SEMÂNTICA DENOTACIONAL ESCALÁVEL DE LINGUAGENS IMPERATIVAS

GUILHERME HENRIQUE DE SOUSA SANTOS

SEMÂNTICA DENOTACIONAL ESCALÁVEL DE LINGUAGENS IMPERATIVAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Roberto da Silva Bigonha
Coorientador: Fabio Tirelo

Belo Horizonte
Março de 2013

Santos, Guilherme Henrique de Sousa
S237s Semântica Denotacional Escalável de Linguagens Imperativas / Guilherme Henrique de Sousa Santos. — Belo Horizonte, 2013
xxvi, 176 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.
Orientador: Roberto da Silva Bigonha
Coorientador: Fabio Tirelo

1. Computação - Teses. 2. Linguagem de programação (Computadores) – Semântica. – Teses. I. Orientador II. Coorientador III. Título.

CDU 519.6*33 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Semântica denotacional escalável de linguagens imperativas

GUILHERME HENRIQUE DE SOUSA SANTOS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ANDRÉ LUIS DE MEDEIROS SANTOS
Departamento de Informática - UFPE

PROF. FABIO TIRELO
Google Inc.

PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 14 de março de 2013.

Agradecimentos

Agradeço a todos que contribuíram para o sucesso da jornada que o mestrado constitui.

Inicialmente aos meus pais, que sempre me apoiaram em todos os empreendimentos. Agradeço a paciência, a motivação e por estarem do meu lado desde os deveres de casa até o vestibular e além. Espero corresponder às suas expectativas.

Aos amigos que compartilharam comigo tantos momentos importantes. À minha antiga turma do CEFET-MG pelo mundo novo que me apresentaram. Aos amigos que conheci na UFMG por terem trilhado comigo grande parte do caminho que levou a este trabalho.

Ao amigo Albert por seguir comigo há tanto tempo. Pela amizade sincera e pelo apoio nos momentos mais difíceis.

Ao amigo Igor por tantas lições importantes, pelas inumeráveis discussões sobre potenciais projetos e pelos produtivos confrontos de ideias.

À minha namorada Gabriela por dividir comigo tantas dores, alegrias e sonhos. E por acreditar em mim.

Ao professor Roberto da Silva Bigonha por ter acolhido o meu interesse no estudo das linguagens de programação, por acreditar no meu trabalho, pelos valiosos conselhos e pelo empenho em produzir um trabalho de alta qualidade.

Ao professor Fábio Tirelo por ter me guiado em vários momentos de dúvidas, por ter apresentado contrapontos maduros quando necessário e pelo entusiasmo durante o desenvolvimento deste trabalho.

Aos colegas do Laboratório de Linguagens de Programação (LLC) pela companhia nessa jornada, pelos risos e muitos cafés nas cantinas do campus.

Aos professores Fernando Magno Quintão Pereira e André Luis de Medeiros Santos por aceitarem participar da banca e pelas valiosas sugestões dadas durante a defesa.

Resumo

Este trabalho de dissertação aborda uma nova solução para o problema de escalabilidade da Semântica Denotacional, intitulada Semântica Denotacional Baseada em Componentes. Essa técnica utiliza uma biblioteca de componentes de semântica denotacional que encapsulam conceitos fundamentais e recorrentes de linguagens de programação imperativas.

Uma das principais funções da biblioteca é remover das equações semânticas a dependência aparente de contexto das construções. Dessa forma, as definições semânticas são definidas pelo mapeamento direto entre as construções da sintaxe abstrata de uma linguagem e as combinações de componentes que modelam sua semântica. O fluxo de informações de contexto é encapsulado pelos componentes.

Como principais contribuições do trabalho, podem-se citar: definição de uma nova metodologia baseada em componentes para obtenção de escalabilidade e reúso em definições de semântica denotacional; a implementação de um ambiente de desenvolvimento para a metodologia, o qual permite a escrita de protótipos de interpretadores para linguagens definidas; identificação e organização dos conceitos fundamentais e recorrentes das linguagens de programação imperativas em uma biblioteca de componentes de definição semântica; abstração do contexto em equações semânticas denotacionais.

Abstract

This dissertation presents a new methodology for the scalability problem in Denotational Semantics, named Component Based Denotational Semantics. This work uses a library of denotational semantics components, which encapsulate fundamental and recurring concepts of imperative programming languages.

One of the main objectives of the library is to remove the apparent dependency on the context of constructs in the composition of their denotations. Thus, semantic definitions map the constructs of the abstract syntax of a language directly to the combination of components which model its semantics. The components encapsulate the flow of context information.

The main contributions of this work are: the definition of a new methodology based on components for the writing of scalable definitions of denotational semantics that promote reuse; the implementation of a development environment for applying the methodology, which generates interpreter prototypes for a language; organization of the fundamental and recurring concepts of imperative programming languages in a library of semantic definition components; abstraction of context in denotational semantics equations.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

SOS semântica operacional estruturada

MSOS semântica operacional modular

I-MSOS semântica operacional implicitamente modular

Listagens

Sumário

Capítulo 1

As Dificuldades de Escalabilidade da Semântica Denotacional

1.1 Introdução

O projeto de linguagens de programação requer a definição de sua sintaxe, semântica e pragmática. Os dois primeiros aspectos dão forma e significado a programas respectivamente, enquanto as boas práticas de uso da linguagem são estudadas pelo terceiro. As definições de sintaxe e semântica possibilitam a implementação de compiladores e interpretadores, além de guiar a escrita de programas. Em geral, a apresentação de uma linguagem em livros e manuais é feita por meio de uma definição formal da sintaxe por meio de gramáticas livres do contexto e uma definição informal da semântica. Mesmo nos casos em que existe uma definição formal da semântica, ela raramente é utilizada efetivamente para comunicação.

No entanto, definições informais de semântica podem ser problemáticas. Por exemplo, considere a definição informal de uma construção apresentada por Gordon (1979), extraída do relatório que define a linguagem PASCAL (Jensen & Wirth 1974): "o comando de atribuição substitui o valor corrente de uma variável por um novo valor especificado por uma expressão". À primeira vista, essa definição parece ser adequada, mas algumas perguntas precisam ser respondidas: *a*) o que é uma variável? *b*) o que é o valor corrente de uma variável? *c*) como um valor pode ser especificado por uma expressão? *d*) a substituição do valor da variável é atômica? Esse pequeno exemplo demonstra a tendência das definições informais a serem ambíguas e incompletas. Isso se deve à grande liberdade que se tem com a informalidade e a falta de rigor.

Para acrescentar rigor às definições de sintaxe e semântica, métodos formais têm

sido desenvolvidos. A forma de Backus-Naur (BNF) é uma notação amplamente utilizada e aceita para a definição formal da sintaxe de linguagens de programação. Por meio do uso de gramáticas livres de contexto, a definição da sintaxe se torna mais clara e precisa. De forma similar, diversas técnicas de Semântica Formal foram criadas para definir a semântica de linguagens de programação. Essas técnicas facilitam a criação de definições livres de ambiguidade e independentes dos detalhes de implementação, além de propiciar o raciocínio confiável acerca de programas e a prova de propriedades (Gordon 1979).

No entanto, diferentemente do que ocorre com a sintaxe, o uso real das diversas metodologias de definição semântica formal é reduzido. Segundo Mosses (2001), uma das principais razões é a dificuldade de se ler e escrever definições semânticas. Embora diversas metodologias tenham sido propostas para resolver esse problema, nenhuma delas se popularizou. Isso se deve ao fato de que nenhuma delas se aproxima da simplicidade de formalismos sintáticos, tais como a de gramáticas livres de contexto.

À luz dos vários formalismos que surgiram para fomentar a definição formal da semântica de linguagens de programação, discutidos mais detalhadamente e em conjunto com suas limitações no Capítulo 2, o presente trabalho tem como objeto de estudo a semântica denotacional. A característica principal desse formalismo, a qual lhe dá nome, é o uso de denotações para definir a semântica das construções. A semântica de cada construção, incluindo-se o programa em si, é definida em sua denotação por meio da semântica de seus constituintes, sendo geralmente cada denotação uma função de ordem mais alta (Mosses 2001).

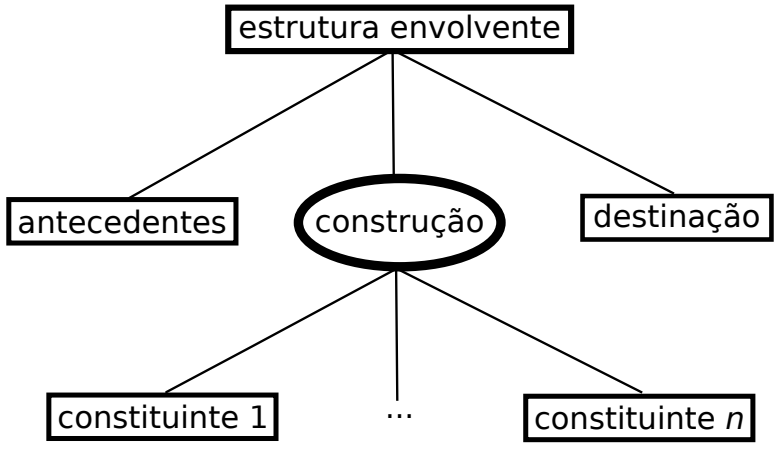

Figura 1.1: O contexto de uma construção é composto por antecedentes, destinação e estrutura envolvente (Tirelo et al. 2005).

No entanto, Tirelo et al. (2005, 2008, 2009) observam que, como esquematizado na Figura 1.1, a semântica de cada construção depende não somente de seus consti-

tuintes, mas também do contexto em que a construção está inserida. O contexto pode ser dividido em: (i) *antecedentes* da construção; (ii) *destinação* da construção e (iii) *estrutura envolvente*. Os *antecedentes* de uma construção compreendem os efeitos dos passos anteriores sobre a sua semântica e são geralmente propagados por estruturas como estados (*stores*) e ambientes (*environments*). Por exemplo, valores atribuídos às variáveis utilizadas fazem parte dos *antecedentes* de uma expressão. A *destinação* de uma construção consiste no passo seguinte à sua execução e é geralmente representada por continuações. Por exemplo, em uma sequência de comandos, no caso mais simples, a *destinação* do primeiro comando é a sequência de comandos que o seguem. Por fim, a *estrutura envolvente* consiste no local na árvore de sintaxe abstrata em que a construção ocorre. Por exemplo, para se definir a semântica do comando *break* é necessário saber se este ocorre em um laço ou em um comando *switch*. E, caso haja aninhamento, é necessário saber qual construção terá sua execução interrompida. Esse grande número de dependências torna as equações semânticas complexas.

Também afetado por uma grande complexidade, o desenvolvimento de grandes sistemas de software é facilitado pelo uso de módulos, responsáveis por esconder detalhes de implementação e organizar o código. Funcionalidades não relacionadas são separadas e unidades coesas podem ser analisadas individualmente. No entanto, a semântica denotacional Clássica não possui notação para expressar módulos. De fato, o λ-cálculo permite apenas a abstração de funções, de forma que não é possível esconder os detalhes internos dos domínios semânticos, os quais são as estruturas de dados utilizadas. Como os domínios são visíveis nas assinaturas das funções, estas dependem da forma como os domínios são implementados. Observada também por diversos autores (Liang et al. 1995, Mosses 1988, 2001, Zhang & Xu 2004), a carência de modularidade é a principal limitação da metodologia clássica de semântica denotacional.

Consequentemente, essa metodologia não é facilmente extensível. Para a inclusão de cada nova construção, alterações podem ser necessárias nas equações semânticas das construções previamente definidas, já que estão intimamente ligadas ao contexto em que se inserem. Casos extremos podem exigir que grande parte das construções sejam redefinidas ao se inserir um novo construto na linguagem. Por exemplo, ao se adicionar um mecanismo de exceções a uma definição, grande parte das equações relacionadas a estruturas de controle de fluxo deverá ser reformulada – deve-se prever, entre outros casos, a interrupção abrupta de laços e, em linguagens como Java e C++, a execução de blocos *finally* antes do retorno de funções. Isso significa que, a menos que a definição semântica da linguagem possa ser feita prevendo-se todas as possíveis demandas futuras, o processo de desenvolvimento envolve diversos ciclos de escrita e reescrita de equações.

Outra consequência de sua pouca modularidade é que a semântica denotacional clássica não favorece o reúso. Isso se deve à possível incompatibilidade entre o contexto da definição que se deseja reutilizar e aquele da nova definição. O consequente esforço de adaptação torna o reaproveitamento de definições inviável, o que é uma grande perda, pois as linguagens de um mesmo paradigma, notoriamente aquelas que pertencem a uma mesma família, possuem muitas construções e funcionalidades semelhantes. Por exemplo, laços *for* estão presentes – com algumas peculiaridades em cada caso – em linguagens tão distintas quanto FORTRAN, Algol, Cobol, ADA, Pascal, C, Smalltalk, Java, PHP, Perl, Python e Ruby. A cada vez que uma nova linguagem é criada, a semântica de suas construções deve ser especificada, mesmo que elas não sejam novas, como é o caso do *for*. Uma metodologia de definição que promova o reúso permite a exploração das mais diversas combinações de construções e a criação de protótipos de linguagens. Dado o grande número de linguagens de programação existentes e semelhanças entre si, as oportunidades para o reúso de construções são muitas.

Finalmente, pequenos exemplos didáticos são legíveis e fáceis de se escrever, dada a sua simplicidade, mas a definição formal de linguagens de grande porte como C++ e Java representa um desafio. Dada a forte dependência entre as construções e o contexto em que estão inseridas, a compreensão de cada equação semântica tende a requerer o entendimento de todos os detalhes envolvidos, até mesmo daqueles que não são diretamente utilizados pela construção considerada. Assim, a legibilidade de definições extensas fica comprometida, criando mais um empecilho no uso das metodologias de definição semântica. Em um primeiro momento, a legibilidade é importante para o projetista, pois torna o trabalho de definição mais natural e facilita o aprendizado do método. Além disso, na maioria das vezes, a definição semântica é escrita com o objetivo de ser lida, utilizada ou, ao menos, verificada por outras pessoas. Uma definição legível é mais acessível, principalmente se a legibilidade for suficiente para permitir a leitura sem treinamento específico. Finalmente, uma definição legível pode ser utilizada como documentação, apresentando a linguagem a programadores e implementadores.

Somada à inviabilidade de reaproveitar partes de definições semânticas disponíveis e a dificuldade de escrever a definição de forma incremental, a baixa legibilidade mostra que a semântica denotacional é pouco escalável. Traçando um paralelo entre sintaxe e semântica, o sucesso das gramáticas como ferramentas de definição sintática mostra que a falta de escalabilidade é o principal obstáculo para a adoção dos métodos denotacionais de definição semântica. Com efeito, uma metodologia escalável torna viável a definição semântica de linguagens de grande porte, as quais são de grande interesse para a indústria. Além disso, é natural que as linguagens evoluam agregando novas construções. Uma metodologia escalável torna esse processo mais fácil, uma vez

que as definições produzidas podem ser gradualmente estendidas.

Diversas abordagens foram propostas para resolver os problemas apresentados nesta seção, as quais são discutidas em maiores detalhes no Capítulo 2. Alguns exemplos são: semântica de ações (Seção 2.3), semântica de mônadas (Seção 2.4), semântica operacional modular (Seção 2.6) e semântica incremental (Seção 2.7). No entanto, as soluções não são definitivas. Por exemplo, a semântica de mônadas encapsula alguns dos domínios semânticos da abordagem denotacional em mônadas, abstraindo seus detalhes das equações semânticas, o que favorece a legibilidade das definições. Todavia, como destacado por Liang et al. (1995), as combinações de transformadores de mônadas podem requerer uma especificação manual – no pior caso, a complexidade dessa tarefa é quadrática no número de transformadores – o que compromete a escalabilidade do modelo.

Com foco na escalabilidade, a semântica incremental (Tirelo et al. 2005, 2008, 2009) acrescenta à semântica denotacional um mecanismo de redefinição cujo objetivo é permitir uma apresentação gradual da semântica de uma linguagem. Dessa forma, a definição semântica pode ser apresentada de forma didática, partindo-se dos conceitos fundamentais e acrescentando detalhes progressivamente, alterando as definições iniciais quando necessário. Comparada à semântica monádica, a abordagem incremental já não possui um custo quadrático de transformação, mas as definições resultantes ainda são de difícil entendimento para leitores não treinados.

De forma similar, Mosses (2004) cria uma abordagem modular de semântica operacional ao encapsular os componentes das transições das regras semânticas. Com essa abstração de contexto, a semântica operacional modular possibilita a escrita de definições extensíveis e favorece o reúso das regras semânticas. No entanto, essa abordagem está restrita à semântica operacional, exceto por seu uso como base da semântica de ações. Essa é uma abordagem híbrida que une a teoria acessível da semântica operacional à elegância da semântica denotacional, buscando melhor legibilidade por meio do uso de uma notação semelhante às linguagens naturais. Porém, essa notação distancia a semântica de ações das abordagens denotacionais, foco deste trabalho.

Conclui-se que ainda há demanda por melhorias. As limitações da semântica de mônadas mostram que a carência de modularidade, escalabilidade e reusabilidade presente na semântica denotacional não está resolvida. E, embora a abordagem incremental favoreça a legibilidade ao organizar as definições em torno dos aspectos transversais, os mecanismos de transformação utilizados podem não ser suficientes para aliviar o custo da criação de diversas versões parciais da mesma definição. Vindas de outra linha metodológica, a semântica operacional modular e a semântica de ações mostram-se opções maduras, mas o tradicional método denotacional se mostrou relevante durante

décadas de pesquisa e não deve ser abandonado.

1.2 Objetivos

Observando esse cenário e inspirado nos avanços das demais abordagens, o objetivo principal deste trabalho é obter um método escalável de semântica denotacional, atacando o problema da falta de modularidade com a remoção da dependência aparente do contexto das construções na redação de suas denotações. Tomando como guia a Figura 1.1, antecedentes, destinação e estrutura envolvente são encapsulados em uma biblioteca e apenas a forma como as denotações podem ser combinadas permanece visível.

O escopo de aplicação do método é limitado às linguagens de programação imperativas, permitindo a escolha de um conjunto de conceitos fundamentais e recorrentes dessas linguagens coeso e ainda assim expressivo o suficiente para definir linguagens inteiras. Essa opção permite também o uso de um modelo semântico bem estabelecido como base da biblioteca e fornece um contexto mais simples para a nomenclatura utilizada. Os conceitos são modelados por uma biblioteca de componentes semânticos, os quais podem ser usados separadamente e compostos de diversas formas. As definições semânticas são organizadas por esses componentes com o objetivo de facilitar sua leitura.

Em resumo, os mecanismos de abstração disponíveis em outras abordagens semânticas servem de inspiração para o desenvolvimento de uma nova abordagem de estruturação de definições semânticas denotacionais baseada em componentes para o paradigma imperativo, discutido em detalhes nos capítulos 3 e 4. Dessa forma, espera-se satisfazer os critérios de avaliação sugeridos por Gayo (2002), Mosses (1988), Moura (1996), a saber: ausência de ambiguidade, possibilidade de demonstração, prototipação, modularidade, reusabilidade, legibilidade, flexibilidade, escalabilidade, abstração e comparação.

1.3 Contribuições do Trabalho

As contribuições deste trabalho podem ser sumarizadas como:

1. um método para remoção da dependência aparente do contexto das construções na redação de suas denotações;

2. identificação e organização dos conceitos fundamentais e recorrentes das linguagens de programação imperativas em uma biblioteca de componentes de definição semântica;

3. aplicação de uma metodologia de composição de definições formais por meio de blocos básicos à semântica denotacional;

4. um novo método para obtenção de escalabilidade e reúso em definições de semântica denotacional;

5. implementação de uma ferramenta para prototipação de linguagens imperativas de porte real.

Organização da Dissertação. O Capítulo 2 discute os mecanismos de abstração em modelos semânticos encontrados em várias abordagens de definição semântica. O Capítulo 3 apresenta a principal contribuição deste trabalho, um método para obter escalabilidade em definições semânticas denotacionais. O Capítulo 4 apresenta uma biblioteca de componentes que implementa esse método. O Capítulo 5 apresenta uma validação do trabalho desenvolvido. Finalmente, o Capítulo 6 apresenta as considerações finais.

Capítulo 2

Mecanismos de Abstração em Modelos Semânticos

Este capítulo apresenta uma descrição geral dos formalismos de definição semântica formal com foco na semântica denotacional. As diversas abordagens são analisadas de modo a expor soluções que serviram de inspiração para este trabalho, mas também salientando as limitações encontradas. Por meio dessa apresentação, explicita-se a motivação para o desenvolvimento de uma nova abordagem denotacional e expor alguns dos conceitos nos quais essa abordagem se baseia.

2.1 Semântica Formal de Linguagens de Programação

Os métodos formais de definição semântica são geralmente classificados pela literatura em três categorias: denotacionais, operacionais e axiomáticos. Todavia, essa divisão não é sempre exata, frequentemente havendo uso de certas características de uma categoria em outra (Zhang & Xu 2004). Há ainda formalismos híbridos como, por exemplo, a semântica de ações, a qual combina estilo denotacional e formalismo operacional. As principais diferenças entre as categorias estão nos mecanismos de definição, e o nível de detalhes dos modelos semânticos produzidos.

Baseada na lógica de predicados, a semântica axiomática proposta por Hoare (1969) define a semântica de uma linguagem por meio de propriedades de suas construções. O objetivo principal dessa abordagem é a prova de propriedades de programas. Dessa forma, as definições semânticas axiomáticas são as mais abstratas, tratando de propriedades das construções, mas sem indicar como a computação se dá.

Uma definição axiomática é construída a partir de axiomas e regras de inferência. Os primeiros são definidos na forma $P\{Q\}R$, sendo Q uma construção da linguagem definida, P uma pré-condição para sua execução e Q a representação de seu efeito. Já as regras de inferência são definidas como:

$$\frac{R_1 \qquad R_2 \qquad \ldots \qquad R_n}{R}$$

em que $R_1, R_2, \ldots, R_n$ são premissas e R é a conclusão da regra.

Regra de condicional:

$$\frac{(B \wedge P)\{Q_1\}R \qquad (\neg B \wedge P)\{Q_2\}R}{P\{if\ B\ then\ Q_1\ else\ Q_2\}R}$$

Figura 2.1: Exemplo de definição de um comando *if-then-else* utilizando semântica axiomática – extraído de Tirelo et al. (2009).

Indo além das propriedades das construções, o enfoque da semântica denotacional é a definição do efeito das computações, mas não a forma como elas são executadas (Scott 1975). Isso permite definir a semântica da linguagem sem se preocupar com detalhes concretos, embora seja possível estruturar a definição de forma a expor sugestões de implementação.

Seguindo esse enfoque, a característica fundamental da abordagem denotacional é definir a semântica por meio do mapeamento dos termos sintáticos em objetos matemáticos chamados denotações. Nessa abordagem, a cada frase válida da linguagem é atribuído um significado ou denotação. Com isso, a semântica dos programas é definida de forma composicional. Cada construção tem sua semântica definida por meio da composição da semântica de seus constituintes. Essa composição é feita tradicionalmente com o uso do λ-cálculo como notação.

Denotação de condicional:

$\mathcal{E}[\![\texttt{if E then C}_1\ \texttt{else C}_2]\!]$ r c = $\mathcal{E}[\![\texttt{E}]\!]$; λb.b $\rightarrow \mathcal{C}[\![\texttt{C}_1]\!]$ r c, $\mathcal{C}[\![\texttt{C}_2]\!]$ r c

Figura 2.2: Exemplo de definição de um comando *if-then-else* utilizando semântica denotacional – baseado em Gordon (1979).

Finalmente, a semântica operacional tem como foco a forma como as computações são executadas. A semântica de um programa é então definida pela composição dos passos necessários para executar cada um dos seus constituintes. Definições operacionais são compostas por sistemas de relações de transição, ou máquinas abstratas. A

característica definidora da semântica operacional é que os estados intermediários que compõem a execução de um programa podem ser identificados. Isso faz com que uma definição operacional seja bastante próxima da implementação, tornando-a um bom guia para o desenvolvimento de compiladores.

Transição de condicional:

$$\frac{\langle e, \sigma\rangle \rightarrow^* \langle true, \sigma'\rangle}{\langle if\ e\ then\ c_1\ else\ c_2, \sigma\rangle \rightarrow \langle c_1, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \rightarrow^* \langle false, \sigma'\rangle}{\langle if\ e\ then\ c_1\ else\ c_2, \sigma\rangle \rightarrow \langle c_2, \sigma'\rangle}$$

Figura 2.3: Exemplo de definição de um comando *if-then-else* utilizando semântica operacional – extraído de Tirelo et al. (2009). Nesse sistema, σ, σ' representam *stores*.

Os três arcabouços semânticos apresentados são complementares. Por exemplo, o desenvolvimento de uma nova linguagem pode utilizar a uma definição axiomática para facilitar a prova de certas propriedades de maior interesse, enquanto uma definição operacional serve de guia para a implementação da linguagem. No entanto, apesar dos avanços na obtenção de expressividade e as bases teóricas para a prova de propriedades, diversos aspectos pragmáticos desfavorecem a adoção em larga escala dos métodos formais de definição semântica. O restante deste capítulo apresenta algumas das principais propostas criadas para sanar as deficiências dos métodos de semântica formal e como essas propostas se relacionam com o presente trabalho.

2.2 Semântica Operacional Estruturada

Proposta por Plotkin (1981), a SOS, é uma das principais abordagens operacionais. Uma definição nesse formalismo é constituída por um conjunto de regras de transição que descrevem os passos de um programa em execução. A interpretação de um programa por meio desse formalismo acontece da seguinte forma: a cada passo, a árvore de sintaxe abstrata e as informações de contexto, como *store* e *environment*, são analisadas de forma a encontrar uma regra de transição aplicável. Caso não haja regra aplicável, a interpretação para. Durante uma computação, parte da árvore de sintaxe abstrata do programa é substituída pelo resultado da computação, enquanto o contexto também é atualizado.

A semântica operacional estruturada é uma técnica poderosa de definição semântica. É possível modelar praticamente todos os recursos presentes nas linguagens de

programação por meio desse formalismo. Especialmente, alguns conceitos difíceis para a semântica denotacional como *interleaving*, ausência de ordem definida de avaliação e paralelismo são facilmente expressos. Por fim, a geração de interpretadores a partir de definições semânticas em SOS é simples.

No entanto, SOS apresenta falta de modularidade causada pela dependência entre as regras de transição e as informações de contexto, as quais são manipuladas por meio de componentes rotulados das relações de transição. Consequentemente e de forma similar à semântica denotacional, todo o contexto deve estar presente em todas as transições, mesmo que parte dele não seja utilizada.

2.3 Semântica de Ações

Durante a segunda metade da década de 1970, a semântica denotacional era considerada a abordagem semântica mais promissora, e imaginava-se que logo todas as linguagens de programação relevantes teriam definições denotacionais de sua semântica (Mosses 1996). No entanto, essa predição não se cumpriu, e a adoção dos métodos de definição semântica continuou baixa. Mosses (1996) argumenta que isso é resultado de dois problemas fundamentais da semântica denotacional: baixa modularidade inerente à sua notação e dificuldade de resgatar conceitos importantes modelados nas definições denotacionais.

Em primeiro lugar, a notação utilizada para definir domínios semânticos não permite abstração. De fato, o λ-cálculo torna as equações semânticas altamente dependentes da estrutura interna dos domínios. Dessa forma, modificar ou estender uma definição semântica, assim como reutilizar fragmentos, exige a reescrita de várias equações, aumentando o custo dessas práticas.

A segunda constatação de Mosses está relacionada com o uso de uma metalinguagem puramente funcional para definição de todos os conceitos de uma linguagem. Segundo esse autor, detalhes importantes são obscurecidos pela notação, e, além disso, a definição de alguns conceitos por meio de funções de ordem mais alta pode não ser natural, dificultando sua compreensão.

Para combater essas deficiências, a semântica de ações (Mosses 1977) foi criada. Esse método é basicamente uma derivação da semântica denotacional, mas as denotações consistem em ações definidas de forma operacional. Com efeito, essa técnica se manteve bastante fiel à semântica denotacional no início, utilizando combinadores para estruturar as definições e evitar a dependência da estrutura dos domínios semânticas. No entanto, valorizando a estruturação das definições semânticas por meio desses

combinadores, o autor decidiu abandonar as definições puramente funcionais e adotar definições operacionais.

Uma definição semântica com ações é estruturada por três elementos: ações, produtores e dados. Estes últimos são as entidades semânticas que modelam conceitos primitivos, como por exemplo números reais e valores booleanos. De forma geral, pode-se imaginar que a função dos produtores é examinar uma estrutura com diversos valores, os quais descrevem o estado corrente e informações de contexto, e extrair algum dado. Já as ações são responsáveis por modelar computações e alterar informação. Os diversos conceitos básicos das linguagens de programação, como por exemplo fluxo de controle, armazenamento e alteração de valores na memória e troca de mensagens, são representados por facetas. Cada ação apresenta uma ou mais facetas, de modo que os diversos tipos de computação podem ser modelados combinando-se ações.

Utilizando o estilo denotacional, uma definição semântica com ações é composta por três seções: uma define a sintaxe abstrata da linguagem, outra define as entidades semânticas e a última define as equações semânticas. Estas são definidas pela composição das ações, fazendo uso da sintaxe abstrata e das entidades semânticas. A Figura 2.4 mostra as equações semânticas que descrevem por meio de ações uma linguagem simples de expressões aritméticas. Nesse exemplo, and e then são combinadores, given é um produtor e give e sum são ações.

Semântica de Expressões:

- evaluate _ :: Expr → Action [giving Integer]

  (1) evaluate ⟦n : Numeral ⟧ = give n

  (2) evaluate ⟦E_1 : Expr "+" E_2 : Expr ⟧ =
  ```
  | evaluate E1
  | and
  | evaluate E2
  then give sum(given Integer#1, given Integer#2)
  ```

Figura 2.4: Exemplo de semântica de ações para uma linguagem simples de expressões aritméticas – extraído de Tirelo et al. (2009).

Um aspecto importante da semântica de ações é o uso de uma notação altamente verbosa (Mosses 1988), lembrando uma versão restrita da língua inglesa. Atribui-se à aparência informal das equações uma grande legibilidade, o que encorajaria os programadores a apreciar a definição, mesmo que de forma superficial. Todavia, um leitor mais interessado poderia compreender mais a fundo a semântica da linguagem,

possivelmente utilizando a interpretação operacional das ações, visto que a definição é formal. Embora a preferência por essa notação mais verbosa ou o sucinto λ-cálculo seja subjetiva, o autor argumenta que os símbolos utilizados ficam a critério do definidor, de forma que notações mais sucintas possam ser utilizadas quando necessário, como quando da construção de provas de teoremas.

Como os combinadores dependem apenas dos detalhes das ações que precisam manipular, há grande liberdade na forma como as ações são compostas. Por exemplo, um combinador que passa o valor computado por uma ação a outra precisa supor que a primeira ação computa um valor, mas não deve fazer qualquer suposição a respeito de efeitos sobre o estado ou lançamento de exceções. Com isso, não só modificações da definição podem ser facilmente introduzidas, mas também sua extensão é fácil, não requerendo a reescrita de equações semânticas. Como a notação suporta o uso de módulos, podem-se criar novas definições reutilizando e estendendo módulos pré-definidos (Doh & Mosses 2003). A interpretação operacional das ações, por sua vez, favorece o uso das definições produzidas com a semântica de ações como guia para a implementação de linguagens.

2.4 Semântica de Mônadas

Como explicado na Seção 1.1, a abordagem clássica de semântica denotacional carece de modularidade. Isso se deve ao fato de que as equações semânticas são definidas em função de informações de contexto comunicadas por meio de parâmetros, e o λ-cálculo não fornece mecanismos de abstração que os escondam. Dessa forma, as equações que utilizam informações de contexto se tornam dependentes da formulação dos domínios semânticos que representam tais informações. Além disso, equações que não fazem uso direto desses parâmetros são também obrigadas a comportá-los, visto que devem passar tal informação aos seus constituintes. Tomemos como exemplo as equações da Figura 2.5, que definem parte de uma simples linguagem de expressões.

$$
\begin{array}{l}
\mathcal{E} : Term \to Env \to (Value \to Ans) \to Ans \\
\mathcal{E}[\![\mathtt{n}]\!]\ \mathrm{r}\ \mathrm{k} = \mathrm{k}\ \mathrm{n} \\
\mathcal{E}[\![\mathtt{i}]\!]\ \mathrm{r}\ \mathrm{k} = \mathrm{k}\ (\mathrm{r}\ \mathrm{i}) \\
\mathcal{E}[\![E_1 + E_2]\!]\ \mathrm{r}\ \mathrm{k} = \mathcal{E}[\![E_1]\!]\mathrm{r};\ \lambda e_1.\mathcal{E}[\![E_2]\!]\mathrm{r};\ \lambda e_2.\mathrm{k}\ (e_1 + e_2)
\end{array}
$$

Figura 2.5: Exemplo de parte da definição semântica de uma linguagem de expressões aritméticas com suporte a variáveis utilizando semântica denotacional clássica – baseado em Gordon (1979) e Liang (1997).

As três equações acima definem a semântica das expressões dessa linguagem de exemplo. Há duas informações de contexto utilizadas: um *environment* e uma continuação. A continuação representa o ponto de destino, sendo portanto necessária em todas as equações. Todavia, o *environment* é utilizado apenas na segunda equação, a qual define a semântica das variáveis, e, ainda assim, todas as equações o recebem. A primeira equação ignora completamente o *environment*, mas, como o tipo que modela as denotações de expressão envolve esse parâmetro, ela também deve recebê-lo. Finalmente, a terceira equação também não utiliza o *environemnt*, mas é obrigada a passar essa informação explicitamente para seus constituintes, os quais obtém os valores dos operandos utilizados na soma modelada por essa equação.

Com o objetivo de obter uma técnica modular, Moggi (1989, 1991) criou uma abordagem de semântica denotacional estruturada em torno de mônadas. Estas são responsáveis por encapsular elementos da linguagem como, por exemplo, *stores*, *environments*, não-terminação, não-determinismo e continuações. As mônadas podem ser combinadas, compondo a semântica de uma linguagem. Enquanto a abordagem tradicional de semântica denotacional mapeia termos de uma linguagem a valores, a semântica monádica mapeia os termos a computações, as quais são modeladas por mônadas.

Uma definição de semântica monádica é estruturada em três camadas. A mais baixa delas é composta por um conjunto de operações básicas que manipulam elementos básicos da linguagem, como, por exemplo, *store* e *environment*. Alguns exemplos de tais operações são a obtenção de valores guardados em um *store* e o *binding* entre um nome e um endereço de memória em um *environment*. Essas operações são utilizadas para dar suporte às mônadas, as quais formam a segunda camada.

Para que a definição seja de fato modular, é necessário que as mônadas modelem conceitos ortogonais, os quais possam ser considerados separadamente quando de uma prova ou da leitura da definição. A composição desses conceitos é feita por meio de transformadores de mônadas. Estes consistem em operações de composição de mônadas e operações de *lift*, responsáveis por prover acesso às operações das mônadas originais após sua composição. Com isso os elementos da linguagem podem ser manipulados separadamente, sem a necessidade de se utilizar diretamente a formulação das mônadas nas equações semânticas. E assim, as equações semânticas, as quais formam a camada superior da definição, podem manipular os elementos da linguagem utilizando apenas as operações básicas, as quais se encarregam de obter as informações necessárias das mônadas. A Figura 2.6 mostra a definição da mesma linguagem da Figura 2.5 utilizando semântica de mônadas.

Como as equações semânticas dependem apenas das operações básicas, diferen-

$$
\begin{array}{l}
\mathcal{E} : Term \rightarrow M\ Value \\
\mathcal{E}[\![\mathtt{n}]\!] = \mathit{return}\ \mathrm{n} \\
\mathcal{E}[\![\mathtt{i}]\!] = \{\mathrm{r} \leftarrow \mathrm{rdEnv};\ \mathrm{r\ i}\} \\
\mathcal{E}[\![E_1 + E_2]\!] = \{e_1 \leftarrow \mathcal{E}[\![E_1]\!];\ e_2 \leftarrow \mathcal{E}[\![E_2]\!];\ \mathit{return}\ (e_1 + e_2)\}
\end{array}
$$

Figura 2.6: Exemplo de parte da definição semântica de uma linguagem de expressões aritméticas com suporte a variáveis utilizando semântica de mônadas – extraído de Liang (1997).

tes mônadas podem ser utilizadas pela mesma definição, resultando em semânticas diferentes. Por isso, a semântica monádica é considerada paramétrica. Quando uma funcionalidade, como por exemplo o suporte a variáveis, deve ser adicionado a uma linguagem, basta adicionar as novas equações e operações que definem a semântica dessa funcionalidade e escrever transformadores que lidem com a nova mônada. A menos que devam fazer uso da nova funcionalidade, as equações semânticas já definidas não precisam ser alteradas. Essa característica favorece a extensibilidade e o reúso, enquanto a legibilidade das definições é beneficiada pelo isolamento de conceitos ortogonais em mônadas e pelas equações semânticas livres de detalhes desnecessários.

No entanto, os transformadores de mônadas são funções complexas, o que dificulta tanto sua escrita quanto sua leitura. Além disso, seu principal papel é tornar as informações de uma mônada acessíveis após sua combinação com outras. Como as mônadas são acopladas formando diversas camadas, o acesso à mônada mais interna deve passar por todas as mônadas superiores. Logo, o número de operações de transformação de mônadas é no pior caso quadrático no número de transformadores. Liang et al. (1995) argumentam que isso não constitui um grande problema, pois, embora muitas vezes esses transformadores devam ser definidos *ad-hoc*, geralmente o número de transformadores usado na prática é pequeno. No entanto, a dificuldade de extensão compromete a escalabilidade do método.

A semântica de mônadas é um método poderoso, capaz de abstrair *stores*, *environments* e outros conceitos importantes das linguagens de programação. O resultado são definições modulares e legíveis, com grande potencial para o reúso. No entanto, o custo da modularidade que confere flexibilidade à técnica é o uso dos complexos transformadores de mônadas, cuja difícil extensão compromete a escalabilidade da semântica monádica.

2.5 Semântica de Ações com Mônadas

A formulação original da semântica de ações inclui uma interpretação operacional das ações utilizando semântica operacional estruturada, a qual possui baixa modularidade. Com isso, extrair partes para reúso ou estender a notação de ações requeria extensa reformulação das regras operacionais. Para resolver esse problema, Wansbrough & Hamer (1997) propuseram a união entre ações e mônadas. No novo método, as primitivas utilizadas para a criação da ações são definidas por meio de mônadas, fornecendo um núcleo modular e facilmente extensível. Além disso, a base matemática que ampara a semântica de mônadas possibilita o raciocínio equacional, facilitando a prova de teoremas. No entanto, ao trocar a semântica operacional estruturada por semântica de mônadas, a expressividade do modelo diminui, tornando, por exemplo, inviável lidar com múltiplos processos que se comuniquem, como apontado por Wansbrough & Hamer (1997).

2.6 Semântica Operacional Modular

Ao expor a falta de modularidade do núcleo da semântica de ações, Wansbrough & Hamer (1997) motivaram Mosses (2004) a melhorar a modularidade da semântica operacional, de modo a manter a interpretação operacional da notação de ações e evitar as desvantagens do uso de mônadas. O resultado é uma técnica que produz definições modulares, extensíveis e facilmente reusáveis. Inspirada no uso de mônadas para trazer modularidade à semântica denotacional, essa nova abordagem de semântica operacional abstrai informações de contexto ao incorporar todas as entidades semânticas nos rótulos das transições. Dessa forma, as configurações utilizadas nas transições envolvem apenas elementos sintáticos e valores computados.

Os rótulos da MSOS possuem diversos componentes que modelam as informações de contexto utilizadas. Tais componentes podem ser lidos e modificados em cada transição, bastando para isso mencioná-los nos rótulos. No entanto, caso os componentes não sejam mencionados, o seu valor é mantido implicitamente durante a transição. Com isso, a regra de transição para uma construção como um *if-then-else*, a qual trabalha apenas com o fluxo de controle, pode ser escrita de forma genérica, independentemente de detalhes como efeito colateral, lançamento de exceções ou interação com processos concorrentes.

A flexibilidade que os rótulos de MSOS conferem à técnica tornam possível um processo incremental de definição semântica, pois novos construtos podem ser adicionados sem a necessidade de redefinição. Além disso, funcionalidades presentes em

diversas linguagens, como a construção *if-then-else* mencionada anteriormente, podem ter sua definição reutilizada sem reformulação (Mosses 2002b). Todavia, o autor alerta para o fato de que mudar o estilo de uma definição em MSOS entre *small-step* (Plotkin 1981) e *big-step* (Kahn 1987) requer extensa reformulação das regras de transição.

Apesar da modularidade alcançada por MSOS e pela semântica de ações, cujo núcleo passou a ser definido em MSOS, Mosses (1999) advoga a preferência pela semântica de ações. Segundo o autor, a principal vantagem da notação de ações é que seus combinadores proveem abreviações concisas para certos padrões de regras de transição. Por exemplo, o combinador para execução sequencial sem fluxo de dados (escrito A_1 `and_then` A_2) abrevia um padrão que ocorre em muitas definições de avaliação feita da esquerda para a direita em MSOS ou SOS. No entanto, a notação utilizada por MSOS, ao se manter próxima à notação tradicional de semântica operacional, pode ser mais familiar do que a notação de ações. Além disso, MSOS possui base teórica mais poderosa do que a semântica de ações (Mosses 1999), sendo portanto mais adequada à prova de propriedades da linguagem definida.

Recentemente, essa abordagem sofreu uma pequena alteração que simplifica sua notação e favorece ainda mais o reúso (Mosses & New 2009). A I-MSOS difere do método anterior na forma como entidades auxiliares não mencionadas nas regras de transição são tratadas. Enquanto MSOS agrupa todas as entidades não mencionadas em um único rótulo, fazendo assim com que seus valores sejam transmitidos sem alteração, I-MSOS estabelece que toda entidade não mencionada em uma regra tem seu valor transmitido implicitamente. Isso torna as regras de transição de I-MSOS consideravelmente mais legíveis.

A Tabela 2.1 mostra uma comparação entre definições semânticas de valores constantes utilizando SOS, MSOS e I-MSOS. No primeiro caso, o estado da máquina, representado por σ e σ', é citado na regra de transição, embora seu valor não influencie a transição. No caso de I-MSOS, a transição inspeciona o valor da constante x no *environment*, mas nenhuma informação de contexto é alterada. Algo semelhante ocorre com a definição em MSOS, mas é necessário explicitar no rótulo da transição que o valor do *environment* no fim da transição e a transmissão das outras informações de contexto sem alterações. A primeira linha de cada exemplo mostra o formato das transições e, no caso de MSOS, o rótulo utilizado.

SOS:	**MSOS:**	**I-MSOS:**
$\boxed{\rho \vdash \langle e, \sigma\rangle \to \langle e', \sigma'\rangle}$	$\boxed{e \xrightarrow{X} e'}$ $\quad Label{=}\{\rho{:}Env, \ldots\}$	$\boxed{\rho \vdash e \to e'}$
$\dfrac{\rho(x) = con}{\rho \vdash \langle x, \sigma\rangle \to \langle con, \sigma\rangle}$	$\dfrac{\rho(x) = con}{x \xrightarrow{\{\rho,-\}} con}$	$\dfrac{\rho(x) = con}{\rho \vdash x \to con}$

Tabela 2.1: Comparação da semântica da avaliação de valores constantes utilizando SOS, MSOS e I-MSOS – extraída de Mosses & New (2009).

2.7 Semântica Incremental

Um problema não resolvido pelas demais metodologias é a dificuldade de se estender uma definição semântica. Para resolver esse problema, a semântica incremental proposta por Tirelo et al. (2005, 2008, 2009) foca na falta de escalabilidade e legibilidade das metodologias de semântica denotacional, desenvolvendo o conceito de vagueza, muito comum na prática de ensino. Quando um professor apresenta um conceito a seus alunos, ele primeiro trabalha o núcleo de ideias e depois acrescenta outros conceitos relacionados de menor relevância. Dessa forma, os alunos concentram-se primeiro nos aspectos mais importantes para depois explorar detalhes.

Na abordagem incremental, a definição semântica é dada por uma sequência de definições e funções de transformação. Cada versão da definição é obtida pela união de novos conceitos e uma transformação aplicada à versão anterior – a função de transformação é responsável por adaptar as demais regras semânticas aos novos conceitos. O ponto forte dessa abordagem é a possibilidade de se organizar os conceitos em módulos transversais às equações semânticas, o que estimula um desenvolvimento incremental da linguagem e torna a definição mais legível. De fato, a abordagem traz uma forma disciplinada de se alterar denotações para inserir novas construções.

Entretanto, a metodologia não é indicada para a prototipação, mas apenas para a apresentação de linguagens completas. Embora as funções de transformação permitam uma definição incremental, a adição de construções não antecipadas pode ser complexa. Conhecer todas as demandas *a priori* permite ao projetista modelar cada versão da definição de forma a simplificar as transformações necessárias.

Finalmente, assim como a programação orientada a aspectos, a semântica incremental dificulta o raciocínio equacional. Ao se fazer demonstrações de propriedades, sugere-se primeiro fazer a demonstração com base na definição inicial e depois verificar a cada transformação quais propriedades da especificação anterior se mantêm válidas.

2.8 Semântica Baseada em Componentes

A natureza modular da semântica de ações inspirou Doh & Mosses (2003) a buscarem uma abordagem de definição semântica focada na definição de pequenos fragmentos de linguagem altamente reutilizáveis, em contraste com o enfoque usual de se definir a linguagem inteira. Dessa forma, a definição da semântica de uma linguagem poderia ser feita por meio da composição de definições menores. Uma analogia razoável pode ser feita com um restaurante em que o cliente monta o próprio prato escolhendo alguns dos vários alimentos disponíveis. De forma semelhante, aproveitando a recorrência de certas construções nas linguagens de programação, uma nova linguagem poderia ter sua semântica definida ao se escolher algumas entre várias construções previamente definidas.

