em https://validar.iti.gov.br

Hugo Musso Gualandi
D.Sc. (UFRJ)

AGRADECIMENTOS

Agradeço ao professor Hugo Nobrega, por aceitar ser meu orientador e me acompanhar ao longo desta aventura, além do apoio extraordinário oferecido para levar esse projeto até a linha de chegada; à minha família, pelo suporte fornecido durante todo esse processo; aos professores João Paixão e Hugo Gualandi, por aceitarem fazer parte da banca, pelos comentários oferecidos, e pela paciência; e aos cientistas e matemáticos responsáveis pelo desenvolvimento dos métodos que serviram de inspiração para esse trabalho.

“We do these things not because they are easy,
but because we thought they were going to be easy."

RESUMO

A execução de código em tempo de compilação é uma técnica importante no desenvolvimento de software com aplicações diversas, incluindo a otimização de programas, meta-programação, e a validação de expressões em tempo de compilação. Embora a execução de expressões em tempo de compilação seja possível em linguagens como C++ ou Scheme, essa tarefa costuma complexa e, em alguns casos, imprática. Além disso esse processo exige, no mínimo, a anotação manual das expressões do programa informando ao compilador (ou interpretador) quais partes do programa podem (ou devem) ser executadas estaticamente. Neste trabalho, exploramos estratégias de análise estática com o objetivo de pré-computar, de forma automática, parte das expressões de um programa escrito em uma linguagem de programação funcional baseada no cálculo lambda não tipado.

Palavras-chave: programação funcional; análise estática de programas; interpretação abstrata; avaliação parcial.

ABSTRACT

Compile-time code execution is an important technique in software development which has wide-ranging applications, including but not limited to: program optimization, validation, and meta-programming. Although executing code at compile-time is possible to some extent in languages like C++ or Scheme, doing so is often challenging and, in some cases, rather impractical. Moreover, this process requires that expressions be hand-annotated by the programmer, informing the compiler (or interpreter) whether or not an expression can (or should) be statically executed. In this text, we seek to explore methods of performing this partial evaluation of a program in an automated manner, through the static analysis of a higher-order, functional programming language based on the untyped lambda calculus.

Keywords: functional programming; static program analysis; abstract interpretation; partial evaluation.

SUMÁRIO

1 INTRODUÇÃO

A execução de código em tempo de compilação se refere ao processo através do qual parte das expressões de um programa podem ser avaliadas *estaticamente*, isto é, antes do tempo de execução (*runtime*) do programa. A aplicação mais direta da execução de código em tempo de compilação é na otimização de programas através do pré-cálculo de expressões computacionalmente caras ou, quando realizada de forma automática, a eliminação de expressões constantes durante o processo de compilação de um programa. Além disso a técnica também tem inúmeras outras aplicações, desde a geração automática de código (meta-programação), a validação de expressões em tempo de compilação, e a implementação de linguagens de domínio específico.

Em linguagens estaticamente tipadas, por exemplo, não é raro que a execução de código em tempo de compilação seja permitida de maneira restrita. Quando a linguagem adota mecanismos de programação genérica, essa funcionalidade é importante para a validação do código gerado. O sistema de *templates* da linguagem C++ é um exemplo de implementação desse tipo de mecanismo: apesar de inicialmente não ter sido concebido com esse intuito, esse sistema pode ser usado para a implementação de algoritmos arbitrariamente complexos através do uso de técnicas de *template metaprogramming*. Porém essa é uma tarefa complexa, pouco intuitiva, e propensa a erros.

Sistemas de macro sintáticos, popularizados por linguagens da família Lisp (como Scheme, Racket, Common Lisp, etc.), são outro exemplo de aplicação da execução de código em tempo de compilação. Esses sistemas permitem a definição de funções (chamadas *macro transfomers*) capazes de modificar algoritmicamente a árvore sintática de um programa. Macros sintáticos podem ser usados para aumentar a expressividade de uma linguagem ao permitir sua extensão através da definição de novas construções sintáticas. Embora esses sistemas também possam ser usados de forma mais abrangente (CHANG; KNAUTH; GREENMAN, 2017), a adoção de uma estratégia de execução em fases limita sua praticidade como ferramenta de execução de código em tempo de compilação devido às restrições impostas sobre o compartilhamento de definições entre as fases de expansão de macros e de *runtime* do programa (KOHLBECKER et al., 1986; FLATT; PLT, 2024).

Apesar de serem ferramentas poderosas, esses sistemas compartilham de uma mesma limitação: embora a execução de código em tempo de compilação seja possível, essa tarefa requer um processo manual onde, no mínimo, o programador é obrigado a comunicar ao compilador quais partes do programa devem ser executadas estaticamente e quais não.

Nesse trabalho estamos interessados em estudar formas de se realizar essa distinção de forma automática; em outras palavras, queremos explorar métodos através dos quais podemos determinar, sem a necessidade de interação do usuário, quais partes do programa podem ser pré-computadas (i.e. são estáticas) e quais não podem (i.e. são dinâmicas).

A partir dessa classificação, se torna possível avaliar (parcialmente) as expressões de um programa em tempo de compilação. Entretanto, como veremos, para que esse processo de avaliação parcial seja computável ele deve ser realizado de forma aproximada. Isso significa que nem sempre é possível detectar e pré-computar todas as expressões estáticas de um programa, mesmo quando na prática (isto é, em *runtime*) o seu resultado seja sempre o mesmo.

No contexto da otimização automática de programas, em particular quando falamos de compiladores, uma técnica comumente usada para essa tarefa consiste em realizar uma análise de *data-flow* do código do programa. Tradicionalmente, essa estratégia consiste em construir um grafo representando o fluxo de controle do programa analisado, a partir do qual se torna possível determinar como valores (ou, em geral, o estado de execução do programa como um todo) é modificado e se propaga ao longo da sua execução. Porém, a construção desse grafo requer conhecimento sobre o fluxo de controle do programa, o que pode ser difícil de se obter. Em particular, essa análise de fluxo de controle é uma tarefa bastante complexa para programas escritos no paradigma de programação funcional, principalmente quando comparado à mesma análise para linguagens imperativas (SHIVERS, 1991).

Neste trabalho, escolhemos explorar estratégias de avaliação parcial para uma linguagem funcional simples baseada no cálculo lambda. Desta forma, optamos adotar técnicas mais favoráveis à formalização da propagação de constantes nesse contexto; em particular, enquadramos o método utilizado como uma análise estática e rascunhamos uma prova da sua corretude como uma *interpretação abstrata* dessa linguagem. Conforme exploramos a construção desta análise, ao longo do trabalho também introduzimos o embasamento teórico necessário para a formalização das técnicas utilizadas, desde princípios básicos como os fundamentos da programação funcional e da teoria da computação, a formalização da semântica de programas, e por fim a análise estática através da interpretação abstrata.

No Capítulo 2 exploramos a relação entre a programação funcional e o cálculo lambda e, ao fim deste capítulo, apresentamos uma linguagem funcional simples que será usada de exemplo ao longo do trabalho (Seção 2.4). Depois, no Capítulo 3, introduzimos as noções básicas da teoria de domínios e semântica denotacional que serão necessárias para a formalização da análise de propagação de constantes. Nesse mesmo capítulo também descrevemos a semântica da linguagem apresentada anteriormente (Seção 3.4). O Capítulo 4 introduz os conceitos básicos da interpretação abstrata (do inglês *abstract interpretation*) que usaremos de base para a construção desta análise. No Capítulo 5, apresentamos estratégias de aproximação que devem ser adotadas para tornar essa análise computável.

2 PROGRAMAÇÃO FUNCIONAL

O paradigma de programação funcional é caracterizado pela construção de programas através da definição, composição, e aplicação de funções. Em contraste ao paradigma imperativo onde o comportamento de um programa é descrito através de sequências de comandos que, quando executados, podem alterar o estado do programa, no paradigma funcional programas são descritos de forma declarativa através de equações matemáticas.

Embora o emprego de uma mistura desses paradigmas seja possível através de uso de linguagens multi-paradigma, quando adotamos uma abordagem exclusivamente funcional buscamos construir programas apenas através do uso de funções *puras*. O adjetivo "pura", nesse caso, se refere à ausência de efeitos colaterais como a mutação de variáveis, a execução de operações de I/O, etc. Nesse regime, o resultado de chamadas de função $f(x)$ depende única e exclusivamente do valor do argumento x, ao contrário do que acontece no paradigma imperativo onde o valor de retorno de uma rotina pode interagir com e/ou depender do estado global do programa de formas inesperadas. Chamamos essa propriedade das funções no paradigma funcional puro de *transparência referencial*.

A ausência de efeitos colaterais e a transparência referencial das funções puras tem consequências fundamentais para a análise estática de programas escritos nesse paradigma. Em particular, a presença dessas características permite a aplicação de otimizações de código mais agressivas que seriam impossíveis na sua ausência, como a eliminação de subexpressão comum e a memoização (*caching*) de chamadas de função (APPEL, 1997).

Além do uso de funções puras, uma das características mais destacantes de programas escritos no paradigma de programação funcional é a aplicação extensiva de funções recursivas, assim como o uso de funções de alta ordem (isto é, funções que recebem ou retornam outras funções). A definição deste tipo de função se torna possível devido ao fato que, nessas linguagens, funções são tratadas como "cidadão de primeira classe", o que significa que podem ser manipuladas, armazenadas, etc. assim como qualquer outro valor.

Nesse capítulo vamos explorar os fundamentos teóricos da programação funcional através do cálculo lambda (Seções 2.1 e 2.2), em particular veremos como a recursão é modelada por meio desse sistema (Seção 2.3). Ao final do capítulo, relacionamos essa teoria à linguagens de programação de maneira mais concreta através da descrição de uma linguagem funcional simples baseada no cálculo lambda, que será usada como exemplo prático ao longo desse trabalho (Seção 2.4).

2.1 O CÁLCULO LAMBDA

O cálculo lambda é um sistema formal inicialmente concebido por Alonzo Church na década de 1930 (CARDONE; HINDLEY, 2006 apud CHURCH, 1932) como parte de um

esforço em desenvolver uma fundação alternativa para a lógica matemática. A descoberta de inconsistências no sistema original levaram Church a trabalhar em uma revisão desse sistema (CARDONE; HINDLEY, 2006 apud CHURCH, 1933), dando origem ao que chamamos hoje do *cálculo lambda não tipado* (ao qual nos referimos simplesmente como "o cálculo lambda" nesse trabalho).

Apesar de inicialmente não ter encontrado muitas aplicações fora do ramo da lógica matemática, esse sistema acabou obtendo grande sucesso como um modelo computacional, servindo como base teórica para o desenvolvimento e estudo das linguagens de programação funcional. O cálculo lambda formaliza a noção de computabilidade, sendo equivalente nesse quesito ao modelo das máquinas de Turing (TURING, 1936; TURING, 1937).

Nota. Na busca de um sistema consistente, Church também desenvolveu uma versão do sistema com tipos, hoje conhecido como o *cálculo lambda (simplesmente) tipado*. A introdução de tipos ao cálculo lambda garantiu sua consistência como um sistema lógico, porém o tornou mais fraco em relação à versão não tipada do sistema. Essa versão "simples" do cálculo tipado foi desenvolvida por outros autores ao longo dos anos, dando origem a uma família diversa de cálculos. O(s) cálculo(s) lambda tipado(s) servem de base teórica para as linguagens de programação funcional estaticamente tipadas, sendo também relacionados aos sistemas lógicos construtivistas através do isomorfismo de Curry-Howard (HOWARD, 1980).

2.1.1 Gramática

Formalmente, o cálculo lambda é caracterizado através de uma gramática de *termos* e um conjunto de transformações sintáticas entre esses termos. Nesse trabalho adotamos a notação convencional de usar letras maiúsculas (e.g. M, N, etc.) para representar termos arbitrários dessa linguagem. Assim, exceto quando indicado ao contrário, assume-se que M, N, $\ldots \in \mathit{Term}$.

A gramática do cálculo lambda é composta por *variáveis*: e.g., x, y, z, $\ldots \in \mathit{Var}$; *abstrações lambda*: expressões da forma $(\lambda x.\, M)$ representando funções anônimas de um argumento; e *aplicações*: expressões da forma $(M\ N)$ representando a aplicação de uma função M em um argumento N:

$$\mathit{Term} ::= x \mid (\lambda x.\, M) \mid (M\ N)$$

Nota. Por questão de legibilidade, adotamos convenções sintáticas bem estabelecidas para o cálculo lambda na literatura, i.e., a omissão de parênteses externos: $M\ N \equiv (M\ N)$; aplicações associam à esquerda: $M\ N\ P \equiv ((M\ N)\ P)$; o corpo de abstrações lambda estende o máximo possível à direita: $\lambda x.\, M\ N \equiv \lambda x.\, (M\ N)$, etc.

É importante notar que, ao contrário do que acontece com a maioria das linguagens de programação[1], a gramática do cálculo lambda admite expressões contendo variáveis livres: dizemos que uma variável x está *ligada* à uma abstração $(\lambda x.\, M)$ quando pertence ao seu *escopo* M; caso contrário, dizemos que x está *livre*. Escrevemos $\mathrm{FV}(M)$ para denotar o conjunto das variáveis livres de um termo M e dizemos que M é um termo *fechado* quando não possui variáveis livres, isto é, quando $\mathrm{FV}(M) = \emptyset$. Nesse caso, também é comum se referir a esse termo como um *combinador*.

