SEMÂNTICA INCREMENTAL DE LINGUAGENS DE PROGRAMAÇÃO

FABIO TIRELO
Orientador: Roberto da Silva Bigonha
Co-Orientador: João Alexandre Baptista Saraiva

SEMÂNTICA INCREMENTAL DE LINGUAGENS DE PROGRAMAÇÃO

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Belo Horizonte, Março de 2009

Tirelo, Fabio
T596S Semântica Incremental de Linguagens de Programação [manuscrito] / Fabio Tirelo. – 2009.
xii, 121f., enc. : il.

Orientador: Roberto da Silva Bigonha
Co-orientador: João Alexandre Baptista Saraiva

Tese (doutorado) – Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

1. Computação – Teses. 2. Linguagens de Programação – Teses. 3. Semântica Processamento de Dados – Teses. I. Bigonha, Roberto da Silva. II. Saraiva, João Alexandre Baptista. III. Universidade Federal de Minas Gerais, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*33

Agradecimentos

Agradeço a todos que contribuíram de alguma maneira para a conclusão deste trabalho.

Inicialmente a toda a minha família, em especial a Leonor, minha mãe, pelo apoio constante, não só durante o doutorado, mas também em todos os momentos.

A todos os meus amigos pelo companheirismo e pelos bons momentos, que foram essenciais para suportar as etapas mais difíceis do trabalho. Ao Alexandre Bustamante, pela paciência e por estar sempre do meu lado, dando apoio e coragem durante todo o processo. Às amigas Érika e Ludmila Andrade, pela excelente acolhida na volta ao Brasil. Ao amigo Vladimir Oliveira di Iorio, pelo exemplo e pela motivação em iniciar, continuar e concluir este trabalho. Às amigas Ana Paula Amazonas Soares e Raquel Hennig, pelo companheirismo e acolhida nos meses em que passei no exterior. Às amigas Cassia Prates-Clarke e Adriana Andrade Oliveira, pelas palavras de apoio, por sempre motivar a concluir este trabalho, e pelo incentivo a buscar a experiência de fazer parte deste trabalho no exterior.

Aos colegas e amigos da PUC Minas, em especial Juliana do Amaral Baroni, Lucila Ishitani, Maria Cristina Martins de Andrade (Tininha), Marcelo Souza Nery, Marco Túlio Oliveira Valente, Marlete Barroso Schreiber, Nelma Rocha, Rodrigo Baroni, Raquel A. F. Mini, Rosilane Ribeiro da Mota, Silvio Jamil F. Guimarães, Soraia Lúcia da Silva, Zenilton Kleber G. Patrocínio Jr., pela motivação em continuar este trabalho, pelas palavras de apoio, e, por tantas vezes, auxiliarem no cumprimento de atividades a mim designadas.

Ao professor Roberto Bigonha, pela orientação precisa, por confiar em meu trabalho, pelas palavras de apoio e motivação nos momentos mais necessários, pela empolgação pelo projeto mesmo nas épocas em que eu não acreditava que seria possível concluí-lo.

Ao professor João Saraiva, pela excepcional acolhida em Portugal e por todo o apoio dado durante a execução do trabalho. À professora Mariza Andrade da Silva Bigonha, pelas muitas e valiosas contribuições em diversos momentos deste trabalho. Ao professor Peter Mosses, da Universidade de Walles em Swansea, pelas valiosas sugestões que levaram à versão final do trabalho.

Aos colegas e amigos do Laboratório de Linguagens de Programação da UFMG, em especial, a Eduardo Santos Cordeiro, Tays Cristina Soares, Italo Giovanni, Leonardo Passos e Kristian Magnani, pelos bons momentos durante o tempo em que compartilhamos o laboratório, pela confiança e pelo apoio dado mesmo após o término do período em que trabalhamos juntos.

Aos colegas e amigos do gabinete dos "Sem Estatuto"na Universidade do Minho, em especial, ao Jácome Cunha, João Paulo Fernandes, Paulo Filipe Silva, Miguel Vilaça e Ricardo Vilaça, pela excelente recepção e pelos ótimos momentos compartilhados durante o período em que morei em Portugal.

Aos professores Marco Tulio Oliveira Valente, Roberto Ierusalimskchy, Mariza D Bigonha e Newton José Vieira, por aceitarem participar da banca e pelas valiosas sugestões dadas durante a defesa.

À Secretaria do Programa de Pós-Graduação em Ciência da Computação da UFMG, em especial, a Renata Viana Moraes Rocha e a Sheila ?, pela simpatia e pela disponibilidade em fornecer as informações solicitadas.

À CAPES, pela bolsa de doutorado no período que passei no exterior, e à FAPEMIG, pelo apoio financeiro ao projeto de pesquisa.

A todos os outros que contribuíram direta ou indiretamente para a conclusão deste trabalho e que não tenham sido citados aqui.

Resumo

Este trabalho de tese aborda uma nova metodologia para o problema da escalabilidade em Semântica Denotacional, denominada Semântica Incremental. Neste trabalho, considera-se que uma linguagem de programação possa ser definida por meio de uma sequência de especificações, onde cada especificação inclui novos conceitos, com a possibilidade de fornecer novas reinterpretações às equações existentes na especificação anterior. Os conceitos mais importantes utilizados são a vagueza, no sentido que equações semânticas são vagas em relação a conceitos a serem abordados em especificações mais avançadas, e funções de transformação, que permitem a evolução das equações semânticas ao longo da sequência de especificações. Neste contexto, a vagueza, comum em definições informais, traz melhorias na legibilidade de definições formais, uma vez que permite a remoção do entrelaçamento de equações semânticas.

Como principais contribuições do trabalho, podem-se citar: a definição de uma nova metodologia para a escrita de definições incrementais e modulares; a definição de um ambiente de desenvolvimento para a metodologia, que permite a escrita de protótipos de interpretadores para linguagens definidas, incluindo análises léxica, sintática e semântica; utilização e formalização da vagueza em métodos formais.

Abstract

This thesis presents a new methodology for the scalability problem in Denotational Semantics, named Incremental Semantics. This work considers that a programming language can be defined as a sequence of specifications, so that each new specification includes new elements into an existing one, and, if necessary, provides new reinterpretation to existing equations. The main concepts of the method are vagueness, in the sense that semantic equations are vague with respect to concepts to be presented in advanced specifications, and transformation functions, which allows semantic equations to evolve throughout specification sequences. In this context, vagueness, usually applied in informal definitions, brings about improvements in the readability of formal definitions, once semantic equations may be disentangled.

The main contributions of this work are: the definition of a new methodology for writing incremental and modular definitions; the implementation of a programming environment for applying the methodology, which generates interpreter prototypes for a language, including lexer, parser, and semantics; usage and formalization of vagueness in formal methods.

Conteúdo

Lista de Tabelas

Lista de Figuras

Listagens

Capítulo 1

Introdução

Apesar da grande disponibilidade de formas de definição de semântica de linguagens de programação e do seu amplo escopo de aplicação, o uso real dos diversos arcabouços semânticos é ainda bastante reduzido. Segundo *Mosses* (2001), isto se deve principalmente às dificuldades em se ler e escrever definições semânticas e ao pequeno número de ferramentas de apoio para escrita e verificação das especificações, assim como para geração automática de interpretadores e compiladores. Itens que também levam à popularização de um formalismo são a existência de bibliotecas ricas, documentação, definição formal e suporte ferramental.

De acordo com *Zhang and Xu* (2004), um formalismo popular para definição semântica deveria possuir como características básicas: legibilidade, modularidade, existência de recursos adequados para abstração, facilidade em realizar comparações entre linguagens, elementos teóricos que propiciem o raciocínio e a prova de propriedades, aplicabilidade, e existência de ferramentas de suporte.

Essas dificuldades ficam muito evidenciadas na definição da semântica formal de linguagens de grande porte, pois o número de detalhes que devem ser tratados costuma ser tão grande que a tarefa de especificação torna-se muito trabalhosa e sujeita a erros. Descrições semânticas de linguagens de grande porte são inerentemente complexas devido ao grande número de detalhes entrelaçados com que se deve lidar (*Tirelo et al.* 2009). Desta maneira, é desejável que tais especificações sejam modulares e extensíveis, e possam ser escritas e processadas de modo incremental, de forma que construções possam ser sucessivamente adicionadas à definição de uma linguagem base. Além disso, este processo incremental não deve exigir a redefinição de módulos previamente definidos e, adicionalmente, deve requerer

por parte do projetista da linguagem apenas o uso de recursos simples.

No entanto, linguagens de programação de grande porte usualmente são compostas por construções que nem sempre são ortogonais, e, para as quais, não é trivial escrever definições separadas. Um exemplo deste problema pode ser encontrado nas definições de métodos e tratamento de exceções em Java: para a definição do comando *return* deve-se estar ciente de possíveis blocos *finally* que devem ser executados antes de restaurar o controle da execução para a função chamadora, seja por retorno normal ou via mecanismo de tratamento de exceções. Além disso, como a semântica de um programa é produzida pelas equações semânticas de maneira *top-down*, cada construção pode ser responsável por preparar o contexto para seus possíveis constituintes. Como consequência, equações semânticas são não apenas responsáveis por especificar o significado de construções, mas também por definir como tais construções interagem.

Assim, para a definição da interação entre duas construções, pelo menos uma das equações semânticas correspondentes deve estar ciente da existência da outra. Por exemplo, o problema da ocorrência de comandos *return* dentro de blocos *try* pode ser resolvido por meio da redefinição da continuação associada ao sequenciador. Como nas abordagens tradicionais para semântica denotacional (ver Capítulo 3) há um mapeamento de um para um das construções linguísticas nas equações semânticas, tais interações necessariamente induzem elementos entrelaçados em pelo menos uma das equações[1].

Esta propriedade tem impacto direto na modularidade de definições de semântica denotacional de linguagem, uma vez que a descrição de uma construção pode conter elementos não relacionados, o que viola o princípio da alta coesão (*Meyer* 1997). Além disso, a definição incremental de uma linguagem geralmente requer que módulos previamente definidos sejam reescritos toda vez que alguma construção é definida, tornando o processo de escrita e reescrita tedioso e sujeito a erros. Do ponto de vista do leitor, informações cruciais a respeito da linguagem podem ser obscurecidas por diversos detalhes presentes nas equações semânticas, e, por isso, compreender completamente certas construções pode ser uma tarefa bastante complicada.

Por outro lado, em desenvolvimento de sistemas de software, altos níveis de modularidade em sistemas de grande porte têm sido alcançados por meio da aplicação de técnicas de decomposição aderentes às necessidades dos sistemas desenvolvidos (*Dijkstra* 1976, *Parnas*

[1]Isto é consequência direta do *pigeon-hole principle*. Tal observação não é verdadeira para sistemas baseados em semântica operacional estruturada, pois mais de uma cláusula pode ser utilizada para definir uma construção.

1972, *von Staa* 2000). Nos últimos anos, um importante avanço nas técnicas de modularização foi obtido a partir das propostas de decomposição multidimensional (*Ossher and Tarr* 2000, 2001) e da orientação por aspectos (*Kiczales et al.* 1997), pois ao se quebrarem as barreiras de uma única forma de decomposição dominante, foi possível modularizar elementos de projeto que não eram adequadamente tratados pelas técnicas até então utilizadas.

O objetivo principal deste trabalho de doutorado é aperfeiçoar as técnicas de definição da semântica denotacional de linguagens de programação, oferecendo meios apropriados para tratamento dos elementos inerentes às linguagens de grande porte que ainda são impedimentos à escrita de definições modulares, em especial o entrelaçamento de equações decorrente da interdependência de construções. A técnica proposta utiliza recursos da orientação por aspectos para melhoria do processo de escrita incremental da semântica denotacional de linguagens de programação, e se baseia em um recurso fundamental à legibilidade de definições informais, a *vagueza*. Com efeito, ao se ensinar uma linguagem de programação, um professor apresenta inicialmente os conceitos fundamentais, sem se preocupar com a forma como aqueles conceitos se relacionam com outros que serão posteriormente definidos. Somente quando fizer sentido à lógica de apresentação dos conceitos tais detalhes devem ser especificados. Entretanto, para a inclusão de novos elementos, alguns conceitos devem ser revisitados, sendo necessária a adaptação do que foi visto à luz do que será apresentado, sem que isso necessariamente implique na sua redefinição completa.

Organização da Tese. O Capítulo 2 discute a modularidade de sistemas de software, métricas de modularidade e alguns avanços recentes na área. O Capítulo 3 apresenta as soluções mais importantes para a definição da semântica de linguagens de programação e para a modularidade de semântica denotacional. O Capítulo 4 apresenta a principal contribuição deste trabalho, que é uma técnica orientada por aspectos para a melhoria do processo de definição incremental da semântica denotacional de linguagens de programação, denominada *Semântica Incremental*. O Capítulo 5 apresenta a linguagem de especificação Notus, cujo objetivo é permitir a definição incremental de uma linguagem de programação, incluindo elementos sintáticos e semânticos. O Capítulo 6 apresenta uma avaliação do trabalho, de acordo com critérios previamente discutidos. Finalmente, o Capítulo 7 apresenta as considerações finais.

Capítulo 2

Modularidade de Sistemas de Software

Uma das qualidades mais importantes que se deseja alcançar na implementação de um sistema de software de grande porte é, junto com a correção e a eficiência, a sua modularidade. Um módulo é um artefato de programação que pode ser desenvolvido e compilado separadamente de outras partes que compõem o programa (*Dijkstra* 1976, *Parnas* 1972, *von Staa* 2000). Ao possibilitar que se trate separadamente cada componente de um programa, a boa divisão modular traz consigo a redução dos seus custos de desenvolvimento e manutenção. O módulo é um dos recursos mais importantes para quebrar a complexidade inerente de sistemas de grande porte.

Especificações de semântica formal de linguagens de grande porte são em geral bastante complexas e apresentam diversos problemas de modularidade em função de limitações dos modelos correntes. Neste capítulo, apresentam-se os principais conceitos de modularidade de sistemas, que serão aplicados nos Capítulos 3 a 5 na modularidade de definições de semântica denotacional.

Organização do Capítulo. A Seção 2.1 apresenta um breve histórico das soluções mais relevantes para modularidade de sistemas, e a Seção 2.2 apresenta as principais métricas para se avaliar o grau de modularidade de um dado sistema. Em seguida, são descritos na Seção 2.3 os principais conceitos da *Programação Orientada por Aspectos* e como esta metodologia permite melhorar a modularidade de sistemas de software; esta seção é uma versão resumida de *Tirelo et al.* (2004). Finalmente, a Seção 2.4 apresenta as conclusões

do capítulo.

2.1 Evolução das Técnicas de Modularidade

Na programação estruturada, a modularização se limita à implementação de abstrações dos comandos e expressões (procedimentos e funções) necessários à realização de uma tarefa. Com o advento da orientação por objetos (*Kay* 1993, *Meyer* 1997, *Nygaard and Dahl* 1981), pode-se obter um grau mais elevado de modularização, por ser possível realizar abstrações de estruturas de dados, tipos e de suas operações por meio da implementação de interfaces. Além disso, com o polimorfismo (*Cardelli and Wegner* 1985), é possível definir comportamentos distintos para interface comuns a objetos relacionados, sem a necessidade de permitir o acesso a elementos básicos da implementação.

Desenvolvedores criam sistemas a partir de requisitos que podem ser classificados como *requisitos funcionais*, que constituem o objetivo final do sistema, e *requisitos não-funcionais*, que compreendem elementos específicos de projeto, muitas vezes sem relação direta com o problema em questão. Por exemplo, em um sistema de processamento de cartões de crédito, um requisito funcional é o processamento de pagamentos, ao passo que a geração de registro de operações (*logging*), a garantia de integridade de transações, a autenticação de serviços, a segurança e o desempenho são requisitos não-funcionais.

Ao permitir codificar os objetos do problema em questão, a orientação por objetos permite a boa modularização dos requisitos funcionais do sistema (*Meyer* 1997). No entanto, um problema que este modelo não é capaz de resolver adequadamente está relacionado à modularização de requisitos não-mapeáveis diretamente em uma ou poucas classes de um sistema, pois estes tendem a se espalhar por todo o código do programa (*Kiczales and Hilsdale* 2001, *Kiczales et al.* 1997) e, por esta razão, são denominados preocupações ortogonais.

A Programação Orientada por Aspectos foi desenvolvida por *Kiczales et al.* (1997) a partir da constatação de que certos requisitos não se encaixam em um único módulo de programa, ou pelo menos em um pequeno conjunto de módulos altamente relacionados. Em sistemas orientados por objetos, a unidade de modularização é a classe, e as preocupações ortogonais se espalham por múltiplas classes.

A Separação Multidimensional de Preocupações, proposta por *Ossher and Tarr* (2000, 2001), foi desenvolvida a partir da observação de que a boa decomposição dos requisitos

de sistema exige que mais de uma forma de decomposição seja utilizada. Na terminologia utilizada, diz-se que os requisitos do sistema formam um espaço multidimensional, em que cada requisito ocupa uma dimensão; em sua dimensão, um requisito pode evoluir de maneira independente dos demais. Entretanto, ao se utilizar um único mecanismo de decomposição, torna-se necessário adaptar o projeto de um requisito aos processos previstos pela técnica utilizada. Assim, nessa metodologia, cada requisito é decomposto utilizando-se a técnica que for mais conveniente e, no fim, definem-se regras de composição, cujo objetivo é integrar os módulos desenvolvidos independentemente.

2.2 Métricas de Modularidade

Para se avaliar o grau de modularidade de um sistema, utilizam-se métricas como *coesão*, *conectividade* e *acoplamento*, descritos a seguir.

Coesão Modular. A coesão modular é uma medida do grau de relacionamento entre os constituintes de um módulo, tal que um módulo é coeso se os seus elementos forem fortemente relacionados. A coesão de um módulo pode ser medida de acordo com os níveis mostrados na Tabela 2.1, segundo a divisão feita por *Myers* (1975), sendo o maior grau de coesão o funcional, e o menor, o coincidental. Geralmente é mais fácil fazer manutenção em um módulo coeso, além de ser mais fácil reutilizá-lo em sistemas diferentes daquele no qual tenha sido primeiramente implementado.

Conectividade e Acoplamento de Módulos. A conectividade de um sistema é uma medida do número de relações que há entre os seus módulos, sendo desejável que o número de conexões seja o menor possível. Uma forma intuitiva de compreender a conectividade consiste em modelar um sistema por meio de um grafo, cujos vértices são seus módulos, e cada aresta indica se há comunicação entre os módulos representados por suas extremidades. O grau de conectividade do sistema está relacionado à densidade de arestas do grafo correspondente. Assim, se este grafo for esparso, diz-se que o sistema possui baixa conectividade, ao passo que, se o grafo for denso, então o sistema possui alta conectividade.

O acoplamento por sua vez é uma medida da complexidade de uma conexão entre módulos, servindo como medida do grau de independência entre os módulos de um sistema. Assim como se deseja que o número de conexões seja o menor possível, espera-se, em um sistema modular, que as conexões entre os módulos sejam tão fracas quanto possível. Segundo

Classificação	**Características**
funcional	o módulo realiza uma e exatamente uma função
informacional	o módulo implementa um tipo abstrato de dados
comunicacional	as ações do módulo são dependentes de dados comuns
procedimental	as ações do módulo estão relacionadas à resolução de um problema e existe uma ordem específica de execução
clássica	as ações do módulo operam sobre dados diferentes e existe uma ordem de execução pré-definida
lógica	o módulo possui mais de uma ação a ser selecionada pelo usuário por meio de parâmetros para uma ação seletora
coincidental	as ações de um módulo não possuem relação entre si

Tabela 2.1: Níveis de Coesão Modular

Myers (1975), o acoplamento pode ser dividido nos níveis mostrados na Tabela 2.2, sendo preferível que dois módulos sejam desacoplados. A presença de conexão entre dois módulos tem impacto direto no custo de manutenção, uma vez que, ao se alterar um módulo, pode ser necessário fazer adaptações nos módulos com que ele se relaciona. A probabilidade de ser necessário ajustar um módulo em razão da modificação em outro está diretamente relacionada ao grau de dependência entre os dois.

2.3 Programação Orientada por Aspectos

Especificações da semântica denotacional de linguagens de programação apresentam muitos problemas de modularidade em função do entrelaçamento de conceitos expressos em suas equações semânticas, que não podem ser plenamente resolvidos pelas técnicas correntes, apresentadas no Capítulo 3.

Neste contexto, a Programação Orientada por Aspectos, definida por *Kiczales et al.* (1997), oferece novos recursos para modularização de sistemas que podem ser aplicados à estruturação de definições de semântica denotacional. A sua apresentação neste capítulo será baseada na plataforma orientada por objetos, onde é mais comumente utilizada, para prover o arcabouço necessário para o entendimento da técnica e sua aplicação no ambiente denotacional.

Classificação	**Características**
desacoplados	não há comunição entre os módulos, isto é, eles são completamente independentes
por informação	a comunicação entre módulos é feita por meio de chamadas de funções com parâmetros passados por valor
por referência	a comunicação entre os módulos é realizada por meio de chamadas de funções com parâmetros passados por referência
por controle	um módulo deve conhecer a estrutura de outro, ou então passar um parâmetro para uma de suas operações com o objetivo de controlar o seu fluxo de execução
por dado externo	os módulos têm acesso a um conjunto comum de dados, sem que a alteração destes dados afete todos os módulos envolvidos
por dado comum	os módulos têm acesso a um conjunto comum de dados, e a alteração de algum destes dados afetar todos os módulos envolvidos
por conteúdo	um módulo acessa de forma direta, mas não autorizada, dados não exportados por outro

Tabela 2.2: Níveis de Acoplamento entre Módulos

A programação orientada por aspectos tem por objetivo oferecer meios para a modularização de demandas do sistema que não podem ser adequadamente modularizadas utilizando as técnicas de decomposição vigentes, em particular, a orientação por objetos. Tais demandas são denominadas preocupações ortogonais (*Kiczales et al.* 1997). Por estar espalhado em diversos módulos do sistema, torna-se difícil conceber, implementar e modificar código relacionado à implementação de uma preocupação ortogonal. Essa dificuldade é exemplificada na classe definida na Listagem 2.1, extraída de *Gradecki and Lesieck* (2003).

Nesse código, alguns problemas podem ser identificados: a parte "Outros dados membros" (linha 4) não se refere ao objetivo principal da classe; o método performSomeOperation (linhas 8–17) faz mais operações do que realizar a operação objetivo: registra e autentica a operação, sincroniza *threads*, valida contrato e gerencia cache; não é claro se as operações save e load (linhas 21 e 22), que realizam a gerência de persistência, fazem parte do objetivo principal da classe.

Outro problema é que a modificação em alguns dos requisitos, como a alteração da política de registro de operações, pode originar modificações em diversas partes do programa, o que é uma tarefa desagradável e propensa a erros.

Assim, a orientação por aspectos preocupa-se com duas formas de manifestação da falta de modularidade de um sistema: *espalhamento* e *intrusão*.

Espalhamento (*scattering*) diz respeito a código para implementação de preocupações ortogonais estar disperso no programa; por exemplo, o código de geração de registro de operações ou o código para verificar coerência de cache tende a ficar distribuído ao longo de toda a implementação. O espalhamento apresenta como principal desvantagem o fato de a preocupação ortogonal não poder ser encapsulada em uma unidade modular, além do seu impacto direto na conectividade do sistema.

Intrusão (*tangling*) diz respeito à confusão gerada por códigos de mais de uma preocupação ortogonal estarem presentes em uma única região do programa, como ocorre no método performSomeOperation da Listagem 2.1. A intrusão tem impacto direto na coesão de um módulo uma vez que este contém, além do tratamento de seu objetivo principal, trechos de código responsáveis pelo tratamento de algum outro requisito a que seja transversal.

Segundo *Gradecki and Lesieck* (2003), as implicações desses problemas são diversas. Primeiramente, observa-se dificuldade em se fazer o rastreamento do programa, pois a implementação simultânea de vários requisitos em um único método torna obscura a correspondência

```
1  public class SomeBusinessClass extends OtherBusinessClass {
2
3      "Dados membros do módulo"
4      "Outros dados membros: stream de logging, flag de consistência de dados"
5
6      "Métodos redefinidos da superclasse"
7
8      public void performSomeOperation(OperationInformation info) {
9          "Garante autenticidade"
10         "Garante que info satisfaça contrato"
11         "Bloqueia o objeto para garantir consistência caso outras threads o acessem"
12         "Garante que a cache está atualizada"
13         "Faz o logging do início da operação"
14         "REALIZA A OPERAÇÃO OBJETIVO"
15         "Faz o logging do fim da operação"
16         "Desbloqueia o objeto"
17     }
18
19     "Outras operações semelhantes à anterior"
20
21     public void save( PersistenceStorage ps) { "..." }
22     public void load( PersistenceStorage ps) { "..." }
23
24 }
```

Listagem 2.1: Exemplo de Entrelaçamento de Código – Extraído de *Gradecki and Lesieck* (2003)

entre os requisitos e suas implementações, resultando em um mapeamento pobre entre os dois.

Além disso, obtém-se *baixa produtividade*, pois a implementação simultânea de várias preocupações em um módulo pode desviar a atenção do desenvolvedor do objetivo principal para os periféricos, levando à ocorrência constante de erros. Outro ponto importante é que esse modelo possui *baixo grau de reúso*, pois, dado que um único módulo implementa várias preocupações, outros sistemas que precisarem de implementar funcionalidades semelhantes podem não ser capazes de utilizar prontamente os módulos correspondentes, diminuindo ainda mais a produtividade do desenvolvedor.

Por esses pontos, pode-se perceber que o código apresenta *pouca qualidade interna*, com baixa coesão modular e alto grau de acoplamento de módulos, e com possíveis problemas ocultos, pois ao se lidar com diversas preocupações ao mesmo tempo, o desenvolvedor pode não dar atenção suficiente a uma ou mais preocupações.

Todos estes problemas levam à produção de código de *difícil evolução* ou *baixa extensibilidade* (*Meyer* 1997), visto que modificações posteriores no sistema podem ser dificultadas pela pouca modularização. Por exemplo, modificações na política de consistência da cache ou na política de validação de contratos podem levar à reorganização de diversos módulos de um sistema.

2.3.1 Metodologia de Desenvolvimento

Segundo *Gradecki and Lesieck* (2003), o desenvolvimento de software orientado por aspectos é realizado em três fases: a decomposição, a implementação e a recomposição de requisitos.

A *decomposição de preocupações* consiste na separação das preocupações ortogonais dos requisitos funcionais do sistema. No exemplo da classe SomeBusinessClass da Seção 2.3, pode-se identificar as seguintes preocupações ortogonais à operação objetivo: autenticação e registro de operação, sincronização de *threads*, validação de contrato, coerência de cache e persistência. A operação objetivo deve ser implementada na classe; todas as demais preocupações devem ser implementadas em outros módulos.

Segundo *Atkinson and Kühne* (2003), preocupações ortogonais são classificadas em infra-estrutura, de serviços e de paradigmas. Preocupações de infra-estrutura são responsáveis por coordenação (escalonamento e sincronização), distribuição (tolerância a falhas, comu-

nicação, serialização e replicação) e persistência. Preocupações de serviços são representadas por segurança, recuperação de erros, operações de retrocesso (*undo*), rastreamento e registro de operações. Exemplos de preocupações de paradigma são os padrões *visitor* e *observer* descritos por *Gamma et al.* (1995).

Após a decomposição, realiza-se a *implementação em separado*. No exemplo da classe SomeBusinessClass, deve-se implementar as unidades de lógica do negócio, registro de operações, autorização, entre outros, em módulos separados. Para a implementação de cada módulo, utilizam-se os recursos da programação orientada por objetos padrão. Por exemplo, a geração de registro de operações pode ser feita por meio da implementação de classes e interfaces específicas para o problema, possivelmente organizadas por meio de padrão de projetos.

Após a implementação separada de cada preocupação, especificam-se *as regras de recomposição do sistema*. Estas regras são implementadas em módulos denominados *aspectos*. Os aspectos definem como as preocupações são compostas para formar o sistema, em um processo denominado *costura* (*weaving*). Ainda no exemplo da classe SomeBusinessClass, um aspecto pode conter regras que definem os pontos do programa onde chamadas de métodos de registro de operações devem ser inseridas.

2.3.2 Linguagens Orientadas por Aspectos

Algumas linguagens e arcabouços de desenvolvimento permitem a implementação de programas orientados por aspectos, dentre eles, destacam-se as implementações para as linguagens: Java, representada por *AspectJ* (*Kiczales et al.* 2001), *Hyper/J* (*Ossher and Tarr* 2000, 2001) e AspectWerkz (*AspectWerks* 2005); C++, com a implementação de *AspectC++* (*Spinczyk et al.* 2002); C, com a implementação de *AspectC* (*Coady et al.* 2001); C#, com as linguagens *AspectC#* (*Kim* 2002) e *Eos* (*Rajan and Sullivan* 2003, 2005). Existem também propostas para a AOP independente de linguagem (*Lafferty and Cahill* 2003).

A proposta original de programação orientada por aspectos feita por *Kiczales et al.* (1997) utiliza uma extensão da linguagem funcional *Scheme* como linguagem hospedeira. Ainda no paradigma funcional, podem-se citar a implementação de *AspectH*, para a linguagem Haskell feita por *Andrade et al.* (2004), e a implementação de *Tucker and Krishnamurthi* (2003) para a linguagem Scheme. Destaca-se também o trabalho de *Lämmel* (1999), no qual se propõe uma técnica geral de compilação para linguagens funcionais orientadas por

aspectos.

As linguagens orientadas por aspectos oferecem recursos para resolução de dois problemas de entrelaçamento de código: a *transversalidade dinâmica*, que permite definir implementação adicional em pontos bem definidos da execução do programa, e a *transversalidade estática*, que afeta as assinaturas dos tipos de um programa. Nesta seção, optou-se por mostrar os principais conceitos de orientação por aspectos, sem detalhar as diversas possibilidades e recursos oferecidos. Para maior detalhamento, sugerem-se: *Tirelo et al.* (2004); *Kiczales et al.* (2001); *Laddad* (2003); e *Gradecki and Lesieck* (2003).

2.3.2.1 Transversalidade Dinâmica

Na transversalidade dinâmica, *pontos de junção* (*joinpoints*) são definidos como pontos da execução do programa. Exemplos de pontos de junção são as chamadas de métodos e de construtores de uma classe. Associados aos pontos de junção mais dois conceitos são importantes: *conjuntos de junção* e *regras de junção*.

As construções do programa que contêm pontos de junção são denominadas *conjuntos de junção* (*pointcuts*) e têm a função de reunir informações a respeito do contexto destes pontos. Os códigos relativos aos requisitos transversais que devem ser executados em pontos de junção são denominados *adendos* (*advices*).

Linguagens orientadas por aspectos possuem como elementos básicos: um modelo de definição de *pontos de junção*, uma forma de identificar pontos de junção, isto é, definir conjuntos de junção, e uma forma de interferir na execução de pontos de junção. Estes elementos são encapsulados em módulos denominados *aspectos*, que podem ser definidos como unidades de implementação modular das regras de recomposição do sistema e contêm definições de conjuntos de junção e adendos.

Considere, por exemplo, a definição em AspectJ do aspecto LoggingAspect da Listagem 2.2, que define os seguintes conjuntos de junção:

- publicMethods: engloba os pontos de junção de execução dos métodos públicos do programa;
- logObjectCalls: engloba os pontos de junção de execução dos métodos da classe Logger;
- loggableCalls: engloba os pontos de junção de execução dos métodos públicos do programa que não pertençam à classe Logger.

```
1  public aspect LoggingAspect {
2
3      pointcut publicMethods(): execution(public * *(..));
4      pointcut logObjectCalls() : execution(* Logger.*(..));
5      pointcut loggableCalls() : publicMethods() && ! logObjectCalls();
6
7      before(): loggableCalls() {
8          Logger.logEntry(thisJoinPoint.getSignature().toString());
9      }
10
11     after(): loggableCalls() {
12         Logger.logExit(thisJoinPoint.getSignature().toString());
13     }
14
15 }
```

Listagem 2.2: Aspecto Para Logging

São também definidos os adendos before e after, contendo os códigos para serem executados antes e depois das execuções dos métodos identificados nos conjuntos de junção indicados. Na regra before, define-se o código a ser executado antes da execução dos métodos públicos, que corresponde à geração do registro de entrada no método; este código corresponde à chamada do método doEntry da classe Logger, passando-lhe a assinatura do ponto de junção[1]. Da mesma forma, na regra after, define-se o código a ser executado após a execução dos métodos públicos, que corresponde à geração do registro de saída do método.

2.3.2.2 Transversalidade Estática

Algumas implementações de aspectos, como por exemplo a implementação de padrões de projeto (*Gamma et al.* 1995, *Hannemann and Kiczales* 2002), necessitam de recursos para alterar tanto o comportamento das classes em tempo de execução, quanto a sua estrutura estática.

A transversalidade dinâmica, obtida a partir de adendos, permite modificar o comportamento da execução do programa, ao passo que a transversalidade estática permite redefinir

[1] A expressão thisJoinPoint.getSignature().toString() obtém uma cadeia de caracteres que corresponde à assinatura do método que contém o ponto de junção ao qual a regra se aplica (thisJoinPoint)

a estrutura estática dos tipos – classes, interfaces ou outras estruturas – e o seu comportamento em tempo de execução. Dentre os recursos mais importantes de linguagens orientadas por aspectos, destacam-se modificação da hierarquia de tipos e introdução de campos e métodos em classes e interfaces de forma transversal.

2.3.3 Aplicações

Nesta seção são apresentados como exemplos de aplicação as implementações dos padrões de projeto Observador e *Wormhole*.

2.3.3.1 Padrão de Projetos Observador

O padrão de projeto *observador* tem por objetivo definir uma dependência de um para muitos entre objetos, de modo que, quando o estado de um objeto for alterado, todos os seus dependentes serão notificados e atualizados automaticamente (*Gamma et al.* 1995).

O padrão observador é utilizado para desacoplar o objeto alvo de observação de seus observadores, permitindo maior grau de reúso. Por exemplo, em bibliotecas de implementação de interfaces gráficas com o usuário, os objetos alvo de observação são componentes de interface e os observadores são responsáveis pela realização das operações de tratamento da manipulação da interface pelo usuário. Uma das aplicações do padrão observador é a separação entre dados e apresentação.

A implementação do padrão observador por meio de linguagens orientadas por objetos puras, como a sugerida por *Gamma et al.* (1995), apresenta problemas de modularização, devido a código de atualização dos observadores estar espalhado ao longo do código dos objetos observados. Nesta seção, apresenta-se uma implementação do padrão observador utilizando a linguagem AspectJ. A implementação por meio de aspectos para este padrão foi adaptada da solução proposta por *Hannemann and Kiczales* (2002).

O aspecto abstrato ObserverProtocol, definido na Listagem 2.3, é responsável por mapear objetos em seus observadores.

Este aspecto possui os seguintes elementos:

- as interfaces Subject e Observer representam, respectivamente, um objeto alvo de observação, doravante denominado *alvo*, e um observador; estas interfaces são utilizadas para consistências de tipos nos demais elementos do aspecto;

```
1  import java.util.*;
2
3  public abstract aspect ObserverProtocol {
4      public interface Subject {}
5      public interface Observer {}
6      private WeakHashMap perSubjectObservers;
7      private List getObservers(Subject s) {
8          if (perSubjectObservers == null)
9              perSubjectObservers = new WeakHashMap();
10         List observers = (List) perSubjectObservers.get(s);
11         if (observers == null) {
12             observers = new LinkedList();
13             perSubjectObservers.put(s, observers);
14         }
15         return observers;
16     }
17     public void addObserver(Subject s, Observer o) {
18         getObservers(s).add(o);
19         updateObserver(s,o);
20     }
21     public void removeObserver(Subject s, Observer o) {
22         getObservers(s).remove(o);
23     }
24     abstract public pointcut subjectChange(Subject s);
25     abstract public void updateObserver(Subject s, Observer o);
26     after(Subject s): subjectChange(s) {
27         Iterator it = getObservers(s).iterator();
28         while (it.hasNext())
29             updateObserver(s, (Observer) it.next());
30     }
31 }
```

Listagem 2.3: Protocolo Para o Padrão de Projetos Observador

- o mapeamento perSubjectObservers é responsável por armazenar as listas de observadores de cada alvo;
- os métodos addObserver e removeObserver são responsáveis por adicionar e remover um observador da lista de observadores de um alvo, respectivamente;
- o método getObserver é responsável por retornar a lista de observadores de um alvo, armazenada na tabela perSubjectObservers; é responsável também por criar uma nova lista vazia e associá-la a um alvo, caso este ainda não esteja cadastrado na tabela;
- o conjunto de junção abstrato subjectChange deve ser redefinido nos subaspectos de ObserverProtocol; nos subaspectos, ele define os métodos responsáveis por alterar o estado interno dos objetos observados (ver aspecto TemperatureObservation, definido a seguir);
- o método abstrato updateObserver deve ser redefinido nos subaspectos, de modo que defina as ações de notificação dos observadores quando o estado interno do sujeito for alterado;
- a regra after definida para o conjunto de junção subjectChange é responsável por obter a lista de observadores de um alvo e chamar o método de notificação para cada observador; este código é incluído após cada ponto de junção que represente uma mudança de estado do objeto alvo.

Para implementar o padrão observador utilizando aspectos, deve-se definir: (i) os objetos alvo de observação; (ii) os objetos observadores; (iii) as operações dos alvos que definem as alterações do estado interno; (iv) o algoritmo de atualização de cada observador; (v) os momentos em que a observação sobre um alvo inicia e termina.

Para definir que uma classe é observadora de outra, deve-se implementar um aspecto que estenda ObserverProtocol e que defina o conjunto de junção de modificação do estado interno e o método de notificação de observadores.

Considere, por exemplo, as classes Termometer e Temperature definidas nas Listagens 2.4 e 2.5, respectivamente.

O aspecto TemperatureObservation, mostrado na Listagem 2.6, estende o aspecto ObserverProtocol, definindo que Termometer implementa a interface Observer, Temperature implementa a interface Subject. Define-se também que o conjunto de junção subjectChange representa todas as execuções do método Temperature.setValue(), e que a notificação dos

```
1 public abstract class Termometer {
2     public abstract void printTemperature(double value);
3 }
```

Listagem 2.4: Classe Termometer

```
1  public class Temperature {
2      private double value;
3      public Temperature(double initialValue ) {
4          value = initialValue ;
5      }
6      public double getValue() {
7          return value;
8      }
9      public void setValue(double value) {
10         this.value = value;
11     }
12 }
```

Listagem 2.5: Classe Temperature

```
1  public aspect TemperatureObservation extends ObserverProtocol {
2
3      declare parents: Temperature implements Subject;
4      declare parents: Termometer implements Observer;
5
6      public pointcut subjectChange(Subject s)
7          : target(s) && execution(void Temperature.setValue(double));
8
9      public void updateObserver(Subject s, Observer o) {
10         Temperature temperature = (Temperature) s;
11         Termometer termometer = (Termometer) o;
12         termometer.printTemperature(temperature.getValue());
13     }
14
15 }
```

Listagem 2.6: Aspecto Para Observação de Temperatura

```
1  public class TermometerCelsius extends Termometer {
2      public void printTemperature(double value) {
3          System.out. println ("Celsius: " + value);
4      }
5  }
```

Listagem 2.7: Classe TermometerCelsius

observadores é feita por meio da chamada do método de impressão da temperatura, Termometer.printTemperature().

A seguir, definem-se duas subclasses de Termometer: TermometerCelsius (Listagem 2.7) e TermometerFahrenheit (Listagem 2.8).

Para associar uma instância de uma classe termômetro como observadora de uma instância de temperatura, utiliza-se o método addObserver do aspecto, como mostrado na Listagem 2.9.

O método aspectOf é utilizado para referenciar a instância única de um aspecto.