A partir desse primeiro trabalho, uma lista de módulos semânticos vem sendo desenvolvida (Doh & Mosses 2003, Iversen & Mosses 2004, Mosses 2002a). Cada um desses módulos representa uma construção abstrata, modelando uma funcionalidade presente nas linguagens de programação. A definição semântica de uma construção concreta, como um comando *for*, pode ser feita por meio da combinação de uma ou mais dessas construções abstratas, também conhecidas como blocos básicos e, mais recentemente (Johnstone et al. 2010), *funcons* (construtos fundamentais independentes de linguagem). Desde o início, essa lista é vista como um trabalho em aberto, sempre extensível, acomodando novos conceitos à medida que tiverem sua semântica definida.

Em uma definição semântica baseada em componentes, a semântica das construções da linguagem definida é dada por combinações de *funcons*. Os *funcons* por sua vez são definidos por meio de semântica de ações ou semântica operacional modular. Esses métodos de definição semântica foram escolhidos por serem considerados altamente modulares, porém os autores convidam outros pesquisadores a aplicarem a mesma ideia a outros métodos de semântica formal(Mosses 2005). A Figura 2.7 mostra a semântica de uma construção *while* extraída de Johnstone et al. (2010).

$$
\begin{array}{l}
Cmd[\![\texttt{while (}E\texttt{) }C]\!] = \text{catch(cond-loop(}Exp[\![E]\!],\ Cmd[\![C]\!]\text{),} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{abs(eq(breaking), skip))} \\
Cmd[\![\texttt{break}]\!] = \qquad\quad \text{throw(breaking)}
\end{array}
$$

Figura 2.7: Exemplo de definição semântica de um comando *while* utilizando semântica baseada em componentes – extraído de Johnstone et al. (2010).

Inicialmente, essa abordagem foi chamada de semântica construtiva (Mosses 2005), observando a forma como uma definição semântica poderia ser construída a

partir de blocos menores. Posteriormente, o nome foi alterado para semântica baseada em componentes (Mosses 2009), dando enfoque à característica principal do método: o uso de componentes para modelar conceitos básicos das linguagens de programação.

Um dos objetivos dessa abordagem é criar um repositório *online* de *funcons* que possam ser utilizados diretamente em definições semânticas. Como cada *funcon* é definido de forma completamente independente de linguagem, o reúso pode ser feito sem a necessidade de alterações, copiando a definição disponível *online* ou apenas fazendo uma referência a ela. Com isso, espera-se que a Semântica Formal se torne mais acessível, encorajando alunos e pesquisadores a conhecer os construtos básicos e utilizá-los em suas próprias definições semânticas. Em Setembro de 2011, *Swansea University*, *Royal Holloway University of London* e *City University London* deram início a um projeto de pesquisa conjunta intitulado *Programming Language Components and Specifications* (PLanCompS)[1]. Previsto para durar quatro anos, o projeto tem como objetivos o desenvolvimento de uma abordagem de semântica baseada em componentes, o estudo de seu uso prático e o estabelecimento de uma biblioteca de *funcons* a ser disponibilizada em um repositório aberto em formato de biblioteca digital.

2.9 Conclusões

Como apresentado neste capítulo, a semântica de mônadas e a semântica operacional modular estabeleceram abordagens para combater antigas deficiências pragmáticas dos métodos de semântica formal. Ambos promovem a abstração de informações de contexto, aumentando assim a modularidade dos arcabouços dos quais fazem parte. No entanto, ainda não há consenso em relação a um método definitivo de Semântica Formal.

A recente iniciativa representada pela semântica baseada em componentes parece seguir um caminho interessante, rumo a uma técnica mais atraente aos programadores e ainda assim interessante para a academia. Nesse período inicial de desenvolvimento, a oportunidade de estender o alcance dessa técnica se faz mais presente. Consequentemente, este trabalho busca aplicar abstração por meio de componentes à semântica denotacional, visando melhorar sua escalabilidade.

O Capítulo 3 apresenta um método de definição semântica denotacional baseado em componentes. As duas principais fontes de inspiração desse método são a semântica de mônadas e a semântica baseada em componentes. A primeira demonstra que uma técnica denotacional de definição semântica se torna mais legível e modular ao abstrair

[1]Notícias, publicações e outras informações sobre o projeto PLanCompS podem ser encontradas no endereço http://www.plancomps.org/.

informações de contexto, enquanto a segunda apresenta uma organização dos conceitos fundamentais das linguagens de programação capaz de aproveitar a modularidade da técnica de definição subjacente para alcançar um nível elevado de reusabilidade e escalabilidade.

Capítulo 3

Semântica Denotacional Baseada em Componentes

Este capítulo apresenta o método desenvolvido neste trabalho. A solução é apresentada na forma de uma biblioteca, acompanhada do modelo semântico que lhe serve de base. São também apresentados o escopo deste trabalho e o método de criação dos componentes que formam a biblioteca. A linguagem utilizada para descrever semântica denotacional é o λ-cálculo com tipos definido por Gordon (1979).

3.1 Ideia Central

Na abordagem denotacional, a semântica de uma construção é dada pela combinação da semântica de seus constituintes. A Figura 3.1 mostra um esquema dos mapeamentos que modelam a semântica denotacional das construções. Nesse esquema, os terminais da produção são identificados na forma r_i, enquanto os não terminais da construção A são identificados na forma B_i. A semântica da construção A, $h(A)$, é uma função g das denotações de seus constituintes, $h(B_i)$.

No entanto, como mostrado na Figura 1.1, a semântica das construções também pode depender de informações de contexto. Por exemplo, o contexto relevante à semântica de acesso a variáveis geralmente envolve um mecanismo de armazenamento de dados e um ambiente que mapeia nomes de variáveis a posições de armazenamento. Embora o contexto não seja sempre relevante para a semântica de uma dada construção, limitações do λ-cálculo discutidas na Seção 1.1 fazem com que o contexto esteja presente em todas as equações semânticas.

Tome por exemplo a definição semântica de uma linguagem simples inspirada na definição de Tiny encontrada em Gordon (1979) e cuja sintaxe abstrata é apresentada

Produção:

$$p : A \to r_0 B_1 r_1 B_2 \ldots B_n r_n$$

Diagrama de Mapeamentos Semânticos:

$$\begin{array}{ccccccccc} T_{B_1} & \times & T_{B_2} & \times & \ldots & \times & T_{B_n} & \xrightarrow{\Pi_p} & T_A \\ \downarrow h_1 & & \downarrow h_2 & & & & \downarrow h_n & & \downarrow h \\ S_1 & \times & S_2 & \times & \ldots & \times & S_n & \xrightarrow{g} & S \end{array}$$

Mapeamentos Semânticos:

$$\begin{aligned} h(\Pi_p(T_{B_1}, T_{B_2}, \ldots, T_{B_n})) &= g(h_1(T_{B_1}), h_2(T_{B_2}), \ldots, h_n(T_{B_n})) \\ h[\![A]\!] &= g(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!]) \\ h[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] &= g(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!]) \end{aligned}$$

Figura 3.1: Esquema de mapeamentos semânticos da abordagem denotacional – extraído de Bigonha (2004).

pela Figura 3.2. Um programa nessa linguagem é formado por uma série de declarações de variáveis e um comando. Este comando pode ser um laço *while*, uma atribuição ou um comando condicional. As expressão são compostas por acesso a variáveis, literais, somas e testes de igualdade.

Sintaxe Abstrata:

Pro:	Domínio dos Programas	- **P**
Dec:	Domínio das Declarações	- **D**
Com:	Domínio dos Comandos	- **C**
Exp:	Domínio das Expressões	- **E**
Bas:	Domínio dos Valores	- **B**

P	→	'let' **D** 'in' **C**
D	→	**I** '=' **E** |**D**$_1$ ',' **D**$_2$
C	→	**I** ':=' **E** |'if' **E** 'then' **C** |'while' **E** 'do' **C**
E	→	**I** |**B** |**E**$_1$ '+' **E**$_2$ |**E**$_1$ '=' **E**$_2$

Figura 3.2: Exemplo de sintaxe de uma linguagem simples - inspirada na definição de Tiny (Gordon 1979).

A Figura 3.3 apresenta a definição semântica denotacional da linguagem de exemplo. Nessa definição, comandos, expressões e declarações utilizam continuações diferentes, de modo a permitir o fluxo do contexto relevante para cada um. Por exemplo, enquanto declarações precisam transmitir à continuação alterações feitas ao *environment*, comandos e expressões apenas utilizam *environments* recebidos das construções envolventes. Além das continuações, são manipulados *store*, *environment* e valores

básicos. A resposta final é modelada como o *store* válido no fim da execução ou erro.

Domínios Semânticos:

Cc	= Store →Ans	- continuações de comando **c**
Dc	= Env →Cc	- continuações de declaração **u**
Ec	= Bv →Cc	- continuações de expressão **k**
Store	= Loc →[Bv + {unused}]	- *stores* **s**
Env	= Id →[Loc + {unbound}]	- *environments* **r**
Bv	= Num + Bool	- valores básicos **b**
Ans	= {error} + Store	- respostas finais **a**

Equações Semânticas:

$\mathcal{P}$: **Pro** →Ans
$\mathcal{D}$: **Dec** →Env →Dc →Cc
$\mathcal{C}$: **Com** →Env →Cc →Cc
$\mathcal{E}$: **Exp** →Env →Ec →Cc
$\mathcal{B}$: **Bas** →B

$\mathcal{P}[\![$`'let' D 'in' C`$]\!]$ = $\mathcal{D}[\![$`D`$]\!]$ r_0 (λr.$\mathcal{C}[\![$`C`$]\!]$ r c_0) s_0
  where r_0 = λi.**unbound**
  s_0 = λl.**unused**
  c_0 = λs.s

$\mathcal{D}[\![$`I '=' E`$]\!]$ r u = $\mathcal{E}[\![$`E`$]\!]$ r;λb s.u ((new s)/`I`) s[b/(new s)]
$\mathcal{D}[\![\mathtt{D}_1$ `','` $\mathtt{D}_2]\!]$ r u = $\mathcal{D}[\![\mathtt{D}_1]\!]$ r;$\lambda r_1.\mathcal{D}[\![\mathtt{D}_2]\!]$ r[r_1];λr_2.u r[r_1][r_2]

$\mathcal{C}[\![$`'if' E 'then' C`$]\!]$ r c = $\mathcal{E}[\![$`E`$]\!]$ r;cond($\mathcal{C}[\![$`C`$]\!]$ r c,c)
$\mathcal{C}[\![$`'while' E 'do' C`$]\!]$ r c = **fix** (λf r c.$\mathcal{E}[\![$`E`$]\!]$;cond($\mathcal{C}[\![$`C`$]\!]$ r;f r c,c)) r c
$\mathcal{C}[\![$`I ':=' E`$]\!]$ r c = (r `I`) = **unbound** →error,
  $\mathcal{E}[\![$`E`$]\!]$ r;λb s.c s[b/(r `I`)]
$\mathcal{E}[\![$`B`$]\!]$ r k = k $\mathcal{B}[\![$`B`$]\!]$
$\mathcal{E}[\![\mathtt{E}_1$ `'+'` $\mathtt{E}_2]\!]$ r k = $\mathcal{E}[\![\mathtt{E}_1]\!]$ r;$\lambda b_1.\mathcal{E}[\![\mathtt{E}_2]\!]$ r;λb_2.k (b_1 + b_2)
$\mathcal{E}[\![\mathtt{E}_1$ `'='` $\mathtt{E}_2]\!]$ r k = $\mathcal{E}[\![\mathtt{E}_1]\!]$ r;$\lambda b_1.\mathcal{E}[\![\mathtt{E}_2]\!]$ r;λb_2.k (b_1 = b_2)
$\mathcal{E}[\![$`I`$]\!]$ r k s = (r `I`) = **unbound** →error,
  (s (r `I`)) = **unused** →error,
  k (s (r `I`)) s

Figura 3.3: Exemplo de definição semântica denotacional.

Nas equações apresentadas, pode-se ver que a maior parte das construções passam o *environment* recebido a seus constituintes sem qualquer alteração. A presença explícita do contexto nas equações torna sua leitura mais difícil, devido ao aumento da complexidade. Logo, é desejável que se abstraia o contexto nas equações semânticas sempre que este não influir diretamente na semântica da construção modelada, como é o caso da semântica de literais apresentada. No entanto, na forma em que estão apresentadas as equações não é possível abstrair o *environment*.

Esse tipo de problema pode ser resolvido pelo uso de combinadores (Curry & Feys, 1958 apud Hughes, 1982), os quais são especialmente úteis na manipulação de expressões lógicas e fórmulas do λ-cálculo. Por exemplo, o combinador **B** pode ser utilizado para alterar a ordem dos parâmetros de uma função:

$$\mathbf{B}\, f\, x\, y = f\, y\, x$$

Considere o uso do combinador **B** e da redução η para simplificar uma fórmula:

$$\begin{aligned} f\, x\, y &= g\, y\, x \\ f\, x\, y &= \mathbf{B}\, g\, x\, y \\ f &= \mathbf{B}\, g \end{aligned}$$

Pode-se utilizar uma estratégia semelhante para simplificar as equações semânticas apresentadas na Figura 3.3. Considere a produção A:

$$A \to r_0 B_1 r_1 B_2 \ldots B_n r_n$$

e sua denotação:

$$\begin{aligned} &\mathcal{S} : A \to \text{Contexto} \to \text{Ans} \\ &\mathcal{S}[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!]\, c = g(\mathcal{S}[\![B_1]\!], \mathcal{S}[\![B_2]\!], \ldots, \mathcal{S}[\![B_n]\!], c) \end{aligned}$$

Por meio de um combinador k

$$k\, (x_1, x_2, \ldots, x_n)\, c = g(x_1, x_2, \ldots, x_n, c)$$

podemos encapsular o fluxo das informações de contexto na denotação de A e, por meio de redução η, simplificar a equação semântica:

$$\begin{aligned} \mathcal{S}[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] c &= k(\mathcal{S}[\![B_1]\!], \mathcal{S}[\![B_2]\!], \ldots, \mathcal{S}[\![B_n]\!]) c \\ \mathcal{S}[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] &= k(\mathcal{S}[\![B_1]\!], \mathcal{S}[\![B_2]\!], \ldots, \mathcal{S}[\![B_n]\!]) \end{aligned}$$

O resultado é um mapeamento direto entre a construção sintática abstrata e sua semântica, como idealizado pela Figura 3.1. Além disso, o combinador k também encapsula a semântica da construção e se for genérico o suficiente poderá ser reutilizado em outras definições como um componente de semântica denotacional.

Para ilustrar a aplicação dessa estratégia, observe a equação semântica denotacional clássica da construção *if-then*. O *environment* é mencionado três vezes na equação

com único objetivo de passá-lo sem alteração aos constituintes da construção. Criando-se um combinador específico que encapsula a passagem de parâmetros para as funções $\mathcal{E}[\![\texttt{E}]\!]$ e $\mathcal{C}[\![\texttt{C}]\!]$, a equação pode ser reescrita como:

$$\begin{aligned}\mathcal{C}[\![\texttt{`if' E `then' C}]\!]\ \text{r c} &= \mathcal{E}[\![\texttt{E}]\!]\ \text{r;cond}(\mathcal{C}[\![\texttt{C}]\!]\ \text{r c,c})\\ = \mathcal{C}[\![\texttt{`if' E `then' C}]\!]\ \text{r c} &= \text{ifThen}(\mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![\texttt{C}]\!])\ \text{r c}\end{aligned}$$

Aplicando-se redução η, essa equação torna-se:

$$\mathcal{C}[\![\texttt{`if' E `then' C}]\!] = \text{ifThen}(\mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![\texttt{C}]\!])$$

Como pode ser observado, o combinador criado encapsula o fluxo das informações de contexto, simplificando a leitura do mapeamento semântico. Para isso, encapsula-se a semântica da construção, transformando as denotações de seus constituintes em parâmetros do combinador. Nesse sentido, o combinador *ifThen* se assemelha aos combinadores g da Figura 3.1. E, assim como esses recebem como parâmetro a semântica dos constituintes das construções, o combinador *ifThen* recebe os parâmetros $\mathcal{E}[\![\texttt{E}]\!]$ e $\mathcal{C}[\![\texttt{C}]\!]$. Dada sua função, o combinador se assemelha à equação semântica original:

$$\text{ifThen}\ (\mathsf{E}, \mathsf{C})\ \text{r c} = \mathsf{E}\ \text{r;cond}(\mathsf{C}\ \text{r c,c})$$

Esse processo de encapsulamento pode ser aplicado às equações semânticas de todas as construções, resultando em um mapeamento semântico mais legível que a definição inicial, como mostrado pela Figura 3.4. Observe que as equações semânticas se utilizam de combinadores que denotam comandos, expressões, declarações e programas. O fluxo de informações de contexto é encapsulado pelos combinadores, exceto na denotação de programas. Esse é o ponto em que o contexto inicial é definido. A partir dele o contexto flui de acordo com a semântica denotada pelos combinadores.

Os combinadores de denotações criados formam uma biblioteca de componentes de semântica denotacional, apresentada na Figura 3.5. Conceitos fundamentais de linguagens de programação podem ser encapsulados em componentes, e esses utilizados em diversas definições semânticas. Assim, o exercício de definição passa a ser a combinação de componentes previamente definidos de acordo com a semântica pretendida.

No entanto, dada a constante evolução das linguagens de programação, não é possível definir uma biblioteca de componentes definitiva. Os esforços são assim voltados para a criação de uma biblioteca abrangente, permitindo a definição semântica de um número significativos de linguagens. Sob essa perspectiva, quanto mais genéricos e abrangentes os componentes, maior a qualidade da biblioteca.

Mapeamento Semântico:

$\mathcal{P}$: **Pro** $\rightarrow$Pd
$\mathcal{D}$: **Dec** $\rightarrow$Dd
$\mathcal{C}$: **Com** $\rightarrow$Cd
$\mathcal{E}$: **Exp** $\rightarrow$Ed
$\mathcal{B}$: **Bas** $\rightarrow$B

$\mathcal{P}[\![\texttt{`let' D `in' C}]\!] = \text{run}(\mathcal{D}[\![\texttt{D}]\!], \mathcal{C}[\![\texttt{C}]\!])\ \text{r}_0\ \text{s}_0$
where $\text{r}_0 = \lambda\text{i}.\textbf{unbound}$
$\text{s}_0 = \lambda\text{l}.\textbf{unused}$

$\mathcal{D}[\![\texttt{I `=' E}]\!] = \text{bindValue}(\texttt{I}, \mathcal{E}[\![\texttt{E}]\!])$
$\mathcal{D}[\![\texttt{D}_1\ \texttt{`,'}\ \texttt{D}_2]\!] = \text{elabSeq}(\mathcal{D}[\![\texttt{D}_1]\!], \mathcal{D}[\![\texttt{D}_2]\!])$

$\mathcal{C}[\![\texttt{`if' E `then' C}]\!] = \text{ifThen}(\mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![\texttt{C}]\!])$
$\mathcal{C}[\![\texttt{`while' E `do' C}]\!] = \text{while}(\mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![\texttt{C}]\!])$
$\mathcal{C}[\![\texttt{I `:=' E}]\!] = \text{assign}(\texttt{I}, \mathcal{E}[\![\texttt{E}]\!])$

$\mathcal{E}[\![\texttt{B}]\!] = \mathcal{B}[\![\texttt{B}]\!]$
$\mathcal{E}[\![\texttt{E}_1\ \texttt{`+'}\ \texttt{E}_2]\!] = \text{sum}(\mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!])$
$\mathcal{E}[\![\texttt{E}_1\ \texttt{`='}\ \texttt{E}_2]\!] = \text{equal}(\mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!])$
$\mathcal{E}[\![\texttt{I}]\!] = \text{lookup}(\texttt{I})$

Figura 3.4: Exemplo de definição semântica baseada em combinadores de denotações.

Por isso, a criação de uma biblioteca de componentes a partir da análise das construções das linguagens mais importantes e do entendimento dos conceitos essenciais para a sua semântica. Com isso, os conceitos recorrentes das linguagens são analisados em conjunto, de forma a se estabelecer pontos em comum. Após essa análise, os conceitos fundamentais que permeiam diversas linguagens são encapsulados em componentes. Esses componentes são então parametrizados e recebem configurações por meio do contexto para permitir que características específicas de cada linguagem possam ser expressas. No fim do processo, os componentes são genéricos o suficiente para serem utilizados na definição de diferentes linguagens, mas abrangentes o suficiente para permitir a definição de semânticas diversas.

Um exemplo dessa flexibilidade é o componente `parametrize`. O conceito de parametrização, encapsulado pelo componente, é algo fundamental para as linguagens de programação. Entretanto, como cada linguagem implementa um determinado conjunto de modos de passagem de parâmetro, estes são determinados por meio dos parâmetros de `parametrize`. Assim, o componente pode ser configurado para expressar diferentes semânticas.

Domínios Semânticos:

Cc	= Store →Ans	- continuações de comando **c**
Dc	= Env →Cc	- continuações de declaração **u**
Ec	= Bv →Cc	- continuações de expressão **k**
Store	= Loc →[Bv + {unused}]	- *stores* **s**
Env	= Id →[Loc + {unbound}]	- *environments* **r**
Bv	= Num + Bool	- valores básicos **b**
Ans	= {error} + Store	- respostas finais **a**

Componentes:

Pd : Env →Store →Ans
Dd : Env →Dc →Cc
Cd : Env →Cc →Cc
Ed : Env →Ec →Cc

run : (Dd,Cd) →Pd
run (D, C) r s = D r;λr′.C r′(λs.s)
bindValue : (Id,Ed) →Dd
bindValue (I, E) r u = $\mathcal{E}[\![\mathtt{E}]\!]$ r;λb s.u ((new s)/I) s[b/(new s)]

elabSeq : (Dd,Dd) →Dd
elabSeq (D_1,D_2) r u = $\mathcal{D}[\![\mathtt{D}_1]\!]$ r;λr_1.$\mathcal{D}[\![\mathtt{D}_2]\!]$ r[r_1];λr_2.u r[r_1][r_2]

ifThen : (Ed,Cd) →Cd
ifThen (E,C) r c = $\mathcal{E}[\![\mathtt{E}]\!]$;cond($\mathcal{C}[\![\mathtt{C}]\!]$ r c,c)

while : (Ed,Cd) →Cd
while (E,C) r c = **fix** (λf r c.$\mathcal{E}[\![\mathtt{E}]\!]$;cond($\mathcal{C}[\![\mathtt{C}]\!]$ r;f r c,c)) r c

assign : (Id,Ed) →Cd
assign (I,E) r c = (r I) = **unbound** →error,
$\mathcal{E}[\![\mathtt{E}]\!]$ r;λb s.c s[b/(r I)]
sum : (Ed,Ed) →Ed
sum (E_1,E_2) r k = $\mathcal{E}[\![\mathtt{E}_1]\!]$ r;λb_1.$\mathcal{E}[\![\mathtt{E}_2]\!]$ r;λb_2.k (b_1 + b_2)

equal : (Ed,Ed) →Ed
equal (E_1,E_2) r k = $\mathcal{E}[\![\mathtt{E}_1]\!]$ r;λb_1.$\mathcal{E}[\![\mathtt{E}_2]\!]$ r;λb_2.k (b_1 = b_2)

lookup : (Id) →Ed
lookup (I) r k s =
(r I) = **unbound** →error,
(s (r I)) = **unused** →error,
k (s (r I)) s

Figura 3.5: Pequena biblioteca de componentes de semântica denotacional.

Neste trabalho, uma biblioteca de componentes foi desenvolvida para a definição da semântica de linguagens de programação imperativas. O paradigma imperativo se caracteriza por modelar a computação por meio da alteração do estado da máquina, geralmente fazendo uso de variáveis. A principal razão para a limitação do escopo é que ao se restringir a solução a um único paradigma facilita-se construir uma biblioteca de componentes mais coesa. Como a biblioteca deve ser capaz de expressar linguagens inteiras, abordar outros paradigmas implicaria a inclusão de conceitos específicos a um ou outro paradigma. Por exemplo, a inclusão do paradigma funcional exigiria a criação de componentes para casamento de padrões e uma forma diferente de tratar a memória. Por fim, o paradigma imperativo foi escolhido por sua notável disseminação na indústria.

3.2 Projeto do Sistema

A biblioteca desenvolvida neste trabalho é organizada em três partes: domínios semânticos, componentes de semântica denotacional e funções auxiliares.

Os domínios semânticos definem as denotações que representam o modelo computacional característico do paradigma imperativo. Por exemplo, há um domínio que representa parte do estado da computação por meio de um mecanismo de armazenamento e outros que representam valores. Do ponto de vista do usuário, além dos tipos dos componentes, os domínios mais importantes são: `Bv`, `Sv` e `Dv`. O primeiro é o domínio dos valores básicos, composto por valores numéricos, valores booleanos, caracteres e *strings*. O domínio dos valores que podem ser armazenados no *store*, `Sv`, é composto pelos valores básicos, arranjos, registros, objetos e arquivos – usados para modelar entrada e saída. Por fim, o domínio `Dv` determina os valores que podem ser ligados no *environment*. Esses são formados por todos os valores de `Sv`, procedimentos, funções, configurações da biblioteca, classes, informações de tipo, comandos, expressões e declarações. Esses domínios devem ser conhecidos para que o usuário saiba quais valores podem ser manipulados e como podem ser usados.

Os componentes modelam conceitos fundamentais e recorrentes das linguagens de programação imperativas. Esses conceitos fazem parte da semântica das construções dessas linguagens. Por exemplo, a essência da construção *if-then* da linguagem Tiny é executar um comando caso uma determinada condição seja respeitada. Esse conceito é modelado pelo componente `ifThen`. O componente apenas encapsula a forma como o contexto é manipulado e o processo de decisão, mas, uma vez configurado com a semântica do comando e da condição, atua como uma denotação de comando. Analisando as

equações, vemos que o tipo das equações semânticas de comandos é

$$\textbf{Com} \rightarrow \text{Env} \rightarrow \text{Cc} \rightarrow \text{Cc}$$

enquanto o tipo de `ifThen` é

$$\begin{aligned} &[\text{Ed} \times \text{Cd}] \rightarrow \text{Cd} \\ &\quad \text{onde Ed} = \text{Env} \rightarrow \text{Ec} \rightarrow \text{Cc} \\ &\qquad \text{Cd} = \text{Env} \rightarrow \text{Cc} \rightarrow \text{Cc} \end{aligned}$$

Por isso o componente `ifThen` faz parte dos componentes que representam comandos.

Há também grupos de componentes que representam expressões e declarações. No entanto, em alguns casos os conceitos são genéricos o suficiente para serem classificados em qualquer uma das categorias sintáticas. Expandindo `ifThen`, tem-se o componente `choose`, o qual modela uma escolha entre dois comandos, expressões ou declarações. Quando configurado com dois comandos, o componente resulta em uma execução de comando, enquanto a configuração com duas expressões resulta na avaliação de uma expressão. Refletindo essa flexibilidade, sua equação de tipo é:

$$\begin{aligned} &(\text{Ed, x, x}) \rightarrow \text{x, x} \in \text{Xd} \\ &\quad \text{onde Xd} = \text{Cd} + \text{Dd} + \text{Ex} \\ &\qquad \text{Ex} = \text{Ed} + \text{Ev} \end{aligned}$$

O domínio Xd é uma união disjunta das denotações de todas as classes sintáticas. O tipo Ev é incluído para permitir definição explícita da ordem de avaliação de expressões, discutida na Seção 4.3.7.

As funções auxiliares são utilizadas para simplificar as definições dos componentes, abstraindo algumas operações comuns, como a procura de ligações no *environment* e o armazenamento de valores no *store*. Essas funções não precisam ser conhecidas pelo usuário da biblioteca, mas facilitam a criação de novos componentes e facilitam a leitura das definições dos componentes.

O Capítulo 4 apresenta a biblioteca de componentes desenvolvida neste trabalho. Uma implementação dos componentes utilizando λ-cálculo é apresentada no Apêndice C e uma implementação na linguagem Haskell pode ser encontrada no Apêndice E e no endereço eletrônico `http://www2.dcc.ufmg.br/laboratorios/llp/wiki/doku.php?id=guisousa_cbds`.

3.3 Uso da Biblioteca

A biblioteca de componentes pode ser utilizada para modelar a semântica dinâmica de linguagens de programação. Utilizando-se a implementação em Haskell, é possível construir um interpretador. Esta seção apresenta os passos necessários para a construção de um interpretador, de forma a ilustrar o uso da biblioteca. Para isso é apresentado um exemplo de interpretador construído na linguagem Haskell na Listagem 3.1. Exemplos de definições semânticas utilizando apenas a implementação em λ-cálculo são apresentados nos Apêndices B e A.

```
1  module Main where
2
3  import Prelude hiding (exp)
4  import Domains
5  import Components
6  import Auxiliar
7
8  -- Abstract Syntax Tree
9
10 data Prog = PROG Exp
11 data Exp = SUM Exp Exp
12           | SUB Exp Exp
13           | MUL Exp Exp
14           | DIV Exp Exp
15           | NUM String
16
17 -- Semantic Equations
18
19 prog :: Prog -> File -> Ans
20
21 prog (PROG e) input = run(exp e) input r0 s0
22
23 exp :: Exp -> Ed
24
25 exp (NUM n) = int(n)
26 exp (SUM e1 e2) = apply("+", exp e1, exp e2)
27 exp (SUB e1 e2) = apply("-", exp e1, exp e2)
28 exp (MUL e1 e2) = apply("*", exp e1, exp e2)
29 exp (DIV e1 e2) = apply("/", exp e1, exp e2)
30
31 -- Main Function
32
33 main = result (prog (parse) input)
34   where result p = case p of
35                      AnsHalt (Stop,e,r,s) -> print e
36                      AnsHalt (RError Error,e,r,s) -> print Error
37         input = File []
38
39 parse = example
40   where example = PROG (SUM (NUM "1") (NUM "7"))
```

Listing 3.1: Exemplo de interpretador de uma linguagem simples de expressões.

O primeiro passo na construção de um interpretador é transformar o programa em uma árvore de sintaxe abstrata. A definição da árvore é ilustrada nas linhas 10 a 15 do exemplo. Essa árvore é então utilizada para representar o resultado da análise sintática do programa. No exemplo, as linhas 39 e 40 mostram a construção de uma árvore de sintaxe abstrata simples. No entanto, num interpretador real esse passo é complexo, sendo melhor executado por um *parser* acoplado ao sistema.

Em seguida, é necessário apresentar as equações semânticas, responsáveis por definir o mapeamento entre as construções da sintaxe abstrata e os componentes. As equações podem ser organizadas segundo a árvore de sintaxe abstrata, como ilustrado nas linhas 19 a 29. Essas equações são responsáveis por definir a semântica das construções. A equação mais importante é aquela que descreve a semântica de um programa. Essa equação utiliza o componente `run`, o qual é o ponto de entrada da biblioteca, fornecendo o contexto inicial e direcionando a interpretação.

Finalmente, a função principal do interpretador, apresentada nas linhas 33 a 37, dá início à interpretação executando a análise sintática e avaliando a equação semântica principal. Dada a simplicidade da linguagem apresentada, não é necessário fazer configurações iniciais. Entretanto, linguagens mais complexas podem exigir a definição de entidades auxiliares. Essas podem ser transmitidas à definição por meio do *environment* inicial. Por exemplo, pode-se definir funções que simplificam a escrita de valores na saída do programa. Outra possibilidade é a definição do valor inicial das posições do *store*.

Para facilitar o entendimento, o exemplo apresentado é simples, não utilizando entrada e saída ou configurações iniciais. No entanto, é possível ver o uso de todos os módulos do sistema. Os componentes `apply`, `int` e `run` pertencem ao módulo *Components*. Os domínios *Ed*, *File* e *Ans* pertencem ao módulo *Domains*. As definições iniciais de *store* e *environment*, s_0 e r_0, pertencem ao módulo *Auxiliar*. Além disso, o exemplo apresentado é um programa Haskell válido, podendo servir de ponto de partida para a criação de outras definições por parte do usuário da biblioteca. Durante o desenvolvimento desse exemplo, foi utilizado o compilador GHC[1], versão 7.0.4, implementando a linguagem Haskell 98 e algumas extensões utilizadas pela biblioteca.

3.4 Conclusões

Este capítulo apresentou o método desenvolvido neste trabalho, cujo objetivo é melhorar a escalabilidade da semântica denotacional. Essa técnica, melhora a modula-

[1] http://www.haskell.org/ghc/

ridade das definições semânticas por meio da remoção da dependência aparente das informações de contexto na redação das equações semânticas. A solução desenvolvida consiste em uma abordagem de estruturação de semântica denotacional baseada em componentes, encapsulando os conceitos fundamentais das linguagens de programação e permitindo a configuração dos aspectos variáveis. Com o objetivo de equilibrar a coesão e a expressividade dos componentes, o escopo de aplicação do método se restringe ao paradigma imperativo.

O próximo capítulo apresenta a biblioteca de componentes de semântica denotacional que implementa o método desenvolvido. São apresentados exemplos de uso e a semântica de cada componente é detalhada de acordo com o modelo semântico exposto neste capítulo. Ao longo da apresentação, o modelo semântico é detalhado de modo a facilitar o entendimento dos componentes.

Capítulo 4

Biblioteca de Componentes para Linguagens Imperativas

Este capítulo apresenta a biblioteca de componentes desenvolvida como proposto no capítulo anterior. O capítulo é organizado em função dos aspectos importantes para as linguagens de programação imperativas, como comandos, atribuições, declarações e procedimentos. Os vários componentes são apresentados em cada seção de acordo com sua função. Uma apresentação centrada nos componentes pode ser encontrada no Apêndice C, servindo de guia referência para sua utilização. Para facilitar a leitura dos exemplos deste capítulo, o Apêndice D apresenta um glossário dos domínio semânticos e componentes.

Idealmente, a biblioteca de componentes deveria ser suficiente para a definição de todas as linguagens do paradigma imperativo. Neste trabalho, o objetivo é capturar a essência do paradigma imperativo por meio de componentes que encapsulem conceitos genéricos e recorrentes das linguagens que o representam. No entanto, é possível que definições semânticas exijam a criação de novos componentes ou a alteração de componentes já existentes. Por exemplo, um número limitado de modos de passagem de parâmetros é previsto pela biblioteca, e a definição de outros modos de passagem requer a adição do tratamento adequado às funções auxiliares relacionadas. Entretanto, os conceitos de parametrização e chamada de procedimentos tendem a permanecer inalterados.

4.1 Modelo Semântico

A biblioteca de semântica denotacional proposta no capítulo anterior inclui diversos domínios semânticos, os quais dão suporte aos componentes. Esta seção apresenta os

domínios mais importantes e a sua relação com os componentes apresentados neste capítulo.

Uma parte importante da biblioteca é a modelagem das informações de contexto. Essas são formadas por continuação, *store* e *environment*. Cada componente recebe o contexto proveniente da equação semântica que o envolve. Da mesma forma, cada componente deve passar informações de contexto aos componentes que combina e à continuação. Assim as informações de contexto fluem por todo o sistema.

A continuação é uma forma de se representar o fluxo de controle. A continuação contém o resto do programa, ou seja, o código que utiliza o contexto modificado por uma construção. Esse conceito está presente nas equações a seguir:

$$K_1\, c = K_2\, (K_3\, c)$$
$$K_1\, c = K_2; K_3; c$$

As duas equações são equivalentes. O caractere ';' indica que o restante da fórmula compõe um único parâmetro. Nessas equações, os componentes K_2 e K_3 são executados sequencialmente e são seguidos pela continuação de K_1. À primeira vista, esse mecanismo pode parecer desnecessário, mas se mostra útil para modelar erros, como no exemplo a seguir:

$$K\, a\, c \;=\; (a < 0) \;\rightarrow error,\; K'\, a\, c$$

Nesse exemplo, a computação representada por K' só é realizada caso o valor de a seja nulo. Caso contrário, o fluxo de controle é interrompido e o valor *error* é retornado. Na biblioteca, erros interrompem a execução do programa, retornando o valor *error* e o contexto válido no momento do erro.

O *store*, representado pelo domínio *Store* e pela letra s, é uma entidade abstrata que representa um mecanismo de armazenamento. Sua função é mapear endereços chamados de *locations* a valores. Dessa forma, detalhes pertinentes à memória principal, discos e outros dispositivos de armazenamento são abstraídos das definições semânticas. O *store* representa o estado da máquina, e, por consequência, a cada momento durante a execução existe apenas um *store* válido. Considere a equação a seguir:

$$K_1\, s\, c \;=\; K_2\, s\, ;\, \lambda s'.K_3\, s'; \lambda s''.c\, s''$$

O componente K_1 passa o *store* recebido, s, a K_2. K_3 recebe o *store* atualizado, s', e o *store* produzido por este, s'', é então passado à continuação. Outros detalhes são ignorados. Nessa equação é possível perceber que cada componente utiliza o *store* corrente e passa suas alterações adiante.

O *environment*, representado pelo domínio *Env* e pela letra *r*, tem o papel de mapear nomes a valores, como procedimentos, valores básicos e *locations*. Cada um dos mapeamentos do *environment* é chamado de ligação. Esse mecanismo permite a criação de variáveis e a abstração de procedimentos, entre outros. No entanto, ao contrário do que ocorre com o *store*, as ligações do *environment* são passadas por cada componente a seus constituintes, mas não necessariamente transmitidas à continuação. Assim, alguns componentes delimitam seções do sistema em que determinadas ligações são válidas. Esse mecanismo de delimitação do *environment* permite a criação de módulos. Considere a equação a seguir:

$$M\,r\,s\,c\;=\;K\,r\,s;\lambda r's'.c\,r\,s'$$

O componente M passa a K o contexto recebido, mas não passa à continuação o *environment* r', produzido por K. Dessa forma, o componente M implementa um módulo completamente opaco.

Outra parte importante da biblioteca é a gama de valores manipulados pelos programas. Os valores básicos, representados pelo domínio *Bv*, tem o objetivo de representar valores primitivos. Os valores básicos são booleanos, *strings*, caracteres e diversos tipos de números. Esse domínio foi inspirado na linguagem Java e abrange grande parte dos tipos encontrados em linguagens de programação imperativas. Outros tipos devem ser adicionados à biblioteca caso necessário ou representados por classes. Essa estratégia é utilizada para manter a simplicidade da biblioteca.

Os programas também podem manipular comandos, expressões, declarações, arranjos, registros, objetos, classes e procedimentos. Em conjunto com os valores básicos, esses tipos formam o domínio dos valores expressáveis, *Ev*. É possível substituir expressões por valores expressáveis nos parâmetros dos componentes, o que é explorado na Seção 4.3.7 para definição explícita da ordem de avaliação de expressões. Os valores de cada um dos tipos pertencentes aos valores expressáveis podem ser ligados no *environment*, mas procedimentos, comandos, expressões, declarações e classes não podem ser armazenados no *store*.

Os componentes da biblioteca são classificados em representações de programas, comandos, expressões e declarações e coletores de rótulos. Os primeiros, representados pelo domínio *Pd*, modelam programas inteiros, lidando com entrada e saída e relatando o resultado da execução. Enquanto a principal função das denotações de comandos, *Cd*, é alterar o estado da máquina, denotações de expressões, *Ed*, são principalmente responsáveis por produzir valores e denotações de declarações, *Dd*, por produzir ligações. No entanto, essa separação não é total, havendo componentes que encorporam

característica de múltiplos grupos. Esses pertencem ao domínio *Xd*. Por fim, coletores de rótulos pertencem ao domínio *Jd* e são responsáveis por produzir ligações que indicam pontos nos programas que podem ser alvos de saltos.

De forma geral, os exemplos apresentados neste capítulo apresentam componentes que manipulam valores expressáveis e alteram o contexto. Embora os detalhes sejam abordados apenas no Apêndice C, o leitor deve se atentar para a composição do contexto e para o comportamento básico de cada classe de componentes. Um glossário completo com definições informais de cada um dos componentes e domínios abordando seus tipos e funções é apresentada no Apêndice D.

4.2 O Programa

O componente `run` é responsável por modelar a execução do programa. Seus parâmetros são um comando, a entrada e valores iniciais para *environment* e *store*. Definições iniciais desses dois últimos são fornecidas pelas funções auxiliares $\mathtt{r}_0$ e $\mathtt{s}_0$, as quais indicam que não há ligação definida ou valor armazenado. O componente `run` retorna uma tupla composta por um código de saída (`stop` ou `error`), um valor de retorno e *environment* e *store* válidos no momento do término da execução. Assim, o usuário pode inspecionar o *store* para apresentar a saída ou implementar outros comportamentos mais complexos.

A Figura 4.1 mostra um exemplo de definição da execução de um programa Java. Nesse exemplo, a construção abstrata `'java' Id 'in' D` representa um programa cuja classe principal é `Id`. A declaração `D` contém a descrição das classes utilizadas. $\mathtt{r}_0$ e $\mathtt{s}_0$ representam respectivamente *environment* e *store* iniciais, ambos vazios. A semântica de execução envolve a chamada do método *main* da classe `Id` utilizando um *environment* composto pelas ligações produzidas pela declaração `D`. Quando do término da execução, o resultado passado ao sistema é composto pelo valor `e` e pela saída do programa. Nesse exemplo, o código `t` e o *environment* são ignorados.

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{'java' Id 'in' D}]\!]\ \text{i args} = \text{result}(\text{run c i } \mathtt{r}_0\ \mathtt{s}_0) \\
\quad \text{where} \qquad \text{c} = \text{addEnv}(\text{accum}(\mathtt{D}), \text{call}(\text{lookup}(\text{Id}, \text{`main'}), \text{args})) \\
\qquad\quad \text{result} = \lambda(\text{t, e, r, s}) \rightarrow (\text{e, s output})
\end{array}
$$

Figura 4.1: Exemplo de definição da execução de um programa Java.

4.2.1 Entrada e Saída

Entrada e saída são modeladas por posições especiais no *store*. O componente `run` é responsável por colocar a entrada na memória onde pode ser lida pelo componente `input`, bem como criar um arquivo vazio onde o componente `output` escreve a saída. Os valores da entrada são sempre retirados do início do arquivo, enquanto na saída valores são adicionados no final.

$$\mathcal{C}[\![\texttt{`print(' E ')'}]\!] = \text{output}(\mathcal{E}[\![\texttt{E}]\!])$$
$$\mathcal{E}[\![\texttt{`raw\_input()'}]\!] = \text{input}$$

Figura 4.2: Exemplo de definição de mecanismos de entrada e saída na linguagem Python.

A Figura 4.2 mostra um exemplo de definição de mecanismos de entrada e saída na linguagem Python. O mecanismo *print* armazena o resultado da avaliação da expressão `E` na saída. O mecanismo *raw_ input* retira uma linha da entrada, passando-a como *string* à continuação. Entretanto, o componente `input` trata a entrada como uma lista, retirando um elemento por vez. Consequentemente, a entrada de programas da linguagem Python deve ser organizada de modo que cada elemento corresponda a uma linha. Para simplificar o exemplo, não é apresentada a semântica de *raw_ input* com parâmetros.

4.3 Manuseio de Valores

Em um programa, os dados são representados por entidades chamadas valores. Estes podem ser retornados como resultado de funções, passados como argumentos, armazenados e serem alvo de operações como somas, subtrações e testes lógicos.

Geralmente, as linguagens de programação organizam os valores em tipos. Por exemplo, em Java, `true` é classificado como um valor booleano, enquanto "casa" tem tipo *String*. A cada valor está associado pelo menos um tipo, o qual define as operações aplicáveis a esse valor, bem como a semântica dessas operações. O sistema de tipos resolve diversas questões abordadas por cada linguagem de programação. Por exemplo, a linguagem deve definir quando dois tipos são considerados equivalentes, qual a semântica das operações quando seus operandos possuem tipos diferentes e como deve ser definido o tipo de um valor. Com o objetivo de manter a simplicidade, a biblioteca aqui abordada deixa essas definições a cargo de outras ferramentas desenvolvidas especificamente para a definição de sistemas de tipos. Ainda assim, algumas operações básicas relacionadas a tipos são suportadas e são abordadas na Seção 4.3.5.

A característica principal das expressões é que, quando avaliadas, produzem um valor. No entanto, algumas expressões fazem mais do que apenas produzir valores. Essas são chamadas de expressões com efeito colateral em alusão às alterações que causam no estado da máquina. As seções a seguir detalham os tipos de expressão previstos na biblioteca e as formas como podem ser modelados por meio de componentes.

4.3.1 Literais

O tipo mais simples de expressão geralmente presente nas linguagens de programação é o literal, o qual denota um valor simples de algum tipo previsto na linguagem. O literal é a representação primitiva dos valores. Ou seja, se um programador precisa utilizar um valor específico em um trecho do programa, ele provavelmente utilizará um literal.

A biblioteca possui componentes para a criação de valores a partir de literais. Cada um desses componentes recebe um literal representado por uma *string* e retorna o valor correspondente. Os tipos disponíveis são *string*, *char*, *bool*, números inteiros e números de ponto flutuante. Estes dois últimos estão disponíveis com uma gama representativa de opções de precisão. A Figura 4.3 mostra um exemplo de uso dos componentes que modelam a semântica dos literais.

$\mathcal{E}[\![$`'byte' b`$]\!]$ = byte(b)
$\mathcal{E}[\![$`'short' s`$]\!]$ = short(s)
$\mathcal{E}[\![$`'int' i`$]\!]$ = int(i)
$\mathcal{E}[\![$`'long' l`$]\!]$ = long(l)
$\mathcal{E}[\![$`'float' f`$]\!]$ = float(f)
$\mathcal{E}[\![$`'double' d`$]\!]$ = double(d)
$\mathcal{E}[\![$`'char' c`$]\!]$ = char(c)
$\mathcal{E}[\![$`'string' s`$]\!]$ = string(s)
$\mathcal{E}[\![$`'bool' b`$]\!]$ = bool(b)

Figura 4.3: Exemplo de semântica de literais para a linguagem Java.