2.1.2 Reduções e equivalência

No cálculo lambda, termos são relacionados entre si através de *reduções*: transformações sintáticas que preservam alguma noção de equivalência entre termos. A conversão-α, por exemplo, formaliza a noção de *renomeação de variáveis*: informalmente, essa regra diz que dois termos M e N são *α-equivalentes*, escrito $M =_\alpha N$, quando um pode ser obtido a partir do outro com uma (ou mais) troca de variáveis ligadas.

A noção de "execução" no cálculo lambda é capturada de forma similar: a redução-β descreve como um termo da forma $((\lambda x.\, M)\ N)$ pode ser *reduzido* através da substituição da variável x por N no corpo da função, i.e., o termo $((\lambda x.\, M)\ N)$ *reduz* para $M[N/x]$, escrito

$$((\lambda x.\, M)\ N) \to_\beta M[N/x]$$

Aqui a notação $M[N/x]$ é usada para representar o termo resultante da substituição da variável x por N no termo M, incluindo a renomeação de variáveis conforme necessário para evitar a captura acidental de variáveis livres.

Chamamos um termo dessa forma de *expressão redutível*, abreviado como *redex* (do inglês *reducible expression*). A redução-β de um termo, na interpretação computacional do cálculo lambda, pode ser vista como a execução de uma operação elementar (como, por exemplo, a execução de uma instrução de máquina).

A redução-β também induz uma noção de equivalência entre termos, chamada de β-equivalência: intuitivamente, dizemos que dois termos M e N são β-equivalentes, escrito $M =_\beta N$, quando reduzem para um mesmo termo (potencialmente após múltiplos passos). Escrevemos $M = N$ quando M e N são α-, β-, ou uma combinação das duas equivalências. Por pequeno abuso de notação, também escrevemos $M = N$ quando M e N são *extensionalmente equivalentes*, isto é, quando $(M\,P) = (N\,P)$ para todo $P \in \mathit{Term}$.

2.1.3 Forma normal

A noção de "valor" no cálculo lambda também segue da redução-β: se enxergamos uma expressão redutível como uma computação pendente, então um termo que não possui

[1] Dizemos que uma linguagem com essa característica adota o escopo *estático* (ou *léxico*) para variáveis; caso contrário dizemos que a linguagem comporta variáveis com escopo *dinâmico*.

expressões redutíveis pode ser visto como o valor final ou resultado de alguma computação. Formalmente, quando um termo não possui expressões redutíveis dizemos que ele está em *forma normal*.

É possível, porém, que um termo possua mais de uma expressão redutível: nesse caso, não há uma ordem determinística para sua redução. Entretanto, o teorema de Church–Rosser (CHURCH; ROSSER, 1936) garante que a β-redução é *confluente*, isto é, que independente da sequência de reduções adotada para reduzir um termo à sua forma normal, ela é única. Note que isso não significa que todo termo necessariamente tem uma forma normal: o termo Ω definido abaixo, por exemplo, nunca reduz a uma forma normal:

$$\Omega \equiv (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \tag{2.1}$$

2.1.4 Estratégias de redução

Para que o processo de β-redução se dê de maneira algorítmica, é necessário adotar alguma *estratégia de redução*. Estratégias comuns são a *call-by-value*, onde o argumento N de uma aplicação $(M\ N)$ é sempre reduzido antes da sua substituição em M; e *call-by-name*, onde a redução do argumento N só ocorre (caso necesário) após essa substituição. A estratégia *call-by-value* corresponde à semântica de avaliação *estrita* de linguagens como Scheme, ML, etc., enquanto que a estratégia *call-by-name* é relacionada à semântica de avaliação "*lazy*" de linguagens como Haskell.

Nem toda estratégia de redução é garantida de reduzir um termo à sua forma normal, mesmo que ela exista (SESTOFT, 2002). Dessa forma, a escolha de estratégia de redução de uma linguagem tem consequências importantes para sua semântica.

2.2 CODIFICAÇÃO DE DADOS

Na seção anterior descrevemos uma versão minimalista do cálculo lambda, conhecida como o cálculo lambda *puro*, pois trata única e exclusivamente de variáveis, funções, e da aplicação de funções. Esse sistema não inclui tipos de dados básicos que se espera de uma linguagem de programação real como números, booleanos, etc., ou mesmo estruturas de dados simples como tuplas, listas, etc..

Quando o cálculo lambda é aplicado à modelagem de linguagens de programação de forma mais prática, é comum estender essa versão pura do sistema com a introdução de constantes representando esses tipos de dados, assim como a inclusão de operações primitivas como a soma, subtração, multiplicação, etc., tornando o tratamento de expressões envolvendo essas operações mais natural. Embora essa versão "aplicada" do cálculo seja mais conveniente para para a modelagem de linguagens de programação reais, os dois sistemas são na verdade computacionalmente equivalentes pois a representação desses objetos também é possível no cálculo lambda puro através de *codificações*.

2.2.1 Numerais de Church

Vejamos, por exemplo, o caso dos números naturais: desejamos encontrar uma sequência de termos $\underline{0}$, $\underline{1}$, ..., $\underline{n}$, ... para representar o conjunto dos números naturais de forma que seja possível definir as operações de soma, subtração, multiplicação, etc. através de combinadores, e.g., (add $\underline{m}$ $\underline{n}$) $= \underline{m+n}$, (mul $\underline{m}$ $\underline{n}$) $= \underline{m \times n}$, etc.

Uma solução é dada por Church através dos epônimos *numerais de Church* (**??**): nessa codificação, os termos $\{\mathrm{c_n}\}$ são definidos indutivamente através de um combinador succ que produz o *sucessor* de um numeral $\mathrm{c_n}$ qualquer, isto é, $\mathrm{c_{n+1}} \equiv (\mathrm{succ\ c_n})$. Desta forma se torna necessário definir apenas os termos $\mathrm{c_0}$ e succ para obter essa sequência completa:

$$\mathrm{c_0} \equiv \lambda f.\ \lambda x.\ x$$
$$\mathrm{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$$

A partir dessa codificação podemos definir as operações aritméticas entre os números naturais dentro do cálculo lambda. Alguns exemplos relevantes são:

$$\mathrm{add} \equiv \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$
$$\mathrm{mul} \equiv \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x$$

É importante notar que uma escolha de codificação é, de certa forma, apenas uma questão de convenção. No caso dos números naturais, por exemplo, existem outras codificações possíveis como a codificação de Scott (SCOTT, 1963). Na verdade, qualquer escolha de termos $\{\,\underline{n}\,\}$ seria apropriada desde que fosse possível definir as operações de soma, subtração, multiplicação, etc. de maneira que as propriedades de interesse dessas operações (e.g., associatividade, comutatividade, distributividade, etc.) fossem mantidas.

2.2.2 Álgebra booleana

Outro exemplo notável é a codificação dos valores booleanos $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$. Tradicionalmente, esses valores são codificados no cálculo lambda através dos termos:

$$\mathrm{T} \equiv \lambda x.\ \lambda y.\ x$$
$$\mathrm{F} \equiv \lambda x.\ \lambda y.\ y$$

Essa codificação é conveniente pois torna a implementação da álgebra booleana como um todo bastante simples. Em particular, a implementação de expressões condicionais da forma (if $B\ M\ N$) pode ser trivialmente definida através do combinador:

$$\mathrm{if} \equiv \lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z = \lambda x.\ x$$

Outros exemplos de combinadores usando valores booleanos incluem:

$$\text{not} \equiv \lambda x.\ \lambda y.\ \lambda z.\ x\ z\ y \tag{2.2}$$

$$\text{and} \equiv \lambda x.\ \lambda y.\ \text{if}\ x\ y\ \text{F} \tag{2.3}$$

$$\text{or} \equiv \lambda x.\ \lambda y.\ \text{if}\ x\ \text{T}\ y \tag{2.4}$$

$$\text{IsZero} \equiv \lambda n.\ n\ (\lambda x.\ \text{T})\ \text{F}$$

2.2.3 Outros dados

Estratégias similares podem ser usadas para codificar outros tipos de dados como números inteiros, racionais, tuplas, listas, etc.; porém estão fora do escopo deste trabalho.

2.3 RECURSÃO NO CÁLCULO LAMBDA

A recursão é um mecanismo essencial no paradigma de programação funcional pois, na ausência de funções recursivas, uma linguagem puramente funcional não seria sequer Turing-completa. Em teoria o cálculo lambda serve de modelo para essa classe de linguagens, porém em um primeiro momento não é óbvio como a definição de funções recursivas pode ser feita através das primitivas do cálculo.

Tipicamente, a definição de funções em linguagens sem suporte à expressões lambda, como C, é feita através de uma declaração (*statement*). Dessa forma, a definição de uma função está sempre acompanhada por um identificador representando o nome da função (isso reflete o fato de que, em baixo nível, toda rotina está associada a um endereço estático em memória, i.e., seu ponto de entrada). Isso torna possível escrever definições recursivas de maneira direta através do uso de autorreferências; veja, por exemplo, a definição da função fatorial na linguagem C abaixo:

```
unsigned fac(unsigned n) {
    return (n == 0) ? 1 : n * fac(n - 1);
}
```

Em contraste, a definição de funções no cálculo lambda é feita através de expressões lambda. Como as expressões lambda representam funções anônimas, não podemos recorrer à autorreferência da mesma forma que no exemplo anterior. Observe:

$$\text{fac} = \lambda n.\ \text{if}\ (\text{IsZero}\ n)\ \underline{1}\ (\text{mul}\ n\ (\text{fac}\ (\text{sub}\ n\ \underline{1}))) \tag{2.5}$$

Note que a equação acima não serve como uma definição da função fatorial pois o termo à direta da igualdade, quando considerado de forma isolada, não é fechado: a variável fac (que deveria estar ligada à definição da função fatorial) está livre! É possível, porém, reescrever essa equação da seguinte maneira:

$$\text{fac} = F\ \text{fac}, \tag{2.6}$$

$$\text{onde}\ F \equiv \lambda f.\ \lambda n.\ \text{if}\ (\text{IsZero}\ n)\ \underline{1}\ (\text{mul}\ n\ (f\ (\text{sub}\ n\ \underline{1})))$$

Note que o combinador F acima não é autorreferencial, embora a equação como um todo ainda seja recursiva. Desta forma a solução da equação se torna um *ponto fixo* de F, que pode ser encontrado através de um combinador de ponto fixo.

2.3.1 Combinadores de ponto fixo

Definição 2.3.1. *Dizemos que um termo fix é um* combinador de ponto fixo *quando, para qualquer termo F, fix é tal que*

$$F\ (\mathit{fix}\ F) = \mathit{fix}\ F \tag{2.7}$$

Pode não ser óbvio que tal termo exista, mas de fato existem diversas soluções que satisfazem essa equação (BARENDREGT, 1984, p. 42). O exemplo clássico é dado por Curry e Feys (1958) através do seu "combinador paradoxal" Y, que pode ser definido como

$$\mathsf{Y} \equiv \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)) \tag{2.8}$$

Proposição 2.3.2. Y *é um combinador de ponto fixo.*

Como a equação (2.7) é válida para *qualquer* termo F, a existência de combinadores de ponto fixo como o combinador Y nos leva ao seguinte resultado:

Teorema 2.3.3 (Ponto fixo)**.** *Todo termo tem um ponto fixo.*

Em geral, dada uma equação recursiva como (2.5), sempre podemos aplicar uma transformação similar à usada acima de maneira que a solução é dada por um ponto fixo: se g pode ser descrita recursivamente como $g = M$, então $g = \mathrm{fix}\, G$ onde $G \equiv \lambda f.\ M[f/g]$. Dessa forma se torna possível escrever qualquer função (numérica) recursiva através da linguagem do cálculo lambda. No caso de funções mutuamente recursivas, estratégias similares podem ser usadas (BARENDREGT, 1984, Teorema 6.5.2).

2.4 UMA LINGUAGEM DE PROGRAMAÇÃO FUNCIONAL

A Figura 1 descreve a gramática de uma linguagem funcional que usaremos como exemplo prático ao longo deste trabalho. Essa linguagem é uma extensão do cálculo lambda e portanto inclui todas as primitivas desse sistema: $x \in \mathit{Var}$, $(E_0\ E_1)$, e $(\lambda x.\ E)$.

Introduzimos constantes à linguagem através do conjunto *Const* que contém literais representando os números naturais: $\underline{n} \in \mathit{Const}$, para todo $n \in \mathbb{N}$; além dos valores booleanos $\{\underline{\mathrm{tt}}, \underline{\mathrm{ff}}\} \in \mathit{Const}$. Por abuso de notação, escrevemos k no lugar de $\underline{k}$ quando a distinção entre os dois for clara. O conjunto *Primop* contém símbolos representando operações primitivas entre essas constantes, como a adição, subtração, multiplicação, etc. Por questão de legibilidade, também usamos a notação infixa para a aplicação dessas operações binárias quando o significado da expressão é claro (e.g., $(1 + 2) \equiv (\mathrm{add}\ \ 1\ 2)$).

Assim como no cálculo lambda, vamos restringir a linguagem à funções de uma variável por questão de simplicidade; porém, como é bem conhecido, não há perda de expressividade nessa restrição pois podemos converter funções de mais de um argumento em funções de um único argumento (e vice-versa) através de *currying* (e *uncurrying*, respectivamente) (SCHÖNFINKEL, 1924).

Adotamos a estratégia de redução *call-by-value* para a linguagem. Desta forma, a avaliação de uma expressão sempre avalia todas as suas subexpressões, exceto durante a avaliação de expressões condicionais (**if** E_0 E_1 E_2): nesse caso, o resultado da expressão de teste E_0 determina qual dentre as expressões E_1 ou E_2 é avaliada, conferindo o resultado da expressão condicional como um todo.