As principais vantagens da abordagem são:

```
1 public class TermometerFahrenheit extends Termometer {
2     public void printTemperature(double value) {
3         System.out. println ("Fahrenheit: " + value);
4     }
5 }
```

Listagem 2.8: Classe TermometerFahrenheit

```
1 TermometerCelsius term1 = new TermometerCelsius();
2 TermometerFahrenheit term2 = new TermometerFahrenheit();
3 TemperatureObservation.aspectOf().addObserver(t,term1);
4 TemperatureObservation.aspectOf().addObserver(t,term2);
```

Listagem 2.9: Exemplo do Uso do Protocolo de Observação

- a modularização completa do padrão: foram retirados das classes do objeto observado e do observador os códigos necessários para implementação do padrão observador;
- aumento do grau de reúso do padrão: pode-se definir de maneira não-intrusiva qualquer classe como observador ou alvo de observação, definindo-se somente um aspecto que implemente o respectivo protocolo de observação.

2.3.3.2 Padrão de Projetos *Wormhole*

Considere o conjunto de classes ilustrado na Figura 2.1, em que uma chamada típica de operação gera a execução de diversas operações secundárias, representadas na forma do grafo da Figura 2.2. A implementação por meio de aspectos para este padrão foi adaptada da solução apresentada em *Gradecki and Lesieck* (2003).

Se um objeto Worker precisar de informações a respeito do contexto do objeto Caller que iniciou a execução das operações, pode-se definir parâmetros nas chamadas de métodos dos objetos Operation para que este contexto seja propagado até os objetos Worker. Esta implementação gera intrusão, pois os objetos Operation podem não necessitar das informações de contexto. Neste caso, a existência desses parâmetros pode tornar complexas as interfaces das classes Operation, produzindo assim alto grau de acoplamento entre módulos. Outra solução possível é definir uma área de acesso global, onde o objeto Caller armazena seus dados para leitura pelos objetos Worker. Essa situação gera acoplamento de dado externo entre os objetos Caller e Worker, o que também desvaloriza a solução.

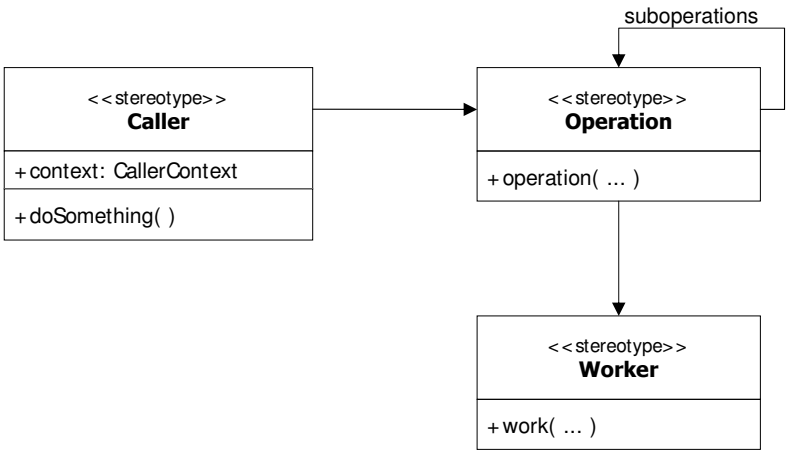

Figura 2.1: Representação em UML do padrão *wormhole*.

O padrão *wormhole* tem por objetivo definir de forma modular um canal direto de comunicação entre os objetos Caller e Worker, tornando disponíveis para os objetos Worker as informações de contexto que ele precisar. Esse padrão permite encapsular a comunicação entre esses objetos, propiciando uma implementação com grau de acoplamento menor que as implementações que passam o contexto como parâmetro ou o armazenam em uma área de acesso global.

A Figura 2.3 esquematiza a implementação da passagem de contexto por meio do padrão *wormhole*. Nesta implementação, a informação de contexto é transmitida diretamente do objeto Caller para os objetos Worker, sem a necessidade de alterar a interface dos objetos Operation.

O aspecto WormholePattern define o padrão *wormhole*, conforme mostrado na Listagem 2.10.

Neste aspecto, definem-se os seguintes conjuntos de junção:

- invocations: pontos de chamadas de métodos da classe Operation, ou suas subclasses, que ocorrerem em métodos cujo objeto receptor é da classe Caller;

- workPoints: pontos de chamadas do método doWork da classe Worker;

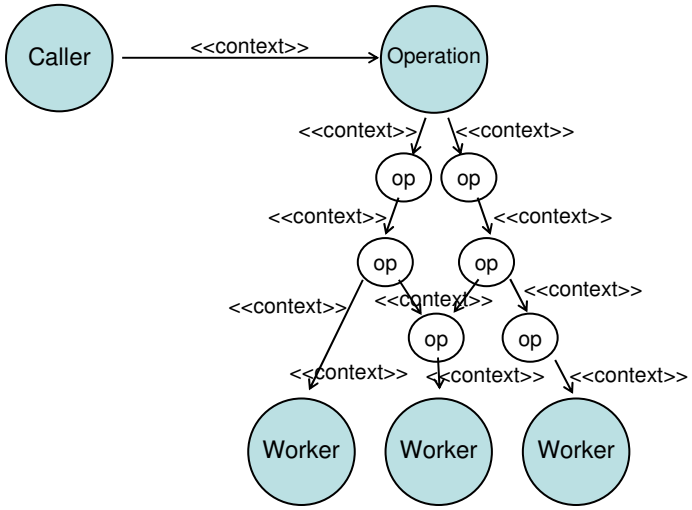

Figura 2.2: Implementação de transmissão de contexto via passagem de parâmetros – Extraída de *Kiczales* (2002).

- perCallerWork: pontos de chamadas do método Worker.doWork() ou suas subclasses, que ocorrerem no fluxo de execução de uma invocação de operação a partir de métodos da classe Caller.

A regra before aplicada ao conjunto de junção perCallerWork propaga para o objeto Worker o contexto do objeto Caller cujo fluxo de execução gerou a chamada ao método Worker.doWork().

A implementação por meio de aspectos permite obter a propagação direta de informações de contexto, sem aumentar a interface da classe Operation ou aumentar o acoplamento entre as classes Caller e Worker por meio de valores globais. Além disso, todo o acoplamento entre as classes Caller e Worker é controlado pelo aspecto WormholePattern, que funciona como mediador. Quaisquer alterações no contexto geram alterações somente neste aspecto, de modo que as classes participantes não precisam ser alteradas para implementar a transmissão do contexto.

É possível definir também um aspecto abstrato que represente o protocolo do padrão *wormhole* de maneira semelhante à feita nas implementações do padrão observador. Com isso, este padrão também pode possuir uma implementação genérica e reutilizável.

```
1  public aspect WormholePattern {
2
3      pointcut invocations( Caller  c):  this(c)  && call(* Operation+.*(..));
4
5      pointcut workPoints(Worker w): target(w) && call(void Worker.doWork());
6
7      pointcut perCallerWork( Caller  c,  Worker w): cflow( invocations(c))  && workPoints(w);
8
9      before ( Caller  c,  Worker w): perCallerWork(c,  w) {
10         w.propagateCallerContext(c.getContext());
11         System.out. println (thisJoinPoint.getClass ());
12     }
13
14 }
```

Listagem 2.10: Protocolo Para o Padrão *Wormhole* – Extraído de *Gradecki and Lesieck* (2003)

2.3.4 Avaliação da Técnica

O desenvolvimento de sistemas orientados por aspectos possui semelhanças com o desenvolvimento de sistemas orientados por objetos. A proposta metodológica consiste em definir e implementar os requisitos do sistema por meio de orientação por objetos e, em seguida, definir regras para o entrelaçamento das preocupações em diferentes módulos. Com isto, é possível definir componentes de software com baixo acoplamento e alta coesão.

No paradima imperativo, a programação orientada por aspectos não se trata de um substituto para a programação orientada por objetos. Os requisitos funcionais de um sistema continuarão a ser implementados por meio da POO. A orientação por aspectos simplesmente adiciona novos conceitos à POO, facilitando a implementação de preocupações ortogonais e retirando grande parte da atenção dada a tais preocupações nas fases iniciais de desenvolvimento. Com efeito, o desenvolvedor pode se concentrar na implementação dos módulos de requisitos funcionais do sistema, permitindo que a implementação da distribuição de preocupações ortogonais aos módulos seja feita separadamente.

Apesar de a proposta inicial de orientação por aspectos ter sido feita utilizando uma linguagem funcional, ainda há poucas implementações de linguagens funcionais orientadas por aspectos, de modo que este é um campo de trabalho ainda bastante inexplorado.

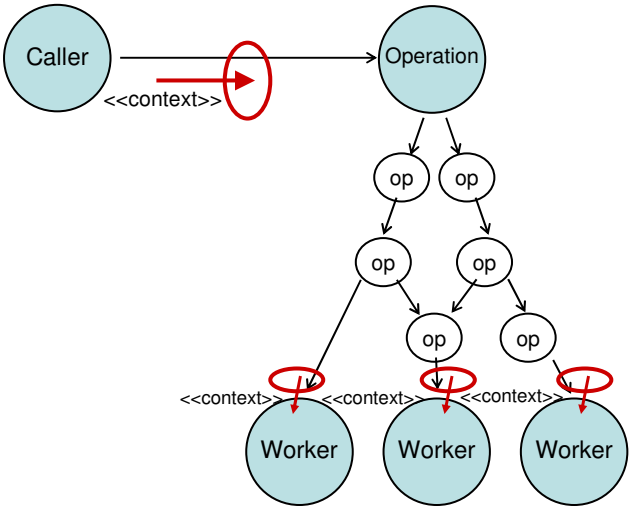

Figura 2.3: Representação gráfica do padrão *wormhole* – Extraída de *Kiczales* (2002).

O desenvolvimento orientado por aspectos permite definir claramente as responsabilidades dos módulos individuais, uma vez que cada módulo é responsável unicamente por seu requisito principal. Além disso, o nível de modularização do sistema é melhorado, com baixo acoplamento e alta coesão modular. Com efeito, ao retirar código intruso dos módulos, é possível diminuir a interface de cada módulo e, ao mesmo tempo, aumentar o seu nível de coesão, por tratar somente um requisito. As consequências diretas são a melhoria do processo de manutenção de sistemas e aumento no grau de reúso. Além disso, a evolução de sistemas é facilitada, visto que, se novos aspectos forem criados para implementar novas preocupações ortogonais, as classes do programa podem permanecer inalteradas. Da mesma forma, ao adicionar novas classes ao programa, os aspectos existentes também são transversais a essas classes.

Como principal ponto negativo, tem-se que a orientação por aspectos pode quebrar a encapsulação de módulos, ao permitir acesso a detalhes de sua implementação e a alteração da estrutura estática de tipos. Além disso, os mecanismos para definição de conjuntos de junção apresentam uma dificuldade inerente denominada fragilidade no sentido que alterações nas interfaces de módulos podem tornar obsoletas as expressões de definição dos pontos afetados, exigindo sua redefinição.

Outro ponto relevante de pesquisa é a definição formal completa das linguagens orientadas por aspectos. *Wand et al.* (2004) definiram a semântica formal de conjuntos de junção e adendos da transversalidade dinâmica de AspectJ utilizando a Semântica Denotacional. *Walker et al.* (2003) definem a Semântica Operacional Estruturada de uma linguagem conceitual orientada por aspectos, com o objetivo de oferecer um arcabouço para definição da semântica de linguagens orientadas por aspectos. O trabalho de *Lämmel* (1999) mostra a semântica formal de uma linguagem funcional orientada por aspectos e formaliza o mecanismo de costura de código realizado também por meio da Semântica Operacional Estruturada.

Aldrich (2004), *Clifton and Leavens* (2002), *Kiczales and Mezini* (2005), *Krishnamurthi et al.* (2004) discutem o impacto da utilização da orientação por aspectos no raciocínio modular. Segundo os autores, a leitura de um módulo em um sistema orientado por aspectos exige algum conhecimento global do sistema. No entanto, pode-se argumentar que, na presença de preocupações ortogonais, o acoplamento tende a ser muito alto e a coesão dos módulos, muito baixa, o que pode resultar na mesma necessidade de conhecimento global no sistema. A vantagem destacada do uso da orientação por aspectos neste ponto reside principalmente no fato de que essa necessidade se torna explícita e o raciocínio modular pode ser feito uma vez terminada essa análise global.

2.4 Conclusões

A modularidade é um elemento chave na implementação de sistemas de grande porte, por permitir que o desenvolvedor se concentre em pequenas partes do problema a cada momento. Recentemente, percebeu-se que certas preocupações em sistemas não eram adequadamente separáveis de outras, principalmente pela falta de construções linguísticas para lidar com partes do programa cuja codificação se apresenta entrelaçada. Essa dificuldade gera problemas a longo prazo, uma vez que, com o tempo, pode ser necessário que elementos entrelaçados evoluam independentemente.

A partir dessa constatação, definiram-se mecanismos teóricos de decomposição de sistemas em várias dimensões, nomenclatura oriunda da interpretação das etapas da evolução independente dos requisitos como pontos de um eixo em um espaço multidimensional.

As técnicas propostas para solução desse problema, a orientação por aspectos e a decomposição multidimensional, encontram-se em estado de pleno desenvolvimento, cujas pesquisas têm gerados resultados práticos de grande impacto na modularidade de sistemas.

Definições de semântica denotacional de linguagens de programação apresentam problemas de modularidade devidos a entrelaçamento de constituintes nas equações, conforme será mostrado nos Capítulo 3, de modo que a utilização dos mecanismos de decomposição multidimensional e orientação por aspectos podem trazer ganhos significativos na definição de módulos independentes e na escrita incremental de especificações.

O próximo capítulo aborda as principais técnicas para definição semântica, com ênfase dada especialmente às técnicas para modularizar especificações de semântica denotacional.

Capítulo 3

Modularidade da Semântica Denotacional

Este capítulo apresenta uma descrição geral das técnicas de especificação da semântica de linguagens de programação, com ênfase na semântica denotacional e nas metodologias correntes para modularização de especificações nesse arcabouço.

Organização do Capítulo. Na Seção 3.1, são relatadas as principais metodologias de definição de semântica de linguagens de programação. A modularidade das definições de Semântica Denotacional e suas técnicas de modularização mais importantes são mostradas na Seção 3.2. Finalmente, a Seção 3.3 apresenta as conclusões do capítulo.

3.1 Semântica de Linguagens de Programação

Definições formais da semântica de linguagens de programação são de grande relevância, uma vez que possibilitam a definição não-ambígua dos constituintes de uma linguagem, o que não pode ser garantido ao se definir a semântica de maneira informal por meio da linguagem natural. A consequência direta de seu uso é a viabilização de uma série de atividades de importâncias teórica e prática, dentre as quais destacam-se: projeto e prototipação de novas linguagens, geração automática de interpretadores e compiladores, verificação de propriedades e padronização da linguagem (*Mosses* 2001, *Schmidt* 1996, *Zhang and Xu* 2004). *Tennent* (1976) argumenta ainda que, mesmo se uma definição

formal não for acessível ao programador médio, ela ainda será útil ao prover a base para uma descrição informal mais precisa.

Zhang and Xu (2004) mostram sete áreas de aplicação para semântica formal de linguagens de programação:

1. a semântica é o produto do processo de elaboração de uma linguagem, podendo ser usada para comunicar as decisões tomadas pelos projetistas;
2. durante a implementação da linguagem, a especificação da semântica permite garantir o comportamento correto da implementação;
3. a padronização da linguagem pode ser obtida publicando-se semântica não-ambígua;
4. a compreensão dos programadores sobre a linguagem requer o aprendizado do seu comportamento, isto é, da sua semântica;
5. a semântica auxilia o programador a raciocinar a respeito de seu programa, permitindo por exemplo verificar se este faz o que deveria fazer;
6. a semântica permite que os teóricos da área obtenham novos *insights* sobre conceitos de programação, abrindo assim novas áreas de pesquisa;
7. descrições semânticas podem ser utilizadas na geração automática de interpretadores e compiladores.

Segundo *Boute* (2005) um formalismo é composto por uma notação e um conjunto de regras formais de manipulação, de modo que o seu mérito é medido não somente pelo seu escopo e expressividade, mas principalmente pelo grau em que auxilia no raciocínio e na prova de propriedades. Com isso, é essencial que a obtenção de um alto nível de abstração seja um dos objetivos principais do projeto de qualquer mecanismo de definição formal.

Para avaliar as técnicas de especificação semântica, *Gayo* (2002), *Mosses* (1988), *Moura* (1996) sugerem os seguintes critérios:

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.

Os arcabouços de definição semântica são comumente classificados em três categorias: (i) *axiomática*, que descreve propriedades de programas por meio de asserções; (ii) *operacional*, que especifica as diversas construções de uma linguagem por meio de uma sequência de passos para se obter um resultado em alguma máquina abstrata hipotética; (iii) *denotacional*, que define a semântica de uma linguagem por meio de mapeamentos de suas construções em objetos matemáticos denominados denotações.

3.1.1 Semântica Axiomática

A semântica axiomática definida por *Hoare* (1969) é baseada na lógica de predicados e descreve a semântica de uma linguagem por meio das propriedades de suas construções. O propósito deste modelo é oferecer uma base lógica para a prova de propriedades de programas. Para isso, a semântica de construções é dada por meio de asserções sobre os valores de variáveis antes e depois das suas execuções.

Axioma de atribuição: $(P[x \mapsto a])\{x \leftarrow a\}P$

Regras de consequência:

$$\frac{P\{Q\}R \qquad R \Rightarrow S}{P\{Q\}S} \qquad\qquad \frac{P\{Q\}R \qquad S \Rightarrow P}{S\{Q\}R}$$

Regra de composição:

$$\frac{P\{Q_1\}R_1 \qquad R_1\{Q_2\}R}{P\{Q_1; Q_2\}R}$$

Regra de condicional:

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

Regra de repetição:

$$\frac{(B \wedge P)\{Q\}P}{P\{while\ B\ do\ Q\}(\neg B \wedge P)}$$

Figura 3.1: Exemplo de semântica axiomática de uma linguagem contendo atribuições, condicionais e repetição. As regras são baseadas em *Hoare* (1969), *Zhang and Xu* (2004).

Uma especificação semântica axiomática é um conjunto de asserções da forma $P\{Q\}R$, onde Q é uma construção da linguagem, P é uma pré-condição à execução de Q, e R é a descrição do resultado da execução de Q. Também é possível definir regras de inferência da forma:

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

em que $R_1, R_2, \cdots, R_n$ são as premissas, e R é a conclusão. Por exemplo, a Figura 3.1 mostra as regras para os comandos de uma linguagem simples que contém atribuições, condicionais e repetição.

A semântica axiomática é uma forma bastante simples de se definir propriedades das construções de uma linguagem, sendo portanto uma boa técnica para se provar propriedades de programas. Entretanto, conforme destacado por *Mosses* (2001), uma dificuldade do modelo está no fato de que expressões da linguagem não podem ter efeitos colaterais nem gerar erros ao serem avaliadas, uma vez que a semântica de comandos de atribuição envolve a substituição nas cláusulas lógicas de variáveis pelos valores das expressões no lado direito das atribuições. Além disso, *Zhang and Xu* (2004) destacam que esta técnica não

é adequada para geração automática de compiladores.

3.1.2 Semântica Operacional

A semântica operacional define a semântica das construções de uma linguagem por meio dos passos para a sua execução em uma máquina abstrata hipotética. O foco dessa técnica está, portanto, na forma como um resultado é obtido e não no seu significado (*Zhang and Xu* 2004). Segundo *Mosses* (2001), a computação geralmente é dada por uma sequência possivelmente infinita de passos entre estados, de modo que a semântica operacional tem por objetivo modelar as computações que um programa realiza para chegar a um resultado.

Existem diversas técnicas baseadas em semântica operacional, dentre as quais destacam-se: a semântica operacional estruturada (*Plotkin* 1981), as máquinas de estado abstratas (*Gurevich* 1994, 1995) e a semântica operacional modular *Mosses* (2004).

3.1.2.1 Semântica Operacional Estruturada

Na semântica operacional estruturada (*Plotkin* 1981), uma computação é modelada como uma sequência de transições em um sistema de transições rotuladas, isto é, uma estrutura $\langle \Gamma, A, \rightarrow \rangle$, onde Γ é um conjunto de configurações – os estados do sistema –, A é um conjunto de rótulos, e $\rightarrow \subseteq \Gamma \times A \times \Gamma$ é uma relação de transição. A definição das transições na máquina são dadas em geral por meio de regras de inferência.

Durante uma computação em semântica operacional estruturada, a árvore de sintaxe abstrata do programa é atualizada, substituindo-se partes da árvore pelos valores correspondentes gerados pela computação. As informações de contexto necessárias para cálculo dos resultados, tais como estados e ambientes, pertencem em geral aos estados da computação. A Figura 3.2, inspirada em *Plotkin* (1981), mostra um exemplo de definição da semântica operacional estruturada de uma linguagem simples contendo constantes, variáveis, atribuições e condicionais.

A semântica operacional estruturada é uma técnica bastante poderosa de definição semântica, permitindo a especificação de praticamente todos os recursos presentes em linguagens de programação. Além disso, a geração de interpretadores a partir da definição de uma linguagem utilizando semântica operacional estruturada é bastante simples.

Atribuição:

$$\frac{\langle e, \sigma\rangle \rightarrow^* \langle m, \sigma'\rangle}{\langle v \leftarrow e, \sigma\rangle \rightarrow \sigma'[m/v]}$$

Composição:

$$\frac{\langle c_0, \sigma\rangle \rightarrow \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \rightarrow \langle c_0'; c_1, \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma\rangle \rightarrow \langle c_1, \sigma'\rangle}$$

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 3.2: Exemplo de semântica operacional estruturada de uma linguagem simples contendo atribuições e condicionais. As equações foram adaptadas de *Plotkin* (1981). Neste sistema, m representa um valor, σ, σ' representam *stores*, c, c_0, c_1, c_2 representam comandos, e representa uma expressão, e os estados são elementos de $\Gamma = \{\langle e, \sigma\rangle\} \cup \{\langle c, \sigma\rangle\} \cup \{\sigma\}$.

Entretanto, conforme apontado por *Gayo* (2002), além de ser difícil escrever especificações modulares e reutilizáveis, algumas definições formais são muito longas, o que dificulta a verificação de propriedades de programas.

3.1.2.2 Máquinas de Estado Abstratas

O modelo de Máquinas de Estado Abstratas[1] (*Gurevich* 1994, 1995) tem por objetivo favorecer a definição de semântica formal operacional com nível de abstração mais alto que o obtido com técnicas anteriores, ao mesmo tempo em que possui base matemática precisa que permite a demonstração de propriedades da linguagem. Nesta técnica, os estados são definidos por meio de funções e relações sobre um conjunto denominado superuniverso; as transições de estado são feitas por meio da redefinição das interpretações de funções e relações. Com efeito, uma regra de transição é composta por regras de atualização cuja função é atualizar funções em pontos definidos. O exemplo da Figura 3.3 mostra uma definição da semântica operacional de uma linguagem simples composta por valores

[1] Uma introdução a máquinas de estado abstratas é dada por *Tirelo et al.* (1999).

Funções dinâmicas:

$$val : Expr \rightarrow Int$$

$$loc : Var \rightarrow Value$$

Regras de transição:

if *task* **is** *num* **then**
    val(task) := num

if *task* **is** $e_1 + e_2$ **then**
    val(task) := val(e_1) + *val*(e_2)

if *task* **is** *var* **then**
    val(task) := loc(var)

if *task* **is** $x := e$ **then**
    loc(x) := val(e)

Figura 3.3: Exemplo de semântica operacional de uma linguagem simples contendo expressões, variáveis e atribuições, por meio de máquinas de estado abstratas. As equações foram adaptadas de *Gayo* (2002).

inteiros, operações aritméticas, variáveis e atribuições. Esse exemplo, foi extraído de *Gayo* (2002), inspirado em *Börger and Schulte* (1998).

Ao se definir uma especificação de máquinas de estado abstratas, parte-se de um modelo de alto nível denominado *ground model* (*Börger and Schulte* 1998, *Gurevich* 1995), que é reificado até se obter uma versão executável da especificação. Esse método permite a definição de especificações modulares e executáveis. Além disso, o modelo é relativamente simples, sendo as definições bastante parecidas com programas em uma linguagem de programação imperativa de alto nível. A vantagem principal deste método, no entanto, é a possibilidade de se realizar o raciocínio equacional que leva a prova de propriedades da especificação.

3.1.2.3 Semântica Operacional Modular

Mosses (2004) propôs um modelo inspirado em semântica monádica para a definição da semântica operacional estruturada modular (MSOS) de linguagens de programação. Em sua abordagem, à medida que novas construções são adicionadas à especificação, o contexto pode evoluir sem exigir que equações previamente escritas sejam redefinidas. Informação de contexto é transmitida por meio de rótulos nas regras de transição, e modularidade e extensibilidade são obtidas ao se deixar que as transições sejam independentes da estrutura dos rótulos. O resultado é um conjunto de regras de transição abstratas que não fazem referência explícita a informações de contexto. Além disso, o modelo preserva a propriedade da semântica operacional estruturada em permitir definição separada da interação entre construções.

Com o objetivo de simplificar a escrita de definições de semântica operacional estruturada, associa-se às regras de transição uma mônada (*Moggi* 1989, 1991, *Wadler* 1990, 1992). Esta mônada possibilita que *stores*, *environments* e continuações sejam passadas implicitamente nas regras, sem a necessidade da menção implícita feita na semântica operacional estruturada. Resultado semelhante foi obtido por *Turi and Plotkin* (1997), em que as regras de transição são modeladas como transformações da teoria das categorias (*Fokkinga* 1992, *Fokkinga and Meertens* 1994, *Menezes and Haeusler* 2001, *Shutt* 2003).

No entanto, a definição de certas interações podem exigir a listagem de todos os possíveis casos, como na definição de Java fornecida por *Cenciarelli et al.* (1999); mesmo sem a definição de repetição e sequenciadores, há trinta e três regras de transição para a definição de chamada e retorno de função, tratamento de exceções e suas interações[2].

3.1.3 Semântica Denotacional

A semântica denotacional (*Gordon* 1979, *Scott and Strachey* 1971, *Stoy* 1981, *Tennent* 1976) define a semântica de uma linguagem mapeando suas construções em objetos matemáticos denominados denotações. Geralmente, denotações são funções contínuas de ordem mais alta dos domínios de *Scott* (1976), cujos argumentos são informações do contexto e o resultado é a influência da construção no comportamento global do programa. Uma diferença

[2]Na verdade, as regras apenas definem o comportamento de comandos de retorno em blocos *try*, sem especificar o seu comportamento dentro de blocos *catch*; assim, pelas regras apresentadas, se um retorno for executado dentro de um bloco *catch*, o bloco *finally* correspondente não será executado, diferentemente da especificação padrão da linguagem.

básica entre esse modelo e a semântica operacional é que, enquanto a semântica operacional define *como o resultado de um programa é obtido*, a semântica denotacional define *o efeito de cada construção no resultado final do programa*.

Definições de semântica denotacional podem ser feitas de duas maneiras: utilizando-se *semântica direta*, em que o sequenciamento é feito por composição de funções; ou por meio da *semântica de continuação* (*Strachey and Wadsworth* 1974), em que o sequenciamento é feito por meio de funções de continuação. Enquanto na semântica direta, a semântica de um programa é dada pela composição das funções semânticas, na semântica de continuação cada construção semântica propaga o efeito de sua execução por meio de suas continuações. A semântica de continuação torna possível definir a semântica de sequenciadores, pois permite que a construção continue sua execução em outro ponto do programa, ignorando a continuação normal recebida como argumento.

Uma definição de semântica denotacional é constituída por: sintaxe abstrata da linguagem, domínios sintáticos e semânticos, funções semânticas e equações semânticas. Tipicamente, as funções semânticas possuem como parâmetros as seguintes informações de contexto: (i) *environments*, ou ambientes, que mapeiam identificadores em localizações na memória; (ii) *stores*, que mapeiam localizações na memória em valores; (iii) *continuações*, que representam o passo seguinte à execução de cada construção.

A Figura 3.4 mostra um exemplo de definição da semântica denotacional de uma linguagem simples contendo somente constantes, variáveis, expressões aritméticas, atribuições e condicionais. Neste exemplo, definem-se os domínios semânticos, as funções semânticas para expressões, comandos e programas, e as equações semânticas para as construções da linguagem por meio da semântica direta.

A semântica denotacional é um mecanismo bastante poderoso e preciso para definição de semântica de linguagens de programação. Sua base matemática permite a prova de propriedades da definição e inclui elementos importantes tais como não-terminação e condições de erro. Entretanto, segundo *Gayo* (2002), *Mosses* (2004), a definição denotacional de não-determinismo e concorrência requer o uso de domínios potência, o que exige uma quantidade significativa de notação extra. Além disso, é possível gerar protótipos para linguagens a partir de definições semânticas denotacionais. No entanto, o grande problema do modelo de semântica denotacional reside na falta de modularidade e dificuldade em reúso das definições, conforme discutido na Seção 3.2.

Domínios Semânticos:

$$\mathbf{Num} = \{0, 1, 2, \cdots\}$$
$$\mathbf{State} = [\mathbf{Ide} \mapsto \mathbf{Num}]$$

$$n \in \mathbf{Num}, s \in \mathbf{State}$$

Funções Semânticas:

$$\mathcal{E} : \mathbf{Exp} \mapsto \mathbf{State} \mapsto \mathbf{Num}$$
$$\mathcal{C} : \mathbf{Com} \mapsto \mathbf{State} \mapsto \mathbf{State}$$
$$\mathcal{P} : \mathbf{Prog} \mapsto \mathbf{Num} \mapsto \mathbf{Num}$$

Equações Semânticas:

$$\mathcal{E}[\![0]\!]s = 0$$
$$\mathcal{E}[\![I]\!]s = s\ I$$
$$\mathcal{E}[\![\mathbf{suc}\ E]\!]s = \mathcal{E}[\![E]\!]s + 1$$
$$\mathcal{C}[\![I\ :=\ E]\!]s = s[\mathcal{E}[\![E]\!]s/I]$$
$$\mathcal{C}[\![C_1; C_2]\!]s = \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!]s)$$

Figura 3.4: Exemplo de semântica denotacional de uma linguagem simples contendo expressões, variáveis e atribuições e repetições – extraído de *Bigonha* (2003).

3.2 Modularidade em Semântica Denotacional

Modularizar uma especificação semântica significa ser capaz de dividi-la em unidades coesas e independentes, isto é, com baixa conectividade e baixo acoplamento. Segundo *Mosses* (1988), a modularidade de uma especificação pode ser avaliada por meio de três critérios: a *granularidade* da definição, que indica a capacidade de um sistema em ser dividido em partes independentes; a *independência* entre seus módulos, isto é, a conectividade do sistema e o acoplamento das conexões; e a *padronização* de seus constituintes.

Ainda de acordo com *Mosses* (1988), a modularidade de uma especificação semântica é muito importante, uma vez que definições com alto grau de modularidade tendem a possuir as seguintes características, desejáveis em qualquer projeto de grande porte:

1. *extensibilidade* (*Meyer* 1997), isto é, pequenas modificações na linguagem devem exigir pequenas alterações na sua especificação semântica;

2. *reusabilidade*, isto é, ao se escrever a especificação de uma linguagem, deve ser possível

utilizar prontamente módulos já escritos em definições anteriores, em especial quando as linguagens apresentam muitas semelhanças;

3. *legibilidade*, no sentido que deve ser possível pela definição formal da linguagem compreender e provar suas propriedades mais importantes.

Como observado na literatura (*Liang et al.* 1995, *Mosses* 1988, 2001, *Zhang and Xu* 2004), a semântica denotacional apresenta problemas sérios de modularidade. O mais importante desses problemas está relacionado ao fato de as funções semânticas dependerem totalmente das definições dos domínios semânticos, o que pode acarretar modificações ao longo de toda a definição quando a inclusão de novos recursos na linguagem exigirem a alteração desses domínios. Um exemplo significativo desse problema está relacionado à dificuldade em se passar de semântica direta para semântica de continuação, pois isto implica na alteração de *todas* as equações semânticas da especificação.

Além disso, o λ-cálculo, notação padrão utilizada na semântica denotacional, apesar de permitir isolar detalhes da teoria de domínios, não é suficiente para abstrair *stores*, *environments* e continuações. Por exemplo, a equação semântica de um comando condicional, apesar de não precisar fazer referência direta ao *environment* e à continuação correntes, deve incluí-los em sua definição, como mostrado na equação a seguir[3]:

$$\mathcal{C}[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2]\!] = \lambda rc.\mathcal{R}[\![E]\!]r; \mathit{cond}(\mathcal{C}[\![C_1]\!]rc, \mathcal{C}[\![C_2]\!]rc)$$

Como solução aos problemas apresentados, várias abordagens já foram propostas, dentre as quais destacam-se a semântica de ações (Seção 3.2.1), a semântica de mônadas (Seção 3.2.2), e a semântica de ações com mônadas (Seção 3.2.3). Mais recentemente, foram propostas abordagens para a separação de preocupações em especificações semânticas, com destaque para as gramáticas de atributos de primeira classe (Seção 3.2.4) e a abordagem construtiva para semântica (Seção 3.2.5).

3.2.1 Semântica de Ações

Semântica de Ações (*Mosses* 1977, 1993, *Mosses and Watt* 1987, *Moura* 1996) é uma proposta híbrida de semântica denotacional e operacional, que tem por objetivo permitir

[3]Neste exemplo, r representa o *environment* e c, a continuação; a notação utilizada é baseada naquela utilizada por *Gordon* (1979).

Semântica de Expressões:

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

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

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

Figura 3.5: Exemplo de semântica de ações para as expressões de uma linguagem simples contendo expressões, variáveis, atribuições e sequenciamento – extraído de *Gayo* (2002).

a criação de definições de semântica denotacional legíveis e modulares. Em semântica de ações, define-se uma linguagem por meio do mapeamento de suas construções em ações, que são por sua vez definidas por meio de semântica operacional estruturada. Segundo *Zhang and Xu* (2004), este elemento é chave para a semântica de ações, pois sua base teórica é mais acessível que a teoria de funções contínuas de ordem mais alta.

O elemento mais importante da semântica de ações é a *ação*, que consiste em uma entidade que pode ser executada, recebendo e produzindo dados e/ou alterando *stores*. Associados às ações, definem-se *combinadores de ações*, cujo objetivo é definir ações compostas. A *execução* de uma ação consiste em uma sequência de passos atômicos executados por um *agente*, que pode ser visto como uma máquina abstrata da semântica operacional.

Um dos pontos fortes do modelo é a legibilidade das definições semânticas, devido principalmente à sua "aparência completamente verbal"(*Mosses* 1988) que é gerada pela escolha por exemplo de verbos para nomear ações e substantivos para dados. Este fator pode ser claramente observado na definição das linguagens Pascal (*Mosses and Watt* 1993), JOOS[4] (*Watt* 1997) e Standard ML (*Watt* 1988). As Figuras 3.5 e 3.6 contêm um exemplo extraído de *Gayo* (2002), em que é dada a semântica de ações de uma pequena linguagem contendo números, expressões aritméticas, variáveis, atribuições e sequenciamento.

[4]JOOS é um subconjunto da linguagem Java contendo classes, herança, chamada dinâmica de métodos e construtores.

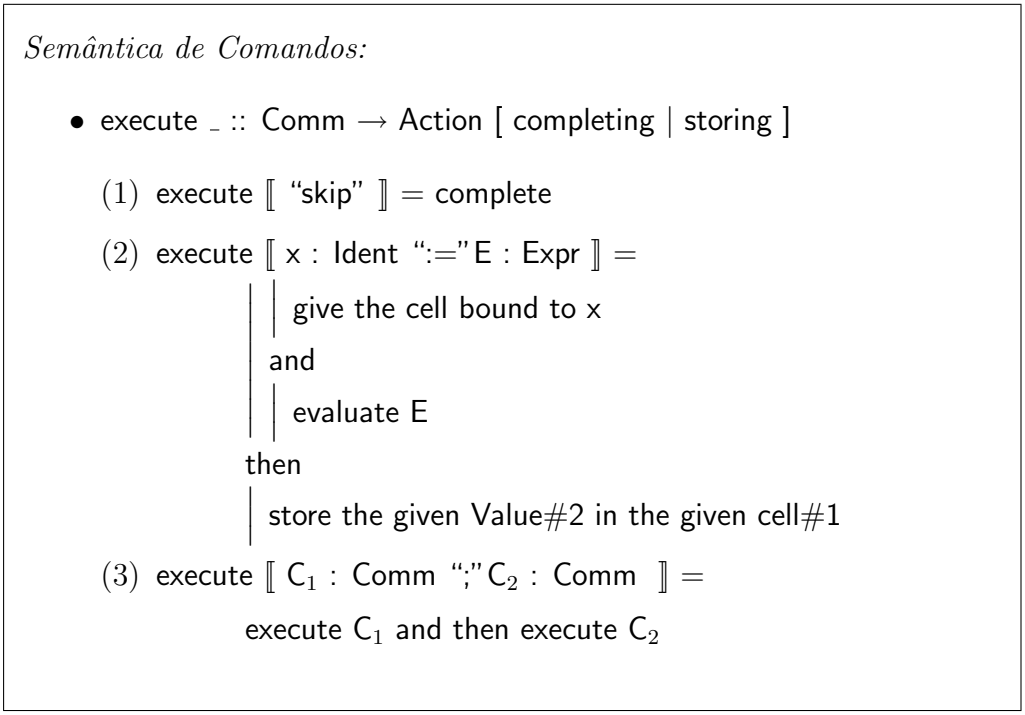

Figura 3.6: Exemplo de semântica de ações para os comandos de uma linguagem simples contendo expressões, variáveis, atribuições e sequenciamento – extraído de *Gayo* (2002).

Outro ponto forte deste modelo é a possibilidade de se reutilizar partes de definições anteriores, como pode ser visto por exemplo na definição da linguagem Pascal (*Mosses and Watt* 1993) em que se incluem elementos definidos por *Mosses* (1992). Um dos elementos que melhoram o grau de reúso obtido é o fato da notação de ações (*Mosses* 2000) ser padronizada.

Quanto à sua expressividade, segundo *Mosses* (2001), semântica de ações permite o suporte direto a fluxo de controle e de dados, amarrações de identificadores, efeitos colaterais, abstração de comandos e expressões, abstração de tipos, comunicação assíncrona de processos. Além disso, segundo *Moura* (1996), a modelagem de concorrência pode ser feita por meio da definição de múltiplos agentes.

No entanto, este modelo apresenta limitações no que diz respeito à modularidade. A principal dificuldade, apontada por *Wansbrough and Hamer* (1997), *Zhang and Xu* (2004), está no fato de que o modelo é incompleto, no sentido que os únicos conceitos de linguagens de programação que podem ser representados diretamente são aqueles definidos para compôr o núcleo do sistema, e a inclusão de novos elementos neste núcleo é uma tarefa bastante complexa. Esta dificuldade se deve principalmente ao fato de que, apesar de haver modu-

laridade no nível das equações semânticas, as ações são definidas de forma monolítica e não há uma técnica padronizada para a inclusão de novas definições. Um exemplo de recurso que não pode ser incluído diretamente no modelo é continuação de primeira classe.

Outra dificuldade do modelo é que, apesar de conseguir abstrair elementos de contexto tais como *stores* e *environments*, as continuações não são completamente abstraídas, como pode ser visto na definição da semântica da linguagem Pascal (*Mosses and Watt* 1993). Nessa definição, a presença do comando *goto* interfere nas equações semânticas de comandos, pois, em vez da ação execute comumente utilizada, é necessário o uso da ação jumpily execute, que tem por objetivo coletar os rótulos da definição e ativar a função execute. A Figura 3.7 mostra as definições do comando condicional *if-then-else* por meio de semântica de ações, na ausência e na presença do comando *goto*. Estas definições foram extraídas respectivamente das definições das linguagens JOOS (*Watt* 1997) e Pascal (*Mosses and Watt* 1993). Apesar de não fazer muita diferença na legibilidade da definição, percebe-se que existe uma limitação em relação ao reúso, uma vez que, apesar de tratarem a mesma construção, as duas definições são distintas em razão da existência de comando *goto* em uma das linguagens definidas.