4.3.2 Acesso a Variáveis e Constantes

Uma variável é uma unidade de armazenamento de valores. Como tal, ela é usada em duas operações básicas: acesso ao seu valor e atribuição de um novo valor. Como mostrado na Seção 4.4.7, variáveis são criadas por meio do componente `ref`, enquanto variáveis constantes são criadas por meio do componente `const`. A diferença entre

variáveis e variáveis constantes é que as últimas não podem ter seu valor alterado. Há ainda constantes cujo valor não é armazenado no *store*. Estas são geralmente tratadas durante a análise sintática do programa, mas também podem ser criadas por meio do componente `bind` e acessadas como se fossem variáveis. Esta seção trata do acesso ao valor de uma variável, o qual é feito passando-se o nome da variável ao componente `lookup`.

No entanto, o componente `lookup` não acessa diretamente o valor guardado no *Store*, mas sim o mapeamento entre o nome da variável e o seu identificador no *store*, o *location*. Isso é feito para que, quando necessário, referências à variável possam ser passadas a outros componentes, como, por exemplo, ocorre numa chamada de procedimento passando a variável como parâmetro por referência. Dessa forma, o uso da referência ou acesso ao valor é feito pelo componente que recebe o resultado de `lookup`. Por exemplo, na primeira equação da Figura 4.4, a expressão do lado esquerdo da atribuição é avaliada até se encontrar o *location*, enquanto a expressão do lado direito é avaliada até se encontrar o valor armazenado. No entanto, caso seja necessário acesso ao valor da variável diretamente, o componente `deref`, introduzido na Seção 4.4.7 pode ser utilizado.

$\mathcal{E}[\![\texttt{E}_1\ \texttt{':='}\ \texttt{E}_2]\!] = \text{assign}(\mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!])$
$\mathcal{E}[\![\texttt{I}]\!] = \text{lookup}(\texttt{I})$

Figura 4.4: Exemplo de semântica de acesso a variáveis.

4.3.3 Valores Compostos

Além dos valores simples definidos anteriormente, programas lidam com valores compostos. Por exemplo, uma lista de valores é um valor composto. A presença de valores compostos em uma linguagem facilita lidar com grandes quantidades de dados. Valores compostos podem ser classificados em duas categorias: valores compostos homogêneos e valores compostos heterogêneos. Todos os valores simples que compõem um valor composto homogêneo possuem o mesmo tipo. O exemplo mais simples de valor composto homogêneo é o arranjo.

Arranjos são sequências indexadas. Cada elemento do arranjo pode ser simples ou composto, mas todos os elementos devem pertencer à mesma hierarquia de tipos. Por exemplo, podemos ter arranjos de inteiros e arranjos cujos elementos são arranjos de

números de ponto flutuante, mas não podemos ter arranjos em que alguns elementos são inteiros e outros números de ponto flutuante.

Cada elemento do arranjo é identificado por meio de um índice. Os índices são formados por um intervalo de números consecutivos. O menor índice do arranjo é chamada de limite inferior e o maior é chamado de limite superior. Embora conceitualmente os índices de um arranjo possam pertencer a qualquer tipo que possua ordem, os componentes aqui definidos utilizam números inteiros. Essa é uma simplificação do modelo e pode ser facilmente mapeada para outras situações em que outros tipos são usados, bastando definir um mapeamento entre os elementos que se queira usar como índices e os números utilizados nos componentes. O intervalo ao qual os índices pertencem pode ser definido no momento da declaração do arranjo.

Há duas operações básicas sobre arranjos: a construção do arranjo e o acesso a um elemento. Para construir um arranjo, basta conhecer os índices e o valor de cada elemento. O componente `array` recebe os dois limites do arranjo e um conjunto de valores e produz um arranjo. Cada valor é armazenado no `store`, onde podem ser acessados posteriormente. A Figura 4.5 ilustra a criação de arranjos. Nesse exemplo, uma tupla é representada por um arranjo com dois elementos. Para isso, os limites do arranjo são as posições 0 e 1, e os valores armazenados correspondem aos valores dos dois elementos da tupla.

$$\mathcal{E}[\![\text{`tuple' `[' } E_1\text{, } E_2\text{`]'}]\!] = \text{array}(\text{int}(\text{`0'}), \text{int}(\text{`1'}), [E_1 \bullet E_2])$$

Figura 4.5: Exemplo de semântica de criação de um arranjo.

O acesso a um elemento do arranjo é feito pelo componente `index`. Por meio do nome associado ao arranjo e o índice do elemento desejado, assim como ocorre com variáveis simples, o componente `index` retorna o *location* associado ao elemento no *store*. O exemplo da Figura 4.6 mostra o acesso a um elemento de arranjo.

$$\begin{aligned}\mathcal{E}[\![\text{I `[' } E_1\text{`]' `[' } E_2\text{ `]'}]\!] = \text{index}(&\\ &\text{index}(\text{lookup}(\text{I}), \mathcal{E}[\![E_1]\!]),\\ &\mathcal{E}[\![E_2]\!])\end{aligned}$$

Figura 4.6: Exemplo de semântica de acesso a um elemento de arranjo.

Primeiro o identificador `I` é utilizado para recuperar o arranjo. Em seguida, a primeira expressão é utilizada para definir o índice na primeira dimensão do arranjo. Por fim, a segunda expressão define o índice na segunda dimensão do arranjo, e o *location* associado à posição acessada é retornado. De maneira similar, acessos a arranjos

com mais dimensões podem ser feitos. A cada nova dimensão n, basta passar ao componente `index` o arranjo encontrado no acesso à dimensão $n-1$ e o índice da dimensão n. A razão para isso é que um arranjo de n dimensões é, na verdade, um arranjo de $n-1$ dimensões cujos elementos são arranjos unidimensionais.

Registros são mapeamentos entre nomes e valores. Por exemplo, uma lista telefônica pode ser vista como um conjunto de registros. A cada entrada na lista estão associados campos como telefone, endereço e nome do proprietário. Um registro possui campos, cada qual com seu nome e valor. Registros são valores compostos heterogêneos.

Assim como o arranjo, o registro possui operações de construção e acesso, ilustradas pela Figura 4.7. O componente `record` pode ser utilizado para criar um registro a partir de uma declaração primitiva ou de uma declaração composta. O acesso aos campos de um registro é feito por meio do componente `lookup`, utilizando-se o valor do registro, ou seu nome, e o nome do campo que se deseja acessar, de forma similar ao que é mostrado no exemplo de acesso em arranjos.

$$\mathcal{E}[\![\texttt{`record'\ D}]\!] = \text{record}(\mathcal{D}[\![\texttt{D}]\!])$$
$$\mathcal{E}[\![\texttt{I}_1\ \texttt{`.'}\ \ \texttt{I}_2]\!] = \text{lookup}(\text{I}_1, \text{I}_2)$$

Figura 4.7: Exemplo de semântica de registro.

4.3.4 Operadores

A aplicação de operadores pode ser modelada pelo componente `apply`. Este recebe como parâmetros um operador e uma lista de expressões. O operador recebe uma lista de valores como parâmetro e seu resultado pode ser um valor ou erro. A semântica de `apply` é aplicar o operador aos valores resultantes da avaliação das expressões. O exemplo da Figura 4.8 demonstra como o componente `apply` pode ser utilizado para modelar a aplicação de um operador.

$$\mathcal{E}[\![\texttt{E}_1\ \texttt{**}\ \texttt{E}_2]\!] = \text{apply}(\text{exp}, [\mathcal{E}[\![\texttt{E}_1]\!] \bullet \mathcal{E}[\![\texttt{E}_2]\!]])$$
$$\text{where} \quad \text{exp} = \lambda \text{e*.e*} = (\text{a,b}) \rightarrow (\text{b} >= 0 \rightarrow \text{exp}_{imp}\ \text{a b 1 0, error})$$
$$\text{exp}_{imp} = \lambda \text{a b c n.n} < \text{b} \rightarrow \text{exp}_{imp}\ \text{a b (a} \times \text{c) (n + 1), c}$$

Figura 4.8: Exemplo de semântica de um operador de exponenciação.

Há ainda alguns operadores pré-definidos na biblioteca, são eles soma, multiplicação, subtração, divisão, teste de igualdade, teste de desigualdade, comparação "menor

que"e comparação "maior que". Para utilizar um desses operadores, basta passar ao componente `apply` a representação do operador pretendido, ou seja, um dos membros do conjunto {'+','-','/','*','=','!=','<','>'}, e dois operandos. A Figura 4.9 mostra um exemplo de semântica de operações aritméticas.

$$\mathcal{E}[\![\texttt{E}_1\ \texttt{`+'}\ \texttt{E}_2]\!] = \text{apply}(\text{`+'}, \mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!])$$
$$\mathcal{E}[\![\texttt{E}_1\ \texttt{`-'}\ \texttt{E}_2]\!] = \text{apply}(\text{`-'}, \mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!])$$
$$\mathcal{E}[\![\texttt{E}_1\ \texttt{`*'}\ \texttt{E}_2]\!] = \text{apply}(\text{`*'}, \mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!])$$
$$\mathcal{E}[\![\texttt{E}_1\ \texttt{`/'}\ \texttt{E}_2]\!] = \text{apply}(\text{`/'}, \mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!])$$

Figura 4.9: Exemplo de semântica de operadores aritméticos pré-definidos.

4.3.5 Tipos

Em alguns casos, é necessário incluir a verificação dos tipos das variáveis na definição da semântica dinâmica de uma linguagem. Por exemplo, pode ser necessário verificar o tipo dinâmico de um objeto, o qual pode ser diferente do seu tipo estático. Da mesma forma, não é possível determinar o tipo de variáveis em linguagens dinamicamente tipadas por meio de semântica estática. No entanto, a modelagem de sistemas de tipo é um trabalho complexo, que foge ao escopo deste trabalho. Por isso, a biblioteca de componentes oferece apenas testes de tipo básicos.

A inspeção do tipo de variáveis e valores é realizada por meio do componente `getType`. Quando o tipo da variável não é declarado, `getType` retorna 'unknown'. O mesmo ocorre caso seja passado a `getType` um valor que não é primitivo, classe ou objeto. A Figura 4.10 mostra exemplos de checagem de tipo. Para melhorar a legibilidade das definições, utiliza-se o componente `type` para construir os tipos com os quais se deseja comparar o resultado de `getType`.

Com o componente `getType`, é possível realizar testes de tipo de variáveis e valores. Com isso, testes simples em atribuições e operações lógicas e aritméticas podem ser implementados. Além disso, é possível verificar o tipo dinâmico de objetos. Em uma atribuição, é possível verificar a compatibilidade de um objeto com uma variável inspecionando os tipos da cadeia de objetos *super*. Além disso, é possível recuperar a definição da classe correspondente ao tipo estático do objeto, o que permite verificar se os atributos acessados são compatíveis com esse tipo. No entanto, a biblioteca implementa apenas operações simples de tipos. Sistemas de tipo mais complexos devem ser implementados a parte, na semântica estática. Ou, caso seja necessário fazer testes mais complexos em tempo de execução, por meio de novos componentes.

$$\mathcal{E}[\![\texttt{E}_1\ \texttt{`[' E}_2\ \texttt{`]'}]\!] = \text{choose}(\text{apply}(\text{`='}, \text{getType}(\mathcal{E}[\![\texttt{E}_2]\!]), \text{type}(\text{`int'})), \text{index}(\mathcal{E}[\![\texttt{E}_1]\!],\mathcal{E}[\![\texttt{E}_2]\!]), \text{abort})$$

$$\mathcal{E}[\![\texttt{I}_1\ \texttt{=}\ \texttt{I}_2]\!] = \text{choose}(\text{apply}(\text{`='},\text{getType}(\texttt{I}_1), \text{getType}(\texttt{I}_2)), \text{assign}(\mathcal{E}[\![\texttt{I}_1]\!],\mathcal{E}[\![\texttt{I}_2]\!]), \text{abort})$$

$$\mathcal{E}[\![\texttt{`outputStr' E}]\!] = \text{choose}(\text{apply}(\text{`='},\text{getType}(\mathcal{E}[\![\texttt{E}]\!]), \text{type}(\text{`string'})), \text{output}(\mathcal{E}[\![\texttt{E}]\!]), \text{abort})$$

Figura 4.10: Exemplos de teste de tipo. Caso o teste falhe, a execução é abortada pelo sequenciador `abort`.

Para garantir o correto funcionamento dos componentes, em sua implementação, os valores de tipo primitivo podem ter seu tipo inspecionado por operações como `Bool?` ou `Int?`. Há ainda a operação `Num?`, a qual verifica se o valor possui tipo numérico. Caso o tipo do valor seja diferente do que se espera, o sistema entra em estado de erro e a execução do programa é interrompida. Há ainda operações como `isBool`, `isInt` e `isNum`, as quais retornam valor verdadeiro caso o valor tenha o tipo esperado ou falso caso contrário. Entretanto, essas operações não devem ser utilizadas diretamente nas definições, pois não são componentes. Um exemplo de uso dessas operações é a verificação de que a avaliação da condição do componente `choose` resulta em um valor booleano.

4.3.6 Expressões Condicionais

Expressões condicionais possuem duas sub-expressões e uma condição. De acordo com a condição, uma das sub-expressões é escolhida e o resultado de sua avaliação é o resultado da expressão condicional. De forma semelhante, computações condicionais que não retornam um resultado podem ser modeladas por comandos condicionais, os quais são abordados na Seção 4.4.4.

Expressões condicionais podem ser modeladas pelo componente `choose`, como demonstrado no exemplo da Figura 4.11. Primeiro, a expressão $\texttt{E}_1$ é avaliada. A expressão $\texttt{E}_2$ é escolhida caso o resultado seja verdadeiro, caso contrário a expressão $\texttt{E}_3$ é escolhida.

$$\mathcal{E}[\![\texttt{E}_1\ \texttt{`?'}\ \ \texttt{E}_2\ \texttt{`:'}\ \ \texttt{E}_2]\!] = \text{choose}(\mathcal{E}[\![\texttt{E}_1]\!], \mathcal{E}[\![\texttt{E}_2]\!], \mathcal{E}[\![\texttt{E}_3]\!])$$

Figura 4.11: Exemplo de semântica de expressões condicionais.

4.3.7 Ordem de Avaliação

Expressões são entidades recursivas. Para avaliar uma expressão, pode ser necessário avaliar primeiro diversas expressões menores, como os parâmetros de uma chamada de função. E, como essas subexpressões podem ter efeito colateral, a ordem de avaliação pode afetar a semântico dos programas. Por isso, as linguagens geralmente definem a ordem de avaliação de subexpressões. Algumas avaliam sempre da esquerda para a direita, outras avaliam primeiro as subexpressões mais complexas e outras ainda avaliam todas as subexpressões até obterem os valores envolvidos antes de começar a avaliar a expressão em si.

Cada componente apresentado neste trabalho implementa uma ordem de avaliação das expressões que recebe, de modo a se obter equações legíveis e eficientes. Embora seja possível determinar essa ordem observando a implementação de cada componente, o usuário da biblioteca não deve fazer suposições sobre a ordem de avaliação de expressões empregada em cada caso.

Se a ordem de avaliação for relevante para a semântica descrita, então esta deve ser definida explicitamente, o que pode ser feito por meio do componente `pipe`. Este recebe como parâmetros uma denotação de expressão e uma abstração λ. Aquela é avaliada e seu valor é passado a esta. Em relação ao contexto, o componente `pipe` define que este flui de seus antecedentes para a expressão e desta para a abstração λ, de forma análoga ao sequenciamento de comandos.

$$\begin{aligned}\mathcal{E}[\![\texttt{I '(' } \texttt{E}_1 \texttt{ ',' } \texttt{E}_2 \texttt{ ')'}]\!] = \text{pipe}(&\mathcal{E}[\![\texttt{E}_2]\!],\\ &\lambda \text{e}_2.\text{pipe}(\mathcal{E}[\![\texttt{E}_1]\!],\\ &\lambda \text{e}_1.\text{call}(\text{lookup}(\texttt{I}), [\text{e}_1 \bullet \text{e}_2])))\end{aligned}$$

Figura 4.12: Exemplo de definição de ordem de avaliação de expressões utilizando o componente `pipe`.

A Figura 4.12 mostra a semântica de uma chamada de função em que o segundo parâmetro é avaliado antes do primeiro. Se, por exemplo, os dois parâmetros são lidos a partir da entrada, então o estado da máquina é alterado durante sua avaliação. Neste caso, suponha que os próximos dois valores da entrada correspondem aos números 5 e 7. Como o segundo parâmetro é avaliado com o contexto corrente, seu valor passa a ser 5, o próximo valor da entrada. Quando o primeiro parâmetro é avaliado pelo segundo componente `pipe`, o próximo valor da entrada é 7, e esse passa a ser o primeiro parâmetro da função.

É importante notar que o valor recebido pela abstração λ pode ser utilizado posteriormente sem a necessidade de reavaliação. Assim, é possível utilizar o compo-

nente `pipe` para manipular valores durante a avaliação de uma expressão que possui efeito colateral. A Figura 4.13 mostra um exemplo de semântica do operador '++'. O componente `pipe` é utilizado para manipular o valor da variável antes do incremento, possibilitando que este seja o resultado da expressão. O componente `deref` é utilizado para obter o valor da variável, pois o valor associado ao `location` é alterado pelo incremento.

$$\mathcal{E}[\![\texttt{I '++'}]\!] = \text{pipe(deref(lookup(I)), } \lambda\text{e.valof(increment, resultIs(e)))}$$
$$\text{where increment} = \text{assign(lookup(I), apply('+', e, int('1')))}$$

Figura 4.13: Exemplo de semântica de incremento utilizando o componente `pipe`.

4.4 Manipulação do Estado da Máquina

Os comandos podem ser divididos de acordo com a forma como manipulam o estado da máquina. Alguns comandos, como a atribuição e o incremento, alteram o valor das variáveis. Outros comandos manipulam a entrada e a saída do programa – estes são apresentados na Seção 4.2.1. E alguns comandos manipulam o fluxo de execução, permitindo a implementação de comportamentos complexos, que se adaptam ao contexto da execução.

Nas linguagens de programação, é comum encontrar construções que compartilham características com expressões e comandos. Estas construções são consideradas expressões com efeito colateral. Dizer que uma expressão possui efeito colateral significa dizer que ela altera o estado da máquina. Um exemplo simples é a função, que em muitas linguagens imperativas pode conter comandos que alteram variáveis visíveis no resto do programa. Em contrapartida, há construções cujo objetivo primária é alterar o estado, mas que também possuem um valor como resultado. Um exemplo é a atribuição, que em muitas linguagens, além de alterar o valor de uma variável, retorna o novo valor como resultado. Apesar dessa proximidade, os componentes são organizados neste texto de acordo com sua função principal.

4.4.1 Atribuição

A atribuição é o comando que caracteriza as linguagens imperativas. Por meio dela, valores são atribuídos às variáveis de um programa. O alvo da atribuição – a variável ou posição de memória que receberá o novo valor – é chamado de lado esquerdo em referência à sintaxe da construção. O lado esquerdo pode conter uma expressão complexa,

como uma indexação de arranjo feita por chamada de função, mas o seu resultado deve sempre ser uma posição de memória. Seguindo a mesma lógica, a expressão que gera o valor a ser guardado é chamada de lado direito. Uma atribuição do tipo `a += b` nada mais é que uma forma sucinta de se escrever `a = a' + b`, onde `a'` é o valor associado ao *location* produzido pela avaliação de `a`.

É comum tratar o operador '=' como um operador binário que produz como resultado o valor de seu segundo operando e, como efeito colateral, a atribuição especificada. Por conseguinte, torna-se possível escrever `a = (b = c)`, cuja semântica é atribuir o valor da variável `c` às variáveis `a` e `b`. Dessa forma, a atribuição passa a ser vista como uma expressão.

$$\mathcal{E}[\![\texttt{E}_1\ \texttt{':='}\ \texttt{E}_2]\!] = \text{assign}(\mathcal{E}[\![\texttt{E}_1]\!],\mathcal{E}[\![\texttt{E}_2]\!])$$

Figura 4.14: Exemplo de semântica de atribuição utilizando o componente `assign`.

A Figura 4.14 ilustra o uso do componente `assign`. O lado esquerdo é avaliado para se obter o *location* onde o valor deve ser armazenado. Este é obtido por meio da avaliação do lado direito. Caso essa avaliação resulte em um *location*, o valor correspondente armazenado no *store* é recuperado. Assim como ocorre em outros componentes, a ordem de avaliação dos dois lados da expressão é feita internamente da esquerda para a direita. Outras ordens de avaliação devem ser modeladas explicitamente por meio do componente `pipe`, apresentado na Seção 4.3.7.

4.4.2 Skips

O comando vazio nada mais é do que uma forma de denotar a ausência de computação. Sem qualquer efeito colateral, o comando vazio, representado pelo componente `skip`, é geralmente utilizado em conjunto com estruturas mais complexas. Por exemplo, pode-se representar o comando *if-then* por meio de um comando *if-then-else* cujo braço *else* possui apenas um comando vazio. Um exemplo de uso do componente `skip` pode ser visto na Seção 4.4.4.

4.4.3 Comandos Sequenciais

Um programa em linguagem imperativa é um conjunto de passos que devem ser executados em sequência. De fato, a execução de um programa é formada pela execução de uma sequência de comandos. Esse comportamento básico pode ser modelado pelo componente `seq`. Um exemplo pode ser visto na Figura 4.15.

$\mathcal{C}[\![\mathtt{C}_1\ \text{`;'}\ \mathtt{C}_2]\!] = \text{seq}\ (\mathtt{C}_1,\mathtt{C}_2)$

Figura 4.15: Exemplo de semântica de sequenciamento de comandos utilizando o componente `seq`.

No entanto, a implementação de uma lógica complexa depende do uso de construções capazes de alterar o fluxo de controle do programa, modelando a sequência de passos de acordo com as particularidades de cada execução. Para alterar o fluxo de controle, são utilizadas construções como saltos, condicionais, laços e exceções.

4.4.4 Comandos Condicionais

A forma mais simples que a execução condicional de seções de código pode assumir é o *if-then-else*. Modelada pelo componente `choose`, a semântica de execução condicional é executar uma de duas seções de código de acordo com a avaliação de uma condição. Um exemplo de uso do componente `choose` pode ser encontrado na Figura 4.16. Nesse exemplo, a expressão de condição `E` é avaliada, e, caso o resultado seja verdadeiro, a primeira seção de código ($\mathtt{C}_1$) é executada, caso contrário a segunda seção ($\mathtt{C}_2$) é executada.

$\mathcal{E}[\![\text{`if'}\ \mathtt{E}\ \text{`then'}\ \mathtt{C}_1\ \text{`else'}\ \mathtt{C}_2]\!] = \text{choose}(\mathcal{E}[\![\mathtt{E}]\!],\mathcal{C}[\![\mathtt{C}_1]\!],\mathcal{C}[\![\mathtt{C}_2]\!])$
$\mathcal{E}[\![\text{`if'}\ \mathtt{E}\ \text{`then'}\ \mathtt{C}]\!] = \text{choose}(\mathcal{E}[\![\mathtt{E}]\!],\mathcal{C}[\![\mathtt{C}]\!],\text{skip})$

Figura 4.16: Exemplo de semântica de comando condicional utilizando o componente `choose`.

4.4.5 Comandos Iterativos

Um laço é uma construção sintática cuja semântica é executar um determinado comando ou conjunto de comandos enquanto uma certa expressão for avaliada como verdadeira. Um dos usos mais comuns do laço é realizar um mesmo conjunto de operações sobre múltiplos dados ou variáveis. No paradigma imperativo, cada ciclo do laço afeta os seguintes por meio do *store*. Em linguagens funcionais, laços são reproduzidos por meio de recursão, sendo a comunicação entre ciclos desempenhada pelos valores passados aos parâmetros de cada ciclo.

Quanto à semântica, a expressão que determina se o laço será interrompido pode ser avaliada no início ou no final do laço. Exemplos na linguagem C são os laços *while* e *do-while*. Com o objetivo de possibilitar a modelagem de diferentes semânticas de laço,

o componente `loop` testa duas expressões, uma antes e outra depois de cada iteração. Para modelar um *while*, basta passar o valor verdadeiro como a segunda expressão do componente `loop`, enquanto o *do-while* pode ser expresso pelo uso da expressão verdadeiro como primeiro parâmetro. A Figura 4.17 mostra alguns exemplos de laços.

$\mathcal{C}[\![$`'while' E 'do' C`$]\!]$ = loop($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$, bool('true'))
$\mathcal{C}[\![$`'repeat' C 'until' E`$]\!]$ =
loop(bool('true'), $\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$, apply('=', $\mathcal{E}[\![$`E`$]\!]$, bool(''false)))

Figura 4.17: Exemplo de semântica de laço.

O *for* é outro exemplo de comando iterativo. Geralmente, o *for* itera em um conjunto de valores a partir dos quais a computação é feita. Algumas vezes essa característica é utilizada para repetir um mesmo comando um certo número de vezes. Primeiro, considere o *for* da linguagem ALGOL 60. Esta construção é composta por três tipos de iteração que podem ser compostas livremente. A primeira delas é apenas um passo do *for*, a segunda é semelhante a um *while* e a terceira permite iterar por um conjunto de valores, possivelmente saltando alguns deles. A cada passo do *for*, um valor é atribuído a uma variável de controle. A Figura 4.18 mostra os tipos de iteração do *for* de ALGOL 60 e a sua semântica.

$\mathcal{C}[\![$`'for' I ':=' F 'do' C`$]\!]$ = for(`I`, $\mathcal{F}[\![$`F`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$)
$\mathcal{F}[\![$`E`$]\!]$ = singleStep($\mathcal{E}[\![$`E`$]\!]$)
$\mathcal{F}[\![$`E`$_1$ `'while' E`$_2]\!]$ = while($\mathcal{E}[\![$`E`$_1]\!]$, $\mathcal{E}[\![$`E`$_2]\!]$)
$\mathcal{F}[\![$`E`$_1$ `'step' E`$_2$ `'until' E`$_3]\!]$ = range($\mathcal{E}[\![$`E`$_1]\!]$, $\mathcal{E}[\![$`E`$_2]\!]$, $\mathcal{E}[\![$`E`$_3]\!]$)
$\mathcal{F}[\![$`F`$_1$ `',' F`$_2]\!]$ = seq($\mathcal{F}[\![$`F`$_1]\!]$, $\mathcal{F}[\![$`F`$_2]\!]$)

Figura 4.18: Exemplo de semântica do *for* de ALGOL 60.

A cada passo, o iterador calcula um valor e passa para o corpo do *for*. O primeiro iterador passa o valor de $\mathcal{E}[\![$`E`$]\!]$ ao corpo do *for* e termina. O segundo iterador passa ao corpo o valor de $\mathcal{E}[\![$`E`$_1]\!]$ enquanto $\mathcal{E}[\![$`E`$_2]\!]$ for verdadeiro. O terceiro iterador passa ao corpo do *for* um valor a cada passo. No primeiro passo, o valor $\mathcal{E}[\![$`E`$_1]\!]$ é usado, daí em diante, o último valor usado é somado a $\mathcal{E}[\![$`E`$_2]\!]$ e passado ao corpo até que esse valor exceda $\mathcal{E}[\![$`E`$_3]\!]$. A combinação dos iteradores resulta na execução de cada um deles na sequência em que são combinados.

Algumas linguagens mais recentes apresentam uma construção *for* mais simples e mais genérica. Esse *for* possui quatro parâmetros. O primeiro é no início da execução

do *for*. O segundo é um teste a ser realizado antes de cada passo, caso o resultado seja falso a execução do *for* termina. O terceiro é executado após cada passo e o terceiro é o próprio corpo do *for*. Com essa construção é possível emular cada um dos iteradores do *for* de ALGOL 60, embora a emulação do passo único possa ser um pouco confusa – e mais facilmente emulada por uma declaração envolvida em um bloco juntamente com o corpo do *for*.

$\mathcal{C}[\![$`'for' '(' D E` C_1 `')'` $C_2$$]\!]$ =
    escape(
        for($\mathcal{D}[\![$D$]\!]$, $\mathcal{E}[\![$E$]\!]$, $\mathcal{C}[\![C_1]\!]$, escape($\mathcal{C}[\![C_2]\!]$, 'continue')),
    'break')

Figura 4.19: Exemplo de semântica de um *for* genérico.

A Figura 4.19 mostra a semântica de um *for* genérico. O primeiro e o segundo parâmetros podem ser comandos ou declarações. O corpo do *for*, último parâmetro, pode ser uma expressão, um comando ou uma expressão. No entanto, é necessário que o segundo parâmetro seja sempre uma expressão, posto que seu valor é utilizado no teste realizado a cada passo. Nesse exemplo, o componente `escape`, apresentado na Seção 4.8.1, é utilizado para modelar a semântica de saída das construções *break* e *continue*.

4.4.6 Comandos que Retornam Valor

Como ocorre no caso da atribuição, pode ser necessário que um comando resulte em um valor. Por exemplo, em Java o corpo de função é um bloco de comandos que retorna um valor. O componente `valof` é responsável por executar um bloco de comandos e retornar um valor. O retorno é feito pelo componente `resultIs`, o qual recebe uma expressão cujo valor é utilizado como resultado do bloco de comandos. Por exemplo, o corpo de uma função poderia ser modelado por meio do componente `valof`. Exceto pela semântica de retorno ao ponto de chamada, a construção *return* poderia ser modelada por meio do componente `resultIs`, interrompendo a execução do corpo da função, avaliando a expressão recebida e retornando o seu valor.

O exemplo da Figura 4.20 ilustra o uso desses componentes por meio da tradução do corpo de uma função. O componente `resultIs` atribui o valor 1 da variável `a` ao componente `valof`. A segunda atribuição não é realizada por causa da mudança no fluxo de controle.

```
fun ... {                    | ... valof(
                             |         seq(
    a = 1;                   |             assign('a', int('1')),
    return a;                |             seq(resultIs(lookup('a')),
    a = 2;                   |             assign('a',2))
                             |         )
}                            |     )
```

Figura 4.20: Exemplo de semântica do corpo de uma função.

4.4.7 Gerência de Memória

O *store* é utilizado para armazenar valores. As variáveis constituem o maios comum mecanismo de armazenamento encontrado nas linguagens de programação imperativas. Como mencionado na Seção 4.3.2, seu valor pode ser lido e atualizado durante a execução do programa. Variáveis são criadas por meio do componente `ref`. Este avalia uma expressão recebida como parâmetro e armazena seu valor no *store*, passando para a continuação o *location* associado ao valor. No entanto, o componente `ref` não inspeciona o valor de variáveis. Portanto, quando a expressão recebida é uma variável, a semântica de `ref` é criar um ponteiro. Quando, porém, a expressão passada é um valor ou uma operação, cria-se uma variável comum. De forma semelhante, pode-se criar variáveis constantes por meio do componente `const`. A semântica deste componente é análoga à de `ref`. No entanto, não se pode alterar o valor da variável criada.

$\mathcal{D}[\![\mathtt{I_1 = `const' `\&' I_2}]\!] = \text{bind}(\mathrm{I_1}, \text{const}(\text{ref}(\text{lookup}(\mathrm{I_2}))))$
$\mathcal{D}[\![\mathtt{I = `int' i}]\!] = \text{bind}(\mathrm{I}, \text{ref}(\text{int}(\mathrm{i})))$
$\mathcal{E}[\![\mathtt{`\&' I}]\!] = \text{ref}(\text{lookup}(\mathrm{I}))$
$\mathcal{E}[\![\mathtt{`*' I}]\!] = \text{deref}(\text{lookup}(\mathrm{I}))$

Figura 4.21: Exemplo de semântica de variáveis.

A Figura 4.21 ilustra a criação de variáveis, constantes e ponteiros. Como mostra a última equação do exemplo, o valor da variável apontada por um ponteiro pode ser obtido por meio do componente `deref`. Note que o valor de qualquer variável pode ser acessado por meio do componente `deref`. No caso de ponteiros, esse valor é um *location*. De forma semelhante, o *location* de qualquer variável pode ser acessado por meio do componente `lookup`.

O *store* possui duas partes. A primeira é uma estrutura chamada de *stack*. Unidades de armazenamento são criadas de forma contígua nessa estrutura. Quando é

necessário remover unidades, uma seção inteira do *stack* pode ser removida. Esse comportamento está intimamente ligado com a forma que procedimentos são chamados, ou, às vezes, como blocos são executados. Quando um procedimento é chamado, uma nova seção do *stack* é delimitada, onde suas variáveis são guardadas. Quando o procedimento termina, sua seção do *stack* é apagada, liberando espaço de armazenamento.

A segunda parte é chamada de *heap*. Essa estrutura permite criação e liberação de memória sem a delimitação de áreas específicas. É possível remover uma variável qualquer do *heap* diretamente.

Normalmente, os componentes armazenam valores no *stack*. Para criar uma nova seção de armazenamento, basta utilizar o componente `push`. Para remover a última seção de armazenamento criada, basta utilizar o componente `pop`. O exemplo da Figura 4.22 mostra como criar e remover seções do `stack`.

$\mathcal{C}[\![\texttt{`\{' C `\}'}]\!] = \text{seq}(\text{push},\text{seq}(\mathcal{C}[\![\texttt{C}]\!],\text{pop}))$

Figura 4.22: Exemplo de semântica de *stack*.

Valores podem ser armazenados no *heap* por meio do componente `alloc`. No entanto, utilizando esse componente em uma declaração, a variável criada é na verdade um ponteiro, guardando apenas uma referência ao valor armazenado no *heap*. Além disso, não se pode passar um valor complexo já computado a esse componente, apenas a expressão em si, dado que é necessário armazenar cada um dos valores primitivos no *heap*. Uma unidade do *heap* pode ser liberada por meio do componente `free`. A Figura 4.23 mostra exemplos de uso desses componentes.

$\mathcal{C}[\![\texttt{`new' E}]\!] = \text{alloc}(\mathcal{E}[\![\texttt{E}]\!])$
$\mathcal{C}[\![\texttt{`free' I}]\!] = \text{free}(\text{lookup}(\text{I}))$

Figura 4.23: Exemplo de semântica de alocação, liberação e ponteiros.

Em contraste com comandos e abstrações, a gerência de memória é um conceito concreto, diretamente ligado à implementação da linguagem. Sua inclusão na biblioteca de componentes tem o objetivo de facilitar a prototipação de analisadores semânticos. Dada a importância da gerência de memória nas linguagens atuais, é necessário que se tenha mecanismos para representá-la. Por exemplo, a definição da linguagem C++ sem a definição da alocação de memória dificulta o entendimento da construção delete e dos possíveis erros relacionados a ponteiros. Esse conceito também é importante para definir destrutores de objetos. No entanto, sua inclusão em definições é opcional, o que

possibilita definições abstratas que dão liberdade aos implementadores para escolher o tipo gerência de memória que sirva melhor à linguagem implementada.

4.5 Definição do Ambiente de Execução

O *binding*, ou ligação, é um mapeamento entre um identificador e um objeto, como uma variável ou um procedimento. O conjunto de *bindings* válidos em um determinado ponto do programa é chamado de *environment*. O valor associado a um *binding* pode ser obtido por meio do componente `lookup`. A maioria das linguagens de programação permite que se utilize os mesmos nomes em diferentes partes de um programa.

4.5.1 Escopo e Visibilidade

Escopo é o nome que se dá à parte de um programa em que um determinado *binding* é válido. Por exemplo, o escopo da declaração de uma variável `x` qualquer é a parte do programa em que o significado de `x` é dado por essa declaração. Entretanto, o escopo não se refere à parte do programa em que a variável é válida, a isso se dá o nome de *lifetime* e é a parte do programa em que a variável permanece alocada na memória. De forma simples, o escopo se refere à parte do programa em que um *binding* é válido, enquanto *lifetime* se refere à parte do programa em que uma alocação é válida.

Essa diferença é importante porque é muito comum que o escopo de uma variável seja apenas uma porção do seu *lifetime*. Isso se deve ao fato de que as linguagens de programação possuem diversos mecanismos de controle de visibilidade. Alguns mecanismos permitem isolar um trecho de código, fazendo com que todos *bindings* definidos no resto do programa sejam invisíveis naquele trecho. Outros mecanismos permitem selecionar quais *bindings* serão visíveis. O exemplo mais emblemático de controle de visibilidade é o módulo, o qual engloba os dois mecanismos descritos.

O módulo é uma estrutura que separa uma seção de código do resto do programa. Seu objetivo é definir quais declarações externas são visíveis dentro do módulo e quais declarações internas são visíveis fora do módulo. Um exemplo de módulo é a classe de C++. Toda declaração pública contida em uma classe é visível para o resto do programa, mas as declarações privadas são visíveis apenas dentro da classe. No entanto, todas as declarações visíveis fora da classe são visíveis dentro dela. Outros módulos são mais restritivos. Em Haskell, por exemplo, um módulo deve importar declarações externas para que elas lhe sejam visíveis, assim como deve exportar dentre suas declarações aquelas que devem ser visíveis ao resto do programa.

Contrastando o módulo de Haskell e a classe de C++, pode-se perceber que há variadas configurações para as restrições de visibilidade entre uma seção de código e o resto do programa. O componente `makeAbstraction` tem por objetivo modelar a semântica do módulo de forma genérica. O componente recebe duas opções de visibilidade, uma para cada sentido. Quando é feita alguma restrição, torna-se necessário dizer quais nomes devem ser importados ou exportados.

$$\mathcal{D}[\![\texttt{`module' I `\{' D `\}' `import' } \mathtt{I}_i \texttt{ `export' } \mathtt{I}_e]\!] = \\ \text{makeAbstraction}(\mathcal{D}[\![\mathtt{D}]\!], (\text{Opaque}, \mathtt{I}_i), (\text{Encapsulated}, \mathtt{I}_e))$$

Figura 4.24: Exemplo de semântica de módulos.

A Figura 4.24 mostra um exemplo de definição semântica de um módulo. A opção `Opaque` faz com que apenas os identificadores $\mathtt{I}_i$ tenham suas ligações importadas para o módulo. Um módulo que importa todas as ligações externas pode ser modelado por meio da opção `Transparent`. A opção `Encapsulated` faz com que apenas os identificadores $\mathtt{I}_e$ tenham suas ligações exportadas para o resto do programa. A opção `Visible` faz com que todas as ligações sejam exportadas.

O bloco é outra construção relacionada ao escopo. A semântica de um bloco determina que nenhuma das declarações feitas dentro do bloco seja válida no resto do programa. O *closure*, por outro lado, recebe um conjunto de declarações a serem utilizados durante sua execução. A Figura 4.25 mostra exemplos de definição semântica de *closure* e bloco. Os componentes `makeBlock`, `makeClosure` e `makeAbstraction` modelam respectivamente blocos, *closures* e módulos de comandos, expressões ou declarações.

$$\mathcal{E}[\![\mathtt{E \texttt{ `where' } D}]\!] = \text{makeClosure}(\mathcal{D}[\![\mathtt{D}]\!], \mathcal{E}[\![\mathtt{E}]\!])$$
$$\mathcal{C}[\![\texttt{`\{' C `\}'}]\!] = \text{makeBlock}(\mathcal{C}[\![\mathtt{C}]\!])$$

Figura 4.25: Exemplos de semântica de blocos.

Esses componentes podem ainda ser utilizados para definir a semântica de outras construções que delimitam escopo. Por exemplo, `makeBlock` pode ser utilizado para manter a definição de procedimentos aninhados restrita aos procedimentos circundantes em uma linguagem como Pascal, e *makeAbstraction* pode ser utilizado para controlar a visibilidade de *inner classes* em Java. Para definir casos em que declarações sobrescrevem ligações prévias com o mesmo identificador, basta não restringir o escopo.

O *enviroment* se organiza de tal forma que novas ligações com o mesmo identificador prevalecem sobre as anteriores.

4.5.2 Escopo Dinâmico e Escopo Estático

Quando um nome não declarado no corpo do procedimento é utilizado, resta a questão de decidir onde a declaração deve ser procurada. Nomes não declarados utilizados dentro do corpo do procedimento poderiam ser procurados tanto no ponto de declaração quanto no ponto de chamada. Quando a procura é feita no ponto de declaração, o escopo é dito estático, pois não depende do fluxo de execução do programa, mas sim da sua estrutura. Quando a procura é feita no ponto de chamada, o escopo é dito dinâmico.

O tipo de escopo utilizado pode ser definido por meio do terceiro parâmetro do componente `bind` para cada declaração de procedimento ou função, como pode ser visto na Seção 4.6. Há duas funções pré-definidas, `static` e `dynamic`, que podem ser utilizadas para representar escopo estático e dinâmico respectivamente. Outras funções podem ser definidas pelo usuário da biblioteca, que, dados os *environments* do ponto de declaração e do ponto de chamada, retornam o *environemnt* a ser utilizado dentro do corpo do procedimento.

A diferença entre escopo estático e dinâmico também se aplica a saltos, saídas, exceções e declarações de objeto. De forma semelhante ao *binding* de procedimentos, o último parâmetro dos componentes *addLabel*, *escape*, *catch* e *class* é a função de escopo. Caso esse parâmetro não seja fornecido, o tipo de escopo definido globalmente é utilizado. Essa definição pode ser feita por meio dos componentes: `useStaticScope` e `useDynamicScope`, como no exemplo da Figura 4.26. Caso não haja definição global, os componentes utilizam escopo estático.

$$\mathcal{P}[\![\texttt{`java' Id `in' D}]\!]\ \text{i args} = \text{result}(\text{run}\ (\text{useStaticScope}(\text{c}))\ \text{i}\ \text{r}_0\ \text{s}_0)$$
$$\text{where} \quad \text{c} = \text{addEnv}(\text{accum}(\texttt{D}), \text{call}(\text{lookup}(\text{Id}, \text{`main'}), \text{args}))$$
$$\text{result} = \lambda(\text{t, e, r, s}) \rightarrow (\text{e, s output})$$

Figura 4.26: Exemplo de definição de tipo de escopo.

O uso de escopo dinâmico indica que o *environment* corrente deve ser utilizado após um salto no caso do componente *addLabel*, após o escape de uma construção no caso de `escape`, durante o tratamento de uma exceção no caso de `catch` e durante a execução das declarações de objeto no caso de `class`. Entende-se por *environment* corrente aquele válido no momento em que o fluxo de controle é alterado. Por exemplo,

no caso do salto com escopo dinâmico, o `environment` usado é aquele válido quando do uso do componente `jump`.

4.5.3 Declarações

Declarações têm por objetivo criar um nome visível no *environment* e associar a esse nome algum valor semântico. Por exemplo, ao se declarar um procedimento, associa-se seu nome ao *environment* corrente e aos comandos que devem ser executados quando o procedimento for chamado. Quando uma variável é declarada, há dois passos observáveis. Primeiro, cria-se o nome no *environment* e associa-se a ele um location, fazendo assim a ligação com o *store*. Depois, associa-se um valor ao location, dando valor à variável. O primeiro passado é chamado de declaração, pois declara a existência da variável, o segundo é chamado de definição, pois define o valor da variável.

Na biblioteca, decidiu-se separar a elaboração da declaração – fase em que se estabelece a relação entre nome e valor semântico – e o acúmulo da declaração no *environment*. Isso foi feito para que diferentes formas de se administrar o environment sejam expressáveis. Dessa forma, criou-se um componente chamado `accum`, cuja semântica é acrescentar uma declaração ao *environment* corrente. Diferentes semânticas são possíveis ao se variar o *environment* passado ao componente `accum`.

Para declarar variáveis é utilizado o componente `bind`. Este associa um nome recebido como parâmetro a um valor. Para criar variáveis, o valor associado deve ser um *location*. No entanto, é também possível dar nome a procedimentos, classes e valores básicos da linguagem. O componente `bind` pode receber também um terceiro parâmetro, o qual define o tipo do valor nomeado.

$\mathcal{D}[\![$`'var' I`$]\!]$ = bind(I, ref(undefined))
$\mathcal{D}[\![$`'const' I '=' E`$]\!]$ = bind(I, const($\mathcal{E}[\![$`E`$]\!]$))
$\mathcal{D}[\![$`'int' I`$]\!]$ = bind(I, ref(int('0')), 'int')
$\mathcal{D}[\![$`'const' 'string' I '=' E`$]\!]$ = bind(I, ref(deref($\mathcal{E}[\![$`E`$]\!]$)), 'string')
$\mathcal{D}[\![$`'def' I '=' E`$]\!]$ = bind(I, deref($\mathcal{E}[\![$`E`$]\!]$))

Figura 4.27: Exemplo de semântica de declarações de variáveis e variáveis constantes.

O exemplo da Figura 4.27 ilustra a declaração de variáveis. Como mostrado no exemplo, qualquer valor expressável da linguagem pode ser armazenado em variáveis. Para declarar um arranjo, basta passar o arranjo construído ao componente de declaração. O valor *undefined* indica que a variável não foi inicializada. O exemplo também mostra que é possível associar um tipo à variável. Enquanto as anteriores envolvem

armazenamento de valores no *store*, a última equação mostra a associação de um nome a um valor no *environment*.

4.5.4 Declarações Compostas

Em um programa, geralmente são realizadas diversas declarações, e estas podem ser dependentes entre si. Quando vistas em conjunto, as declarações podem ser sequenciais, colaterais ou mutuamente recursivas. Quando sequenciais, cada declaração conhece e pode fazer uso das anteriores. Quando colaterais, as declarações devem ser completamente independentes uma das outras. Por fim, em um conjunto de declarações mutuamente recursivas, cada declaração conhece e pode fazer uso de todas as outras.