A expressão (**rec** $x.\ E$) faz o papel do combinador de ponto fixo na linguagem: o resultado de uma expressão (**rec** $x.\ E$) é definido como o ponto fixo da expressão $E[\ \cdot\ /x]$. Também introduzimos expressões da forma (**let** $x = E_0$ **in** E_1) como açúcar sintático para $((\lambda x.\ E_1)\ E_0)$ por questão de conveniência/legibilidade do código.

A escolha dessa linguagem tem o intuito de ser simples, porém representativa de uma linguagem de programação funcional típica. Isso é possível pois, como baseamos essa linguagem no cálculo lambda, ela compartilha do mesmo núcleo funcional de outras linguagens mais complexas. Essa similaridade pode ser observada, por exemplo, na gramática das expressões primitivas da linguagem Scheme (Figura 2), que serviram de inspiração para a nossa linguagem. Também podemos ver essa semelhança com a linguagem Haskell ao examinar a representação intermediária usada para a linguagem durante o processo de compilação (Código 1). É possível notar que, apesar de pequenas diferenças relacionadas à anotação de tipos, a presença/ausência de primitivas de *pattern matching*, etc., o "coração" dessas linguagens ainda é bastante similar.

Figura 1 – Gramática da linguagem

$$
\begin{aligned}
E \in \mathit{Expr} ::= {} & x \quad (\in \mathit{Var}) \\
\mid {} & k \quad (\in \mathit{Const}) \\
\mid {} & o \quad (\in \mathit{Primop}) \\
\mid {} & (E_0\ E_1) \\
\mid {} & (\lambda x.\ E) \\
\mid {} & (\textbf{if}\ E_0\ E_1\ E_2) \\
\mid {} & (\textbf{rec}\ x.\ E) \\
\mid {} & (\textbf{let}\ x = E_0\ \textbf{in}\ E_1)
\end{aligned}
$$

Figura 2 – Sintaxe das expressões primitivas de Scheme (ABELSON et al., 1998)

$$
\begin{array}{lll}
E \in \mathit{Expr} ::= & I & \text{(Variáveis)} \\
& \mid K & \text{(Constantes)} \\
& \mid \texttt{(}E_0\ E^*\texttt{)} & \\
& \mid \texttt{(lambda (}I^*\texttt{)}\ E^*\ E_0\texttt{)} & \\
& \mid \texttt{(lambda (}I^*\ \texttt{.}\ \ I\texttt{)}\ E^*\ E_0\texttt{)} & \\
& \mid \texttt{(lambda}\ I\ E^*\ E_0\texttt{)} & \\
& \mid \texttt{(if}\ E_0\ E_1\ E_2\texttt{)} & \\
& \mid \texttt{(if}\ E_0\ E_1\texttt{)} & \\
& \mid \texttt{(set!}\ \ I\ E\texttt{)} &
\end{array}
$$

Código 1 – Definição simplificada do GHC Core (The GHC Team, 2024)

```
data Program = [Expr]

data Expr
  = Var   Id
  | Lit   Literal
  | App   Expr Expr
  | Lam   Var  Expr
  | Let   Bind Expr
  | Case  Expr Var Type [Alt]
  | Cast  Expr Coercion
  | Type  Type
  -- etc.
```

3 INTERPRETAÇÃO DE UM PROGRAMA

Formalizar uma noção de corretude de uma análise é importante para estabelecer certas garantias a respeito do seu resultado. Se esta análise é usada, por exemplo, como base para a otimização de um programa, é importante saber que as transformações aplicadas a partir do resultado da análise não alteram o comportamento esperado desse programa.

Portanto só é possível estabelecer essa noção de corretude em relação a uma especificação do que, exatamente, se considera o "comportamento esperado" de um programa. No caso do cálculo lambda, por exemplo, isso poderia ser feito ao comparar o algoritmo de análise ao processo de redução-β de termos; porém essa estratégia não seria ideal pois ficaria muito atrelada aos detalhes sintáticos desse processo.

Como nosso foco está no *resultado* de uma expressão, esses detalhes "operacionais" se tornam menos importantes. Por conta disso usaremos o formalismo da *semântica denotacional* para essa tarefa, que modela a interpretação de uma linguagem de maneira mais abstrata. Por outro lado, se o foco da nossa análise estivesse em propriedades mais concretas de um programa como seu tempo de execução, uso de memória, etc. talvez uma outra formalização fosse mais apropriada.

Anteriormente, na Seção 2.2, exploramos brevemente como certos termos no cálculo lambda podem ser interpretados como representações de objetos matemáticos, e.g., $\mathrm{c_n}$ é o numeral de Church que denota o número $n \in \mathbb{N}$, $\mathrm{T} \equiv \lambda xy.\, x$ denota o valor *verdadeiro*, etc. Nesse capítulo examinamos como essa noção da "denotação" de uma expressão pode ser generalizada quando consideramos termos arbitrários do cálculo.

Na Seção 3.1 descrevemos os desafios de se atribuir uma interpretação matemática para os termos do cálculo lambda e, por extensão, para expressões de um programa escrito no paradigma funcional. Na Seção 3.2 desenvolvemos as ferramentas necessárias para realizar essa tarefa e, através dessas ferramentas, construímos um modelo semântico do cálculo lambda (Seção 3.3). Por fim, na Seção 3.4, descrevemos como as mesmas técnicas podem ser usadas para formalizar a semântica de uma linguagem de programação funcional e fornecemos uma interpretação para a linguagem descrita na Seção 2.4.

3.1 SEMÂNTICA DENOTACIONAL

A semântica denotacional busca formalizar a interpretação das expressões de uma linguagem de programação como elementos de um conjunto D, dito o *domínio semântico*.

Escrevemos $[\![E]\!] = d$ para dizer que a expressão E *denota* o valor d, isto é, que $[\![E]\!] = d$ é a *denotação* de E. Dada uma linguagem L e um domínio D, dizemos que $[\![\cdot]\!] : L \to D$ é uma *função semântica* (ou *interpretação*) dessa linguagem. Em geral, queremos que essa

função seja definida de maneira *composicional*, ou seja, que a denotação de uma expressão $[\![E]\!]$ seja determinada a partir da denotação das suas subexpressões.

3.1.1 Valores indefinidos

Intuitivamente, faz sentido que o domínio semântico de uma interpretação deve, no mínimo, incluir denotações para as constantes de interesse: se queremos, por exemplo, interpretar os numerais de Church c_0, c_1, ..., c_n, ... como números naturais, então D deve conter $\mathbb{N}$, isto é, $\mathbb{N} \subseteq D$. De forma similar, se queremos atribuir significado a termos como T (verdadeiro) e F (falso), então D deve conter $\mathbb{B}$, isto é, $\mathbb{B} \subseteq D$.

Além disso, D também deve incluir denotações para funções entre esses valores básicos: se, por exemplo, estamos interessado em interpretar termos como proposições lógicas, então faria sentido imaginar que o domínio semântico deve conter as funções $\mathbb{B} \to \mathbb{B}$ para denotar termos como "not" (2.2); funções $\mathbb{B} \to (\mathbb{B} \to \mathbb{B})$ para termos como "and" e "or" (2.3, 2.4); e assim por diante.

Porém, apesar de intuitiva, essa interpretação dos termos "not", "and", "or", etc. não é apropriada à luz de definições recursivas. Considere, por exemplo, o termo $X \equiv (\mathsf{Y}\ \mathrm{not})$, onde Y é o combinador de ponto fixo definido em (2.8). Nesse caso, se $[\![\mathrm{not}]\!]$ realmente é uma função $\mathbb{B} \to \mathbb{B}$, então $(\mathrm{not}\, X)$ deveria denotar um valor booleano, isto é, $[\![\mathrm{not}\, X]\!] \in \mathbb{B}$. Exceto que, de acordo com a equação (2.7), $X = \mathrm{not}\ X$, o que é uma contradição! Ou seja, $[\![\mathrm{not}\, X]\!] \notin \mathbb{B}$, e portanto a denotação de $[\![\mathrm{not}]\!]$ não pode ser uma função $\mathbb{B} \to \mathbb{B}$. Além disso também precisamos atribuir um significado a termos como Ω, definido na equação (2.1), que não reduzem a uma forma normal.

Para isso introduzimos um valor especial $\bot$, chamado *bottom*, que é usado para denotar expressões com valor *indefinido*. Assim, podemos dizer que $[\![\mathsf{Y}\ \mathrm{not}]\!] = \bot$, $[\![\Omega]\!] = \bot$, etc. Na interpretação de programas reais, esse valor também pode ser usado para denotar erros (e.g., divisão por zero), em caso de loop infinito, etc. (em geral, $[\![E]\!] = \bot$ quando a execução de E diverge de alguma forma). Dado um conjunto V, convencionalmente escrevemos $V_\bot$ para $V \cup \{\bot\}$.

3.1.2 Auto-aplicação

Dessa forma, parece que além dos valores básicos (e funções entre esses valores) o domínio semântico D também deve conter $\bot$. Nesse caso, como $[\![X]\!] = [\![\mathrm{not}\ X]\!] = \bot$, talvez fizesse sentido dizer que a denotação de "not" é, na verdade, uma função $\mathbb{B}_\bot \to \mathbb{B}_\bot$. De fato, em um sistema (ou linguagem) com tipos essa interpretação estaria correta. Porém, no cálculo lambda a aplicação de funções como "not" não está restrita apenas a valores booleanos: nada impede, por exemplo, que se escreva o termo $(\mathrm{not}\ \underline{10})$ aplicando a função "not" a um número natural, ou até mesmo o termo $(\mathrm{not}\, \mathrm{not})$ que aplica a função "not" a si mesma!

Como, de forma geral, termos podem ser aplicados a termos de maneira arbitrária, parece que também precisamos que o domínio D contenha todas as funções $D \to D$ (simbolicamente escrito $D^D \subseteq D$, onde D^D denota o conjunto das funções $D \to D$). Porém, devido ao Teorema de Cantor, isso não é possível para uma escolha não trivial de D (isto é, quando $|D| > 1$), pois nesse caso $|D| < |2^D| \leq |D^D|$. Em outras palavras, a cardinalidade de D^D é estritamente maior que a de D, assim $D^D \not\subseteq D$. Dessa forma, precisamos tomar D como algum *subconjunto* das funções $D \to D$. Como veremos mais a frente, uma escolha apropriada para esse subconjunto são as funções *Scott-contínuas*.

3.2 TEORIA DE DOMÍNIOS

Nessa seção definimos os conceitos básicos da teoria de domínios necessários para contextualizar o resto deste trabalho. Em particular, as noções de supremo, completude, e continuidade são definidas no contexto de ordens parciais. Sempre que possível, essas definições são acompanhadas de exemplos práticos motivando suas aplicações para a análise de programas. Para uma exposição mais completa sobre o tema, veja Winskel (1993).

3.2.1 Conjuntos parcialmente ordenados

Definição 3.2.1 (Conjunto parcialmente ordenado)**.** *Dizemos que P é um* conjunto parcialmente ordenado (poset) *quando está associado a uma relação de ordem parcial $\sqsubseteq$. Dizemos que uma relação $\sqsubseteq$ é uma* relação de ordem parcial *quando é*

1. reflexiva*: para todo $p \in P$, $p \sqsubseteq p$;*
2. antissimétrica*: para todos $p, p' \in P$, se $p \sqsubseteq p'$ e $p \neq p'$, então $p' \not\sqsubseteq p$;*
3. transitiva*: para todos $p, p', p'' \in P$, se $p \sqsubseteq p'$ e $p' \sqsubseteq p''$, então $p \sqsubseteq p''$.*

Nesse caso é comum usar a notação $(P, \sqsubseteq)$ para se referir a essa estrutura como um todo.

Dizemos que P é *parcialmente ordenado* pois podem existir elementos $p, p' \in P$ que sejam *incomparáveis* entre si, isto é, $p \not\sqsubseteq p'$ e $p' \not\sqsubseteq p$. Caso não existam tais $p, p' \in P$, $(P, \sqsubseteq)$ é dito um conjunto *totalmente ordenado*.

Exemplo. O conjunto dos intervalos da forma $[a, b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$ para $a, b \in \mathbb{Z}$, $a \leq b$, forma um poset quando ordenado por inclusão, isto é, $[a, b] \subseteq [a', b']$ sse. $a' \leq a$ e $b \leq b'$. Esse conjunto *não é* totalmente ordenado pois existem intervalos incomparáveis entre si, por exemplo: $[1, 3]$ e $[2, 4]$ são incomparáveis pois $[1, 3] \not\subseteq [2, 4]$ e $[2, 4] \not\subseteq [1, 3]$.

Definição 3.2.2 (Supremo)**.** *Dado $X \subseteq P$, um subconjunto de um poset $(P, \sqsubseteq)$, dizemos que um elemento $p \in P$ é* limitante superior *de X quando $x \sqsubseteq p$ para todo $x \in X$. Além disso chamamos p de* supremo *de X quando é o menor limitante superior de X, isto é,*

quando $p \sqsubseteq p'$ para todo $p' \in P$, p' limitante superior de X. O supremo de X pode não existir; porém, caso exista, ele é único: nesse caso escrevemos $\bigsqcup X$ para o supremo de X.