3.2.2 Semântica de Mônadas

A semântica de mônadas teve início com os trabalhos de *Moggi* (1989, 1991), nos quais durante a interpretação de uma linguagem de programação em uma categoria $\mathcal{C}$ (ver Figura 3.8), distinguem-se dois objetos: A, representando o domínio dos valores, e $T\ A$, que representa o domínio das computações de valores. Neste contexto, a denotação de programas é um elemento de $T\ A$. Assim, uma vez identificado o tipo A dos valores de uma computação, o objeto de computação é obtido aplicando-se o transformador T a A. O construtor T é denominado *noção de computação*, uma vez que abstrai os tipos de valores que a computação pode produzir. A Figura 3.9 mostra alguns exemplos de noções de computação comuns em definições de semântica denotacional, extraídas de *Moggi* (1991). Intuitivamente, uma mônada para *Moggi* (1989, 1991) consiste em um trio (T, η, μ), tal que T é uma noção de computação, $\eta_A : A \rightarrow T\ A$ é um operador de inclusão de valores em computações, e $\mu_A : T^2\ A \rightarrow T\ A$ é um operador de composição de computações.

O ponto forte deste modelo consiste no fato de que uma vez que uma noção de computação T pode incorporar estados, continuações e erros, pode-se aumentar o grau de modularidade por meio da abstração destes elementos das assinaturas das funções semânticas.

(a) *Semântica do comando* if-then-else *na ausência do comando* goto*:*

execute ⟦ "if" E : Expr "then" S_1 "else" S_2 ⟧ =
  evaluate E then
    check (the given value is true) then execute S_1
  and
    check (the given value is false) then execute S_2

(b) *Semântica do comando* if-then-else *na presença do comando* goto*:*

execute ⟦ "if" E : Expr "then" S_1 "else" S_2 ⟧ =
  evaluate E then
    check (the given value is true) then jumpily execute S_1
  and
    check (the given value is false) then jumpily execute S_2

Figura 3.7: Definição em semântica de ações do comando condicional *if-then-else*: (a) na ausência de comando *goto* – extraído de *Watt* (1997); (b) na presença do comando *goto* – extraído de *Mosses and Watt* (1993).

A partir dos trabalhos de Moggi, *Wadler* (1990) propôs uma nova técnica para estruturação de programas em linguagens funcionais baseado no conceito de mônadas. Os resultados mais importantes da aplicação de mônadas nesse trabalho foi uma forma de encapsular conceitos impuros em linguagens funcionais – em especial, entrada/saída e estados –, além de permitir a unificação de propostas anteriores de incorporação de vários recursos em linguagens funcionais, dentre os quais se destacam *estados*, *exceções* e *não-determinismo*. A incorporação de forma transparente de entrada e saída e recursos inerentemente imperativos em linguagens funcionais é mostrada por *Jones and Wadler* (1993), *Wadler* (1997), com exemplos na linguagem Haskell.

Em outro trabalho, *Wadler* (1992) estende as propostas de Moggi, mostrando como estruturar em linguagem funcional um interpretador para uma linguagem de programação simples, utilizando uma mônada para representar a computação de um valor. As principais contribuições desta proposta são uma nomenclatura simples para representação de mônadas (ver Figura 3.10) e uma técnica incremental de escrita de um interpretador. Nesta

Uma *categoria* é uma sêxtupla $\mathcal{C} = \langle \mathsf{Obj}_\mathcal{C}, \mathsf{Mor}_\mathcal{C}, \partial_0, \partial_1, \iota, \circ \rangle$, onde:

- $\mathsf{Obj}_\mathcal{C}$ é uma coleção de elementos denominados *objetos*;
- $\mathsf{Mor}_\mathcal{C}$ é um conjunto de *morfismos* entre objetos de $\mathsf{Obj}_\mathcal{C}$;
- ∂_0 e ∂_1 são operações sobre morfismos denominadas *domínio* e *contra-domínio*, respectivamente; quando $\partial_0(f) = A$ e $\partial_1(f) = B$, então escreve-se $f : A \to B$;
- $\circ : \mathsf{Mor}_\mathcal{C} \times \mathsf{Mor}_\mathcal{C} \to \mathsf{Mor}_\mathcal{C}$ é uma operação associativa de *composição de morfismos*, de modo que a cada par de morfismos $\langle f : A \to B, g : B \to C \rangle$, existe um morfismo $g \circ f : A \to C$;
- ι é uma operação denominada *identidade*, tal que para cada objeto $X \in \mathsf{Obj}_\mathcal{C}$, existe um morfismo $\iota_X : X \to X \in \mathsf{Mor}_\mathcal{C}$, que satisfaz a propriedade:

$$\forall f : A \to B \in \mathsf{Mor}_\mathcal{C} : f \circ \iota_A = \iota_B \circ f = f$$

Figura 3.8: Definição de uma categoria, extraída de *Menezes and Haeusler* (2001). Para referências adicionais a respeito da teoria de categorias, recomendam-se: *Fokkinga* (1992), *Fokkinga and Meertens* (1994), *Shutt* (2003).

técnica, novos recursos são adicionados à linguagem alterando-se a mônada que representa a computação, sendo necessário alterar um único ponto do programa para incorporar estados, ambientes e continuações. As Figuras 3.11 e 3.12 contêm um extrato do interpretador definido por Wadler e a incorporação de alguns recursos ao interpretador. Em *Wadler* (1995), estas ideias são estendidas, mostrando-se como incorporar de forma transparente *parsers* e arranjos em linguagens funcionais.

Os trabalhos de *Jones and Duponcheel* (1993), *King and Wadler* (1993), *Steele Jr.* (1994) foram baseados na percepção de que apesar de mônadas serem um recurso de abstração interessante na definição da semântica de linguagens, havia uma dificuldade em se utilizar mais de uma mônada em uma mesma definição. Por exemplo, para se compôr as mônadas de estado e continuação era necessária a escrita de uma nova mônada, totalmente independente das já existentes, diminuindo significativamente o potencial de reúso do modelo.

Exemplos de noções de computação:

- *não-terminação:* $T\ A = A_\perp = A + \{\perp\}$, onde $\perp$ é a computação divergente;
- *não-determinismo:* $T\ A = \mathcal{P}(A)$;
- *efeitos colaterais:* $T\ A = S \to (A \times S)$;
- *exceções:* $T\ A = A + E$, onde E é o conjunto das exceções;
- *continuações:* $T\ A = (A \to R) \to R$, onde R é o conjunto das respostas;
- *entrada interativa:* $T\ A = \mu\,(\lambda\gamma.A + (U \to \gamma))$, onde μ é o operador de ponto fixo;
- *saída interativa:* $T\ A = \mu\,(\lambda\gamma.A + (U \times \gamma))$.

Figura 3.9: Noções de computação definidas por *Moggi* (1991). Variando-se a noção de computação, isto é, o construtor de tipos T, pode-se alterar os domínios das denotações de cada construção linguística.

Este problema foi resolvido inicialmente por *Espinosa* (1995) na técnica denominada *Semantic Lego*. Nesse trabalho, Espinosa propõe *transformadores de mônadas*, e oferece um mecanismo incremental de definição da mônada subjacente à definição semântica. Um transformador de mônadas é um par (t, lift), tal que t é um construtor de tipos e $\mathit{lift} : m\ a \to t\ m\ a$ é uma operação que transforma uma mônada m em uma nova mônada $t\ m$. A proposta original foi estendida por *Liang* (1997), *Liang et al.* (1995), que formalizaram o modelo e propuseram um arcabouço completo de escrita de definição semântica executável para uso na linguagem Haskell.

Os principais transformadores propostos por *Liang et al.* (1995) são os seguintes:

- transformador de estados:

  type $\mathit{StateT}\ s\ m\ a = s \to m\ (s, a)$

- transformador de erros:

  type $\mathit{Err}\ a = \mathit{Ok}\ a \mid \mathit{Err}\ \mathit{String}$

Uma mônada é uma tripla $(M, return, \gg\!=)$, em que M é um construtor de tipos, e $return$ e $\gg\!=$ são funções polimórficas com assinaturas:

$$\begin{aligned} return &:: a \to M\ a \\ \gg\!= &:: M\ a \to (a \to M\ b) \to M\ b \end{aligned}$$

e que satisfazem as seguintes propriedades:

$$\begin{aligned} &\textit{Left unit:} && (return\ a) \gg\!= k = k\ a \\ &\textit{Right unit:} && m \gg\!= return = m \\ &\textit{Associatividade:} && m \gg\!= (\lambda a.(k\ a) \gg\!= (\lambda b.h\ b)) \\ & && \quad = (m \gg\!= (\lambda a.(k\ a)) \gg\!= (\lambda b.h\ b) \end{aligned}$$

Os constituintes de uma mônada neste modelo são mapeados diretamente no modelo proposto por Moggi, de modo que M corresponde à noção de computação T, $return$ corresponde ao transformador η e $\gg\!=$ corresponde a μ.

Figura 3.10: Definição de mônadas de acordo com *Wadler* (1992). A nomenclatura utilizada é baseada em *Liang et al.* (1995).

$$\textbf{type}\ ErrT\ m\ a = m\ (Err\ a)$$

- transformador de environments:

$$\textbf{type}\ EnvT\ r\ m\ a = r \to m\ a$$

- transformador de continuações:

$$\textbf{type}\ ContT\ c\ m\ a = (a \to m\ c) \to m\ c$$

Assim, a mônada utilizada na especificação das equações semânticas passa a ser a seguinte:

```
type M a  =  EnvT Env                    -- environments
               (ContT Answer             -- continações
                  (StateT Store          -- stores
                     (StateT IO          -- entrada e saída
                        ErrT))) a        -- tratamento de erros
```

Uma consequência do mecanismo de transformadores de mônadas é abordada por *Liang and Hudak* (1996), em que geração de código em um compilador é generalizado por meio da definição de uma mônada padrão com um conjunto de combinadores primitivos, mapeáveis diretamente para a linguagem do código gerado. Os trabalhos de *Harrison and Kamin*

Equações semânticas:

```
type Environment = [(Name,Value)]

interp  ::  Term → Environment → M Value
interp  (Var x)  e  = lookup x e
interp  (Con i)  e  = return (Num i)
interp  (Add u v) e = interp  u  e  >>= \a → interp v e
                      >>= \b → add a b
interp  (Lam x v) e = return (Fun (\a → interp v ((x,a):e)))
interp  (App t u) e = interp  t  e  >>= \f → interp u e
                      >>= \a → apply f a
```

Figura 3.11: Exemplo de interpretador escrito em linguagem Haskell, utilizando mônadas – extraído de *Wadler* (1992).

(2000, 1998) também utilizam transformadores de mônadas para a geração de compiladores, mas diferentemente de *Liang and Hudak* (1996), o processo de geração de compiladores consiste em aplicar avaliação parcial (*Jones et al.* 1993) no programa do interpretador.

Ivanović and Kuncak (2000) proveem uma forma de definição modular de sintaxe e semântica de uma linguagem utilizando Haskell e semântica de mônadas. A técnica aplicada nesse trabalho é a modularização em duas dimensões: as fases de compilação, isto é, análises léxica, sintática e semântica; e as construções da linguagem. A técnica proposta consiste em um conjunto de diretrizes para a implementação rápida de protótipos de linguagens, permitindo a sua execução para depuração. Além disso, a escrita da especificação é incremental, de modo que a cada nova construção linguística, cria-se um módulo que define seus elementos léxicos, sintáticos e semânticos.

A semântica de mônadas é um modelo computacional bastante poderoso e oferece meios para se abstrair *stores*, *environments* e continuações, muitas vezes indesejáveis nas equações semânticas. Além disso, o modelo é completamente denotacional, de modo que a possibilidade de se provar propriedades da especificação é compartilhado pelas duas técnicas.

Por outro lado, o poder de expressão deste modelo depende da forma como os transformadores de mônadas são combinados. Como destacado por *Liang et al.* (1995), algumas operações de *lift* sobre funções básicas não podem ser feitas de forma automática, sendo necessário um tratamento caso a caso em diversas situações. No pior caso, para n trans-

Exemplos de Mônadas:

- Mônada de estado:

```
type M a = State → (a, State)

return a = \s0 → (a, s0)
m >>= k = \s0 → let (a,s1) = m s0 in k a s1
```

- Mônada de estado:

```
type M a = (a -> Answer) -> Answer

return a = \c -> c a
m >>= k = \c -> m (\a -> k a c)
```

Figura 3.12: Exemplos de mônadas para o interpretador escrito em linguagem Haskell da Figura 3.11.

formadores de mônadas, pode ser necessário especificar $O(n^2)$ operações *ad-hoc* de *lift*. Segundo *Liang et al.* (1995), isto não constitui um grande problema, pois o número de transformadores utilizados na prática é relativamente pequeno; no entanto, mostra que a escalabilidade do modelo é comprometida.

3.2.3 Semântica de Ações com Mônadas

A semântica de ações com mônadas foi proposta por *Wansbrough and Hamer* (1997) e é uma fusão da semântica de mônadas de *Liang et al.* (1995) e da semântica de ações de *Mosses* (1993). Nesse formalismo, a semântica de cada construção linguística é uma sequência de ações, definidas por meio de semântica de mônadas, e não por meio de semântica operacional estruturada.

Segundo *Wansbrough and Hamer* (1997), *Zhang and Xu* (2004), esse modelo compartilha as principais vantagens das semântica de ações e de mônadas: as definições semânticas são legíveis, uma vez que a notação utilizada é a mesma da semântica de ações; por utilizar a semântica de mônadas, a definição da notação de ações fica modularizada, diferente da definição monolítica oferecida pela semântica operacional estruturada; a inclusão de novas

noções de computação pode ser feita por meio dos mecanismos apresentados por *Liang et al.* (1995), o que torna o modelo mais flexível que a semântica de ações; o raciocínio equacional é proporcionado pela base matemática subjacente à semântica denotacional e à semântica de mônadas.

Entretanto, de acordo com *Wansbrough and Hamer* (1997), ao substituir a semântica operacional estrutura pela semântica de mônadas, limita-se o poder da semântica de ações, tornando inviável lidar com múltiplos processos que se comuniquem. Além disso, no modelo original de Mosses é possível que a semântica de um programa não-determinista seja o conjunto de todas as possíveis respostas geradas pelo programa.

3.2.4 Gramáticas de Atributos de Primeira Classe

de Moor et al. (2000) propõem uma técnica orientada por aspectos para melhorar a modularidade de gramáticas de atributos, que pode ser aplicada a definições semânticas. No modelo apresentado, atributos podem ser definidos em seções separadas e podem ser intercalados para formar uma gramática de atributos pura. Estas seções permitem que a definição de um atributo indique a forma como ele se comporta ao longo da especificação, podendo ser sintetizado, herdado ou transmitido automaticamente ao longo de uma cadeia de definições recursivas.

O suporte de aspectos na definição de atributos permite, por exemplo, a criação de definição para repetição que é vaga em relação a sequenciadores. No entanto, interações entre construções linguísticas pode precisar de informações que não estão disponíveis para definição do atributo, especialmente quando forem desacopladas da estrutura sintática. Por exemplo, a relação entre chamadas de métodos e tratamento de exceções em uma linguagem orientada por objetos típica não pode ser expressa diretamente por meio de atributos herdados, sintetizados ou encadeados, uma vez que em geral não há relação sintática entre estas construções em um programa típico.

3.2.5 Semântica Construtiva

Reusabilidade pode ainda ser obtida por meio da abordagem construtiva proposta por *Mosses* (2005). Essa abordagem parte do pressuposto que uma linguagem pode ser definida como a combinação de um conjunto representativo de construções básicas abstratas, que são formalmente definidas. Assim, a semântica das diversas construções de uma linguagem

pode ser traduzida nestas abstrações, sendo possível a definição direta da semântica da linguagem.

Por exemplo, sequenciadores podem ser definidos como exceções a serem tratadas por meio de um mecanismo de tratamento de exceções associado aos comandos de repetição. No entanto, como resultado, elementos para tratamento de sequenciadores podem permanecer entrelaçados à definição de repetições, o que pode trazer prematuramente a preocupações sobre o efeito de tais exceções.

3.3 Conclusões

A especificação da semântica formal é um campo de pesquisa bastante relevante, motivado especialmente pelo pontencial de aplicações possíveis. Entretanto, ao contrário da especificação formal da sintaxe, que é amplamente utilizada em diversas sub-áreas da Computação, há grandes barreiras à popularização das técnicas de especificação formal da semântica de linguagens de programação.

Dentre os principais dificultadores encontra-se o fato de que nenhum dos modelos vigentes tenha se consolidado, oferecendo um arcabouço de uso prático e teórico. Em particular, o baixo grau de modularidade que se obtém nos diversos modelos dificulta a escrita de novas definições e o seu uso para a prova de propriedades.

Do ponto de vista da apresentação de uma linguagem, uma característica desejável é definir uma ordem de leitura na qual, a partir de definições mais simples, o leitor é levado sucessivamente a níveis de maior complexidade, até chegar à compreensão de toda a linguagem. No entanto, nas técnicas vigentes, as equações semânticas tendem a não ser coesas, uma vez que a definição de uma construção pode conter muitos detalhes relacionados a outras construções. Neste caso, a leitura tende a ser mais difícil, pois não há uma ordem para a leitura que leve o aprendiz de um nível mais simples para um nível de maior complexidade. Com efeito, encontram-se dificuldades relacionadas à apresentação para as técnicas de definição de semântica denotacional vigentes.

As técnicas de semântica de ações e semântica de ações com mônadas definem um vocabulário mais rico para especificação da semântica, permitindo que o uso de elementos prosaicos, tais como *stores* e *environments*, sejam abstraídos nas equações semânticas. No entanto, não é possível separar as equações semânticas para as diversas construções. Com isso, não é possível derivar uma ordem de leitura sugerida para a definição. Por exemplo,

em ambas as técnicas, a definição de um comando de repetição deve levar em consideração a existência de possíveis sequenciadores que levem ao término abrupto das iterações.

Em semântica de mônadas, as funções semânticas mapeiam cada construção em uma computação de valores, de modo que a forma como a computação é definida pode evoluir durante a apresentação. No entanto, a questão da separação de construções ainda não é resolvida, o que leva a equações semânticas não-coesas, de maneira semelhante à Semântica de Ações. Com isso, ainda existe o entrelaçamento de definições, o que pode dificultar encontrar uma ordem de leitura para as equações semânticas. Por exemplo, o término abrupto de repetições por meio de sequenciadores deve também ser levado em consideração ao se especificar comandos de repetição.

A técnica de gramáticas de atributos de primeira classe é um passo à frente na solução deste problema, uma vez que permite a definição separada da interação entre construções que possuam relação de hierarquia na árvore de sintaxe abstrata de um programa. No entanto, a técnica não é apropriada quando a comunicação se dá de maneira independente da árvore, como por exemplo, no retorno abrupto de função devido a lançamento de exceções.

Uma abordagem diferente para o problema é a semântica construtiva, em que as construções de uma linguagem de programação são definidas como combinações de um conjunto de construções padrão de uma linguagem base. Neste caso, o leitor deve inicialmente compreender algumas construções da linguagem base, pois as equações semânticas da linguagem em estudo consistem na tradução de cada construção em uma combinação de construções da linguagem base. Esta técnica permite a leitura de um nível mais simples para níveis mais sofisticados, mas ainda não permite a definição separada de construções mais elementares da própria linguagem.

O próximo capítulo apresenta a técnica de semântica incremental, que é a principal contribuição deste trabalho de Tese. A técnica proposta permite que o projetista apresente uma especificação inicial vaga da linguagem, em que detalhes e construções mais complexas possam ser ignorados. Para permitir extensões linguísticas, a linguagem de definição permite que as denotações de construções já definidas evoluam de maneira uniforme, de modo que novas construções possam ser adicionadas. Além disso, a forma como a definição incremental é feita não impede que se utilizem as contribuições das demais técnicas de modularidade de semântica denotacional existentes.

Com as técnicas existentes, se o projetista desejar apresentar versões iniciais mais simples ao leitor, torna-se necessário criar e entregar-lhe várias versões de um único documento, em que todas as construções são novamente definidas. Na técnica proposta neste trabalho,

o projetista pode entregar um único documento que apresenta versões simplificadas das construções e como estas evoluem para que novas construções possam ser acrescentadas.

Capítulo 4

Semântica Incremental

Este capítulo apresenta a metodologia desenvolvida como parte da solução para os problemas levantados nos capítulos anteriores, apresentada resumidamente por *Tirelo et al.* (2009). A premissa desta abordagem consiste no conjunto de vantagens que se obtêm se definições da semântica de linguagens de programação forem apresentadas de maneira incremental, uma vez que tais definições são geralmente sistemas muito grandes e complexos. Desta maneira, a partir de uma definição existente da linguagem, seria desejável que novas construções pudessem ser definidas sobre as equações já existentes.

O cenário imaginado no desenvolvimento deste trabalho consiste em um projetista (especialista) de linguagem que tenta desenvolver uma versão legível da especificação semântica para um leitor (não-especialista). É foco do trabalho do projetista encontrar a maneira mais clara de apresentar as construções ao leitor, e, por isso, o trabalho de reescrita de construções já definidas é muitas vezes preferível, se isso facilitar posteriormente a compreensão de construções mais avançadas. Do ponto de vista do leitor, o documento gerado apresenta inicialmente versões simplificadas das construções fundamentais de sua linguagem, que posteriormente evoluem para versões mais completas e que englobem todos os aspectos de seu funcionamento.

Utilizando as técnicas existentes, o projetista consegue esse efeito fornecendo versões completas da especificação, em que cada equação semântica pode ser totalmente redefinida. Na técnica proposta, tem-se por objetivo permitir que o projetista forneça um único documento, no qual a denotação de cada construção seja inicialmente definida de maneira simplificada (ou vaga) e evolua à medida que novas construções sejam adicionadas.

Organização do Capítulo. A Seção 4.1 discute aspectos essenciais de uma definição incremental e a importância da vagueza, conceito fundamental à metodologia proposta. A Seção 4.2 apresenta a forma geral de uma definição incremental segundo o modelo proposto. A Seção 4.3 elabora os mecanismos linguísticos utilizados para definição incremental. A Seção 4.4 apresenta a formalização do modelo proposto. A Seção 4.5 contém estudos de caso com o intuito de se verificar a aplicabilidade da técnica a linguagens de grande porte e da teoria subjacente baseada em semântica denotacional clássica e semântica monádica. Finalmente, a Seção 4.6 apresenta as conclusões do capítulo.

4.1 Vagueza e Legibilidade

Neste trabalho, utiliza-se a vagueza como mecanismo de melhoria da legibilidade das definições semânticas. Esta ideia se baseia em uma prática muito usual por parte de professores de programação, que consiste em abstrair conceitos avançados durante a explicação de conceitos básicos. Em um momento futuro, a introdução destes conceitos avançados pode exigir a adequação dos conceitos previamente apresentados, sem no entanto implicar na completa quebra de sua natureza básica. Tal explicação *vaga* é adequada naquele contexto porque em geral requer menos esforço do aprendiz para o entendimento dos conceitos linguísticos.

Tal prática é também amplamente utilizada em livros e manuais de linguagens de programação, onde ao se introduzir um novo conceito, basta explicar como todo o corpo de conhecimento já adquirido deve ser repensado para se adequar aos novos conceitos. Por exemplo, na definição de uma linguagem orientada por objetos típica, considere que o capítulo onde se explicam métodos anteceda o capítulo correspondente a tratamento de exceções; é plenamente possível explicar métodos sem mencionar tratamento de exceções, apesar de ambos compartilharem contexto comum. Neste caso, a definição de métodos não pode ser considerada incorreta, pois o programador pode construir uma série de programas corretos sem se preocupar naquele primeiro momento com a forma com que casos excepcionais devam ser tratados. A apresentação pode ser feita de maneira incremental, fornecendo-se definições vagas em relação a conceitos ainda não explorados, ação em que se espera que o aprendiz adquira experiência em conceitos básicos antes de estudar aqueles mais avançados.

Porém, no momento da explicação dos conceitos mais avançados, pode ser necessário rever os conceitos básicos sob uma nova ótica, pois o contexto fornecido pelas explicações vagas

anteriores pode ser insuficiente para as novas definições. No exemplo anterior, a explicação de tratamento de exceções exige explicar que o retorno de um método pode passar pela execução do corpo de cláusula *finally*, antes do retorno ao ponto de chamada. Vale ressaltar no entanto que esta reinterpretação consiste na escrita de alguns poucos parágrafos e não na reescrita completa dos capítulos anteriores. No texto informal, tais reinterpretações podem ser feitas de diversas maneiras, com destaque para algumas táticas, que são as tratadas neste trabalho de Tese dentro do contexto de métodos formais.

A primeira tática consiste na definição de um novo elemento associado a diversas construções da linguagem com influência direta em algumas poucas construções. Por exemplo, considere a definição de sequenciador *break* após a definição de comandos de repetição em uma linguagem imperativa típica. O ponto de desvio do sequenciador é definido pelo comando de repetição para ser utilizado somente na definição do sequenciador; todos os demais comandos da linguagem, tais como comandos condicionais, ignoram este valor e simplesmente o propagam para seus sub-comandos.

Outra tática usual é a redefinição de valores propagados pelas construções para seus constituintes. No exemplo anterior de definição de sequenciadores após a definição de comandos de repetição, pode-se considerar que o ambiente de execução de comandos mapeia uma posição especial no ponto de desvio do sequenciador.

Além destas duas táticas, pode-se considerar ainda que valores compostos produzidos durante a computação incorporem informações adicionais. Por exemplo, considere a definição semântica da linguagem Java, em que a hierarquia de tipos é explicada após a apresentação de arranjos de objetos na linguagem; erros como o da Figura 4.1 não podem ser detectados na semântica estática, e para detectá-los, pode-se considerar que cada expressão do lado esquerdo de uma atribuição carregue consigo o tipo mais alto na hierarquia que pode ser atribuído.

No contexto de definições informais, a vagueza, aliada a um mecanismo de reinterpretação de conceitos anteriormente definidos, é recurso essencial à escrita de explicações incrementais legíveis.

Ao contrário, especificações de semântica denotacional possuem um único conjunto de equações que fornecem um único mapeamento do universo sintático no universo semântico, e, por isso, a especificação de uma construção não pode ser fornecida em etapas. Assim, dois cenários típicos podem ser vislumbrados tendo como atores o projetista da linguagem e o leitor da especificação.

```
1 // Consider A superclass of B
2 A f(int x) {
3     if (x >= 0) return new B();
4     else return new A();
5 }
6 ...
7 int y = ...;
8 A arr [] = new B[10];
9 arr [0] = f(y); // If y < 0, an execution error is issued
```

Listagem 4.1: Interdependência de Arranjos de Objetos e Hierarquia de Tipos em Java

No primeiro, o projetista fará a definição incremental da linguagem e fornecerá somente a versão final ao leitor. Do ponto de vista do projetista, essa atividade pode requerer a reescrita de diversas equações principalmente quando as alterações necessárias exigirem, por exemplo, a inclusão de um novo argumento em algumas equações. Do ponto de vista do leitor, a definição fornecida pode ser bastante confusa e com diversos elementos intrusos e espalhados nas equações. Tome, por exemplo, a definição denotacional de uma linguagem orientada por objetos típica contendo métodos e tratamento de exceções. Uma forma de definir as construções correspondentes consiste no seguinte conjunto de equações semânticas, onde os elementos destacados são intrusos nas equações:

- $\mathcal{C}[\![\textbf{try } C_1 \textbf{ catch } I\ C_2 \textbf{ finally } C_3]\!]rc = \mathcal{C}[\![C_1]\!]r';\ \mathcal{C}[\![C_3]\!]rc$

  where $r' = r[\underline{k'_R}, k'_T/\underline{\mathsf{return}}, \mathsf{throw}]$
  $\underline{k'_R} = \underline{\lambda v.\mathcal{C}[\![C_3]\!]r; r\ \mathsf{return}\ v}$
  $k'_T = \mathit{ref}; \lambda l.\mathcal{C}[\![C_2]\!]r'';\ \mathcal{C}[\![C_3]\!]r;\ r\ \mathsf{throw}\ v$
  $r'' = r[\underline{k'_R}, k''_T, l/\underline{\mathsf{return}}, \mathsf{throw}, I]$
  $k_{T''} = \lambda v.\mathcal{C}[\![C_3]\!]r; r\ \mathsf{throw}\ v$

- $\mathcal{C}[\![\textbf{throw } E]\!]rc = \mathcal{E}[\![E]\!]r; r\ \mathsf{throw}$

- $\mathcal{D}[\![I_1\ (I_2)\ \{\ C\ \}]\!]ru = u\ r[f/I_1]$

  where $f = \lambda k_R \underline{k_T}.\mathit{ref};\ \lambda l.\mathcal{C}[\![C]\!]\ r'; k_R\ \mathit{novalue}$
  $r' = r[f, l, k_R, \underline{k_T}/I_1, I_2, \mathsf{return}, \underline{\mathsf{throw}}]$

- $\mathcal{C}[\![\textbf{return } E]\!]rc = \mathcal{E}[\![E]\!]r; r\ \mathsf{return}$

- $\mathcal{E}[\![E_1(E_2)]\!]rk = \mathcal{E}[\![E_1]\!]r; \lambda f.\mathcal{E}[\![E_2]\!]r; f\ k\ \underline{(r\ \mathsf{throw})}$

No segundo cenário, o projetista fornecerá todas as especificações intermediárias da linguagem, e com isso o leitor pode estudar os conceitos fundamentais em um conjunto simplificado das equações. Esta solução ainda não é satisfatória, pois outro problema ainda mais complexo pode ser vislumbrado tanto para o projetista quanto para o leitor. Do ponto de vista do projetista, alterações feitas nas especificações iniciais podem ser replicadas ao longo de diversas versões do documento. Além disso, não há garantias de que as propriedades válidas em uma especificação continuem válidas na seguinte; esta garantia só pode ser obtida a partir de argumentação favorável para cada reificação realizada. Do ponto de vista do leitor, o trabalho de releitura das diversas equações pode ser bastante trabalhoso, e a evolução das denotações pode ficar perdida no meio de um grande volume de dados. Além disso, a reescrita de um grande número de equações continua uma exigência em potencial, mesmo para alterações simples.

A metodologia desenvolvida tem por objetivo permitir a especificação incremental de linguagens de programação com recurso de vagueza. A técnica subjacente é a usualmente utilizada em livros de linguagens de programação:

- primeiramente, apresentam-se em um primeiro capítulo os elementos mais fundamentais para a escrita de um programa completo; em geral, tal definição pode incluir o conceito de programas, módulos, funções, comandos básicos e expressões básicas;
- os capítulos subsequentes têm por objetivo agregar novos conceitos ao conjunto já existente, o que por vezes pode exigir adequação dos elementos já apresentados.

As próximas seções deste capítulo apresentam os conceitos de especificação semântica e definição incremental e apresenta os mecanismos linguísticos propostos para permitir alcançar adequado grau de incrementabilidade em definições semânticas.

4.2 Definições Incrementais

Uma definição incremental da semântica de uma linguagem é uma sequência $S_0, S_1, \cdots, S_n$, onde cada S_i é a especificação da semântica de um subconjunto da linguagem, tal que S_0 é uma especificação inicial, e cada S_{i+1} define uma extensão de S_i. Cada nova especificação pode adicionar novos elementos, mas por vezes pode ser necessário adaptar equações previamente definidas. Dada uma especificação S_i, uma nova especificação $S_{i+1} = t_i(S_i) + \Delta S_i$ pode ser obtida incluindo-se novos elementos (ΔS_i) a uma versão adaptada de S_i por meio de uma função de transformação t_i.

A transformação t_i é uma função que mapeia especificações semânticas em especificações semânticas e tem por objetivo adaptar o comportamento de equações semânticas. Uma inclusão em uma especificação S, denotada por ΔS, representa um conjunto de novos elementos adicionados à especificação, usualmente elementos relacionados a novas construções linguísticas.

O exemplo explorado ao longo deste capítulo consiste em uma especificação S_i de uma linguagem hipotética L, composta por expressões, comandos e declarações. A sintaxe abstrata, as funções semânticas e as equações semânticas da especificação S_i para as construções mais relevantes para a discussão são apresentadas na Figura 4.1. Nestas equações[1], a definição do comando *while* é vaga em relação à existência de sequenciadores. No caso de a especificação S_{i+1} definir o sequenciador *break*, torna-se necessário adaptar, por meio da transformação t_i, a equação relativa ao comando *while* para preparar o contexto no qual eventuais sequenciadores possam ser executados. As inclusões correspondentes consistem, desta maneira, em definir uma nova regra na gramática abstrata para o sequenciador *break* e em uma nova equação semântica que o defina.

O restante deste texto tratará das funções de transformação, que são a principal contribuição deste trabalho. Outros tipos de inclusão podem ser obtidos por meio de recursos de modularidade usuais em linguagens de programação modernas e, portanto, não serão discutidos em profundidade. Apesar de o exemplo utilizado ser baseado em semântica de continuações tradicional, a técnica proposta é adequada ao uso com outros recursos de melhoria de modularidade, tais como mônadas, conforme mostrado no estudo de caso da Seção 4.5.2.

4.3 Funções de Transformação

Uma extensão da linguagem pode afetar funções e equações semânticas existentes ao requerer que:

1. assinaturas de funções sejam redefinidas para inclusão de novos argumentos (Seção 4.3.1);

[1] FIX representa o operador de ponto fixo.

Abstract syntax:

$P \in Prog \rightarrow D; C$
$C \in Com \rightarrow \textbf{while}\ E\ C \mid C_1; C_2 \mid \cdots$
$E \in Exp \rightarrow \cdots$
$D \in Dec \rightarrow \cdots$

Semantic Functions:

$\mathcal{P} : Prog \rightarrow Ans$
$\mathcal{C} : Com \rightarrow Env \rightarrow Cc \rightarrow Cc$
$\mathcal{E} : Exp \rightarrow Env \rightarrow (Val \rightarrow Cc) \rightarrow Cc$
$\mathcal{D} : Dec \rightarrow Env \rightarrow (Env \rightarrow Cc) \rightarrow Cc$

Semantic Equations:

$\mathcal{P}[\![D; C]\!] = \mathcal{D}[\![D]\!]\ r_0\ (\lambda r.\mathcal{C}[\![C]\!]\ r\ (\lambda s.stop))\ s_0$
$\mathcal{C}[\![C_1; C_2]\!]\ r\ c = \mathcal{C}[\![C_1]\!]\ r;\ \mathcal{C}[\![C_2]\!]\ r\ c$
$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r = \text{FIX}\ \lambda f c.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ (f\ c)\ \textbf{else}\ c$
Other equations defining $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{D}$

Figura 4.1: Exemplo utilizado na explicação dos transformadores – apenas construções relevantes à discussão são apresentadas.

2. argumentos de função e valores de retorno sejam decorados[2], no intuito de tratar situações não previstas ou adequar as equações a alterações de assinatura (Seção 4.3.2);

3. algumas equações sejam completamente redefinidas no caso de transformações automáticas não serem adequadas ou simples de definir (Seção 4.3.3).

Uma função é promovida por uma transformação t se for sujeita a alguma modificação definida em t. Transformações podem ser definidas por meio das seguintes construções, cujos constituintes serão definidos nas Seções 4.3.1-4.3.3:

transformation *transformation-name*
signature *signature clauses*
default *default clauses*
replace *decoration clauses*
redefine *redefinition clauses*

Dada uma especificação S e uma transformação t, $S' = t(S)$ é definida em duas etapas: a primeira coleta as transformações que deverão ser realizadas em cada função individualmente, produzindo uma sequência $\langle t_1, t_2, \cdots, t_m \rangle$, onde cada t_i pode ser uma inclusão de argumento, uma decoração ou uma redefinição de equação; o segundo passo aplica

[2]Decoração possui o sentido estabelecido na definição de padrões de projeto por *Gamma et al.* (1995). Na verdade, neste trabalho, uma decoração pode ser vista como a implementação orientada por aspectos deste padrão por meio de adendos de contorno (*Kiczales et al.* 1997).

cada transformação a cada uma das funções promovidas. A ordem de aplicação destas transformações é a ordem em que elas aparecem na sua definição.

4.3.1 Inclusão de Argumentos

A inclusão de argumentos é feita por meio da cláusula *signature*, que possui o seguinte formato:

$$\textbf{signature} \quad f_1 : T'_1, f_2 : T'_2, \cdots, f_m : T'_m$$

Nesta cláusula, cada função f_i do tipo T_i deve ser transformada em uma função cujo tipo é T'_i. Cada T'_i é um tipo funcional com a forma $t_1 \to t_2 \to \cdots \to t_n$, em que cada t_j pode ser uma expressão de tipos ou uma expressão de tipos rotulada, com a forma $l : t$, onde l é um rótulo e t é uma expressão de tipos. Tipos rotulados são utilizados para indicar inclusão de novos argumentos na função.

Uma cláusula de alteração de assinatura é válida somente se as seguintes regras de validação puderem ser aplicadas:

- (única ocorrência) cada rótulo aparece uma única vez em cada T'_i;
- (consistência interna) se $(l : t_1)$ é um componente de T'_i, e $(l : t_2)$ é um componente de T'_j, então $t_1 = t_2$;
- (consistência externa) se $T_i = t_1 \to t_2 \to \cdots \to t_n$ e $T'_i = t'_1 \to t'_2 \to \cdots \to t'_m$, então a expressão de T'_i deve apenas prever a inclusão de novos argumentos, respeitando os tipos e a ordem dos argumentos já existentes; para isso, $t_1 \to t_2 \to \cdots \to t_n$ e $t'_1 \to t'_2 \to \cdots \to t'_m$ devem obedecer às seguintes regras de consistência:
  - o tipo do resultado da função não pode ser alterado, isto é, $t_n = t'_m$;
  - se t'_1 não for um tipo rotulado, então $t_1 = t'_1$ e $t_2 \to \cdots \to t_n$ e $t'_2 \to \cdots \to t'_m$ são consistentes;
  - se t'_1 for um tipo rotulado, então $t_1 \to t_2 \to \cdots \to t_n$ e $t'_2 \to \cdots \to t'_m$ são consistentes, e t'_1 corresponde a um novo argumento que deve ser incluído na função correspondente.

Novos argumentos incluídos por meio de cláusulas de alteração de assinatura são automaticamente transmitidos ao longo das aplicações recursivas das funções envolvidas. Na falta

de tal argumento, um valor padrão a ser inserido nas aplicações originais pode ser definido por meio da cláusula *default* da transformação, que possui o formato:

$$\textbf{default} \quad l_1 = c_1, l_2 = c_2, \cdots, l_n = c_n$$

onde cada l_i é um rótulo para o tipo t_i em algum tipo rotulado nas cláusulas de alteração de assinatura, e cada c_i é uma expressão cujo tipo é t_i.

Por exemplo, considere a necessidade de se adaptarem equações do exemplo da Figura 4.1 para a inclusão do sequenciador *break*. Uma possível solução consiste em incluir um novo argumento na função semântica $\mathcal{C}$, representando a continuação para o *break*, conforme definido na transformação *include_break_a*:

transformation *include_break_a*
signature $\mathcal{C} : Com \to Env \to b : Cc \to Cc \to Cc$
default $b = \lambda s.error$

Em toda aplicação da função $\mathcal{C}$ um novo terceiro argumento $\lambda s.error$ será automaticamente inserido, exceto quando se tratar de uma aplicação recursiva da função, situação em que o parâmetro formal correspondente será simplesmente propagado. A aplicação da transformação *include_break_a* produz as seguintes versões modificadas das equações semânticas da Figura 4.1[3]:

$\mathcal{C} : Com \to Env \to Cc \to Cc \to Cc$
$\mathcal{P}[\![D; C]\!] = \mathcal{D}[\![D]\!]\ r_0\ (\lambda r.\mathcal{C}[\![C]\!]\ r\ (\lambda s.error)\ (\lambda s.stop))\ s_0$
$\mathcal{C}[\![C_1; C_2]\!]\ r\ b\ c = \mathcal{C}[\![C_1]\!]\ r\ b;\ \mathcal{C}[\![C_2]\!]\ r\ b\ c$
$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r\ b\ c = (\text{FIX}\ \lambda f c'.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ b\ (f\ c')\ \textbf{else}\ c')\ c$

O sequenciador *break* pode então ser definido como: $\mathcal{C}\ [\![\textbf{break}]\!]\ r\ b\ c = b$.