Esses três relacionamentos entre declarações são modelados respectivamente pelos componentes `elabSeq`, `elabCol` e `elabRec`. Esses componentes podem receber declarações simples, ou declarações compostas, criando assim elaborados fluxos de declaração. `elabRec` pode receber duas declarações mutuamente recursivas, ou apenas um declaração recursiva. Isso permite por exemplo, criar no primeiro caso uma lista de funções que chamam uma às outras, ou no segundo caso um procedimento recursivo. A Figura 4.28 mostra exemplos de semântica de declarações compostas.

$$\mathcal{D}[\![\texttt{D}_1\ \texttt{`;'}\ \texttt{D}_2]\!] = \text{elabSeq}(\mathcal{D}[\![\texttt{D}_1]\!], \mathcal{D}[\![\texttt{D}_2]\!])$$
$$\mathcal{D}[\![\texttt{D}_1\ \texttt{`,'}\ \texttt{D}_2]\!] = \text{elabCol}(\mathcal{D}[\![\texttt{D}_1]\!], \mathcal{D}[\![\texttt{D}_2]\!])$$
$$\mathcal{D}[\![\texttt{`rec'}\ \texttt{D}_1\ \texttt{`,'}\ \texttt{D}_2]\!] = \text{elabRec}(\mathcal{D}[\![\texttt{D}_1]\!], \mathcal{D}[\![\texttt{D}_2]\!])$$
$$\mathcal{D}[\![\texttt{`proc'}\ \texttt{I}\ \texttt{`()'}\ \texttt{C}]\!] = \text{elabRec}(\text{bind}(\texttt{I}, \mathcal{C}[\![\texttt{C}]\!]))$$

Figura 4.28: Exemplos de semântica de declarações compostas.

4.6 Abstração Procedural

Procedimentos são mecanismos para realizar a abstração de instruções de um programa. Em vez de repetir a mesma sequência de instruções diversas vezes no mesmo programa, abstrai-se essa sequência, substituindo as suas diversas ocorrências por uma chamada de procedimento. Podem-se ainda criar procedimentos mais genéricos por meio da introdução de parâmetros. Dessa forma, evitam-se as desvantagens da duplicação de código e facilita-se o reúso.

O princípio da abstração estabelece que é possível abstrair em procedimentos quaisquer construções sintáticas, desde que estas especifiquem algum tipo de compu-

tação (Watt & Findlay 2004). Por exemplo, funções, comandos e unidades genéricas são abstrações de expressões, comandos e declarações respectivamente.

O componente utilizado para essa abstração é `proc`. Esse componente encapsula a computação para que esta possa ser executada posteriormente pelo componente `call`. Além disso, procedimentos podem utilizar escopo estático ou dinâmico. Utilizando o primeiro, o *environment* da declaração do procedimento é utilizado para resolver nomes não declarados no corpo do procedimento. Utilizando escopo dinâmico, o *environment* corrente no ponto de chamada é utilizado para esse fim. O tipo de escopo é definido pelo terceiro parâmetro do componente `proc`. A biblioteca provê duas implementações básicas, `static` e `dynamic`, utilizadas para representar escopo estático e dinâmico respectivamente. Um exemplo de definição da semântica de procedimentos pode ser visto na Figura 4.29. Nesse exemplo, o procedimento recebe um nome, o qual pode ser utilizado posteriormente em chamadas. No entanto, o procedimento, por ser um valor expressável, poderia também ser armazenado em uma variável. A primeira equação mostra a criação de um procedimento dinâmico, por meio do parâmetro *dynamic*. Na segunda equação, o tipo de escopo é omitido. Caso não tenha sido especificado um tipo padrão por meio de `useDynamicScope` ou `useStaticScope`, utiliza-se escopo estático.

$$\mathcal{D}[\![\texttt{`dyn-proc' I `=' `\{' C `\}'}]\!] = \text{bind}(\texttt{I}, \text{proc}(\mathcal{C}[\![\texttt{C}]\!], \text{dynamic}))$$
$$\mathcal{D}[\![\texttt{`proc' I `=' `\{' C `\}'}]\!] = \text{bind}(\texttt{I}, \text{proc}(\mathcal{C}[\![\texttt{C}]\!]))$$

Figura 4.29: Exemplo de semântica de procedimento com escopo estático.

Algumas questões relacionadas à semântica de procedimentos não são abordadas pela biblioteca de componentes de semântica denotacional, ficando a cargo da definição de semântica estática das linguagens. Em relação ao polimorfismo, fica a cargo da semântica estática atribuir identificadores diferentes às diversas definições de uma função polimórfica e fazer a chamada correta de acordo com os parâmetros passados. Valores *default* de parâmetros também não são modelados pela biblioteca. Outra questão não abordada é a verificação de caminhos. Por exemplo, geralmente, as linguagens de programação exigem que todos os caminhos em uma função retornem valores. Algumas linguagens também exigem que não haja código escrito após o sequenciador de retorno, visto que esse código não é alcançável. Essas duas verificações ficam a cargo da semântica estática. Finalmente, os tipos dos parâmetros formais também não são modelados pelos componentes.

4.6.1 Parâmetros e Argumentos

Os parâmetros são conexões entre o ponto de chamada de uma função ou procedimento e o seu corpo. Por meio dos parâmetros é possível informar os valores a serem utilizados nas operações abstraídas e comunicar alterações ao ponto de chamada – o que é semelhante a uma função retornando múltiplos valores. No momento da declaração, definem-se os parâmetros formais, nomes que identificam os valores passados permitindo seu uso pelo código abstraído. No momento da chamada, os valores passados ao procedimento ou função são conectados aos parâmetros formais, dá-se a esses valores o nome parâmetros reais. Essa conexão é chamada de passagem de parâmetros.

O componente `parametrize` é responsável por conectar os parâmetros reais aos parâmetros formais e, no fim da chamada, comunicar aos parâmetros reais quaisquer alterações necessárias. A forma como parâmetros reais e formais são conectados é definida pelo que tipo de passagem de parâmetros.

```
int f (int x) { if (x > 0) x = 3; return 4; }
a = 2;
b = f(a);
```

Figura 4.30: Exemplo de passagem de parâmetros.

Observe o exemplo da Figura 4.30. Caso o tipo de passagem de parâmetros utilizado seja por valor, o parâmetro `x` de `f` receberá o valor 2. Caso o tipo utilizado seja *value and result*, `x` receberá o valor 2, mas no fim da chamada a variável a receberá o valor 3. Esse tipo de passagem é semelhante à passagem por referência, porém, a alteração só é comunicada ao parâmetro real no fim da chamada. Caso o parâmetro `x` seja constante, o comportamento será o mesmo apresentado na passagem por valor, mas a tentativa de atribuir o valor 3 a `x` passa a ser proibida. Por fim, a passagem de parâmetros pode ser feito por nome. Nessa modalidade, o parâmetro real não é avaliado até o momento em que se faz necessário, no exemplo isso ocorre na condição do comando *if*. Um exemplo de parametrização pode ser visto na Figura 4.31.

```
int f (int x) {          | bind('f', proc(parametrize(
    C                    |                  [(value, 'i')],
                         |                  C
}                        | )))
```

Figura 4.31: Exemplo de semântica de parametrização de um procedimento.

4.6.2 Chamadas de Procedimentos e Funções

A chamada de um procedimento é o processo que reinsere a seção de código abstraída no corpo do programa. Uma chamada pode ser vista como uma marcação indicando que ali deve ser inserido o código correspondente ao procedimento chamado. Abstraído o gerenciamento de memória, uma chamada de procedimento sem parâmetros faz exatamente isso. Ou seja, o código do procedimento é executado no ponto do programa marcado pela chamada. O componente `call` é responsável pela chamada.

Adicionar parâmetros a esse cenário não é complicado. No momento da chamada, os parâmetros são avaliados de acordo com o tipo de passagem definido para cada um e os valores obtidos são associados aos nomes dos parâmetros formais. Como pode ser visto no exemplo a seguir, acrescentar parâmetros a declaração de um procedimento consiste em utilizar o componente `parametrize` e passar os parâmetros reais ao componente `call`. A chamada em si consiste em passar ao componente `call` o procedimento e a lista de parâmetros reais, como mostra o exemplo da Figura 4.32.

$\mathcal{D}[\![\texttt{I '(' A ')' '\{' C '\}'}]\!] = \text{bind(I, proc(parametrize(}\mathcal{A}[\![\texttt{A}]\!]\text{,}\mathcal{C}[\![\texttt{C}]\!]\text{), static))}$
$\mathcal{C}[\![\texttt{I '(' E ')'}]\!] = \text{call(lookup(I), }\mathcal{E}[\![\texttt{E}]\!]\text{)}$

Figura 4.32: Exemplo de semântica de chamada de procedimentos.

Assim como pode ser feito com expressões em outras situações, tanto o procedimento quanto os parâmetros reais podem ser avaliados antes de serem passados ao componente `call`. Além disso é possível utilizar um procedimento anônimo, definido apenas no momento da chamada, como pode ser visto no exemplo da Figura 4.33.

$\mathcal{E}[\![\texttt{'('}\ \lambda\ \texttt{A '.' E ')'}\ ]\!] = \text{proc(parametrize(}\mathcal{A}[\![\texttt{A}]\!]\text{, }\mathcal{E}[\![\texttt{E}]\!]\text{))}$
$\mathcal{E}[\![\texttt{E}_1\ \texttt{E}_2]\!] = \text{pipe(}\mathcal{E}[\![\texttt{E}_1]\!]\text{, }\lambda\text{f.pipe(}\mathcal{E}[\![\texttt{E}_2]\!]\text{, }\lambda\text{a.call(f, a)))}$

Figura 4.33: Exemplo de semântica de aplicação de funções.

Na biblioteca aqui apresentada o trabalho envolvido na chamada é dividido entre os quatro componentes relacionados à abstração de procedimentos. O componente `bind`, além de dar um nome ao procedimento, define o tipo de escopo a ser utilizado, o componente `parametrize` faz a associação entre parâmetros reais e formais e, finalmente, o componente `call` avalia os parâmetros e dispara a execução do procedimento.

4.6.3 As Etapas da Chamada

Há três componentes que delimitam as etapas de uma chamada. Esses componentes tratam da manipulação dos parâmetros implementando os diferentes tipos de passagem. É nesse aspecto que os componentes `call` e `parametrize` se encontram, pois as etapas aqui descritas são caracterizadas pela manipulação dos parâmetros. A classificação aqui utilizada foi adaptada de Gordon (1979), feitas algumas alterações na organização das etapas.

É na primeira etapa, representada por `evalPar`, que os parâmetros são avaliados antes de serem passados ao procedimento ou função. Nesse ponto, a passagem por nome se diferencia das demais, visto que o parâmetro não chega a ser avaliado, o que é feito apenas dentro do corpo do procedimento utilizando o `Store` corrente e o `Environment` do ponto de chamada. Sendo uma forma de *lazy evaluation*, o parâmetro só é avaliado quando seu valor se faz necessário.

O componente `entry` demarca o ponto de entrada do procedimento, onde o valor do parâmetro real é associado ao parâmetro formal. Nesse ponto, parâmetros passados por valor são armazenados no *Store* e parâmetros passados por referência tem seu *location* associado ao nome dos parâmetros formais. É também nesse ponto que parâmetros passam a ser tratados como constantes quando esse tipo de passagem é utilizado. Por fim, o componente `exit` demarca o ponto de saída do procedimento, onde valores são repassados aos parâmetros reais na passagem *value and result*.

4.7 Abstração de Tipos

A Engenharia de Software depende da divisão de sistemas complexos em unidades menores, que podem ser gerenciadas mais facilmente. Algumas das unidades mais simples de um programa são os procedimentos e as funções, os quais encapsulam comandos e expressões respectivamente. No entanto, há outras unidades capazes de encapsular funcionalidades mais complexas. O objeto é uma delas.

Assim como registros, objetos são compostos por valores heterogêneos, mas diferem daqueles ao serem equipados com operações. Essas operações geralmente fazem uso dos dados do objeto. Além disso, os objetos geralmente possuem algum controle de visibilidade, de forma a facilitar a implementação de tipos abstratos de dados. Estes são entidades que oferecem determinadas operações, mas têm sua implementação abstraída.

Programas geralmente contêm a criação de diversos objetos similares. Por essa razão, utilizam-se classes para definir o conteúdo dos objetos. Todos os objetos da

mesma classe possuem as mesmas variáveis e operações, chamadas também de atributos e membros respectivamente. Há ainda atributos e membros estáticos. Em contraste com os demais constituintes de um objeto, estes são entidades compartilhadas por todos os objetos da mesma classe.

$$\mathcal{E}[\![\texttt{`new' Id}]\!] = \text{object(lookup(id))}$$

$$\mathcal{D}[\![\texttt{`class' Id `\{' D}_c\ \texttt{D}_o\ \texttt{`\}'}]\!] = \text{bind(Id, class}(\mathcal{D}[\![\texttt{D}_c]\!], \mathcal{D}[\![\texttt{D}_o]\!], \text{static}))$$

Figura 4.34: Exemplo de semântica de classes e objetos.

A Figura 4.34 mostra um exemplo de semântica de instanciação de um objeto e de declaração de uma classe. Nesse exemplo, $\texttt{D}_c$ é a declaração dos membros e atributos estáticos pertencentes à classe, e $\texttt{D}_o$ é a declaração dos membros e atributos pertencentes ao objeto. Durante a criação da classe, o parâmetro *static* determina que a resolução de nomes livres localmente durante a declaração dos atributos do objeto deve ser feita utilizando-se escopo estático. A declaração dos atributos estáticos no entanto sempre usa escopo estático. Esse parâmetro pode ser omitido, caso em que o tipo de escopo utilizado é aquele configurado previamente por meio de `useDynamicScope` ou `useStaticScope` ou escopo estático, caso não haja configuração. Classes nomeadas por meio de *bindings* possuem um campo chamado *classname*, o qual guarda o nome da classe e é utilizado na verificação de tipo de objetos.

A Figura 4.35 mostra um exemplo de semântica de chamada de membros de objeto e membros estáticos. Nesse exemplo, a chamada ocorre dentro de um membro do objeto. Por essa razão, no *environment* corrente estão ligações para os identificadores *this* – associado ao objeto corrente – e *class* – associado à classe do objeto corrente. É importante observar que o corpo do procedimento ou função é executado utilizando o *environment* válido no momento de sua declaração, o qual contém apenas ligações de atributos estáticos e corresponde à representação da classe. Por isso, a semântica estática deve garantir que atributos dinâmicos sejam acessados por meio do primeiro parâmetro da chamada, o qual corresponde ao objeto corrente. A biblioteca também não oferece qualquer tipo de anotação separando os atributos estáticas dos atributos dinâmicos.

Assim como uma classe C descreve um objeto, uma subclasse de C descreve um objeto mais elaborado do que o objeto descrito por C. Essa subclasse pode acrescentar atributos e métodos ao objeto descrito por C e alterar o comportamento de alguns de seus métodos. A Figura 4.36 mostra um exemplo de semântica de subclasse, em que

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{`fun' Id `(' APar `)'}]\!] = \text{call(method, arg)} \\
&\quad \text{where method} = \text{lookup(this, Id)} \\
&\qquad\qquad \text{arg} = [\text{this} \bullet \mathcal{E}[\![\texttt{APar}]\!]] \\
&\qquad\qquad \text{this} = \text{lookup(`this')} \\
&\mathcal{E}[\![\texttt{`static-fun' Id `(' APar `)'}]\!] = \text{call(method, arg)} \\
&\quad \text{where method} = \text{lookup(class, Id)} \\
&\qquad\qquad \text{arg} = [\text{class} \bullet \mathcal{E}[\![\texttt{APar}]\!]] \\
&\qquad\qquad \text{class} = \text{lookup(`class')}
\end{aligned}
$$

Figura 4.35: Exemplo de semântica de chamada de membros.

a classe $\texttt{I}_1$ estende a classe $\texttt{I}_2$. Quando uma subclasse é instanciada, é criada uma ligação entre o identificador *super* e a parte do objeto que herda suas características da classe original, chamada de superclasse.

$$
\begin{aligned}
&\mathcal{D}[\![\texttt{`class' I}_1\ \texttt{E `extends' I}_2]\!] = \text{bind}(\texttt{I}_1, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{subclass}(\text{lookup}(\texttt{I}_2), \mathcal{E}[\![\texttt{E}]\!]) \\
&\qquad\qquad\qquad\qquad\qquad)
\end{aligned}
$$

Figura 4.36: Exemplo de semântica de subclasse.

A biblioteca possibilita a definição de uma semântica de classes com atributos estáticos e dinâmicos e herança. Além de *this* e *class*, cada objeto possui uma referência à instância da superclasse, guardada no campo *super*. Os métodos e atributos da subclasse prevalecem sobre aqueles da superclasse em caso de sobrecarga, mas, por meio do campo *super*, é possível acessar os atributos e métodos da superclasse. Assim, é possível definir outras semânticas de chamada de métodos em hierarquias além de *dynamic dispatch*. Não há componente específico para definição de herança múltipla, mas o comportamento esperado pode ser modelado por meio da composição de *environments* para composição do objeto. Certos conceitos usuais na programação orientada por objetos, tais como sobrecarga e os problemas de colisão de nomes e herança repetida em linguagens com herança múltipla, podem ser resolvidos em uma etapa anterior de análise de semântica estática.

4.8 Sequenciadores

Os comandos condicionais, iterativos e sequenciais apresentados na Seção 4.4 podem ser utilizados para implementar fluxos de controle com uma entrada e uma saída. No entanto, as construções apresentadas neste capítulo permitem implementar uma grande

variedade de fluxos de controle ao remover a restrição de uma entrada e uma saída. Aqui são apresentados: saídas, saltos e exceções. Estes são chamados de sequenciadores, dada sua habilidade de manipular o fluxo de controle.

4.8.1 Saídas

As saídas são construções que permitem ao fluxo de controle escapar de uma construção envolvente. Por exemplo, na linguagem C, o sequenciador *break* permite o término da execução de um laço envolvente. No caso de várias construções aninhadas, há ainda sequenciadores que permitem escapar de uma construção mais externa. A Figura 4.37 mostra um exemplo de semântica de saída normal e de saída de construção mais externa. O exemplo é inspirado na construção *for* da linguagem Java, mas o sequenciador *continue* e a semântica do *for* são omitidos para facilitar a leitura. O sequenciador `break` sem rótulo causa o término da execução da construção `F` mais interna, enquanto o sequenciador `break` rotulado faz com que a execução da construção `F` correspondente seja terminada. Como pode ser visto no exemplo, o componente `jump` é utilizado em conjunto com `escape`. O último define a saída e o primeiro executa a mudança no fluxo de controle.

$\mathcal{C}[\![\texttt{I: F}]\!] = \text{escape}(\mathcal{C}[\![\texttt{F}]\!], \texttt{I})$
$\mathcal{C}[\![\texttt{F}]\!] = \text{escape}(\mathcal{F}[\![\texttt{F}]\!], \text{‘break’})$
$\mathcal{C}[\![\texttt{‘break’}]\!] = \text{jump}(\text{‘break’})$
$\mathcal{C}[\![\texttt{‘break’ I}]\!] = \text{jump}(\texttt{I})$

Figura 4.37: Exemplo de semântica de saída.

Há linguagens que permitem o fim da execução de um programa por meio de um sequenciador como *exit*, *halt* ou *abort*. De forma semelhante a *break* e *continue*, esses sequenciadores podem ser representados por pares de componentes `escape` e `jump`. Entretanto, dada sua utilidade para representar situações de erro na semântica dinâmica de uma linguagem, o término da execução em estado de erro é representado pelo componente `abort`.

É importante observar que a biblioteca de componentes de semântica denotacional não garante que a saída de um sequenciador seja definida. Por isso, fica a cargo da semântica estática estabelecer se o uso de um sequenciador é válido, evitando, por exemplo, que o sequenciador *break* seja utilizado fora de laços.

4.8.2 Saltos

Saltos são construções básicas, as quais estão presentes em outras mais complexas como os laços. A semântica de um salto é definir qual será a próxima instrução a ser executada, o que pode resultar em um salto para trás ou para frente no corpo do programa. Em sua forma mais simples, o salto sempre é realizado, mas também há os chamados saltos condicionais, que só são realizados se uma certa condição for satisfeita. Estes porém são mais bem representados pelo uso do salto e da condição como componentes independentes. Esta seção trata apenas dos componentes relacionados aos saltos.

O uso de saltos em linguagens de programação envolve dois conceitos: a coleta de rótulos e a execução do salto. A função dos rótulos é definir pontos do programa para os quais a execução pode saltar. A coleta dos rótulos deve ser feita antes da execução do programa, por meio dos componentes `addEnv`, `addLabel`, `elabSeq` e `elab`.

O componente `addEnv` recebe como parâmetros a coleta de rótulos e um comando e sua semântica é executar o comando recebido em um *environment* que conhece os rótulos coletados. A coleta em si é realizada pelo componente `addLabel`. Sua semântica é atribuir no *environment* um rótulo a um ponto do programa. Para dar sequência à coleta de rótulos o componente `elabSeq` recebe duas coletas e executa a primeira e então a segunda, como pode ser visto no exemplo da Figura 4.38. Nesse exemplo, veem-se dois outros componentes relacionados à coleta de rótulos. O componente `skipLabel` tem por objetivo indicar que não há rótulo para coletar em uma seção de código, da mesma forma que o componente `skip` indica ausência de computação.

$\mathcal{C}[\![$`‘{’ C ‘}’`$]\!]$ = addEnv($\mathcal{J}[\![$`C`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$)
$\mathcal{C}[\![$`I ‘:’ C`$]\!]$ = addLabel(I, $\mathcal{C}[\![$`C`$]\!]$, static)
$\mathcal{J}[\![$`C`$_1$ `‘;’ C`$_2]\!]$ = elabSeq(elab($\mathcal{J}[\![$`C`$_1]\!]$, $\mathcal{C}[\![$`C`$_2]\!]$), $\mathcal{J}[\![$`C`$_2]\!]$)
$\mathcal{J}[\![$`‘while’ E ‘do’ C`$]\!]$ = elab($\mathcal{J}[\![$`C`$]\!]$, $\mathcal{C}[\![$`‘while’ E ‘do’ C`$]\!]$)
$\mathcal{J}[\![$`‘print’ E`$]\!]$ = skipLabel

Figura 4.38: Exemplo de coleta de rótulos.

O componente `elab` é essencial para a coleta de rótulos. Sua função é modelar o resto do programa num salto. No exemplo, caso seja executado um salto para um rótulo definido em `C`$_1$, primeiro `C`$_1$ será executado, depois será executado `C`$_2$ e então o resto do programa. Por meio do componente `elab`, o resto do programa é conhecido no momento da coleta de rótulos pelo componente `addLabel`. Num salto realizado dentro de um bloco, por exemplo, `C`$_2$ poderia representar o resto do bloco, enquanto o resto do programa é comunicado por meio do contexto ao componente `elab` e então ao

componente `addLabel`.

Por fim, o componente `jump` é responsável por realizar salto. Esse componente procura no *environment* o ponto do programa associado ao rótulo recebido como parâmetro e salta para aquele ponto. Por exemplo, um salto realizado para dentro de um laço definido no exemplo anterior traria a execução para dentro do laço. Os comandos dentro do corpo do laço a partir do ponto definido pelo rótulo seriam então executados. Depois o próprio laçado seria executado novamente caso a condição de controle seja verdadeira, como é a semântica usual do laço e em concordância com o exemplo mostrado.

4.8.3 Exceções

Outro recurso importante relacionado ao fluxo de controle é algo que poderia ser chamado de programação reativa. Presentes também no próprio hardware dos computadores, as exceções têm como objetivo acionar uma parte do programa. Expandindo o conceito, as exceções podem ser vistas como eventos, os quais podem ocorrer sem que isso signifique que haja um erro. Muitos mecanismos como entrada de dados e comunicação em rede são modelados dessa forma, porém nesse caso o nome utilizado é interrupção, fazendo alusão ao fato de que o programa em execução é interrompido e o controle é dado momentaneamente a outra seção de código.

Os eventos ou exceções são modelados por meio de dois componentes: `throw` e `catch`. O primeiro deles é responsável por sinalizar o evento, enquanto o segundo detecta a sua ocorrência e assume o controle. O componente `catch` possui dois parâmetros. O primeiro deles é uma seção de código a ser executada. Caso uma exceção seja lançada durante essa execução, esta é interrompida, e o segundo parâmetro recebe o valor passado pela exceção e é executado. O exemplo da Figura 4.39 mostra uma semântica de tratamento de exceções.

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{`try' C P}]\!] = \text{catch}(\mathcal{C}[\![\texttt{C}]\!], \mathcal{C}[\![\texttt{P}]\!]) \\
&\mathcal{C}[\![\texttt{`catch' } \texttt{I}_1 \texttt{ I}_2 \texttt{ C}]\!] = \text{parametrize}((\text{value}, \texttt{I}_2), \\
&\qquad\qquad \text{choose}(\text{checkType}(\text{getType}(\texttt{I}_2), \texttt{I}_1), \\
&\qquad\qquad\qquad \mathcal{C}[\![\texttt{C}]\!], \\
&\qquad\qquad\qquad \text{throw}(\mathcal{E}[\![\texttt{I}_2]\!])) \\
&\qquad\qquad) \\
&\mathcal{C}[\![\texttt{`throw' E}]\!] = \text{throw}(\mathcal{E}[\![\texttt{E}]\!])
\end{aligned}
$$

Figura 4.39: Exemplo de semântica de tratamento de exceções.

4.8.4 Múltiplas Entradas e Múltiplas Saídas

Por fim, temos as construções como *switch*, as quais possuem múltiplas entradas e múltiplas saídas. Com esse tipo de construção é possível condensar vários testes condicionais. O *switch* escolhe a entrada identificada pelo valor resultante da avaliação de uma expressão. No entanto, a forma como o controle flui a partir da entrada varia de acordo com a linguagem de programação. Em C, por exemplo, o fluxo segue até o fim do *switch* ou até um sequenciador *break*. No entanto, outras linguagens executam apenas o código identificado pelo resultado da expressão. Há ainda uma opção de entrada utilizada no caso de todas as outras falharem.

$\mathcal{C}[\![$`‘switch’ E CaseExps ‘default’ C`$]\!] =$
  escape(
    trap(E, $\mathcal{SC}[\![$`CaseExps`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$),
    ‘break’
)
$\mathcal{SC}[\![$`‘case’ E C CaseExps`$]\!] =$
  $[(\mathcal{E}[\![$`E`$]\!]$, seq($\mathcal{C}[\![$`C`$]\!]$, $\mathcal{C}[\![$`CaseExps`$]\!]$)) $\bullet$ $\mathcal{SC}[\![$`CaseExps`$]\!]]$
$\mathcal{SC}[\![$`‘case’ E C`$]\!] = (\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$]\!])$
$\mathcal{C}[\![$`‘case’ E C CaseExps`$]\!] =$ seq($\mathcal{C}[\![$`C`$]\!]$, $\mathcal{C}[\![$`CaseExps`$]\!]$)
$\mathcal{C}[\![$`‘case’ E C`$]\!] = \mathcal{C}[\![$`C`$]\!]$

Figura 4.40: Exemplo de semântica de um comando *switch* na linguagem C.

A Figura 4.40 mostra um exemplo de semântica desse tipo de fluxo de controle utilizando os componentes `trap`, `escape` e `skip`. Nesse exemplo, a expressão de cada entrada do *switch* é avaliada e seu valor associado à posição no código em que se encontra. Posteriormente a expressão principal do *switch* é avaliada e seu valor é comparado com os valores das entradas. Caso nenhuma das entradas possua o valor correto, o terceiro parâmetro de `trap` é executado.

4.9 Conclusões

Este capítulo apresentou os componentes de definição semântica denotacional e o modelo semântico que implementam a biblioteca proposta no capítulo anterior. A implementação da biblioteca é apresentada no Apêndice C e um glossário pode ser visto no Apêndice D. A biblioteca é constituída por domínios semânticos e componentes que representam os mecanismos essenciais do paradigma imperativo. A definição semântica

completa de uma linguagem pode ser vista no Apêndice A e uma definição semântica parcial da linguagem Java pode ser vista no Apêndice B.

O capítulo seguinte apresenta uma avaliação da solução proposta por este trabalho. São abordados um estudo da escalabilidade da técnica apresentada, uma análise qualitativa do trabalho e uma comparação com trabalhos relacionados.

Capítulo 5

Demonstração da Escalabilidade

Seção 5.1 apresenta um estudo de caso que ilustra as funcionalidades da abordagem de estruturação de definições semânticas denotacionais apresentada. Seção 5.2 apresenta uma avaliação da metodologia. Seção 5.3 compara a metodologia proposta com o estado da arte, de modo a apresentar algumas das inspirações para este trabalho e posicioná-lo em relação a outras abordagens.

5.1 Escalabilidade

Nesta seção é apresentado um simples estudo de caso para demonstrar a escalabilidade da semântica denotacional baseada em componentes. O estudo envolve a definição semântica da linguagem Small, extraída de Gordon (1979). São apresentadas cinco versões da linguagem, Small0, Small1, Small 2, Small3 e Small4. São apresentadas definições utilizando semântica denotacional clássica e abordagem baseada em componentes. O impacto de cada extensão nas duas definições é avaliado. A definição semântica denotacional baseada em componentes de Small pode ser vista no Apêndice A.

Os componentes de semântica denotacional são utilizados para definir a semântica das linguagens da família Small, comparando essa abordagem de componentes com a semântica denotacional padrão. É apresentada também a extensão da biblioteca para adicionar um novo conceito, ignorado na biblioteca deliberadamente para ilustrar a flexibilidade da abordagem proposta.

5.1.1 Semântica Denotacional de Small0

Small0 é uma linguagem simples com expressões, declarações e comandos. A gramática a seguir ilustra a sintaxe abstrata dessa linguagem:

P	→	program C
D	→	var I = E |D_1;D_2
C	→	E_1 := E_2 |output E |if E then C_1 else C_2 |while E do C
	|	begin D;C end |C_1;C_2
E	→	N |true |false |read |I |E_1 O E_2
O	→	+ |- |× |÷ |= |!=

A semântica denotacional de Small0 é apresentada de forma simplificada na Figura 5.2, e os domínios semânticos e as assinaturas das equações são apresentados na Figura 5.1. Essa definição semântica utiliza como informações de contexto: *environment*, *store* e continuação. Esse contexto está presente em cada uma das funções de mapeamento semântico.

Domínios Semânticos:

Dv	= Loc + Rv	-valores denotáveis **d**
Sv	= File + Rv	-valores armazenáveis **v**
Ev	= Dv	-valores expressáveis **e**
Rv	= Bool + Bv	-*R-values* **e**
Bv	= Num + Bool	-valores básicos **e**
File	= Rv^*	-arquivos **i**
Env	=Id →[Dv + {unbound}]	-*environments* **r**
Store	= Loc →[Sv + {unused}]	-*stores* **s**
Cc	= Store →Ans	-continuações de comando **c**
Ec	= Ev →Cc	-continuações de expressão **k**
Dc	= Env →Cc	-continuações de declaração **u**
Ans	= {error,stop} × File	-respostas finais **a**

Equações Semânticas:

Valores Básicos:	$\mathcal{B}$	= N →Bv
Operações Aritméticas	$\mathcal{O}$	= O →Bv →Bv →Bv
Programas	$\mathcal{P}$	= P →File →Ans
Expressões de *R-values*	$\mathcal{R}$	= E →Env →Ec →Cc
Expressões:	$\mathcal{E}$	= E →Env →Ec →Cc
Comandos:	$\mathcal{C}$	= C →Env →Cc →Cc
Declarações:	$\mathcal{D}$	= D →Env →Dc →Cc

Figura 5.1: Domínios Semânticos de Small0 - baseado em Gordon (1979).

Programas:
$\mathcal{P}[\![\texttt{program C}]\!]$ i = $\mathcal{C}[\![\texttt{C}]\!]$ r c s[i/'input'][[]/'output']
where r = λi.unbound
c = λs.(stop, s 'output')
s = λl.unused

Declarações:
$\mathcal{D}[\![\texttt{var I = E}]\!]$ r u = $\mathcal{R}[\![\texttt{E}]\!]$ r;ref λl.u [l/I]
$\mathcal{D}[\![\texttt{D}_1\texttt{;D}_2]\!]$ r u = $\mathcal{D}[\![\texttt{D}_1]\!]$ r;λr$_1$.$\mathcal{D}[\![\texttt{D}_2]\!]$ r[r$_1$];λr$_2$.u (r$_1$[r$_2$])

Expressões:
$\mathcal{E}[\![\texttt{true}]\!]$ r k = k true
$\mathcal{E}[\![\texttt{false}]\!]$ r k = k false
$\mathcal{E}[\![\texttt{N}]\!]$ r k = k $\mathcal{B}[\![\texttt{N}]\!]$
$\mathcal{E}[\![\texttt{I}]\!]$ r k = (r I = unbound) $\rightarrow$err, k (r I)
$\mathcal{E}[\![\texttt{read}]\!]$ r k s = null (s 'input') $\rightarrow$err s,
k (hd (s 'input')) s[tl (s 'input')/'input']
$\mathcal{E}[\![\texttt{E}_1\ \texttt{O}\ \texttt{E}_2]\!]$ r k = $\mathcal{R}[\![\texttt{E}_1]\!]$ r;λe$_1$.$\mathcal{E}[\![\texttt{E}_2]\!]$ r;λe$_2$.k ($\mathcal{O}[\![\texttt{O}]\!]$ e$_1$ e$_2$)

Comandos:
$\mathcal{C}[\![\texttt{E}_1\ \texttt{:=}\ \texttt{E}_2]\!]$ r c = $\mathcal{E}[\![\texttt{E}_1]\!]$ r;Loc?λl.$\mathcal{R}[\![\texttt{E}_2]\!]$ r;update l;c
$\mathcal{C}[\![\texttt{output E}]\!]$ r c = $\mathcal{R}[\![\texttt{E}]\!]$ r; λe s.c s[[e • s'output']/'output']
$\mathcal{C}[\![\texttt{if E then C}_1\ \texttt{else C}_2]\!]$ r c =
$\mathcal{R}[\![\texttt{E}]\!]$ r;Bool?;cond($\mathcal{C}[\![\texttt{C}_1]\!]$ r c, $\mathcal{C}[\![\texttt{C}_2]\!]$ r c)
$\mathcal{C}[\![\texttt{while E do C}]\!]$ = fix λf r c.$\mathcal{R}[\![\texttt{E}]\!]$ r;Bool?;cond($\mathcal{C}[\![\texttt{C}]\!]$ r;f r c, c)
$\mathcal{C}[\![\texttt{C}_1\texttt{;C}_2]\!]$ r c = $\mathcal{C}[\![\texttt{C}_1]\!]$ r;$\mathcal{C}[\![\texttt{C}_2]\!]$ r c
$\mathcal{C}[\![\texttt{begin D;C end}]\!]$ r c = $\mathcal{D}[\![\texttt{D}]\!]$ r;λr'.$\mathcal{C}[\![\texttt{C}]\!]$ r[r'] c

Figura 5.2: Semântica denotacional clássica de Small0 - extraído de Gordon (1979).

A semântica baseada em componentes de Small0 é apresentada na Figura 5.3. Nessa definição, as informações de contexto não se fazem presentes nas equações semânticas, com exceção da equação semântica de programas. Com a exceção de possível extensões dos componentes, esta é a única equação semântica em que o contexto deve ser explicitado. A semântica de programas define o contexto em que estes são executados e utiliza o *store* para determinar o resultado da execução. O mesmo pode ser visto na definição denotacional clássica.

A abordagem baseada em componentes explora o encapsulamento de conceitos fundamentais em componentes. Dessa forma, uma vez que o leitor tenha conhecimento da semântica dos componentes, a leitura da definição se torna mais direta. A remoção da dependência do contexto na redação das equações semânticas e o uso de componentes conferem maior destaque aos conceitos modelados e à forma como eles se combinam.

Programas:
$\mathcal{P}[\![\texttt{program C}]\!]$ i = result(run($\mathcal{C}[\![\texttt{C}]\!]$) i r_0 s_0)
where result (t,e,r,s) = (t,s 'output')

Declarações:
$\mathcal{D}[\![\texttt{var I = E}]\!]$ = bind(I, ref(deref($\mathcal{E}[\![\texttt{E}]\!]$)))
$\mathcal{D}[\![\texttt{D}_1\texttt{;D}_2]\!]$ = elabSeq($\mathcal{D}[\![\texttt{D}_1]\!]$, $\mathcal{D}[\![\texttt{D}_2]\!]$)

Expressões:
$\mathcal{E}[\![\texttt{true}]\!]$ = bool('true')
$\mathcal{E}[\![\texttt{false}]\!]$ = bool('false')
$\mathcal{E}[\![\texttt{N}]\!]$ = int(N)
$\mathcal{E}[\![\texttt{I}]\!]$ = lookup(I)
$\mathcal{E}[\![\texttt{read}]\!]$ = input
$\mathcal{E}[\![\texttt{E}_1\ \texttt{O}\ \texttt{E}_2]\!]$ = apply(O,$\mathcal{E}[\![\texttt{E}_1]\!]$,$\mathcal{E}[\![\texttt{E}_2]\!]$)

Comandos:
$\mathcal{C}[\![\texttt{E}_1\ \texttt{:=}\ \texttt{E}_2]\!]$ = assign($\mathcal{E}[\![\texttt{E}_1]\!]$, $\mathcal{E}[\![\texttt{E}_2]\!]$)
$\mathcal{C}[\![\texttt{output E}]\!]$ = output($\mathcal{E}[\![\texttt{E}]\!]$)
$\mathcal{C}[\![\texttt{if E then C}_1\ \texttt{else C}_2]\!]$ = choose($\mathcal{E}[\![\texttt{E}]\!]$, $\mathcal{C}[\![\texttt{C}_1]\!]$, $\mathcal{C}[\![\texttt{C}_2]\!]$)
$\mathcal{C}[\![\texttt{while E do C}]\!]$ = loop($\mathcal{E}[\![\texttt{E}]\!]$,$\mathcal{C}[\![\texttt{C}]\!]$)
$\mathcal{C}[\![\texttt{C}_1\texttt{;C}_2]\!]$ = seq($\mathcal{C}[\![\texttt{C}_1]\!]$,$\mathcal{C}[\![\texttt{C}_2]\!]$)
$\mathcal{C}[\![\texttt{begin D;C end}]\!]$ = makeClosure($\mathcal{D}[\![\texttt{D}]\!]$, $\mathcal{C}[\![\texttt{C}]\!]$)

Figura 5.3: Semântica baseada em componentes de Small0.

Por fim, a nomenclatura utilizada expõe a semântica dos componentes, favorecendo a leitura mesmo sem treinamento prévio do leitor.

5.1.2 Semântica Denotacional de Small1

A primeira extensão apresentada está relacionada à manipulação do fluxo de controle. São adicionados os sequenciadores *break* e *continue*. O primeiro termina a execução de um laço, e o segundo termina a execução de uma iteração de um laço.

Para que os sequenciadores sejam adicionados utilizando semântica denotacional clássica, é necessário que laço informe a seus constituintes o ponto do programa em que a execução deve continuar após um *break* ou *continue*. Uma forma de fazer essa comunicação é armazenar as continuações associadas a cada uma dessas saídas no *environment*. Para isso as continuações devem ser valores denotáveis. Além disso, é necessário alterar a equação que define a semântica do laço para informar as continuações aos constituintes. A Figura 5.4 mostra as alterações das equações semânticas.

$$
\begin{array}{l}
\text{Dv} = \text{Loc} + \text{Rv} + \text{Cc} \text{ -valores denotáveis } \mathbf{d} \\
\\
\mathcal{C}[\![\texttt{while E do C}]\!]\ \text{r c} = \text{fix } \lambda\text{f.}\mathcal{R}[\![\texttt{E}]\!]\ \text{r;Bool?;cond}(\mathcal{C}[\![\texttt{C}]\!]\ \text{r[r}']\text{;f r c, c)} \\
\quad \text{where r}' = [\text{`break'/c}][\text{`continue'/f r c}] \\
\mathcal{C}[\![\texttt{break}]\!]\ \text{r c} = (\text{r `break'}) \\
\mathcal{C}[\![\texttt{continue}]\!]\ \text{r c} = (\text{r `continue'})
\end{array}
$$

Figura 5.4: Semântica denotacional clássica de sequenciadores de Small1.

Por outro lado, a abordagem de componentes modela diretamente sequenciadores, visto que estes são conceitos fundamentais das linguagens de programação. Dessa forma, além das equações que definem a semântica dos sequenciadores, é necessário apenas alterar a equação semântica de laços. Esta alteração é semelhante àquela feita na definição denotacional clássica. No entanto, a equação é organizada em função dos conceitos e de sua combinação. O leitor não precisa conhecer a implementação dos sequenciadores, e o modelo semântica que suporta as equações não é alterado.

$$
\begin{array}{l}
\mathcal{C}[\![\texttt{while E do C}]\!] = \text{escape}(\text{loop}(\mathcal{E}[\![\texttt{E}]\!], \\
\qquad\qquad\qquad\qquad \text{escape}(\mathcal{C}[\![\texttt{C}]\!],\text{`continue'})), \\
\qquad\qquad\qquad \text{`break'}) \\
\mathcal{C}[\![\texttt{break}]\!] = \text{jump}(\text{`break'}) \\
\mathcal{C}[\![\texttt{continue}]\!] = \text{jump}(\text{`continue'})
\end{array}
$$

Figura 5.5: Semântica baseada em componentes de sequenciadores de Small1.

Como pode ser visto nas equações apresentadas, a abordagem baseada em componentes não requer alterações dos domínios semânticos, além de resultar em equações mais legíveis. Isso ocorre porque os sequenciadores são conceitos recorrentes das linguagens de programação imperativas e, por isso, são previstos pela biblioteca. No entanto, os componentes são genéricos o suficiente para expressar funcionalidades menos comuns. Por exemplo, o *break* com múltiplos níveis de Java também poderia ser modelado pelos mesmos componentes `jump` e `escape`. Bastaria, por meio do componente `escape`, definir saídas identificadas pelos rótulos dos níveis e passar o rótulo do nível desejado como parâmetro do componente `jump`.

A Figura 5.6 mostra um programa Small1 e sua denotação representada por componentes. Nesse exemplo, o sequenciador *break* é denotado pelo componente `jump`. Os pontos de saída para os sequenciadores *break* e *continue* são representados pelo componente `escape`.

```
program                          run(makeClosure(
begin                              bind('x', int('2')),
  var x = 2;                       escape(loop(escape(
  while x > 0 do                     apply('>', lookup('x'), int('0'))),
    x = x * x;                       seq(
    if x > 100 then                    assign(
      break                                lookup('x'),
    else                                   apply('*', lookup('x'), lookup('x'))),
      output x;                        choose(apply('>', lookup('x'), int('100')))
end                                        jump('break'),
                                           output(lookup('x'))
                                       )
                                     ), 'continue)'
                                   ), 'break')
                                 ))
```

Figura 5.6: Exemplo de programa de Small1 e sua denotação.

5.1.3 Semântica Denotacional de Small2

A segunda extensão adiciona saltos à semântica de Small1. Para isso, é necessário coletar os rótulos do programa e definir a semântica do salto. Utilizando semântica denotacional clássica, é necessário criar uma função coletora de rótulos e utilizá-la ao se entrar em um bloco. A forma como os rótulos são recolhidos determina a semântica do salto. Por exemplo, caso os dois comandos de uma construção *if-then-else* possuam o mesmo rótulo, a coleta realizada no segundo comando prevalece. Por fim, a coleta de rótulos em um bloco exige avaliação recursiva, o que torna a equação semântica mais complexa.

As alterações feitas utilizando a abordagem de componentes são semelhantes àquelas apresentadas anteriormente. No entanto, as várias operações de coleta de rótulos e o próprio salto são encapsulados em componentes. Além da adição de novas equações, a equação semântica de blocos precisa ser alterada. No entanto, a alteração é simples e semelhante à execução de declarações. De fato, o componente `addEnv` pode ser utilizado para adicionar declarações ou rótulos ao contexto de um comando.

O salto de Small2 é um conceito fundamental das linguagens de programação. Portanto, está previsto na biblioteca de componentes. Por isso, as equações semânticas baseadas em componentes são mais legíveis, identificando explicitamente os conceitos abordados.

A Figura 5.9 mostra um programa Small2 e sua denotação. Nesse exemplo, o componente `addLabel` faz a coleta dos rótulos `L1` e `L2`. O componente `elab` é utilizado

Coleta de Rótulos:
$\mathcal{J}$ = C →Env →Cc →Env

$\mathcal{J}[\![\texttt{I: C}]\!]$ r c = $\mathcal{J}[\![\texttt{C}]\!]$ r c[$\mathcal{C}[\![\texttt{C}]\!]$ r c/I
$\mathcal{J}[\![\texttt{while E do C}]\!]$ = fix λf r c.$\mathcal{J}[\![\texttt{C}_1]\!]$ r (f r c)
$\mathcal{J}[\![\texttt{if E then C}_1\texttt{ else C}_2]\!]$ r c = $\mathcal{J}[\![\texttt{C}_1]\!]$ r c[$\mathcal{J}[\![\texttt{C}_2]\!]$ r c]
$\mathcal{J}[\![\texttt{C}_1\texttt{;C}_2]\!]$ r c = $\mathcal{J}[\![\texttt{C}_1]\!]$ r ($\mathcal{C}[\![\texttt{C}_2]\!]$ r c)[$\mathcal{J}[\![\texttt{C}_2]\!]$ r c]
$\mathcal{J}[\![\texttt{begin D;C end}]\!]$ r c = []
$\mathcal{J}[\![\texttt{goto I}]\!]$ r c = []
$\mathcal{J}[\![\texttt{E}_1\texttt{ := E}_2]\!]$ r c = []
$\mathcal{J}[\![\texttt{output E}]\!]$ r c = []
$\mathcal{J}[\![\texttt{break}]\!]$ r c = []
$\mathcal{J}[\![\texttt{continue}]\!]$ r c = []

$\mathcal{C}[\![\texttt{begin D;C end}]\!]$ r c = $\mathcal{D}[\![\texttt{D}]\!]$ r;λr′.$\mathcal{C}[\![\texttt{C}]\!]$ r[r′][r″] c
  where r″= $\mathcal{J}[\![\texttt{C}]\!]$ r[r′][r″] c
Salto:
$\mathcal{J}[\![\texttt{I: C}]\!]$ r c = $\mathcal{C}[\![\texttt{C}]\!]$ r c
$\mathcal{C}[\![\texttt{goto I}]\!]$ r c = (r I)

Figura 5.7: Semântica denotacional clássica de saltos de Small2.