Definição 3.2.3 (Cadeia)**.** *Um subconjunto $X \subseteq P$ de um poset $(P, \sqsubseteq)$ é chamado de* cadeia *quando é totalmente ordenado, isto é, quando $x \sqsubseteq x'$ ou $x' \sqsubseteq x$ para todos $x, x' \in X$. Em particular, qualquer sequência crescente $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$ em P é uma cadeia pois, para quaisquer $m, n \in \mathbb{N}$, temos que $x_m \sqsubseteq x_n$ sempre que $m \leq n$.*

Exemplo. $\emptyset \subseteq \{1\} \subseteq \{1, 2\} \subseteq \ldots$ é um cadeia em $\wp(\mathbb{N})$. O supremo dessa cadeia é $\mathbb{N}$.

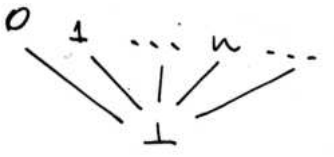
3.2.2 Domínios

Definição 3.2.4 (Domínio)**.** *Dizemos que um poset $(D, \sqsubseteq)$ é* completo para cadeias (cpo) *quando toda cadeia $X \subseteq D$ tem um supremo. Em particular, isso implica a existência de um elemento $\bot = \bigsqcup \emptyset$, dito o* menor elemento *de D, onde $\bot$ é tal que $\bot \sqsubseteq d$ para todo $d \in D$. Nesse caso $(D, \sqsubseteq)$ também é chamado de* domínio*.*

No contexto da semântica denotacional, onde o domínio $(D, \sqsubseteq)$ representa o *domínio semântico*, estamos interessados na interpretação da relação $\sqsubseteq$ como uma espécie de *ordenação por aproximação*: isto é, se d e d' são denotações, então interpretamos $d \sqsubseteq d'$ como indicação de que d' é uma denotação *mais informativa* (ou *mais definida*) do que d.

Exemplo. Em geral, dado um conjunto S qualquer, temos que $S_\bot = S \cup \{\bot\}$ é um domínio quando ordenado de forma que, para todos $d, d' \in S_\bot$, $d \sqsubseteq d'$ sse. $d = \bot$ ou $d = d'$. Note que essa ordenação é *distinta* de qualquer ordenação preexistente em S: se tomamos $S = \mathbb{N}$, por exemplo, então $n \not\sqsubseteq m$ sempre que $n \neq m$, mesmo quando $n \leq m$!

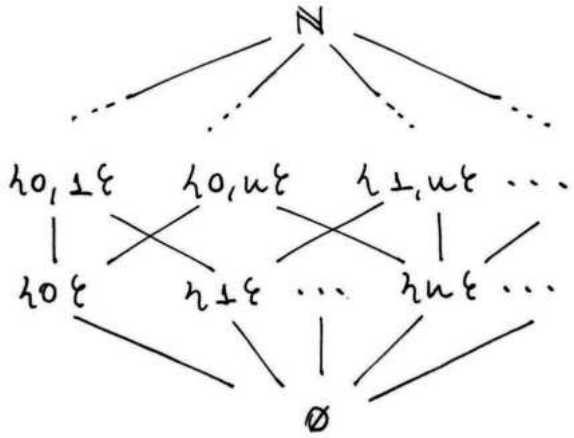
Intuitivamente, se interpretamos $n \sqsubseteq m$ como "m é *mais informativo* que n", então faz sentido que $n \neq m$ sejam incomparáveis pois representam informações *incompatíveis* entre si; por outro lado, $\bot \sqsubseteq n$ pois $\bot$ representa um valor indefinido. Podemos ilustrar essa ordenação através de um *diagrama de Hasse*:



Neste tipo de diagrama, a presença de uma aresta entre dois elementos d e d' (onde d está localizado abaixo de d' no diagrama) indica que $d \sqsubseteq d'$.

Exemplo. O conjunto das partes (também chamado *powerset*) de um S qualquer, dado por $\wp(S) = \{X \mid X \subseteq S\}$, forma um domínio $(\wp(S), \subseteq)$ quando ordenado por inclusão. O menor elemento de $\wp(S)$ é o conjunto vazio $\emptyset = \bigcup \emptyset$ e o maior elemento é o próprio S. O diagrama na Figura 3 ilustra a ordenação desse domínio para o caso em que $S = \mathbb{N}$.

Figura 3 – Diagrama de Hasse para o domínio $(\wp(\mathbb{N}), \subseteq)$



Exemplo. Se $(D_1, \sqsubseteq_1)$ e $(D_2, \sqsubseteq_2)$ são domínios, então seu produto cartesiano

$$D_1 \times D_2 = \{(d_1, d_2) \mid d_1 \in D_1,\ d_2 \in D_2\}$$

também é um domínio quando ordenamos as tuplas (d_1, d_2) de acordo com a ordem de produtos, isto é, quando $(d_1, d_2) \sqsubseteq (d_1', d_2')$ sse. $d_1 \sqsubseteq_1 d_1'$ e $d_2 \sqsubseteq_2 d_2'$. Em geral, se $(D_1, \sqsubseteq_1)$, $(D_2, \sqsubseteq_2)$, ..., $(D_n, \sqsubseteq_n)$ são domínios, então $D_1 \times D_2 \times \cdots \times D_n$ também é um domínio quando ordenado de forma análoga.

O produto cartesiano de domínios pode ser usado na modelagem de valores compostos como estruturas de dados, tuplas, etc.

3.2.3 Funções Scott-contínuas

Definição 3.2.5 (Função monótona)**.** *Dados dois posets $(P, \sqsubseteq_P)$ e $(Q, \sqsubseteq_Q)$, dizemos que uma função $f : P \to Q$ é* monótona *quando, para todos $p, p' \in P$, temos que*

$$p \sqsubseteq_P p' \implies f(p) \sqsubseteq_Q f(p').$$

Definição 3.2.6 (Função Scott-contínua)**.** *Dados dois posets P e Q, dizemos que uma função monótona $f : P \to Q$ é* Scott-contínua *quando, para toda cadeia $X \subseteq P$ com supremo em P, temos que*

$$f(\bigsqcup X) = \bigsqcup \{f(x) \mid x \in X\}.$$

Escrevemos $[P \to Q]$ para denotar o conjunto das funções Scott-contínuas de P para Q.

Proposição 3.2.7. *Se $(P, \sqsubseteq_P)$ e $(Q, \sqsubseteq_Q)$ são domínios, então $[P \to Q]$ também é um domínio dada a ordenação de funções ponto-a-ponto, isto é, quando para quaisquer $f, f' \in [P \to Q]$ temos que*

$$f \sqsubseteq f' \iff \forall p \in P,\ f(p) \sqsubseteq_Q f'(p).$$

É fácil mostrar que nesse caso o menor elemento $\bot$ é tal que $\bot(p) = \bot_Q$ para todo $p \in P$.

Teorema 3.2.8 (Teorema do Ponto Fixo de Kleene)**.** *Se* $(D, \sqsubseteq)$ *é um poset completo para cadeias (i.e., um domínio) e* $f \in [D \to D]$ *uma função Scott-contínua, então* f *tem um ponto fixo* $d = f(d)$ *em* D*. Nesse caso é possível obter o* menor *ponto fixo de* f*, escrito* $\operatorname{lfp} f$*, como o supremo da cadeia* $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \ldots \sqsubseteq f^n(\bot) \sqsubseteq \ldots$*, isto é,*

$$\operatorname{lfp} f = \bigsqcup \{f^n(\bot)\}.$$

3.3 SEMÂNTICA DO CÁLCULO LAMBDA

No cálculo lambda, a denotação de um termo fechado E é sempre fixa $[\![E]\!] = k \in D$, independente do contexto em que ele aparece. Porém isso não é verdade na presença de variáveis livres; nesse caso, o significado de um termo é contextual. Por exemplo: o termo (add x $\underline{3}$), quando considerado isoladamente, não tem uma denotação numérica clara. Porém, no contexto da expressão $((\lambda x.\ \text{add}\ x\ \underline{3})\ \underline{10})$, o mesmo termo denota o valor 13. Assim, no caso geral, se torna necessário fornecer uma *atribuição de valores* para que seja possível determinar a denotação de uma expressão de forma inequívoca. Quando usada na interpretação de uma expressão E, uma atribuição ρ associa um valor $\rho(x) \in D$ a cada variável livre $x \in \mathrm{FV}(E)$ da expressão.

Definição 3.3.1. *(Atribuição) Uma* atribuição de valores *ou* valoração *é uma função* $\rho : \mathit{Var} \to D$ *que associa um valor* $d = \rho(x)$*,* $d \in D$*, a cada variável* $x \in \mathit{Var}$*.*

A atribuição de valores modela o processo de substituição de variáveis usado na redução-β: da mesma forma que um *redex* $((\lambda x.\ M)\ N)$ reduz para o termo $M[N/x]$ através da substituição de x por N, a denotação de $((\lambda x.\ M)\ N)$ dada uma atribuição ρ, $[\![(\lambda x.\ M)\ N]\!]\rho$, é a denotação de M quando atribuímos o valor de $[\![N]\!]\rho$ à variável x, i.e.,

$$[\![(\lambda x.\ M)\ N]\!]\rho = [\![M]\!]\rho[x \mapsto d].$$

Aqui usamos a notação $\rho[x \mapsto d]$ para representar a atribuição de variáveis ρ', dita a *extensão* de ρ, onde ρ' é tal que $\rho'(y) = d$ quando $x = y$, caso contrário $\rho'(y) = \rho(y)$. Por pequeno abuso de notação, às vezes escrevemos $[\![E]\!]$ no lugar de $[\![E]\!]\rho$ quando a denotação da expressão E independe da atribuição ρ ou quando a atribuição é óbvia dado o contexto.

Dessa forma, temos que a função semântica para o cálculo lambda deve ser do tipo

$$[\![\ \cdot\ ]\!] : \mathit{Term} \to (\mathit{Var} \to D) \to D.$$

Porém, como mencionado anteriormente, nem toda escolha de D como domínio semântico basta: devido à auto-aplicação de termos, é necessário que D seja "equivalente" a algum subconjunto das funções $D \to D$. Além disso, queremos que a nossa interpretação do cálculo lambda atenda ao Teorema 2.3.3, isto é, que a denotação de todo termo $\phi = [\![M]\!]\rho$ tenha um ponto fixo em D. O Teorema 3.2.8 sugere uma possível solução a esse

problema: se tomamos D como o conjunto das funções Scott-contínuas $D = [D \to D]$, então ϕ sempre teria um ponto fixo $d = \phi(d)$, $d \in D$. Através desse teorema também ganhamos uma forma de obter um valor para esse ponto fixo como o supremo da sequência $\bot \sqsubseteq \phi(\bot) \sqsubseteq \ldots \sqsubseteq \phi^n(\bot) \sqsubseteq \ldots$

Nesse caso, ao contrário do que acontece quando tentamos tomar D como o conjunto de *todas* as funções $D \to D$, é possivel construir um domínio D tal que $D \cong [D \to D]$ (SCOTT, 1970). Intuitivamente, D é o limite de uma sequência de domínios $D_0 \subseteq D_1 \subseteq \ldots \subseteq D_n \subseteq \ldots$ onde $D_{n+1} = [D_n \to D_n]$: isto é, o próprio D é uma espécie de ponto fixo! Por abuso de notação, tratamos como se $D = [D \to D]$ ao longo desse trabalho, embora isso não seja estritamente verdade (tecnicamente D é *isomórfico* a $[D \to D]$, não igual).

A Figura 4 descreve a semântica do cálculo lambda para esse domínio $D = [D \to D]$: a denotação das variáveis, por definição, é dada de acordo com a atribuição ρ; a denotação de uma abstração lambda $(\lambda x.\, M)$ é uma função que recebe um argumento $d \in D$, atribui seu valor à variável x, e retorna a denotação de M nesse contexto; a denotação da aplicação de dois termos $(M\ N)$ é dada pela aplicação da denotação de M à de N.

Figura 4 – Semântica denotacional do cálculo lambda

$$[\![x]\!]\rho = \rho(x)$$
$$[\![\lambda x.\, M]\!]\rho = \phi, \text{ onde } \phi(d) = [\![M]\!]\rho[x \mapsto d]$$
$$[\![M\ N]\!]\rho = ([\![M]\!]\rho)([\![N]\!]\rho)$$

3.4 SEMÂNTICA DE LINGUAGENS FUNCIONAIS

Assim como acontece no cálculo lambda, a interpretação das expressões em uma linguagem de programação funcional (pura) é contextual, dependendo de uma valoração ρ (porém nesse contexto é mais comum chamar ρ de *ambiente de variáveis*). Dessa forma, a função semântica da nossa linguagem deve ser do tipo

$$[\![\cdot]\!] : \mathit{Expr} \to (\mathit{Var} \to D) \to D.$$

Note que, na presença de efeitos colaterais, *não* é verdade que uma expressão fechada é sempre constante: nesse caso, devido à mutação, operações de I/O, etc., a denotação de uma expressão pode depender do estado global do programa. Normalmente isso requer a introdução de parâmetros adicionais à função semântica; porém, como nossa linguagem é pura, isso não é necessário. Dessa forma, a escolha de uma linguagem pura torna a formalização da sua semântica mais simples em comparação a uma linguagem impura.