Em certas situações, a inserção de valor *default* para argumento pode ser feita de maneira mais simples se for possível associar na expressão a ser inserida informações de contexto da função em que a chamada ocorre. Por exemplo, para a inclusão do sequenciador *break* na definição da Figura 4.1, poder-se-ia considerar que o valor da nova continuação seria propagado nas chamadas recursivas da função semântica de comandos, e a continuação de saída do comando de repetição seria automaticamente inserida como continuação de interrupção na chamada do comando a ser repetido.

[3]Equações não afetadas pela transformação não foram reproduzidas. Esta versão ainda não permite a incorporação completa do sequenciador *break*, pois será ainda necessário o uso de decoradores.

Isto pode ser feito associando-se a cada cláusula *default* uma indicação de contexto definida por meio de um nome de função e dos parâmetros que identifiquem a equação onde a inserção se dará. Desta maneira, cada cláusula $l_i = c_i$ pode opcionalmente possuir identificação de contexto, assumindo nestes casos a forma:

$$l_i \ [\ \textbf{within}\ g_i\ p_{i1}\ \cdots\ p_{ik_i}\ ] = c_i,$$

onde g_i é um nome de função, e cada p_{ij} é um padrão utilizado para identificar a equação onde a inserção será feita. Os identificadores presentes em cada p_{ij} podem ser utilizados na expressão c_i e serão amarrados aos valores correspondentes nos parâmetros reais da aplicação da função g_i.

Por exemplo, a transformação a seguir inclui a continuação de saída do comando de repetição como valor *default* na chamada da função semântica referente ao corpo da repetição:

transformation *include_break_b*
signature $\mathcal{C} : Com \rightarrow Env \rightarrow b : Cc \rightarrow Cc \rightarrow Cc$
default $b = \lambda s.error,$
b **within** $\mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r\ c = c$

4.3.2 Decoradores de Função

Certas extensões linguísticas podem ser implementadas de maneira bastante simples se os argumentos passados para as funções semânticas forem adaptados. Por exemplo, o sequenciador *break* pode ser incluído sem dificuldades na especificação alterando-se o ambiente de execução do corpo do comando *while* para armazenar a continuação de saída da repetição. Tal adaptação é feita por meio de cláusulas de decoração, cujo formato é o seguinte:

$$\begin{aligned}\textbf{replace}\quad & f_1\ p_{11}\ \cdots\ p_{1k_1}\ [\ \textbf{within}\ g_1\ p'_{11}\ \cdots\ p'_{1m_1}\ ]\ \textbf{by}\ e_1,\\ & \cdots,\\ & f_n\ p_{n1}\ \cdots\ p_{nk_n}\ [\ \textbf{within}\ g_n\ p'_{n1}\ \cdots\ p'_{nm_n}\ ]\ \textbf{by}\ e_n\end{aligned}$$

onde cada f_i é uma função do tipo $t_{i1} \rightarrow t_{i2} \rightarrow \cdots \rightarrow t_{ik_i} \rightarrow t_i$, p_{ij} é um padrão para o j-ésimo argumento de f_i, e e_i é uma expressão do tipo t_i. O efeito desta transformação é substituir todas as aplicações da função f_i que casem com os padrões correspondentes pela expressão e_i. Opcionalmente, cada cláusula *replace* pode definir o contexto no qual a decoração se aplica, de maneira semelhante às cláusulas *default*. Neste caso, a decoração só é feita para aplicações da função f_i que ocorrerem nas equações correspondentes à função g_i definidas pelos padrões p'_{ij}.

O ambiente de avaliação da expressão e_i amarra os identificadores presentes nos padrões p_{ij} e p'_{ij} aos valores deduzidos pelo casamento dos padrões com os parâmetros reais correspondentes. Desta maneira, uma definição de decoração típica no formato:

$$\textbf{replace} \quad f\ p_1\ \cdots\ p_k\ \textbf{within}\ g\ p'_1\ \cdots\ p'_m \qquad \textbf{by} \qquad f\ e_1\ \cdots\ e_k$$

pode elaborar os novos argumentos para a função a partir dos argumentos passados nas equações já existentes.

Por exemplo, pode-se completar a transformação *include_break_a* da Seção 4.3.1, incluindo-se uma cláusula de decoração, o que produz a seguinte transformação:

$$\begin{array}{l}\textbf{transformation}\ \textit{include\_break\_a}\\ \quad \textbf{signature} \quad \mathcal{C} : \mathit{Com} \to \mathit{Env} \to b : \mathit{Cc} \to \mathit{Cc} \to \mathit{Cc}\\ \quad \textbf{default} \quad b = \lambda s.\mathit{error}\\ \quad \textbf{replace} \quad \mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r\ b\ c \qquad \textbf{by} \qquad \mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r\ c\ c\end{array}$$

A cláusula de decoração acima afeta somente a equação que define o comando de repetição alterado da Seção 4.3.1, e a versão modificada é:

$$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r\ b\ c = (\textsc{fix}\ \lambda f c'.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ c\ (f\ c')\ \textbf{else}\ c')\ c.$$

Outra forma de preparar a definição para a inclusão do sequenciador *break* é apresentada na transformação *include_break_c* a seguir, em que não há a necessidade de se alterar a assinatura da função de denotação de comandos. Esta solução consiste em substituir a aplicação da função semântica de comandos, no contexto da definição da repetição, para que o ambiente considerado associe a continuação de saída da repetição a uma posição especial break:

$$\begin{array}{l}\textbf{transformation}\ \textit{include\_break\_c}\\ \quad \textbf{replace} \quad \mathcal{C}\ [\![C]\!]\ r\ c\ \textbf{within}\ \mathcal{C}\ [\![\textbf{while}\ E\ C']\!]\ r'\ c'\\ \qquad\qquad \textbf{by}\ \mathcal{C}\ [\![C]\!]\ r[c'/\mathsf{break}]\ c\end{array}$$

Esta cláusula de decoração afetará somente a equação que define o comando de repetição na Figura 4.1, e a versão modificada é:

$$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r\ c = (\textsc{fix}\ \lambda f c'.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r[c/\mathsf{break}]\ (f\ c')\ \textbf{else}\ c')\ c.$$

4.3.3 Redefinição de Equações

Em algumas situações, pode ser mais apropriado reescrever a definição de algumas construções por meio de cláusulas de redefinições do que tentar adaptar equações existentes. Estas cláusulas possuem o seguinte formato:

$$\textbf{redefine} \quad f_1\ p_{11}\ \cdots\ p_{1k_1}\ \textbf{by}\ e_1, \cdots, f_n\ p_{n1}\ \cdots\ p_{nk_n}\ \textbf{by}\ e_n$$

onde cada f_i é uma função do tipo $t_{i1} \rightarrow t_{i2} \rightarrow \cdots \rightarrow t_{ik_i} \rightarrow t_i$, p_{ij} é um padrão para o j-ésimo argumento de f_i, e e_i é um expressão do tipo t_i. Por exemplo, o comando *while* pode ser redefinido por meio da seguinte transformação, cujo objetivo é fazer com que a continuação para o sequenciador seja passada via ambiente de execução para o corpo da repetição:

transformation *include_break_d*
  redefine $\mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r$ **by**
    FIX $\lambda f c.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r[c/\mathsf{break}]\ (f\ c)\ \textbf{else}\ c$

4.4 Formalização do Modelo

Esta seção apresenta a semântica denotacional de definições incrementais, por meio dos domínios sintáticos, domínios semânticos, funções semânticas e equações semânticas. Considera-se que uma especificação pode ser uma especificação inicial ou uma extensão de alguma especificação existente.

4.4.1 Domínios Sintáticos

Uma especificação inicial é composta por um conjunto de módulos nos quais se pode definir sintaxe concreta, sintaxe abstrata, domínios semânticos e funções semânticas da especificação. Uma extensão é composta por uma transformação, a especificação transformada e um conjunto de módulos. Especificações são, desta maneira, definidas por meio da seguinte gramática abstrata:

$$\begin{array}{lcl} s \in S & \rightarrow & \textbf{specification}\ M^* \\ & | & \textbf{extension}\ T\ S\ M^* \end{array}$$

onde M representa o domínio dos módulos e T o domínio das transformações. Uma transformação pode ser uma inclusão de argumento, uma decoração, uma redefinição ou um conjunto de transformações, e possui a estrutura a seguir:

$$
\begin{array}{rcl}
t \in T & \rightarrow & \textbf{include}\ I\ Q\ X^* \\
& | & \textbf{replace}\ I\ P^*\ W\ E \\
& | & \textbf{redefine}\ I\ P^*\ E \\
& | & T_1; T_2
\end{array}
$$

A cláusula **include** corresponde a uma inclusão de um mesmo argumento definida por meio de cláusulas **signature** com valores especificados por meio de cláusulas **default**. Os domínios D, P, E e I representam, respectivamente, domínios, padrões, expressões e identificadores. O domínio Q representa a inclusão de um mesmo argumento em uma função, possuindo o seguinte formato:

$$
\begin{array}{rcl}
q \in Q & \rightarrow & \textbf{inclusion}\ I\ D^*\ D\ D \\
& | & Q_1; Q_2
\end{array}
$$

O domínio X corresponde a uma definição de valor *default* sendo representado por um contexto do domínio W e uma expressão. O domínio W pode ser representado por um nome de função e uma lista de padrões que designem uma equação, ou pelo símbolo **anywhere**, que representa que não há contexto específico para inserção do valor *default*.

$$
\begin{array}{rcl}
x \in Q & \rightarrow & \textbf{default}\ W\ E \\
w \in W & \rightarrow & \textbf{within}\ I\ P^* \\
& | & \textbf{anywhere}
\end{array}
$$

Por exemplo, o nó de árvore de sintaxe abstrata correspondente à transformação *include_break_b* da Seção 4.3.1 (página 62) é igual a $t_b = \textbf{include}\ b\ q\ \langle x_1, x_2 \rangle$, onde:

$$
\begin{array}{rcl}
q & = & \textbf{inclusion}\ \mathcal{C}\ \langle Com, Env \rangle\ Cc\ (Cc \rightarrow Cc) \\
x_1 & = & \textbf{default anywhere}\ (\lambda s.error) \\
x_2 & = & \textbf{default}\ (\textbf{within}\ \mathcal{C}\ p^*)\ (\lambda s.error) \\
p^* & = & \langle [\![\textbf{while}\ E\ C]\!], r, b, c \rangle
\end{array}
$$

O nó de árvore de sintaxe abstrata correspondente à transformação *include_break_a* da Seção 4.3.2 (página 62) é igual a $t_a = t'_a; t''_a$, onde:

$$
\begin{array}{rcl}
t'_a & = & \textbf{include}\ b\ q\ \langle x \rangle \\
q & = & \textbf{inclusion}\ \mathcal{C}\ \langle Com, Env \rangle\ Cc\ (Cc \rightarrow Cc) \\
x & = & \textbf{default anywhere}\ (\lambda s.error)
\end{array}
$$

$$
\begin{array}{rcl}
t''_a & = & \textbf{replace}\ \mathcal{C}\ p^*\ \textbf{anywhere}\ e \\
p^* & = & \langle [\![\textbf{while}\ E\ C]\!], r, b, c \rangle \\
e & = & \mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r\ c\ c
\end{array}
$$

A árvore de sintaxe abstrata correspondente à transformação *include_break_c* da Seção 4.3.2 (página 4.3.2) é igual a $t_c = \textbf{replace}\ \mathcal{C}\ p^*\ w\ e$, onde:

$$\begin{array}{lcl} p^* & = & \langle [\![C]\!], r, c\rangle \\ w & = & \textbf{within}\ \mathcal{C}\ p^{*\prime} \\ p^{*\prime} & = & \langle [\![\textbf{while}\ E\ C']\!], r', c'\rangle \\ e & = & \mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r[c'/\mathsf{break}]\ c \end{array}$$

A árvore de sintaxe abstrata correspondente à transformação *include_break_d* da Seção 4.3.3 (página 4.3.3) é igual a $t_d = \textbf{redefine}\ \mathcal{C}\ p^*\ e$, onde:

$$\begin{array}{lcl} p^* & = & \langle [\![\textbf{while}\ E\ C]\!], r\rangle \\ e & = & \textsc{fix}\ \lambda f c.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r[c/\mathsf{break}]\ (f\ c)\ \textbf{else}\ c \end{array}$$

4.4.2 Domínios Semânticos

A denotação de uma especificação é um valor $\phi \in \Phi = \mathit{Str} \to \mathit{Str} \to \mathit{Str}$, isto é, uma função cujos argumentos são uma cadeia de caracteres representando um programa na linguagem especificada e uma cadeia de caracteres representando a sequência de entrada, e produz uma cadeia de caracteres representando a saída produzida.

Transformações são funções que operam sobre a estrutura de uma especificação, de modo que antes de se produzir um valor $\phi \in \Phi$, elas são denotadas por um valor $\sigma \in \Sigma$. Assim, internamente, uma especificação é representada por meio da tupla $\sigma = (g_c, g_a, d^*, \rho, \mathit{main}) \in \Sigma$, onde $g_c : G$ representa a gramática concreta da especificação, $g_a : G$, a sua gramática abstrata, $d^* : D^*$ uma lista dos domínios sintáticos e semânticos, $\rho : \mathit{Env}$ um ambiente que mapeia nomes de função em listas de cláusulas de definição, isto é, nas equações semânticas correspondentes, e $\mathit{main} : I$ é o nome da função semântica que produz a denotação de programas na linguagem especificada. Se P for o domínio sintático de programas na linguagem especificada, então o domínio da função *main* deverá ser $P \to \mathit{Str} \to \mathit{Str}$.

O domínio *Env* é definido por $\mathit{Env} = I \to F^*$, onde $F = P^* \times E$ representa o domínio das cláusulas de função, compostas por uma lista de padrões, representando seus argumentos declarados, e uma expressão, representando o seu corpo. Em todas as funções da especificação é acrescentado um argumento $\rho : \mathit{Env}$, para permitir a amarração dinâmica de funções. Com isso, toda ocorrência do identificador de função f no corpo de uma das cláusulas da função g é substituída pela expressão $((\mathit{retrieve\text{-}f}\ \rho)\ \rho)$, onde ρ corresponde ao argumento inserido na cláusula correspondente da função g, e *retrieve-f* é uma função

que recupera a definição de f do ambiente ρ e monta uma função que possa ser utilizada na aplicação correspondente. Para cada função $f_i : T$ da especificação, será criada uma função especial *retrieve-*f_i : $Env \rightarrow Env \rightarrow T$. Este artifício tem por objetivo tornar mais simples as definições dos transformadores, uma vez que várias das transformações executadas consistirão simplesmente em alterar o ambiente de amarração de identificadores.

Por exemplo, considere uma especificação composta pela função $\mathcal{E} : Exp \rightarrow R \rightarrow Val$, cujas cláusulas são as seguintes:

- $\mathcal{E}[\![Id]\!]\ r = r\ Id$;
- $\mathcal{E}[\![E_1 + E_2]\!]\ r = \mathcal{E}[\![E_1]\!]\ r + \mathcal{E}[\![E_2]\!]\ r$.

O ambiente ρ é definido por $\rho = \{\mathcal{E} \mapsto \{f_1, f_2\}\}$, onde:

- $f_1 = (\langle \rho, [\![Id]\!], r \rangle, r\ Id)$;
- $f_2 = (\langle \rho, [\![E_1 + E_2]\!], r \rangle, ((retrieve\text{-}\mathcal{E}\ \rho)\ \rho)\ [\![E_1]\!]\ r + ((retrieve\text{-}\mathcal{E}\ \rho)\ \rho)\ [\![E_2]\!]\ r)$

Além desses domínios, definem-se ainda os domínios $\Omega = \{anywhere\} + I \times P^*$, que representa uma informação de contexto para cláusulas **default** e **replace**, e $\Xi = \Omega \times E$, que representa uma cláusula **default** com expressão do domínio E a ser incluída em um contexto representado por um valor do domínio Ω.

4.4.3 Funções Semânticas

A especificação do modelo consiste nas seguintes funções semânticas:

- $run : S \rightarrow \Phi$
  $\phi = run\ [\![s]\!]$ é a denotação da especificação s;
- $\mathcal{S} : S \rightarrow \Sigma$
  $\sigma = \mathcal{S}[\![s]\!]$ é a representação interna de uma especificação, possivelmente alvo de transformação;
- $\mathcal{T} : T \rightarrow Env \rightarrow Env$
  $\rho' = \mathcal{T}[\![t]\!]\ \rho$ é o ambiente resultante da aplicação da transformação t ao ambiente ρ;

- $\mathcal{M} : M^* \to \Sigma \to \Sigma$
  $\sigma' = \mathcal{M}[\![m^*]\!]\ \sigma$ é a especificação resultante da inclusão dos módulos na lista m^* na especificação σ;

- $\mathcal{E} : E \to E$
  $e' = \mathcal{E}[\![e]\!]$ é uma expressão equivalente a e em que as ocorrências dos identificadores de função são substituídas pela expressão de pesquisa no ambiente dinâmico;

- $\mathcal{P} : P^* \to P^*$
  $p^{*\prime} = \mathcal{P}[\![p^*]\!]$ é uma lista de padrões equivalentes a p^* em que as adaptações pertinentes são realizadas;

- $\mathcal{Q} : Q \to \Gamma$
  $\gamma = \mathcal{Q}[\![q]\!]$ é uma função que mapeia cada nome de função f em que houve inclusão de argumento em um trio $\gamma\ f = ((d_1, \cdots, d_n), d, d')$, indicando que a função f cujo tipo era $d_1 \to \cdots \to d_n \to d'$ passará a ter tipo $d_1 \to \cdots \to d_n \to d \to d'$;

- $\mathcal{W} : W \to \Omega$
  $\omega = \mathcal{W}[\![w]\!]$ denota a informação de contexto corresponde a w;

- $\mathcal{X} : X \to \Xi$, $\mathcal{X}^* : X^* \to \Xi^*$
  $\chi = \mathcal{X}[\![x]\!]$ denota a cláusula *default* x, e $\chi^* = \mathcal{X}^*[\![x^*]\!]$ denota a lista de cláusulas *default* x^*.

4.4.4 Equações Semânticas

Execução da Especificação

A execução de uma especificação s consiste em aplicar a função semântica que gera a denotação de programas ao programa fonte e à sequência de entrada. A equação semântica correspondente é dada por:

$$\mathit{run}\ [\![s]\!]\ \mathit{str}_1\ \mathit{str}_2 = \mathit{apply\text{-}main}\ \sigma\ (\mathit{parse}\ \sigma\ \mathit{str}_1)\ \mathit{str}_2\ \textbf{where}\ \sigma = \mathcal{S}[\![s]\!]$$

onde as funções auxiliares *parse* e *apply-main* são responsáveis respectivamente por gerar a árvore de sintaxe abstrata correspondente ao programa fonte e aplicar a função semântica de programas à árvore gerada.

Geração da Representação Interna da Especificação

Dada uma especificação s, a função semântica $\mathcal{S}$ tem por objetivo agrupar as definições presentes nos módulos e, no caso de extensões, aplicar a função de transformação, produzindo como resultado uma representação interna da especificação.

No caso de especificação inicial, deve-se compilar todos os módulos da especificação, incluindo os itens definidos em uma especificação σ_0 inicialmente vazia:

$$\mathcal{S}\ [\![\textbf{specification}\ m^*]\!] = \mathcal{M}[\![m^*]\!]\ \sigma_0$$

No caso de extensão de uma especificação, deve-se inicialmente compilar a especificação original, em seguida aplicar o transformador correspondente produzindo a especificação adaptada e, por fim, processar todos os módulos da especificação, incluindo os novos elementos definidos:

$$\mathcal{S}\ [\![\textbf{extension}\ t\ s\ m^*]\!] = \textbf{let}\ (g_c, g_a, d^*, \rho) = \mathcal{S}[\![s]\!]\ \textbf{in}\ \mathcal{M}[\![m^*]\!]\ (g_c, g_a, d^*, \mathcal{T}[\![t]\!]\ \rho)$$

Aplicação de Transformações em Sequência

Transformações são denotadas por funções que mapeiam especificações em especificações e são produzidas pela função $\mathcal{T}$.

Sequências de transformações são processadas na ordem em que são definidas, de modo que a segunda transforma a especificação resultante da primeira transformação:

$$\mathcal{T}[\![T_1; T_2]\!] = \mathcal{T}[\![T_2]\!]\ \circ\ \mathcal{T}[\![T_1]\!]$$

Aplicação de Transformações de Inclusão de Argumentos

A inclusão de argumento com rótulo i, com valor default dado pela expressão e e com cláusulas de inclusão q, tal como definido na Seção 4.3.1, produz um novo ambiente de amarração de identificadores de função a partir de um ambiente existente, sendo definida

por meio da seguinte equação[4]:

$$\begin{array}{l}\mathcal{T}[\![\textbf{include } i\ q\ x^*]\!]\ \rho = \rho_1 \cup \rho_2 \\ \quad \textbf{where} \quad \rho_1 = \{(f_i \mapsto \textsc{map}\ (\mathit{include}\ i\ \gamma)\ f^*) \mid f^* = \rho\ f_i, f_i \text{ bound in } \gamma\} \\ \qquad\quad \rho_2 = \{(f_i \mapsto \textsc{map}\ (\mathit{include}'\ f_i\ \gamma\ \chi^*)\ f^*) \mid f^* = \rho\ f_i, f_i \text{ unbound in } \gamma\} \\ \qquad\quad \gamma = \mathcal{Q}[\![q]\!] \\ \qquad\quad \chi^* = \mathcal{X}^*[\![x^*]\!]\end{array}$$

A função *include* é responsável pela inclusão do novo argumento em funções promovidas e é definida por:

$$\mathit{include}\ i\ \gamma\ (\rho : p_1^*, e) = (\rho : p_2^*, e[\rho'/\rho]),$$

onde p_2^* corresponde à inclusão de um novo padrão identificador i na posição esperada em p_1^*, e ρ' é o ambiente no qual o argumento i é propagado para toda aplicação de função amarrada em γ.

A função $\mathit{include}'$ ajusta o corpo de funções não-promovidas e é definida por:

$$\mathit{include}'\ f_i\ \gamma\ \chi^*\ (\rho : p^*, e) = (\rho : p^*, e[\rho'/\rho]),$$

onde ρ' é o ambiente no qual o valor default e' é inserido em toda aplicação de função amarrada em γ, nos casos em que $\chi = (p^{*\prime}, e')$ é uma informação de contexto em χ^*, tal que $p^{*\prime}$ é a generalização mais estrita[5] de p^*.

Aplicação de Transformações de Decoração

A decoração da função de nome i com lista de padrões p^*, contexto de aplicação w e expressão de substituição e consiste na criação de um novo ambiente no qual a função é substituída por uma versão em que todas as chamadas da função são decoradas, tal como definido na Seção 4.3.2, caso os padrões que designem a equação sejam aplicáveis:

$$\begin{array}{l}\mathcal{T}[\![\textbf{replace } i\ p^*\ w\ e]\!]\ \rho = \rho' \\ \quad \textbf{where} \quad \rho' = \{(i' \mapsto \textsc{map}\ \mathit{dec}\ f^*) \mid (i' \mapsto f^*) \in \rho\} \\ \qquad\quad \mathit{dec} = \mathit{decorate}\ i\ \mathcal{P}[\![p^*]\!]\ \mathcal{W}[\![w]\!]\ \mathcal{E}[\![e]\!]\end{array}$$

[4] A função MAP f l aplica a função f a cada elemento na l, produzindo uma lista composta pelos resultados.

[5] Um padrão p é uma generalização do padrão p' se todas as expressões que casarem com p' também casarem com p. Um padrão p é uma generalização mais estrita de p' se p for generalização de p' e não houver padrão p'' tal que p é generalização de p'' e p'' é generalização de p'. Ver definição por casos na Seção 5.4.2.

A função *decorate* toma como argumentos os dados de uma decoração e uma cláusula de função da especificação e faz a substituição da função decorada, caso a cláusula seja compatível com o contexto designado por ω, e é definida por:

$$decorate\ i\ p^*\ \omega\ e\ f = f',$$

onde f' é definida de acordo com os seguintes casos:

1. se $\omega = anywhere$, então $f' = replace\ i\ p^*\ e\ f$;
2. se $\omega = (i, p^{*\prime\prime})$, $f = (\rho' : p^{*\prime}, e')$ e $p^{*\prime\prime}$ for generalização de $p^{*\prime}$, então a nova função é dada por $f' = replace\ i\ p^*\ e''\ f$, onde $e'' = e[\rho^{*\prime}/\rho^{*\prime\prime}]$ corresponde à expressão e na qual os identificadores de contexto são amarrados aos valores correspondentes dos parâmetros formais da cláusula decorada;
3. se as condições 1 e 2 não puderem ser aplicadas, então a cláusula não deve ser decorada, e, portanto, $f' = f$.

A função *replace* tem como argumentos o nome da função decorada i, o padrão aplicável à decoração p^*, a expressão que substitui a aplicação e, e a cláusula onde a substituição deve ocorrer $(\rho : p^{*\prime}, e')$ e gera uma nova cláusula $(\rho'' : p^{*\prime}, e')$. Nesta nova cláusula, o ambiente ρ'' mapeia a função decorada i na expressão e, amarrando os identificadores em p^* aos parâmetros formais originais em $p^{*\prime}$ e o identificador i à versão antiga da função, conforme amarração feita no ambiente original ρ. Para que isso seja possível, p^* deve ser generalização de $p^{*\prime}$.

Aplicação de Transformações de Redefinição

A redefinição de uma equação referente a uma função de nome i com lista de padrões p^* e expressão de substituição e, tal como definido na Seção 4.3.3, consiste na criação de um novo ambiente no qual a cláusula correspondente da função é substituída pela nova cláusula:

$$\mathcal{T}[\![\textbf{redefine}\ i\ p^*\ e]\!]\ \rho = \rho[(\textsc{map}\ (redefine\ i\ \mathcal{P}[\![p^*]\!]\ \mathcal{E}[\![e]\!])\ (\rho\ i))/i]$$

A function *redefine* recebe como argumentos os dados da redefinição e uma cláusula da função redefinida, produzindo um novo ambiente no qual a cláusula especificada é substituída. Esta função é definida por[6]:

$$redefine\ i\ (p_1, \cdots, p_m)\ e\ ((p'_1, \cdots, p'_n), e') = f',$$

[6]Compare com a definição de *replace*, que aplica a versão original da função.

onde se $(p_1, \cdots, p_m)$ for uma generalização de $(p'_1, \cdots, p'_m)$, então:

$$\begin{aligned} f' &= ([p_1, \cdots, p_m, p'_{m+1}, \cdots, p'_n], e''\ p'_{m+1} \cdots p'_n), \\ e'' &= e[p'_1/p_1, \cdots, p'_m/p_m], \end{aligned}$$

ou

$$f' = ((p'_1, \cdots, p'_n), e'),$$

caso contrário.

O efeito da função *redefine* é substituir toda ocorrência da função de nome i no ambiente por sua nova versão, em que toda vez que os argumentos da aplicação casarem com $p_1, \cdots, p_m$, a nova expressão substitui a aplicação original da função. Se os padrões não casarem, então a versão original da função é utilizada. Vale ressaltar que todas as ocorrências livres do nome da função em e são localizadas no ambiente corrente, e, por esta razão, qualquer aplicação da função se refere à sua nova definição.

4.5 Estudos de Caso

Para validação, a metodologia foi aplicada na definição incremental da parte sequencial de Java (*Crepalde et al.* 2008a). Esta definição é composta por três partes:

- **Java 0**: o programa é composto por uma única classe e contém os tipos primitivos inteiro e lógico, comandos básicos da linguagem, expressões e funções estáticas;
- **Java 1**: estende a especificação inicial incluindo todos os tipos primitivos, arranjos e sequenciadores;
- **Java 2**: estende a especificação anterior permitindo a definição de instâncias de classes, métodos dinâmicos, arranjos de objetos e alocação dinâmica de objetos.

O Apêndice B apresenta a definição de linguagens utilizando os recursos da metodologia proposta.

Este capítulo apresenta os alguns estudos de caso simples para compreensão dos recursos propostos:

1. definição incremental de métodos e tratamento de exceções em linguagem orientada por objetos típica;

2. definição incremental de procedimentos e adendos em linguagem orientada por aspectos.

4.5.1 Métodos e Tratamento de Exceções

Esta seção apresenta uma solução para o problema da separação da definição de métodos e tratamento de exceções em uma linguagem orientada por objetos típica, cuja solução entrelaçada é apresentada na Seção 4.1, página 56.

Considere inicialmente a definição semântica da Figura 4.1, página 59. Para a inclusão de métodos, será criada uma nova continuação de retorno associada a todos os comandos e propagada por meio do *environment*. Esta continuação será definida no início da execução de corpo de método e será propagada recursivamente ao longo das aplicações da função de denotação de comandos. Uma continuação de erro será inserida em todas as chamadas de $\mathcal{C}$ que não se originarem da execução de métodos. As construções relativas a métodos podem ser definidas por meio das seguintes equações:

- $\mathcal{D}[\![I_1\ (I_2)\ \{\ C\ \}]\!]r\ u = u\ r[(f, r)/I_1]$
  where $f = \text{FIX}\ (\lambda f r'.\mathit{ref};\ \lambda l.\mathcal{C}[\![C]\!]r'[(f, r), l/I_1, I_2]; r'\ \mathsf{return}\ \mathit{novalue})$
- $\mathcal{C}[\![\textbf{return}\ E]\!]r\ c = \mathcal{E}[\![E]\!]r; r\ \mathsf{return}$
- $\mathcal{E}[\![E_1(E_2)]\!]r\ k = \mathcal{E}[\![E_1]\!]r; \lambda(f, r').\mathcal{E}[\![E_2]\!]r; \mathit{apply}\ (f, r'[k/\mathsf{return}])$

onde a função *apply* é definida por $\mathit{apply}\ (f, r) = f\ r$.

Para a inclusão de tratamento de exceções, considera-se que uma nova continuação para o sequenciador de lançamento de exceções será incluída nas funções semânticas de expressões e comandos. Para isso, define-se a transformação *include_exceptions* que realiza a propagação da nova continuação também por meio do *environment*:

transformation *include_exceptions*
replace $\mathit{apply}\ (f, r')$ **within** $\mathcal{E}[\![E_1(E_2)]\!]\ r$
by $\mathit{apply}\ (f, r'[r\ \mathsf{throw}/\mathsf{throw}])$

A decoração especificada recupera do *environment* da chamada de função o valor da continuação de exceção e a propaga via *environment* para a execução do corpo da função.

Ao se comparar esta solução com a solução apresentada anteriormente na Seção 4.1, página 56, notam-se as seguintes melhorias:

- as equações que apresentam as construções relacionadas a chamadas de métodos estão completamente independentes da existência de tratamento de exceções;
- está explícito nas equações o efeito da inclusão de tratamento de exceções sobre as construções existentes, no caso, as relacionadas a métodos.

As equações que definem as construções relativas ao mecanismo de tratamento de exceções permanecem como anteriormente definidas.

4.5.2 Definição de Procedimentos e Adendos

Adendos e pontos de junção dinâmicos da programação orientada por aspectos (*Kiczales et al.* 1997) são formalmente definidos por *Wand et al.* (2004), que apresentam uma especificação da semântica denotacional monádica para uma linguagem simples semelhante a Scheme, composta por procedimentos, adendos e descritores de conjuntos de junção. As equações semânticas apresentadas naquela especificação contêm elementos entrelaçados, o que dificulta a compreensão dos conceitos chaves da definição.

Este estudo de caso é uma simplificação daquela formalização, por meio da aplicação dos mecanismos apresentados para especificações incrementais. Esta versão não considera as cláusulas *within* e *proceed*, que podem ser incorporadas por meio de decoradores.

A Figura 4.2 apresenta os principais elementos da especificação semântica, a saber seus domínios e a mônada para execução, que foram tomadas *ipsis litteris* do artigo original (*Wand et al.* 2004). Aquele trabalho apresenta ainda detalhes a respeito da álgebra de pontos de junção, funções auxiliares e operações monádicas, recomendado para leitores que se interessarem em entender a formalização com maior profundidade.

Definição de Procedimentos. Procedimentos são o ponto de partida da definição. A denotação da declaração de procedimentos é resultado da função $\mathcal{P}$, que tem por objetivo criar um ambiente de procedimentos que associe nome de procedimento à semântica correspondente:

$$
\begin{aligned}
&\mathcal{P} : \mathit{Procedure} \to \mathit{PE} \to \mathit{PE} \\
&\mathcal{P}[\![(\textbf{procedure } \mathit{pname}\ (x_1, \cdots, x_n)\ e)]\!]\ \phi = [\mathit{proc}/\mathit{pname}] \\
&\text{where } \mathit{proc} = \lambda(v_1, \cdots, v_n).\ \textbf{let } l_1 \Leftarrow \mathit{alloc}\ v_1; \cdots; l_n \Leftarrow \mathit{alloc}\ v_n \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } \ \mathcal{E}[\![e]\!]\ [l_1/x_1, \cdots, l_n/x_n]\ \phi
\end{aligned}
$$

Sets:

v	$\in$	Val	Expressed values
l	$\in$	Loc	Locations
s	$\in$	Sto	Stores
id	$\in$	Id	Identifiers
$pname$, $wname$	$\in$	$Pname$	Procedure names
v	$\in$	Val	Expressed values

Join points, pointcut designators:

jp	$\in$	JP
jp	$\rightarrow$	$\langle\rangle \mid \langle k, pname, wname, v^*, jp\rangle$
k	$\rightarrow$	pcall | pexecution | aexecution
pcd	$\rightarrow$	$\cdots$

Execution monad:

$$T(A) = JP \times Sto \rightarrow (A \times Sto)_\perp$$

Semantic Domains:

π	$\in$	$Proc = Val^* \rightarrow T(Val)$	Procedures
α	$\in$	$Adv = JP \rightarrow Proc \rightarrow Proc$	Advices
ϕ	$\in$	$PE = Pname \rightarrow Proc$	Procedure environments
γ	$\in$	$AE = Adv^*$	Advice environments
ρ	$\in$	$Env = [Id \rightarrow Loc]$	Environments

Figura 4.2: Definições Básicas para a Semântica de Adendos e Pontos de Junção da Programação Orientada por Aspectos – Extraída de *Wand et al.* (2004).

Chamadas de procedimento são definidas por meio da função $\mathcal{E}$, que avalia os argumentos e aplica a denotação de procedimento amarrada no environment corrente:

$$\begin{aligned}
&\mathcal{E} : Exp \rightarrow Env \rightarrow PE \rightarrow T(Val)\\
&\mathcal{E}[\![(pname\ e_1\ \cdots\ e_n)]\!]\ \rho\ \phi = \ \textbf{let}\ v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\ \rho\ \phi; \cdots ; v_n \Leftarrow \mathcal{E}[\![e_n]\!]\ \rho\ \phi\\
&\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \phi\ pname\ (v_1, \cdots, v_n)
\end{aligned}$$

Definição de Adendos. O primeiro passo para a inclusão dos recursos da orientação por aspectos consiste em adaptar as equações apresentadas para mostrar como procedimentos dependem de ambientes de adendos; esta adaptação é feita por meio da transformação

include-advices:

transformation *include-advices*
signature $\mathcal{E} : Exp \to Env \to PE \to \gamma : AE \to T(Val)$,
$\mathcal{P} : Procedure \to PE \to PE$ **to** $Procedure \to PE \to (\gamma : AE) \to PE$
default $\gamma = []$
replace $\underline{\mathcal{P}[\![(\textbf{procedure } pname\ (x_1, \cdots, x_n)\ e)]\!]\ \phi\ \gamma}$
by $[(enter\text{-}jp\ \gamma\ (new\text{-}pexecution\ pname)\ proc)/pname]$
where $proc = \underline{\mathcal{P}[\![(\textbf{procedure } pname\ (x_1, \cdots, x_n)\ e)]\!]\ \phi\ \gamma}$,
$\underline{\mathcal{E}[\![(pname\ e_1\ \cdots\ e_n)]\!]\ \rho\ \phi\ \gamma}$
by $\underline{\mathcal{E}[\![(pname\ e_1\ \cdots\ e_n)]\!]\ \rho\ (\phi[proc/pname])\ \gamma}$
where $proc = \lambda v^*.enter\text{-}jp\ \gamma\ (new\text{-}pcall\ pname\ v^*)\ (\phi\ pname)$

Esta transformação: (a) inclui o ambiente de adendos como argumento das funções $\mathcal{E}$ e $\mathcal{P}$ por meio de uma cláusula de alteração de assinatura; este ambiente é propagado por meio das chamadas recursivas da função $\mathcal{P}$, e, na falta de um ambiente para ser propagado, um ambiente vazio é utilizado como valor default; (b) decora o ambiente de procedimentos para costurar a execução de adendos, por meio da primeira cláusula **replace**, que decora o resultado da função $\mathcal{P}$ com marcas de pontos de junção; e (c) decora a execução de expressões de chamada de procedimentos para costurar adendos de execução, por meio da segunda cláusula **replace**, que decora o argumento da função $\mathcal{E}$ correspondente ao ambiente de procedimento com marcas específicas de pontos de junção.

A semântica de adendos é resultado da função $\mathcal{A}$, que executa o adendo e o corpo do procedimento na ordem correta, caso o ponto de junção correspondente seja aplicável ao procedimento; caso contrário, apenas o corpo do procedimento é executado.

$$\mathcal{A} : Advice \to PE \to AE \to JP \to Proc \to Proc$$
$$\mathcal{A}[\![(\textbf{before } pcd\ e)]\!]\ \phi\ \gamma\ jp\ \pi =$$
$$\lambda v^*.\mathcal{PCD}[\![pcd]\!]\ jp\ (\lambda\rho.\textbf{let } v_1 \Leftarrow \mathcal{E}[\![e]\!]\ \rho\ \phi\ \gamma; v_2 \Leftarrow \pi\ v^* \textbf{ in } v_2)\ (\pi\ v^*)$$
$$\mathcal{A}[\![(\textbf{after } pcd\ e)]\!]\ \phi\ \gamma\ jp\ \pi =$$
$$\lambda v^*.\mathcal{PCD}[\![pcd]\!]\ jp\ (\lambda\rho.\textbf{let } v_1 \Leftarrow \pi\ v^*; v_2 \Leftarrow \mathcal{E}[\![e]\!]\ \rho\ \phi\ \gamma \textbf{ in } v_1)\ (\pi\ v^*)$$

A aplicação das técnicas de definição incremental permite a definição vaga de procedimentos, que pode ser estendida, por meio de funções de transformação, para suportar posteriormente adendos da orientação por aspectos.

4.6 Conclusões

Este capítulo apresentou a técnica de Semântica Incremental, que tem por objetivo permitir a definição incremental da semântica denotacional de uma linguagem. Nesta técnica, define-se uma versão inicial da linguagem, contendo seus elementos mais relevantes. A partir desta definição inicial, pode-se definir uma sequência de especificações, em que cada uma estende a anterior. Para que a extensão seja possível, uma especificação pode aplicar uma função de transformação, que define como adaptar as denotações existentes. Desta maneira, a denotação das construções pode evoluir ao longo das especificações intermediárias, até a sua versão final.

Esta técnica se baseia no uso de vagueza de definições iniciais, um conceito que é muito comum nas descrições informais. Geralmente, manuais de linguagem e materiais didáticos tendem, nos capítulos iniciais, a explicar a estrutura geral da linguagem e suas construções mais básicas; conforme novas construções são explicadas, novas interpretações podem ser dadas ao que já foi discutido. O uso de descrições vagas permite ao projetista apresentar a linguagem em etapas, aumentando o grau de detalhamento ao longo do tempo.

Durante a escolha das possíveis transformações, foi dada ênfase a casos mais comuns de transformações de equações semânticas em definições denotacionais. Assim, não houve o intuito neste trabalho em buscar um conjunto completo de transformadores que pudessem lidar com todos os casos possíveis. No entanto, as primitivas de transformação disponíveis tratam os casos mais comuns, permitindo manipular equações semânticas sem quebrar a abstração do modelo denotacional.