Coleta de Rótulos:
$\mathcal{J}[\![\texttt{I: C}]\!]$ = addLabel(I, $\mathcal{C}[\![\texttt{C}]\!]$)
$\mathcal{J}[\![\texttt{while E do C}]\!]$ = elab($\mathcal{J}[\![\texttt{C}]\!]$,$\mathcal{C}[\![\texttt{while E do C}]\!]$)
$\mathcal{J}[\![\texttt{if E then C}_1\texttt{ else C}_2]\!]$ = elabSeq($\mathcal{J}[\![\texttt{C}_1]\!]$, $\mathcal{J}[\![\texttt{C}_2]\!]$)
$\mathcal{J}[\![\texttt{C}_1\texttt{;C}_2]\!]$ = elabSeq(elab($\mathcal{J}[\![\texttt{C}_1]\!]$, $\mathcal{C}[\![\texttt{C}_2]\!]$), $\mathcal{J}[\![\texttt{C}_2]\!]$)
$\mathcal{J}[\![\texttt{begin D;C end}]\!]$ = skipLabel
$\mathcal{J}[\![\texttt{goto I}]\!]$ = skipLabel
$\mathcal{J}[\![\texttt{E}_1\texttt{ := E}_2]\!]$ = skipLabel
$\mathcal{J}[\![\texttt{output E}]\!]$ = skipLabel
$\mathcal{J}[\![\texttt{break}]\!]$ = skipLabel
$\mathcal{J}[\![\texttt{continue}]\!]$ = skipLabel

$\mathcal{C}[\![\texttt{begin D;C end}]\!]$ = addEnv($\mathcal{J}[\![\texttt{C}]\!]$, makeClosure($\mathcal{D}[\![\texttt{D}]\!]$,$\mathcal{C}[\![\texttt{C}]\!]$))
Salto:
$\mathcal{C}[\![\texttt{I:C}]\!]$ = $\mathcal{C}[\![\texttt{C}]\!]$
$\mathcal{C}[\![\texttt{goto I}]\!]$ = jump(I)

Figura 5.8: Semântica baseada em componentes de saltos de Small2.

para passar o resto do programa a `addLabel`, de modo que o controle possa fluir após o salto. Note que o segundo parâmetro de `elab` é sempre o segundo comando após o

```
program begin
  var x = 1;
L1:
   x = x + 1;
  if x < 10 then
       jump L1
   else
       jump L2
  output -1
L2:
  output x
end
```

```
run(addEnv(
   elabSeq(elab(
       addLabel(
         'L1',
         assign(lookup('x'),
            apply('+', lookup('x'), int('1')))
       ),
       f
     ),
     elabSeq(
       elab(skipLabel, output('x')),
       addLabel('L2', output('x'))
     )
   ),
   makeClosure(
     bind('x', int('1')),
     seq(
       assign(lookup('x'), apply('*', lookup('x'), int('1'))),
       f
     )
   )
))
    where f =seq(
                choose(
                        apply('<', lookup('x'), int('10')),
                        jump('L1'),
                        jump('L2')
                ),
                output('x')
             )
```

Figura 5.9: Exemplo de programa de Small2 e sua denotação.

rótulo. Caso haja mais comandos além deste, o componente `elab` passa a informação ao componente `addLabel` por meio do contexto.

5.1.4 Semântica Denotacional de Small3

Esta seção adiciona procedimentos à linguagem Small2. Para isso, devem ser adicionadas equações que descrevem a semântica da declaração e da chamada dos procedimentos. Além disso, a chamada de procedimentos é responsável por criar uma nova área de memória para as variáveis locais aos procedimentos. Ao fim da chamada, essa área de memória deve ser liberada.

Alterações nos Store:
Store = [{level,new} + Loc] →[Sv + {unused}] - *stores* **s**
Loc = address + {input,output} - *locations* **l**

$\mathcal{P}[\![\texttt{program C}]\!]$ i = $\mathcal{C}[\![\texttt{C}]\!]$ r c s[i/'input'][[]/'output']
where r = λi.unbound
c = λs.(stop, s 'output')
s = s_0 = λl.l = level →1, l = new →0, unused

Chamada de Procedimentos:
$\mathcal{C}[\![\texttt{E}_1\texttt{(L)}]\!]$ r k s = $\mathcal{E}[\![\texttt{E}_1]\!]$ r;λp.$\mathcal{L}[\![\texttt{L}]\!]$ r p k′s′
where k′ = λe s.k e (λl.l = level →(s level) - 1,
l = ((s level), _) →unused, s l)
s′ = (λl.l = level →(s level) + 1, s l)

$\mathcal{L}[\![\texttt{E, L}]\!]$ r p c = $\mathcal{E}[\![\texttt{E}]\!]$ r;λe.$\mathcal{L}[\![\texttt{L}]\!]$ r (p e) c
$\mathcal{L}[\![\texttt{E}]\!]$ r p c = $\mathcal{E}[\![\texttt{E}]\!]$ r;λe.p e c

$\mathcal{C}[\![\texttt{return}]\!]$ r c = (r 'return')
Declarações:
$\mathcal{D}[\![\texttt{I(A);C}]\!]$ r u = u [($\mathcal{A}[\![\texttt{A}]\!]$ ($\mathcal{C}[\![\texttt{C}]\!]$)) r/I

$\mathcal{A}[\![\texttt{I, A}]\!]$ r x = λc.deref;ref λe.$\mathcal{A}[\![\texttt{A}]\!]$ r[e/$\texttt{I}_1$] x c
$\mathcal{A}[\![\texttt{I}]\!]$ r x = λc.deref;ref λe.x r[e/$\texttt{I}_1$][c/'return'] c

Figura 5.10: Semântica denotacional clássica de procedimentos de Small3.

A Figura 5.10 mostra a semântica denotacional clássica de procedimentos em Small3. São adicionadas complexas equações descrevendo a semântica de chamada e declaração de procedimentos. Como os procedimentos recebem múltiplos parâmetros – pelo menos um –, é necessário fazer a leitura de equações recursivas que lidam com listas de parâmetros formais e reais. No entanto, a parte mais complexa dessa extensão é a gerência de memória. Para tornar possível o empilhamento de seções de memória no momento da chamada, é necessário alterar o *store*. São então alterados o domínio de *stores* e a definição do *store* inicial na semântica de programas. Além disso, a complexa manipulação do *store* é explicitada nas equações de semântica de chamada de procedimentos.

Uma vez que procedimentos e gerência de memória são conceitos fundamentais das linguagens de programação, podem-se utilizar componentes de semântica denotacional na extensão, como mostra a Figura 5.11. Procedimentos são declarados por meio do componente `bind`, parametrizados por meio de `parametrize` e chamados por meio de `call`. A gerência de memória tem sua semântica modelada pelos componentes `push` – que cria uma nova área de memória – e `pop` – que libera a nova área de memória.

$\mathcal{D}[\![\texttt{I(A);C}]\!]$ = bind(I, proc(parametrize($\mathcal{A}[\![\texttt{A}]\!]$, escape($\mathcal{C}[\![\texttt{C}]\!]$, 'return'))))

$\mathcal{A}[\![\texttt{I}]\!]$ = (value, I)
$\mathcal{A}[\![\texttt{I, A}]\!]$ = [$\mathcal{A}[\![\texttt{I}]\!]$ • $\mathcal{A}[\![\texttt{A}]\!]$]

$\mathcal{C}[\![\texttt{return}]\!]$ = jump('return')
$\mathcal{C}[\![\texttt{E}_1\texttt{(L)}]\!]$ = createMemoryBlock(call($\mathcal{E}[\![\texttt{E}_1]\!]$,$\mathcal{E}[\![\texttt{L}]\!]$))
where createMemoryBlock p = seq(push, seq(p, pop))

$\mathcal{E}[\![\texttt{E, L}]\!]$ = [$\mathcal{E}[\![\texttt{E}]\!]$ • $\mathcal{E}[\![\texttt{L}]\!]$]

Figura 5.11: Semântica denotacional baseada em componentes de procedimentos de Small3.

Gerência de memória e procedimentos são funcionalidades complexas, embora fundamentais. Por isso, a definição baseada em componentes é mais legível que a definição denotacional clássica. Note que o impacto da gerência de memória sobre a definição baseada em componentes é mais localizada, afetando apenas a equação semântica da chamada de procedimentos. E, graças à inclusão do mecanismo de gerência de memória na biblioteca, a versão baseada em componentes dessa equação é mais simples do que a versão denotacional clássica.

```
fat (x) ;
  if x > 1 then
    fat (x - 1);
    y = x * y
  else
    output y
```

```
bind('fat',proc(parametrize((value, 'x'),
   choose( apply('>'), lookup('x'), int('1')
     seq(
       assign(lookup('y'),
          apply('*', lookup('x'), lookup('y'))),
       seq(push, call(lookup('fat'),
          [apply('-', lookup('x'), int('1')]), pop)
     ),
     output('y')
   )
)))
```

Figura 5.12: Exemplo de trecho de um programa de Small3 e sua denotação.

A Figura 5.12 mostra um trecho de um programa Small3 e sua denotação. Neste exemplo, o componente `parametrize` é utilizado para parametrizar o procedimento *fat*, com o parâmetro *x*. A chamada `fat(x-1)` é realizada por meio do componente `call`. Os componentes `push` e `pop` são utilizados para criar a área de memória do procedimento e liberá-la respectivamente.

5.1.5 Semântica Denotacional de Small4

A última extensão adiciona exceções à linguagem Small3. A construção *try-catch-finally* é responsável por executar um comando e, caso uma exceção seja lançada, tratá-la executando um comando específico. Por fim, o comando do bloco *finally* é executado. No entanto, é importante observar que este comando deve sempre ser executado. Consequentemente, caso haja um retorno do procedimento durante a execução da construção, é necessário executar o bloco *finally* antes de deixar o procedimento.

Utilizando semântica denotacional clássica, é necessário definir os pontos de saída no *environment*. Consequentemente é necessário alterar as equações que definem declaração e chamada de procedimentos para fazer as alterações necessárias no *environment*. Além disso, é necessário definir a semântica das construções *try-catch-finally* e *throw*. Como pode ser visto na Figura 5.13, as equações são complexas e envolvem muita manipulação de contexto.

Exceções:

$\mathcal{C}[\![\texttt{try C}_1\ \texttt{catch I C}_2\ \texttt{finally C}_3]\!]$ r c = $\mathcal{C}[\![\texttt{C}_1]\!]$ r′;$\mathcal{C}[\![\texttt{C}_3]\!]$ r c
  where r′ = r[k_R,k_T/'return','throw']
  k_R = $\mathcal{C}[\![\texttt{C}_3]\!]$ r;(r 'return')
  k_T = ref;λl.$\mathcal{C}[\![\texttt{C}_2]\!]$ r″; $\mathcal{C}[\![\texttt{C}_3]\!]$ r c
  r″ = r[k_R,k'_T,l/'return','throw',I]
  k'_T = λv.$\mathcal{C}[\![\texttt{C}_3]\!]$ r;(r 'throw') v
$\mathcal{C}[\![\texttt{throw E}]\!]$ r c = $\mathcal{E}[\![\texttt{E}]\!]$ r;(r 'throw')

Procedimentos:

$\mathcal{C}[\![\texttt{return}]\!]$ r c = $\mathcal{E}[\![\texttt{E}]\!]$ r;(r 'return')
$\mathcal{C}[\![\texttt{E}_1\texttt{(L)}]\!]$ r k s = $\mathcal{E}[\![\texttt{E}_1]\!]$ r;λp.$\mathcal{L}[\![\texttt{L}]\!]$ r p k′(r 'throw')s′
  where k′ = λe s.k e (λl.l = level →(s level) - 1,
    l = ((s level), _) →unused, s l)
  s′ = (λl.l = level →(s level) + 1, s l)

$\mathcal{D}[\![\texttt{I(A);C}]\!]$ r u = u [p/I
  where p = λk_R k_T.ref;($\mathcal{A}[\![\texttt{A}]\!]$ $\mathcal{C}[\![\texttt{C}]\!]$) r′;$k_R$ undefined
  r′ = r[p,k_R,k_T/I,'return','break']

Figura 5.13: Semântica denotacional de exceções de Small4 - extraído de Tirelo et al. (2009).

Como exceções são conceitos fundamentais de linguagens de programação, sua semântica pode ser definida por meio dos componentes `catch` e `throw`. O código para tratamento da exceção é parametrizado por meio do componente `parametrize`. Note que, como mostra a Figura 5.14, não é necessário alterar equações previamente definidas. A interceptação do retorno de procedimento é feita utilizando-se uma nova

definição do componente `trap` criada pelo usuário da biblioteca. Esta definição permite executar um comando com várias saídas possíveis.

Exceções:
$\mathcal{C}[\![\texttt{try C}_1\ \texttt{catch I C}_2\ \texttt{finally C}_3]\!]$ = trap(catch(try, exception),
('return', return))
where try = seq(C_1,C_3)
exception = parametrize((value,I), C_2)
return = seq(C_3, jump('return'))
$\mathcal{C}[\![\texttt{throw E}]\!]$ = throw($\mathcal{E}[\![\texttt{E}]\!]$)

Novo Componente:
trap : (Cd, (Id, Cd)*) →Cd
trap (C, L) r q = C r[traps L] q
where traps L = (null L) →[], (hd L) = (i,c) →[c′/i][traps (tl L)]
c′ = λe r′.c r′q

Figura 5.14: Semântica baseada em componentes de exceções de Small4.

A interceptação de desvios do fluxo de controle é uma operação complexa, como pode ser visto na Figura 5.13. No entanto, a Figura 5.14 mostra que esse conceito pode ser facilmente encapsulado em um componente. Essa funcionalidade foi deliberadamente ignorada na biblioteca de componentes para permitir a construção deste exemplo. O Exemplo mostra que conceitos relevantes das linguagens de programação não previstos podem ser encapsulados estendendo a biblioteca apresentado neste trabalho. Embora essa extensão exija conhecimentos avançados, o usuário possui mecanismos para realizá-la.

A Figura 5.15 mostra um programa de Small4 e sua denotação. Neste exemplo, o componente `catch` é utilizado para modelar o tratamento de exceções lançadas por meio do componente `throw`. O componente `trap` é utilizado para interceptar um possível retorno de função, garantindo que o código *finally*, representado por C_3 é sempre executado.

Como pode ser observado nas extensões apresentadas, o encapsulamento de conceitos fundamentais das linguagens de programação em componentes favorece a escalabilidade das definições semânticas. Embora o número de equações cresça à medida em que novos conceitos são adicionados, a definição permanece legível, o que se deve à abstração do contexto. Além disso, a definição denotacional clássica sofre maior impacto com as extensões do que aquela que utiliza componentes. Isso se deve ao fato de que a semântica baseada em componentes modela os conceitos fundamentais e recorrentes das linguagens de programação e assim alivia o custo de extensões.

program
begin
  var x = 1;
  try
  throw 0
  catch e
  output e
  finally
  x = 0;
  output x
end
	run(makeClosure(
	  bind(‘x’, int(‘1’)),
	  seq(
	  trap(
	  catch(
	  seq(throw(int(‘0’)), C_3),
	  parametrize((value,‘e’), output(‘e’))
	  ),
	  (‘return’, seq(C_3, jump(‘return’)))
	  ),
	  output(lookup(‘x’))
	  )
	))
	where C_3 = assign(lookup(‘x’), int(‘0’))

Figura 5.15: Exemplo de um programa de Small4 e sua denotação.

Finalmente, caso não haja um componente que modele o conceito a ser adicionado, é possível estender a biblioteca. Um exemplo disso é a nova definição de `trap` apresentada na Figura 5.14. No entanto, os componentes apresentados neste trabalho são genéricos o suficiente para expressar a semântica de uma gama significativa de construções. Os componentes de tipo genérico são exemplos disso. Pode-se, por exemplo, utilizar o componente *choose* para implementar comandos, expressões ou mesmo declarações condicionais, como mostrado na Figura 5.16.

$$\mathcal{C}[\![\texttt{`if' E `then' } C_1 \texttt{ `else' } C_2]\!] = \text{choose}(\mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![C_1]\!], \mathcal{C}[\![C_2]\!])$$

$$\mathcal{E}[\![E_1 \texttt{ `?' } E_2 \texttt{ `:' } C_3]\!] = \text{choose}(\mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!], \mathcal{E}[\![E_3]\!])$$

$$\mathcal{D}[\![\texttt{`ifdef' Id } D_1 \texttt{ `else' } D_2 \texttt{ `endif'}]\!] = \text{choose}(\text{lookup}(\text{Id}), \mathcal{D}[\![D_1]\!], \mathcal{D}[\![D_2]\!])$$

Figura 5.16: Semântica de execução condicional de comandos, expressões e declarações. Nesse exemplo, a opção `Id` estar definida equivale ao valor verdadeiro estar associado a ela no *environment*.

5.2 Avaliação da Metodologia

A avaliação do trabalho é feita com base nos critérios sugeridos por Gayo (2002), Mosses (1988), Moura (1996), a saber:

1. ausência de ambiguidade: a técnica deve possibilitar o desenvolvimento de descrições precisas que identifiquem claramente o comportamento de um dado programa;

2. possibilidade de demonstração: a técnica deve possuir base matemática que permita demonstrar propriedades de programas;

3. prototipação: deve ser possível gerar automaticamente algum ambiente de execução a partir da descrição semântica da linguagem;

4. modularidade: deve ser possível escrever a especificação de maneira incremental e adicionar recursos ortogonais à linguagem sem a necessidade de alterar definições já existentes;

5. reusabilidade: a técnica deve permitir a reutilização da especificação de recursos comuns entre linguagens diferentes;

6. legibilidade: descrições semânticas devem ser legíveis para pessoas com diferentes conhecimentos de programação;

7. flexibilidade: deve ser possível especificar a grande variedade de linguagens, paradigmas e recursos de programação;

8. escalabilidade: deve ser possível aplicar a técnica formal a linguagens de programação de grande porte;

9. abstração: deve ser possível ao projetista concentrar-se nos elementos de projeto mais relevantes, sem a necessidade de se preocupar com detalhes da implementação;

10. comparação: deve ser fácil comparar duas linguagens a partir de suas definições formais.

Ausência de ambiguidade. As equações semânticas são compostas por combinações de componentes. Por consequência, o entendimento de uma equação semântica depende do entendimento dos componentes e da forma como eles são combinados. A biblioteca de componentes é definida por meio de semântica denotacional clássica sem ambiguidade. A técnica não pode introduzir ambiguidade, pois, essencialmente, ela apenas encapsula elementos da definição clássica.

Possibilidade de demonstração. A técnica apresentada se baseia no uso de componentes para definir a semântica de linguagens de programação. A prova de propriedades pode ser feita primeiro no nível da implementação dos componentes e depois no nível das combinações. Alguns componentes são definidos a partir de componentes mais simples, o que favorece essa abordagem em níveis. Além disso, o uso de componentes não fere o raciocínio equacional, visto que a semântica de cada componente é definida isoladamente.

Prototipação. A partir da biblioteca de componentes implementada na linguagem Haskell, é possível realizar a prototipação da semântica de uma linguagem. Aliando-se essa ferramenta a um analisador sintático, é possível criar um interpretador para a linguagem, com o qual se pode fazer testes e provas de conceito. É importante observar que este trabalho oferece mecanismos limitados para a definição de sistemas de tipos. Abordagens mais complexas requerem o uso de outras ferramentas em conjunto com a biblioteca de componentes e um analisador sintático.

Modularidade. A técnica apresentada se baseia na separação de interesses por meio do encapsulamento do contexto na redação das equações semânticas. Esse encapsulamento é realizado por componentes de semântica denotacional que se tornam os átomos das definições semânticas. No entanto, não são introduzidos mecanismos de controle de visibilidade de sua implementação, no sentido de que não é possível definir um módulo a partir de um subconjunto da biblioteca de componentes. Assim, a técnica proposta não melhora a modularidade das equações semânticas, mas também não impede sua apresentação de forma hierárquica conforme a sintaxe abstrata da linguagem definida.

Reusabilidade. Os componentes modelam conceitos fundamentais das linguagens de programação. Logo, podem ser utilizados sem alterações em todas as linguagens em que os conceitos que encapsulam estejam presentes. Além disso, construções recorrentes cuja semântica seja idêntica são modeladas pela mesma combinação de componentes. Portanto, a definição de uma construção pode ser reutilizada sem modificações para definir uma construção de mesma semântica em outra linguagem. Como exemplo, observe na Figura 5.17 a definição semântica da construção *if-then-else* nas linguagens C e PASCAL.

Legibilidade. As contribuições da técnica apresentada em relação à legibilidade são a remoção da dependência das informações de contexto na redação das equações semânticas e a estruturação da definição a partir dos conceitos fundamentais da linguagem. A

Comando condicional em C:

$\mathcal{C}[\![$`'if' E 'then' C`$_1$ `'else' C`$_2$$]\!]$ =
makeBlock(choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$_1]\!]$, $\mathcal{C}[\![$`C`$_2]\!]$))

$\mathcal{C}[\![$`'if' E 'then' C`$]\!]$ = makeBlock(choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$, skip))

Comando condicional em PASCAL:

$\mathcal{C}[\![$`'If' E 'Then' C`$_1$ `'Else' C`$_2$$]\!]$ =
makeBlock(choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$_1]\!]$, $\mathcal{C}[\![$`C`$_2]\!]$))

$\mathcal{C}[\![$`'If' E 'Then' C`$]\!]$ = makeBlock(choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$, skip))

Figura 5.17: Exemplo de semântica estrutura condicional nas linguagens C e Pascal.

abstração do contexto na redação das equações semânticas evita que, durante a leitura de uma equação, seja necessário lidar com entidades que não contribuem diretamente para o entendimento da semântica descrita. A utilização de conceitos fundamentais na estruturação da definição faz com que seja possível identificar semelhanças com outras linguagens, como mostrado na Figura 5.17. Além disso, conceitos fundamentais da linguagem são obstáculos menores à compreensão de um leitor não especializado do que a manipulação direta de entidades semânticas.

Flexibilidade. Embora os componentes sejam projetados para serem genéricos e abrangentes, a biblioteca é naturalmente incompleta. Mas, como a técnica desenvolvida se baseia em semântica denotacional clássica, é possível expressar recursos que não estão disponíveis na biblioteca de componentes. No entanto, o abandono dos mecanismos de combinação de componentes resulta na quebra do encapsulamento do contexto das equações semânticas, visto que ele é feito pelos componentes. Assim, a forma mais aconselhável de se expressar novos recursos é a implementação de novos componentes, como mostrado na Seção 5.1.5. Essa atividade requer do usuário conhecimento da semântica denotacional e da implementação da biblioteca. Além disso, os domínios semânticos e os mecanismos de combinação de componentes restringem as extensões aplicáveis à biblioteca de componentes. Essa limitação é intrínseca à abordagem utilizada, visto que a biblioteca é uma implementação de um modelo semântico específico. Extensões que entrem em conflito com esse modelo não são compatíveis com a biblioteca apresentada.

Escalabilidade. Como apresentado no estudo de caso da Seção 5.1, o método desenvolvido pode ser aplicado à definição semântica de linguagens de programação de pequeno e grande porte. A adição de novos conceitos não compromete a legibilidade

da definição, o que se deve à remoção da dependência de informações de contexto na redação das equações semânticas. Por fim, o impacto das extensões sobre as equações previamente definidas é pequeno, visto que grande parte dos conceitos fundamentais e recorrentes de linguagens imperativas é abordada pela biblioteca. Além disso, é possível criar novos componentes que encapsulam os conceitos adicionados, mantendo a separação de interesses das definições. No estudo de caso e na definição parcial de Java, observou-se que a complexidade da definição cresce em proporção linear ao tamanho da linguagem definida. Em geral, o impacto de cada funcionalidade fica restrito às suas equações semânticas, não afetando o restante da definição.

Abstração. A técnica apresentada possibilita a definição semântica de linguagens de programação a partir de seus conceitos fundamentais. Isso significa que o fluxo das informações de contexto das construções é encapsulado juntamente com a implementação desses conceitos. De fato, conceitos como continuação e o uso do *environment* para passar configurações aos constituintes de uma construção não ficam explícitos nas definições semânticas. Esses detalhes são encapsulados pelos componentes, possibilitando a leitura da definição com base nos conceitos apresentados e não na sua implementação, privilegiando separação de interesses.

Comparação. Como mostrado na Figura 5.17, desde que sua semântica seja a mesma, duas construções de linguagens diferentes possuem a mesma representação baseada em componentes. Além disso, caso haja diferenças, como mostrado na Figura 5.18, a organização da definição semântica por meio dos conceitos fundamentais das linguagens facilita a comparação de recursos linguísticos.

5.3 Comparação com Trabalhos Relacionados

Diversos trabalhos abordam o problema de modularidade de definições semânticas, com destaque para os trabalhos citados no Capítulo 2.

A semântica incremental (Tirelo et al. 2009), descrita na Seção 2.7, aborda o conceito de vagueza nas definições semânticas. Detalhes são acrescentados de forma incremental a uma definição incompleta da semântica de uma linguagem. Os incrementos são compostos por transformações das denotações. Com inspiração na programação

[1] Algumas formas sintáticas do comando condicional da linguagem TCL foram omitidas para simplificar o exemplo, mas sua semântica equivale à combinação de dois ou mais comandos condicionais sem prejuízo ao argumento apresentado

Comando condicional em C:

$\mathcal{C}[\![$`‘if’ E ‘then’ C`$_1$ `‘else’ C`$_2$$]\!]$ =
makeBlock(choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$_1]\!]$, $\mathcal{C}[\![$`C`$_2]\!]$))

$\mathcal{C}[\![$`‘if’ E ‘then’ C`$]\!]$ = makeBlock(choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$, skip))

Comando condicional em TCL:

$\mathcal{C}[\![$`‘If’ ‘{’ E ‘}’ ‘{’ C`$_1$`‘}’ ‘else’ ‘{’ C`$_2$ `‘}’`$]\!]$ =
choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$_1]\!]$, $\mathcal{C}[\![$`C`$_2]\!]$)

$\mathcal{C}[\![$`‘If’ ‘{’ E ‘}’ ‘{’ C`$_1$`‘}’`$]\!]$ = choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$, skip)

Figura 5.18: Exemplo de semântica estrutura condicional nas linguagens C e TCL[1]. Note que, ao contrário do que ocorre na linguagem C, o comando condicional da linguagem TCL não restringe o escopo de declarações.

orientada a aspectos, a natureza transversal dos conceitos abordados nas definições semânticas é explorada na criação de módulos. Consequentemente, atinge-se um alto nível de modularidade ao se evitar o entrelaçamento de conceitos expressos nas equações semânticas. Já os componentes da técnica apresentada evitam o espalhamento do contexto e melhoram a separação de interesses ortogonais. No entanto, ao contrário da transformação de denotações, a combinação de componentes permite o raciocínio equacional.

Assim como a técnica apresentada, a semântica de mônadas (Moggi 1989, 1991) aborda o problema de modularidade de definições semânticas ao abstrair o contexto das equações semânticas. No entanto, as mônadas são mecanismos de abstração complexos. Consequentemente, as definições semânticas monádicas atingem um elevado nível de modularidade, mas dependem de complexas operações de transformação de mônadas. A técnica apresentada é mais simples, explorando a padronização de interfaces das denotações e o encapsulamento de conceitos fundamentais em componentes.

Além disso, a semântica monádica permite a definição incremental do modelo semântico. Pode-se por exemplo definir primeiro um núcleo simples, acrescentando depois continuação, *environment* e *store*. A técnica apresentada, por outro lado, ao limitar o escopo de sua aplicação ao paradigma imperativos, define um modelo semântico fixo. No entanto, é possível que recursos disponíveis no modelo semântico não se façam presentes nas equações semânticas por omissão dos componentes relacionados. O custo da flexibilidade da semântica monádica se faz visível na complexidade das operações de transformação de mônadas, enquanto a abordagem denotacional baseada em componentes se mantém simples ao utilizar um modelo semântico fixo.

A semântica de ações (Mosses 1977) se assemelha à técnica apresentada ao en-

capsular alguns conceitos fundamentais das linguagens de programação em ações. Essa organização somada à nomenclatura próxima à uma linguagem natural permite a criação de definições legíveis. No entanto, as ações encapsulam conceitos mais simples do que aqueles encapsulados pelos componentes. De fato, a semântica de ações é utilizada como base para a abordagem baseada em componentes de Mosses (2005). Por fim, a diferença principal entre a semântica de ações e a semântica denotacional baseada em componentes está na escolha do modelo subjacente. Aquela possui um núcleo operacional, enquanto esta se baseia em semântica denotacional.

A abordagem implícita de semântica operacional modular (Mosses & New 2009) possibilita definições modulares ao abstrair as informações de contexto nas regras de transição. Essa abstração é feita por meio da passagem implícita de informações nos rótulos das transições. Esse mecanismo se assemelha à comunicação de contexto entre componentes utilizada neste trabalho. No entanto, quando necessário, o contexto é utilizado de forma explícita em algumas transições de I-MSOS, o que não ocorre na técnica apresentada neste trabalho.

A semântica baseada em componentes (Mosses 2005) é a principal inspiração deste trabalho. No entanto, esta utiliza a semântica de ações e I-MSOS como base para a definição da semântica de seus componentes. Além disso, em contraste com a semântica baseada em componentes, este trabalho tem seu escopo de aplicação limitado às linguagens de programação imperativas. O uso da semântica denotacional como modelo subjacente apresenta desafios para a criação de componentes. Como sugerido por Mosses (2005), a baixa legibilidade e escalabilidade decorrente do uso explícito de informações de contexto nas denotações é um obstáculo para o uso da semântica denotacional como base de uma técnica de definição por meio de componentes. Esse obstáculo é transposto por meio do uso de uma biblioteca de componentes, a qual encapsula o fluxo de informações de contexto entre as denotações.

5.4 Conclusões

A metodologia proposta tem impacto positivo direto nos critérios de prototipação, modularidade, reusabilidade, legibilidade, escalabilidade e comparação sem prejuízo dos demais critérios citados.

Este trabalho complementa as contribuições de trabalhos anteriores, sendo possível aliar uma metodologia já existente com a metodologia proposta. Com isso, acredita-se que este trabalho contribui para a popularização da semântica formal de linguagens de programação.

O próximo capítulo apresenta as conclusões desta dissertação, as principais contribuições e possíveis trabalhos futuros.

Capítulo 6

Conclusão

Este trabalho apresentou uma nova abordagem de semântica denotacional baseada em componentes. Essa técnica utiliza uma biblioteca de componentes que permitem combinar denotações para compor a semântica de construções e encapsulam conceitos fundamentais e recorrentes das linguagens de programação.

A biblioteca de componentes permite a remoção da dependência aparente das informações de contexto na redação das equações semânticas, tornando-as mais modulares. Para isso, as assinaturas dos componentes são padronizadas e o fluxo de informações de contexto é encapsulado nos componentes. Consequentemente, as definições semânticas mapeiam as construções da sintaxe abstrata das linguagens de programação diretamente a combinações de componentes denotacionais. Como resultado, as definições semânticas se tornam mais legíveis.

O encapsulamento de conceitos fundamentais e recorrentes das linguagens de programação imperativas em componentes facilita a definição semântica de linguagens imperativas de grande porte. Isso ocorre porque grande parte dos conceitos abordados por essas definições é abordada por componentes previamente definidos. O exercício de definição se resume então a combinar os componentes de forma a modelar a semântica desejada. Assim, eleva-se o nível de abstração das definições semânticas compondo as denotações das construções de linguagens a partir de conceitos e da forma como estes são combinados. O objetivo principal dessa abordagem é melhorar a escalabilidade da semântica denotacional. Esse objetivo é buscado por meio da melhora da modularidade das definições semânticas.

No entanto, a biblioteca de componentes é essencialmente um trabalho em aberto. Conforme as linguagens de programação evoluem, novos conceitos precisam ser incorporados. A abordagem recomendada nesse caso é o encapsulamento desses conceitos em novos componentes, como demonstrado na Seção 5.1. Não obstante, caso maior li-

berdade se faça necessária, é também possível utilizar semântica denotacional clássica em conjunto com os componentes na redação de equações semânticas. No pior caso, a incorporação de novos componentes pode conflitar com os mecanismos já previstos pela biblioteca e, potencialmente, exigir a restruturação de um número muito grande de componentes pré-existentes.

Por fim, para conferir maior modularidade à definição da semântica dos componentes, pode-se substituir a semântica denotacional clássica utilizada por outras abordagens denotacionais.

6.1 Contribuições do Trabalho

A principal contribuição deste trabalho é a criação de uma biblioteca de componentes genéricos para formulação de definições denotacionais, um novo método para a obtenção de escalabilidade e reúso em definições de semântica denotacional. O reúso obtido é fruto do encapsulamento de conceitos fundamentais e recorrentes de linguagens de programação em componentes. Estes podem ser reaproveitados sem necessidade de redefinição em diversas definições semânticas. A escalabilidade obtida é fruto da melhora da modularidade e da legibilidade das definições semânticas.

A técnica apresentada promove a remoção da dependência aparente do contexto das construções na redação de suas denotações. Com isso, melhora-se a modularidade das definições e obtêm-se definições semânticas mais legíveis, compostas pelo mapeamento direto entre construções e combinações de componentes.

A abstração do contexto permite a aplicação de uma metodologia de composição de definições formais por meio de blocos básicos à semântica denotacional. A abordagem de Mosses (2005) faz uso da semântica de ações e da semântica operacional modular com propagação implícita. Isso se deve à modularidade apresentada por essas técnicas. Consequentemente, um dos maiores desafios deste trabalho foi melhorar a modularidade da semântica denotacional clássica.

Conforme mencionado anteriormente, é importante ressaltar que este trabalho complementa as contribuições de trabalhos anteriores, sendo possível aliar uma metodologia já existente com a metodologia proposta. Com isso, acredita-se que este trabalho contribui para a popularização da semântica formal de linguagens de programação.

Por fim, foi implementada uma ferramenta para prototipação de linguagens imperativas de porte real. Essa ferramenta é composta por uma implementação na linguagem Haskell da biblioteca de componentes de semântica denotacional apresentado

neste trabalho. Unindo o sistema implementado por meio da biblioteca a um *parser* que crie a árvore de sintaxe abstrata dos programas, é possível criar um interpretador. Embora ineficiente, este pode ser utilizado para validar definições semânticas e realizar experimentos com extensões de linguagens.

6.2 Trabalhos Futuros

A complexidade inerente a grandes linguagens de programação inviabilizou seu uso nos estudos de caso. A linguagem Java, por exemplo, teve sua semântica parcialmente definida, como pode ser visto no Apêndice B, mas sua complexidade dificultaria a exposição de ideias. Consequentemente, seria interessante terminar esse esforço de definição e avaliar os benefícios da técnica apresentada, comparando a definição resultante com outras definições semânticas de linguagens de grande porte.

Como a biblioteca de componentes está sujeita a extensões futuras, seria interessante comparar os componentes definidos com aqueles resultantes do trabalho do grupo *Programming Language Components and Specifications* (PLanCompS). Este grupo se propôs no ano de 2011 a criar uma lista abrangente de componentes semânticos e compor uma metodologia de criação de novos componentes. Seria interessante também, estender a biblioteca apresentada com definições denotacionais de alguns dos componentes criados pelo grupo.

O trabalho apresentado tem seu escopo de aplicação limitado à definição semântica de linguagens de programação imperativas. Isso foi feito para que se obtivesse uma lista coesa, mas ainda assim abrangente, de componentes semânticos. Seria interessante avaliar possíveis extensões que possibilitassem a definição semântica de linguagens de programação pertencentes a outros paradigmas.

Finalmente, a semântica denotacional é uma boa ferramenta para a demonstração de propriedades de linguagens de programação. Seria interessante avaliar o impacto da organização das definições semânticas por meio de componentes à demonstração de propriedades. Como os componentes possuem definições semânticas denotacionais, é possível substitui-los por suas definições nas equações semânticas. Assim, a demonstração pode ser feita utilizando-se semântica denotacional clássica diretamente. No entanto, seria interessante explorar outras técnicas de demonstração que se beneficiassem dos componentes. Além disso, beneficiando-se da padronização trazida pelos componentes, poderia-se explorar transposição de demonstrações de propriedades entre definições semânticas de linguagens diferentes.

Apêndice A

Definição da Linguagem Small

Este apêndice apresenta uma definição semântica da linguagem Small por meio de componentes de semântica denotacional – baseada em Gordon (1979). Essa linguagem possui declarações, comandos, expressões, procedimentos e funções.

A.1 Sintaxe Abstrata

Esta seção apresenta a sintaxe abstrata da linguagem Small. Dado o enfoque deste trabalho, a sintaxe concreta da linguagem é omitida.

P	→	program C
D	→	const I = E |var I = E |I(A);C |D_1;D_2
A	→	M I |A M I |empty
M	→	value |ref |const
C	→	E_1 := E_2 |output E |E_1(L) |if E then C_1 else C_2 |while E do C
	|	begin D;C end |C_1;C_2 |goto I |I:C |continue |break
E	→	N |true |false |read |I |if E then E_1 else E_2 |E_1 O E_2
L	→	E |E, L
O	→	+ |- |× |÷ |= |!=

A.2 Semântica

A.2.1 Programa

A execução de um programa escrito em Small produz uma saída e, possivelmente, uma mensagem indicando um erro na execução. O programa também pode ler dados recebidos como entrada.

$$\begin{array}{l}\mathcal{P}[\![\texttt{program C}]\!]\ \text{i} = \text{result}(\text{run}(\text{addEnv}(\mathcal{J}[\![\texttt{C}]\!], \mathcal{C}[\![\texttt{C}]\!])\ \text{i}\ \text{r}_0\ \text{s}_0 \\ \quad \text{where result (t,e,r,s)} = [\text{s `output'} \bullet \text{errorState}] \\ \qquad\qquad \text{errorState t} = (\text{t} = \text{stop}) \rightarrow \text{`'}, \text{`Error'}\end{array}$$

Figura A.1: Semântica de programa de Small.

A.2.2 Declarações

Em Small, as declarações são feitas de forma sequencial, ou seja, toda declaração pode fazer uso das declarações que a precedem. É possível declarar variáveis constantes. Finalmente, a linguagem permite passagem de parâmetros por valor e por referência, além de parâmetros constantes.

$$\begin{array}{l}\mathcal{D}[\![\texttt{D}_1\texttt{;D}_2]\!] = \text{elabSeq}(\mathcal{D}[\![\texttt{D}_1]\!], \mathcal{D}[\![\texttt{D}_2]\!]) \\ \mathcal{D}[\![\texttt{const I = E}]\!] = \text{bind}(\texttt{I}, \text{const}(\text{deref}(\mathcal{E}[\![\texttt{E}]\!]))) \\ \mathcal{D}[\![\texttt{var I = E}]\!] = \text{bind}(\texttt{I}, \text{ref}(\text{deref}(\mathcal{E}[\![\texttt{E}]\!]))) \\ \mathcal{D}[\![\texttt{I(A);C}]\!] = \text{bind}(\texttt{I}, \text{parametrize}(\mathcal{A}[\![\texttt{A}]\!], \mathcal{C}[\![\texttt{C}]\!])) \\ \\ \mathcal{A}[\![\texttt{const I}]\!] = (\text{constant}, \texttt{I}) \\ \mathcal{A}[\![\texttt{ref I}]\!] = (\text{ref}, \texttt{I}) \\ \mathcal{A}[\![\texttt{value I}]\!] = (\text{value}, \texttt{I}) \\ \mathcal{A}[\![\texttt{M I, A}]\!] = [\mathcal{A}[\![\texttt{M I}]\!] \bullet \mathcal{A}[\![\texttt{A}]\!]] \\ \mathcal{A}[\![]\!] = []\end{array}$$

Figura A.2: Semântica de declarações de Small.

A.2.3 Expressões

A linguagem Small possui valores booleanos e inteiros, variáveis, operações aritméticas e lógicas, funções, expressões condicionais e leitura da entrada. Observe que, no momento da chamada de função, é criado um novo bloco de memória, onde as variáveis da função são guardadas. Ao fim da chamada, o bloco é destruído.

$\mathcal{E}[\![\texttt{true}]\!]$ = bool('true')
$\mathcal{E}[\![\texttt{false}]\!]$ = bool('false')
$\mathcal{E}[\![\texttt{N}]\!]$ = int(N)
$\mathcal{E}[\![\texttt{I}]\!]$ = lookup(I)
$\mathcal{E}[\![\texttt{read}]\!]$ = input
$\mathcal{E}[\![\texttt{E}_1\ \texttt{O}\ \texttt{E}_2]\!]$ = apply(O,$\mathcal{E}[\![\texttt{E}_1]\!]$,$\mathcal{E}[\![\texttt{E}_2]\!]$)
$\mathcal{E}[\![\texttt{if E then E}_1\ \texttt{else E}_2]\!]$ = choose($\mathcal{E}[\![\texttt{E}]\!]$, $\mathcal{E}[\![\texttt{E}_1]\!]$, $\mathcal{E}[\![\texttt{E}_2]\!]$)
$\mathcal{L}[\![\texttt{E, L}]\!]$ = [$\mathcal{E}[\![\texttt{E}]\!]$ • $\mathcal{L}[\![\texttt{L}]\!]$]
$\mathcal{L}[\![\texttt{E}]\!]$ = $\mathcal{E}[\![\texttt{E}]\!]$

Figura A.3: Semântica de expressões de Small.

A.2.4 Comandos

Os comandos de Small compreendem assinalamento, escrita da saída, comando condicional, laço com sequenciadores *break* e *continue*, blocos e chamadas de procedimentos. A chamada de procedimentos possui o mesmo mecanismo de bloco de memória presente na chamada de funções.

$\mathcal{C}[\![\texttt{E}_1\ \texttt{:=}\ \texttt{E}_2]\!]$ = assign($\mathcal{E}[\![\texttt{E}_1]\!]$, $\mathcal{E}[\![\texttt{E}_2]\!]$)
$\mathcal{C}[\![\texttt{output E}]\!]$ = output($\mathcal{E}[\![\texttt{E}]\!]$)
$\mathcal{C}[\![\texttt{if E then C}_1\ \texttt{else C}_2]\!]$ = choose($\mathcal{E}[\![\texttt{E}]\!]$, $\mathcal{C}[\![\texttt{C}_1]\!]$, $\mathcal{C}[\![\texttt{C}_2]\!]$)
$\mathcal{C}[\![\texttt{while E do C}]\!]$ = escape(loop($\mathcal{E}[\![\texttt{E}]\!]$,
escape($\mathcal{C}[\![\texttt{C}]\!]$,'continue'))
'break')
$\mathcal{C}[\![\texttt{break}]\!]$ = throw("break")
$\mathcal{C}[\![\texttt{continue}]\!]$ = throw("continue")
$\mathcal{C}[\![\texttt{C}_1\texttt{;C}_2]\!]$ = seq($\mathcal{C}[\![\texttt{C}_1]\!]$,$\mathcal{C}[\![\texttt{C}_2]\!]$)
$\mathcal{C}[\![\texttt{begin D;C end}]\!]$ = makeClosure($\mathcal{D}[\![\texttt{D}]\!]$, $\mathcal{C}[\![\texttt{C}]\!]$)
$\mathcal{C}[\![\texttt{E}_1\texttt{(L)}]\!]$ = createMemoryBlock(call($\mathcal{E}[\![\texttt{E}_1]\!]$,$\mathcal{E}[\![\texttt{L}]\!]$))
where createMemoryBlock f = seq(push, seq(f, pop))

Figura A.4: Semântica de comandos de Small.

A.2.4.1 Saltos

$\mathcal{C}[\![\texttt{I:C}]\!]$ = $\mathcal{C}[\![\texttt{C}]\!]$
$\mathcal{C}[\![\texttt{goto I}]\!]$ = jump(I)

Figura A.5: Semântica de saltos de Small.

A linguagem Small oferece suporte a saltos. Observe que, como indicado pelo uso do componente `elabCol`, não é permitido o uso de rótulos com o mesmo nome nas duas opções de um comando condicional. Caso essa regra seja ignorada, um dos rótulos perderá sua validade. A coleta de rótulos em procedimentos é realizada no momento de sua declaração. Como o corpo das funções de Small é uma expressão, não é permitido o salto dentro de funções.

$\mathcal{J}[\![\texttt{I: C}]\!] = \text{addLabel}(\text{I}, \mathcal{C}[\![\texttt{C}]\!])$
$\mathcal{J}[\![\texttt{while E do C}]\!] = \text{elab}(\mathcal{J}[\![\texttt{C}]\!],\mathcal{C}[\![\texttt{while E do C}]\!])$
$\mathcal{J}[\![\texttt{if E then } \texttt{C}_1 \texttt{ else } \texttt{C}_2]\!] = \text{elabSeq}(\mathcal{J}[\![\texttt{C}_1]\!], \mathcal{J}[\![\texttt{C}_2]\!])$
$\mathcal{J}[\![\texttt{C}_1\texttt{;C}_2]\!] = \text{elabSeq}(\text{elab}(\mathcal{J}[\![\texttt{C}_1]\!], \mathcal{C}[\![\texttt{C}_2]\!]), \mathcal{J}[\![\texttt{C}_2]\!])$
$\mathcal{J}[\![\texttt{begin D;C end}]\!] = \text{skipLabel}$
$\mathcal{J}[\![\texttt{goto I}]\!] = \text{skipLabel}$
$\mathcal{J}[\![\texttt{E}_1 \texttt{ := } \texttt{E}_2]\!] = \text{skipLabel}$
$\mathcal{J}[\![\texttt{output E}]\!] = \text{skipLabel}$
$\mathcal{J}[\![\texttt{break}]\!] = \text{skipLabel}$
$\mathcal{J}[\![\texttt{continue}]\!] = \text{skipLabel}$

$\mathcal{C}[\![\texttt{begin D;C end}]\!] = \text{addEnv}(\mathcal{J}[\![\texttt{C}]\!], \text{makeClosure}(\mathcal{D}[\![\texttt{D}]\!],\mathcal{C}[\![\texttt{C}]\!]))$

Figura A.6: Coleta de rótulos de Small.