O domínio semântico da interpretação D é construído de maneira similar ao domínio usado na interpretação do cálculo lambda, porém nesse caso costuma ser útil distinguir

entre o conjunto de valores básicos da linguagem V e o conjunto das funções $[D \to D]$. Isso se torna possível se definimos D como a *soma* de $V_\perp$ com $[D \to D]$ (SCOTT, 1970):

$$D = V_\perp + [D \to D].$$

Nessa definição, um elemento $d \in D$ do domínio semântico pode ser um valor básico $d \in V$, um valor indefinido $d = \perp$, *ou* uma função $d \in [D \to D]$. Dessa forma, a partir do conjunto de valores básicos V da linguagem, temos em D um domínio apropriado para a interpretação. Em particular, para a linguagem definida na Seção 2.4, temos que o conjunto de valores básicos é dado por $V = \mathbb{N} + \mathbb{B}$.

3.4.1 Semântica de uma linguagem exemplo

A semântica da linguagem definida na Seção 2.4 é resumida através da Figura 5. Como no cálculo lambda, a denotação das variáveis é determinada de acordo com o ambiente ρ. A denotação das expressões lambda também é similar; porém, devido ao fato que nossa linguagem utiliza a estratégia *call-by-value*, toda função é estrita em seu argumento. Dessa forma, se $\phi = [\![\lambda x.\ E]\!]\rho$ é a denotação de uma expressão lambda, é sempre verdade que $\phi(\perp) = \perp$. A interpretação da aplicação de funções é definida assim como no cálculo lambda, porém a aplicação de valores básicos (como números naturais ou booleanos) na posição de função é indefinida, isto é, $[\![E_0\ E_1]\!]\rho = \perp$ quando $[\![E_0]\!]\rho \notin [D \to D]$.

A denotação de toda constante $\underline{k} \in \mathit{Const}$ é dada por um $k \in V$. Quando $\underline{n} \in \mathit{Const}$ representa um número, sua interpretação é o valor de $n \in \mathbb{N}$ correspondente. As constantes $\underline{\mathrm{tt}}$ e $\underline{\mathrm{ff}}$ denotam os valores booleanos *verdadeiro* e *falso*, respectivamente.

A denotação de toda operação primitiva $\underline{o} \in \mathit{Primop}$ é dada por uma função Scott-contínua $o : [D \to D]$ que a implementa. A definição usual da operação é extrapolada para que seu domínio/contradomínio sejam D; operações binárias são convertidas para funções de um único argumento por *currying*; por exemplo: se o símbolo $\underline{\mathrm{quot}} \in \mathit{Primop}$ representa a operação que retorna o quociente da divisão entre dois números naturais $\lfloor m/n \rfloor$, então este símbolo está associado à função $\mathrm{quot} : [D \to [D \to D]]$, onde $\mathrm{quot}(m)(n) = \lfloor m/n \rfloor$ para todos $m, n \in \mathbb{N}$, $n \neq 0$. Porém $\mathrm{quot}(m)(n) = \perp$ quando $n = 0$, $n \notin \mathbb{N}$, ou $m \notin \mathbb{N}$.

A interpretação de expressões condicionais (**if** E_0 E_1 E_2) é feita da maneira usual, isto é, $[\![\textbf{if}\ E_0\ E_1\ E_2]\!]\rho = [\![E_1]\!]\rho$ quando $[\![E_0]\!]\rho = \mathrm{tt}$, caso contrário $= [\![E_2]\!]\rho$. Além disso, expressões condicionais são bem-definidas apenas quando $[\![E_0]\!]\rho$ é um valor booleano. Assim, $[\![\textbf{if}\ E_0\ E_1\ E_2]\!]\rho = \perp$ sempre que $[\![E_0]\!]\rho \notin \{\mathrm{tt}, \mathrm{ff}\}$. A denotação de expressões (**rec** $x.\ E$) é, por definição, dada por $[\![\textbf{rec}\ x.\ E]\!]\rho = \mathrm{lfp}\ [\![E]\!]\rho[x \mapsto\ \cdot\ ]$; da mesma forma, $[\![\textbf{let}\ x = E_0\ \textbf{in}\ E_1]\!]\rho = [\![(\lambda x.\ E_1)\ E_0]\!]\rho$.

Figura 5 – Semântica denotacional da linguagem

$$
\begin{aligned}
[\![x]\!]\rho &= \rho(x) \quad (x \in \mathit{Var}) \\
[\![\underline{k}]\!]\rho &= k \qquad (\underline{k} \in \mathit{Const}) \\
[\![\underline{o}]\!]\rho &= o \qquad (\underline{o} \in \mathit{Primop}) \\
[\![E_0\ E_1]\!]\rho &= \begin{cases} ([\![E_0]\!]\rho)([\![E_1]\!]\rho) & \text{se } [\![E_0]\!]\rho \in [D \to D] \\ \bot & \text{caso contrário} \end{cases} \\
[\![\lambda x.\ E]\!]\rho &= \phi, \\
\text{onde } \phi(d) &= \begin{cases} [\![E]\!]\rho[x \mapsto d] & \text{se } d \neq \bot \\ \bot & \text{caso contrário} \end{cases} \\
[\![\textbf{if } E_0\ E_1\ E_2]\!]\rho &= \begin{cases} [\![E_1]\!]\rho & \text{se } [\![E_0]\!]\rho = \mathrm{tt} \\ [\![E_2]\!]\rho & \text{se } [\![E_0]\!]\rho = \mathrm{ff} \\ \bot & \text{caso contrário} \end{cases} \\
[\![\textbf{rec } x.\ E]\!]\rho &= \mathrm{lfp}\ [\![E]\!]\rho[x \mapsto\ \cdot\ ] \\
[\![\textbf{let } x = E_0 \textbf{ in } E_1]\!]\rho &= [\![(\lambda x.\ E_1)\ E_0]\!]\rho
\end{aligned}
$$

Proposição 3.4.1. *A função semântica* $[\![\ \cdot\ ]\!]$ *definida acima é Scott-contínua.*

Demonstração. Por argumento análogo ao usado por Reynolds (1998, Proposição 10.8), isto é, $[\![\ \cdot\ ]\!]$ pode ser escrita como a composição de operações Scott-contínuas como a aplicação de funções, a extensão de uma atribuição de variáveis, *currying* de funções, etc. □

4 ANÁLISE ESTÁTICA DE PROGRAMAS

> *Due to the unsolvability of the halting problem (and nearly any other question concerning program behaviour), no analysis that always terminates can be exact. [...] Consequently, most research in abstract interpretation has been concerned with effectively finding "safe" descriptions of program behaviour, yielding answers which, though sometimes too conservative in relation to the program's actual behaviour, never yield unreliable information. In a formal sense we seek a $\sqsubseteq$ relation instead of equality. The effect is that the price paid for exact computability is loss of precision. (NIELSON; JONES, 1994)*

Algoritmos de análise costumam ser organizados em dois tipos: análises *estáticas* e análises *dinâmicas*. É comum considerar que um algoritmo de análise é dinâmico quando ele recorre à execução do programa, e estático caso contrário; no entanto, essa distinção nem sempre é muito clara: a otimização por *constant folding*, por exemplo, é considerada um algoritmo de análise estática, apesar de executar (em parte) o código do programa.

Seria mais preciso dizer que uma análise dinâmica, de forma geral, busca examinar as propriedades de um programa de maneira amostral, através da instrumentação de sua execução ou técnicas similares. Por outro lado, uma análise estática tem como objetivo inferir propriedades *universais* de um programa ao considerar, simultaneamente, todos os possíveis contextos em que ele pode ser executado.

Podemos descrever algoritmos de análise estática e formalizar uma noção de corretude através de um método chamado *interpretação abstrata* (do inglês *abstract interpretation*). Apesar de se enquadrar como um método de análise estática, a interpretação abstrata faz uma ponte entre esses dois paradigmas de análise de programas ao descrever algoritmos de análise como interpretadores para uma linguagem usando um domínio especial de valores, cujos elementos denotam propriedades a respeito da sua interpretação "padrão". Em geral não é possível determinar essas propriedades de maneira exata, portanto essa interpretação "abstrata" do programa deve, necessariamente, ser uma aproximação da sua interpretação padrão. Assim se torna necessário garantir que essas aproximações são corretas em relação à interpretação padrão do programa.

Consideramos a interpretação abstrata de um programa como correta quando sua resposta pode ser julgada "segura" no sentido de ser uma aproximação conservadora do resultado real do programa, por exemplo: a análise de uma expressão booleana pode concluir que ela é sempre verdadeira, sempre falsa, ou ter resultado inconclusivo. Por outro lado, a análise de uma expressão aritmética pode concluir, por exemplo, que seu resultado é uma constante, que é sempre positivo, que pertence a um intervalo de valores, etc.

4.1 INTERPRETAÇÃO ABSTRATA

A corretude de uma interpretação abstrata é estabelecida em relação a uma interpretação *padrão* do programa. Existem diversos métodos de se formalizar essa interpretação padrão; nesse trabalho, adotaremos a semântica denotacional para realizar essa tarefa.

Como a semântica denotacional especifica a interpretação correta do programa, uma forma de se desenvolver uma análise que concorda com essa interpretação seria de construí-la diretamente a partir dessa interpretação. Porém, a semântica denotacional assume informação total sobre o contexto de execução do programa, o que nem sempre é possível determinar em tempo estático. Se a propriedade de interesse é, por exemplo, o resultado de uma expressão (como no caso da análise de constantes), essa incerteza se manifesta no fato de que nem sempre é possível determinar de forma exata o valor dessa expressão.

Considere, por exemplo, uma função `time()` que retorna a data e hora atual: nesse caso, claramente é impossível determinar o valor de retorno dessa função estaticamente pois isso depende do conhecimento de quando o programa será executado. Outro exemplo seria uma função `read()`, que recebe um valor do usuário através da entrada padrão. Expressões que envolvem operações de I/O são, em geral, sempre dinâmicas.

O valor dessas expressões *dinâmicas* só pode ser determinado em tempo de execução. Dessa forma, pode haver mais de uma (ou potencialmente infinitas) possibilidades de resultado para a expressão quando considerada estaticamente. Assim, em geral devemos considerar o *conjunto* de possíveis valores que uma expressão do programa pode adotar.

Estaticamente, modelamos a interpretação dessas expressões através da *semântica coletiva* do programa. Como nesse contexto as expressões do programa admitem um conjunto de possíveis valores, a semântica coletiva é dada ao considerar, *coletivamente*, todas as possíveis valorações para um determinado trecho do programa.

Formalmente, isso se reflete na adoção de um conjunto de ambientes de variável $R \subseteq (\mathit{Var} \to D)$ como parâmetro da função semântica. Da mesma forma, o resultado dessa função também é dado por um conjunto de denotações. Podemos determinar a semântica coletiva de um programa diretamente a partir da sua semântica *padrão*, que para a análise de constantes tomaremos como $[\![\cdot]\!]$ (Figura 5).

Definição 4.1.1 (Semântica coletiva)**.** *Dada uma função semântica padrão*

$$[\![\cdot]\!] : L \to (\mathit{Var} \to D) \to D$$

de uma linguagem L para um domínio D, sua semântica coletiva é dada por:

$$\{\![\cdot]\!\} : L \to \wp(\mathit{Var} \to D) \to \wp(D)$$
$$\{\![E]\!\}R = \{[\![E]\!]\rho \mid \rho \in R\}$$

Em geral, determinar o resultado dessa função diretamente é inviável pois os conjuntos envolvidos podem ser muito grandes (ou até mesmo infinitos). Dessa forma, se torna

necessário aproximar a solução para que esse problema se torne tratável. Formalizamos essa estratégia de aproximação através de conexões/inserções de Galois.

Definição 4.1.2 (Conexão de Galois)**.** *Sejam $\alpha : P \to P'$ e $\gamma : P' \to P$ um par de funções monótonas entre dois conjuntos parcialmente ordenados $(P, \sqsubseteq)$ e $(P', \sqsubseteq')$. Dizemos que (α, γ) é uma* conexão de Galois *de P' em P quando, para todos $p \in P$, $p' \in P'$, temos que*

$$\alpha(p) \sqsubseteq' p' \iff p \sqsubseteq \gamma(p').$$

Quando $\alpha \circ \gamma$ é a função identidade em P' (isso é, quando $\alpha(\gamma(p')) = p'$ para todo $p' \in P'$), dizemos que (α, γ) é uma inserção de Galois*.*

Dada uma conexão de Galois (α, γ) de P' em P, dizemos que P é o *conjunto concreto* da conexão e P' o *conjunto abstrato*. Dizemos que $\alpha(p)$ é a *abstração* de $p \in P$, portanto α é chamada de *função de abstração*. Analogamente, $\gamma(p')$ é dita a *concretização* de $p' \in P'$, portanto γ é chamada de *função de concretização*.

Intuitivamente, podemos interpretar como se $d' = \alpha(d)$ fosse uma aproximação para o valor de $d \in D$ dentro do conjunto D'. Assim, há (potencialmente) uma perda de informação ao se tomar essa aproximação, ou seja: $d \sqsubseteq \gamma(\alpha(d))$. Seguindo essa mesma interpretação, $\alpha(\gamma(d')) = d'$ sugere que d' é a *melhor aproximação* de $d = \gamma(d')$ em D'.

Exemplo. (Sinais) Um exemplo clássico de conexão de Galois é a abstração de valores inteiros através do seu sinal (positivo ou negativo). Nesse caso, tomamos $(\wp(\mathbb{Z}), \subseteq)$ como o conjunto concreto da conexão e definimos $(\wp(\{-, 0, +\}), \subseteq)$ como o conjunto abstrato. Intuitivamente, essa abstração aproxima um conjunto (potencialmente infinito) de números inteiros $X \subseteq \mathbb{Z}$ através do sinal dos valores que ele contém: "+" para valores positivos ($X \cap \mathbb{Z}_+ \neq \emptyset$), "−" para negativos ($X \cap \mathbb{Z}_- \neq \emptyset$), "0" para zero ($0 \in X$), ou uma combinação dessas possibilidades. A Figura 6 ilustra essa conexão em forma de um diagrama.