Com relação aos recursos usuais da orientação por aspectos, optou-se neste trabalho por definir um ambiente controlado para os transformadores que não comprometesse o raciocínio modular.

O próximo capítulo apresenta a linguagem de especificação Notus, que implementa as técnicas definidas neste capítulo. A linguagem Notus possui recursos para a definição incremental de linguagens de programação, sendo possível especificar sintaxe e semântica. O compilador da linguagem Notus gera interpretadores executáveis para a linguagem definida, permitindo acelerar o processo de prototipação da linguagem.

Capítulo 5

Ambiente Para Semântica Incremental

Este capítulo tem por objetivo apresentar os recursos mais importantes da linguagem Notus, que é o elemento central de um ambiente de desenvolvimento para formulação de semântica baseada na metodologia descrita no Capítulo 4.

A linguagem Notus é uma evolução da linguagem $\mathcal{SCRIPT}$ (*Costa* 2000, *Maia* 1994, *Oliveira* 1998) possui recursos para a definição incremental completa de uma linguagem de programação, incluindo sintaxe e semântica. A sintaxe da linguagem Notus pode ser encontrada no Apêndice A. Uma apresentação do uso da linguagem e de seu compilador é feita por *Crepalde et al.* (2008b) no relatório técnico do *Manual do Usuário de Notus*. Um compilador completo para a linguagem foi desenvolvido inicialmente por *Soares* (2007) e estendido por *Loredo et al.* (2008). O autor desta tese participou de toda a concepção e projeto do compilador.

Optou-se neste trabalho por desenvolver uma nova linguagem, em vez de estender uma linguagem funcional já existente, como Haskell. Esta decisão deveu-se principalmente à necessidade de se definir um conjunto de construções mais próximo das equações encontradas na literatura da área, permitindo a escrita de código mais legível.

Organização do Capítulo. A Seção 5.1 apresenta os mecanismos para divisão de uma definição incremental em uma sequência de especificações, e cada uma destas em pacotes e módulos. A Seção 5.2 mostra os mecanismos de definição de domínios sintáticos e semânticos em definições de semântica incremental. Na Seção 5.3, são discutidos os re-

cursos para definição da sintaxe de linguagem de programação, da qual é possível derivar sistema de pré-processamento de arquivo fonte, analisador léxico e analisador sintático. Na Seção 5.4, apresentam-se os principais recursos para definição de funções semânticas, a partir das quais pode-se gerar um interpretador da linguagem especificada. A Seção 5.5 apresenta os recursos linguísticos para definição de transformadores de especificação. A Seçao 5.6 discute a criação de componentes de linguagens em Notus. A Seção 5.7 apresenta a estrutura do compilador e discute as principais dificuldades da implementação. E, finalmente, a Seção 5.8 apresenta as conclusões do capítulo.

5.1 Organização Modular

A definição incremental de uma linguagem é organizada em um conjunto de pastas, cada uma referente a uma especificação, todas dentro de uma mesma pasta raiz. Considere por exemplo que a definição incremental de uma linguagem L seja feita por meio de três especificações, L_0, L_1, L_2. Os módulos que compõem estas especificações devem então estar presentes em subpastas `L0`, `L1` e `L2`, dentro da pasta raiz. Cada especificação L_i pode ser dividida em módulos organizados em uma estrutura recursiva de pacotes com controle de visibilidade semelhante à utilizada nas linguagens Haskell e Java, conforme mostrado na Figura 5.1.

5.1.1 Estrutura de Pacotes

Dentro de uma especificação, módulos podem ser organizados em pacotes. Um módulo M pertence a um pacote P se:

- o cabeçalho do módulo M for: `module P.M`;
- o arquivo *M.nts* estiver na pasta P.

Pacotes podem ser estruturados de maneira recursiva, de modo que um pacote pode conter sub-pacotes. Por exemplo, existe uma sequência de pacotes P_1, P_2, ..., P_n, tal que P_i é sub-pacote de P_{i-1}, para $2 \leq i \leq n$, e o módulo M pertence ao pacote P_n se:

- o cabeçalho do módule M for: `module` $\texttt{P}_1\texttt{.P}_2\texttt{.} \cdots \texttt{.P}_n\texttt{.M}$;

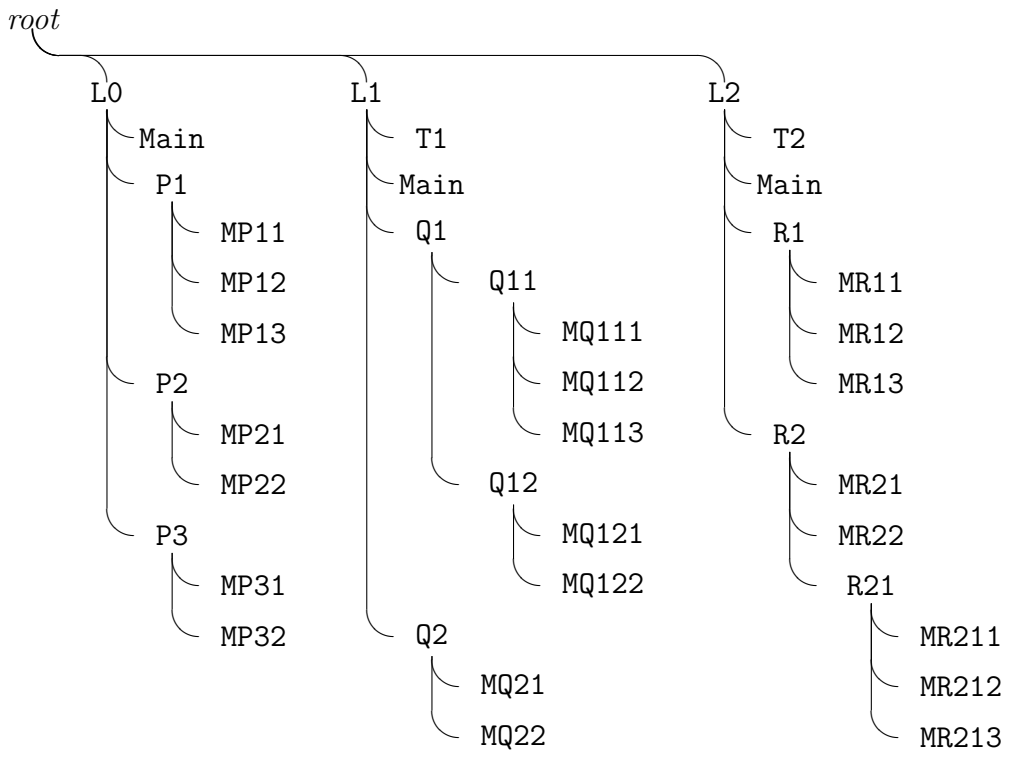

Figura 5.1: Exemplo de Organização Modular de Definição Incremental de uma Linguagem Hipotética L.

- existir uma sequência de pastas aninhadas P_1, P_2, ..., P_n, tal que a pasta P_i pertence à pasta P_{i-1}, para $2 \leq i \leq n$;
- o arquivo[1] *M.nts* estiver na pasta P_n.

5.1.2 Módulos e Visibilidade

Módulo é o principal mecanismo de encapsulação das descrições de recursos linguísticos e pode ser composto pelos seguintes constituintes:

1. seção de importações: enumera os módulos dos quais o módulo depende;
2. definições léxicas: definem ou estendem elementos léxicos da linguagem, que podem ser tokens ou macros para definição de tokens;

[1] A extensão de módulos em notus é *nts*.

```
1 module Pa.Pb.Pc.M
2     ...
3     public token x = "x"
4     private token y = "y"
5     token w = "w"
6     ...
```

Listagem 5.1: Exemplo do Uso de Visibilidade em Notus

3. definições sintáticas: definem ou estendem domínios sintáticos e definem novas regras de produção das gramáticas concreta e abstrata;

4. definições de domínios: definem ou estendem domínios semânticos da especificação;

5. declarações de funções: introduzem novas funções à especificação por meio de definição de assinatura;

6. definições de função: definem corpo de função.

Os componentes de um módulo podem ser definidos em qualquer ordem, desde que a seção de importações anteceda todas as demais seções. Cada identificador é definido com sua visibilidade, que pode ser: pública, se for utilizada a palavra chave **public**; privada ao módulo, se for utilizada a palavra chave **private**; ou pública somente para o pacote a que o módulo pertence, caso a visibilidade não seja explicitada pelo projetista.

Por exemplo, na Listagem 5.1: o token x é definido como público, podendo assim ser acessado por todos os módulos da especificação; por outro lado, o token y é privado podendo ser utilizado somente dentro do módulo M; finalmente, o token w possui visibilidade de pacote, podendo assim ser acessado somente por módulos pertencentes ao pacote $Pa.Pb.Pc$.

5.1.3 Regras de Importação de Identificadores

A seção de importação enumera módulos dos quais um módulo depende. Se um módulo M_2 importar um módulo M_1, então todos os componentes públicos definidos em M_1 podem ser utilizados em M_2 sem qualificação. Componentes privados definidos em M_1 não são acessíveis de maneira alguma a partir de M_2. Além disso, se M_1 e M_2 pertencerem a um mesmo pacote, então todos os identificadores de M_1 com visibilidade de pacote também são acessíveis a partir de M_2 sem a necessidade de qualificação.

```
1  module M
2      import A
3      function g : Int -> Int
4      g x = f x + B.f x // Correct
5       ...
6  end
7  module N
8      import A, B
9      function h : Int -> Int
10     h x = f x // Incorrect. Should have used either A.f x or B.f x
11      ...
12 end
```

Listagem 5.2: Exemplo de Resolução de Colisão de Nomes

Um componente visível a um módulo pode também ser referenciado por meio da qualificação completa de seu identificador com o nome do módulo. Por exemplo, um componente de nome x definido no módulo M_1 pode ser referenciado por $M_1.x$. Neste caso, não é necessário importar M_1. Tal qualificação é também utilizada para resolver colisões de identificadores importados em um módulo. Por exemplo, suponha que os módulos M_1 e M_2 definam componentes distintos, mas ambos com o nome x, e que um módulo M_3 importe tanto M_1 quanto M_2. Esta colisão deve ser resolvida utilizando-se $M_1.x$ ou $M_2.x$.

Considere os módulos M e N, definidos na Listagem 5.2. Suponha que os módulos A e B definem funções públicas distintas, ambas com nome f, pertencentes ao domínio $Int \rightarrow Int$. Módulo M importa o módulo A, sendo possível utilizar a função f de A sem qualificação; no entanto, para utilizar a função f de B é necessário fazer a qualificação. No módulo N, não é possível distinguir qual função f é utilizada em cada aplicação, uma vez que ambos os módulos A e B foram importados. Neste caso, a qualificação é obrigatória para todas as ocorrências do nome f.

5.1.4 Extensão de Componentes

Durante a escrita de uma especificação incremental, pode ser necessário redefinir algum elemento básico de definição da linguagem. Por exemplo, a inclusão de um novo comando pode exigir a definição de uma nova regra de produção que tenha como lado esquerdo

o não-terminal correspondente a comandos. Tipicamente, as alterações necessárias ao se definir uma nova construção podem ser:

- incluir uma nova expressão regular para definição de um token já existente ou de uma macro para definição de tokens; tais extensões são feitas por meio de cláusulas **extend** associadas a tokens e macros (ver Seção 5.3.3.2);
- incluir uma nova regra de produção na gramática concreta, tendo como lado esquerdo algum não-terminal já definido; tais extensões são feitas por meio de cláusulas **extend** associadas a não terminais da gramática concreta (ver Seção 5.3.4.5);
- incluir uma nova regra de produção na gramática abstrata, tendo como lado esquerdo algum domínio sintático já definido; tais extensões são feitas automaticamente, ao se incluir uma nova regra de produção à gramática concreta, ou por meio de cláusulas **extend** associadas a domínios sintáticos (ver Seção 5.2.4);
- definir relação de tipo e subtipo entre algum domínio semântico já existente e um novo domínio; tais extensões são feitas por meio do uso de cláusulas **extend** associadas a domínios semânticos (ver Seção 5.2.4);
- definir um novo caso para o corpo de uma função, particularmente para associar a uma função semântica já existente a equação semântica que define uma nova construção; tais extensões são feitas por meio da definição de novos casos para função existente (ver Seção 5.4.1.2).

5.1.5 Especificação Inicial

A pasta da especificação inicial, `L0`, possui um módulo principal de nome *Main*, que define os seguintes elementos utilizados para geração do interpretador da linguagem:

- símbolo de partida da gramática concreta, a partir do qual se obtém o tipo do nó raiz da árvore de sintaxe abstrata;
- função de denotação de programas para a linguagem, que deve ser uma função do tipo do nó raiz da árvore de sintaxe abstrata da linguagem;
- função de pré-processamento (opcional), utilizada para processamento do arquivo de programa para o analisador léxico, caso necessário;

```
1 module Main
2     import p1.*, p2.*;
3     syntax prog;
4     semantics p;
5     input stdin, infile ;
6     output stdout, outfile;
7     preproc prep;
8 end
```

Listagem 5.3: Exemplo de Módulo Principal da Linguagem L

- abstrações dos fluxos de entrada e saída (opcionais) do programa interpretado.

Os elementos utilizados no módulo principal podem ser importados de quaisquer módulos definidos nos pacotes da especificação, de modo que todos os elementos públicos possam ser referenciados. Caso necessário, o módulo principal também pode definir funções auxiliares ao processamento.

Um exemplo típico de módulo principal é exibido na Listagem 5.3, que define *prog* como o símbolo de partida da gramática concreta (ver Seção 5.3), p como a função semântica de programas (ver Seção 5.4.1.5), entradas feitas a partir da entrada padrão (*stdin*) e de um arquivo associado ao identificador *infile*, saídas feitas para a saída padrão (*stdout*) e para um arquivo associado ao identificador *outfile* (ver Seção 5.4.1.5), e *prep* como a função de pré-processamento do programa fonte (ver Seção 5.4.1.4).

O compilador Notus gera, a partir da especificação inicial, um programa executável L, ativado por meio da seguinte linha de comando:

$$L\ program.l\ infile{=}input.txt\ outfile{=}output.txt$$

cujo comportamento pode ser abstraído pela equação:

$$(stdout, outfile) = (p \circ parser \circ scanner \circ prep)\ source\ (stdin, infile).$$

Nesta equação:

- *stdin* é uma cadeia de caracteres que representa os dados lidos a partir da entrada padrão, *stdout* é uma cadeia de caracteres que representa o texto a ser escrito na saída padrão;

- *infile* é a cadeia de caracteres lida do arquivo de entrada *input.txt*, conforme especificado na linha de comando de ativação do interpretador; *outfile* é a cadeia de caracteres a ser escrita no arquivo de saída *output.txt*, conforme especificado na linha de comando de ativação do interpretador;

- *source* é a cadeia de caracteres lida do arquivo fonte *program.l*;

- *prep* é a função de pré-processamento, que, caso não tenha sido especificada pelo programador, é a função identidade; esta função gera uma cadeia de caracteres representando o programa fonte pré-processado a partir de uma cadeia de caracteres representando o programa fonte, originalmente lido do arquivo de entrada;

- *scanner* é o analisador léxico do interpretador, que produz uma sequência de tokens para o analisador sintático a partir de uma cadeia de caracteres que represente o programa fonte;

- *parser* é o analisador sintático[2] do interpretador, que produz a árvore de sintaxe abstrata a partir da sequência de tokens gerados pelo analisador léxico; o símbolo de partida é o especificado no comando **syntax** no módulo principal;

- *p* é a função semântica, que produz as sequências de saída a partir da árvore de sintaxe abstrata e das sequências de entrada do interpretador.

A sintaxe de cada um destes elementos do módulo principal pode ser econtrada na Seção A.2.

5.1.6 Extensão de Linguagem

A partir da especificação inicial, L_0, toda especificação L_i ($i > 0$) é definida a partir de L_{i-1} como $L_i = T_i(L_{i-1}) + \Delta L_i$, onde T_i é o nome de um módulo de transformação (ver Seção 5.5) e ΔL_i representa o conjunto de novas construções a serem incluídas na linguagem, por meio da definição de novos módulos. Assim, a pasta correspondente a uma especificação da linguagem deve conter um módulo de extensão e um módulo de definição de transformador.

O módulo de extensão define qual transformador deve ser aplicado, a especificação a ser estendida e os módulos que formam a nova especificação. O arquivo deve ter o nome `Main.nts` e estar localizado na raiz pasta da especificação, como no exemplo do arquivo

[2] A versão atual do compilador de Notus produz parser LALR(1).

```
1 extension T1(L0)
2     contains Q1.Q11.∗, Q1.Q12.∗, Q2.∗
```

Listagem 5.4: Exemplo de Módulo de Extensão `Main.nts` Para a Linguagem L

`Main.nts` na pasta `L1` na Figura 5.1. O módulo de extensão para a especificação L_i possui a forma:

extension $T(L_{i-1})$
contains *modules*

onde T representa um nome de módulo de transformação, L_i representa o nome de uma especificação e *modules* representa a lista dos módulos pertencentes a pacotes da pasta de L_i utilizados na especificação. Por exemplo, o módulo `Main.nts` da especificação da Figura 5.1 pode ser definido como na Listagem 5.4. Este módulo define que a especificação L_1 será obtida por meio da transformação da especificação L_0 utilizando-se o transformador T_1 e com a adição das novas construções definidas nos módulos pertencentes aos pacotes Q_i. O módulo de transformação define uma função de transformação e será definido na Seção 5.5.

5.2 Domínios

5.2.1 Notação Básica

Tipos em Notus são domínios de Scott (*Scott* 1976). Nestes domínios, existe implicitamente um elemento *bottom* ($\perp$), que será omitido em todos os domínios listados no Capítulo. O símbolo de conjunto vazio, $\emptyset$, denota o domínio *flat* "vazio", que por definição é o conjunto $\{\perp\}$. Além disso, as seguintes operações em domínios são bem definidas: união, interseção, complemento, união disjunta, produto cartesiano, função.

O símbolo $\bigcup$ é utilizado para denotar a união disjunta de domínios quando for possível associar um índice a cada domínio da união. Por exemplo, $\bigcup_{i=1}^{n} d_i = d_1 + d_2 + \cdots + d_n$, ao passo que $\bigcup_{s_i \in S} d_i = d_{s_1} + d_{s_2} + \cdots + d_{s_m}$, onde $S = \{s_1, s_2, \cdots, s_m\}$. Cada d_i, $1 \leq i \leq n$, é uma *parcela* da união.

```
1 token fun = ...
2 token var = ...
3 token num : Exp = ...
4 exp ::= ...
5 primary : Exp ::= ...
```

Listagem 5.5: Exemplos de Definição de Domínio Sintático

5.2.2 Domínios Sintáticos

Domínios sintáticos são formados pelos domínios dos terminais e não-terminais da gramática concreta (ver tokens na Seções 5.3.3 e variáveis na Seção 5.3.4). Em Notus, identificadores de domínios sintáticos iniciam com letras maiúsculas e podem ser deduzidos automaticamente a partir de nomes de tokens e variáveis da gramática. Assim, se uma definição de token ou variável não definir o seu domínio, ele pode ser obtido automaticamente capitalizando a primeira letra de seu identificador.

Por exemplo, considere o trecho de módulo da Listagem 5.5, que define os tokens *fun*, *var* e *num*, e as variáveis *exp* e *primary*. Neste exemplo, os domínios *Fun*, *Var* e *Exp* são automaticamente detectados da definição dos tokens *fun* e *var* e da variável *exp*; além disso, o domínio de *num* e *primary* é *Exp* conforme especificado pelas suas cláusulas de declaração.

Elementos de domínios e suas expressões de definição são automaticamente coletadas a partir das definições léxica e sintática da linguagem por meio das cláusulas de definição de tokens (ver Seção 5.3.3.3), extensão de tokens (ver Seção 5.3.3.2), definição de variáveis de gramática concreta (ver Seção 5.3.4.4) e extensão de variáveis de gramática concreta (ver Seção 5.3.4.5).

Domínios sintáticos são visíveis a todos os módulos do pacote onde foram definidos. Para definir domínios sintáticos como públicos ou privados, é necessário utilizar a cláusula de definição de domínio sintático, que possui a forma:

visibility **syntactic** *domain-name* ;

O universo de domínios sintáticos é definido recursivamente pelas seguintes regras:

- $\emptyset$ é o domínio *flat* vazio;
- se D é um domínio padrão, então D é um domínio sintático (ver Seção 5.2.3.2);

```
1 public Value = Int | Bool;
2 public Loc = Int;
3 public State = Loc -> (Value | {unused});
```

Listagem 5.6: Exemplos de Definição de Domínio Semântico

- se D é um domínio de token ou variável, então D é um domínio sintático;
- se D_1 e D_2 são domínios sintáticos, então $D_1 + D_2$ é um domínio sintático que representa a união disjunta de D_1 e D_2;
- se D_1 e D_2 são domínios sintáticos, então $D_1 \times D_2$ é um domínio sintático que representa o produto cartesiano de D_1 e D_2;
- se D é um domínio sintático, então $D*$ é o domínio das listas (possivelmente vazias) cujos elementos pertencem ao domínio D;
- se D é um domínio sintático, então $D+$ é o domínio das listas não-vazias cujos elementos pertencem ao domínio D.

A definição da sintaxe de domínios sintáticos pode ser encontrada na Seção A.3.

5.2.3 Domínios Semânticos

Domínios semânticos em Notus são utilizados na definição do universo dos valores presentes nas definições semânticas e são definidos por meio da cláusula de domínio semântico, cuja forma é:

visibility domain-name = *domain-expression* ;

Por exemplo, o trecho de código da Listagem 5.6 define domínios semânticos *Value*, *Loc* e *State*.

A definição da sintaxe de domínios semânticos pode ser encontrada na Seção A.3.

5.2.3.1 Expressões de Domínio

Domínios semânticos podem ser compostos a partir de domínios básicos por meio de operações de domínio. Os domínios semânticos básicos são os domínios pré-definidos (ver Seção 5.2.3.2, página 90) e enumerações (ver Seção 5.2.3.3, página 91).

precedência mais alta	tuple, built-in, identifier, enumeration
	list
	instantiation
	functional
precedência mais baixa	union

Tabela 5.1: Precedência de Operações de Domínio

Domínios são compostos em Notus por meio das seguintes operações, com precedência especificada na Tabela 5.1:

- definição de domínio de união disjunta (ver Seção 5.2.3.4, página 92);
- definição de domínios de tuplas (ver Seção 5.2.3.5, página 92);
- definição de domínios de lista (ver Seção 5.2.3.6, página 93);
- definição de domínios funcionais (ver Seção 5.2.3.7, página 94);
- definição de domínios de sintaxe abstrata (ver Seção 5.2.3.8).

5.2.3.2 Domínios Pré-definidos

Os seguintes domínios são pré-definidos em Notus:

- `Bool`: valores booleanos **true** e **false**;
- `Byte`, `Short`, `Int`, `Long`: números inteiros com sinal contendo respectivamente 8, 16, 32 e 64 bits;
- `Float`: números de ponto flutuante com 32 bits;
- `Double`: números de ponto flutuante com 64 bits;
- `Character`: caracteres delimitados por aspas simples;
- `String`: cadeias de caracteres delimitadas por aspas, que são semanticamente equivalentes a listas de caracteres.

```
1 public LocationStatus = public {unused, occupied, deleted};
2 public Month = {jan, feb, mar, apr, may, jun,
3                 jul, aug, sep, oct, nov, dec};
4 public Color = private {red, green, blue};
5 Language = {java, haskell, notus};
6 FileExtension = private {java, hs, nts, exe, txt};
7 private Fruit = {apple, orange, strawberry, lemon};
```

Listagem 5.7: Exemplos de Definição de Enumerações

5.2.3.3 Domínios de Enumeração

Enumerações definem conjuntos de constantes simbólicas e possuem a forma:

$$\{\mathtt{id}_1, \mathtt{id}_2, \cdots, \mathtt{id}_n\},$$

onde cada $\mathtt{id}_i$ é um identificador denominado enumerando. A Listagem 5.7 contém exemplos de definição de enumerações.

Os elementos de uma enumeração podem ser públicos, privados, ou visíveis somente para os módulos do mesmo pacote. Enumerandos podem ser definidos como públicos ou privados por meio do uso da palavra chave correspondente antes das chaves que delimitam os elementos. A visibilidade de uma enumeração nunca é mais restrita que a de seus enumerandos. Se um domínio enumeração for privado, então seus enumerandos também são privados e o uso da palavra chave **private** é opcional. Nos demais casos, se nenhuma palavra chave for utilizada, então os enumerandos possuem visibilidade de pacote. Só é possível definir enumerandos públicos para enumerações públicas.

Por exemplo, considere as enumerações definidas na Listagem 5.7: *LocationStatus* é uma enumeração pública, assim como seus enumerandos; a enumeração *Month* é pública, mas seus enumerandos possuem visibilidade de pacote; a enumeração *Color* é pública, mas seus enumerandos são privados; a enumeração *Language* e seus enumerandos possuem visibilidade de pacote; a enumeração *FileExtension* possui visibilidade de pacote, e seus enumerandos são privados; a enumeração *Fruit* e seus enumerandos são privados.

Se um mesmo identificador for utilizado como enumerando em duas enumerações distintas, então a disambiguação pode ser feita por meio da qualificação com o nome da enumeração. Por exemplo, o identificador *java* definido na Listagem 5.7 deve aparecer nas expressões do módulo como *Language.java* e *FileExtension.java*.

```
1 Value = Int | Bool | {error};
```

Listagem 5.8: Exemplo de Domínio União Disjunta

5.2.3.4 Definição de Domínios União Disjunta

Um domínio união disjunta representa uma soma separada de domínios e possui a forma:

$$d_1 \mid d_2 \mid \cdots \mid d_n,$$

onde cada d_i é um domínio; o domínio correspondente é $d_1 + d_2 + \cdots + d_n$.

Por exemplo, o domínio *Value* definido na Listagem 5.8 é a união disjunta dos domínios *Int*, *Bool* e $\{error\}$.

5.2.3.5 Definição de Domínios Tupla

Um domínio tupla representa o produto cartesiano de domínios e possui a forma:

$$constructor\text{-}name\ (d_1, d_2, \cdots, d_n),$$

onde cada d_i é um domínio; o domínio correspondente é $d_1 \times d_2 \times \cdots \times d_n$. O nome de construtor é utilizado em expressões de agregados de tupla (ver Seção 5.4.3.2). Se um domínio é definido por meio de apenas uma expressão tupla, então o nome de construtor não é obrigatório. Domínios de tupla podem ser definidos recursivamente, tais como *datatypes* da linguagem Haskell.

Por exemplo, considere domínios *Pair*, *AtMostThree* e *Tree*, definidos por meio de tuplas na Listagem 5.9. Neste exemplo,

- *Pair* representa o domínio $Int \times Bool$; o nome de construtor é *Pair*;
- *AtMostThree* representa o domínio $() + Int + (Int \times Int) + (Int \times Int \times Int)$ (ver uniões disjuntas na Seção 5.2.3.4);
- *Tree* é definido recursivamente e representa o domínio das árvores binárias não vazias de elementos do tipo *Int*.

A estrutura de um domínio tupla possui visibilidade de pacote. Visibilidade pública ou privada pode ser obtida inserindo a palavra chave correspondente antes do nome do construtor. A visibilidade da estrutura de uma tupla nunca é menos restrita que o identificador

```
1 Pair = (Int,Bool);
2 AtMostThree = Nil () | One (Int) | Two (Int,Int) | Three (Int,Int,Int);
3 Tree = Leaf Int | Branch (Int,Tree,Tree);
```

Listagem 5.9: Exemplos de Definições de Domínios de Tuplas

```
1 public Complex = public (Real,Real)
2 public LinkedList = List(Node,Node) // first and last elements
3 public Queue = private Queue(Int,LinkedList) // size and list
4 Node = {nil} | Node(Int,Node) // nil or pair of value and next node
5 Loc = private({location},Int}
```

Listagem 5.10: Exemplos de Controle de Visibilidade em Domínios de Tupla

da tupla em si. A estrutura de um domínio tupla privado é sempre privada, de modo que o uso da palavra chave **private** é opcional.

O exemplo da Listagem 5.10 define: um domínio público *Complex* cuja estrutura é também pública; um domínio público *LinkedList* cuja estrutura é visível somente no pacote; um domínio público *Queue* cuja estrutura é privada; domínio *Node*, que é completamente visível somente no pacote; e um domínio *Location*, que é visível no pacote, mas possui estrutura privada ao módulo. Note que não existe relação entre a definição de visibilidade e a definição do nome do construtor.

5.2.3.6 Definição de Domínios de Lista

Um domínio de lista representa listas de elementos em um dado domínio e possui a forma:

$$d*$$

onde d é um domínio; neste caso, o domínio representado contém todas as listas cujos elementos pertencem ao domínio d. Por exemplo, o domínio *Input* definido na Listagem 5.11 é uma lista de cadeias de caracteres.

```
1 Input = String*;
```

Listagem 5.11: Exemplo de Definição de Domínios de Lista

```
1 Store = Loc -> Value;
2 Econt = Value -> Store -> Store;
```

Listagem 5.12: Exemplos de Definição de Domínios Funcionais

```
1 PrefixExp = Int | [Operator PrefixExp PrefixExp]
2 Operator = "+" | "-" | "*" | "/"
```

Listagem 5.13: Exemplo de Definição de Domínios de Lista

5.2.3.7 Definição de Domínios Funcionais

Um domínio funcional representa as funções contínuas (*Scott* 1976) de um domínio em outro, e possui a forma:

$$d_1 \rightarrow d_2$$

onde d_1 e d_2 são domínios; neste exemplo, o domínio representado contém todas as funções contínuas de d_1 em d_2.

Por exemplo, considere os domínios funcionais *Store* e *Econt* definidos na Listagem 5.12. O operador $\rightarrow$ é associativo à direita, fazendo com que *Econt* represente o domínio $Value \rightarrow (Store \rightarrow Store)$.

5.2.3.8 Definição de Domínios de Sintaxe Abstrata

Algumas funções, tais como transformações de sintaxe abstrata, podem exigir a definição de novas estruturas de nós de árvore de sintaxe abstrata. Uma definição de domínio de sintaxe abstrata possui a forma:

$$[\ d_1\ d_2\ \cdots\ d_n\ ],$$

onde cada d_i representa uma cadeia de caracteres ou um domínio sintático válido.

Por exemplo, o domínio *PrefixExp* definido na Listagem 5.13 representa um domínio de sintaxe abstrata.

```
1 Value = Bool;
2 extend Value with Int;
```

Listagem 5.14: Exemplo de Extensão de Domínios

5.2.4 Extensões de Domínios

Domínios sintáticos e semânticos podem ser estendidos por meio de cláusulas de extensão de domínios, que possuem a seguinte forma:

extend *domain-name* **with** *domain-expression*;

Se um domínio D é definido por d_1, então a extensão inclui d_2 na definição de D, de modo que este passe a representar a união disjunta $d_1 + d_2$. Se o domínio estendido for um domínio sintático, então a expressão de domínio deve necessariamente ser uma especificação de domínio sintático, isto é, uma especificação de domínio de nó de árvore de sintaxe abstrata (ver Seção 5.3.4.3).

Por exemplo, no fragmento de código na Listagem 5.14, o identificador *Value* representa, após a extensão, o domínio `Bool` + `Int`.

5.3 Definição Sintática

5.3.1 Notação Básica

Expressões Regulares. Expressões regulares são utilizadas nas definições de analisador léxico e sua notação é similar à notação usual de expressões regulares encontrada em livros de Teoria de Linguagens. Ao longo do capítulo, dada uma expressão regular R, $L(R)$ é a linguagem denotada por R.

Gramáticas Livres Contexto. Gramáticas livres de contexto são escritas de maneira semelhante à Forma de Backus-Naur (BNF) com notação especial para repetição de símbolos. Assim como definido para expressões regulares, dada uma gramática G, $L(G)$ denota a linguagem gerada por G.

```
1 module Main
2     ... // module constituents
3     preproc f;
4     ... // other module constituents
5 end
```

Listagem 5.15: Exemplo de Definição de Função de Pré-processamento do Arquivo Fonte

5.3.2 Pré-processamento de Arquivo Fonte

Algumas linguagens, tais como Haskell, definem regras de simplificação da sintaxe, que auxiliam o programador a não poluir o seu código com delimitadores dedutíveis a partir do contexto. Tais simplificações podem ser implementadas em Notus por meio de funções de pre-processamento de arquivo fonte, que mapeia o fluxo de entrada fornecido no arquivo fonte em um novo fluxo de entrada para o analisador léxico.

A função de pré-processamento para uma linguagem é definida por meio de cláusula de pré-processamento no módulo principal da especificação inicial, e possui a forma:

preproc *function-name*

onde *function-name* é um nome de função visível no módulo principal. O domínio desta função deve ser $String \rightarrow String$. Funções são apresentadas na Seção 5.4.1 (página 113), e funções de pré-processamento, na Seção 5.4.1.4 (página 114).

Por exemplo, seja *Main* o módulo principal da especificação de uma linguagem L e seja f a função de pré-processamento de L. Então, o módulo *Main* deve ser definido como na Listagem 5.15.

5.3.3 Definição do Analisador Léxico

O analisador léxico é gerado automaticamente a partir da definição dos tokens da linguagem. Cada módulo pode introduzir novos identificadores representando:

- tokens da linguagem, i.e., identificadores que são símbolos terminais na definição sintática (ver Seção 5.3.4);
- macros, que podem representar partes da definição de tokens complexas.

```
1 public token id : Id = seqld ;
2 public token num : Num = digit+ is asInteger ;
3 element seqld = letter ( letter | digit ) * ;
4 public element letter = [A-Za-z];
5 public element digit = [0-9] ;
```

Listagem 5.16: Exemplos de Definição de Tokens e Macros

Tanto tokens quanto macros são definidos por meio de notação de expressões regulares (ver Seção 5.3.3.1) e podem ser definidos com visibilidade pública, privada ou de pacote. Tokens podem ainda pertencer a domínios sintáticos (ver Seção 5.2.2). Por exemplo, considere uma linguagem cujos tokens são sequências de dígitos de zero a nove e identificadores, representados por sequências de letras maiúsculas e minúsculas e dígitos, e que iniciem com uma letra. O trecho de código mostrado na Listagem 5.16 define o token *num*, que representa números inteiros, o token *id*, que representa identifiers, a macro *letter*, que representa letras, a macro *digit*, que representa dígitos, e a macro *seqld*, que representa sequências de letras e dígitos iniciando com letra. Os tokens *num* e *id*, e as macros *letter* e *digit* são públicas, ao passo que a macro *seqld* possui visibilidade de pacote.

A especificação de um token pode definir seu domínio sintático e como o lexema correspondente é interpretado. No exemplo da Listagem 5.16, o token *num* é definido com elemento do domínio *Num*, e o lexema associado é interpretado como um número inteiro. Definições de lexemas são possíveis somente para tokens que possuírem um domínio especificado. É possível também associar a definição de um lexema a uma função de interpretação (ver Seção 5.4.1) que recebe uma cadeia de caracteres como argumento e retorna um valor pertencente ao domínio sintático do token correspondente. As seguintes funções padrão podem ser utilizadas para a construção de tokens:

- *asBoolean*: $String \rightarrow Bool$;
- *asByte*: $String \rightarrow Byte$;
- *asShort*: $String \rightarrow Short$;
- *asInteger*: $String \rightarrow Int$;
- *asLong*: $String \rightarrow Long$;
- *asFloat*: $String \rightarrow Float$;

```
1 ignore layoutchar | comment ;
2 element layoutchar = " " | "\n" | "\t" | "\r" | "\f" ;
3 element comment = "//" .* ;
```

Listagem 5.17: Exemplos do Uso de Cláusulas *Ignore*

- *asDouble*: *String* → *Double*;
- *asCharacter*: *String* → *Char*;

Por outro lado, quando uma definição de token não contém a cláusula ***is***, o lexema correspondente é tratado como uma cadeia de caracteres e a função de interpretação é a identidade, $\lambda s.s$. Se o domínio de um token não for definido, então o domínio pode ainda ser obtido tornando a primeira letra de seu identificador maiúscula. Se não existir domínio com o nome obtido, então tem-se um token anônimo. No exemplo da Listagem 5.16, o token *id* pertence ao domínio *Id*, e o lexema correspondente é uma cadeia de caracteres correspondente ao texto casado na análise léxica.

Especificações léxicas podem ainda definir símbolos, tais como espaços em branco, que devem ser ignorados pelo analisador léxico. Essa definição é feita por meio do uso de cláusula *ignore*. Por exemplo, o trecho de código na Listagem 5.17 define que alguns caracteres especiais e comentários de linha devem ser ignorados.

5.3.3.1 Expressões Regulares

Expressões regulares em Notus são formadas por caracteres, cadeias de caracteres, identificadores e as operações: concatenação, representada por justaposição de expressões; união, por meio do operador `"|"`; fecho de Kleene, por meio dos operadores `"*"` e `"+"`; opção, por meio do `"?"`.

Além disso, expressões regulares que representem conjuntos de caracteres podem ser definidos pela enumeração dos caracteres entre colchetes, como em *[abc]*. É também possível definir *charsets* por meio de intervalos, designando-se o primeiro e o último elementos, como em *[a-z]*. Se necessário, intervalos e caracteres individuais podem ser utilizados para definir conjuntos mais complexos, como em *[a-zA-Z_]*, que representa o conjunto de caracteres formado por letras minúsculas, letras maísculas e o símbolo _.

Além disso, *charsets* podem ainda ser definidos por meio do complemento de um conjunto, como em *[^abc]*, que representa qualquer caractere diferente de *a*, *b* e *c*, ou *[^a-zA-Z]* que

precedência mais alta	"*", "+", "?"
	concatenação
precedência mais baixa	"|"

Tabela 5.2: Precedência das Operações de Expressões Regulares

Expressão	**Descrição**
`"//".*`	Comentário de linha
`[0-9]+("."[0-9]+)?([eE]"-"?[0-9]+)?`	Número de ponto flutuante
`[a-zA-Z_][a-zA-Z_0-9]*`	Identificador
`[+\-*/%]`	Operadores aritméticos
`[\n\f\r\t\a\b"\']`	Caracteres de escape

Tabela 5.3: Exemplos de Expressões Regulares

casa com qualquer caractere que não seja uma letra. Em definições de *charsets*, barra invertida (\) pode ser utilizada como escape para os seguintes símbolos:

\n, \r, \f, \t, \a, \b, \" , \', \-, \^, \[, \], \\

O símbolo "."representa qualquer caractere diferente de fim de linha ("\n").

Concatenação e união são associativos à esquerda, ao passo que as demais operações não são associativas. A ordem de precedência é mostrada na Tabela 5.2. A Tabela 5.3 contém exemplos de expressões regulares válidas em Notus.

5.3.3.2 Extensão de Tokens e Macros

Se necessário, é possível estender definições de tokens e macros por meio da união da expressão corrente com uma nova expressão regular. Definições de tokens e macros podem ser estendidas por meio de cláusulas de extensão, que possuem a forma:

extend *id* ***with*** *expression;*

Por exemplo, considere os tokens *number* e *id* definidos nas linhas 1 e 2 da Listagem 5.18, respectivamente. A inclusão de números inteiros em hexa-decimal com formato semelhante ao utilizado na linguagem C pode ser feita por meio da cláusula de extensão da linha 4 na mesma listagem. A definição de identificadores iniciados com o caractere _ pode ser feita por meio da cláusula de extensão da linha 5 do mesmo exemplo.

```
1 token number : Num = [0−9]+ is getNumber
2 token id : Id = [a−z]+[0−9]∗ is getId
3 ...
4 extend token number with "0x"[0−9A−Fa−f]+ is getNumberInHexa
5 extend token id with "_"[a−z]+[0−9]∗
```

Listagem 5.18: Exemplos de Extensão de Tokens

```
1 element letter = [a−z] ;
2 extend letter with [A−Z] ;
```

Listagem 5.19: Exemplo de Equivalência da Extensão de Macros – Parte I

Uma extensão de token pode ainda redefinir o seu domínio sintático. No entanto, só é possível definir uma nova função a ser aplicada ao lexema quando a nova expressão regular for casada. No exemplo da Listagem 5.18, quando uma cadeia de caracteres s casar com a expressão regular *[0-9]+*, a função *getNumber* é aplicada a s; por outro lado, se s casar com *"0x"[0-9A-Fa-f]+*, então a função *getNumberInHexa* é aplicada a s.