Apêndice B

Definição de um Subconjunto da Linguagem Java

Este apêndice apresenta a definição semântica baseada em componentes de um subconjunto da linguagem Java, intitulado Java0. Algumas das funcionalidades de Java ausentes em Java0 são: classes genéricas, pacotes, opções de visibilidade para os atributos e membros de classes, exceções e instanciação de arranjos de objetos por meio de construtores. Também não é abordado o gerenciamento de memória. Java0 possui um sistema de tipos simples, sem coerção de tipos, exceto *casting* de objetos. Por fim, Java0 não possui sobrecarga de métodos, mas possui *dynamic dispatch*.

Além das limitações de funcionalidades, a definição semântica de Java0 utiliza algumas simplificações na sintaxe abstrata. Por exemplo, o acesso a atributos e métodos na semântica abstrata deve mencionar o parâmetro *this*. No caso de membros e atributos estáticos, o acesso pode ser feito por meio do nome da classe.

B.1 Sintaxe Abstrata

B.1.1 Comandos

P	→	'java' Id 'from' D
C	→	'skip' |'break' |'continue' |C_1 ';' C_2 |Loop |Choice |E
	|	'print' E |'return' E |'return' |D |Block
Block	→	'{' C '}'
Choice	→	'if' E 'then' C_1 'else' C_2 |'switch' E CaseExps 'default' C
CaseExps	→	'case' E C CaseExps |'case' E C |empty
Loop	→	'while' E C |'for' C_1 E C_2 C_3

B.1.2 Expressões

E	→	E_1 O '=' E_2 |E_1 O E_2 |Id '(' EA ')' |Id |B |'new' Id '(' EA ')'
	|	'new' Id {' D '}' |'new' Id '(' EA ')' {' D '}' |E 'instanceof' Id
	|	Reference |E '.' Id |'(' Type ')' E |E_1 '[' E_2 ']' |ArrayExpression
EA	→	E EA |empty
ArrayExpression	→	'new' Id ArrayIndex |'new' BasicType ArrayIndex
ArrayIndex	→	'[' E ']' |'[' E ']' ArrayIndex
Reference	→	'super' |'this'
O	→	'+' |'-' |'*' |'/' |'^' |'%' |'<<' |'>>' |'>>>' |'&' |'|' |'&&'
	|	'||' |'<=' |'>=' |'==' |'<' |'>'
B	→	Integer |Boolean |Float |Double |Long |Char |String |Byte |Short

B.1.3 Declarações

D	→	Variable |Class |Member |D_1 ';' D_2
Class	→	'class' Id '{' D '}'
Member	→	Modifier Method |Modifier Variable
Modifier	→	'static' |empty
Variable	→	Type Id |Type Id '=' E |Type Id '[]' |Type Id '[]' = '{' ArrayInitilizer '}'
	|	Type Id '[]' = ArrayExpression
ArrayInitializer	→	E ',' ArrayInitializer |'{' ArrayInitializer '}' |E
Method	→	Type Id '(' DA ')' Block |'void' Id '(' DA ')' Block
DA	→	Type Id ',' DA |Type Id '[]' ',' DA |emtpy

B.2 Semântica

B.2.1 Programa

$$\mathcal{P}[\![\texttt{`java' I `from' D}]\!]\ \text{i args} = \text{run}(\text{addEnv}(\text{dec}, \text{apply}(\text{main}, \text{args}))\ \text{i}\ r_0\ s_0$$
$$\text{where}\quad \text{dec} = \text{accum}(\text{elabRec}(\mathcal{D}[\![\texttt{D}]\!]))$$
$$\text{main} = \text{lookup}(\text{I},\text{'main'})$$

Um programa em Java0 é constituído por uma série de declarações e uma classe principal. A execução de um programa consiste em executar o método *main* da classe principal.

B.2.2 Comandos

$\mathcal{C}[\![\texttt{`skip'}]\!] = \text{skip}$
$\mathcal{C}[\![\texttt{D}]\!] = \text{accum}(\mathcal{D}[\![\texttt{D}]\!])$
$\mathcal{C}[\![\texttt{E}]\!] = \mathcal{E}[\![\texttt{E}]\!]$
$\mathcal{C}[\![\texttt{C}_1 \ \texttt{`;'} \ \texttt{C}_2]\!] = \text{seq}(\mathcal{C}[\![\texttt{C}_1]\!], \mathcal{C}[\![\texttt{C}_2]\!])$
$\mathcal{C}[\![\texttt{`\{'} \ \texttt{C} \ \texttt{`\}'}]\!] = \text{makeBlock}(\mathcal{C}[\![\texttt{C}]\!]))$
$\mathcal{C}[\![\texttt{`print'} \ \texttt{E}]\!] = \text{output}(\mathcal{E}[\![\texttt{E}]\!])$

Entre comandos, podem-se encontrar comandos vazios, declarações, blocos, sequências de comandos e comandos de impressão.

$\mathcal{C}[\![\texttt{`while'} \ \texttt{E} \ \texttt{C}]\!] = \text{makeBlock}($
$\quad \text{escape}(\text{loop}(\mathcal{E}[\![\texttt{E}]\!], \text{escape}(\mathcal{C}[\![\texttt{C}]\!], \text{`continue'})), \text{`break'})$
$)$
$\mathcal{C}[\![\texttt{`do'} \ \texttt{C} \ \texttt{`while'}]\!] = \text{makeBlock}($
$\quad \text{escape}(\text{loop}(\text{escape}(\mathcal{C}[\![\texttt{C}]\!], \text{`continue'}), \mathcal{E}[\![\texttt{E}]\!]), \text{`break'})$
$)$
$\mathcal{C}[\![\texttt{`for'} \ \texttt{F} \ \texttt{C}_1 \ \texttt{E} \ \texttt{C}_2 \ \texttt{C}_3]\!] = \text{makeBlock}($
$\quad \text{escape}($
$\qquad \text{for}(\mathcal{C}[\![\texttt{C}_1]\!], \mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![\texttt{C}_2]\!], \text{escape}(\mathcal{C}[\![\texttt{C}_3]\!], \text{`continue'})),$
$\qquad \text{`break'})$
$)$

Os laços de Java0 são *while*, *do-while* e *for*. A definição de cada um dos laços especifica os pontos de saída utilizados pelos sequenciadores *break* e *continue*.

$\mathcal{C}[\![\texttt{`if'} \ \texttt{E} \ \texttt{`then'} \ \texttt{C}_1 \ \texttt{`else'} \ \texttt{C}_2]\!] = \text{makeBlock}(\text{choose}(\mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![\texttt{C}_1]\!], \text{C}_2))$
$\mathcal{C}[\![\texttt{`switch'} \ \texttt{E} \ \texttt{CaseExps} \ \texttt{`default'} \ \texttt{C}]\!] =$
$\quad \text{escape}($
$\qquad \text{trap}(\text{E}, \mathcal{SC}[\![\texttt{CaseExps}]\!], \mathcal{C}[\![\texttt{C}]\!]),$
$\qquad \text{`break'}$
$\quad)$
$\mathcal{K}[\![\texttt{`case'} \ \texttt{E} \ \texttt{C} \ \texttt{CaseExps}]\!] = [(\mathcal{E}[\![\texttt{E}]\!], \text{seq}(\mathcal{C}[\![\texttt{C}]\!], \mathcal{C}[\![\texttt{CaseExps}]\!])) \bullet \mathcal{K}[\![\texttt{CaseExps}]\!]]$
$\mathcal{K}[\![\texttt{`case'} \ \texttt{E} \ \texttt{C}]\!] = (\mathcal{E}[\![\texttt{E}]\!], \mathcal{C}[\![\texttt{C}]\!])$
$\mathcal{K}[\![]\!] = []$
$\mathcal{C}[\![\texttt{`case'} \ \texttt{E} \ \texttt{C} \ \texttt{CaseExps}]\!] = \text{seq}(\mathcal{C}[\![\texttt{C}]\!], \mathcal{C}[\![\texttt{CaseExps}]\!])$
$\mathcal{C}[\![\texttt{`case'} \ \texttt{E} \ \texttt{C}]\!] = \mathcal{C}[\![\texttt{C}]\!]$

Os comandos condicionais de Java0 são *if-then-else* e *switch*. A ausência da segunda alternativa daquele ou da alternativa *default* deste pode ser modelada pelo

uso do comando vazio e, por isso, não é explicitada nas equações semânticas.

$\mathcal{C}[\![\texttt{`break'}]\!]$ = throw('break')
$\mathcal{C}[\![\texttt{`continue'}]\!]$ = throw('continue')
$\mathcal{C}[\![\texttt{`return' E}]\!]$ = resultIs($\mathcal{E}[\![\texttt{E}]\!]$)
$\mathcal{C}[\![\texttt{`return'}]\!]$ = throw('return')

Os sequenciadores de Java0 são *break*, *continue* e *return*.

B.2.3 Declarações

$\mathcal{D}[\![\texttt{D}_1\ \texttt{`;'}\ \texttt{D}_2]\!]$ = accum(elabSeq($\mathcal{D}[\![\texttt{D}_1]\!]$, $\mathcal{D}[\![\texttt{D}_2]\!]$))
$\mathcal{D}[\![\texttt{Type I}]\!]$ = bind(I, ref(getDefault Type), Type)
$\mathcal{D}[\![\texttt{Type I `=' E}]\!]$ = bind(I, ref(deref($\mathcal{E}[\![\texttt{E}]\!]$)), Type)
$\mathcal{D}[\![\texttt{Type I`[]'}]\!]$ = bind(I, ref(getDefault Type), Type '[]')
$\mathcal{D}[\![\texttt{Type I`[]' `=' `\{' ArrayInitializer `\}'}]\!]$ =
    bind(I, ref(array(0,size values,values)), Type '[]')
  where values = $\mathcal{AD}[\![\texttt{ArrayInitializer}]\!]$

$\mathcal{AD}[\![\texttt{E `,' ArrayInitializer}]\!]$ = [$\mathcal{E}[\![\texttt{E}]\!]$ • $\mathcal{D}[\![\texttt{ArrayInitializer}]\!]$]
$\mathcal{AD}[\![\texttt{E }]\!]$ = $\mathcal{E}[\![\texttt{E}]\!]$
$\mathcal{AD}[\![\texttt{`\{' ArrayInitializer `\}'}]\!]$ = array(0,size values,values)
  where values = $\mathcal{AD}[\![\texttt{ArrayInitializer}]\!]$

Java0 possui declarações de variáveis. Quando valores não são especificados, o valor *default* associado ao tipo da variável é armazenado.

$\mathcal{D}[\![\texttt{void I `(' DA `)' Block}]\!]$ = bind(I, parametrize(arg, escape($\mathcal{C}[\![\texttt{Block}]\!]$, 'return')))
  where arg = [(ref,'this') • $\mathcal{DA}[\![\texttt{DA}]\!]$]
$\mathcal{D}[\![\texttt{Type I `(' DA `)' Block}]\!]$ = bind(I, parametrize(arg, valof($\mathcal{C}[\![\texttt{Block}]\!]$)))
  where arg = [(ref,'this') • $\mathcal{DA}[\![\texttt{DA}]\!]$]

$\mathcal{DA}[\![\texttt{Type I DA}]\!]$ = ((I, value), $\mathcal{DA}[\![\texttt{DA}]\!]$)
$\mathcal{DA}[\![\texttt{Type I `[]' FPar}]\!]$ = ((I, value), $\mathcal{DA}[\![\texttt{DA}]\!]$)
$\mathcal{DA}[\![]\!]$ = ()

Procedimentos e funções podem ser declarados. A diferença entre eles está no retorno. Procedimentos podem ser abandonados, enquanto funções retornam valores.

$\mathcal{D}[\![\texttt{`class' Id `\{' D `\}'}]\!] =$
$\quad \mathcal{Z}[\![\texttt{D}]\!]$ (λc o.elabRec(bind(Id, class(elabRec(c), elabRec(o)), static))) skip skip
$\mathcal{Z}[\![\texttt{D}_1\ \texttt{`;'}\ \texttt{D}_2]\!]$ q = $\mathcal{Z}[\![\texttt{D}_1]\!]$; $\mathcal{Z}[\![\texttt{D}_2]\!]$ q
$\mathcal{Z}[\![\texttt{Modifier Method}]\!]$ q c o = q elabCol(c,$\mathcal{D}[\![\texttt{Modifier Method}]\!]$) o
$\mathcal{Z}[\![\texttt{`static' Variable}]\!]$ q c o = q elabCol(c,$\mathcal{D}[\![\texttt{`static' Variable}]\!]$) o
$\mathcal{Z}[\![\texttt{`' Variable}]\!]$ q c o = q c elabCol(o,$\mathcal{D}[\![\texttt{`' Variable}]\!]$) o

A declaração de classes exige que as declarações de membros e atributos estáticos sejam separadas das demais.

B.2.4 Expressões

$\mathcal{E}[\![\texttt{I}]\!]$ = lookup(I)
$\mathcal{E}[\![\texttt{B}]\!] = \mathcal{B}[\![\texttt{B}]\!]$
$\mathcal{B}[\![\texttt{Integer}]\!]$ = int(Integer)
$\mathcal{B}[\![\texttt{Boolean}]\!]$ = bool(Boolean)
$\mathcal{B}[\![\texttt{Float}]\!]$ = float(Float)
$\mathcal{B}[\![\texttt{Double}]\!]$ = double(Double)
$\mathcal{B}[\![\texttt{ Long}]\!]$ = long(Long)
$\mathcal{B}[\![\texttt{Char}]\!]$ = char(Char)
$\mathcal{B}[\![\texttt{String}]\!]$ = string(String)
$\mathcal{B}[\![\texttt{Byte}]\!]$ = byte(Byte)
$\mathcal{B}[\![\texttt{Short}]\!]$ = short(Short)

As expressões mais simples de Java0 são os literais e os acessos a variáveis.

$\mathcal{E}[\![\texttt{E}_1\ \texttt{O}\ \texttt{E}_2]\!]$ = apply($\mathcal{O}[\![\texttt{O}]\!]$, $\mathcal{E}[\![\texttt{E}_1]\!]$, $\mathcal{E}[\![\texttt{E}_2]\!]$)
$\mathcal{E}[\![\texttt{E}_1\ \texttt{O `='}\ \texttt{E}_2]\!]$ = assign($\mathcal{E}[\![\texttt{E}_1]\!]$, apply($\mathcal{O}[\![\texttt{O}]\!]$, $\mathcal{E}[\![\texttt{E}_1]\!]$, $\mathcal{E}[\![\texttt{E}_2]\!]$))
$\mathcal{E}[\![\texttt{E}_1\texttt{`['}\ \texttt{E}_2\ \texttt{`]'}]\!]$ = index($\mathcal{E}[\![\texttt{E}_1]\!]$, $\mathcal{E}[\![\texttt{E}_2]\!]$)

Operações básicas como soma, subtração e testes de igualdade são realizados por meio do componente `apply`. Definições das operações são omitidas neste apêndice.

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{`fun' E `.' Id `(' EA `)'}]\!] = \text{apply(method, arg)}\\
&\quad \text{where } \text{method} = \text{lookup(target, Id)}\\
&\qquad\qquad \text{arg} = [\text{target} \bullet \mathcal{EA}[\![\texttt{EA}]\!]]\\
&\qquad\qquad \text{target} = \mathcal{E}[\![\texttt{E}]\!]\\
&\mathcal{E}[\![\texttt{`static-fun' E `(' EA `)'}]\!] = \text{apply(method, arg)}\\
&\quad \text{where } \text{method} = \text{lookup(target, Id)}\\
&\qquad\qquad \text{arg} = [\text{target} \bullet \mathcal{EA}[\![\texttt{EA}]\!]]\\
&\qquad\qquad \text{target} = \text{lookup}(\mathcal{E}[\![\texttt{E}]\!], \text{`class'})\\
&\\
&\mathcal{E}[\![\texttt{`fun' Id `(' EA `)'}]\!] = \text{apply(method, arg)}\\
&\quad \text{where } \text{method} = \text{lookup(this, Id)}\\
&\qquad\qquad \text{arg} = [\text{this} \bullet \mathcal{EA}[\![\texttt{EA}]\!]]\\
&\qquad\qquad \text{this} = \text{value(`this')}\\
&\mathcal{E}[\![\texttt{`static-fun' Id `(' EA `)'}]\!] = \text{apply(method, arg)}\\
&\quad \text{where } \text{method} = \text{lookup(class, Id)}\\
&\qquad\qquad \text{arg} = [\text{class} \bullet \mathcal{EA}[\![\texttt{EA}]\!]]\\
&\qquad\qquad \text{class} = \text{value(`class')}\\
&\\
&\mathcal{EA}[\![\texttt{E EA}]\!] = [\mathcal{E}[\![\texttt{E}]\!] \bullet \mathcal{EA}[\![\texttt{EA}]\!]]\\
&\mathcal{EA}[\![]\!] = []
\end{aligned}
$$

As chamadas de funções estáticas passam a própria classe como primeiro parâmetro. Isso é feito para manter uma certa semelhança com as demais chamadas de funções, as quais passam o próprio objeto *target* como primeiro parâmetro. No entanto, os *bindings* necessários para a aplicação das funções estáticas já são conhecidos no momento de sua declaração, logo é desnecessário passar a classe como parâmetro. Caso o método acessado seja redefinido por uma subclasse, no momento da chamada, o método encontrado pelo componente `lookup` é aquele pertencente à subclasse. Para acessar o método da superclasse, é necessário fazer a busca no pequeno *environment* que constitui o *super*.

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{E `instanceof' Id}]\!] = \text{apply(`=', getType}(\mathcal{E}[\![\texttt{E}]\!]), \text{string(`id'))}\\
&\mathcal{E}[\![\texttt{`this'}]\!] = \text{lookup(`this')}\\
&\mathcal{E}[\![\texttt{`super'}]\!] = \text{lookup(lookup(`this'),`super')}\\
&\mathcal{E}[\![\texttt{E `.' Id}]\!] = \text{lookup}(\mathcal{E}[\![\texttt{E}]\!], \text{Id})
\end{aligned}
$$

Em Java0, é possível checar se um objeto é instância de uma determinada classe. Como toda expressão é avaliada dentro de uma função, o objeto corrente pode ser

acessado por meio do parâmetro *this*. Já o *super*, assim como outros atributos, pode ser encontrado no pequeno *environment* que constitui o objeto corrente.

$\mathcal{E}[\![$`'(' Type ')' E`$]\!]$ = cast($\mathcal{E}[\![$`E`$]\!]$, string(Type))
  where cast (e,t) = choose(apply('=', getType(e), t), e, cast(super e, t))
        super e = lookup(e, 'super')

A única coerção de tipo permitida em Java0 é o uso de um objeto com o tipo de uma de suas superclasses.

$\mathcal{E}[\![$`'new' Id '{' D '}'`$]\!]$ = construct objDef []
  where objDef = object(subclass(lookup(Id), $\mathcal{Z}[\![$`D`$]\!]$ λc o.class(elabRec(c), elabRec(o))))
$\mathcal{E}[\![$`'new' Id '(' EA ')' '{' D '}'`$]\!]$ = construct objDef $\mathcal{EA}[\![$`EA`$]\!]$
  where objDef = object(subclass(lookup(Id), $\mathcal{Z}[\![$`D`$]\!]$ λc o.class(elabRec(c), elabRec(o))))
$\mathcal{E}[\![$`'new' Id '(' EA ')'`$]\!]$ = construct objDef $\mathcal{EA}[\![$`EA`$]\!]$
  where objDef = object(lookup(Id))

construct objDef args = pipe(objDef, λobj.valof(seq(callConstructor, resultis(obj))))
  where callConstructor = call(lookup(obj, 'constructor'), args)

Ao se instanciar objetos, o membro construtor é executado e o resultado da instanciação é o próprio objeto.

$\mathcal{E}[\![$`'new' Id ArrayIndex`$]\!]$ = newArray(reverse $\mathcal{I}[\![$`ArrayIndex`$]\!]$ 0)
  where newArray es v = pipe(hd es, λe.newArray (tl es) (array(0, e, deref(e))))
        newArray [] v = v
              values n = [undefined • values (n -1)]
$\mathcal{E}[\![$`'new' B ArrayIndex`$]\!]$ = newArray(reverse $\mathcal{I}[\![$`ArrayIndex`$]\!]$ 0)
  where newArray es v = pipe(hd es, λe.newArray (tl es) (array(0, e, deref(e))))
        newArray [] v = v
              values n = [getDefault B • values (n -1)]
$\mathcal{I}[\![$`'[' E ']' ArrayIndex`$]\!]$ = [$\mathcal{E}[\![$`E`$]\!]$ • $\mathcal{I}[\![$`ArrayIndex`$]\!]$]
$\mathcal{I}[\![$`'[' E ']'`$]\!]$ = $\mathcal{E}[\![$`E`$]\!]$

A instanciação de arranjos exige que cada valor seja criado e armazenado na posição correspondente. Note que é necessário obter o valor de variáveis com o componente `deref` para determinar a quantidade de valores a serem passados ao componente `array`.

```
getDefault Integer = int('0')
getDefault Boolean = bool('false')
getDefault Float = float('0.0f')
getDefault Double = double('0.0d')
getDefault Long = long('0L')
getDefault Char = char('0̆000')
getDefault String = undefined
getDefault Byte = byte(''0)
getDefault Short = short('0')
```

Para cada tipo primitivo de Java0 é definido um valor *default*, em geral o valor 0.

B.3 Exemplos

A Figura B.1 mostra um trecho de um programa Java0 e sua denotação representada por meio de componentes. Neste exemplo, está representada a semântica de *dynamic dispatch*. A classe A define o método f, cujo valor de retorno é 1. O objeto b é instância de uma subclasse anônima de A, que redefine o método f, cujo resultado passa a ser o valor de f na superclasse mais um. Como os métodos das subclasses prevalecem sobre os métodos da classe no momento da resolução de nomes, o valor impresso por esse trecho é 2.

A Figura B.2 mostra a semântica de uma construção *switch* em Java0. Neste exemplo, os dois *cases* são modelados por tuplas cujo primeiro elemento é o valor do *case* e o segundo é o comando a ser executado. Caso nenhum *op* tenha um valor diferente de "sum" e "sub", o comando correspondente ao caso *default* é executado.

A Figura B.3 mostra a semântica de um exemplo de *inner class* em Java0. A classe B pode ser acessada como um campo da classe A, o que é feito na instanciação do objeto b. Durante a instanciação, primeiro o objeto é criado por meio da declaração de seus campos e depois o método construtor é chamado. O resultado da instanciação é o próprio objeto criado.

```
class A {
    int f () {return 1;}
}

...
A b = new A {
int f () {
return super.f() + 1; }
}

print b.f();

...
```

```
elabRec(bind('A',
   class(elabRec(skip), elabRec(f), static)
))
  where f =bind('f', proc(parametrize([],
                    valof(resultIs(int('1')))
                )))
...
bind('b', object(subclass(lookup('A'), B))
  where f = bind( 'f', proc(parametrize([],
                    valof(resultIs(v))
          )),  type( 'A'))
          B =class( elabRec(skip), elabRec(f))
          v = apply('+',
                    call(lookup(lookup('super'), 'f'), []),
                    int('1'))

output(apply(method, arg))
  where method = lookup(target, 'f'),
          target   = lookup('b')
          args     = []
...
```

Figura B.1: Exemplo de *dynamic dispatch* em Java0.

```
switch(op) {
  case "sum": x = a + b;
              break;
  case "sub": x = a - b;
              break;
  default: x = a;
}
```

```
escape(
   trap(lookup('op'), [sum • sub], default)
  'break'
)
  where sum     = (string('sum'),
                   seq(assign(x, apply('+', a, b)),
                   jump('break')))
          sub     = (string('sum'),
                   seq(assign(x, apply('-', a, b)),
                   jump('break'))
          default = assign(x, b)
          a       = lookup('a')
          b       = lookup('b')
          x       = lookup('x')
```

Figura B.2: Exemplo de semântica de *switch* de Java0.

class A {
  class B {
  ...
  }
}
...
 	
A.B b = **new** A.B()
	elabRec(bind('A',
	  class(elabRec(B), elabRec(skip), static)
	))
	  where B = bind('B',
	  class(elabRec(...), elabRec(...), static))
	...
	
	bind('b', build, type('A.B'))
	  where build = pipe(AB, λo.valof(seq(cons, resultIs(o))))
	  AB = object(lookup(lookup('A'), 'B'))
	  cons = call(lookup(o, 'constructor'), [])

Figura B.3: Exemplo de *inner class* em Java0.

Apêndice C

Implementação de Componentes de Semântica Denotacional

Este apêndice apresenta a implementação de uma biblioteca de componentes de semântica denotacional. São apresentados os domínios semânticos, as equações que definem os componentes e um conjunto de funções auxiliares. O objetivo dessa apresentação é fornecer uma visão mais aprofundada da semântica dos componentes denotacionais. Para o leitor das definições semânticas, este apêndice apresenta uma definição formal da semântica dos componentes, resolvendo possíveis dúvidas resultantes da leitura do Capítulo 4. Para o usuário avançado que pretenda estender a biblioteca, este apêndice pode ser utilizado como referência para a criação de novos componentes.

Nome	Notação	Significado
Produto	$D_1 \times D_2 \times \cdots \times D_n$	tupla
Sequência	$D*$	tupla de tamanho indefinido
Soma	$D_1 + D_2 + \cdots + D_n$	alternativas de domínios

Tabela C.1: Notação utilizada na definição dos domínios semânticos.

O λ-cálculo utilizado como notação e a escolha dos domínios semânticos são baseados em Gordon (1979). A Tabela C.1 apresenta a notação utilizada para definir domínios. O λ-cálculo utilizado inclui noções de tipos e subtipos. Considere o exemplo da Tabela C.2. Nesse exemplo, o domínio C é formado pela união dos domínios A e B. Consequentemente, os valores a e b possuem o tipo A e o tipo C. De forma análoga, os valores 1 e 2 possuem o tipo B e o tipo C.

Algo semelhante ocorre com os domínios `Ev`, `Bv` e `Number` presentes na biblioteca. Os valores numéricos do domínio são também considerados valores básicos e valores expressáveis. Dessa forma, as equações podem trabalhar apenas no nível mais alto,

Domínios	Exemplos de Valores	
$A = \{a, b\}$	Valor	Tipos
$B = \{1, 2\}$	a	A, C
$C = A + B$	$(a, 2)$	D
$D = A \times B$	1	B, C
$E = A^*$	$[a, b, a]$	E

Tabela C.2: Exemplo de domínios semânticos e os tipos associados aos seus valores.

manipulando valores primitivos diretamente apenas quando necessário. A passagem de um valor numérico a uma equação que manipula valores expressáveis é um exemplo de injeção, a qual ocorre de forma implícita nas equações da biblioteca. No entanto, a operação contrária, chamada de projeção, ocorre de forma explícita. A projeção, como pode ser visto na Figura C.1, é feita por meio de casamento de padrões. A falha do casamento feito em um parâmetro resulta em um erro, o que pode ocasionar o fim abrupto da execução. No entanto, quando o casamento de padrões é realizado no corpo de uma equação, é possível definir uma alternativa executada apenas quando o casamento falha. Além disso, pode-se utilizar algumas funções de teste de tipo para tornar as projeções e o tratamento de erros mais legível.

Casamento de padrões no corpo da função:

$$f\,a = a = (x, y) \to x, f'\,a$$

Casamento de padrões no parâmetro da função:

$$g = \lambda(x, y).x$$

$$h(x, y) = x$$

Testes de tipo:

$$i = \lambda a.D?; \lambda(x, y).x$$

$$j = \lambda a.isD\,a \to \lambda(x, y).x, j'\,a$$

Figura C.1: Exemplo de projeção de domínios. As funções f e j fornecem alternativas no caso de falha do casamento de padrões. Nessa mesma situação, as funções g, h e i acusam erro.

C.1 Domínios Semânticos

Esta seção apresenta os domínios semânticos cujos valores são manipulados pelos componentes de semântica denotacional. As letras minúsculas que precedem alguns dos domínios apresentados a seguir são utilizadas nas equações de forma a melhorar a legibilidade. Por exemplo, a letra b é utilizada para nomear variáveis booleanas. Também com o objetivo de melhorar a legibilidade, em várias equações o valor `undefined` é representado pela letra ϕ.

	Ans	= [{error,stop} × Ev × Env × Store]	
		+ [Ev × Env × Store]	- respostas finais
b ∈	Bool	= {bool,false}	- valores booleanos
n ∈	Number	= Int + Float + Double + Long	
		+ Short + Byte	- valores numéricos
	Bv	= Number + Bool + String + Char	
		+ {undefined}	- valores básicos
	Array	= [Loc* × Number × Number]	- arranjos
l,i ∈	Id	= String	- identificadores
o ∈	Op	= {+,-,×,÷,=,!=,>-<}	- operadores básicos
c ∈	Class	= Env	- classes
o ∈	Object	= Env	- objetos
	Rv	= Bv + Array + Record + Object	- *R-values*
	File	= Rv*	- arquivos
	Scope	= (Env,Env) →Env	- escopo dinâmico ou estático
	Dv	= Loc + Rv + Proc + Q	
		+ Xd + Scope + Class	- valores denotáveis
	Sv	= File + Rv	- valores armazenáveis
e,v ∈	Ev	= Dv	- valores expressáveis
r ∈	Env	= [Id + {type} × Id] →[Dv + {unbound}]	- *environments*
l ∈	Loc	= [{const,var} × address] + {input,output}	- *locations*
s ∈	Store	= [{level,new} + Loc] →[Sv + {unused}]	- *stores*
	Adress	= int × int	- endereços do *store*
q ∈	Q	= Ev →Env →Store →Ans	- continuações
p ∈	Proc	= (Ad*, Env →Cc →Ev* →Store →Ans)	- procedimentos
t ∈	Type	= Type	- tipos
m ∈	Mode	= {ref,valueAndResult,value,constant,name}	- modos de passagem de parâmetros
a ∈	Ad	= Mode × Id	- parâmetros formais

C.2 Interfaces dos Componentes

Os componentes possuem cinco interfaces: programas, comandos, expressões, declarações e coletores de rótulos. Por meio da abstração de parâmetros, tem-se também os componentes que denotam procedimentos. Estes recebem um valor expressável como parâmetro do procedimento. Passado esse parâmetro, a equação restante possui a mesma interface do componente abstraído, comando, expressão ou declaração. As interfaces são:

Programas - P ∈ Pd: File →Env →Store →Ans
Comandos - C ∈ Cd: Env →Q →Store →Ans
Expressões - E ∈ Ed: Env →Q →Store →Ans
Declarações - D ∈ Dd: Env →Q →Store →Ans
Coletores de Rótulos - J ∈ Jd: Env →Q →Env

A interface dos parâmetros dos componentes são identificados pelas letras P, C, E, D e J, revelando assim as interfaces a serem implementadas por esses parâmetros. No entanto, alguns parâmetros podem ser comandos, expressões ou declarações. Nesses casos, utiliza-se a letra X e a interface Xd para representar parâmetros genéricos. Por exemplo, considere as equações de tipo dos componentes `choose` e `bind`:

$$\text{choose} : (\text{Ex}, x, x) \to x, x \in \text{Xd}$$

$$\text{bind} : (\text{Id}, (\text{Xd} + \text{Proc})) \to \text{Dd}$$

Os dois últimos parâmetros do primeiro componente possuem o tipo Xd. O tipo de `choose` depende do tipo de seus dois últimos parâmetros. Observe que estes devem possuir o mesmo tipo e que este deve ser Ex, Cd ou Dd. O componente `bind` recebe dois parâmetros. O primeiro deve ser um identificador, enquanto o segundo pode ser um procedimento, um comando, uma expressão ou uma declaração. No entanto, o tipo do componente `bind` não depende de seus parâmetros.

Para que seja possível implementar diferentes ordens de avaliação por meio de `pipe`, todo componente que recebe uma expressão como parâmetro deve também poder receber o valor dessa expressão diretamente. Por isso foi criado o tipo Ex, o qual compreende valores Ev e denotações de expressões e é o tipo do primeiro parâmetro de `choose`.

Por convenção, os parâmetros de um componente visíveis ao usuário são recebidos como uma tupla e as informações de contexto são parâmetros adicionais. Com isso, fica explícita a natureza de cada parâmetro das equações. A única exceção é o componente `run`, o qual recebe informações de contexto explicitamente nas definições semânticas.

No entanto, estas não são passadas por meio de uma tupla, de modo a evidenciar sua natureza e manter o padrão dos outros componentes.

C.3 Componentes

C.3.1 Programas

```
run : Cd →Pd
run C i r s = C r q s′
   where q = (λe r s.(stop, e, r, s))
         s′ = [i,[]/input,output]
```

O componente `run` modela a execução de um programa. A resposta final é obtida por meio da execução do comando C com entrada `i`, *environment* `r` e *store* `s` acrescido da entrada e da saída. Ao fim da execução, o componente `run` retorna uma tupla contendo contendo: um indicador de fim normal da execução, um valor e *environment* e *store* válidos no momento em que a execução termina.

C.3.2 Comandos

```
skip : Cd
skip r q s = q φ r s
```

O componente `skip` não faz qualquer alteração no contexto. Além disso, o valor `undefined`, representado por ϕ, é passado à continuação, visto que a execução de `skip` não resulta em um valor.

```
assign : (Ex, Ex) →Cd
assign (E1,E2) r q = eval E1 r;Loc?;λl.eval E2;deref';Rv?update l q
```

O componente `assign` recebe duas expressões como parâmetro. A primeira modela o lado esquerdo da atribuição e é avaliada para se obter o *location* onde o valor deve ser armazenado. A segunda expressão modela o lado direito e, caso seu resultado seja um *location*, o valor associado é buscado no *store*. O valor do lado direito é então armazenado no *location* do lado esquerdo.

```
output : Ex →Cd
output E r q = eval E r;deref';Rv?;λe r s.q φ r s[[e • s output]/output])
```

O componente `output` avalia a expressão passada como parâmetro e armazena seu valor na saída, buscando-o no *store* se necessário.

```
seq : (Cd,Cd) →Cd
seq (C1,C2) r q = C1 r;λe r.C2 r q
```

O componente `seq` faz o sequenciamento de dois comandos. O componente primeiro executa C_1 e depois C_2.

```
valof : Cd →Ed
valof C r q = C [(λe r′.q e r)/"resuItIs"] err
```

O componente `valof` transforma o comando C em uma expressão. Para isso, o resto do programa é colocado no *environment*, pronto para ser usado por `resultIs`.

```
resultIs : Ex →Cd
resultIs E r q = lookup' r 'resultIs';λq′r′.eval E r′;deref';q′
```

O componente `resultIs` passa o valor resultante da avaliação da expressão E à continuação associada ao nome 'resultIs'. Essa associação é feita pelo componente `valof`.

C.3.3 Fluxo de Controle

```
choose : (Ex, x, x) →x, x ∈ Xd
choose (E,X1,X2) r q = eval E r;deref';Bool?;λb r.cond(eval X1 r q, eval X2 r q) b
```

O componente `choose` modela execução condicional. A avaliação de E deve resultar em um valor booleano. Caso este seja verdadeiro, X_1 é executado, caso contrário X_2 é executado.

```
loop : (Ex, Cd, Ex) →Cd
loop (E1, C, E2) = fix λw.choose(eval E1, seq(C,choose(eval E2, w), skip), skip)
```

O componente `loop` modela laços. Enquanto a avaliação das expressões E_1 e E_2 resulta em verdadeiro, o comando C é executado. Primeiro, a expressão E_1 é testada. Se seu valor é verdadeiro, o comando C é executado. Depois, E_2 é testada, e, se seu valor for verdadeiro, o laço é repetido. Há também duas versões reduzidas deste componente, as quais recebem apenas uma expressão. A outra é substituída pelo valor verdadeiro.

loop : (Ex, Cd) →Cd
loop (E, C) = loop(E, C, bool 'true')

loop : (Cd, Ex) →Cd
loop (C, E) = loop(bool 'true', E, C)

for : (Xd, Ex, Xd, x) →x, x ∈ Xd
for(X_1, E, X_2, X_3) = seq(accum(X_1), loop(E, seq(X_3, X_2), true))

A primeira forma do componente `for` corresponde ao laço *for* de linguagens como Java e C++. Os quatro parâmetros desse componente são em ordem: um componente a ser executado antes da primeira iteração, uma expressão a ser usado como teste do laço, um componente a ser executado após cada iteração e o corpo do laço. Como é comum que o primeiro parâmetro seja uma declaração, sua contribuição para o *environment* é acumulada dentro do componente. Se algum outro parâmetro for uma declaração, o acúmulo deve ser feito explicitamente.

For = Env →Proc →Q →Store →Ans
for : (Id, For, Cd) →Cd
for(I, F, C) r q = accum(bind(I, ref'(ϕ))) r; λe r′. F r′p q
  where p = λe r′q′.assign(I, e) r′q′

A segunda forma do componente `for` corresponde ao laço *for* da linguagem ALGOL 60. O componente recebe três parâmetros. O primeiro deles é um identificador a ser utilizado como variável de controle. O segundo parâmetro modela a iteração. Por fim, o terceiro parâmetro modela o corpo do laço. Durante a execução do componente, o corpo do laço é transformado em um procedimento que armazena o valor recebido na variável de controle.

singleStep : Ex →For
singleStep(E) r p q = eval E r;λe r′.p e r′q

O componente `singleStep` modela um único passo do laço. O valor da expressão `E` é passado ao corpo do laço e este é executado.

while : (Ed, Ed) →For
while(E_1, E_2) = fix λf r p q.E_1 r;λe$_1$.E_2;deref';Bool?;λb r′.cond(p (f r′p q) e, q ϕ r′) b

O componente `while` executa o corpo do laço enquanto o valor de E_2 for verdadeiro. A cada passo, o valor de E_1 é atribuído à variável de controle.

```
range : (Ed, Ed, Ed) →For
range(E1, E2, E3) r p q = eval E1 r;Num?;λn1 r1.step (E2, E3) r1 p q n1
   where step (w2, w3) r p q n1 = w2 r;Num?;λn2 r2.w3 r2;Num?;λn3 r3.(n1 - n3) × (sign n2) < 1
                                  →p n1 r3 (λe r′.step (w1, w2) r′p q)
                                  , q φ r3
```

O componente `range` percorre o intervalo entre o valor de E_1 e o valor de E_3. A cada iteração, o valor é passado ao corpo do laço. O valor utilizado em cada iteração é o resultado de somar o valor de E_2 ao valor utilizado na iteração anterior. A primeira iteração utiliza o valor de E_1, e, quando o valor calculado ultrapassa o valor de E_3, a execução do laço termina.

```
seq : (For, For) →For
seq(F1, F2) r p q = F1 r p;λe r′.F2 r′p q
```

Esta forma do componente `seq` faz o sequenciamento de dois iteradores. Executa-se primeiro F_1 e depois F_2.

C.3.4 Sequenciadores

```
abort →Cd
abort r q s = err φ r s
```

O componente `abort` termina a execução do programa abruptamente, retornando o valor `undefined` e o *environemnt* e o *store* correntes.

```
jump : (Id) →Cd
jump I r q = lookup' I r;Q?;λq′.q′φ
```

O componente `jump` interrompe o fluxo de controle e executa a continuação associada ao rótulo I.

```
escape : (Cd, Id, Scope) →Cd
escape (C,I,scope) r q = C r[λe r′.q e (scope (r,r′))/I] q

escape : (Cd, Id) →Cd
escape (C,I) r q = escape (C,I,getScope r) r q
```

O componente `escape` cria uma saída para o comando C. Para isso o rótulo I é associado à continuação corrente.

```
trap : (Ex, [(Ex,x)], x) →x, x ∈ Xd
trap (E, L, X) r q = (unzip L) = (E*,x*) →evalList E* r;λe* r′.eval E r′;find q X x* e*
where find q X x* e* e r = (null e*) → eval X r q,
                                (hd e*) = e → eval (hd x*) r q,
                                              find q X (tl x*) (tl e*) e r
```

O componente `trap` modela uma seção de código com várias entradas. A expressão E é avaliada e seu valor é comparada com uma série de valores associados a comandos. Quando a comparação é bem sucedida, o comando associado é executado. Caso todas as comparações falhem, o comando C é executado.

```
throw : (Ex) →Cd
throw (I, E) r q = eval E r;λe r′.lookup' 'catch' r′;Q?;λq′.q′e
```

O componente `throw` é semelhante ao componente `jump`, realizando um salto. No entanto, aquele permite que um valor seja passado à continuação, o que permite informar as condições que motivaram o salto.

```
catch : (Cd, Proc) →Cd
catch (C, P) r q = C r[catch′/'catch']
   where catch′  e r′q′ = P e r q
```

O componente `catch` é usado em conjunto com `throw` para modelar o tratamento de exceções. Aquele associa o procedimento P a um rótulo e este realiza o salto. O componente `throw` passa um valor ao procedimento, o qual pode ser utilizado para decidir se a exceção deve ser tratada ou lançada novamente.

C.3.5 Gerência de Memória

```
s0 = λl.l = level →1, l = new →0, unused
```

O *store* inicial, $\mathtt{s}_0$ retorna `unused` para todos os *locations*. Inicialmente, o nível da pilha é 1 e o primeiro *location* livre é 0.

```
push : Cd
push r q s = q φ r (λl.l = level →(s level) + 1, s l)
```

O componente `push` cria um novo nível na pilha do *store*.

```
pop : Cd
pop r q s = (s level) = 1 → err φ r s,
                              q φ r (λl.l = level → (s level) - 1,
                                                    l = (_,((s level), _)) →unused,s l)
```

O componente `pop` apaga o nível corrente da pilha do *store*, removendo todos valores associados a esse nível. No entanto, é necessário que pelo menos um nível tenha sido criado com o componente `push`. Caso contrário, a avaliação de `pop` resulta em erro.

```
alloc : Ex →Ed
alloc E r q = eval E r;λ e r′.ref (λe′r″.q e′r′) e r′[heap/'allocation']
```

O componente `alloc` armazena o valor resultante da avaliação de E em uma nova posição no *heap* e passa o *location* associado para a continuação.

```
free : Ex →Cd
free E r q = eval E r;Loc?;λ(cv,(0,a)) r′s.q φ r′s[unused/(_,(0,a))]
```

O componente `free` avalia a expressão E e libera a posição associada ao *location* resultante no *store*.

```
const : Ex →Ed
const E r q = ref(E) r;λl.q (const' l)
```

O componente `const` utiliza o componente `ref` para armazenar o resultado da avaliação da expressão E no *store*(no *stack*). No entanto, o *location* associado é transformado para que a variável obtida seja constante.

```
ref : Ex →Ed
ref E r q = eval E r;ref';q
```

O componente `ref` avalia a expressão E, armazena o valor resultante no *store*(no *stack*) e passa o *location* associado para a continuação.

```
deref : Ex →Ed
deref E r q = eval E r;deref';q
```

O componente `deref` avalia a expressão E e passa para a continuação o valor armazenado no *store*. Para isso é utilizada a função auxiliar chamada de `deref'`. Caso o resultado da expressão seja um valor e não um *location*, este é passado para a continuação diretamente pela função auxiliar.

C.3.6 Expressões

Os componentes a seguir são responsáveis por criar valores de tipos primitivos.

```
byte : String →Ed
short : String →Ed
int : String →Ed
long : String →Ed
float : String →Ed
double : String →Ed
char : String →Ed
string : String →Ed
bool : String →Ed
```

Cada valor criado possui informação de tipo, a qual pode ser verificada por meio do componente `getType`.

```
getType(Ex+Id) →Ed
getType I r q = (r (type,I)) = unbound →err 'unknown' r s, q (r (type,I)) r s
getType E r q = eval E r;deref';λe.isByte e →q 'byte',
                                   isShort e →q 'short',
                                   isInt e →q 'int',
                                   isLong e →q 'long',
                                   isDouble e →q 'double',
                                   isChar e →q 'char',
                                   isString e →q 'string',
                                   isBool e →q 'bool',
                                   isEnv e →(e 'class') = unbound →'unkown', (e'class')
```

O componente `getType` é utilizado para recuperar o tipo de uma variável, valor primitivo ou objeto.

```
type : String →String
type t = t
```

O componente `type` é utilizado para melhorar a legibilidade de testes de tipo genéricos.

```
input : Ed
input r q s = null(s input) →err φ r s,q hd(s input) r s[tl(s input)/input]
```

O componente `input` lê um valor da entrada e o passa para a continuação.

```
lookup : [Id+(Ex,Id)+(Id,Id)] →Ed
lookup I r q = lookup' I r;λe r′.isnoteval e →eval e r′q, q e r′
lookup (E, I) r q = eval E r;deref';isEnv?;λr′.lookup' I r′;λe r″.q e r
lookup (I, I) r q = lookupi' I r;deref';isEnv?;λr′.lookup' I r′;λe r″.q e r
```

O componente `lookup` é responsável por obter o valor ligado a um identificador. Caso o componente receba apenas um identificador, o *environment* corrente é utilizado na busca. Nesse caso, é utilizada uma função auxiliar com o mesmo nome do componente. Caso contrário, o primeiro parâmetro deve ser um objeto, registro ou classe – ou um identificador associado a um destes valores. O segundo parâmetro é o identificador utilizado na busca feita sobre o primeiro parâmetro. Assim pode-se, por exemplo, buscar o valor associado a um campo de um registro utilizando como parâmetros o próprio registro e o identificador do campo desejado.

```
record : Dd →Ed
record(D) r q = D r; λe r′.q r′r
```

O componente `record` executa a declaração D e passa para a continuação as ligações produzidas. Estas compõem o registro.

```
array : (Ex, Ex, Ex*) →Ed
array(E1, E2, E*) r q =   eval E1 r; isNum?; λn1.eval E2; isNum?;λn2.checkBoundaries;
                          evalList E*;storeValues [];λl*.q (l*,n1,n2)
where checkBoundaries q″ = n2 < n1 → err, size E* ≠ (n2 - n1) →err, q″
      storeValues′ l* e* q′ = null e*  → q′l*,
                                         deref' (ref';λl.storeValues′[l* • l] (tl e*) q′) (hd e*)
```

O componente `array` constrói um arranjo. Seus dois primeiros parâmetros determinam os limites do arranjo. O terceiro parâmetro é uma lista de expressões. Estas são avaliadas e seus valores são armazenados nas posições do arranjo. A posição de cada expressão na lista determina a posição do valor associado no arranjo.

```
index : (Ex,Ex) →Ed
index (E1, E2) r q = eval E1 r;deref';Array?;λe.eval E2;Num?;subscript e q
```

O componente `index` é utilizado para acessar as posições de um arranjo. O primeiro parâmetro é o arranjo em si, enquanto o segundo é o número da posição desejada.