Exemplo. (Constantes) Dado um S qualquer, tome o domínio $(\wp(S), \subseteq)$ como o conjunto concreto. Então podemos definir $S' = S \cup \{\bot, \top\}$, ordenado tal que $\bot \sqsubseteq x \sqsubseteq \top$ para todo $x \in S$, como o conjunto abstrato em uma conexão de Galois (α, γ). Nesse caso, as funções de abstração $\alpha : \wp(S) \to S'$ e concretização $\gamma : S' \to \wp(S)$ são dadas por

$$\alpha(X) = \begin{cases} \bot & \text{se } X = \emptyset \\ x & \text{se } X = \{x\} \\ \top & \text{caso contrário} \end{cases} \qquad \gamma(x') = \begin{cases} \emptyset & \text{se } x' = \bot \\ \{x'\} & \text{se } x' \in S \\ S & \text{se } x' = \top \end{cases}$$

Nesse caso, (α, γ) também é uma inserção de Galois. A Figura 7 ilustra essa conexão de Galois para o caso em que $S = \mathbb{N}$.

Exemplo. (Intervalos) Também é possível definir uma conexão de Galois de $([\mathbb{Z}], \subseteq)$ em $(\wp(\mathbb{Z}), \subseteq)$, onde $[\mathbb{Z}] = \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \{\emptyset\}$ é o conjunto dos intervalos (possivelmente semi-abertos) de $\mathbb{Z}$.

Definição 4.1.3. *(Abstração) Sejam P, P', Q e Q' conjuntos parcialmente ordenados, (α, γ) uma inserção de Galois de P' em P, e $f : P \to Q$ uma função monótona. Dizemos que $f' : P' \to Q'$ é uma* abstração segura *ou* correta *de f se, para todo $p \in P$, temos que*

$$f(p) \sqsubseteq \gamma(f'(\alpha(p))).$$

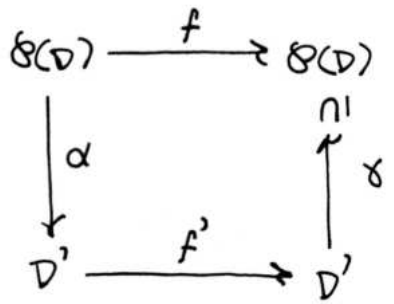
Quando (α, γ) é uma inserção de Galois, esse condição é equivalente a

$$\alpha(f(p)) \sqsubseteq' f'(\alpha(p)). \tag{4.1}$$

Quando $P = Q = \wp(D)$, $f : \wp(D) \to \wp(D)$, como acontece quando f é a denotação de uma função de acordo com a semântica coletiva (por exemplo, quando $f = \{[\![\underline{o}]\!]\}$ para algum $\underline{o} \in \mathit{Primop}$), é possível definir uma abstração segura $f' : D' \to D'$ de f sistematicamente através da composição de α, γ, e f como

$$f' = \alpha \circ f \circ \gamma.$$

Quando definida dessa forma, dizemos que f' é a *melhor abstração* de f, isto é, f' é a abstração *mais precisa* (no sentido de $\sqsubseteq$) de f em $(D' \to D')$ (NIELSON; JONES, 1994). Em forma de diagrama, temos que



Intuitivamente, f' corresponde à estratégia de "testar todas as possibilidades" de valores concretos $d \in \gamma(d')$ que um elemento $d' \in D'$ pode representar. Assim como acontece com $\{[\![\cdot]\!]\}$, em geral o cálculo desse resultado de forma direta não é viável. Nesses casos se torna necessário adotar uma abstração mais aproximada para a função.

Definição 4.1.4. *(Interpretação abstrata) Dada uma função semântica $[\![\cdot]\!] : L \to D$ de uma linguagem L para um domínio D, dizemos que uma função $[\![\cdot]\!]' : L \to D'$ é uma* função semântica abstrata *(ou* interpretação abstrata*) quando $[\![\cdot]\!]'$ é uma abstração segura de $[\![\cdot]\!]$.*

4.2 SEMÂNTICA ABSTRATA DE CONSTANTES

Para o domínio abstrato da interpretação de valores básicos $k \in V$ vamos adotar o domínio de constantes $\widehat{V} = V \cup \{\bot, \top\}$ (vide o exemplo na seção anterior). A partir de $\widehat{V}$ podemos definir $\widehat{D}$, a abstração do domínio $\wp(D)$ de valores da semântica coletiva

Figura 6 – Conexão de Galois de $\wp(\{-, 0, +\})$ em $\wp(\mathbb{Z})$

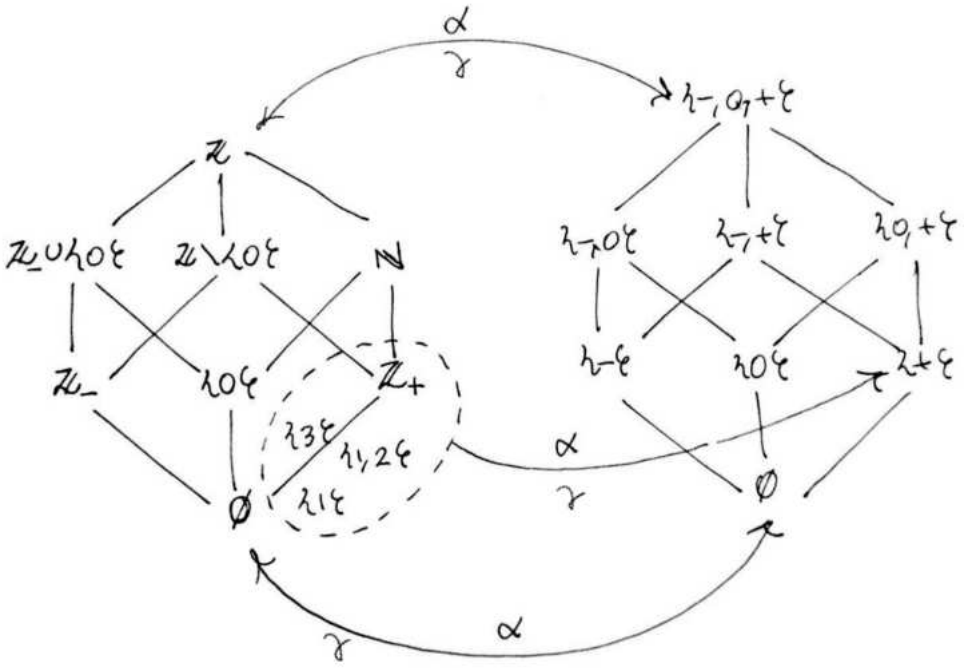
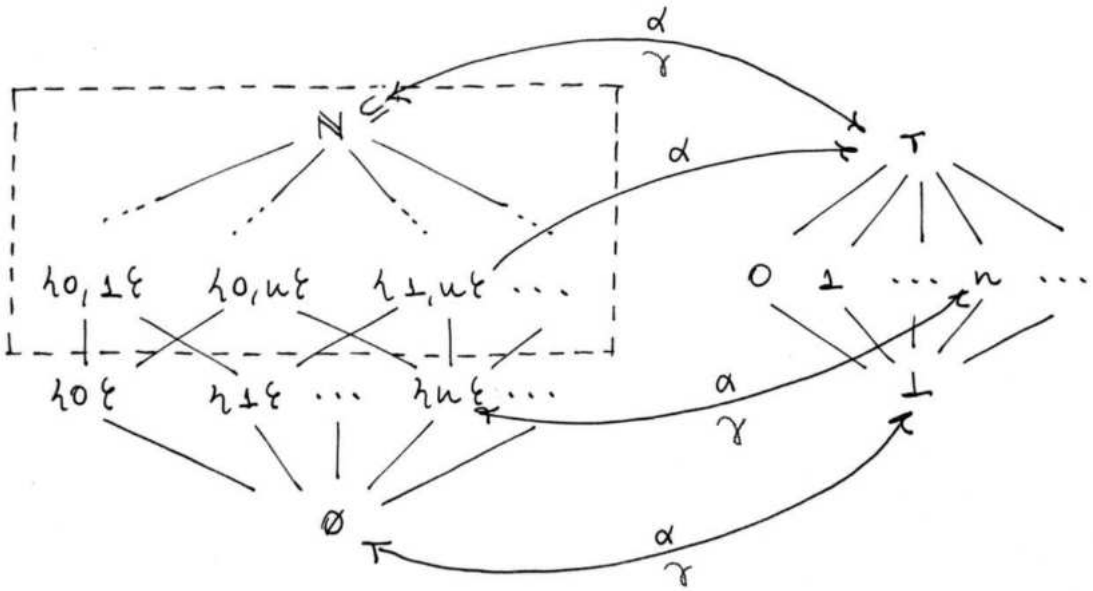


Figura 7 – Conexão de Galois de $\mathbb{N} \cup \{\bot, \top\}$ em $\wp(\mathbb{N})$



$$\widehat{D} = \widehat{V} + [\widehat{D} \to \widehat{D}].$$

Nesse caso, temos que $\top \in \widehat{D}$ é tal que $\top(d) = \top$ para todo $d \in \widehat{D}$. Como o conjunto de constantes da nossa linguagem $V = \mathbb{N} + \mathbb{B}$ é relativamente simples, esse domínio é uma escolha razoável para a abstração de constantes. Porém, para linguagens mais complexas contendo estruturas de dados como tuplas, listas, etc., opções melhores existem (COUSOT; COUSOT, 1994).

A semântica abstrata da linguagem deste trabalho é dada na Figura 8. A interpretação de variáveis, expressões lambda, da aplicação de funções, e das expressões (**rec** $x.\ E$) e (**let** $x = E_0$ **in** E_1) é definida de maneira análoga à sua interpretação padrão (Figura 5).

A abstração das constantes $\underline{k} \in \mathit{Const}$ é dada pela função de abstração: $\widehat{[\![\underline{k}]\!]} = \alpha(\{k\})$. A semântica coletiva de uma operação primitiva $\underline{o} \in \mathit{Primop}$, $\{\![\underline{o}]\!\} : [\wp(D) \to \wp(D)]$, é abstraída por uma função $\hat{o} : [\widehat{D} \to \widehat{D}]$; de forma geral, $\hat{o}$ é definida tal que $\hat{o}(d) = o(d)$ para todo $d \in D$, enquanto que $\hat{o}(\top) = \top$. Na maioria das vezes, essa definição de $\hat{o}$ é uma aproximação razoável da semântica coletiva da operação $\underline{o}$, porém isso nem sempre é verdade: quando $\underline{o} = \underline{\mathrm{mul}}$, por exemplo, $\hat{o} = \widehat{\mathrm{mul}}$ é tal que $\widehat{\mathrm{mul}}(\top)(0) = \top$, embora $\mathrm{mul}(n)(0) = 0$ para todo $n \in \mathbb{N}$. Nesse caso, abstrações mais precisas do que a descrição geral acima podem ser definidas conforme necessário.

A interpretação abstrata de expressões condicionais, por outro lado, merece consideração especial pois é menos óbvia: em uma expressão condicional (**if** E_0 E_1 E_2), quando o resultado da expressão de teste E_0 é uma constante (isto é, quando $\widehat{[\![E_0]\!]}\hat{\rho} \in D$), então a interpretação da expressão condicional é dada de forma análoga à sua interpretação padrão. Porém, quando o valor da expressão de teste é indeterminado (isto é, quando $\widehat{[\![E_0]\!]}\hat{\rho} = \top$), não é possível determinar qual "braço" da expressão condicional (E_1 ou E_2) deve conferir seu resultado.

Nesse caso uma abstração simples seria assumir que o resultado da expressão condicional como um todo é indeterminado. Porém, embora segura, essa interpretação é exageradamente conservadora. É fácil enxergar isso através do seguinte exemplo sintético:

$$\textbf{if } x \ 42 \ 42$$

Nesse exemplo, mesmo quando assumimos que o valor de x é indeterminado (isto é, quando $\hat{\rho}(x) = \top$), é claro de ver que o resultado da expressão é sempre 42. Assim, uma forma mais precisa de definir a interpretação abstrata desse tipo de expressão seria "juntar" o resultado de $\widehat{[\![E_1]\!]}\hat{\rho} = d_1$ com o de $\widehat{[\![E_2]\!]}\hat{\rho} = d_2$. Podemos fazer isso ao tomar o supremo desses dois valores: $d_1 \sqcup d_2 = \bigsqcup\{d_1, d_2\}$.