Se um token t, designado pela expressão regular R_1, for estendido para designar também a expressão R_2, então a linguagem denotada pelo token t é $L(R_1) \cup L(R_2)$.

Extensão de macros é feita de maneira similar à extensão de tokens. Considere a macro e definida pela expressão regular R_1; se e for estendida pela expressão regular R_2, então e passa a denotar a linguagem $L(R_1) \cup L(R_2)$. Por exemplo, a definição na Listagem 5.19 é equivalente à definição na Listagem 5.20.

5.3.3.3 Domínios Sintáticos de Tokens

O domínio de um token t é a união disjunta dos tipos correspondentes aos lexemas representados por t. Por exemplo, considere o trecho de código da Listagem 5.21, que define os domínios *Id* e *Value*, e os tokens *id*, *num* e *bool*.

O token *id* pertence ao domínio *Id*, e os tokens *num* e *bool* pertencem ao domínio *Value*. Os lexemas correspondentes ao token *id* são as cadeias de caracteres casadas pelo analisador

```
1 element letter = [a−z] | [A−Z] ; // or [a−zA−Z]
```

Listagem 5.20: Exemplo de Equivalência da Extensão de Macros – Parte II

```
1 token id = [a−zA−Z]+;
2 token num : Value = [0−9]+ is asInteger ;
3 token bool : Value = "true" | "false" is asBoolean ;
```

Listagem 5.21: Exemplo de Definição de Domínio Sintático de Tokens

léxico, uma vez que não há especificação da função de lexema. Além disso, os lexemas correspondentes ao token *num* são números inteiros não-negativos, e os lexemas correspondentes ao token *bool* são os valores *true* e *false*. Com isso, os domínios *Value* e *Id* são definidos como:

$$\begin{aligned} Value &= \mathbb{N} + \{true, false\} \\ Id &= L\left(\texttt{[a-zA-Z]+}\right) \end{aligned}$$

Formalmente, seja d um domínio sintático, e $t_1, t_2, \cdots, t_m$ os tokens da especificação definidos no domínio d. Suponha que cada token t_i seja definido por meio das expressões regulares $R_{i1}, R_{i2}, \cdots, R_{in_i}$, e cada R_{ij} seja associado a uma função de lexema f_{ij}. O domínio d é definido como:

$$d = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n_i} \{f_{ij}(u) \mid u \in L(R_{ij})\}$$

A partir desta definição, o domínio *Value* no exemplo anterior é definido por:

$$\begin{aligned} Value &= \{asInteger\ u \mid u \in L\left(\texttt{[0-9]+}\right)\} \\ &\quad + \{asBoolean\ u \mid u \in L\left(\texttt{ "true" | "false" }\right)\} \\ &= \mathbb{N} + \{true, false\} \end{aligned}$$

5.3.3.4 Ambiguidades na Definição de Tokens

Ambiguidades na definição de tokens ocorrem quando existe uma cadeia de caracteres s e dois tokens t_1 e t_2, representando expressões regulares R_1 e R_2, tal que $s \in L(R_1) \cap L(R_2)$. Geradores de analisadores léxicos típicos resolvem tais ambiguidades utilizando a ordem em que t_1 e t_2 são definidos no arquivo fonte. No entanto, essa regra não pode ser aplicada em Notus, uma vez que a linguagem não impõe ordem para a definição de tokens em um arquivo, e, além disso, definições de tokens não precisam necessariamente estar confinadas em um único módulo.

```
1 token decNumber = [0−9]+
2 token octNumber = "0"[0−7]+
3 token hexNumber = "0x"[0−9a−f]+
4 token score = [0−9]+"x"[0−9]+
```

Listagem 5.22: Exemplos de Ambiguidades nas Definições de Tokens

Para resolver tais ambiguidades, a linguagem Notus aplica as seguintes regras de desambiguação de uma cadeia de caracteres s que case com as expressões regulares R_1 e R_2 dos tokens t_1 e t_2:

- se $L(R_1) \subset L(R_2)$, $L(R_1) \neq L(R_2)$ e $s \in L(R_1)$, então considera-se que s case com o token t_1;
- se $L(R_1) \not\subset L(R_2)$, $L(R_2) \not\subset L(R_1)$ e $L(R_1) \cap L(R_2) \neq \emptyset$, então um erro de compilação deve ser indicado, pois não é possível decidir se $s \in L(R_1) \cap L(R_2)$ deve ser considerado como token t_1 ou t_2.

No exemplo da Listagem 5.22, seja L_1 a linguagem representada pelo token *decNumber*, L_2 a linguagem representada pelo token *octNumber*, L_3 a linguagem representada pelo token *hexNumber*, e L_4 a linguagem representada pelo token *score*. É fácil verificar que $L_2 \subset L_1$; assim, qualquer cadeia de caracteres que casar com a expressão regular *"0"[0-7]+* pertence à linguagem L_2. Por outro lado, há cadeias que casam tanto com *"0x"[0-9a-f]+* quanto com *[0-9]+*"x"*[0-9]+*, como por exemplo *0x0*; porém nem $L_3 \subset {L_4}^3$ e nem $L_4 \subset {L_3}^4$. Assim, não é possível decidir quais tokens representam algumas strings, e por esta razão, um erro de compilação é gerado.

5.3.4 Definição do Analisador Sintático

O analisador sintático da linguagem é gerado automaticamente a partir das definições de domínios sintáticos, das regras de produção e do símbolo de partida da gramática concreta. O analisador sintático gerado processa a sequência de tokens fornecida pelo analisador léxico e produz como resultado a árvore de sintaxe abstrata correspondente ao programa. Esta árvore será um dos argumentos da função semântica de programas da especificação.

[3] Um contra-exemplo é a cadeia $s = 0x8a$, pois $s \in L_3$ e $s \notin L_4$.
[4] Um contra-exemplo é a cadeia $s = 5x3$, pois $s \in L_4$ e $s \notin L_3$.

```
1 public comm : Comm ;
2 private cond : Comm ;
3 assign : Comm ;
```

Listagem 5.23: Exemplos de Visibilidade das Variáveis de uma Gramática

```
1 public exp : Exp ::= exp "+" term | exp "−" term | term ;
2 term : Exp ::= term "∗" primary | term "/" primary | primary ;
3 primary : Exp ::= id | "(" exp ")" | num ;
```

Listagem 5.24: Exemplos de Definição de Regras de Produção

Uma definição sintática em Notus consiste nas definições de sintaxes concreta e abstrata da linguagem, ambas definidas por meio de gramáticas livres de contexto. Geralmente, o projetista define os nomes dos domínios sintáticos (veja Section 5.2.2) e a gramática concreta, e o compilador Notus gera automaticamente a estrutura dos domínios sintáticos e a gramática abstrata.

Os terminais da gramática concreta são os tokens, definidos por meio das cláusulas de elementos léxicos. Os não-terminais são definidos por meio de declarações de variáveis, que listam identificadores das variáveis e os domínios que elas representam. Assim como na definição de tokens, variáveis podem possuir visibilidade pública, privada ou de pacote. Por exemplo, na Listagem 5.23 a variável *comm* é pública, a variável *assign* é visível somente no pacote, e a variável *cond* é privada. Neste exemplo, todas as variáveis pertencem ao domínio sintático *Comm*.

Regras de produção da gramática concreta podem ser definidas na declaração de suas variáveis do lado esquerdo ou por meio de cláusulas de extensão (ver Seção 5.3.4.5). Por exemplo, o trecho de código na Listagem 5.24 define regras de produção para uma linguagem de expressões simples. Consideram-se nesta definição os tokens *num* e *id*, definidos na Seção 5.3.3.

Nomes que aparecem em uma regra de produção devem identificar tokens ou variáveis da gramática. Símbolos entre aspas duplas são tokens literais da linguagem e são automaticamente incluídas no léxico. No exemplo da Listagem 5.24, os símbolos `"+"`, `"-"`, `"*"`, `"/"`, `"("` e `")"` são automaticamente inseridos no léxico da linguagem, de modo que não é necessário definir tokens que os representem. Tais tokens não possuem, desta maneira, especificação de domínio e, portanto, são associados a domínios anônimos unitários que contêm os valores literais correspondentes.

5.3.4.1 Gramáticas Livres de Contexto

A sintaxe de uma linguagem em Notus deve ser definida por meio de uma gramática livre de contexto LALR(1)[5]. Se $G = (V, T, R, S)$ for uma gramática livre de contexto para alguma linguagem arbitrária L, então G deve ser definida em Notus da seguinte maneira:

- o conjunto de não-terminais V é composto por todas as variáveis declaradas nos módulos da especificação de L;
- o conjunto de terminais T é composto por todos os tokens declarados e por todos os símbolos entre aspas que aparecem em regras de produção definidas nos módulos da especificação de L;
- o conjunto de regras de produção R é composto pelas regras definidas nos módulos da especificação de L;
- o símbolo inicial $S \in V$ é designado pela cláusula de símbolo de partida no módulo principal da especificação de L (ver Seção 5.1.5).

Regras de produção são definidas em notação semelhante a BNF, com atalhos para repetição de símbolos. Uma regra em R possui a forma: $A ::= \alpha_1\alpha_2 \cdots \alpha_n$, onde A é um elemento de V, cada α_i, $1 \leq i \leq n$, é um constituinte de regra, e $n \geq 0$. Cada constituinte pode ser:

- um elemento de V;
- um elemento de T;
- uma repetição de constituintes, de uma das seguintes formas:
  - α`*`, que representa qualquer quantidade de ocorrências de α;
  - α`+`, que representa pelo menos uma ocorrência de α;
  - α`*-`t, que representa qualquer número de ocorrências de α, separadas pelo token t, em que t é um token anônimo[6];
  - α`+-`t, que representa pelo menos uma ocorrência de α, separadas pelo token t, em que t é um token anônimo;

[5]Esta restrição se deve à implementação atual do compilador da linguagem Notus. Versões futuras da linguagem poderão eventualmente utilizar outro formalismo.

[6]Tokens anônimos são cadeias de caracteres que aparecem do lado direito em regras de produção e, geralmente, indicam separadores e palavras chaves.

```
1 prog  ::= decls comms
2 decls ::= decl*−";"
3 decl  ::= "var" id+−","
4 comms ::= comm+
5 comm ::= id "<−" exp
6         | "if" exp "then" comm "else" comm
7         | block
8 block ::= "{" comm* "}"
```

Listagem 5.25: Exemplos de Repetição em Regras de Produção

No exemplo da Listagem 5.25: o não-terminal *decls* produz uma sequência possivelmente vazia de *decl* separadas por ponto-e-vírgula; o não-terminal *decl* produz uma sequência não-vazia de *id* separados por vírgula e precedida pela palavra "`var`"; o não-terminal *comms* produz uma sequência não-vazia de *comm*; o não-terminal *block* produz uma sequência possivelmente vazia de *comm* entre chaves.

A escrita de regras de produção da forma $A \rightarrow \lambda$ pode ser feita de uma das seguintes maneiras:

$$A \quad ::= \quad$$
$$A \quad ::= \quad \textbf{empty}$$

Nesta definição, **empty** é um terminal especial que representa a sequência de símbolos vazia.

Para permitir que as gramáticas definidas em Notus sejam processadas por geradores de parser LALR(1) tradicionais, define-se a função $\zeta : \mathcal{G} \rightarrow \mathcal{G}$, onde $\mathcal{G}$ é o conjunto das gramáticas livres de contexto. Dada uma gramática em Notus G, $G' = \zeta(G)$ é uma nova gramática cujas regras de produção estão em conformidade com a notação BNF e $L(G) = L(G')$, i.e. G e G' geram a mesma linguagem. A função ζ é definida na Seção C.1.

5.3.4.2 Definição de Símbolo de Partida

O símbolo de partida da gramática é definido por meio da cláusula de definição de símbolo de partida no módulo principal da especificação. Esta cláusula possui a forma:

syntax *start−symbol−name;*

```
1 module Main
2     ... // module constituents
3     syntax program;
4     ... // other module constituents
```

Listagem 5.26: Exemplo de Definição do Símbolo de Partida de uma Linguagem

onde *start-symbol-name* é um não-terminal da gramática visível a partir do módulo principal. O símbolo de partida deve ser definido unicamente no módulo principal da especificação inicial e não pode ser alterado nas demais extensões da linguagem.

Por exemplo, o módulo principal da especificação de uma linguagem hipotética L, mostrado na Listagem 5.26, define o não-terminal *program* como o símbolo de partida da gramática de L.

5.3.4.3 Definição da Sintaxe Abstrata

A sintaxe abstrata da linguagem é obtida automaticamente a partir da especificação da sintaxe concreta, por meio de uma função $\chi : \mathcal{G} \rightarrow \mathcal{G}$, tal que dada uma gramática Notus $G = (V, T, R, S)$, $G' = \chi(G)$ é a gramática abstrata correspondente a G. A função χ está definida na Seção C.3. A versão final da gramática abstrata é obtida a partir de G', eliminando-se símbolos e regras de produção inúteis, conforme os algoritmos descritos na Seção C.2.

Por exemplo, considere o módulo *Main* da Listagem 5.27, em que se define uma linguagem simples de expressões.

A gramática abstrata da linguagem é $G' = (V', T', R', \texttt{Exp})$, onde:

- $V' = \{\texttt{Id}, \texttt{Num}, \texttt{Exp}\}$;

- $T' = \{\texttt{"+"}, \texttt{-}, \texttt{"*"}, \texttt{"/"}, \texttt{"("}, \texttt{")"}, \texttt{id}, \texttt{num}\}$

```
1  module Main
2
3      // Lexis
4      public token id = seqld ;
5      public token num = digit+ is asInteger ;
6
7      element seqld = letter ( letter | digit ) * ;
8      public element letter = [A-Za-z] ;
9      public element digit = [0-9] ;
10
11     // Syntax
12     syntax exp
13
14     public exp ::= exp "+" term | exp "-" term | term ;
15     public term : Exp ::= term "*" factor | term "/" factor | factor ;
16     public factor : Exp ::= "(" exp ")" | id | num ;
17
18     // Semantics
19     ...
```

Listagem 5.27: Exemplo de Definição de Gramática Concreta Para Expressões

```
1  public exp : Exp ::=
2              exp "+" term : [ "add" exp term ]          // rule 1
3          |   exp "−" term : [ "sub" exp term ]          // rule 2
4          |   term ;                                     // rule 3
5  term : Exp ::=
6              term "∗" primary : [ "mul" term primary ]  // rule 4
7          |   term "/" primary : [ "div" term primary ]  // rule 5
8          |   primary ;                                  // rule 6
9  primary : Exp ::=
10             id : id                                    // rule 7
11         |   num : num                                  // rule 8
12         |   "(" exp ")" : exp ;                        // rule 9
```

Listagem 5.28: Exemplo de Definição Alternativa de Sintaxe Abstrata

- R' : `Exp` → `Exp "+" Exp`
  | `Exp - Exp`
  | `Exp "*" Exp`
  | `Exp "/" Exp`
  | `"(" Exp ")"`
  | `Id`
  | `Num`

  `Id` → `id`

  `Num` → `num`

Além disso, caso uma especificação de sintaxe abstrata seja desejável, então é possível definir a tradução anexando-se regras de definição alternativas às regras de produção da gramática concreta. Por exemplo, considere a definição de expressões na Listagem 5.28, que ilustra os mecanismos de definição alternativa de sintaxe abstrata de Notus.

O primeiro mecanismo é a definição de regra de gramática abstrata que consiste em anexar à regra de produção um novo lado direito; esta construção possui a forma:

$$a \rightarrow a_1 a_2 \cdots a_n : [\ b_1 b_2 \cdots b_m\ ]$$

em que cada b_i é um dos a_j ou uma cadeia de caracteres entre aspas. Se a pertencer ao domínio d e cada b_i ao domínio d_i, então a regra de gramática abstrata é $d \rightarrow d_1 d_2 \cdots d_m$.

O segundo mecanismo é a definição de regras para ignorar indireção, que consiste em anexar à regra de produção uma das variáveis presentes no lado direito da regra de produção

concreta, e possui a forma:

$$a \rightarrow a_1 a_2 \cdots a_n : a_i$$

onde a_i é um não-terminal ou um terminal pertencente ao conjunto $\{a_1, a_2, \cdots, a_n\}$. Neste caso, considerando-se que a pertença ao domínio d e que a_i pertença ao domínio d_i, então a gramática abstrata possui uma regra $d \rightarrow b_1 b_2 \cdots b_m$ para cada $d_i \rightarrow b_1 b_2 \cdots b_m$ pertencente à gramática abstrata. O efeito desta transformação, em vez de criar um novo nó quando esta regra de produção for reduzida, reutilizar o nó correspondente ao não-terminal a_i.

No exemplo da Listagem 5.28, as regras de produção 3 e 6 não possuem definições alternativas e, por esta razão, uma nova regra $Exp \rightarrow Exp$ é incluída na gramática abstrata [7]. Por outro lado, as regras 1, 2, 4 e 5 definem regras alternativas. Em cada caso, define-se uma nova regra de produção na gramática abstrata tendo como lado esquerdo o domínio sintático *Exp*. Além disso, as regras 7, 8 e 9 definem indireções a ser ignoradas. Para a regra 7, uma nova regra $Exp \rightarrow id$ é incluída na gramática abstrata, uma vez que existe a regra $Id \rightarrow id$ [8]. Da mesma forma, para a regra 8, uma nova regra $Exp \rightarrow num$ é incluída na gramática abstrata. Para a regra 9, uma nova regra $Exp \rightarrow Exp$ é adicionada à gramática abstrata[9].

A gramática abstrata correspondente ao exemplo, após simplificada pelos algoritmos apresentados na Seção C.2 é $G^A_{Exp} = (V', T', R', \texttt{Exp})$, onde:

- $V' = \{\texttt{Id}, \texttt{Num}, \texttt{Exp}\}$;
- $T' = \{\texttt{"add"}\ , \texttt{"sub"}\ , \texttt{"mul"}\ , \texttt{"div"}\ , \texttt{id}, \texttt{num}\}$
- R' :

```
Exp → "add" Exp Exp
    | "sub" Exp Exp
    | "mul" Exp Exp
    | "div" Exp Exp
    | id
    | num
```

Na definição alternativa de regras de produção, se algum símbolo aparecer mais de uma vez do lado direito de uma regra, então é possível distinguir cada ocorrência pelo uso de sufixos numéricos nos identificadores. Por exemplo, considere a definição de comandos no

[7] Esta regra será sujeita a simplificação da gramática discutida na Seção C.2
[8] Ver função χ' na Seção C.3
[9] Esta regra será sujeita a simplificação da gramática discutida na Seção C.2

```
1  public comm : Comm ::=
2                    id "=" exp ";" : [ id ":=" exp ]
3                    |   "if" "(" exp ")" comm1 "else" comm2
4                          : [ "if" exp "then" comm1 "else" comm2 ] ;
```

Listagem 5.29: Exemplo do Uso de Decoração na Definição de Gramática Concreta

```
1  module Main
2
3      // Lexis
4      token fun = [fgh][0−9]∗ ;
5      token var = [xyz][0−9]∗ ;
6      token num = [0−9]+ is asInteger ;
7
8      // Syntax
9      syntax exp
10
11     exp ::= fun "(" exp+−"," ")" : fun exp+
12                  |   var : var
13                  |   num : num ;
14
15     // Semantics
16      ...
```

Listagem 5.30: Exemplo de Linguagem de Funções com Múltiplos Argumentos

trecho de código da Listagem 5.29. Neste exemplo, a variável *comm* aparece duas vezes no lado direito da regra que define o comando `if-then-else`, e portanto cada ocorrência é diferenciada por meio de uma decoração numérica.

Outro exemplo de geração de gramática abstrata é dado no trecho de código da Listagem 5.30, que define uma linguagem de funções com múltiplos argumentos.

Neste exemplo, a sintaxe abstrata é definida pela gramática $G^A_{MF} = (V, T, R, Exp)$, onde:

- $V = \{Exp, Fun\}$;
- $T = \{fun, var, num\}$;

- R : `Exp` $\rightarrow$ `Fun Exp+`
  | `var`
  | `num`
  `Fun` $\rightarrow$ `fun`

5.3.4.4 Definição de Domínios Sintáticos

Cada regra de produção definida na gramática abstrata estende o domínio de sua variável do lado esquerdo por meio da inclusão de sua definição como uma nova parcela derivada do lado direito da regra.

Sejam $G = (V, T, R, S)$ a gramática abstrata de uma linguagem e uma regra de produção $d \rightarrow \alpha$ pertencente a R, onde d é um nome de domínio sintático pertencente a V, $\alpha = a_1 a_2 \cdots a_n$ e cada a_i é um constituinte de gramática abstrata (ver Seção 5.3.4.3), então a nova parcela no domínio é definida dada por $\delta(\alpha)$, onde a função δ é definida em casos da seguinte maneira:

- $\delta(\alpha) = \{\alpha\}$, se α for um não-terminal que não possui definição de domínio;
- $\delta(\alpha) = d'$, se α for um não-terminal cujo domínio é d';
- $\delta(\alpha) = d'$, se α for um terminal cujo domínio é d';
- $\delta(\alpha\texttt{*}) = \delta(\alpha)^*$;
- $\delta(\alpha\texttt{+}) = \delta(\alpha)^*$; [10]
- $\delta(a_1 a_2 \cdots a_n) = \delta(a_1) \times \delta(a_2 \cdots a_n)$;

Com isso, o domínio sintático d pode ser definido como:

$$d = \bigcup_{d \rightarrow \alpha \in R} \delta(\alpha)$$

[10] Não há representação especial para o domínio de listas não-vazias; por isso, os domínios correspondentes às operações de fecho de Kleene e fecho de Kleene positivo são iguais.

Por exemplo, na gramática G^A_{Exp} da Seção 5.3.4.3 (página 109), o domínio *Exp* é definido por:

$$\begin{aligned} Exp &= Id + Num \\ &\quad + (\{\text{“}\texttt{add}\text{”}\} \times Exp \times Exp) \\ &\quad + (\{\text{“}\texttt{sub}\text{”}\} \times Exp \times Exp) \\ &\quad + (\{\text{“}\texttt{mul}\text{”}\} \times Exp \times Exp) \\ &\quad + (\{\text{“}\texttt{div}\text{”}\} \times Exp \times Exp) \end{aligned}$$

Considere ainda o módulo da Listagem 5.30 e sua gramática abstrata, G^A_{MF}, definida na página 110. Os domínios sintáticos desta especificação são definidos a partir da gramática abstrata, de modo que:

- $Exp = Num + Var + (Fun \times Exp^*)$;
- $Num = \{asInteger\ u \mid u \in L\,([0\text{-}9]+)\}$;
- $Var = \{(\lambda s.s)\ u \mid u \in L\,([xyz][0\text{-}9]*)\}$;
- $Fun = \{(\lambda s.s)\ u \mid u \in L\,([fgh][0\text{-}9]*)\}$.

5.3.4.5 Extensão de Gramática

Para possibilitar a definição separada, Notus permite adicionar à gramática novas regras de produção tendo como lados esquerdos não-terminais que já tenham sido definidos. O objetivo é permitir organizar gramáticas longas em módulos e definir extensões da linguagem sem a necessidade de alterar módulos existentes.

Extensões da gramática são definidas por meio da cláusula de extensão de variáveis, que possui a forma:

extend `id` **with** *grammar-constituent-list* ;

Por exemplo, o trecho de código da Listagem 5.31 define separadamente regras para expressões aritméticas. Não há distinção entre regras definidas diretamente na declaração de uma variável e regras definidas em cláusulas de extensão. Portanto, nenhuma modificação é necessária na definição de gramáticas abstratas e domínios sintáticos.

```
1 public exp ::= term ;
2 public term : Exp ::= primary ;
3 public primary : Exp ;
4  ...
5 extend exp with exp "+" term | exp "−" term ;
6 extend term with term "∗" primary | term "/" primary ;
7 extend term with "(" exp ")" | num | id ;
```

Listagem 5.31: Exemplo de Extensão de Gramática

```
1     public function denexp : Exp -> Value
```

Listagem 5.32: Exemplo de Declaração de Função

5.4 Definição Semântica

5.4.1 Funções Semânticas

Para definir uma função semântica em Notus, é necessário em primeiro lugar definir a sua assinatura por meio de uma declaração de função, e então definir seu corpo, possivelmente em casos, por meio de pelo menos uma definição de função. Definições de função podem aparecer em qualquer ordem e em módulos distintos e definir o resultado de aplicações de função para padrões distintos.

5.4.1.1 Declaração de Função

Uma declaração de função define a assinatura de uma função e possui a forma:

visibility **function** *function-name* : *domain-expression*;

A Listagem 5.32 contém um exemplo de definição de função *denexp* cuja assinatura é $denexp : Exp \rightarrow Value$.

5.4.1.2 Definição de Função

Funções podem ser definidas por casos, baseados em casamento de padrões, sendo cada caso uma definição de função da forma:

function-name parameter-patterns = *expression*;

```
1    public function fact : Int -> Int;
2    fact 0 = 1;
3    fact n = n * fact (n - 1);
4
5    public function pow : Int -> Int -> Int;
6    pow a 0 = 1;
7    pow a n = a * pow a (n - 1);
```

Listagem 5.33: Exemplos de Definição de Função

Parâmetros de função são padrões (ver Seção 5.4.2, página 120) que devem ser casados com os argumentos em aplicações de função (ver Seção 5.4.3.6, página 132).

Por exemplo, considere as funções *fact* e *pow* definidas na Listagem 5.33. A definição da função *fact* utiliza padrões `0` e `n`. Com isso, o resultado da aplicação da função *fact* ao valor *0* é *1*; caso contrário, o valor do argumento é associado ao identificador *n* e o resultado é o valor da expressão correspondente ao seu corpo neste novo ambiente. Identificadores utilizados nos padrões de uma definição de função devem ser distintos.

5.4.1.3 Funções Pré-Definidas

A Tabela 5.4 apresenta as funções pré-definidas de Notus. Nesta tabela, a primeira coluna contém os nomes das funções, a segunda coluna, seus domínios, e a terceira coluna, as descrições.

5.4.1.4 Funções de Pré-Processamento de Arquivo Fonte

Funções de pré-processamento de arquivo fonte pertencem ao domínio $String \rightarrow String$ e são utilizadas para produzir uma versão pré-processada do arquivo fonte para o analisador léxico.

Considere, por exemplo, uma linguagem L composta por variáveis, expressões aritméticas, funções, comandos para entrada e saída e atribuições. Cada linha representa uma atribuição e há um ponto e vírgula opcional no fim de cada linha, indicando que o valor atribuído deve ser impresso na saída padrão. Além disso, linhas com muitos caracteres podem ser quebradas e ambiguidades são resolvidas por indentação. A Listagem 5.34 mostra um exemplo de programa que lê três números, a, b e c, e computa os valores de d, e e f. As

Name	Domains	Description
`acos`	$Real \rightarrow Real$	inverse cosine of a number in $[-1, 1]$
`asin`	$Real \rightarrow Real$	inverse sine of a number in $[-1, 1]$
`atan`	$Real \rightarrow Real$	inverse tangent of a number
`ceil`	$Real \rightarrow Real$	round toward infinity
`cos`	$Real \rightarrow Real$	cosine of an angle in radians
`cosh`	$Real \rightarrow Real$	hyperbolic cosine of a number
`empty`	$a* \rightarrow Bool$	checks if a list is empty
`exp`	$Real \rightarrow Real$	exponential of a number
`fix`	$(a \rightarrow a) \rightarrow a$	least fix point of a function
`floor`	$Real \rightarrow Real$	round toward minus infinity
`hd`	$a* \rightarrow a$	first element of a non-empty list
`log`	$Real \rightarrow Real$	natural logarithm of a positive number
`max`	$Real* \rightarrow Real$ $Int* \rightarrow Int$	the largest element of a non-empty list
`min`	$Real* \rightarrow Real$ $Int* \rightarrow Int$	the smallest element of a non-empty list
`pow`	$Real \times Real \rightarrow Real$	power function
`round`	$Real \rightarrow Real$	round to the nearest integer
`sqr`	$Real \rightarrow Real$	square power of a number
`sqrt`	$Real \rightarrow Real$	square root of a number
`sin`	$Real \rightarrow Real$	sine of an angle in radians
`sinh`	$Real \rightarrow Real$	hyperbolic sine of a number
`tan`	$Real \rightarrow Real$	tangent of an angle in radians
`tanh`	$Real \rightarrow Real$	hyperbolic tangent of a number
`tl`	$a* \rightarrow a*$	tail of a non-empty list
`trunc`	$Real \rightarrow Real$	round toward zero

Tabela 5.4: Funções Pré-Definidas em Notus

```
1 a = read
2 b = read
3 c = read;
4 d = a + b + c
5 e = pow a b * pow c d / sin c * cos b +
6       pow a c
7 f = pow a b * pow b d + sin c * cos b + pow a
8       c;
```

Listagem 5.34: Exemplo de Programa na Linguagem L

linhas 5 e 6 representam um único comando; o mesmo ocorre com as linhas 7 e 8. Além disso, os valores de c e f serão impressos na tela, uma vez que os comandos das linhas correspondentes terminam com um ponto e vírgula.

Considere que a Listagem 5.35 contenha o módulo principal da especificação inicial de L, que define a função de pré-processamento como *preprocessing* do módulo *Preprocessing*, definida na Listagem 5.36. A função recebe como argumento uma cadeia de caracteres representando o código fonte e produz uma nova cadeia representando o código fonte a ser lido pelo analisador léxico. Esta função conta o número de espaços em branco antes do primeiro caractere não vazio em uma linha para decidir se um novo comando foi iniciado ou se a linha é uma continuação da anterior. Além disso, para simplificar o analisador sintático, toda vez que o fim de um comando for encontrado, um ponto-e-vírgula é adicionado como delimitador do comando. Se um ponto-e-vírgula for encontrado pelo pré-processador no fim de uma linha, então adiciona-se à entrada o texto `"print` x`;"`, onde x é o identificador do lado esquerdo da última atribuição.

Portanto, a especificação correspondente a ser lida pelo analisador léxico quando a entrada for o programa da Listagem 5.34 é semânticamente equivalente ao programa da Listagem 5.37.

5.4.1.5 Função Semântica da Especificação

A denotação de um programa é a aplicação da função semântica da especificação à árvore de sintaxe abstrata do programa fonte gerado pelo analisador sintático. A função semântica da linguagem deve ser definida no módulo principal da especificação inicial da linguagem

```
1  module Main
2
3      preproc Preprocessing.preprocessing
4
5      // lexis
6      token id = [a-z]+
7      token num = [0-9]+ is asInteger
8      ignore [ \n\r\t]
9
10     // syntax
11     syntax program
12
13     program ::= command+
14     command ::= assignment | printing
15     assigment : Command ::= id "=" expression ";"
16     printing : Command ::= "print" id ";"
17     ...
18
19     ... // semantics
```

Listagem 5.35: Exemplo de Função de Pré-Processamento

```
1  module Preprocessing
2      public function preprocessing: String -> String
3      preprocessing s = lhs s 0 ""
4
5      function lhs: String -> Int -> String -> String
6      lhs () _ _ = ()
7      lhs '\n':s1 _ s2 = '\n' : lhs s1 0 s2
8      lhs '=':s1 i s2 = '=' : rhs s1 (i + 1) (i + 1) s2
9      lhs c:s1 i s2 = c : lhs s1 (i + 1) (s2 ++ c)
10
11     function rhs: String -> Int -> Int -> String -> String
12     rhs () _ _ _ = ()
13     rhs ' ':s1 i1 i2 s2 = ' ' : rhs s1 i1 (i2 + 1) s2
14     rhs '\n':s1 i1 i2 s2 = '\n' : rhs s1 i1 0 s2
15     rhs c:s1 i1 i2 s2 =
16             if ge i1 i2 then c : rsh_f s i1 i2 s2 else "?"
17
18     function rsh_f: String -> Int -> Int -> String -> String
19     rsh_f () _ _ _ = ";"
20     rsh_f ' ':s1 i1 i2 s2 = ' ' : rsh_f s1 i1 (i2 + 1) s2
21     rsh_f '\n':s1 i1 i2 s2 = '\n' : rsh_f s1 i1 (i2 + 1) s2
22     rsh_f c:s1 i1 i2 s2 =
23             if ge i1 i2 then
24                 if eq c ';' then
25                     c : " print" ++ s2 ++ ";" ++ rsh_f s1 i1 (i2 + 1)
26                 else
27                     c : rsh_f s ind_st (i2 + 1)
28             else ';' : c : lhs s indent
```

Listagem 5.36: Definição da Função de Pré-Processamento

```
1 a = read
2 ;b = read
3 ;c = read; print c ;
4 d = a + b + c
5 ;e = pow a b * pow c d / sin c * cos b +
6       pow a c
7 ;f = pow a b * pow b d + sin c * cos b + pow a
8       c; print f ;
```

Listagem 5.37: Programa correspondente ao programa da Listagem 5.34 (página 116) a ser lido pelo analisador léxico

e possui a forma:

`semantics` *semantic-function*

onde *semantic-function* é uma função cujo primeiro argumento pertence ao domínio sintático do símbolo de partida da gramática da linguagem. Entradas do interpretador para a linguagem são argumentos da função semântica da especificação; há duas possibilidades de definição:

- se o segundo argumento for uma cadeia de caracteres, então o interpretador possui uma única entrada, que é lida da entrada padrão;
- se o segundo argumento for uma lista de cadeias de caracteres, então o interpretador possui mais de uma entrada, definidas conforme regras da Seção 5.1.5.

O resultado da função semântica corresponde à saída do interpretador. Também há duas possibilidades de definição:

- se o tipo de retorno for uma cadeia de caracteres, então o interpretador possui uma única saída, que é escrita na saída padrão;
- se o tipo de retorno for uma lista de cadeias de caracteres, então o interpretador possui mais de uma saída, definidas conforme regras da Seção 5.1.5.

Os tamanhos das listas devem estar de acordo com o número de fluxos de entradas e saídas especificados. Caso contrário, uma mensagem de erro é gerada durante a execução.

Por exemplo, na Listagem 5.38, o símbolo de partida da gramática é *prog*, que possui domínio sintático *Prog* (ver linhas 2 e 8). A função semântica é *p* (ver linhas 3, 11 e 12),

```
1  module Main
2      syntax prog
3      semantics p
4      input stdin, infile
5      output stdout, outfile
6
7      // grammar
8      prog ::= ...
9
10     // semantics
11     function p : Prog -> String* -> String*
12     p prog (i1,i2) = (o1,o2) where o1 = ...; o2 = ...
```

Listagem 5.38: Exemplo de Função Semântica

cujo primeiro argumento pertence ao domínio *Prog*. O segundo argumento é uma lista de cadeias de caracteres, que corresponde às entradas do interpretador, definidas na linha 4; o primeiro elemento é lido da entrada padrão, e o segundo, de um arquivo a ser definido na linha de comando de execução do interpretador, conforme regras da Seção 5.1.5. O resultado da função é uma lista de cadeias de caracteres, que corresponde às saídas do interpretador, definidas na linha 5; o primeiro elemento é escrito na saída padrão, e o segundo, em um arquivo a ser definido na linha de comando de execução do interpretador, conforme regras da Seção 5.1.5.

5.4.2 Padrões

A linguagem Notus possui os seguintes padrões: valores literais; idenficadores; padrão anônimo; agregados para tuplas; listas; nós de árvore de sintaxe abstrata.

5.4.2.1 Valores Literais

Um padrão pode ser um valor literal (ver Seção 5.4.3.1), conforme mostrado na definição da função *fact* na Listagem 5.33, linha 2, da Seção 5.4.1.2. Este padrão casa apenas com o valor correspondente.

```
1    Value = Int | Bool | {error};
2
3    function sum : Value -> Value -> Value;
4    sum int1 int2 = add int1 int2;
5    sum _ _ = error;
```

Listagem 5.39: Exemplo de Padrões Identificadores

5.4.2.2 Identificadores e Padrões Anônimos

Um padrão pode ser designado por um identificador, que casa com qualquer valor. Os casos especiais de padrão identificador são:

- *elementos de enumeração*, que, assim como valores literais, casam apenas com o próprio elemento;
- *identificadores decorados de nomes de domínio*, que casam apenas com valores do domínio dado; o domínio correspondente é obtido ignorando-se a decoração do identificador e transformando a primeira letra em maiúscula.

O padrão anônimo é representado pelo caractere _ e casa com qualquer valor; o valor associado não é amarrado a identificador algum. Em geral, seu uso é feito em casos default em definições de função.

Por exemplo, considere a função *sum* da Listagem 5.39. Quando a função *sum* é aplicada a dois números inteiros, então o resultado é a soma destes valores; caso contrário, a soma é igual a *error*.

5.4.2.3 Tuplas

Um padrão de tupla casa com valores em um domínio de tupla e são representados pelo nome de construtor e uma sequência de padrões separados por vírgulas e delimitados por parênteses. Por exemplo, considere o trecho de código da Listagem 5.40[11]. A função m calcula o módulo de um número complexo. A função t calcula o argumento de um número complexo e é definido por casos: o primeiro casa apenas com a tupla *C(0,0)*; o segundo casa com qualquer número complexo cujo primeiro elemento seja igual a *0*; a terceira cláusula define o comportamento geral da função.

[11]Nesta definição, se $c = ae^{i\theta}$ então $m\ c = a$ e $t\ c = \theta$.

```
1  C = (Double,Double); // complex numbers
2
3  function m : C -> Double;
4  m C(r1,r2) = sqrt ((sqr r1) + (sqr r2));
5
6  function t : C -> Double;
7  t C(0,0) = 0;
8  t C(0,r) = if r > 0 then pi / 2 else -pi / 2;
9  t C(r1,r2) = atan (r2 / r1)
```

Listagem 5.40: Exemplos de Padrão Tupla em Definições de Função

5.4.2.4 Listas

Padrões de lista casam com listas de elementos e podem ser especificados por meio de padrões de agregados para listas, padrões de identificadores decorados para listas, ou pares cabeça e cauda.

Em um padrão agregado para lista, padrões são listados entre parênteses e separados por vírgula. Tais padrões podem ser quaisquer padrões definidos nesta seção. Por exemplo, considere o trecho de código da Listagem 5.41. A função f mapeia listas de inteiros em listas de inteiros; se o argumento recebido for uma lista vazia, então o resultado é a lista vazia; caso contrário, se o argumento recebido for a lista $(1, 2, 3)$, então o resultado é a lista $(1, 2, 5)$; por outro lado, se a lista contiver três elementos, onde o primeiro é igual a 1, e o último, a 3, então o resultado é $(1, x, 4)$, onde x é o segundo elemento da lista recebida; finalmente, se o argumento for uma lista com quatro elementos, então o resultado é a lista formada pela remoção do terceiro elemento. Do mesmo modo, a função g mapeia listas de listas de inteiros da forma $((1), (), (x, y))$ na lista $(1, x + y)$.

Listas podem também ser definidas por pares cabeça e cauda; a cabeça e a cauda da lista são separadas por um símbolo de dois pontos, que é associativo à direita. A Listagem 5.42 apresenta exemplos de uso deste padrão.

Identificadores decorados como listas servem para associar listas cujos elementos são de um domínio específico. Por exemplo, considere o trecho de código da Listagem 5.43. A função f mapeia listas de elementos do domínio V em listas de valores lógicos, tal que: quando aplicada a listas cujo primeiro elemento é um número inteiro, o resultado é a lista $(true, true)$; quando aplicada a uma lista contendo pelo menos dois valores em que o

```
1    function f : Int* -> Int*
2    f () = ()
3    f (1,2,3) = (1,2,5)
4    f (1,x,3) = (1,x,4)
5    f (a,b,_,c) = (a,b,c)
6
7    function g : Int** -> Int*
8    g ((1), (), (x,y)) = (1,x+y)
```

Listagem 5.41: Exemplos de Padrões de Listas

```
1    function f : Int* -> Int*
2    f () = ()
3    f 3:s = 4 : f s
4    f 1:2:s = 1 : 2 : f s
5    f int:s = int * int : f s
```

Listagem 5.42: Exemplos de Uso de Pares Cabeça e Cauda

primeiro é um valor lógico, então o resultado é a aplicação da função à cauda da lista; caso contrário, o resultado é a lista (*false*).