O ∈ Operator = (Ev* →Ev + Error)
apply : (Operator, Ex*) →Ed
apply (O, E*) r q = eval E* [] (λe*.Ev?;O e* = error →err ϕ, q (O e*)) r
  where eval′e*$_1$ e*$_2$ q = null e*$_1$ →q e*$_2$, eval (hd e*$_1$);deref';λv.eval′(tl e*$_1$) [e*$_2$ • v] q

O componente `apply` aplica uma operação definida pelo usuário a uma lista de valores. A operação deve retornar um único valor. Algumas operações predefinidas estão disponíveis:

apply : (Op,Ex,Ex) →Ed
apply(+,E$_1$,E$_2$) = apply(λe*.(el 1 e*) + (el 2 e*),[E$_1$ • E$_2$])
apply(-,E$_1$,E$_2$) = apply(λe*.(el 1 e*) - (el 2 e*),[E$_1$ • E$_2$])
apply(×,E$_1$,E$_2$) = apply(λe*.(el 1 e*) × (el 2 e*),[E$_1$ • E$_2$])
apply(÷,E$_1$,E$_2$) = apply(λe*.(el 2 e*) = 0 →error, (el 1 e*) ÷ (el 2 e*),[E$_1$ • E$_2$])
apply(=,E$_1$,E$_2$) = apply(λe*.(el 1 e*) = (el 2 e*),[E$_1$ • E$_2$])
apply(!=,E$_1$,E$_2$) = apply(λe*.(el 1 e*) != (el 2 e*),[E$_1$ • E$_2$])

C.3.7 Abstração de Tipos

class : (Dd, Dd, Scope) →Ed
class (D$_c$, D$_o$, scope) r q = D$_c$ r; λe c.q c[c/'class'][constructor/'constructor'] r
  where constructor = λr′.D$_o$ (scope (r,r′))[c][c/'class']

class : (Dd, Dd) →Ed
class (D$_c$, D$_o$) r q = class (D$_c$, D$_o$, getScope r) r q

O componente `class` recebe duas declarações como parâmetros. A primeira corresponde à declaração dos membros e atributos da classe, chamados de estáticos. A segunda corresponde à declaração de membros e atributos dos objetos. O componente `class` executa a declaração D$_c$ e passa para a continuação o pequeno *environment* resultante acrescido de *bindings* para a declaração D$_o$, responsável por construir o objeto, e para a própria classe.

subclass : (Ex, Ex) →Ed
subclass (E$_1$, E$_2$) r q = eval E$_1$ r; Class?; λc$_1$ r′.eval E$_2$ r′[c$_1$];Class?;
                                λc$_2$.q c$_1$[c$_2$][constructor/'constructor'] r
where constructor r q = object(c$_1$) r; λo$_1$ r′.object(c$_2$) r′[o$_1$];
                        λo$_2$.q o$_1$[o$_2$][o$_1$/'super'] r

O componente `subclass` estabelece a relação de sub-classe entre duas classes. O parâmetro E_1 corresponde à superclasse, enquanto E_2 corresponde à subclasse. O resultado é a classe representada por E_2 acrescida das declarações da superclasse e um *binding* para acesso à parte do objeto associada à superclasse.

```
object : Ex →Ed
object E r q = eval E r;Class?;λc r′.(c 'constructor') r′;λe o.q c[o][o/'this']
```

O componente `object` é responsável por instanciar objetos. A expressão E é avaliada e a classe resultante é utilizada para se obter as declarações do objeto. O resultado é a composição dos pequenos *environments* da classe e do objeto.

C.3.8 Declarações

```
accum : Dd →Dd
accum(D) r q = D (λv′r′.q v′r[r′])
```

Os vários componentes que modelam declarações passam para a continuação apenas as ligações que produzem, abandonando o *environment* que recebem. Isso facilita a modelagem de declarações isoladas, como ocorre, por exemplo, na instanciação de objetos. O componente `accum` executa a declaração D e passa para a continuação o *environment* recebido acrescido das ligações produzidas por D.

```
bind : (Id, Xd) →Dd
bind(I, X) r q = eval X r;λx r′.q φ [f x/I]
   where f x = isClass x →x['classname'/I], x

bind : (Id, Xd, Type) →Dd
bind(I, X, T) r q = bind(I, X) r; λe r′.q e r′[T/(type,I)]
```

O componente `bind` é responsável por criar ligações no *environment*. Seus argumentos são o identificador utilizado na ligação e a entidade a ser ligada. Esta pode ser um procedimento ou valor expressável. É possível também associar um tipo T à entidade ligada.

```
elabSeq : (Dd, Dd) →Dd
elabSeq(D1,D2) r q = D1 r;λe1 r1.D2 r[r1];λe2 r2.q φ r1[r2]
```

O componente `elabSeq` é responsável por combinar declarações sequencialmente. Executa-se primeiro a declaração D_1 e depois D_2. Note que a *environment* utilizado na execução da declaração D_2 contém as ligações produzidas por D_1. Logo, D_2 pode utilizar estas ligações ou alterá-las.

elabCol : (Dd, Dd) →Dd
elabCol(D_1,D_2) r q s = D_1 r;λe_1 r_1.D_2 r;λe_2 r_2.q ϕ r_1[r_2]

O componente `elabSeq` é responsável por combinar declarações em ordem indeterminada. A ordem utilizada na implementação é a mesma de `elabSeq` por simplicidade, mas cada declaração é executada utilizando apenas o *environment* recebido por `ElabCol`. Dessa forma, espera-se que nenhuma das duas declarações pode utilizar ou alterar ligações produzidas pela outra.

elabRec : (Dd,Dd) →Dd
elabRec(D_1, D_2) r q s = elabRec(elabSeq(D_1,D_2))

O componente `elabRec` é responsável por combinar declarações de forma recursiva. Devido a dificuldades na combinação de dois *stores*, utiliza-se o componente `elabRec` com apenas um parâmetro. Dessa forma, as declarações são feitas utilizando um *environment* que contém as ligações que estas produzem. Internamente, utiliza-se `elabSeq` para combinar as duas declarações.

elabRec : Dd →Dd
elabRec(D) r q s = y = error →err ϕ r s, q ϕ r′s′
  where y = D r[r′] (λe r″s″.(e,r″,s″)) s′
    (e′,r′,s′) = y

O componente `elabRec` é responsável por executar uma declaração de forma recursiva. Dessa forma, a declaração é feita utilizando um *environment* que contém as ligações que esta produz.

addEnv : (Dd, x) →x, x ∈ Ed + Cd
addEnv (D,X) r q = D r;λe r′.X r′q

O componente `addEnv` executa X utilizando o *environment* produzido por D.

```
O ∈ ImportOption = [{Opaque} X Id*] + {Transparent}
O ∈ ExportOption = [{Encapsulated} X Id*] +{Visible}
makeAbstraction : (Xd, IOp, EOp) →Xd
makeAbstraction(X, O1, O2) r q = X (imported r); λe r′.q e (exported r′)
  where imported r″= O1 = (Opaque, i*) →filter r″i*, r″
        exported r″= O2 = (Encapsulated, i*) →r[subEnv r″i*], r″
        subEnv r″i* = null i* →φ, (filter r″(tl i*))[r″(hd i*)/(hd i*)]
```

O componente `makeAbstraction` é responsável por modelar módulos. O componente X é executado utilizando apenas as ligações importadas, e, entre as ligações produzidas por X, apenas aquelas exportadas são visíveis fora do módulo. Há duas opções de importação e duas opções de importação. A opção `Transparent` importa todas as ligações vindas de fora do módulo, enquanto `Visible` exporta todas as ligações produzidas por X. As opções `Opaque` e `Encapsulated` permitem determinar listas de importação e exportação respectivamente.

```
useStaticScope : Xd →Xd
useStaticScope X r q = X r[static/'scope'] q
```

O componente `useStaticScope` é responsável por executar X utilizando escopo estático.

```
useDynamicScope : Xd →Xd
useDynamicScope X r q = X r[dynamic/'scope'] q
```

O componente `useStaticScope` é responsável por executar X utilizando escopo dinâmico.

```
static : Scope
static (r,r′) = r

dynamic : Scope
dynamic (r,r′) = r′
```

As funções `static` e `dynamic` podem ser utilizadas para indicar que um determinado componente deve usar escopo estático ou dinâmico respectivamente. Os componentes que podem ter seu escopo configurado são: `bind`, `class`, `addLabel`, `escape` e `catch`.

C.3.9 Saltos

```
addLabel : (Id, Cd, Scope) →Jd
addLabel (I, C, scope) r q = [λe r′.C scope(r,r′) q/I]

addLabel : (Id, Cd) →Jd
addLabel (I, C) r q = addLabel (I,C, getScope r) r q
```

O componente `addLabel` é responsável pela coleta de rótulos. Sua semântica é retornar uma ligação entre o rótulo I e o comando C.

```
skipLabel →Jd
skipLabel r q = []
```

O componente `skipLabel` é utilizado para modelar casos em que uma determinada construção não possui rótulos.

```
elabSeq : (Jd, Jd) →Jd
elabSeq(J1,J2) r q = r[r1][r2]
   where r1 = J1 r q
         r2 = J2 r q
```

O componente `ElabSeq`, ao receber dois coletores de rótulos, executa as duas coletas e retorna o *environment* original acrescido dos rótulos coletados. Observe que, se ambos os coletores coletam rótulos com o mesmo identificador, o rótulo coletado por J_2 sobrescreve aquele coletado por J_1.

```
elab : (Jd, Cd) →Jd
elab(J, C) r q = J r (λe r′.C r′q)
```

O componente `elab` executa o coletor de rótulos J, informando que, após o salto, devem ser executados o comando C e o resto do programa representado pela continuação. Esse componente é essencial para a coleta de rótulos, pois é por meio dele que o resto do programa é comunicado ao componente `addLabel`.

```
addEnv : (Jd,Cd) →Cd
addEnv (J,C) r q = C r[r′] q
   where r′ = J r[r′] q
```

O componente `addEnv` executa o comando C utilizando o *environment* recebido acrescido dos rótulos coletados por J. A definição recursiva é utilizada para permitir

que os rótulos continuem sendo conhecidos após um salto.

C.3.10 Abstração Procedural

```
proc : (Xd + Proc, Scope) →Ed
proc (X, scope) r q = q (λr′.x scope(r,r′)) r
   where x     = isXd X →(args, body), x
         args  = []
         body  = X

proc : (Xd + Proc) →Ed
proc (X) = proc(X, getScope)
```

O componente `proc` é responsável por criar procedimentos. Os procedimentos são uma abstração de comandos, declarações ou expressões. Procedimentos podem ser chamados por meio do componente `call`. Caso o tipo de escopo não seja especificado, o escopo padrão é utilizado. Caso o X seja uma sentença parametrizada, o componente `proc` não precisa fazer nada, posto que `parametrize` cria procedimentos já com o formato necessário.

```
makeClosure : (Dd, x) →x, x ∈ Ed + Cd
makeClosure(D, X) r q = D r (λe r′.X r′(λe′r″.q e′r))
```

A primeira forma do componente `makeClosure` executa o componente `X` utilizando o *environment* recebido acrescido das ligações produzidas por D. Todas as alterações feitas ao *environment* por D e X são descartadas ao fim da execução de X.

```
makeClosure : x →x, x ∈ Ed + Cd
makeClosure(X) r q = X r (λe r′.q e r)
```

A segunda forma do componente `makeClosure` executa o componente `X` utilizando o *environment* recebido. Todas as alterações feitas ao *environment* por X são descartadas ao fim de sua execução.

```
parametrize : (Ad*, Xd) →Proc
parametrize(A*, X) = (A*, λr q v*.addPar (revert (zip A* v*)) X r q)
   where addPar a* X′ = null a* →X′, addPar (tl a*) (body (hd a*) X′)
                 body = λ((m,I),e) X″c′.entry m (λe′r′.X″r′[e′/I] (exit m e c′e′) )
```

O componente `parametrize` é responsável por fazer abstração de parâmetros

em X. O resultado é um procedimento, o qual recebe uma lista de valores, associa-os aos parâmetros formais e executa X. O tipo de passagem de parâmetros altera alguns pontos da semântica, o que pode ser visto nas definições das funções `entry`, `exit` e `evalPar`.

```
isnoteval : Ev →Bool
isnoteval e = false
isnoteval →[Q →Store →Ans] →Bool
isnoteval y = true
```

A passagem de parâmetros por nome permite avaliação preguiçosa dos parâmetros. Para isso, é necessário decidir, no momento de seu uso, se um dado parâmetro já foi avaliado. A função `isnoteval` determina se um parâmetro já foi avaliado.

```
eval : (Xd + Ev + [Q →Store →Ans]) →Ed
eval e r q = isXd e →e r q, isnoteval e →e q, q e r
```

A função `eval` avalia os parâmetros no caso de passagem por nome. Esse componente também é utilizado para se permitir que valores sejam passados no lugar de expressões. Por isso, o parâmetro também pode ser uma expressão.

```
evalPar : Mode* →Ex* →(Ev* →Env →Store →Ans) →Env →Store →Ans
evalPar m e p = evalPar′m e [] p
   where evalPar′ m e es r p = null e →p es,
                               m = name →evalPar′(tl m) (tl e) [(eval (hd e) r) • es] r p,
                               eval (hd e) r (λe′r′.evalPar′(tl m) (tl e) [e′ • es] r′p)
```

A função `evalPar` avalia os parâmetros reais no ponto da chamada do procedimento. Caso a passagem seja por nome, um *closure* é passado.

```
entry  value q = deref';ref';q
entry  valueAndResult = entry value
entry  ref q = Loc?;q
entry  constant q = deref';ref';λl.q (const' l)
entry  name = ref'
```

A função `entry` associa o valor do parâmetro real ao parâmetro formal. No caso de passagens por valor e valor e resultado, o valor do parâmetro real é buscado no *store* e armazenado em uma nova variável associada ao nome do parâmetro formal. No caso de passagem por valor constante, a nova variável é transformada em constante.

No caso de passagem por referência, o parâmetro formal é associado ao *location* do parâmetro real. Por fim, no caso da passagem por nome, o *closure* é armazenado em uma nova variável associada ao parâmetro formal.

```
exit : Mode →Ev →Q →Ev →Env →Store →Context →Ans
exit  valueAndResult e q = cont (update e q)
exit  _ e q e'= q φ
```

A função `exit` é responsável por comunicar alterações do parâmetro formal ao parâmetro real no caso de passagem de parâmetros por valor e resultado.

```
call : (Ex,Ex*) →Cd
call (E1,E2) r q = eval E1;λe.isProc e →e = (a,p) →evalPar (hd (unzip a)) E2 (p r q),
                                                   isXd e →e r q,
                                                   err φ r
```

O componente `call` realiza uma chamada de procedimento. A expressão E_1 é avaliada para se obter o procedimento e E_2 é passada à função `evalPar` para avaliação dos parâmetros e execução do corpo do procedimento. Os modos de passagem associados aos parâmetros são obtidos diretamente da definição do procedimento e passados à função *evalPar*. Nessa definição, encontram-se também os identificadores dos parâmetros formais, esses no entanto são descartados.

C.4 Funções auxiliares

```
err : Ev →Env →Store →Ans
err = λv r s . (error,v,r,s)
```

A função `err` aborta a execução, informando estado de erro, um valor e o *environment* e *store* correntes.

$$\text{cond} : [\text{D X D}] \rightarrow \text{Bool} \rightarrow \text{D}$$
$$\text{cond } (\text{d}_1,\text{d}_2)\ \text{b} = \begin{cases} d_1 & \text{se } b = true \\ d_2 & \text{se } b = false \end{cases}$$

A função `cond` corresponde a d_1 se b for verdadeiro ou d_2 caso contrário. Note que D é um tipo qualquer.

```
ref' : Q →Ev →Env →Store →Ans
ref' q e = new;λl.update l (λe'.q l) e
```

A função `ref'` cria uma nova posição no *store*, armazena nela o valor `e` e passa o *location* associado para a continuação.

```
new : Cd
new q r s = (r 'allocation') = heap →q (const,(0, s new)) r s', q (const,(s level, s new)) r s'
   where s'= λl.l = new →(s new) + 1, s l
```

A função `new` cria uma nova posição no *store* e passa o *location* associado para a continuação. Caso o *heap* esteja configurado como ponto de alocação, a nova posição será criada no *heap*. Caso contrário, ela será criada no *stack*.

```
update : Loc →Q →Ev →Env →Store →Ans
update l q e r s = l = (const, a) →err φ r s,
                   isSv e →q φ r s[e/l], err φ r s
```

A função `update` armazena o valor `e` na posição associada ao *location* `l` no *store*.

```
const' : Loc → Loc
const' (_, a) = (const, a)
```

A função `const` altera o *location* recebido para que este não possa ter seu valor atualizado.

```
deref' : Q →Ev →Env →Store →Ans
deref' q e = isLoc e →cont q e, q e
```

A função `deref'` busca o valor associada a `e` no *store* e o passa para a continuação. Caso `e` não seja *location*, este é passado para a continuação.

```
cont : Q →Ev →Env →Store →Ans
cont q e r s = isLoc e →(s e = unused →err φ r s, q (s e) r s), err φ r s
```

A função `cont` passa para a continuação o valor associado ao *location* `e` no *store*. Caso `e` não seja *location* ou a posição associada não possua um valor, o resultado é um erro.

```
lookup' : Id →Env →(Dv →Env →Store →Ans) →Store →Ans
lookup' i r q s = r i = unbound →err φ r s, q (r i) r s
```

A função `lookup'` é utilizada para buscar o valor associado a um identificador no *environment*.

```
subscript : Array →Num →Env →Q →Store →Ans
subscript (l*,n1,n2) n r q = n < n1 →err φ r s, n > n2 →err φ r s, q (el (n - n1) l*) r
```

A função `subscript` é utilizada para acessar uma posição de um arranjo. Antes do acesso, a posição é testada para garantir que não exceda os limites do arranjo.

```
evalList : Ex* →Env →Store →(Ev* →Env →Store →Ans) →Ans
evalList E* r q = evalList'[] E*
   where evalList'e* E* r q = null E* →q e* r, eval (hd E*) r;λe r'.evalList' [e* • e] (tl E*) r'q
```

A função `evalList` avalia uma lista de expressões e passa seus resultados para a continuação.

```
null : a* →Bool
null [] = true
null _ = false
```

A função `null` verifica se uma lista é vazia.

```
hd : a* →a
hd [x • xs] = x
```

A função `hd` retorna o primeiro elemento de uma lista.

```
el : Int →a* →a
el v xs = v = 0 →hd xs,
          el (v - 1) (tl x)
```

A função `el` retorna um elemento arbitrário de uma lista.

```
tl : a* →a*
tl [] = []
tl [x • xs] = xs
```

A função `tl` retorna todos os elementos de uma lista com exceção do primeiro.

```
size : a* →Num
size xs = size′xs 0
size′[] n = n
size′[x • xs] n = (size′xs 0) + n
```

A função `size` retorna o tamanho de uma lista.

```
zip : a* →b* →(a,b)*
zip [x • xs] [y • ys] = [(x,y) • zip xs ys]
zip [] _ = []
zip _ [] = []
```

A função `zip` recebe duas listas e retorna uma lista de pares. Os pares são formados por um elemento de cada lista.

```
unzip : (a,b)* →(b*,a*)
unzip x = [first x, second x]
   where     first [(x,y) • xys] = [x • first xys]
                        first [] = []
          second [(x,y) • xys] = [y • second xys]
                       second [] = []
```

A função `unzip` realiza a operação contrária a `zip`.

```
revert a* →a*
revert [] = []
revert [x • xs] = [revert xs • x]
```

A função `revert` inverte a ordem dos elementos da lista que recebe.

```
getScope →Env →Scope
getScope r = (r 'scope') = unbound →static, (r 'scope')
```

A função `getScope` retorna a função de escopo configurada no *environment*. Caso nenhuma tenha sido configurada, `static` é retornada.

Apêndice D

Glossário

D.1 Domínios Semânticos

Adress	int × int
	Endereços de baixo nível do *store*.
Ad	Mode × Id
	Parâmetros formais.
Ans	[{error,stop} × Ev × Env × Store] + [Ev × Env × Store]
	Respostas finais.
Array	[Loc* × Number × Number]
	Arranjos.
Bool	{bool,false}
	Valores booleanos.
Bv	Number + Bool + String + Char + {undefined}
	Valores básicos.
Cd	Env →Q →Store →Ans
	Denotações de comandos.
Class	Env
	Classes.

Dd	Env →Q →Store →Ans
	Denotações de declarações.
Dv	Loc + Rv + Proc + Q + Xd + Scope + Class
	Valores denotáveis: são aqueles que podem ser ligados no *environment*.
Ed	Env →Q →Store →Ans
	Denotações de expressões.
Env	[Id + {type} × Id] →[Dv + {unbound}]
	Environments: são tilizados para guardar ligações.
EOp	[{Encapsulated} X Id*] +{Visible}
	Opções de exportação de ligações utilizadas por `makeAbstraction`.
Ev	Dv
	Valores expressáveis: são aqueles que podem ser manipulados por programas.
Ex	Ed + Ev
	Denotações de expressões ou valores expressáveis. Utilizado para possibilitar a definição explícita da ordem de avaliação de expressões por meio do componente `pipe`.
File	Rv*
	Arquivos.
For	Env →Proc →Q →Store →Ans
	Laços *for* de Algol 60.
Id	String
	Identificadores.
IOp	[{Opaque} X Id*] + {Transparent}
	Opções de importação de ligações utilizadas por `makeAbstraction`.
Loc	[{const,var} × address] + {input,output}

	Locations: utilizados como endereços de armazenamento no *store*.
Mode	{ref,valueAndResult,value,constant,name}
	Modos de passagem de parâmetros.
Number	Int + Float + Double + Long + Short + Byte
	Valores numéricos.
Object	Env
	Objetos.
Op	{+,-,×,÷,=,!=,>-<}
	Operadores pré-definidos.
Proc	(Ad*, Env →Cc →Ev* →Store →Ans)
	Procedimentos.
Q	Ev →Env →Store →Ans
	Continuações: utilizadas para modelar fluxo de controle.
Rv	Bv + Array + Record + Object
	R-values: aqueles que podem ser atribuídos a variáveis.
Scope	(Env,Env) →Env
	Escopo dinâmico ou estático.
Store	[{level,new} + Loc] →[Sv + {unused}]
	Stores: modelam mecanismos de armazenamento.
Sv	File + Rv
	Valores armazenáveis: são aqueles que podem ser armazenados no *store*.
Type	String
	Identificadores de tipos.
Xd	Cd + Dd + Ex
	Denotações de expressões, de comandos ou de declarações ou valores expressáveis. Componentes de tipo Xd podem atuar como denotações de expressões, comandos ou declarações.

D.2 Componentes

abort	Cd
	Termina a execução do programa abruptamente, retornando uma tupla em que o primeiro elemento é o código `error`, o segundo é o valor `undefined` e o terceiro e quarto são respectivamente o *environemnt* e o *store* correntes.
accum	Dd →Dd
	Os vários componentes que modelam declarações passam para a continuação apenas as ligações que produzem, abandonando o *environment* que recebem. Isso facilita a modelagem de declarações isoladas, como ocorre, por exemplo, na instanciação de objetos. O componente `accum` executa a declaração D e passa para a continuação o *environment* recebido acrescido das ligações produzidas por D.
addEnv	(Dd, x) →x, x ∈ Ed + Cd
	Executa o segundo parâmetro utilizando o *environment* produzido pela declaração.
	(Jd,Cd) →Cd
	Executa o segundo parâmetro utilizando o *environment* recebido acrescido dos rótulos coletados pelo primeiro parâmetro.
addLabel	(Id, Cd, Scope) →Jd
	Associa o comando ao rótulo e passa a ligação produzida para a continuação. A opção de escopo é utilizado para executar o comando em caso de salto.
	(Id, Cd) →Jd
	Versão do componente `addLabel` utilizando escopo estático.
alloc	Ex →Ed
	Armazena o valor resultante da avaliação da expressão em uma nova posição no *heap* e passa o *location* associado para a continuação.

apply	((Ev*→Ev+Error), Ex*)→Ed
	Aplica uma operação definida pelo usuário a uma lista de valores. A operação deve retornar um único valor.
	(Op,Ex,Ex) →Ed
	Aplica uma operação pré-definida a uma lista de valores. As operações disponíveis são {‘+’,‘-’,‘/’,‘*’,‘=’,‘!=’,‘<’,‘>’}.
array	(Ex, Ex, Ex*) →Ed
	Constrói um arranjo. Seus dois primeiros parâmetros determinam os limites e o terceiro parâmetro é uma lista de expressões que determinam os valores dos elementos.
assign	(Ex, Ex) →Cd
	Modela atribuição. A primeira expressão modela o lado esquerdo da atribuição e é avaliada para se obter um *location*. A segunda expressão modela o lado direito e, caso seu resultado seja um *location*, o valor associado é buscado no *store*. O valor do lado direito é então armazenado no *location* do lado esquerdo.
bind	(Id, Xd+Ev) →Dd
	Instala ligações entre um identificador e um procedimento ou valor expressável no *environment*.
bool	String →Ed
	Transforma a *string* em um valor de tipo *bool*.
byte	String →Ed
	Transforma a *string* em um valor de tipo *byte*.
call	(Ex,Ex*) →Cd
	Realiza uma chamada de procedimento. O primeiro parâmetro é avaliado para se obter o procedimento e o segundo é uma lista de expressões que constituem os parâmetros reais.
catch	(Cd, Proc) →Cd

	Instala o procedimento recebido no *environment* para ser utilizado pelo componente `throw` e executa o comando. Caso o procedimento seja executado, sua continuação é a continuação do componente `catch`.
char	String →Ed
	Transforma a *string* em um valor de tipo *char*.
choose	(Ex, x, x) →x, x ∈ Xd
	Modela execução condicional. A avaliação da expressão deve resultar em um valor booleano. Caso este seja verdadeiro, o segundo parâmetro é executado, caso contrário o terceiro parâmetro é executado.
class	(Dd, Dd) →Ed
	Cria uma classe a partir de duas declarações. A primeira define os atributos estáticos e a segundo define atributos do objeto.
const	Ex →Ed
	Armazena o valor da expressão na pilha do *store* e e passa o *location* associado para a continuação. No entanto, esse *location* é transformado para que a variável obtida seja constante.
deref	Ex →Ed
	Avalia a expressão e, caso o valor resultante seja um *location*, passa para a continuação o valor armazenado no *store*. Caso contrário, o próprio valor da expressão é passado.
double	String →Ed
	Transforma a *string* em um valor de tipo *double*.
dynamic	Scope
	Modela escopo dinâmico. Utilizado como parâmetro dos componentes `bind`, `addLabel`, `escape` e `catch`.
elabCol	(Dd, Dd) →Dd

	Combina declarações em ordem indeterminada. Não deve haver dependência entre as declarações.
elabRec	(Dd,Dd) →Dd
	Combina declarações de forma recursiva. Ambas as declarações podem utilizar ligações produzidas pela outra.
	Dd →Dd
	Executa uma declaração de forma recursiva. A declaração pode usar as ligações que produz.
elabSeq	(Dd, Dd) →Dd
	Combina declarações sequencialmente. A segunda declaração é executada após a primeira e pode utilizar ligações por esta.
	(Jd, Jd) →Jd
	Combina os rótulos coletados por dois coletores. Os rótulos do segundo têm precedência sobre aqueles do primeiro.
elab	(Jd, Cd) →Jd
	Executa o coletor de rótulos, informando que, após o salto, o comando recebido deve ser executado. Essa semântica é útil na coleta de rótulos em laços, pois a coleta é geralmente realizada com base apenas no corpo do laço. Após o salto e a execução do resto do corpo do laço, a próxima iteração deve ser realizada. Nesse caso, o segundo parâmetro corresponderia ao próprio laço.
escape	(Cd, Id, Scope) →Cd
	Instala no *environment* a continuação corrente associando-a ao identificador recebido e executa o comando.
	(Cd, Id) →Cd
	Versão com escopo estático do componente `escape`.
float	String →Ed
	Transforma a *string* em um valor de tipo *float*.

for	(Xd, Ex, Xd, x) →x, x ∈ Xd
	Modela um *for* genérico. Os quatro parâmetros desse componente são em ordem: um componente a ser executado antes da primeira iteração, uma expressão a ser usado como teste do laço, um componente a ser executado após cada iteração e o corpo do laço. Como é comum que o primeiro parâmetro seja uma declaração, sua contribuição para o *environment* é acumulada dentro do componente. Se algum outro parâmetro for uma declaração, o acúmulo deve ser feito explicitamente.
	(Id, For, Cd) →Cd
	Modela o *for* da linguagem ALGOL 60. O componente recebe três parâmetros. O primeiro deles é um identificador a ser utilizado como variável de controle. O segundo parâmetro modela a iteração. Por fim, o terceiro parâmetro modela o corpo do laço. Durante a execução do componente, o corpo do laço é transformado em um procedimento que armazena o valor recebido na variável de controle.
free	Ex →Cd
	Avalia a expressão recebida e libera a posição associada ao *location* resultante no *store*.
getType	(Ex+Id) →Ed
	Recupera o tipo de uma variável, valor primitivo ou objeto.
index	(Ex,Ex) →Ed
	Acessa uma posição de arranjo. O primeiro parâmetro é o arranjo em si, enquanto o segundo é o número da posição desejada.
input	Ed
	Lê um valor da entrada e o passa para a continuação.
int	String →Ed
	Transforma a *string* em um valor de tipo *int*.

isBool	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *bool*.
isByte	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *byte*.
isChar	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *char*.
isDouble	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *double*.
isFloat	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *float*.
isInt	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *int*.
isLong	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *long*.
isShort	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *short*.
isString	(Ex+Id) →Ed
	Verifica se o valor da expressão ou variável recebida possui tipo *string*.
jump	Id →Cd

	Interrompe o fluxo de controle e executa a continuação associada ao rótulo recebido como parâmetro.
long	String →Ed
	Transforma a *string* em um valor de tipo *long*.
lookup	[Id+(Ex,Id)+(Id,Id)] →Ed
	Obtém o valor ligado a um identificador. Caso o componente receba apenas um identificador, o *environment* corrente é utilizado na busca. Caso contrário, o valor associado ao segundo parâmetro é buscado na classe, objeto ou registro correspondente ao primeiro parâmetro – seja esse o próprio valor ou um identificador.
loop	(Ex, Cd, Ex) →Cd
	Modela laços genéricos. Caso o valor da primeira expressão seja verdadeiro, o comando é executado. Caso o valor da segunda expressão seja verdadeiro, o laço é executado novamente. Caso alguma das expressões possua valor falso, o controle é transferido para a continuação.
	(Ex, Cd) →Cd
	Forma simplificada de `loop` em que a segunda expressão é sempre verdadeira. Útil para modelar construções como *while*.
	(Cd, Ex) →Cd
	Forma simplificada de `loop` em que a primeira expressão é sempre verdadeira. Útil para modelar construções como *do-while*.
makeAbstraction	(Xd, IOp, EOp) →Xd

	Modela módulos. O primeiro parâmetro é executado utilizando apenas as ligações importadas, e, entre as ligações produzidas por ele, apenas aquelas exportadas são visíveis fora do módulo. Há duas opções de importação e duas opções de exportação. A opção `Transparent` importa todas as ligações vindas de fora do módulo, enquanto `Visible` exporta todas as ligações produzidas dentro do módulo. As opções `Opaque` e `Encapsulated` permitem determinar listas de importação e exportação respectivamente.
makeClosure	(Dd, x) →x, x ∈ Ed + Cd
	Executa o segundo parâmetro utilizando o *environment* recebido acrescido das ligações produzidas pelo primeiro parâmetro. Todas as alterações feitas ao *environment* pelos dois parâmetros são descartadas ao fim da execução de X.
	x →x, x ∈ Ed + Cd
	Executa o parâmetro utilizando o *environment* recebido. Todas as alterações feitas ao *environment* pelo parâmetro são descartadas ao fim de sua execução.
object	Ex →Ed
	O componente `object` é responsável por instanciar objetos. A expressão E é avaliada e a classe resultante é utilizada para se obter as declarações do objeto. O resultado é a composição dos pequenos *environments* da classe e do objeto.
output	Ex →Cd
	Modela escrita na saída. A expressão é avaliada e seu valor armazenado no final da saída. Caso o valor da expressão seja um *location*, o valor armazenado é aquele associado no *store*.
parametrize	(Ad*, Xd) →Proc

	Realiza abstração de parâmetros. Este componente recebe uma lista de pares que definem os tipo de passagem e o nome dos parâmetros formais e um componente que corresponde ao corpo do procedimento. Aceitam-se passagem por valor, por referência, por nome, por constante e *value and result*.
pop	Cd
	Apaga o nível corrente da pilha do *store*, removendo todos valores associados a esse nível. No entanto, é necessário que pelo menos um nível tenha sido criado com o componente `push`. Caso contrário, a avaliação de `pop` resulta em erro.
proc	(Xd + Proc, Scope) →Ed
	Cria procedimentos. O primeiro parâmetro corresponde ao corpo do procedimento a ser executado e a opção de escopo é utilizada no momento ad chamada. Procedimentos com parâmetros podem ser criados passando-se o componente `parametrize` como primeiro parâmetro de `proc`.
	Xd + Proc →Ed
	Versão de `proc` com escopo estático.
push	Cd
	Cria um novo nível na pilha do *store*.
range	(Ex, Ex, Ex) →For
	Percorre o intervalo entre o valor da primeira expressão e o valor da segunda no *for* da linguagem ALGOL 60. A cada iteração, o valor é passado ao corpo do laço. O valor utilizado em cada iteração é o resultado de somar o valor da segunda expressão ao valor utilizado na iteração anterior. A primeira iteração utiliza o valor da primeira expressão, e, quando o valor calculado ultrapassa o valor da terceira expressão, a execução do laço termina.
record	Dd →Ed

	Executa a declaração e passa para a continuação as ligações produzidas em forma de registro.
ref	Ex →Ed
	Armazena o valor da expressão na pilha do *store* e passa o *location* associado para a continuação.
resultIs	Ex →Cd
	Passa o valor resultante da avaliação da expressão `E` à uma continuação instalada no *environment*, transferindo o controle a essa continuação. Este componente opera em conjunto com *valof*.
run	Cd→File→Env→Store→Ans
	Modela a execução de um programa. O *environment* e o *store* recebidos são utilizados como contexto inicial para a execução do comando. Entrada e saída são acrescentados ao *store*. Ao fim da execução, é retornada uma tupla contendo contendo: um indicador de fim da execução(`stop` ou `error`), um valor e *environment* e *store* válidos no momento em que a execução termina.
seq	(Cd,Cd) →Cd Modela sequenciamento de dois comandos. O segundo comando é executado após o primeiro. (For, For) →For
	Esta forma do componente `seq` faz o sequenciamento de dois iteradores do *for* da linguagem ALGOL 60. Executa-se o primeiro iterador e depois o segundo.
short	String →Ed
	Transforma a *string* em um valor de tipo *short*.
singleStep	Ex →For
	Modela um único passo do *for* da linguagem ALGOL 60. O valor da expressão é passado ao corpo do laço e este é executado.

skipLabel	Jd
	Modela casos em que uma determinada construção não possui rótulos. Este comando não altera o contexto ou produz valor.
skip	Cd
	O componente `skip` não faz qualquer alteração no contexto, passando o controle diretamente para a continuação.
static	Scope
	Modela escopo estático. Utilizado como parâmetro dos componentes `bind`, `addLabel`, `escape` e `catch`.
string	String →Ed
	Retorna a *string* recebida. Utilizado apenas para explicitar o tipo do valor.
subclass	(Ex, Ex) →Ed
	Estabelece que a classe definida pela segunda expressão é uma sub-classe daquela definida pela primeira expressão.
throw	Ex →Cd
	Realiza um salto, passando o valor da expressão para o procedimento instalado no *environment* pelo componente `catch` mais interno.
trap	(Ex, [(Ex,Cd)], Cd) →Cd
	Modela uma seção de código com várias entradas. A primeira expressão é avaliada e seu valor é comparada com uma série de valores associados a comandos. Quando a comparação é bem sucedida, o comando associado é executado. Caso todas as comparações falhem, o último parâmetro é executado.
type	String →String
	Retorna o valor recebido. Utilizado para explicitar que o valor é um nome de tipo.
useDynamicScope	Xd →Xd

	Executa o parâmetro utilizando escopo dinâmico.
useStaticScope	Xd →Xd
	Executa o parâmetro utilizando escopo estático.
valof	Cd →Ed
	Transforma um comando em expressão. O resto do programa é instalado no *environment* para ser usado pelo componente *resultIs* e o comando é executado.
while	(Ex, Ex) →For
	Executa o corpo do *for* da linguagem ALGOL 60 enquanto o valor da segunda expressão for verdadeiro. A cada passo, o valor de da primeira expressão é atribuído à variável de controle.

Apêndice E

Implementação em Haskell

E.1 Domains.hs

Listing E.1: Listagem do módulo de domínios semânticos.

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE CPP #-}

module Domains(
    Ans(AnsHalt, FlexibleAns),
    Result(RError, Stop),
    Error(Error),
    Id,
    Bv(BvNum, BvBool, BvString, BvChar, Undefined),
    Number(NumInt, NumShort, NumLong, NumByte, NumFloat, NumDouble),
    Rv,
    File(File),
    Array(Array),
    Dv(DvLoc, DvBv, DvQ, DvX, DvN, DvProc, DvArray, DvEnv, DvScope),
    EnvValue(EnvDv, Unbound),
    EnvId(Id, Type),
    Sv(SvRv, SvRec, SvArray, SvFile),
    StoreValue(StoreSv, Unused),
    Ev(EvBv, EvDv, EvEnv, EvArray, EvFile),
    Loc(Loc, Input, Output, Level, New),
    Access(Var, Const),
    Env,
    Store,
    Q,
    Pro,
    Cd,
    Ed,
    Ex,
    eval,
    evalList,
```

```
    Dd,
    Xd,
    Jd,
    For,
    Scope,
    Ad(Ad),
    Proc(Proc),
    Mode(Ref, Value, ValueAndResult, Constant, Name),
    Type,
    toSv,
    toEv,
    isSv,
    toBool,
    ImportOption(Opaque, Transparent),
    ExportOption(Visible, Encapsulated),
    error,
    errorMsg,
    undefined,
    (?),
    isByte,
    isInt,
    isShort,
    isLong,
    isDouble,
    isFloat,
    isChar,
    isString,
    isBool
        ) where

import Prelude hiding (error, undefined)
import Debug.Trace
import Data.Int

data Ans = AnsHalt (Result, Ev, Env, Store) | FlexibleAns (Ev, Env, Store)
data Result = RError Error | Stop deriving (Eq,Show)
data Error = Error deriving (Eq,Show)
type Id = String
data Bv = BvNum Number | BvBool Bool | BvString String | BvChar Char | Undefined deriving (Eq,Show)
data Number = NumInt Int | NumLong Int64 | NumFloat Float | NumDouble Double
            | NumByte Int8 | NumShort Int16 deriving (Eq,Ord,Show)
data Array = Array ([Loc],Int,Int) deriving (Eq,Show)
data Dv = DvLoc Loc | DvBv Bv | DvQ (Ev -> Env -> Store -> Ans) | DvX (Env -> Q -> Store -> Ans)
        | DvN (Q -> Store -> Ans) | DvProc Proc | DvArray Array | DvEnv Env | DvScope Scope
data Sv = SvRv Rv | SvFile File | SvRec Env | SvArray Array deriving (Eq,Show)
type Rv = Bv
data File= File [Rv] deriving (Eq,Show)
data Ev = EvBv Bv | EvDv Dv | EvEnv Env | EvArray Array | EvFile File
data Loc = Loc (Access, Address) | Input | Output | Level | New deriving (Eq,Show)
type Address = (Int,Int)
data Access = Const | Var deriving (Eq,Show)
type Env = EnvId -> EnvValue
data EnvId = Id Id | Type Type deriving (Eq,Show)
data EnvValue = EnvDv Dv | Unbound deriving (Eq,Show)
type Store = Loc -> StoreValue
```

```
data StoreValue = StoreSv Sv | Unused deriving (Eq,Show)
type Q = Ev -> Env -> Store -> Ans
data Ad = Ad (Mode,Id)
data Mode = Ref | Value | ValueAndResult | Constant | Name deriving (Eq,Show)
data Proc = Proc ([Ad],Env -> Q -> [Ev] -> Store -> Ans)
type Type = String
type Class = Env
type Object = Env
type For = Env -> (Ev -> Cd) -> Q -> Store -> Ans

type Pro = File -> Env -> Store -> Ans
type Ed = Env -> Q -> Store -> Ans
type Cd = Env -> Q -> Store -> Ans
type Dd = Env -> Q -> Store -> Ans
type Xd = Env -> Q -> Store -> Ans
-- Xd is used to refer to (Cd+Ed+Dd)
type Jd = Env -> Q -> Env
type Scope = (Env,Env) -> Env

-- Library options
data ImportOption = Transparent | Opaque [Id] deriving (Eq,Show)
data ExportOption = Visible | Encapsulated [Id] deriving (Eq,Show)

-- Class to unify Ed and Ev
class Show e => Ex e where
  eval :: e -> Ed
  evalList :: [e] -> Env -> ([Ev] -> Env -> Store -> Ans) -> Store -> Ans
  evalList e r q = evalList' [] e r q
    where evalList' es [] r q = q es r
          evalList' es (e':es') r q = eval e' r $ \e r' -> evalList' (es ++ [e]) es' r' q

instance Ex Ed where
  eval pE r q = \s -> pE r q s

instance Ex Ev where
  -- by name parameters
  eval (EvDv (DvN n)) r q = \s -> n q s
  -- values passed as expressions
  eval e r q = \s -> q e r s

-- Projection to Sv
class ToSv x where
  toSv :: (Sv -> Env -> Store -> Ans) -> x -> Env -> Store -> Ans

instance ToSv Ev where
  toSv q (EvBv b) r s = q (SvRv b) r s
  toSv q (EvDv (DvBv b)) r s = q (SvRv b) r s
  toSv q (EvEnv r') r s = q (SvRec r') r s
  toSv q (EvArray a) r s = q (SvArray a) r s
  toSv q e r s = errorMsg ("Cannot cast " ++ show e ++ " to Sv.") undefined r s


-- Injection to Ev
class ToEv x where
  toEv :: (Ev -> Env -> Store -> Ans) -> x -> Env -> Store -> Ans
```

```
instance ToEv Bv where
  toEv q b = q (EvBv b)

instance ToEv Dv where
  toEv q d = q (EvDv d)

instance ToEv Loc where
  toEv q l = q (EvDv (DvLoc l))

instance ToEv Ev where
  toEv q = q

instance ToEv EnvValue where
  toEv q (EnvDv d) r s = toEv q d r s
  toEv q _ r s = error undefined r s


-- Projection to Bool --
class ToBool x where
  toBool :: (Bool -> Env -> Store -> Ans) -> x -> Env -> Store -> Ans

instance ToBool Bv where
  toBool q (BvBool b) r s = q b r s
  toBool q _ r s = error undefined r s

instance ToBool Ev where
  toBool q (EvBv b) r s = toBool q b r s
  toBool q _ r s = error undefined r s

instance ToBool Dv where
  toBool q (DvBv b) r s = toBool q b r s
  toBool q _ r s = error undefined r s

-- Arithmetic Operations
instance Num Number where
  (NumByte b1) + (NumByte b2) = NumByte (b1 + b2)
  (NumInt b1) + (NumInt b2) = NumInt (b1 + b2)
  (NumShort b1) + (NumShort b2) = NumShort (b1 + b2)
  (NumLong b1) + (NumLong b2) = NumLong (b1 + b2)
  (NumDouble b1) + (NumDouble b2) = NumDouble (b1 + b2)
  (NumFloat b1) + (NumFloat b2) = NumFloat (b1 + b2)
  (NumByte b1) - (NumByte b2) = NumByte (b1 - b2)
  (NumInt b1) - (NumInt b2) = NumInt (b1 - b2)
  (NumShort b1) - (NumShort b2) = NumShort (b1 - b2)
  (NumLong b1) - (NumLong b2) = NumLong (b1 - b2)
  (NumDouble b1) - (NumDouble b2) = NumDouble (b1 - b2)
  (NumFloat b1) - (NumFloat b2) = NumFloat (b1 - b2)
  (NumByte b1) * (NumByte b2) = NumByte (b1 * b2)
  (NumInt b1) * (NumInt b2) = NumInt (b1 * b2)
  (NumShort b1) * (NumShort b2) = NumShort (b1 * b2)
  (NumLong b1) * (NumLong b2) = NumLong (b1 * b2)
  (NumDouble b1) * (NumDouble b2) = NumDouble (b1 * b2)
  (NumFloat b1) * (NumFloat b2) = NumFloat (b1 * b2)
  b1 * b2 = NumInt 0
```