Figura 8 – Semântica abstrata de constantes da linguagem

$$
\begin{aligned}
\widehat{[\![x]\!]}\hat{\rho} &= \hat{\rho}(x) \qquad (x \in \mathit{Var}) \\
\widehat{[\![\underline{k}]\!]}\hat{\rho} &= \alpha(\{k\}) \quad (\underline{k} \in \mathit{Const}) \\
\widehat{[\![\underline{o}]\!]}\hat{\rho} &= \hat{o} \qquad (\underline{o} \in \mathit{Primop}) \\
\widehat{[\![E_0\ E_1]\!]}\hat{\rho} &= \begin{cases} (\widehat{[\![E_0]\!]}\hat{\rho})(\widehat{[\![E_1]\!]}\hat{\rho}) & \text{se } \widehat{[\![E_0]\!]}\hat{\rho} \in [\widehat{D} \to \widehat{D}] \\ \bot & \text{caso contrário} \end{cases} \\
\widehat{[\![\lambda x.\ E]\!]}\hat{\rho} &= \phi \\
\text{onde } \phi(d) &= \begin{cases} \widehat{[\![E]\!]}\hat{\rho}[x \mapsto d] & \text{se } d \neq \bot \\ \bot & \text{caso contrário} \end{cases} \\
\widehat{[\![\textbf{if } E_0\ E_1\ E_2]\!]}\hat{\rho} &= \begin{cases} \widehat{[\![E_1]\!]}\hat{\rho} & \text{se } \widehat{[\![E_0]\!]}\hat{\rho} = \mathrm{tt} \\ \widehat{[\![E_2]\!]}\hat{\rho} & \text{se } \widehat{[\![E_0]\!]}\hat{\rho} = \mathrm{ff} \\ \widehat{[\![E_1]\!]}\hat{\rho} \sqcup \widehat{[\![E_2]\!]}\hat{\rho} & \text{se } \widehat{[\![E_0]\!]}\hat{\rho} = \top \\ \bot & \text{caso contrário} \end{cases} \\
\widehat{[\![\textbf{rec } x.\ E]\!]}\hat{\rho} &= \mathrm{lfp}\ \widehat{[\![E]\!]}\hat{\rho}[x \mapsto\ \cdot\ ] \\
\widehat{[\![\textbf{let } x = E_0 \textbf{ in } E_1]\!]}\hat{\rho} &= \widehat{[\![(\lambda x.\ E_1)\ E_0]\!]}\hat{\rho}
\end{aligned}
$$

5 COMPUTABILIDADE DA ANÁLISE

A semântica abstrata definida no capítulo anterior (Figura 8) descreve uma espécie de "interpretador abstrato" para a nossa linguagem. Porém a tradução direta das equações apresentadas não produz um algoritmo de análise adequado pois, para muitos dos programas que se deseja analisar, a convergência do algoritmo não é garantida.

Considere, por exemplo, o caso degenerado em que toda variável está associada a um valor constante (i.e., quando $\hat{\rho}(x) \neq \top$ para todo $x \in \mathit{Var}$): nesse caso, a interpretação abstrata do programa é equivalente à interpretação padrão. Desta forma, a execução de um algoritmo ingênuo que implemente diretamente as equações da semântica abstrata irá divergir sempre que o programa analisado P diverge (i.e., quando $\widehat{[\![P]\!]}\hat{\rho} = \bot$), assim como aconteceria com um interpretador normal.

Essa dificuldade de se garantir a computabilidade da análise se torna ainda maior na presença de valores indeterminados: devido à forma como definimos a interpretação abstrata de expressões condicionais, nesses casos a possibilidade de divergência do algoritmo é ainda mais provável. Na ausência de uma estratégia inteligente para tratar da recursão, a análise entraria em loop até nos casos mais triviais. O exemplo a seguir ilustra esse problema:

$$\begin{aligned}&\textbf{let } f = \textbf{rec}\, r.\, \lambda n.\, \textbf{if } (n = 0)\ 0\ (r\ (n - 1))\\ &\textbf{in } \ f\ x\end{aligned}$$

A expressão acima, embora seja computável para qualquer atribuição de x, $\rho(x) \in \mathbb{N}$, ainda assim faz com que a análise entre em loop quando consideramos o valor de x como indeterminado (i.e., quando $\hat{\rho}(x) = \top$). Isso acontece devido ao fato que, ao tentar determinar o resultado da condicional, o algoritmo é forçado a analisar os dois ramos da expressão pois o resultado da expressão de teste é indeterminado (i.e., $\widehat{[\![n = 0]\!]} = \top$). Assim, o algoritmo torna a avaliar a função f, se deparando novamente com a mesma expressão condicional (enquanto que o valor da expressão de teste continua indeterminado).

Uma solução trivial para esse problema seria evitar completamente a análise de chamadas recursivas: o analisador poderia assumir que o resultado de uma chamada de função é indeterminado ($\widehat{[\![E_0\ E_1]\!]} = \top$) sempre que se deparar com uma chamada desse tipo. Nesse caso o resultado da análise seria trivialmente seguro pois $d \sqsubseteq \top$ para todo $d \in \widehat{D}$ (vide a equação 4.1). Porém essa estratégia abre mão de muita informação em nome de garantir a finitude da análise, principalmente quando consideramos o fato que chamadas recursivas são usadas extensivamente no paradigma de programação funcional.

Nossa solução para esse problema mantém a estratégia de execução recursiva da interpretação abstrata, porém garantimos um limite finito para o número de vezes que o

algoritmo visita uma mesma função. Fazemos isso ao nos assegurar que, em caso de chamadas recursivas, o argumento fornecido à função é sempre crescente (Seção 5.1). Depois disso, usamos uma estratégia iterativa para determinar o resultado dessas chamadas recursivas (Seção 5.2). Porém, essas estratégias assumem que o domínio abstrato atende à certas propriedades (que não necessariamente são atendidas na análise de constantes); assim, para o caso geral, se torna necessário adotar aproximações. Nesse caso, propomos o uso de operadores de *widening* como uma possível solução a esse problema (Seção 5.3).

5.1 CONTROLANDO A RECURSÃO

Nossa tentativa de garantir a computabilidade da análise seria inútil a menos que limitássemos de alguma forma o uso da recursão no programa analisado. Caso contrário, ao permitir a execução irrestrita do programa, o tamanho do espaço de configurações (i.e., o número de combinações de função/argumento) que o algoritmo deve considerar pode se tornar potencialmente ilimitado. Considere a análise da seguinte expressão, por exemplo:

$$\begin{aligned} &\textbf{let } g = \textbf{rec } r.\ \lambda n.\ r\ (n+1) \\ &\textbf{in } \ g\ 0 \end{aligned}$$

e tome ψ como a abstração da função g definida acima. Note que, para determinar o resultado de $\widehat{[\![g\ 0]\!]} = \psi(0)$ na expressão acima, o algoritmo primeiro precisa determinar $[\![r\ \widehat{(n+1)}]\!] = \psi(1)$; porém, para obter $\psi(1)$, o algoritmo precisa determinar $\psi(2)$; depois $\psi(3)$, $\psi(4)$, ... e assim por diante. Dessa forma, mesmo que fosse possível computar cada $\psi(n)$ em tempo constante, ainda assim a execução do algoritmo nunca terminaria.

Nossa solução proposta para esse problema é de restringir a chamada de funções recursivas a argumentos crescentes. Fazemos isso ao tomar a união dos argumentos em caso de chamadas consecutivas à mesma função: isto é, se durante o processamento de uma chamada de função $\psi(d)$ o algoritmo encontra uma chamada recursiva $\psi(d')$, então o resultado de $\psi(d')$ é aproximado através de $\psi(d'')$, onde $d'' = d \sqcup d'$. Nesse caso $\psi(d'')$ é uma aproximação correta de $\psi(d')$ pois ψ é uma função Scott-contínua (portanto monótona) e, por definição, $d' \sqsubseteq d''$, dessa forma $\psi(d') \sqsubseteq \psi(d'')$.

De forma concreta, se adotamos essa estratégia durante a análise do exemplo acima, então quando o algoritmo encontra a chamada recursiva $\psi(1)$ durante o processamento de $\psi(0)$ o resultado dessa chamada é aproximado através de $\psi(0 \sqcup 1) = \psi(\top) \sqsupseteq \psi(1)$.

Essa estratégia garante que, ao considerar chamadas de função recursivas, a sequência de argumentos encontrados pelo algoritmo forma uma cadeia crescente $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ que, dada a escolha apropriada do domínio da análise $\widehat{D}$, estabiliza para um valor $d \in \widehat{D}$ após um número finito de passos. Dessa forma, o número de configurações de função/argumento que o algoritmo precisa considerar é finito.

Embora a convergência dessa sequência seja garantida devido ao fato que o domínio $\widehat{D}$ é, por definição, completo para cadeias, não é necessariamente verdade que ela estabiliza

após um $n \in \mathbb{N}$ finito: essa propriedade é dependente da escolha de $\widehat{D}$. Quando $\widehat{D}$ é finito, por exemplo, é fácil ver que essa propriedade é atendida pois nesse caso o tamanho da cadeia é limitado pela cardinalidade de $\widehat{D}$: isto é, $n \leq |\widehat{D}|$. Em geral, dizemos que $\widehat{D}$ deve atender à *condição da cadeia crescente*:

Definição 5.1.1 (Condição da cadeia crescente)**.** *Dizemos que um poset $(D, \sqsubseteq)$ atende à* condição da cadeia crescente (ACC) *(do inglês* ascending chain condition*) quando, para toda sequência crescente $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$ em D, existe um $n \in \mathbb{N}$ tal que $d_m = d_n$ para todo $m \geq n$ (isto é, a sequência eventualmente estabiliza para algum $d_n \in D$).*

5.2 APLICAÇÃO DE FUNÇÕES RECURSIVAS

Como estabelecemos anteriormente (Seção 3.3), em geral a denotação de uma função recursiva $\psi : D$ é dada pelo menor ponto fixo de algum $\phi : [D \to D]$, $\psi = \text{lfp}\ \phi$. Em particular, para a interpretação abstrata da nossa linguagem, $\psi = \widehat{[\![\mathbf{rec}\ x.\ E]\!]}\hat{\rho}$ é definido como lfp ϕ, onde $\phi = \widehat{[\![E]\!]}\hat{\rho}[x \mapsto \cdot\,]$. Porém, apesar de que seja possível obter essa função como o limite da sequência $\bot \sqsubseteq \phi(\bot) \sqsubseteq \phi^2(\bot) \sqsubseteq \ldots$, calcular o valor de ψ dessa forma pode ser computacionalmente muito caro. De fato, para algumas escolhas de domínio da análise $\widehat{D}$, calcular esse limite pode ser impossível pois a sequência $\psi_0 \sqsubseteq \psi_1 \sqsubseteq \ldots \psi_n \sqsubseteq \ldots$, $\psi_n = \phi^n(\bot)$ pode ser infinita. Na verdade, já vimos um exemplo disso de maneira implícita para o domínio D através da denotação da função fatorial (2.6) na interpretação padrão: nesse caso, temos que

$$\psi_0 = \bot,\ \psi_1 = \bot[0 \mapsto 1],\ \psi_2 = \psi_1[1 \mapsto 1],\ \ldots,\ \psi_{n+1} = \psi_n[n \mapsto n!],\ \ldots$$

ad infinitum.

Assim, torna-se necessário adotar uma estratégia alternativa para computar o resultado de chamadas de funções recursivas. Ao invés de determinar ψ globalmente, nossa solução é de determinar o valor de ψ ponto-a-ponto, somente conforme necessário. Dessa forma, se o resultado da função para um argumento $d \in D$ não é significativo para a análise como um todo, então o valor de $\psi(d)$ não é computado. Assumindo que o resultado da análise requer o cálculo de $\psi(d)$ para um número finito de pontos, então ela é computável.

Na seção anterior, descrevemos como podemos limitar o número de configurações de função/argumento que o algoritmo deve considerar durante a análise de um programa. A estratégia descrita garante que, na presença de loops infinitos, a sequência de argumentos que o algoritmo encontra ao processar chamadas de função recursivas eventualmente estabiliza para um $d \in \widehat{D}$. A partir desse ponto, o argumento da função apenas se repete.

Quando isso acontece, o resultado da análise para $\psi(d)$ depende do resultado de $\psi(d)$, recursivamente; ou seja, $\psi(d)$ também é, por si só, o ponto fixo de alguma função ϕ_d. Dessa forma, o valor de $\psi(d)$ pode ser calculado como o limite da sequência

$$\bot \sqsubseteq \phi_d(\bot) \sqsubseteq \phi_d^2(\bot) \sqsubseteq \ldots \sqsubseteq \phi_d^n(\bot) \sqsubseteq \ldots$$

Nesse caso restaria apenas determinar ϕ_d. Propomos que ϕ_d é dado por

$$\phi_d = \phi(\psi[d \mapsto \ \cdot\ ])(d).$$

Proposição 5.2.1. *Dada uma função $\psi : D \cong [D \to D]$ que é o menor ponto fixo de um operador Scott-contínuo $\phi : [D \to D]$, $\psi = \text{lfp}\ \phi$, e um argumento $d \in D$ qualquer, então*

$$\phi_d = \phi(\psi[d \mapsto \ \cdot\ ])(d)$$

é tal que $\phi_d(\psi(d)) = \psi(d)$, isto é, $\psi(d)$ é um ponto fixo de ϕ_d.

Demonstração. Como ψ é ponto fixo de ϕ, então $\psi = \phi(\psi)$; além disso, $\psi = \psi[d \mapsto \psi(d)]$, portanto $\psi = \phi(\psi[d \mapsto \psi(d)])$. Dessa forma temos que

$$\psi(d) = \phi(\psi[d \mapsto \psi(d)])(d) = \phi_d(\psi(d)).$$

Então $\psi(d)$ é um ponto fixo de ϕ_d.