O padrão $(p_1, p_2, \cdots, p_n)$ é equivalente a $p_1 : p_2 : \cdots : p_n : ()$, e esta notação será utilizada preferencialmente ao longo deste texto.

5.4.2.5 Nós de Árvores de Sintaxe Abstrata

Padrões para nós de árvore de sintaxe abstrata correspondem ao lado direito de regras de produção da gramática abstrata. Por exemplo, considere o trecho de código da Listagem 5.44,

```
1    V = Bool | Int | {nothing}
2
3    function f : V* -> Bool*
4    f int:v* = (true,true)
5    f bool:v+ = f v+
6    f v* = (false)
```

Listagem 5.43: Exemplos de Padrões de Listas

```
1   syntax Exp;
2   semantics e;
3
4   token num : Exp = [0−9]+ is asInteger;
5
6   exp ::= exp "+" term | term : term;
7   term : Exp ::= term "∗" factor | factor : factor;
8   factor : Exp ::= num | "(" exp ")" : exp;
9
10  function e : Exp −> Int;
11  e int = int;
12  e [exp1 "+" exp2] = (e exp1) + (e exp2);
13  e [exp1 "∗" exp2] = (e exp1) ∗ (e exp2);
```

Listagem 5.44: Exemplo de Padrão Para Nó de Árvore de Sintaxe Abstrata

cujas regras da gramática abstrata são:

$$\begin{aligned} Exp &\rightarrow Exp\ \text{"+"}Exp \\ &\mid Exp\ \text{"*"}Exp \\ &\mid Int \end{aligned}$$

A primeira cláusula define o resultado da função e aplicada a um valor inteiro; a segunda cláusula define o resultado da função aplicada a uma expressão de soma; a terceira cláusula define o resultado da função aplicada a uma expressão de multiplicação.

Os elementos de um padrão para nó de árvore de sintaxe abstrata deve concordar com o domínio sintático correspondente, e nenhuma conversão automática é realizada.

5.4.2.6 Dominância de Padrões

Os conceitos de dominância de padrões e de padrões mais restritos para um valor, definidos nesta seção, são necessários à definição das regras de aplicação de funções (ver Seção 5.4.3.6).

Seja $\mathcal{P}$ o conjunto de todos os padrões em uma especificação em Notus. $\mathcal{P}$ é um conjunto parcialmente ordenado com relação $\preceq$, tal que para padrões $p_1, p_2 \in \mathcal{P}$, $p_1 \preceq p_2$ se pelo menos uma das condições a seguir for verdadeira:

1. (fecho reflexivo) $p_1 = p_2$;

2. (fecho transitivo) existe um padrão p_3, tal que $p_1 \preceq p_3$ e $p_3 \preceq p_2$; por exemplo, pelas condições 5 e 4, $2 \preceq \mathit{int}$ e $\mathit{int} \preceq x$, caso não exista domínio X na especificação, logo $2 \preceq x$;

3. p_2 é o padrão anônimo;

4. p_2 é um identificador que não está associado a qualquer domínio;

5. p_1 é um valor literal, e p_2 é um identificador decorado do domínio correspondente a p_1; por exemplo, $21 \preceq \mathit{int1}$;

6. p_1 é um elemento de enumeração, e p_2 é um identificador decorado do domínio de p_1; por exemplo, se a enumeração E for definida como $E = \{a, b, c\}$, então $a \preceq e$;

7. p_1 é um padrão tupla, e p_2 é um identificador decorado do nome do domínio de p_1; por exemplo, se o domínio T for definido como $T = \mathit{Ta}(\mathit{Int}, \mathit{String})$, então $\mathit{Ta}(1, \mathit{"abc"}) \preceq t$;

8. p_1 é um padrão de lista, e p_2 é um identificador decorado como lista de elementos do domínio dos elementos de p_1; por exemplo, $(5, 6, 4) \preceq \mathit{int}*$;

9. existe um padrão p, tal que $p_1 = p+$ e $p_2 = p*$;

10. p_1 é um padrão de nó de árvore de sintaxe abstrata, e p_2 é um identificador decorado do domínio sintático correspondente a p_1; por exemplo, se a especificação definir a regra sintática $\mathit{exp} \rightarrow \mathit{exp}$ "+" exp, e exp denotar o domínio Exp, então $[\mathit{exp1}$ "+" $\mathit{exp2}] \preceq \mathit{exp}$;

11. p_1 é um identificador decorado para o domínio d_1, p_2 é um identificador decorado para o domínio d_2, e d_1 é um subdomínio de d_2; por exemplo, se o domínio V for definido como $V = \mathit{Int} \mid \mathit{Bool}$, então $\mathit{int} \preceq v$;

12. p_1 é o padrão tupla $T(p_{11}, p_{12}, \cdots, p_{1k})$, p_2 é o padrão tupla $T(p_{21}, p_{22}, \cdots, p_{2k})$, para algum k e algum construtor de tupla T, e $p_{1i} \preceq p_{2i}$, para $1 \leq i \leq k$; por exemplo, se o domínio T for definido por $T = \mathit{Ta}(\mathit{Int}, \mathit{String})$, então $\mathit{Ta}(5, \mathit{"abc"}\ ) \preceq \mathit{Ta}(5, \mathit{string})$;

13. p_1 é um par cabeça e cauda $p_{11} : p_{12}$, p_2 é um par cabeça e cauda $p_{21} : p_{22}$, $p_{11} \preceq p_{21}$, e $p_{12} \preceq p_{22}$; por exemplo, $5 : \mathit{int}* \preceq \_ : x$.

Uma sequência de valores $V = \langle v_1, v_2, \cdots, v_n \rangle$ casa com um sequência de padrões $P = \langle p_1, p_2, \cdots, p_n \rangle$ se p_i casar com o valor v_i, para cada i. A relação de ordem parcial $\preceq$

é definida para sequências de padrões $P_1 = \langle p_{11}, p_{12}, \cdots, p_{1k} \rangle$ e $P_2 = \langle p_{21}, p_{22}, \cdots, p_{2k} \rangle$, ambas do mesmo tamanho, tal que $p_{1i} \preceq p_{2i}$ para cada i.

Dado um conjunto de padrões $S = \{p_1, p_2, \cdots, p_m\}$ e um valor v, o padrão mais restrito de v em S é o padrão $p_i \in S$ tal que:

- p_i casa com v;
- $p_i \preceq p_j$, para todo $p_j \in S$ que case com v.

Se o padrão mais restrito de v em S não puder ser determinado, então S é não-aplicável a v.

Seja $V = \langle v_1, v_2, \cdots, v_k \rangle$ uma sequência de valores, e $\mathcal{S} = \{P_1, P_2, \cdots, P_n\}$, um conjunto de sequências de padrões, onde cada $P_i = \langle p_{i1}, p_{i2}, \cdots, p_{ik} \rangle$ é uma sequência contendo k padrões. O padrão mais restrito de V em $\mathcal{S}$ é a sequência $P_i \in \mathcal{S}$ tal que:

- P_i casa com V;
- $P_i \preceq P_j$, para todo $P_j \in \mathcal{S}$ que case com V.

Se o padrão mais restrito de V em $\mathcal{S}$ não puder ser determinado, então $\mathcal{S}$ é não-aplicável a V.

5.4.3 Expressões

As expressões de Notus são:

1. valores literais (Seção 5.4.3.1, página 127);
2. agregados (Seção 5.4.3.2, página 127);
3. identificadores (Seção 5.4.3.3, página 128);
4. operações (Seção 5.4.3.4, página 128);
5. nós de árvore de sintaxe abstrata (Seção 5.4.3.5, página 131);
6. aplicações de função (Seção 5.4.3.6, página 132);
7. padrões (Seção 5.4.3.7, página 133);

8. abstrações lambda (Seção 5.4.3.8, página 134);
9. expressões *let* e *where* (Seção 5.4.3.9, página 134);
10. atualização de função (Seção 5.4.3.10, página 135);
11. condicionais (Seção 5.4.3.11, página 135).

A sintaxe de expressões pode ser encontrada na Seção A.8.

5.4.3.1 Valores Literais

Valores literais em Notus são definidos para cada domínio padrão apresentado na Seção 5.2.3.2:

- valores booleanos são **true** e **false**;
- números inteiros podem ser escritos nas seguintes bases:
  - decimal: o número é uma sequência de um ou mais dígitos de 0 a 9;
  - octal: o número é uma sequência de um ou mais dígitos de 0 a 7, precedida pelo dígito 0;
  - hexadecimal: o número é uma sequência de um ou mais dígitos hexadecimais, i.e. dígitos de 0 a 9 e letras de `a` a `f`, precedida de `0x`;
- números reais são números de ponto flutuante em 64-bits seguindo o padrão IEEE 754;
- caracteres são delimitados por aspas simples, com as seguintes sequências de escape:

  \n, \r, \f, \t, \a, \b, \\, \', and \"

- cadeias de caracteres são sequências de caracteres delimitadas por aspas duplas, com as mesmas sequências de escapes utilizadas em caracteres.

5.4.3.2 Agregados

Em Notus, existem agregados para tuplas e listas.

Agregados Para Tuplas. Um agregado para tupla é composto por um nome de construtor e uma sequência de expressões, separadas por vírgula e delimitada por parênteses. Considerando A, B, C, D e E nomes de construtores, são exemplos de agregados para tuplas:[12]

```
A()  B(true)  C(1,true,3 + 7)
D(false,3.141592,E(a,if b then 10 else 9))
```

Uma expressão $C(e_1, e_2, \cdots, e_n)$ pertence ao domínio no qual o construtor C foi definido.

Agregados Para Listas. Um agregado para lista é uma sequência de expressões separadas por vírgulas e delimitadas por parênteses. São exemplos de agregados para listas:[13]

```
()  (true)  (1,4,3 + 7)
((4,3),(a,if b then 10 else 9))
```

Uma expressão $(e_1, e_2, \cdots, e_n)$ pertence ao domínio d^*, onde d é o domínio de cada e_i. A expressão () pertence ao domínio x^*, para algum domínio x, que deve ser dedutível a partir do contexto.

5.4.3.3 Identificadores

Identificadores em Notus são sequências de letras, dígitos e caracteres _, que iniciam com letra minúscula. Um identificador pode ser qualificado com o nome do módulo onde foi definido e decorado com números ou como uma lista. O domínio de um identificador pode ser automaticamente deduzido alterando sua primeira letra para maiúsculo e removendo suas decorações; se o nome obtido for um identificador de domínio válido, então não é necessário definir o tipo do identificador; caso contrário, o seu tipo deve ser deduzido pelo contexto da expressão.

5.4.3.4 Operações

Notus possui os operadores definidos nas Tabelas 5.5–5.11, que representam as operações aritméticas, bit-a-bit, relacionais, lógicas, de lista, composição e casamento de padrões. Em

[12] A expressão condicional é definida na Seção 5.4.3.11.
[13] A expressão condicional é definida na Seção 5.4.3.11.

Operador	Objetivo	Preced.	Assoc.	Assinaturas
unary `-`	arithmetic negation	2	–	$I \to I$ $R \to R$
`*`	multiplication	3	left	$I \times I \to I$ $R \times R \to R$
`/`	division result	3	left	$I \times I \to I$ $R \times R \to R$
`mod`	division remainder	3	left	$I \times I \to I$
`+`	addition	4	left	$I \times I \to I$ $R \times R \to R$
`-`	subtraction	4	left	$I \times I \to I$ $R \times R \to R$

Tabela 5.5: Operadores Aritméticos de Notus

cada tabela, a primeira coluna lista os operadores, a segunda, seus objetivos, a terceira, suas ordens de precedência, a quarta, suas associatividades, e a quinta, seus domínios. A precedência máxima é igual a 1 e é dada somente a expressões que não envolvam o uso de operadores.

Tabela 5.5 lista os operadores aritméticos básicos para negação, adição, subtração, multiplicação, divisão real, quociente de divisão inteira e resto de divisão inteira. Todos os operadores, com exceção do resto de divisão inteira, são definidos para os domínios de números inteiros (*Byte*, *Short*, *Int* e *Long*), condensados na tabela por I, e para os domínios de números reais (*Float* e *Double*), condensados na tabela por R.

Tabela 5.6 lista os operadores que representam a manipulação de números em binário: negação, deslocamentos, conjunção, disjunção e ou-exclusivo. Estes operadores podem ser aplicados somente aos domínios de números inteiros (*Byte*, *Short*, *Int* e *Long*), condensados na tabela por I.

Tabela 5.7 lista os operadores relacionais que são definidos para os domínios de números inteiros (*Byte*, *Short*, *Int* e *Long*), condensados na tabela por I, e para os domínios de números reais (*Float* e *Double*), condensados na tabela por R.

Tabela 5.8 lista os operadores lógicos para negação, conjunção e disjunção.

Tabela 5.9 lista os operadores para construção e concatenação de listas. Estas operações são definidas para todos os domínios, representados na tabela por a.

Operador	**Objetivo**	**Preced.**	**Assoc.**	**Assinaturas**
`~`	bitwise negation	2	–	$I \to I$
`>>`	arithmetic shift right	3	left	$I \times I \to I$
`>>>`	logical shift right	3	left	$I \times I \to I$
`<<`	shift left	3	left	$I \times I \to I$
`&`	bitwise conjunction	3	left	$I \times I \to I$
`|`	bitwise disjunction	4	left	$I \times I \to I$
`^`	bitwise exclusive-or	4	left	$I \times I \to I$

Tabela 5.6: Operadores Bit-a-bit de Notus

Operador	**Objetivo**	**Preced.**	**Assoc.**	**Assinatura**
`<`	less than	5	–	$R \times R \to Bool$ $I \times I \to Bool$
`>`	greater than	5	–	$R \times R \to Bool$ $I \times I \to Bool$
`<=`	less than or equal to	5	–	$R \times R \to Bool$ $I \times I \to Bool$
`>=`	greater than or equal to	5	–	$R \times R \to Bool$ $I \times I \to Bool$
`=`	equal to	5	–	$R \times R \to Bool$ $I \times I \to Bool$
`!=`	not equal to	5	–	$R \times R \to Bool$ $I \times I \to Bool$

Tabela 5.7: Operadores Relacionais de Notus

Operador	**Objetivo**	**Prec.**	**Assoc.**	**Assinaturas**
`not`	boolean negation	2	–	$Bool \to Bool$
`and`	boolean conjunction	6	left	$Bool \times Bool \to Bool$
`or`	boolean disjunction	7	left	$Bool \times Bool \to Bool$

Tabela 5.8: Operadores Lógicos de Notus

Operador	Objetivo	Preced.	Assoc.	Assinaturas
:	list construction	8	right	$a \times a* \rightarrow a*$
++	list appending	9	left	$a* \times a* \rightarrow a*$

Tabela 5.9: Operadores de Lista de Notus

Operador	Objetivo	Preced.	Assoc.	Assinatura
.	function composition	10	right	$(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

Tabela 5.10: Operador de Composição de Funções de Notus

Tabela 5.10 define o operador de composição de função.

Tabela 5.11 define o operador de casamento de padrões.

5.4.3.5 Nós de Árvore de Sintaxe Abstrata

Uma expressão para nó de árvore de sintaxe abstrata corresponde ao lado direito de uma regra de produção da gramática abstrata e é utilizada para definição das funções semânticas. A expressão é delimitada por colchetes e é formada por uma sequência de identificadores, cadeias de caracteres, identificadores de lista e outros nós de árvore de sintaxe abstrata.

Por exemplo, considere o trecho de código na Listagem 5.45, que mostra como remover açúcar sintático de uma gramática de comandos de repetição *while* e *do-while*, definidos como na linguagem C, o que possibilita que somente o comando *while* seja apresentado nas equações semânticas. A gramática original é:

$$
\begin{aligned}
\mathit{Com} \;\; \rightarrow \;\; & \text{"}\mathit{while}\text{" "(" } \mathit{Exp} \text{ ")" } \mathit{Com} \\
| \;\; & \text{"}\mathit{do}\text{" } \mathit{Com} \text{ "}\mathit{while}\text{" "(" } \mathit{Exp} \text{ ")"} \\
| \;\; & \text{"\{" } \mathit{Com} \text{ ";" } \mathit{Com} \text{ "\}"}
\end{aligned}
$$

A função *desugar* recebe como argumento um nó que representa um comando e retorna uma nova expressão correspondente sem o açúcar sintático. Os elementos de uma expressão de

Operador	Objetivo	Preced.	Assoc.	Assinatura
`is`	pattern matching	11	–	$a \rightarrow b \rightarrow \mathit{Bool}$

Tabela 5.11: Operador de Casamento de Padrões de Notus

```
1   com ::= "while" "(" exp ")" com
2         |  "do" com "while" "(" exp ")"
3         |  com ";" com
4         |  ... // other clauses
5
6   function desugar : Com -> Com;
7   desugar [ "do" com "while" "(" exp ")" ] =
8               let com1 = desugar com
9               in  [ com1 ";" [ "while" "(" exp ")" com1 ] ]
10  ... // other clauses
```

Listagem 5.45: Exemplos de Nós de Árvore de Sintaxe Abstrata

nó deve concordar com os domínios sintáticos declarados, e nenhuma conversão automática é realizada.

5.4.3.6 Aplicações de Função

Aplicações de função possuem a forma

$$e_0\ e_1\ e_2\ \cdots\ e_n$$

onde cada e_i é uma expressão. Se para cada $i \geq 1$, o valor de e_i pertencer ao domínio d_i, então e_0 deve pertencer a algum domínio $d'_1 \to d'_2 \to \cdots \to d'_n \to d$, tal que $d_i = d'_i$ ou d'_i é subdomínio de d_i; neste caso, a expressão $e_0\ e_1\ e_2\ \cdots\ e_n$ pertence ao domínio d; caso contrário, o compilador gera um erro.

Para uma função definida em casos, a definição utilizada em uma aplicação depende dos valores dos argumentos fornecidos. Sejam: f uma função no domínio $d_1 \to d_2 \to \cdots \to d_n \to d$, definida por meio de m cláusulas, cada uma com sequência de padrões $P_k = \langle p_{k1}, p_{k2}, \cdots, p_{kn} \rangle$, para $1 \leq k \leq m$; $e_1, e_2, \cdots, e_n$ expressões; $f\ e_1\ e_2\ \cdots\ e_n$ uma aplicação de função. Se o valor de avaliação de cada e_i for v_i, então a aplicação de função é avaliada com o corpo de função definido para o padrão mais restrito (ver Seção 5.4.2.6) de $V = \langle v_1, v_2, \cdots, v_n \rangle$ em $\mathcal{S} = \{P_1, P_2, \cdots, P_m\}$. Se $\mathcal{S}$ não for aplicável a V, então o compilador deve gerar um erro para a aplicação de função.

Parênteses em aplicações de função, quando exigidos para o último argumento da aplicação, podem ser omitidos pelo uso de expressões de *evaluate*, que possuem a forma:

$$\texttt{evaluate}\ \{\ \mathit{function};\ \mathit{arg}_1;\ \mathit{arg}_2;\ \cdots;\ \mathit{arg}_{n-1};\ \mathit{arg}_n\ \}$$

```
1    Value = Int | Bool | {error};
2
3    function f : Value -> Int
4    f value = if value is int then 1
5              else if value is bool then 2
6              else 3
```

Listagem 5.46: Exemplo de Expressão de Casamento de Padrão

e é equivalente a: *function* (arg_1 (arg_2 ($\cdots$ (arg_{n-1} (arg_n))))). Por exemplo, a expressão `evaluate { fun a b; c d; e f }` é equivalente a `fun a b (c d (e f))`.

Chaves e pontos-e-vírgulas em expressões *evaluate* podem ser omitidos por meio das regras de leiaute definidas na Seção A.1. Por exemplo, pelo uso destas regras, a expressão anterior pode ser escrita como `evaluate fun a b; c d; e f` .

Nota: A expressão *evaluate* é também equivalente a $(function \circ arg_1 \circ arg_2 \circ \cdots \circ arg_{n-1})\ arg_n$, e, desta forma, o exemplo pode ser ainda escrito como ((`fun a b`) $\circ$ (`c d`)) (`e f`).

5.4.3.7 Padrões e Expressões de Casamento de Padrão

Em Notus, as seguintes expressões podem ser compostas pelos padrões da Seção 5.4.2: expressões *let* e *where*; expressão *case*; expressão de casamento de padrões. Uma expressão de casamento de padrão é utilizada para verificar se o valor de uma expressão casa com um dado padrão e possui a forma:

expression `is` *pattern*

Expressões de casamento de padrão não definem amarrações de identificadores. Portanto, os identificadores definidos em *pattern* não estão disponíveis fora da expressão de casamento de padrão. O operador ***is*** não é associativo e sua precedência é definida de acordo com as regras da Seção 5.4.3.4.

A função f definida na Listagem 5.46 utiliza uma expressão de casamento de padrão para verificar se um dado valor é inteiro, lógico ou se representa um erro.

5.4.3.8 Abstrações Lambda

Abstrações lambda em Notus possuem a forma $\backslash\ p_1\ p_2\ \cdots\ p_n \rightarrow e$, onde cada p_i é um padrão e e é uma expressão. O domínio de uma abstração é obtido a partir do contexto no qual se encontra. Por exemplo a expressão

```
\int1 int2 -> (int1 + int2, int1 - int2, int1 mod int2)
```

pertence ao domínio $Int \rightarrow Int \rightarrow Int*$.

5.4.3.9 Expressões *Let* e *Where*

Expressões *let* e *where* são utilizadas para definir sub-expressões auxiliares em expressões mais complexas. Uma expressão *let* possui a forma:

$$\textbf{let}\ \{\ p_1 = e_1; p_2 = e_2; ...; p_n = e_n\ \}\ \textbf{in}\ e$$

onde cada p_i é um padrão, e cada e_i e e são expressões. O valor desta expressão é o valor de e avaliado em um ambiente onde o valor de cada e_i casa com p_i. As sub-expressões podem ser mutuamente recursivas e, por isso, cada e_i pode utilizar identificadores definidos em qualquer p_j.

Chaves e pontos-e-vírgulas em expressões *let* podem ser omitidos por meio das regras de leiaute definidas na Seção A.1.

O exemplo na Listagem 5.47 define funções f e g por meio de expressões *let*. A função f recebe como argumento um número x e retorna o cosseno hiperbólico de x; a expressão *let* é utilizada para simplificar a expressão, amarrando o valor de e^x a *e1* e o valor de e^{-x} a *e2*. A função g recebe como argumento dois números inteiros, a e b, e tem como resultado uma lista infinita que intercala os valores recebidos na forma $\langle a, b, a, b, a, b, \cdots\rangle$.

Uma expressão *where* da forma:

$$e\ \textbf{where}\ \{\ p_1 = e_1; p_2 = e_2; ...; p_n = e_n\ \}$$

é equivalente à seguinte expressão *let*:

$$\textbf{let}\ \{\ p_1 = e_1; p_2 = e_2; ...; p_n = e_n\ \}\ \textbf{in}\ e$$

O exemplo da Listagem 5.48 define as mesmas funções da Listagem 5.47, utilizando expressões *where* em vez de expressões *let*.

```
1    function f : Real -> Real
2    f x = let e1 = exp(x); e2 = 1.0 / e1 in (e1 + e2) / 2
3
4    function g : Int -> Int -> Int*
5    g a b = let list1 = a:list2
6                list2 = b:list1
7            in list1
```

Listagem 5.47: Exemplos de Expressões *Let*

```
1    function f : Real -> Real
2    f x = (e1 + e2) / 2 where e1 = exp(x); e2 = exp(-x)
3
4    function g : Int -> Int -> Int*
5    g a b = list1 where list1 = a:list2
6                        list2 = b:list1
```

Listagem 5.48: Exemplos de Expressões *Where*

5.4.3.10 Atualização de Funções

Expressões de atualização de função são utilizadas para alterar o valor de uma função em pontos específicos e possuem a forma:

$$function[argument \leftarrow new\text{-}result]$$

O resultado desta expressão é uma nova função que, quando aplicada ao valor *argument* produz como resultado o valor *new-result*, e, quando aplicada a qualquer outro valor produz o mesmo valor que *function*.

Considere as funções *fac* e *g* definida na Listagem 5.49; a função *fac* é a função fatorial, e a função *g* é definida por:

$$g\ x = \begin{cases} 5, \text{ if } x = 1 \\ fac\ x, \text{ otherwise} \end{cases}$$

5.4.3.11 Expressões Condicionais

Expressões condicionais em Notus podem ser definidas na forma de expressões *if* e expressões *case*.

```
1  function fac : Int -> Int
2  fac 0 = 1
3  fac x = x * fac(x - 1)
4
5  function g : Int -> Int
6  g = fac[1 <- 5]
```

Listagem 5.49: Exemplo de Atualização de Função

Expressões *If*. A expressão *if* possui a forma:

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$

onde e_1 é uma expressão cujo valor pertence ao domínio *Bool*, e e_2 e e_3 são expressões pertencentes a um mesmo domínio. O valor da expressão *if* é e_2 se e_1 avaliar em *true*, ou e_3, caso contrário.

Expressão *Case*. A expressão *case* possui a forma:

$$\begin{array}{l}\textbf{case } e \textbf{ of } \{\\ \quad p_1 \Rightarrow e_1;\\ \quad p_2 \Rightarrow e_2;\\ \quad \vdots\\ \quad p_n \Rightarrow e_n\\ \}\end{array}$$

onde e é uma expressão, cada p_i é um padrão, e cada e_i é uma expressão. O valor desta expressão *case* é e_i se p_i for o padrão mais restrito do valor de e em $S = \{p_1, p_2, \cdots, p_n\}$ (ver Seção 5.4.2.6). Se S não for aplicável ao valor de e, então o compilador lança um erro. Casos *default* são obtidos por meio do padrão anônimo. Chaves e pontos-e-vírgulas em expressões *case* podem ser omitidos por meio das regras de leiaute definidas na Seção A.1.

A Listagem 5.50 define uma função f como o inverso multiplicativo de seu argumento, x, se $x \neq 0$; se $x = 0$ então f resulta em um erro. A função g é equivalente à função f utilizando expressão *case* em vez de expressão *if*.

```
1    function f : Real -> ({error} | Real)
2    f x = if x = 0 then error else 1 / x
3
4    function g : Real -> ({error} | Real)
5    f x = case x of 0 => error
6                    _ => 1 / x
```

Listagem 5.50: Exemplos de Expressões Condicionais

5.5 Funções de Transformação

Uma função de transformação, ou simplesmente transformador, é definido em um módulo de transformação, que possui o formato a seguir:

transformation *transform-name*
import *imported-modules*
signature *signature-clauses*
default *default-clauses*
replace *replace-clauses*
redefine *redefine-clauses*

As cláusulas do módulo tem estrutura e funcionamento conforme especificado na Seção 4.3. A sintaxe do módulo pode ser encontrada nas Seções A.2 e A.9. A importação de módulos deve ser utilizada para tornar visíveis identificadores e domínios envolvidos nas cláusulas de transformação e funções auxiliares utilizadas nas expressões das cláusulas de transformação.

Detalhes das funções de transformação foram apresentados na Seção 4.3.

5.6 Especificação Baseada em Componentes

Notus permite, ainda, a especificação de linguagens baseadas em componentes, conforme proposto por *Mosses* (2005). Nesta proposta, define-se um núcleo de construções básicas abstratas de linguagens de programação em geral, de modo que qualquer elemento linguístico se torna uma combinação de elementos mais simples. A especificação da semântica de uma linguagem consiste, desta maneira, na tradução de suas construções nas construções

da linguagem base, cuja semântica formal foi previamente definida. Por exemplo, os comandos de repetição *while*, *do-while* e *for* da linguagem C podem ser definidos por meio da combinação de sequências de comandos e de um comando de repetição básico.

Em Notus, o processo para componentização consiste, inicialmente, na definição de módulos para as construções da linguagem base, contendo a definição de seus domínios semânticos e das equações semânticas correspondentes. A definição da nova linguagem é feita por meio da especificação de seus elementos léxicos e sintáticos, como qualquer outra definição em Notus. As equações semânticas, no entanto, são funções de tradução dos elementos no domínio sintático da nova linguagem em nós de árvore sintática da linguagem básica. A função de interpretação consiste, assim, na composição da função de tradução com a função de interpretação da linguagem base. É importante notar que não é necessário definir sintaxe concreta para a linguagem base.

5.7 Compilador de Notus

O compilador para a linguagem Notus foi concebido e projetado pelo autor e implementado como dois trabalhos do Laboratório de Linguagens de Programação do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais:

- *Soares* (2007) implementou a toda a estrutura geral do compilador, incluindo geração de analisador léxico, geração de analisador sintático, e geração das rotinas semânticas;
- *Loredo et al.* (2008) estendeu o compilador, incluindo as funções de transformação e a compilação de sequências de transformadores.

Além disso, o autor contribuiu na co-orientação destes projetos durante o seu desenvolvimento. A estrutura geral do compilador é apresentada na Figura 5.2. O programa foi implementado em AspectJ e possui aproximadamente 30.000 linhas de código.

A compilação de uma especificação S é definida em três etapas. A primeira, de geração da árvore de sintaxe abstrata e povoamento da tabela de símbolos, é executada recursivamente da seguinte maneira:

- se S for uma especificação inicial, então a compilação:
    1. inicia as estruturas de dados da árvore de sintaxe abstrata da especificação e a tabela de símbolos;

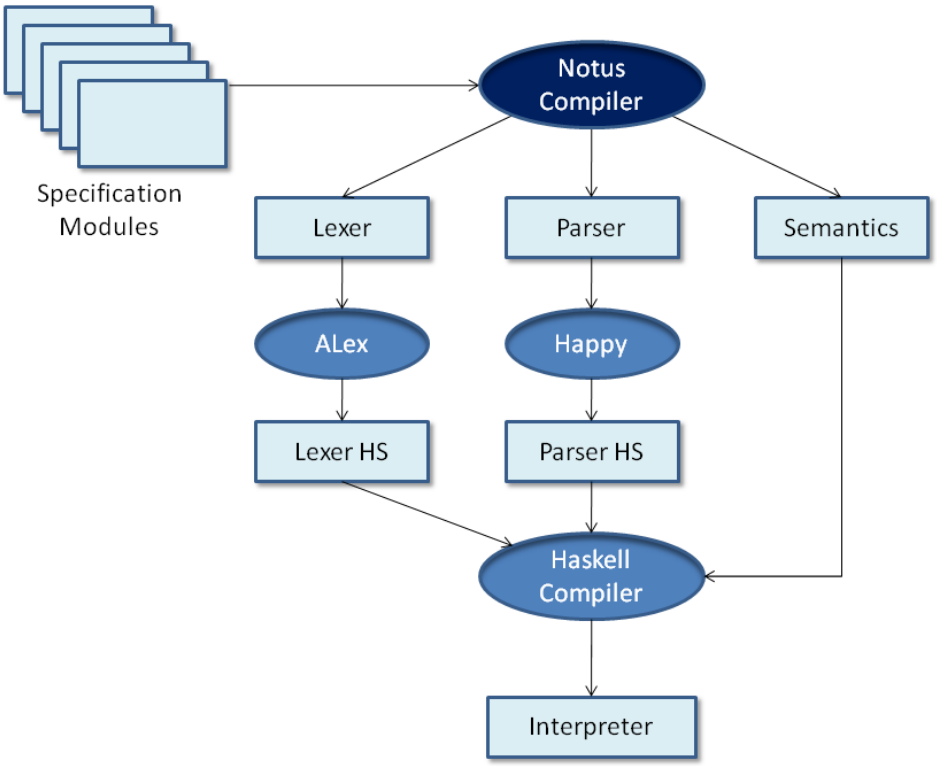

Figura 5.2: Estrutura Geral do Compilador de Notus.

2. lê todos os módulos importados pelo módulo principal, armazenando as informações coletadas nas estruturas de dados;
3. resolve pendências devidas a identificadores definidos em ordem diferente à de leitura do compilador.

- se S for uma extensão da forma $\Delta S + t(S')$, então a compilação:

1. compila recursivamente a especificação S';
2. compila o transformador t e aplica-o à árvore de sintaxe abstrata de S';
3. compila os módulos correspondentes a ΔS, incluindo novos nós à árvore de sintaxe abstrata transformada e novas informações na tabela de símbolos;
4. resolve pendências devidas a identificadores definidos em ordem diferente à de leitura do compilador.

A segunda etapa realiza as seguintes tarefas:

1. resolução das ambiguidades da definição léxica, de acordo com as regras da Seção 5.3.3;
2. definição da estrutura da gramática abstrata e dos domínios sintáticos da especificação, segundo as regras da Seção 5.3.4.3 e o algoritmo da Seção C.3;
3. simplificação das gramáticas concreta e abstrata, de acordo com os algoritmos das Seções C.1 e C.2;
4. definição da ordem de geração dos casos das funções semânticas, por meio do algoritmo de ordenação topológica.

Por fim, a terceira etapa faz a geração de código da seguinte maneira:

1. o analisador léxico gerado é um arquivo de entrada para o sistema *Alex* [14];
2. o analisador sintático gerado é um arquivo de entrada para o sistema *Happy* [15];
3. as demais funções são compiladas para a linguagem Haskell.

[14] Ver `http://www.haskell.org/alex/`
[15] Ver `http://www.haskell.org/happy/`

Para obtenção do interpretador executável, deve-se gerar código Haskell para os analisadores léxico e sintático, utilizando software disponibilizado pelos fabricantes, e, finalmente, utilizar o compilador de Haskell para geração do código executável. Módulos com implementação em Haskell domínios e funções padrão são incluídos durante a geração do código executável.

Extensões na linguagem Notus podem ser feitas por meio da criação de bibliotecas de domínios e funções. Para incluí-las no compilador, basta editar arquivos de configuração que indiquem o que será adicionado a Notus e a implementação correspondente em Haskell. Detalhes podem ser encontrados em *Soares* (2007).

5.8 Conclusões

A linguagem Notus, apresentada neste capítulo, tem por objetivo permitir a definição incremental completa de uma linguagem de programação. A linguagem permite a divisão de uma definição em especificações, que podem por sua vez ser divididas em pacotes e módulos. Para permitir alto grau de modularidade, foram incorporados recursos para controle de visibilidade. Exemplos do uso dos recursos de Notus para definições de linguagens podem ser encontradas no Apêndice B.

A implementação corrente de Notus permite a geração de interpretadores executáveis para a linguagem projetada. A partir da definição incremental, geram-se o analisador léxico, o analisador sintático e as funções semânticas. Assim, o interpretador gerado lê um arquivo fonte na linguagem, faz análises léxica e sintática, gerando uma árvore de sintaxe abstrata para o programa; esta árvore de sintaxe abstrata é, então, argumento das funções semânticas que, ao serem executadas, geram a saída do programa a partir da entrada fornecida.

O analisador léxico é gerado automaticamente a partir das declarações de tokens e de não-terminais anônimos presentes na gramática concreta. A ordem de definição dos elementos é irrelevante, o que torna possível que elementos léxicos sejam definidos em módulos separados. Ambiguidades podem ser resolvidas por meio de regras definidas na linguagem.

O analisador sintático é gerado automaticamente a partir das declarações de regras de produção da gramática concreta da linguagem. As regras de sintaxe concreta e abstrata são ainda utilizadas para a definição da estrutura da árvore de sintaxe abstrata, que também é totalmente gerada pelo compilador da linguagem. As regras semânticas do

analisador sintático gerado constroem a árvore de sintaxe abstrata a partir do programa fonte. Na versão atual, o parser gerado é LALR(1), mas versões futuras poderão adotar outras técnicas mais poderosas, tais como Generalized LR (*Bravenboer and Visser* 2009).

As funções semânticas são geradas a partir das equações semânticas e operam sobre a árvore de sintaxe abstrata do programa fonte para computar, a partir da sequência de entrada, a sequência de saída do programa. A linguagem para escrita das equações baseia-se nos recursos mais comuns de linguagens funcionais típicas, com recursos adicionais para tratamento de árvore de sintaxe, dentre os quais se destacam uma notação para os nós da árvore e casamento de padrões sintático.

Em uma especificação, podem-se adicionar novas construções por meio da extensão de elementos léxicos, de gramáticas concreta e abstrata, e da adição de novas equações semânticas. Com isso, pode-se obter incrementabilidade tanto de definições sintáticas quanto semânticas.

A definição incremental baseada em componentes pode ser feita por meio da criação, em módulos separados, de domínios sintáticos específicos com estrutura definida de acordo com a notação para nós de árvore de sintaxe abstrata existente. A transformação de um programa em um conjunto simplificado de construções pode, então, ser feita por meio de funções que mapeiem o universo sintático do programa no universo sintático das construções básicas.

Optou-se neste trabalho por desenvolver uma nova linguagem, em vez de estender uma linguagem funcional já existente, como Haskell. O principal motivo desta decisão foi a necessidade em se definir um conjunto de construções mais próximo das equações encontradas na literatura da área, permitindo a escrita de código mais legível. Também, o uso de uma linguagem de propósito geral traria dificuldades por obrigar o programador a implementar a parte sintática nesta linguagem.

Além disso, a existência de um compilador completamente desenvolvido dentro do grupo de pesquisa deu mais flexibilidade ao desenvolvimento deste projeto.

O compilador implementado gera código na linguagem Haskell, que pode ser compilador para qualquer máquina alvo para a qual haja compilador Haskell disponível. As similaridades entre as construções e estruturas de Notus e Haskell permitiram que o *overhead* da tradução fosse $O(1)$, de modo que o código gerado a partir de uma especificação Notus tende a ter eficiência similar àquela obtida com o uso direto da linguagem Haskell.

O próximo capítulo apresenta uma avaliação deste trabalho de tese, por meio de análise qualitativa da técnica proposta e uma comparação com trabalhos relacionados.

Capítulo 6

Avaliação

Organização do Capítulo. A Seção 6.1 apresenta uma avaliação da metodologia e da linguagem Notus em relação aos critérios descritos na Seção 3.1. A Seção 6.2 apresenta uma comparação deste trabalho com trabalhos relacionados. E, por fim, a Seção 6.3 apresenta as conclusões do capítulo.

6.1 Avaliação da Metodologia

Para análise da metodologia, serão utilizados os seguintes critérios, sugeridos por *Gayo* (2002), *Mosses* (1988), *Moura* (1996) e descritos na Seção 3.1: ausência de ambiguidade, possibilidade de demonstração, prototipação, modularidade, reusabilidade, legibilidade, flexibilidade, escalabilidade, abstração e comparação.

Ausência de Ambiguidade. Cada uma das especificações pertencentes à definição incremental de uma linguagem podem ser vistas como definições denotacionais completas para um subconjunto da linguagem. A cada especificação, a semântica de algumas construções linguísticas pode ser vaga em relação a recursos ainda não especificados, o que não implica em ambiguidade ou incompleteza, pois compreende-se que a linguagem esteja totalmente definida para o subconjunto em questão.

Possibilidade de Demonstração. A técnica apresentada é baseada na transformação das denotações das construções linguísticas, de modo que a capacidade em se demonstrar resultados depende diretamente da teoria subjacente utilizada para especificação. Por ex-

emplo, para transformações de especificações utilizando semântica denotacional clássica, a base de demonstração é a Teoria de Domínios de Scott (*Scott* 1976); para especificações utilizando semântica monádica, a base é a teoria de mônadas definida por *Moggi* (1989). Demonstrações de propriedade podem ser feitas de modo semelhante ao sugerido por *Gurevich* (1994), interpretando a especificação inicial com o *ground model* e cada passo de transformação como uma reificação da especificação anterior. Assim, demonstrações no *ground model* são mais simples, em função do menor número de detalhes a tratar, e, a cada transformação, deve-se verificar quais propriedades da especificação anterior se mantêm válidas.