```
  abs (NumByte b) = NumByte (abs b)
  abs (NumInt b) = NumInt (abs b)
  abs (NumShort b) = NumShort (abs b)
  abs (NumLong b) = NumLong (abs b)
  abs (NumDouble b) = NumDouble (abs b)
  abs (NumFloat b) = NumFloat (abs b)
  negate (NumByte b) = NumByte (negate b)
  negate (NumInt b) = NumInt (negate b)
  negate (NumShort b) = NumShort (negate b)
  negate (NumLong b) = NumLong (negate b)
  negate (NumDouble b) = NumDouble (negate b)
  negate (NumFloat b) = NumFloat (negate b)
  signum (NumByte b) = NumByte (signum b)
  signum (NumInt b) = NumInt (signum b)
  signum (NumShort b) = NumShort (signum b)
  signum (NumLong b) = NumLong (signum b)
  signum (NumDouble b) = NumDouble (signum b)
  signum (NumFloat b) = NumFloat (signum b)
  fromInteger b = NumInt (fromInteger b)

instance Num Bv where
  (BvNum b1) + (BvNum b2) = BvNum (b1 + b2)
  _ + _ = Undefined
  (BvNum b1) - (BvNum b2) = BvNum (b1 - b2)
  _ - _ = Undefined
  (BvNum b1) * (BvNum b2) = BvNum (b1 * b2)
  _ * _ = Undefined
  abs (BvNum b) = BvNum (abs b)
  abs _= Undefined
  negate (BvNum b) = BvNum (negate b)
  negate _= Undefined
  signum (BvNum b) = BvNum (signum b)
  signum _= Undefined
  fromInteger b = BvNum (fromInteger b)

instance Num Dv where
  (DvBv b1) + (DvBv b2) = DvBv (b1 + b2)
  _ + _ = DvBv Undefined
  (DvBv b1) - (DvBv b2) = DvBv (b1 - b2)
  _ - _ = DvBv Undefined
  (DvBv b1) * (DvBv b2) = DvBv (b1 * b2)
  _ * _ = DvBv Undefined
  abs (DvBv b) = DvBv (abs b)
  abs _= DvBv Undefined
  negate (DvBv b) = DvBv (negate b)
  negate _= DvBv Undefined
  signum (DvBv b) = DvBv (signum b)
  signum _= DvBv Undefined
  fromInteger b = DvBv (fromInteger b)

instance Num Ev where
  (EvBv b1) + (EvBv b2) = EvBv (b1 + b2)
  (EvDv d1) + (EvDv d2) = EvDv (d1 + d2)
  _ + _ = EvBv Undefined
  (EvBv b1) - (EvBv b2) = EvBv (b1 - b2)
```

```
  (EvDv d1) - (EvDv d2) = EvDv (d1 - d2)
  _ - _ = EvBv Undefined
  (EvBv b1) * (EvBv b2) = EvBv (b1 * b2)
  (EvDv d1) * (EvDv d2) = EvDv (d1 * d2)
  _ * _ = EvBv Undefined
  abs (EvBv b) = EvBv (abs b)
  abs (EvDv d) = EvDv (abs d)
  abs _ = EvBv Undefined
  signum (EvBv b) = EvBv (signum b)
  signum (EvDv d) = EvDv (signum d)
  signum _ = EvBv Undefined
  negate (EvBv b) = EvBv (negate b)
  negate (EvDv d) = EvDv (negate d)
  negate _ = EvBv Undefined
  fromInteger b = EvBv (fromInteger b)

instance Fractional Number where
  recip (NumDouble x) =  NumDouble (1 / x)
  recip (NumFloat x)  =  NumFloat (1 / x)
  recip b             =  NumDouble 0
  (NumDouble x) / (NumDouble y) =  NumDouble (x * recip y)
  (NumFloat x) / (NumFloat y) =  NumFloat (x * recip y)
  (NumInt x) / (NumInt y) =  NumInt (x `div` y)
  (NumShort x) / (NumShort y) =  NumShort (x `div` y)
  (NumByte x) / (NumByte y) =  NumByte (x `div` y)
  (NumLong x) / (NumLong y) =  NumLong (x `div` y)
  fromRational x = (NumDouble (fromRational x))

instance Fractional Bv where
  recip (BvNum x) =  BvNum (recip x)
  recip b         =  BvNum (NumDouble 0)
  (BvNum x) / (BvNum y) =  BvNum (x / y)
  fromRational x = (BvNum (fromRational x))

instance Fractional Ev where
  recip (EvBv x) =  EvBv (recip x)
  recip e        =  EvBv (BvNum (NumDouble 0))
  (EvBv x) / (EvBv y) =  EvBv (x / y)
  fromRational x = EvBv (fromRational x)

-- Relational Operations

instance Ord Bv where
  compare (BvNum n1) (BvNum n2) = compare n1 n2
  compare (BvBool b1) (BvBool b2) = compare b1 b2
  compare (BvString s1) (BvString s2) = compare s1 s2
  compare _ _ = GT

instance Ord Dv where
  compare (DvBv b1) (DvBv b2) = compare b1 b2
  compare _ _ = GT

instance Ord Ev where
  compare (EvBv b1) (EvBv b2) = compare b1 b2
  compare (EvDv d1) (EvDv d2) = compare d1 d2
```

```
  compare _ _ = GT

instance Eq Ev where
  (EvBv b1) == (EvBv b2) = b1 == b2
  (EvDv d1) == (EvDv d2) = d1 == d2
  _ == _ = False
  (EvBv b1) /= (EvBv b2) = b1 /= b2
  (EvDv d1) /= (EvDv d2) = d1 /= d2
  _ /= _ = False

instance Show Ev where
  show (EvBv b) = "(EvBv " ++ show b ++ ")"
  show (EvDv d) = "(EvDv " ++ show d ++ ")"
  show (EvEnv r) = "(EvEnv)"
  show (EvArray (Array a)) = "(EvEnv " ++ show a ++ ")"
  show (EvFile f) = "(EvFile " ++ show f ++ ")"


instance Eq Dv where
  (DvBv b1) == (DvBv b2) = b1 == b2
  (DvLoc l1) == (DvLoc l2) = l1 == l2
  _ == _ = False
  (DvBv b1) /= (DvBv b2) = b1 /= b2
  (DvLoc l1) /= (DvLoc l2) = l1 /= l2

instance Show Dv where
  show (DvBv b) = "(DvBv " ++ show b ++ ")"
  show (DvLoc l) = "(DvLoc " ++ show l ++ ")"
  show (DvQ _) = "(DvQ ?)"
  show (DvX _) = "(DvQ X)"
  show (DvProc _) = "(DvProc ?)"
  show (DvArray a) = "(DvArray " ++ show a ++ ")"

instance Eq Env where
  r == r' = False

instance Show Env where
  show r = "(Env)"
-- Type Checking --
class IsSv t where
  isSv :: t -> Bool
  isSv _ = False

instance IsSv Bv where
  isSv _ = True

instance IsSv Ev where
  isSv (EvBv _) = True
  isSv _ = False

class TypeCheck t1 t2 where
  (?) :: String -> (t2 -> Env -> Store -> Ans) -> t1 -> Env -> Store -> Ans
  (?) _ q t r s = errorMsg "TypeCheck" undefined r s

instance TypeCheck Ev Loc where
```

```
  (?) "Loc" q (EvDv (DvLoc l)) = q l

instance TypeCheck Ev Ev where
  (?) "Loc" q (EvDv (DvLoc l)) = q (EvDv (DvLoc l))
  (?) "Bv" q (EvBv b) = q (EvBv b)
  (?) "Rv" q (EvBv b) = q (EvBv b)
  (?) "Array" q (EvArray a) = q (EvArray a)
-- Check Bv internal types
  (?) "Bool" q (EvBv b) = (?) "Bool" (\b -> q (EvBv b)) b
  (?) "Num" q (EvBv b) =  (?) "Num" (\b -> q (EvBv b)) b
  (?) "String" q (EvBv b) =  (?) "String" (\b -> q (EvBv b)) b

instance TypeCheck Bv Bv where
  (?) "Bool" q (BvBool b) = q (BvBool b)
  (?) "Num" q (BvNum n) = q (BvNum n)
  (?) "String" q (BvString s) = q (BvString s)

instance TypeCheck Dv Dv where
  (?) "Loc" q (DvLoc l) = q (DvLoc l)
  (?) "Bv" q (DvBv b) = q (DvBv b)
  (?) "Q" q (DvQ q') = q (DvQ q')
  (?) "Xd" q (DvX a) = q (DvX a)
  (?) "Array" q (DvArray a) = q (DvArray a)

instance TypeCheck Dv Loc where
  (?) "Loc" q (DvLoc l) = q l

instance TypeCheck Dv Proc where
  (?) "Proc" q (DvProc p) = q p
  (?) "Fun" q (DvProc p) = q p

instance TypeCheck Ev Dv where
  (?) "Dv" q (EvBv b) = q (DvBv b)
  (?) "Loc" q (EvDv (DvLoc l)) = q (DvLoc l)
  (?) "Dv" q (EvDv (DvLoc l)) = q (DvLoc l)
  (?) "Array" q (EvDv (DvArray a)) = q (DvArray a)
  (?) "Dv" q (EvDv (DvArray a)) = q (DvArray a)
  (?) "Proc" q (EvDv (DvProc a)) = q (DvProc a)
  (?) "Fun" q (EvDv (DvProc a)) = q (DvProc a)
  (?) "Dv" q (EvDv (DvProc a)) = q (DvProc a)
  (?) n q t = \r s -> errorMsg ("Cannot cast " ++ show t ++ " to Dv as " ++ n ) undefined r s

instance TypeCheck Ev Env where
  (?) "Record" q (EvEnv r) r' s = q r r' s
  (?) "Class" q (EvEnv r) r' s = q r r' s
  (?) "Object" q (EvEnv r) r' s = q r r' s
  (?) _ q t r s = errorMsg ("Cannot cast " ++ show t ++ " to Ev") undefined r s

instance TypeCheck Ev Number where
  (?) "Num" q (EvBv (BvNum n)) = q n

instance TypeCheck Ev Int where
  (?) "Num" q (EvBv (BvNum (NumInt n))) = q n

instance TypeCheck Ev Bool where
```

```
  (?) "Num" q (EvBv (BvBool b)) = q b

instance TypeCheck Ev Array where
  (?) "Array" q (EvArray a) = q a

instance TypeCheck Ev Proc where
  (?) "Proc" q (EvDv d) = (?) "Proc" q d
  (?) "Fun" q (EvDv d) = (?) "Fun" q d

instance TypeCheck Sv Ev where
  (?) "Ev" q (SvFile f) = q (EvFile f)
  (?) "Ev" q (SvRv b) = q (EvBv b)
  (?) "Ev" q (SvRec r) = q (EvEnv r)
  (?) "Ev" q (SvArray a) = q (EvArray a)

instance TypeCheck Ev Rv where
  (?) "Rv" q (EvBv b) = q b
  (?) _ q t = errorMsg ("Cannot cast " ++ show t ++ " to Rv") undefined

isByte :: Ev -> Bool
isByte (EvBv (BvNum (NumByte _))) = True
isInt :: Ev -> Bool
isInt (EvBv (BvNum (NumInt _))) = True
isShort :: Ev -> Bool
isShort (EvBv (BvNum (NumShort _))) = True
isLong :: Ev -> Bool
isLong (EvBv (BvNum (NumLong _))) = True
isDouble :: Ev -> Bool
isDouble (EvBv (BvNum (NumDouble _))) = True
isFloat :: Ev -> Bool
isFloat (EvBv (BvNum (NumFloat _))) = True
isChar :: Ev -> Bool
isChar (EvBv (BvChar _)) = True
isString :: Ev -> Bool
isString (EvBv (BvString _)) = True
isBool :: Ev -> Bool
isBool (EvBv (BvBool _)) = True

-- error --
error :: Ev -> Env -> Store -> Ans
error e r s = trace ("Error ") (AnsHalt (RError Error, e, r, s))

errorMsg :: String -> Ev -> Env -> Store -> Ans
errorMsg m e r s = trace ("Error:" ++ m) (AnsHalt (RError Error, e, r, s))

undefined :: Ev
undefined = (EvBv Undefined)

-- Debug --
instance Show (Env -> Q -> Store -> Ans) where
  show e = "(Env -> Q -> Store -> Ans)"
```

E.2 Components.hs

Listing E.2: Listagem do módulo de componentes.

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Components(
    run,
    pipe,
-- Commands
    skip,
    assign,
    output,
    resultIs,
    seq,
    call,
    loop,
--- Memory Management,
    alloc,
    deref,
    ref,
    const,
--- Sequencers
    abort,
    throw,
    escape,
    catch,
    trap,
-- Expressions
    lookup,
    index,
    apply,
    valof,
    record,
    array,
--- Data Abstraction
    class0,
--- Types
    int,
    short,
    long,
    byte,
    float,
    double,
    char,
    string,
    bool,
-- Declarations
    accum,
    elabSeq,
    elabCol,
    elabRec,
-- Scope
    static,
    dynamic,
    useStaticScope,
```

```
    useDynamicScope,
-- Generic
    addEnv,
    choose,
-- Saltos
    addLabel,
    elab,
    -- elabSeq,
    -- addEnv,
    jump,
-- Abstraction
    proc,
    makeClosure,
    makeAbstraction,
    bind,
    parametrize
    ) where

import Prelude hiding (const,error,seq,lookup,filter,undefined,catch)
import Debug.Trace
import Data.Int
import Domains
import Auxiliar hiding (static,dynamic)

-- run (Exp x Dec + Exp) -> Prog --
class Run r where
  run :: r -> Pro

instance Run Cd where
  run pC i r s = pC r q s''
    where s' = (s <+> (Input, SvFile i))
          s'' = (s' <+> (Output, SvFile (File [])))
          q = \e r s -> AnsHalt (Stop, e, r, s )

-- Commands ---------------------------------------------------------------------
-- skip -> Cmd --
skip :: Cd
skip r q s = q (EvBv Undefined) r s

assign :: (Ex e1, Ex e2) => (e1,e2) -> Cd
assign (pE1,pE2) r q = eval pE1 r $ "Loc" ? (\l r' -> eval pE2 r' $ deref' $ "Rv" ? (update l q))
--assign (pE1,pE2) r q = \s -> (?) "Loc" (\(Loc l) -> q (EvBv Undefined)) (EvBv Undefined) r s

-- output (Ed) -> Cmd --
output :: (Ex e) => e -> Cd
output pE r q = cont (\o r -> eval pE r $ deref' $ "Rv" ? (\e r' s' -> trace ("Add to output:"
 ++ show e) q undefined r' (s' <+> (Output,output' o e)))) Output r
  where output' o e = case o of
                        EvFile (File f) -> SvFile (File (f ++ [e]))

class Seq s where
  seq :: (s,s) -> s

instance Seq Xd where
  seq (pX1,pX2) r q = pX1 r ( \e r' -> pX2 r' q)
```

```
resultIs :: Ex e => e -> Cd
resultIs pE r q = lookup' "resultIs" r $ \(DvQ q') r' -> eval pE r' $ deref' $ q'

call :: (Ex e1, Ex e2) => (e1,[e2]) -> Cd
call (pE1,pE2) r q = eval pE1 r $ \e r' -> case e of
                                             (EvDv (DvProc(Proc (a,p))))->evalPar (zip a pE2) p q r'
                                             (EvDv (DvX x)) -> x r' q
                                             otherwise -> errorMsg ("Calee is not a procedure.")
                                               undefined r'

class Loop l where
  loop :: l -> Cd

instance (Ex e1, Ex e2) => Loop (e1,Cd,e2) where
  loop (pE1,pC,pE2) = fix (\f -> choose(pE1, seq(pC, choose(pE2, f, skip)), skip))
    where fix f = f (fix f)

instance Ex e => Loop (e,Cd) where
  loop (pE,pC) = loop(pE, pC, bool("True"))

--Gera um erro de duplicata, porque nao e possivel distinguir entre Ed e Cd
--instance Loop (Cd,Ed) where
--  loop (pC,pE) = loop(bool("True"), pC, pE)

-- For --

class For' f where
  for :: f -> Xd

instance For' (Xd,Ed,Xd,Xd) where
  for (pX1,pE,pX2,pX3) = seq(accum(pX1), loop(pE, seq(pX3,pX2), bool("true")))

instance For' (Id, For, Cd) where
  for (pI,pF,pC) r q = accum(bind(pI, ref(undefined))) r $ \e r' -> pF r' (p pI) q
    where p i = \e -> assign(lookup(i), e)

singleStep :: Ex e => e -> For
singleStep pE r p q = eval pE r $ \e r' -> p e r' q

while :: (Ex e1, Ex e2) => (e1,e2) -> For
while (pE1,pE2) = fix (\f r p q -> eval pE1 r $ \e r' -> eval pE2 r' $ deref' $
 "Bool" ? (\b r'' -> if (b) then (p e r'' (\e r -> f r p q)) else ( q undefined r'')))
    where fix f = f (fix f)

range :: (Ex e1, Ex e2, Ex e3) => (e1,e2,e3) -> For
range (pE1,pE2,pE3) r p q = eval pE1 r $ "Num" ? \n1 r1 -> step (pE2,pE3) r1 p q n1
  where step (w2,w3) r p q n1 = eval w2 r $ "Num" ? \n2 r2 -> eval w3 r2 $ "Num" ? \n3 r3 ->
          if ((n1 - n3) * (signum n2) < 1)
            then (p n1 r3 (\e r' -> step (w2,w3) r' p  q (n1 + n2)))
            else (q undefined r3)

instance Seq For where
  seq (pF1,pF2) r p q = pF1 r p $ \e r' -> pF2 r' p q
```

```
-- Gerencia de Memoria
push e r q s = q e r (\l -> if (l == Level) then (getNextLevel s) else s l)
  where getLevel s = case (s Level) of
                       (StoreSv (SvRv (BvNum (NumInt i)))) -> i
                       otherwise -> -1
        getNextLevel s = (StoreSv (SvRv (BvNum (NumInt ((getLevel s) + 1)))))

pop e r q s =if ((getLevel s) == 1)
               then (errorMsg ("Already at bottom of stack") undefined r s)
               else  q e r (\l -> if (l == Level) then (getPreviousLevel s) else (case l of
 Loc (_,(n, _)) -> if (n == (getLevel s)) then Unused else ( s l)
 otherwise -> s l))
  where getLevel s = case (s Level) of
                       (StoreSv (SvRv (BvNum (NumInt i)))) -> i
                       otherwise -> -1
        getPreviousLevel s = (StoreSv (SvRv (BvNum (NumInt ((getLevel s) - 1)))))

alloc :: Ex e => e -> Ed
alloc pE r q = eval pE (r <+> ("allocation",(DvBv (BvString ("heap"))))) $ ref' $ (\e r' -> q e r)

deref :: Ex e => e -> Ed
deref pE r q = eval pE r $ deref' $ q

ref :: Ex e => e -> Ed
ref pE r q = eval pE r $ ref' $ q

const :: Ex e => e -> Ed
const pE r q = ref pE r $ "Loc" ? \l -> q (EvDv (DvLoc (const' l)))


-- Sequencers --
abort :: Cd
abort r q = errorMsg "Execution aborted" undefined r

class Escape e where
  escape :: e -> Cd

instance Escape (Cd, Id, Scope) where
  escape (pC, pI, scope) r q = pC (r <+> (pI, DvQ (\e r' -> q e (scope(r,r'))))) q

instance Escape (Cd, Id) where
  escape (pC, pI) r q = escape (pC, pI, getScope r) r q

throw :: Ex e => e -> Cd
throw pE r q = eval pE r $ \e r'-> lookup' "catch" r' $ \(DvProc (Proc (a,p))) r'' -> p r'' q [e]

class Catch c where
  catch :: c -> Cd

instance Catch (Cd, Proc, Scope) where
  catch (pC,pP, scope) r q = pC (r <+> ("catch",catch' pP scope r q)) q
    where catch' (Proc (a,p)) scope r q = DvProc (Proc (a, \r' q' -> p (scope(r,r')) q))

instance Catch (Cd, Proc) where
  catch (pC,pP) r q = catch (pC, pP, getScope r) r q
```

```
trap :: (Ex e1, Ex e2) => (e1,[(e2,Xd)],Xd) -> Xd
trap (pE,pL,pX) r q = case (unzip pL) of
                        (es,cs) -> evalList es r $ \es' r' -> eval pE r' $ deref' $ \e -> find e es'
                                     cs pX q
  where find e (e':es) (c:cs) x q r = deref' (\e'' r' -> if (e == e'')
                                                         then (c r' q)
                                                         else (find e es cs x q r')) e' r
        find e [] [] x q r = x r q

-- Expressions ---------------------------------------------------------------
-- Basic Types --
int :: String -> Ed
int b r q = q (EvBv (BvNum (NumInt (read b :: Int)))) r
short :: String -> Ed
short b r q = q (EvBv (BvNum (NumShort (read b :: Int16)))) r
long :: String -> Ed
long b r q = q (EvBv (BvNum (NumLong (read b :: Int64)))) r
byte :: String -> Ed
byte b r q = q (EvBv (BvNum (NumByte (read b :: Int8)))) r
float :: String -> Ed
float b r q = q (EvBv (BvNum (NumFloat (read b :: Float)))) r
double :: String -> Ed
double b r q = q (EvBv (BvNum (NumDouble (read b :: Double)))) r
char :: String -> Ed
char b r q = q (EvBv (BvChar (read b :: Char))) r
string :: String -> Ed
string b r q = q (EvBv (BvString b)) r
bool :: String -> Ed
bool b r q = q (EvBv (BvBool (read b :: Bool))) r

-- Type Checking --
class GetType t where
  getType :: t -> Ed

instance GetType Id where
  getType pI r q = case (r (Type pI)) of
                     Unbound -> string("unknown") r q
                     (EnvDv d) -> q (EvDv d) r

instance GetType Ev where
  getType e r q = if (isByte e)
                  then string("byte") r q
                  else if (isShort e)
                         then string("short") r q
                         else if (isInt e)
                                then string("int") r q
                                else if (isLong e)
                                       then string("long") r q
                                       else if (isDouble e)
                                              then string("double") r q
                                              else if (isChar e)
                                                     then string("char") r q
                                                     else if (isString e)
                                                            then string("string") r q
```

```
                                                          else if (isBool e)
                                                                then string("bool") r q
                                                                else case (e) of
 (EvEnv o) -> case (o (Id "class")) of
                (EnvDv d) -> q (EvDv d) r
                otherwise ->  string("unkown") r q
 otherwise -> string("unkown") r q

class Lookup v where
  lookup :: v -> Ed

instance Lookup Id where
  lookup i r q = lookup' i r $ toEv $ q

instance Lookup (Id,Id) where
  lookup (pI1,pI2) r q = lookup pI1 r $deref'$"Record" ? (\e r' -> lookup pI2 e (\e' r'' -> q e' r))

instance Lookup (Ed,Id) where
  lookup (pE,pI) r q = eval pE r $ deref' $ "Record" ? (\e r' -> lookup pI e (\e' r'' -> q e' r))

instance Lookup (Ev,Id) where
  lookup (pE,pI) r q = eval pE r $ deref' $ "Record" ? (\e r' -> lookup pI e (\e' r'' -> q e' r))

class Index v where
  index :: v -> Ed

instance Ex e => Index (e,Ed) where
  index (pE1,pE2) r q = eval pE1 r$deref'$"Array" ? (\a r' -> eval pE2 r' $ "Num" ? (subscript a q))

instance Ex e => Index (e,Ev) where
  index (pE1,pE2) r q = eval pE1 r$deref'$"Array" ? (\a r' -> eval pE2 r' $ "Num" ? (subscript a q))

-- apply (Ev* -> Ev + Error, Ed*) -> Ed
class Apply x where
  apply :: x -> Ed

instance Ex e => Apply ([Ev] -> Either Ev Error, [e]) where
  apply (o, es) r q = eval' es [] (\e' -> case (o e') of
                                            Left e -> toEv q e
                                            otherwise -> errorMsg ("Operator failed:" ++ show es)
                                                         undefined) r
    where eval' :: Ex e=> [e] -> [Ev] -> ([Ev] -> Env -> Store -> Ans) -> Env -> (Store -> Ans)
          eval' (e:es) e2s q r = eval e r $ deref' $ \v -> eval' es (e2s ++ [v]) q
          eval' [] e2s q r = q e2s r

instance Ex e => Apply (String, [e]) where
  apply ("+", es) = apply(intOp (+), es)
  apply ("-", es) = apply(intOp (-), es)
  apply ("*", es) = apply(intOp (*), es)
  apply ("/", es) = apply(intOp (/), es)
  apply ("=", es) = apply(boolOp (==), es)
  apply ("!=", es) = apply(boolOp (/=), es)
  apply (">", es) = apply(boolOp (>), es)
  apply ("<", es) = apply(boolOp (<), es)
```

```
instance Ex e => Apply (String, e, e) where
  apply (o,e1,e2) = apply(o, [e1,e2])

intOp :: (Ev -> Ev -> Ev) -> ([Ev] -> Either Ev Error)
intOp o = \e -> case o (e !! 0) (e !! 1) of
                  (EvBv Undefined) -> Right Error
                  e' -> Left e'

boolOp :: (Ev -> Ev -> Bool) -> ([Ev] -> Either Ev Error)
boolOp o = \e -> Left (EvBv (BvBool (o (e !! 0) (e !! 1))))

instance Ex e => Apply (Ed,[e]) where
  apply (pE1,pE2) r q = eval pE1 r $ "Fun" ? (\(Proc (a,f)) -> evalPar (zip a pE2) f q)

instance Ex e => Apply (Ev,[e]) where
  apply (pE1,pE2) r q = eval pE1 r $ "Fun" ? (\(Proc (a,f)) -> evalPar (zip a pE2) f q)

evalPar :: Ex e => [(Ad,e)] -> (Env -> Q -> [Ev] -> Store -> Ans) -> Q -> Env -> Store -> Ans
evalPar a p q r = evalPar' [] a ( \e r' -> p r' (\e' r''-> q e' r) e) r
  where evalPar' es ((Ad (m,i),e):as) q' = evalPar'' m e $ \e -> evalPar' (es ++ [e]) as q'
        evalPar' es [] q = q es
        evalPar'' Name e q r = q (EvDv (DvN (eval e r))) r
        evalPar'' _ e q r = eval e r q

valof :: Cd -> Ed
valof pC r q = pC (r <+> ("resultIs", DvQ (\e r' -> q e r))) error

record :: (Dd) -> Ed
record (pD) r q = pD r $ \e r' -> q (EvEnv r') r

class Ex e => ClassArray e where
  array :: (e,e,[e]) -> Ed
  array (pE1,pE2,pE3) r q = eval pE1 r $
                            "Num" ? (\n1 r' -> eval pE2 r'
                                               $ "Num" ? (\n2 -> checkBoundaries n1 n2
                                               $ \r'' s -> evalList pE3 r''( storeValues [] $ \l ->
 q (EvArray (Array (l,n1,n2)))) s))
    where checkBoundaries n1 n2 q' = if (n2 < n1)
                                     then (errorMsg ("Incorrect array boundaries: (" ++
 show n1 ++ "," ++ show n2 ++ ")") undefined)
                                     else (if ((length pE3) <= (n2 - n1))
 then (errorMsg "Array boundaries and increment are incompatible") undefined
 else q')
          storeValues l q' [] = q' l
          storeValues l q' (e:es) = deref' (ref' $ "Loc" ? (\l' -> storeValues (l ++ [l']) q' es)) e

instance ClassArray Ed
instance ClassArray Ev

-- Data Abstraction --

class Class c where
  class0 :: c -> Ed

instance Class (Dd,Dd,Scope) where
```

```
  class0 (pDc,pDo,scope) r q = pDc r $ \e c -> q (EvEnv (c <+> ("class",DvEnv c) <+> ("constructor",
 constructor pDo scope r c))) r
    where constructor pDo scope r c = DvX (\r' -> pDo ((scope(r,r')) <+> c <+> ("class",DvEnv c)))

instance Class (Dd,Dd) where
  class0 (pDc,pDo) r q = class0 (pDc,pDo,getScope r) r q


subclass :: (Ex e1, Ex e2) => (e1,e2) -> Ed
subclass (pE1,pE2) r q = eval pE1 r $ "Class" ? (\c1 r' -> eval pE2 r' $
 "Class" ? (\c2 -> q (EvEnv (c1 <+> (c2 :: Env) <+> ("constructor", constructor c1 c2)))))
  where constructor c1 c2 = DvX (\r q -> object(EvEnv c1) r $ "Object" ? \o1 r' ->
 object(EvEnv c2) (r'<+>o1)$"Object"? \o2 -> q (EvEnv (o1 <+> (o2 :: Env) <+> ("super", DvEnv o1))))

class Ex o => Object o where
  object :: o -> Ed
  object pE r q = eval pE r $ "Class" ? (\c r' -> construct c r' $ \e o ->
 q (EvEnv (c <+> o <+> ("this",DvEnv o))) r)
    where construct c r q = lookup' "constructor" c $ \(DvX d) c' -> d r q

instance Object Ev
instance Object Ed

-- Declarations ---------------------------------------------------------------
-- accum(Dd) -> Dd --
accum :: Dd -> Dd
accum pD r q = pD r (\e r' -> q e (r <+> r'))

-- elabSeq(Dd,Dd) -> Dd --
class ElabSeq x where
  elabSeq :: (x,x) -> x

instance ElabSeq Dd where
  elabSeq (pD1,pD2) r q = pD1 r $ \e r1 -> pD2 (r <+> r1) $ \e' r2 -> q undefined (r <+> r1 <+> r2)

-- elabCol(Dd,Dd) -> Dd --
elabCol :: (Dd,Dd) -> Dd
elabCol (pD1,pD2) r q = pD1 r $ \e r1 -> pD2 r $ \e' r2 -> q undefined (r <+> r1 <+> r2)


-- elabRec(Dd,Dd) -> Dd --
class ElabRec x where
  elabRec :: x -> Dd

instance ElabRec Dd where
  elabRec (pD) r q = \s -> case (y s) of
                             (AnsHalt _) -> errorMsg "ElabRec failed." undefined r s
                             (FlexibleAns (e,r',s')) -> q (EvBv Undefined) r' s'
    where y s = pD (r <+> (r'' s)) (\e r s -> (FlexibleAns (e,r,s))) s
          r'' s = case (y s) of
                    (AnsHalt _) -> r0
                    (FlexibleAns (e,r',s')) -> r'

instance ElabRec (Dd,Dd) where
  elabRec (pD1,pD2) = elabRec(elabCol(pD1,pD2))
```

```
-- useStaticScope --
useStaticScope :: Xd -> Xd
useStaticScope pX r q = pX (r <+> ("scope",DvScope static)) q

-- useDynamicScope --
useDynamicScope :: Xd -> Xd
useDynamicScope pX r q = pX (r <+> ("scope",DvScope dynamic)) q

static :: Scope
static (r,r') = r

dynamic :: Scope
dynamic (r,r') = r'

-- Generic ---------------------------------------------------------------
class AddEnv x where
  addEnv :: (x,Xd) -> Xd

instance AddEnv Dd where
  addEnv (pD,pX) r q = pD r $ \e r' -> pX r' q

choose :: Ex e => (e,Xd,Xd) -> Xd
choose (pE,pX1,pX2) r q = eval pE r $ deref' $ toBool $ \b r -> cond(pX1 r q, pX2 r q) b

-- Saltos ----------------------------------------------------------------
instance AddEnv Jd where
  addEnv (pJ,pC) r q = pC (r <+> r') q
    where r' = pJ (r <+> r') q

class AddLabel a where
  addLabel :: a -> Jd

instance AddLabel (Id,Cd,Scope) where
  addLabel (pI,pC,scope) r q = r0 <+> (pI,DvQ (\e r' -> pC (scope(r,r')) q))

instance AddLabel (Id,Cd) where
  addLabel (pI,pC) r = addLabel(pI,pC,getScope r) r

skipLabel :: Jd
skipLabel r q  = r0

instance ElabSeq Jd where
  elabSeq (pJ1,pJ2) r q = r <+> r1 <+> r2
    where r1 = pJ1 r q
          r2 = pJ2 r q

elab :: (Jd,Cd) -> Jd
elab (pJ,pC) r q = pJ r (\e r' -> pC r' q)

jump :: Id -> Cd
jump pI r q = lookup' pI r $ \(DvQ q') r' -> q' (EvBv Undefined) r'

-- Abstracao -------------------------------------------------------------
```

```
-- proc --
class Proc' p where
  proc :: p -> Ed

instance Proc' Xd where
  proc pX r q = q (EvDv (DvX pX)) r

instance Proc' Proc where
  proc pP r q = q (EvDv (DvProc pP)) r

instance Proc' (Xd, Scope) where
  proc (pX, scope) r q = q (EvDv (DvX (x' pX r))) r
    where x' x r = (\r' -> x (scope(r,r')))

instance Proc' (Proc, Scope) where
  proc (pP, scope) r q = q (EvDv (DvProc (p' pP r))) r
    where p' (Proc (a,p)) r = Proc (a, \r' -> p (scope(r,r')))

makeClosure :: (Dd,Xd) -> Xd
makeClosure (pD,pX) r q = pD r $ \e' r' -> pX r' $ \e'' r'' -> q e'' r

makeAbstraction :: (Xd,ImportOption, ExportOption) -> Xd
makeAbstraction (pX,pO1,pO2) r q = pX (imported pO1 r) $ \e r' -> q e (exported pO2 r')
  where imported Transparent r'' = r''
        imported (Opaque i) r'' = filter r'' i
        exported Visible r'' = r''
        exported (Encapsulated i) r'' = r <+> (filter r'' i)

-- bind --
class Bind b where
  bind :: b -> Dd

instance Ex e => Bind (Id, e) where
  bind (pI, pE) r q = eval pE r $ "Dv" ? \d r' -> q undefined (r0 <+> (pI, f d pI :: Dv))
    where f (DvEnv e) pI = DvEnv (e <+> ("classname", DvBv (BvString pI)))
          f d pI = d

instance Ex e => Bind (Id, e, Type) where
  bind (pI, pE, pT) r q = bind(pI, pE) r $ \e r' -> q undefined
 (r' <+> (Type pI, (DvBv (BvString pT))))

parametrize :: ([Ad],Xd) -> Proc
parametrize (pA,pX) = Proc (pA,\r q v -> addPar (reverse (zip pA v)) pX r q)
  where addPar [] pX' = pX'
        addPar (y:ys) pX' = addPar ys (body y pX')
        body :: (Ad,Ev) -> (Env -> Q -> Store -> Ans) -> Env -> Q -> Store -> Ans
        body (Ad (m,i),e) pX' r' q' = entry m ("Dv" ? (\e' r'' -> pX' (r'' <+> (i,e'))
 (\e'' -> exit m e (q' e'') e'))) e r'

-- entry
entry :: Mode -> (Ev -> Env -> Store -> Ans) -> Ev -> Env -> Store -> Ans
entry Value q = deref' $ ref' $ q
entry ValueAndResult q = entry Value q
entry Ref q = "Loc" ? q
entry Constant q = deref' $ ref' $ "Loc" ? (\l -> q (EvDv (DvLoc (const' l))))
```

```
entry Name q = ref' $ q

-- exit
exit :: Mode -> Ev -> (Env -> Store -> Ans) -> Dv -> Env -> Store -> Ans
exit ValueAndResult e q e' = (?) "Loc" (\l -> (?) "Loc" (cont (update l (\e'' -> q))) e') e
exit _ _ q _ = q

-- pipe(Ed, Xd) --
pipe :: Ex e => (e, (Ev -> Xd)) -> Xd
pipe (pE,pX) r q = eval pE r $ \e r' -> pX e r' q
```

E.3 Auxiliar.hs

Listing E.3: Listagem do módulo de funções auxiliares.

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE CPP #-}

module Auxiliar (
    update,
    cont,
    r0,
    s0,
    ref',
    deref',
    (<+>),
    const',
    cond,
    lookup',
    filter,
    subscript,
    checkType,
    static,
    dynamic,
    getScope
    ) where

import Prelude hiding (error,seq,filter,undefined)
import Domains
import Debug.Trace

-- r0 --
r0 :: Env
r0 = \i -> Unbound

-- a0 --
s0 :: Store
s0 = (\l -> if (l == Level) then level else (if (l == New) then new else Unused))
  where level = (StoreSv (SvRv (BvNum (NumInt 1))))
        new = (StoreSv (SvRv (BvNum (NumInt 0))))

-- static :: scope --
```

```
static :: Scope
static (r,r') = r

-- dynamic :: scope --
dynamic :: Scope
dynamic (r,r') = r'

-- update --
class Update x y where
  (<+>) :: x -> y -> x

instance Update Store (Loc,Sv) where
  (<+>) s (Loc (_,a),v) = \l -> case l of
                                  (Loc (_,a')) -> if (a == a')
                                                  then (StoreSv v)
                                                  else (s (Loc (Var,a')))
                                  otherwise -> s l
  (<+>) s (Input,v) = \l -> if (l == Input) then (StoreSv v) else (s l)
  (<+>) s (Output,v) = \l -> if (l == Output) then (StoreSv v) else (s l)

instance Update Env (Id,Dv) where
  (<+>) r (i,v) = r <+> (Id i,v)

instance Update Env (EnvId,Dv) where
  (<+>) r (i,v) = \i' -> if (i == i') then (EnvDv v) else (r i')

instance Update Env Env where
  (<+>) r r' = \i -> case (r' i) of
                      Unbound -> r i
                      d -> d

-- ref' --
ref' :: Q -> Ev -> Env -> Store -> Ans
ref' q e = new $ "Loc" ? (\l -> update l (\e' -> q (EvDv (DvLoc l))) e)

-- deref' --
deref' :: Q -> Ev -> Env -> Store -> Ans
deref' q e = case e of
               EvDv (DvLoc l) -> cont q l
               otherise -> q e

-- update --
update :: Loc -> Q -> Ev -> Env -> Store -> Ans
update l q e r s = case l of
                     Loc (Const,a) -> errorMsg ("Cannot update a const location.") undefined r s
                     Loc (Var, a) -> toSv (\e' r' s'->q (EvBv Undefined) r' (s' <+> (l,e'))) e r s


-- cont --
cont :: Q -> Loc -> Env -> Store -> Ans
cont q l r s = case (s l) of
                 Unused -> errorMsg ("Unused location: " ++ show l) undefined r s
                 (StoreSv s') -> (?) "Ev" q s' r s

-- new --
```

```
new :: Q -> Env -> Store -> Ans
new q r s = case (r (Id "allocation")) of
              EnvDv (DvBv (BvString "heap")) -> q (locToEv (Var,(0, getNew s))) r (newStore s)
              otherwise -> q (locToEv (Var,(getLevel s, getNew s))) r (newStore s)
  where newStore s = \l -> if ( l == New) then newNew (s New) else s l
        newNew (StoreSv (SvRv (BvNum (NumInt i)))) = (StoreSv (SvRv (BvNum (NumInt (i + 1)))))
        getNew s = case (s New) of
                     (StoreSv (SvRv (BvNum (NumInt i)))) -> i
                     otherwise -> -1
        getLevel s = case (s Level) of
                     (StoreSv (SvRv (BvNum (NumInt i)))) -> i
                     otherwise -> -1
        locToEv l = (EvDv (DvLoc (Loc l)))

-- const' --
const' :: Loc -> Loc
const' (Loc (_,a)) = Loc (Const,a)

-- cond --
class Cond x where
  cond :: (x,x) -> Bool -> x
  cond (pX1,pX2) b = if b then pX1 else pX2

instance Cond Xd
instance Cond (Env -> Store -> Ans)
instance Cond (Store -> Ans)
instance Cond Ans

-- lookup' --
lookup' :: Id -> Env -> (Dv -> Env -> Store -> Ans) -> Store -> Ans
lookup' i r q s = case (r (Id i)) of
                    Unbound -> errorMsg ("Unbound identifier: " ++ show i ) undefined r s
                    (EnvDv d) -> q d r s

-- filter --
filter :: Env -> [Id] -> Env
filter r [] = r0
filter r (i:is) = case (r (Id i)) of
                    Unbound -> (filter r is)
                    EnvDv d -> case (r (Type i)) of
                                 Unbound -> (filter r is) <+> ((Id i),d)
                                 EnvDv t -> ((filter r is) <+> ((Id i),d)) <+> ((Type i),t)

-- subscript --
subscript :: Array -> Q -> Int -> Env -> Store -> Ans
subscript (Array (l,n1,n2)) q n r s = if (n >= n1 && n <= n2)
                                      then q (EvDv (DvLoc (l !! (n - n1)))) r s
                                      else errorMsg ("Array Index out of bounds: " ++ show n)
                                        undefined r s

-- checkType --
checkType :: Env -> Id -> String -> Ev
checkType r i t = case (r (Type i)) of
                    EnvDv (DvBv (BvString t)) -> (EvBv (BvBool True))
                    otherwise -> (EvBv (BvBool False))
```

```
-- getScope --
getScope :: Env -> Scope
getScope r = case (r (Id "scope")) of
               (EnvDv (DvScope s)) -> s
               otherwise -> static
```

Referências Bibliográficas

Bigonha, R. S. (2004). Notas de Aulas de Semântica Denotacional, Último acesso: 15 de dezembro de 2012.

Curry, H. B. & Feys, R. (1958). *Combinatory Logic I.* North-Holland, Amsterdam.

Doh, K.-G. & Mosses, P. D. (2003). Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3--36.

Gayo, J. E. L. (2002). Reusable semantic specifications of programming languages. Em *6th Brazilian Symposium on Programming Languages, Brazil.*

Gordon, M. J. C. (1979). *The denotational description of programming languages - an introduction.* Springer-Verlag.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576--580.

Hughes, R. J. M. (1982). Super-combinators a new implementation method for applicative languages. Em *Proceedings of the 1982 ACM symposium on LISP and functional programming*, LFP '82, pp. 1--10, New York, NY, USA. ACM.

Iversen, J. & Mosses, P. D. (2004). Constructive action semantics for core ml. *Software IEE Proceedings*, 152(2):79--98.

Jensen, K. & Wirth, N. (1974). *PASCAL user manual and report.* Springer-Verlag New York, Inc., New York, NY, USA.

Johnstone, A.; Mosses, P. & Scott, E. (2010). An agile approach to language modelling and development. *Innovations in Systems and Software Engineering*, 6:145–153.

Kahn, G. (1987). Natural semantics. Em *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, pp. 22--39, London, UK, UK. Springer-Verlag.

Liang, S. (1997). *Modular Monadic Semantics and Compilation*. Tese de doutorado, Yale University. Adviser-Paul Hudak.

Liang, S.; Hudak, P. & Jones, M. (1995). Monad transformers and modular interpreters. Em *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Moggi, E. (1989). Computational lambda-calculus and monads. Em *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pp. 14--23, Piscataway, NJ, USA. IEEE Press.

Moggi, E. (1991). Notions of computation and monads. *Inf. Comput.*, 93(1):55--92.

Mosses, P. D. (1977). Making denotational semantics less concrete. Em *Proc. Int. Workshop on Semantics of Programming Languages, Bad Honnef*, number 41 in Bericht, pp. 102--109. Abteilung Informatik, Universität Dortmund.

Mosses, P. D. (1988). The modularity of action semantics. Internal Report IR–75, Dept. of Computer Science, Univ. of Aarhus. Revised version of a paper presented at a CSLI Workshop on Semantic Issues in Human and Computer Languages, Half Moon Bay, California, March 1987 (proceedings unpublished).

Mosses, P. D. (1996). Theory and practice of Action Semantics. Em *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland*, volume 1113 of *LNCS*, pp. 37--61. Springer.

Mosses, P. D. (1999). A modular SOS for Action Notation. BRICS Research Series RS-99-56, Dept. of Computer Science, University of Aarhus. Full version of **?**.

Mosses, P. D. (2001). The varieties of programming language semantics. Em *Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics: Akademgorodok, Novosibirsk, Russia*, volume 2244 of *PSI '02*, pp. 165--190, London, UK, UK. Springer-Verlag.

Mosses, P. D. (2002a). Fundamental concepts and formal semantics of programming languages. Lecture Notes. Version 0.2, available from http://www.brics.dk/ pdm. Relatório técnico.

Mosses, P. D. (2002b). Pragmatics of modular SOS. Em Ringeissen, C. & Kirchner, H., editores, *AMAST'02, 9th International Conference on Algebraic Methods and Software Technology, Reunion Island, France, Proceedings*, volume 2422 of *LNCS*, pp. 21--40. Springer.

Mosses, P. D. (2004). Modular structural operational semantics. *J. Logic and Algebraic Programming*, 60–61:195--228. Special issue on SOS.

Mosses, P. D. (2005). A constructive approach to language definition. *Journal of Universal Computer Science*, 11(7):1117--1134.

Mosses, P. D. (2009). Component-based semantics. Em *Proceedings of the 8th international workshop on Specification and verification of component-based systems*, SAVCBS '09, pp. 3--10, New York, NY, USA. ACM.

Mosses, P. D. & New, M. J. (2009). Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49 – 66. Proceedings of the Fifth Workshop on Structural Operational Semantics (SOS 2008).

Moura, H. P. (1996). An Overview of Action Semantics. Em *I Simpósio Brasileiro de Linguagens de Programação*.

Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. Relatório técnico, Computer Science Department, Aarhus University, Aarhus, Denmark.

Scott, D. (1975). Data types as lattices. Em Müller, G.; Oberschelp, A. & Potthoff, K., editores, *ISILC Logic Conference*, volume 499 of *Lecture Notes in Mathematics*, pp. 579–651. Springer Berlin / Heidelberg. 10.1007/BFb0079432.

Tirelo, F.; Bigonha, R. S. & Saraiva, J. (2005). Semântica Multidimensional de Linguagens de Programação – Proposta de Tese de Doutorado. Relatório técnico, Laboratório de Linguagens de Programação, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais.

Tirelo, F.; Bigonha, R. S. & Saraiva, J. (2008). Disentangling denotational semantics definitions. *Journal of Universal Computer Science*, 14(21):3592--3607.

Tirelo, F.; Bigonha, R. S. & Saraiva, J. (2009). *Semântica incremental de linguagens de programação*. Tese de doutorado, UFMG.

Wansbrough, K. & Hamer, J. (1997). A modular monadic action semantics. Em *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pp. 13--13, Berkeley, CA, USA. USENIX Association.

Watt, D. A. & Findlay, W. (2004). *Programming language design concepts*. Wiley.

Zhang, Y. & Xu, B. (2004). A survey of semantic description frameworks for programming languages. *SIGPLAN Not.*, 39:14--30.