□

Proposição 5.2.2. *$\psi(d)$ é o* menor *ponto fixo de ϕ_d.*

Baseado nessa formulação, podemos adotar uma estratégia iterativa para realizar a análise de chamadas de função $\psi(d)$ quando $\psi = \text{lfp}\ \phi$ é recursiva. Inuitivamente, essa estratégia consiste em "chutar" que o resultado da chamada é $\psi(d) = \bot$ e refinar esse chute progressivamente até alcançar o resultado correto. Isso acontece quando esse "chute" estabiliza, o que indica o fim da iteração. Dadas as mesmas condições adotadas na Seção 5.1 para garantir a convergência da sequência de argumentos de uma função (isto é, quando o domínio abstrato $\widehat{D}$ é finito ou, em geral, quando atende à condição da cadeia crescente), essa estimativa do valor de retorno da função converge para $\psi(d)$ após um número finito de iterações (i.e., existe um n finito tal que $\phi_d^m(\bot) = \psi(d)$ para todo $m \geq n$).

Se adotamos essa estratégia de forma recursiva, se torna possível determinar o valor de $\psi(d)$ para qualquer $d \in \widehat{D}$ sem que seja necessário determinar $\psi = \text{lfp}\ \phi$ diretamente.

Em geral se torna necessário adotar esta estratégia iterativa para determinar o resultado de toda chamada de função durante a análise, mesmo que a função não tenha sido explicitamente declarada como recursiva através de uma expressão ($\mathbf{rec}\, x.\, E$). Isso se deve ao fato de que na nossa linguagem é possível definir funções recursivas indiretamente, assim como acontece no cálculo lambda através de combinadores de ponto fixo como Y.

5.3 OPERADORES DE *WIDENING*

A combinação das estratégias descritas acima garante que o algoritmo de análise termina em tempo finito para certas escolhas do domínio $\widehat{D}$ (i.e., quando $\widehat{D}$ atende à ACC). Porém, nem sempre é possível (ou desejável) restringir o domínio da análise dessa forma.

Em particular, no caso do domínio de constantes $\widehat{D}$ (Seção 4.1), essas condições não são atendidas pois de fato existem cadeias infinitas $d_0 \sqsubset d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq d_{n+1} \sqsubseteq \ldots$ em $\widehat{D}$.

É fácil ver isso quando consideramos que $\widehat{D}$ é na verdade um *superconjunto* de D: como $\widehat{D} = \widehat{V} + [\widehat{D} \to \widehat{D}]$, $D = V_\perp + [D \to D]$, e $\widehat{V} = V_\perp \cup \{\top\}$, então $D \subseteq \widehat{D}$. Assim, qualquer cadeia em D também é uma cadeia $\widehat{D}$. Dessa forma, existem diversos exemplos de cadeias infinitas (em particular, cadeias infinitas de funções) dentro do domínio de constantes $\widehat{D}$. Um exemplo simples é a cadeia $\psi_0 \sqsubseteq \psi_1 \sqsubseteq \ldots$, mencionada na Seção 5.2, cujo limite define a denotação da função fatorial na semântica padrão da linguagem.

Dessa forma, no caso geral se torna necessário adotar uma estratégia alternativa para garantir a convergência dessas cadeias em tempo finito. Esse é um problema bem estudado (COUSOT; COUSOT, 1992) pois se manifesta com frequência durante a construção de análises, até quando tratamos de linguagens imperativas. Embora a presença de funções de alta ordem torne a situação mais complicada, para certas escolhas de domínio abstrato essas cadeias se manifestam até mesmo quando $d \in \widehat{D}$ representa apenas valores básicos. Esse é o caso com o domínio de intervalos (definido na Seção 4.1), por exemplo: é fácil ver que nesse caso cadeias como $[0,1] \subseteq [0,2] \subseteq [0,3] \subseteq \ldots$ nunca convergem. Além disso, mesmo quando essas cadeias convergem, pode ser vantajoso ter um controle maior sobre a velocidade dessa convergência para tornar o algoritmo mais rápido (embora menos preciso).

Nesse caso podemos adotar uma estratégia baseada no uso de operadores de *widening*. Intuitivamente, um operador de *widening* é um operador binário ∇ que se comporta de maneira similar à operação $d \sqcup d' = \bigsqcup\{d, d'\}$, porém enquanto que o supremo de dois elementos $d, d' \in \widehat{D}$ é definido como o *menor* $d'' \in D$ tal que $d \sqsubseteq d''$, $d' \sqsubseteq d''$, o operador ∇ não sofre dessa restrição.

Definição 5.3.1 (*Widening*)**.** *Dado um poset $(P, \sqsubseteq)$, chamamos de* operador de *widening um operador binário $\nabla : P \times P \to P$ tal que*

1. *para todos $p, p' \in P$, $p \sqsubseteq p \nabla p'$;*
2. *para todos $p, p' \in P$, $p' \sqsubseteq p \nabla p'$;*
3. *para toda cadeia $p_0 \sqsubseteq p_1 \sqsubseteq \ldots$ em P, a cadeia definida por $p'_0 = p_0 \sqsubseteq \ldots \sqsubseteq p'_{n+1} = p'_n \nabla p_{n+1} \sqsubseteq \ldots$ não é estritamente crescente.*

Proposição 5.3.2. *Dado um domínio $(D, \sqsubseteq)$, uma função $f \in [D \to D]$ Scott-contínua, e um operador de* widening ∇*, então a cadeia definida por*

$$d_0 = \bot$$
$$d_{n+1} = \begin{cases} d_n & \text{se } f(d_n) \sqsubseteq d_n \\ d_n \nabla f(d_n) & \text{caso contrário} \end{cases}$$

é sempre tal que $\{d_n\}$ *estabiliza após um número finito de termos e seu limite* $d = \bigsqcup\{d_n\}$ *aproxima* $\operatorname{lfp} f$*, isto é,* $\operatorname{lfp} f \sqsubseteq d$.

Demonstração. Vide (COUSOT; COUSOT, 1992, Proposição 33). □

Exemplo. (Intervalos) Podemos definir um operador de *widening* ∇ para o domínio de intervalos $([\mathbb{Z}], \subseteq)$, onde $[\mathbb{Z}] = \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \{\emptyset\}$, como

$$\begin{aligned} \emptyset \nabla I' &= I' \\ I \nabla \emptyset &= I \\ [a, b] \nabla [a', b'] &= [a \text{ se } a \leq a', \text{ caso contrário } -\infty, \\ &\qquad b \text{ se } b' \leq b, \text{ caso contrário } +\infty] \end{aligned}$$

Exemplo. (Constantes) Para o domínio abstrato de constantes $\widehat{D}$, as cadeias infinitas são sempre por elementos de $[\widehat{D} \to \widehat{D}] \cup \{\bot, \top\}$ pois, caso contrário, a maior cadeia que pode ser construida em $\widehat{V}$ é dada por $\bot \sqsubseteq d \sqsubseteq \top$, $d \in V$ (desconsiderando repetições). Dessa forma, um operador de widening simples para esse domínio poderia ser definido como

$$d \nabla d' = \begin{cases} d & \text{se } d = d' \\ d \sqcup d' & \text{se } d, d' \in \widehat{V} \\ \top & \text{caso contrário} \end{cases}$$

Apesar de não ser ideal do ponto de vista da precisão da análise, esse operador de widening bastaria para garantir que qualquer cadeia em $\widehat{D}$ estabiliza em um número de passos finito.

Embora a definição de operadores de *widening* mais precisos que esse seja possível, essa é uma tarefa difícil quando adotamos uma representação abstrata (i.e., *extensional*) para as funções durante a análise, como fizemos neste trabalho. Por outro lado, a definição destes operadores se torna mais fácil para uma implementação mais concreta da análise, pois nesse caso podemos tirar proveito de detalhes internos (i.e., *intensionais*) de representação da função como sua árvore sintática, o ambiente de variáveis capturadas, etc.

Em conclusão, através da aplicação de um operador de *widening* ∇ apropriado, além da adoção das estratégias descritas nas Seções 5.1 e 5.2, podemos garantir que (uma aproximação da) interpretação abstrata $\widehat{[\![\cdot]\!]}$ é computável para qualquer escolha de domínio abstrato, em particular isso também é verdade para o domínio de constantes $\widehat{D} = \widehat{V} + [\widehat{D} \to \widehat{D}]$. Fornecemos uma implementação concreta desta análise para um subset da linguagem Scheme, disponível em https://github.com/thalesfm/racket-analyzer/tree/tcc.

6 CONCLUSÃO

Nosso objetivo com esse trabalho foi de explorar a análise estática de programas e, em particular, de estudar métodos através dos quais podemos pré-computar, de forma automática, parte das expressões de um programa em tempo de compilação. Enquadramos nosso trabalho em torno da análise de propagação de constantes para uma linguagem funcional pura, dinâmica, e também na presença de funções de alta ordem. Ao longo do texto, usamos a construção desta análise como motivação para a introdução do ferramental teórico necessário para a formalizar as técnicas apresentadas, com foco em particular na teoria da semântica denotacional e na análise estática através da interpretação abstrata.

Buscamos expor esses temas de maneira resumida porém acessível, partindo de princípios básicos como os fundamentos da programação funcional através do cálculo lambda, passando pela teoria de domínios e a semântica denotacional, e finalmente pela formalização da corretude de análises estáticas via interpretação abstrata. Contextualizamos nosso trabalho em torno da análise de propagação de constantes, porém sempre que possível procuramos descrever as técnicas mencionadas de forma mais geral de maneira que também sejam aplicáveis a outros tipos de análise estática.

Na parte final do texto, fazemos apontamentos sobre como a descrição abstrata da análise apresentada na parte inicial do trabalho pode ser transformada em um algoritmo concreto por meio de estratégias de aproximação. Nessa parte focamos em tentar garantir, formalmente, a computabilidade desta análise e também rascunhamos uma demonstração da corretude das estratégias apresentadas.

REFERÊNCIAS

ABELSON, H. et al. Revised 5 report on the algorithmic language scheme. **Higher-Order and Symbolic Computation**, Springer, v. 11, p. 7–105, 1998.

APPEL, A. W. **Modern Compiler Implementation in ML**. Cambridge: Cambridge University Press, 1997.

BARENDREGT, H. **The Lambda Calculus, its Syntax and Semantics**. Amsterdã: North-Holland, 1984.

CARDONE, F.; HINDLEY, J. R. History of lambda-calculus and combinatory logic. **Handbook of the History of Logic**, v. 5, p. 723–817, 2006.

CHANG, S.; KNAUTH, A.; GREENMAN, B. Type systems as macros. In: **Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages**. Nova York: Association for Computing Machinery, 2017. (POPL '17), p. 694–705.

CHURCH, A. A set of postulates for the foundation of logic. **Annals of Mathematics**, Annals of Mathematics, v. 33, n. 2, p. 346–366, 1932.

CHURCH, A. A set of postulates for the foundation of logic (second paper). **Annals of Mathematics**, Annals of Mathematics, v. 34, n. 4, p. 839–864, 1933.

CHURCH, A.; ROSSER, J. B. Some properties of conversion. **Transactions of the American Mathematical Society**, JSTOR, v. 39, n. 3, p. 472–482, 1936.

COUSOT, P.; COUSOT, R. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: **Programming Language Implementation and Logic Programming**. Berlim: Springer, 1992. p. 269–295.

COUSOT, P.; COUSOT, R. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages). In: **Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)**. Los Alamitos, EUA: [s.n.], 1994. p. 95–112.

CURRY, H. B.; FEYS, R. **Combinatory Logic**. Amsterdã: North-Holland Publishing Company, 1958. v. 1. (Combinatory Logic, v. 1).

FLATT, M.; PLT. **Syntax Model — The Racket Reference**. 2024. Disponível em: https://docs.racket-lang.org/reference/syntax-model.html. Acesso em: 2024-04-04.

HOWARD, W. A. The formulae-as-types notion of construction. In: **To HB Curry: essays on combinatory logic, lambda calculus and formalism**. Londres: Academic Press, 1980. p. 479–490.

KOHLBECKER, E. et al. Hygienic macro expansion. In: **Proceedings of the 1986 ACM Conference on LISP and Functional Programming**. Nova York: Association for Computing Machinery, 1986. (LFP '86), p. 151–161.

NIELSON, F.; JONES, N. Abstract interpretation: a semantics-based tool for program analysis. **Handbook of Logic in Computer Science**, v. 4, p. 527–636, 1994.

REYNOLDS, J. C. **Theories of programming languages**. Cambridge, Reino Unido: Cambridge University Press, 1998.

SCHÖNFINKEL, M. Über die bausteine der mathematischen logik [Os elementos básicos da lógica matemática]. **Mathematische Annalen**, v. 92, p. 305–316, 1924.

SCOTT, D. **A system of functional abstraction**. 1963. Manuscrito não publicado.

SCOTT, D. **Outline of a Mathematical Theory of Computation**. Oxford: Oxford University Computing Laboratory, 1970.

SESTOFT, P. Demonstrating lambda calculus reduction. In: **The Essence of Computation: Complexity, Analysis, Transformation**. Berlin: Springer Berlin Heidelberg, 2002. p. 420–435.

SHIVERS, O. G. **Control-flow Analysis of Higher-Order Languages, or Taming Lambda**. Tese (Doutorado) — Carnegie Mellon University, 1991.

The GHC Team. **GHC.Core — ghc: The GHC API**. 2024. Disponível em: https://hackage.haskell.org/package/ghc-9.8.2/docs/GHC-Core.html. Acesso em: 2024-03-23.

TURING, A. M. On computable numbers, with an application to the Entscheidungsproblem. **Proceedings of the London Mathematical Society**, s2-42, n. 1, p. 230–265, 1936.

TURING, A. M. Computability and λ-definability. **Journal of Symbolic Logic**, v. 2, n. 4, p. 153–163, 1937.

WINSKEL, G. **The Formal Semantics of Programming Languages: an introduction**. Cambridge, EUA: MIT press, 1993.