Prototipação. Utilizando-se a linguagem Notus e o ambiente de execução desenvolvido, é possível realizar a prototipação de uma linguagem, definindo-se tanto os seus elementos sintáticos quanto os semânticos. A partir de uma definição semântica escrita em Notus, é possível obter um interpretador completo para a linguagem, que pode ser utilizado para testes e provas de conceito.

Modularidade. A linguagem de especificação possui o conceito de encapsulamento em módulos com mecanismos de controle de visibilidade. As transformações permitem o isolamento de módulos das especificações iniciais da linguagem, de modo que a denotação de uma construção pode ser vista de maneira atômica a partir de especificações subsequentes. Assim, cada módulo pode exportar somente os símbolos terminais e não-terminais da gramática concreta, a estrutura da sintaxe abstrata, as funções semânticas e funções auxiliares.

Com exceção da estrutura da sintaxe abstrata, todos os demais elementos podem ser vistos como átomos da especificação, sendo que as equações semânticas e as regras de produção da gramática concreta são inacessíveis a outros módulos. Regras de produção da sintaxe abstrata da linguagem, no entanto, são públicas, mas não podem ser manipuladas por outros módulos; as únicas operações permitidas a um módulo são incluir novas regras à sintaxe abstrata existente e a consulta à sua estrutura. O grau de coesão nas definições pode ser aumentado uma vez que, ao ser possível remover o entrelaçamento de código, as equações de um módulo podem ser organizadas de modo a definirem uma única funcionalidade – nível de coesão funcional. Quanto ao acoplamento entre módulos, a escrita de equações simples tende a tornar cada módulo mais independente, sendo assim possível obter níveis mais baixos de acoplamento.

A interface de um módulo pode ser no futuro gerada por meio de uma ferramenta de apoio para ser apresentada ao leitor. Apesar de funções de transformação alterarem a interface de um módulo, a leitura de módulos de especificações iniciais pode ser feita na especificação original e não na alterada; com isso, a alteração da interface não traz dificuldades na extração da interface dos módulos.

Além disso, o raciocínio modular é garantido, uma vez que as transformações possíveis foram definidas de tal forma que o entendimento de uma especificação independe das especificações posteriores.

Reusabilidade. A definição de módulos simples, em especial nas especificações iniciais da linguagem, permitem o isolamento da definição de construções, de modo que estas podem ser reutilizadas na escrita de especificações de linguagens correlatas. Além disso, a organização modular da linguagem de definição, com recursos de importação de elementos públicos e ocultação de dados privados permitem a definição de módulos independentes, que tendem a ser mais facilmente reaproveitados.

Legibilidade. A utilização da vagueza tende a tornar as especificações mais simples, pois permite que o leitor isole os elementos de maior relevância no momento, não tendo que se preocupar com itens intrusos nas equações. A leitura de uma especificação denotacional tende a exigir conhecimentos prévios da nomenclatura utilizada. No entanto, o uso de uma teoria subjacente mais arrojada, como a semântica monádica, pode permitir a simplificação das equações, tornando-as mais simples de se compreender.

Flexibilidade. Como a teoria desenvolvida se baseia em semântica denotacional clássica, fornecendo subsídios para a escrita separada de definições, não há restrição à definição de recursos que podem ser definidos por meio da semântica denotacional. O uso de semântica de mônadas como teoria subjacente pode permitir a inclusão de recursos não bem tratados pela semântica denotacional, tais como não-determinismo e concorrência.

Escalabilidade. A metodologia pode ser utilizada para a definição de linguagens de grande porte, a exemplo do trabalho realizado por *Crepalde et al.* (2008a), em que a definição da parte sequencial de Java foi feita de maneira incremental.

Abstração. Altos níveis de abstração podem ser obtidos pelo uso adequado de mecanismos de módulos e controle de visibilidade. No entanto, na presença do entrelaçamento de construções, torna-se muito complicado entender cada denotação de maneira independente, de modo que o leitor pode se ver obrigado a tratar grandes sistemas de equações altamente relacionadas. Ao se prover melhor separação das equações, diminui-se o número de elementos a serem levados em consideração ao se entender um módulo. Além disso, a vagueza das definições iniciais é um importante fator para a melhoria do grau de abstração, por permitir a compreensão de construções sem a necessidade de se entender como ela se relaciona a conceitos ainda não estudados.

No contexto de semântica denotacional, o termo abstração também se refere ao fato de que a denotação de uma construção é uma função somente das denotações de seus constituintes. As funções de transformação são meras substituições textuais das equações existentes, de modo que a sua aplicação não viola a abstração já existente na definição.

Comparação. Este critério não foi analisado neste trabalho, mas acredita-se que a vagueza pode auxiliar também na comparação entre recursos linguísticos, uma vez que permite a análise de equações mais simples.

6.2 Comparação com Trabalhos Relacionados

Vários trabalhos importantes abordam o problema de modularidade de definições semânticas, com destaque para os trabalhos citados no Capítulo 3.

O modelo de Máquinas de Estado Abstratas (*Gurevich* 1994, 1995), descrito na Seção 3.1.2.2, permite a definição incremental de linguagens de programação, como mostrado por *Börger and Schulte* (1998) na definição da linguagem Java. A principal distinção desse trabalho com o trabalho aqui apresentado está no fato de que aquela técnica permite que a definição seja construída de maneira incremental, mas o resultado final, apresentado ao leitor, é um conjunto de regras ainda entrelaçadas.

A Semântica Operacional Estruturada Modular (*Mosses* 2004), descrita na Seção 3.1.2.3, é uma forma elegante de se obter a semântica modular de uma linguagem de programação. No entanto, conforme já discutido no momento em que esta técnica foi apresentada neste texto, certos recursos linguísticos exigem a definição da semântica operacional em pequenos passos, o que pode levar à necessidade de um número elevado de regras de transição. Vale

ressaltar que esta questão está intrinsicamente relacionada ao modelo de semântica operacional estruturada subjacente, e o mesmo não ocorre no modelo de semântica denotacional. Com efeito, no modelo proposto, existe apenas uma equação que define a semântica de uma construção, e esta equação evolui com o intuito de acomodar novos recursos.

A Semântica de Ações (*Mosses* 1977, 1993, *Mosses and Watt* 1987), descrita na Seção 3.2.1, permite a escrita de equações legíveis, muito próximas a descrições em linguagem natural. O trabalho aqui apresentado aborda o problema de entrelaçamento de definições, que não é tratado em semântica de ações. Além disso, a técnica proposta pode ser adaptada para utilização em semântica de ações.

A semântica de mônadas (*Moggi* 1989, *Wadler* 1990), apresentada na Seção 3.2.2, leva a melhoria das equações do modelo denotacional, ao permitir a abstração de conceitos comuns em definições denotacionais, tais como *stores*, *environments* e continuações. O aspecto incremental da técnica é obtido por meio da evolução da mônada subjacente a uma definição por meio de transformadores de mônadas (*Liang et al.* 1995). No entanto, o nível de abstração obtido não permite ainda a definição separada de construções relacionadas, como pode ser percebido na definição da semântica monádica de AspectJ, apresentada por *Wand et al.* (2004). O trabalho aqui descrito é uma solução viável para o problema do entrelaçamento, podendo inclusive ser utilizado em conjunto com a semântica monádica para a obtenção de definições mais legíveis.

A semântica de ações com mônadas (*Wansbrough and Hamer* 1997), descritas na Seção 3.2.3, é uma técnica que modulariza a definição da notação de ações, não abordando a forma como as construções são definidas. Por isso, valem as mesmas conclusões a que se chegou ao comparar a metodologia proposta com a semântica de ações.

A técnica proposta por *de Moor et al.* (2000), descrita na Seção 3.2.4, utiliza os conceitos da orientação por aspectos para melhorar a modularidade de gramáticas de atributos, podendo ser aplicada a definições denotacionais. No entanto, as técnicas propostas têm como foco modularizar a comunicação entre construções que se relacionem por meio da árvore de sintaxe abstrata do programa. Este trabalho melhora as contribuições de *de Moor et al.* (2000), uma vez que permite tratar relações entre construções que não estejam diretamente associadas na árvore de sintaxe abstrata, como ocorre com definição e chamadas de métodos.

Por fim, a abordagem construtiva (*Mosses* 2005), descrita na Seção 3.2.5, permite alcançar altos níveis de modularidade ao introduzir o conceito de *componente* no contexto de definições semânticas. No entanto, não há mecanismos para evitar entrelaçamento de

definições para construções relacionadas durante a tradução das construções de uma linguagem em combinações de construções de uma linguagem base. O trabalho proposto pode ser combinado com as contribuições desta abordagem construtiva, permitindo que as equações de tradução possam evoluir, evitando, assim, o entrelaçamento.

6.3 Conclusões

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

Este trabalho complementa as contribuições de trabalhos anteriores, sendo possível, em algumas situações 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 deste trabalho de tese, as principais contribuições e possíveis trabalhos futuros.

Capítulo 7

Conclusões

Esta tese apresentou uma nova abordagem para melhorar a modularidade de especificações de semântica denotacional. Esta técnica, baseada nas propriedades de vagueza das definições iniciais, transformação de especificação e evolução de denotações, pode melhorar a legibilidade de equações semânticas por meio da separação de preocupações em construções linguísticas que se entrecortam. Em uma definição incremental, é possível incluir novas construções sem a necessidade de reescrever equações previamente definidas, mesmo nos casos em que uma construção deve preparar contexto para outras. Assim, o objetivo do modelo proposto é permitir que as construções de uma linguagem sejam apresentadas de maneira mais intuitiva, o que é mais próximo às descrições em linguagem natural: cada construção pode ser isolada para melhor compreensão.

A metodologia proposta para definição incremental fornece mecanismos simples para a definição da semântica de linguagens de programação. A definição de uma construção pode ser vaga em relação a outras, o que torna mais simples compreender sua semântica, visto que as relações entre construções são definidas separadamente e, portanto, não tendem a atrapalhar o entendimento. Apesar de a vagueza ser um recurso essencial em definições informais, ela não pode ser controlada neste contexto, de modo que definições incompletas podem parecer vagas. No entanto, quando aplicada a definições formais, a vagueza ajuda a construir a ponte com definições informais, levando a melhor legibilidade. Além disso, esta técnica pode ser utilizada juntamente com outras abordagens de modularidade de semântica denotacional, como por exemplo os recursos da semântica monádica, beneficiando-se de seus aspectos positivos.

Um problema muito complicado de se tratar na transformação de semântica denotacional é

a preservação da abstração da definição semântica. Esta propriedade é preservada pois todas as transformações requerem somente substituições textuais para se gerar a especificação costurada. Assim as funções de transformação não violam a abstração das equações existentes, de modo que a denotação de uma construção depende exclusivamente da denotação de seus constituintes.

Além disso, extensibilidade é melhorada por meio da separação de preocupações provida pela vagueza da especificação. Altos graus de extensibilidade só são obtidos em sistemas de software quando módulos são simples e independentes, pois com isso torna-se mais fácil lidar com modificações *Meyer* (1997). Módulos definidos por meio da abordagem proposta tendem a manipular um conjunto mínimo de informações, com impacto direto na qualidade geral da modularidade.

Organização do Capítulo. A Seção 7.1 indica as contribuições deste trabalho, e a Seção 7.2 discute possíveis trabalhos futuros.

7.1 Contribuições do Trabalho

A principal contribuição deste trabalho de tese é a definição da metodologia de semântica incremental, que se baseia na vagueza aplicada a definições formais e na evolução da denotação de construções. O mecanismo proposto permite ao projetista apresentar de maneira incremental a definição de uma linguagem de forma semelhante à presente em livros didáticos e manuais de linguagem.

Nesta técnica, pode ser apresentada ao leitor uma sequência de especificações que descrevem como programas completos podem ser escritos na linguagem. Cada especificação da sequência acrescenta novos recursos à especificação anterior e, se necessário, estabelece a forma como as denotações das construções já definidas devem evoluir para acomodar os novos recursos.

A evolução de denotações é definida por meio da utilização de transformadores de função, que é uma das contribuições deste trabalho. Como principal ponto positivo da técnica, pode-se resolver diversos problemas de entrelaçamento de definições de construções relacionadas, por meio da separação de preocupações, com impacto direto na escalabilidade de semântica denotacional. Com efeito, com a solução do entrelaçamento, remove-se um dos itens que levam à complexidade inerente do formalismo.

Outra importante contribuição do trabalho é a formalização da vagueza e a análise de sua relevância no contexto de métodos formais. Com efeito, a vagueza permite que, ao se ler uma definição, conceitos relacionados, mas não fundamentais à compreensão, possam ser omitidos, com impacto direto na facilidade de leitura. Este recurso é particularmente útil nas versões iniciais de uma definição formal.

Além disso, neste trabalho foi feita ainda a definição da linguagem Notus que permite a definição completa de linguagens de programação utilizando a metodologia proposta. A linguagem possui recursos para a definição da sintaxe e da semântica denotacional da linguagem e contempla recursos para modularidade, controle de visibilidade, definição de componentes, transformações de função e agrupamento de especificações.

A implementação de um compilador para a linguagem Notus é descrita por *Loredo et al.* (2008), *Soares* (2007). O compilador tem como entrada uma especificação de linguagem em Notus e gera, como resultado, um interpretador executável que realiza a leitura do programa fonte e dos dados de entrada, faz análises léxica e sintática e executa as funções semânticas para produção do resultado final.

Conforme mencionado anteriormente, é importante ressaltar que este trabalho complementa as contribuições de trabalhos anteriores, sendo possível, em algumas situações 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.

7.2 Trabalhos Futuros

Algumas restrições deste trabalho podem ser tratadas como possíveis trabalhos futuros.

Do ponto de vista da metodologia, um ponto interessante de pesquisa consiste em avaliar a possibilidade de, em vez de se definir uma sequência de especificações, estas pudessem ser organizadas na forma de um grafo acíclico dirigido representando suas relações de dependência. Neste trabalho, tal questão não foi tratada em virtude das dificuldades em se tratar a possibilidade de uma mesma função ser alvo de transformações distintas e independentes realizadas por transformadores distintos[1].

Outro ponto importante para investigação diz respeito à possibilidade de se incluírem novos tipos de transformadores, tais como transformadores de domínios semânticos. No

[1]Este problema é semelhante ao problema do losango em herança múltipla de classes na programação orientada por objetos.

entanto, a incorporação deste recurso leva a dificuldades em compatibilizar as funções já existentes, uma vez que possíveis operações de injeção e projeção em domínios podem levar a perda de informações anteriormente presentes. Uma possível solução consiste em limitar as operações dos domínios para extensão de tuplas, de maneira semelhante a extensão de classes na programação orientada por objetos. Novas transformações poderiam ainda ser aplicadas em outros elementos da definição, tais como a gramática concreta da linguagem.

Ainda do ponto de vista da teoria proposta, considera-se de grande importância verificar a existência de um conjunto de transformadores completo, com o qual toda transformação computável que preserve abstração do modelo denotacional pudesse ser especificada. É ainda desejável que esta base de transformadores seja ao mesmo tempo mínima e composta por transformações simples, que exijam menos esforço de aprendizado por parte do projetista e do leitor das definições.

Quanto à linguagem Notus, alguns elementos interessantes poderiam ser incorporados tais como o suporte a mônadas e a definição de uma biblioteca rica contendo domínios e funções específicos para a definição de linguagens de programação. Outro recurso que poderia ser incluído diz respeito a mecanismos mais poderosos para a transformação de gramática concreta em gramática abstrata. Os recursos presentes nesta versão não permitem, por exemplo, que a gramática abstrata seja unicamente definida como um núcleo simples cujas construções possam ser combinadas para definição de construções avançadas. Tais recursos seriam particularmente úteis para a definição de componentes linguísticos, que hoje deve ser feita por meio de funções de mapeamentos entre universos sintáticos.

Finalmente, do ponto de vista da implementação realizada, seria importante a incorporação dos novos recursos decorrentes destas novas pesquisas. Além disso, uma limitação à extensibilidade é devida à utilização de parser LALR(1); com efeito, a inclusão de novas regras de produção pode gerar conflitos com regras já existentes, levando à redefinição de módulos já escritos. Uma possível solução consiste na utilização de técnicas baseadas em Generalized LR (*Bravenboer and Visser* 2009) e a inclusão de construções de remoção de ambiguidades em gramáticas.

Apêndice A

The Syntax of Notus

This chapter presents the complete syntax of Notus, by means of its BNF Grammar.

A.1 Lexis Convention

This chapter presents basic elements of the language used throughout the text.

Names Notus defines names for:

- *packages*, *modules*, *transformers*, and *domains*, which start with a capital letter and ends with a letter; these names are generally called *capital names*; examples of those names are:

  `C  String  DomainA  Domain_a  Domain_A  A2b  Pir8_domain  Sk8r`

- *tokens*, *elements*, *variables*, *functions*, *identifier patterns*, *enumerands*, and *labels*, which start with a non-capital letter or the underline character and may be decorated with digits; these names are also generally called ordinary names; examples of those names are:

  `f  string  elem1  elem_a  elem_A1  pir8_domain  test  comm1`

In the text, these names are denoted by *package-name*, *module-name*, *domain-name*, *token-name*, *element-name*, *variable-name*, *function-name*, *id-pattern-name*, *enumerand-name*, *transformer-name*, and *label*.

Due to implementation issues, it is erroneous to define names with two or more adjacent underline symbols. A single underline symbol does not constitute an identifier, since it is reserved to represent the anonymous pattern (see Section 5.4.2).

Sequences of digits finishing ordinary names are considered decorations, so that `comm123` is considered to be name `comm` decorated with `123`. If the component identified by an ordinary name lacks a domain definition, the capital name of the latter may be implied by the compiler by: (i) eliminating its decoration; (ii) capitalizing its first letter. If there exists such domain name, the component is considered to be in such domain. For instance, `comm123` is considered to be in domain `Comm` if there exists such domain name. In the same way, identifiers `int1`, `int25`, `int2001` are considered to be in domain `Int`.

Capital names are defined by means of the regular expression $L((l|L|d|u)^*(l|L|u))^?$, where l represents any lowercase letter, L represents any uppercase letter, d represents any digit from `0` to `9`, and u represents the underline symbol. Ordinary names are defined by means of the regular expression $l(l|L|d|u)^*|u(l|L|d|u)^+$.

Literal Constants Notus defines the following Literal constants:

- `boolean-value` denotes the values **true** and **false**;
- `int-number` denotes 8-bits, 16-bits, or 32-bits integer numbers written in one of the following bases:
  - decimal: the number is a sequence of one or more digits from 0 to 9;
  - octal: the number is a sequence of one or more digits from 0 to 7, preceeded by the digit 0;
  - hexadecimal: the number is a sequence of one or more hexadecimal digits, i.e. digits from 0 to 9 and letters from `a` to `f`, preceeded by `0x`;

  This literal constant is defined by means of the regular expression: $\texttt{0}(\texttt{x}|\texttt{X})(d|l)^+|d^+$, where d represents any digit from `0` to `9`, and l represents any of the letters a, A, b, B, c, C, d, D, e, E, f, or F.
- `long-number` denotes 64-bits integer numbers written in one of the following bases:
  - decimal: the number is a sequence of one or more digits from 0 to 9, ended up with the letter `l` or `L`;

- octal: the number is a sequence of one or more digits from 0 to 7, preceeded by the digit 0, ended up with the letter `l` or `L`;
- hexadecimal: the number is a sequence of one or more hexadecimal digits, i.e. digits from 0 to 9 and letters from `a` to `f`, preceeded by `0x`, ended up with the letter `l` or `L`;

This literal constant is defined by means of the regular expression: $(\texttt{0}(\texttt{x}|\texttt{X})(d|l)^{+}|d^{+})(\texttt{l}|\texttt{L})$, where d represents any digit from `0` to `9`, and l represents any of the letters a, A, b, B, c, C, d, D, e, E, f, or F.

- `double-number` denotes IEEE 754 64-bit floating point numbers, and are defined by the regular expression $d^{+}(\texttt{.}d^{+})^{?}((\texttt{e}|\texttt{E})(\texttt{+}|\texttt{-})^{?}d^{+})^{?}$, where d represents any digit from `0` to `9`.

- `float-number` denotes IEEE 754 32-bit floating point numbers, and are defined by the regular expression $d^{+}(\texttt{.}d^{+})^{?}((\texttt{e}|\texttt{E})(\texttt{+}|\texttt{-})^{?}d^{+})^{?}(\texttt{f}|\texttt{F})$, where d represents any digit from `0` to `9`.

- `character` denotes single characters enclosed by single quotes, with the following escape sequences:

`\a`	alarm	`\t`	tabular
`\b`	backspace	`\\`	backslash
`\f`	form-feed	`\'`	single quote
`\n`	line-feed	`\"`	double quote

Characters are defined by means of the regular expression $'(c|e|o|h|u)'$, where:

- c represents any printable ASCII character;
- e represents any escape sequence listed above;
- o represents ASCII characters denoted by their octal representation and are defined by the regular expression $\backslash ddd$, where d represents a digit from `0` to `7`;
- h represents ASCII characters denoted by their hexadecimal representation and are defined by the regular expression $\backslash\texttt{x}(d|l)(d|l)$, where d represents a digit from `0` to `9`, and l represents one of the letters a, A, b, B, c, C, d, D, e, E, f, or F.

  - u represents Unicode characters denoted by their hexadecimal representation and are defined by the regular expression $\texttt{\textbackslash u}(d|l)(d|l)(d|l)(d|l)$, where d represents a digit from `0` to `9`, and l represents one of the letters a, A, b, B, c, C, d, D, e, E, f, or F.

- `quotation` denotes double quoted sequences of characters, with the same escape sequences as characters. Quotations are defined by the regular expression $\texttt{"}(c|e|o|h|u)^*\texttt{"}$, where c, e, o, h, and u are same as in the definition of characters.

Comments Comments in Notusare line comments and text comments. Line comments start with `"//"` and the compiler ignores everything from the `"//"` to the end of the line. Text comments start with `"/*"`, end with `"*/"`, and the compiler ignores everything between these symbols.

Keywords The keywords of Notus are:

by	**case**	**contains**	**default**	**element**	**else**
end	**evaluate**	**extend**	**extension**	**function**	**if**
ignore	**import**	**in**	**input**	**is**	**let**
module	**of**	**otherwise**	**output**	**preproc**	**private**
public	**redefine**	**replace**	**semantics**	**signature**	**sintactic**
syntax	**then**	**token**	**transformation**	**where**	**with**
within					

Layout Rules Inside a module the delimiters semicolon and braces may be omitted, so that the Notus pre-processor inserts them when their presences may be implied from indentation. The layout rules may be used for omitting the following delimiters:

- the semicolon ending each declaration;
- the braces in function body definitions;
- the braces in the last argument of a function application;
- the braces and semicolons in let, where, and case expressions.

A.2 Modular Organization

module	→	module-header import-section declaration∗ module-end
	|	extension-header contains-section
	|	transf-header import-section transf-section∗ module-end
module-header	→	**module** module-full-name
module-full-name	→	module-qualification *module-name*
module-qualification	→	module-qualification *package-name* "."
	|	λ
import-section	→	**import** module-full-name+−"," ";"
	|	λ
declaration	→	visibility exportable-declaration ";"
	|	non-exportable-declaration ";"
visibility	→	**public** | **private** | λ
module-end	→	**end** | λ
non-exportable-declaration	→	**preproc** expression
	|	**semantics** expression
extension-header	→	**extension** transformer-full-name "(" module-full-name ")"
contains-section	→	**contains** module-full-name+−"," ";"
	|	λ
transformation-header	→	**transformation** transformer-name
transformer-full-name	→	module-qualification *transformer-name*

A.3 Domains

exportable-declaration	→	syntactic-domain-declaration
	|	semantic-domain-declaration
syntactic-domain-declaration	→	**syntactic** *domain-name*
semantic-domain-declaration	→	*domain-name* "=" domain-exp
domain-exp	→	union-domain-exp
union-domain-exp	→	union-domain-exp "|" function-domain-exp
	|	function-domain-exp
function-domain-exp	→	list-domain-exp "->" function-domain-exp
	|	list-domain-exp
list-domain-exp	→	list-domain-exp "*"
	|	primary-domain-exp
primary-domain-exp	→	domain-full-name
	|	enum-domain-exp
	|	tuple-domain-exp
domain-full-name	→	module-full-name "." *domain-name*
	|	*domain-name*
tuple-domain-exp	→	visibility constructor "(" domain-exp*-"," ")"
constructor	→	*constructor-name*
	|	λ
enum-domain-exp	→	visibility "{" *enumerand-name*+-"," "}"

A.4 Lexis Definitions

The following grammar summarizes the syntax for lexical specifications:

exportable-declaration	→	token-declaration
	|	tokelem-declaration
non-exportable-declaration	→	ignore-definition
	|	token-extension
	|	tokenelem-extension
token-declaration	→	**token** *token-name* domain-definition
		"=" regular-expression lexeme-function
domain-definition	→	":" domain-full-name
	|	λ
lexeme-function	→	**is** expression
	|	λ
tokelem-declaration	→	**element** *element-name* "=" regular-expression
ignore-declaration	→	**ignore** regular-expression
token-extension	→	**extend token** qualified-id **with** regular-expression
tokenelem-extension	→	**extend element** qualified-id **with** regular-expression

The following grammar summarizes the syntax for regular expressions:

regular-expression	→	regular-expression "|" regexp-concatenation
	|	regexp-concatenation
regexp-concatenation	→	regexp-concatenation regexp-unary
	|	regexp-unary
regex-unary	|	regexp-primary "*"
	|	regexp-primary "+"
	|	regexp-primary "?"
	|	regexp-primary
regexp-primary	→	"(" regular-expression ")"
	|	`quotation`
	|	`charset`
	|	qualified-id

A.5 Syntax Definition

The following grammar summarizes the syntax of syntax definition.

exportable-declaration	→	variable-declaration
non-exportable-declaration	→	**startsymbol** qualified-id
	|	variable-extension
variable-declaration	→	`id` domain-definition rhs-definition
rhs-definition	→	"`::=`" rule-rhs+-"`|`"
	|	λ
rule-rhs	→	grammar-constituent abs-rhs-definition
abs-rhs-definition	→	abstract-rule-definition
	|	ignore-indirection-definition
	|	λ
abstract-rule-definition	→	"`:`" "`[`" abs-grammar-constituent "`]`"
ignore-indirection-definition	→	"`:`" qualified-id
variable-extension	→	**extend** qualified-id **with** rule-rhs+-"`|`"

The following grammar summarizes the syntax of grammar constituents:

grammar-constituent	→	grammar-contituent-list
	|	**empty**
	|	λ
grammar-constituent-list	→	grammar-contituent-list grammar-term
	|	grammar-term
grammar-term	→	grammar-unit "`*`"
	|	grammar-unit "`+`"
	|	grammar-unit "`*-`" `quotation`
	|	grammar-unit "`+-`" `quotation`
	|	grammar-unit
grammar-unit	→	qualified-id
	|	`quotation`

The following grammar summarizes the syntax of abstract grammar constituents:

abs-grammar-constituent	→	abs-grammar-contituent-list
	|	**empty**
abs-grammar-constituent-list	→	abs-grammar-contituent-list abs-grammar-term
	|	abs-grammar-term
abs-grammar-term	→	grammar-unit "*"
	|	grammar-unit "+"
	|	grammar-unit

A.6 Function Definitions

The syntax for function declaration and definition is given by the following grammar:

exportable-declaration	→	function-declaration
function-declaration	→	**function** `id` ":" domain-exp
non-exportable-declaration	→	function-definition
function-definition	→	qualified-id function-params "=" expression
function-params	→	pattern*

A.7 Patterns

The syntax of patterns is defined by means of the following grammar.

pattern	→	head-tail-pattern
	|	primary-pattern
head-tail-pattern	→	primary-pattern "`:`" head-tail-pattern
	|	primary-pattern "`:`" id-pattern
primary-pattern	→	literal-value
	|	signed-number-pattern
	|	id-pattern
	|	tuple-pattern
	|	list-pattern
	|	node-pattern
signed-number-pattern	→	"`-`" `int-number`
	|	"`-`" `long-number`
	|	"`-`" `double-number`
	|	"`-`" `float-number`
id-pattern	→	id-pattern "`*`"
	|	id-pattern "`+`"
	|	qualified-id
tuple-pattern	→	*constructor-name* "`(`" patterns-list "`)`"
list-pattern	→	"`(`" pattern`*-`"`,`" "`)`"
node-pattern	→	"`[`" node-pattern-element`+` "`]`"
node-pattern-element	→	id-pattern
	|	`quotation`
	|	**empty**

A.8 Expressions

Syntax of Expressions The syntax of expressions is given by the following grammar. All non-terminals are defined throughout this section.

expression	→	operation
compound-expression	→	if-expression
	|	case-expression
	|	let-expression
	|	where-expression
	|	lambda-abstraction
primary-expression	→	literal-value
	|	aggregate
	|	identifier
	|	"(" expression ")"
	|	evaluate-expression
	|	fun-update-expression

Literal Values The grammar for literal values is:

literal-value	→	`boolean-value`
	|	`int-number`
	|	`long-number`
	|	`double-number`
	|	`float-number`
	|	`character`
	|	`quotation`

Aggregates The grammar for aggregates is:

aggregate	→	tuple-aggregate
	|	list-aggregate
tuple-aggregate	→	*constructor-name* "(" expression∗"," ")"
list-aggregate	→	"(" expression∗"," ")"

Identifiers The syntax of identifiers is given by the following grammar.

identifier	→	list-decorated-id
list-decorated-id	→	list-decorated-id "+"
	|	list-decorated-id "*"
	|	qualified-id
qualified-id	→	id-qualification `id`
id-qualification	→	id-qualification `domain-id` "."
	|	λ

Operations The syntax for operations is given by the following grammar:

operation	→	pattern-matching-operation
pattern-matching	→	fun-composition-operation **is** pattern
fun-composition-operation	→	list-append-operation "." fun-composition-operation
	|	list-append-operation
list-append-operation	→	list-append-operation "++" list-cons-operation
	|	list-cons-operation
list-cons-operation	→	or-operation ":" list-cons-operation
	|	or-operation
or-operation	→	or-operation **or** and-operation
	|	and-operation
and-operation	→	and-operation **and** rel-operation
	|	rel-operation
rel-operation	→	additive-operation "=" additive-operation
	|	additive-operation "!=" additive-operation
	|	additive-operation "<" additive-operation
	|	additive-operation "<=" additive-operation
	|	additive-operation ">" additive-operation
	|	additive-operation ">=" additive-operation
	|	additive-operation
additive-operation	→	additive-operation "+" mult-operation
	|	additive-operation "-" mult-operation
	|	additive-operation "|" mult-operation
	|	additive-operation "^" mult-operation
	|	mult-operation
mult-operation	→	mult-operation "*" unary-operation

continues on next page...

...continued from previous page

	|	mult-operation "/" unary-operation
	|	mult-operation **mod** unary-operation
	|	mult-operation "&" unary-operation
	|	mult-operation ">>" unary-operation
	|	mult-operation ">>>" unary-operation
	|	mult-operation "<<" unary-operation
	|	unary-operation
unary-operation	→	"−" compound-expression
	|	"~" compound-expression
	|	**not** compound-expression
	|	compound-expression

Function Application The grammar for function applications and evaluate expressions is given by the following grammar:

function-application	→	function-application primary-expression
	|	primary-expression
evaluate-expression	→	**evaluate** "{" evaluate-part "}"
evaluate-part	→	evaluate-part ";" function-application
	|	function-application

Lambda Abstraction The syntax for lambda abstractions is given by the following grammar:

lambda-abstraction	→	"\" pattern+ "->" expression

Let and Where Expressions The syntax of let and where expressions is given by the following grammar:

let-expression	→	**let** "{" let-clauses "}" **in** expression
where-expression	→	expression **where** "{" let-clauses "}"
let-clauses	→	let-clause+−";"
let-clause	→	pattern "=" expression

Function Update The syntax for function update is given by the following grammar:

fun-update-expression	→	primary-expression update-clause
update-clause	→	"[" expression "←" expression "]"

Syntax of Conditional Expressions

if-expression	→	**if** expression **then** expression **else** expression
case-expression	→	**case** expression **of** "{" case-clauses "}"
case-clauses	→	case-clause+− ";"
case-clause	→	pattern "=>" expression

A.9 Transformation Clauses

The syntax for transformation clauses is given by the following grammar:

transf-section	→	**signature** signature-clause+− "," ";"
	|	**default** default-clause+− "," ";"
	|	**replace** replace-clause+− "," ";"
	|	**redefine** redefine-clause+− "," ";"
signature-clause	→	function-name ":" labelled-domain
labelled-domain	→	l-d-constituent+- "->"
l-d-constituent	→	domain
	|	label ":" domain
default-clause	→	label within-cons* "=" expression
replace-clause	→	function-application-pattern **by** expression
redefine-clause	→	function-header-pattern within-cons* **by** expression
function-application-pattern	→	function-name pattern*
function-header-pattern	→	function-name pattern*
within-cons	→	**within** function-header-pattern

Apêndice B

Examples of Language Definitions in Notus

B.1 Example of Syntax Definition: Language Nano

This section presents an example defining the syntax of *Nano*, a simple language containing expressions and commands. A program in *Nano* is a sequence of readings of values to variables, followed by the execution of a command, and finalized with the writing of the value of an expression. Commands in *Nano* are assignment, conditional, and repetition. Expressions in *Nano* are numbers, boolean literals *true* and *false*, arithmetic, relational, and logical operations.

B.1.1 Language Definition

Kernel Module *Kernel* in Listing B.1 defines white spaces and comments to be ignored by the lexical analyzer.

```
1 module Nano.Kernel
2     ignore layoutchar | comment ;
3     element layoutchar = " " | "\n" | "\t" | "\r" | "\f" ;
4     element comment = "//" .* ;
```

Listagem B.1: Module Kernel of Nano Definition

Expressions Module *Expressions* in Listing B.2 defines all issues concerning expressions, namely syntactic domains, tokens, variables, and concrete grammar rules. This module exports the domains *Exp*, *Id*, and *Value*; the tokens *id*, *num*, *bool*; and the variable *exp*.

Commands Module *Commands* in Listing B.3 defines all issues concerning commands, namely public syntactic domain *Comm*, public variable *comm*, and the concrete grammar rules for assignment, conditional, repetition, and block. This module imports public identifiers from *Expressions*.

Programs Module *Programs* in Listing B.4 defines public syntactic domain *Prog*, public variable *prog*, and the concrete grammar rule for programs. The *startsymbol* clause in its header defines that *prog* is the start symbol of the concrete grammar of *Nano*. Besides, modules *Expressions*, *Commands*, and *Kernel* must be imported.

Main Module The main module of this specification (in Listing B.5) just defines the start symbol of the grammar to be non-terminal *prog*.

B.1.2 Lexis of *Nano*

The lexis of *Nano* comprehends:

- token *id* in domain *Id*, whose lexeme is a string;
- token *num* in domain *Value*, whose lexeme is a integral number;
- token *bool* in domain *Value*, whose lexeme is a boolean value;
- keywords "`or`", "`and`", "`not`", "`if`", "`then`", "`else`", "`while`", "`do`", "`begin`", "`end`", "`read`", "`write`";
- operators "`+`", "`-`", "`*`", "`/`";
- delimiters: "`(`", "`)`", and "`;`".

Besides, white spaces and comments are defined and are ignored by the lexical analyzer.

```
1  module Nano.Expressions
2
3      public token id = $[a-zA-Z_]$[a-zA-Z0-9_]*;
4      token num : Value = [0-9]+ is asInteger;
5      token bool : Value = "true" | "false" is asBoolean;
6
7      public exp ::= exp "or" andexp
8                   | andexp : andexp;
9
10     andexp : Exp ::= andexp "and" addexp
11                    | addexp : addexp;
12
13     addexp : Exp ::= addexp "+" multexp
14                    | addexp "-" multexp
15                    | multexp : multexp;
16
17     multexp : Exp ::= multexp "*" unexp
18                     | multexp "/" unexp
19                     | unexp : unexp;
20
21     unexp : Exp ::= "not" primary
22                   | "-" primary;
23
24     primary : Exp ::= "(" exp ")" : exp
25                     | id
26                     | value;
27
28     value ::= bool
29             | num;
```

Listagem B.2: Module Expressions of Nano Definition

```
1 module Nano.Commands
2     import Expressions, Declarations;
3
4     public comm ::= id ":=" exp ";" : [id ":=" exp]
5                   | "if" exp "then" comm1 "else" comm2
6                         : ["ifthenelse" exp comm1 comm2]
7                   | "while" exp "do" comm : ["while" exp comm]
8                   | "{" decl* comm* "}" : [decl* ";" comm*] ;
```

Listagem B.3: Module Commands of Nano Definition

```
1 module Nano.Programs
2     import Commands, Expressions, Declarations;
3
4     prog ::= "read" id ";" comm ";" "write" exp : ["prog" id comm exp];
```

Listagem B.4: Module Programs of Nano Definition

```
1 module Main
2     import Nano.Programs;
3
4     syntax prog;
```

Listagem B.5: Main Module of Nano Definition

B.1.3 Syntax of *Nano*

The concrete grammar of *Nano* is $G_\eta^C = (V_\eta^C, T_\eta^C, R_\eta^C, prog)$, where:

- $V_\eta^C = \{prog, comm, exp, andexp, addexp, multexp, unexp, primary, value\}$;
- T_η^C is the set of tokens defined in Section B.1.2.
- R_η^C is a set containing rules:

```
prog     →  "read" id ";"comm ";""write" exp
comm     →  id ":=" exp
         |  "if" exp "then" comm1 "else" comm2
         |  "while" exp "do" comm
         |  "begin" comm+-";""end"
exp      →  exp "or" andexp
         |  andexp
andexp   →  andexp "and" addexp
         |  addexp
addexp   →  addexp "+"multexp
         |  addexp "-"multexp
         |  multexp
multexp  →  multexp "*"unexp
         |  multexp "/" unexp
         |  unexp
unexp    →  "not" primary
         |  "-"primary
primary  →  "("exp ")"
         |  id
         |  value
value    →  bool
         |  num
```

B.1.4 Abstract Grammar of *Nano*

The abstract grammar of *Nano* is $G_\eta^{A'} = (V_\eta^{A'}, T_\eta^{A'}, R_\eta^{A'}, Prog)$, where:

- $V_\eta^{A'} = \{Prog, Comm, Exp, Id, Value\}$, which contains all syntactic domains defined in the modules of *Nano* specification;

- $T_\eta^{A'}$ is the set containing:
  - tokens *id*, *num*, and *bool*;
  - keywords "`or`", "`and`", "`not`", "`if`", "`then`", "`else`", "`while`", "`do`", "`prog`";
  - operators "`+`", "`-`", "`*`", "`/`".
- $R_\eta^{A'}$ is a set containing rules:

```
Prog   →  "prog" Id Comm Exp
Comm   →  Id ":=" Exp
       |  "if" Exp "then" Comm "else" Comm
       |  "while" Exp "do" Comm
       |  Comm+
Exp    →  Exp "or" Exp
       |  Exp "and" Exp
       |  Exp "+" Exp
       |  Exp "-" Exp
       |  Exp "*" Exp
       |  Exp "/" Exp
       |  "not" Exp
       |  "-" Exp
       |  Exp
       |  Id
       |  bool
       |  num
Value  →  bool
       |  num
Id     →  id
```

Grammar $G_\eta^{A'}$ may be simplified by Algorithms I and II described in Section C.2, to produce $G_\eta^A = (V_\eta^A, T_\eta^A, R_\eta^A, Prog)$, where:

- $V_\eta^A = \{Prog, Comm, Exp, Id, \}$, which contains all syntactic domains defined in the modules of *Nano* specification;
- T_η^A is the set containing:
  - tokens *id*, *num*, and *bool*;
  - keywords "`or`", "`and`", "`not`", "`if`", "`then`", "`else`", "`while`", "`do`", "`prog`";

– operators "+", "-", "*", "/".

- R_η^A is a set containing rules:

```
Prog  →  "prog" Id Comm Exp
Comm  →  Id ":=" Exp
      |  "if" Exp "then" Comm "else" Comm
      |  "while" Exp "do" Comm
      |  Comm+
Exp   →  Exp "or" Exp
      |  Exp "and" Exp
      |  Exp "+" Exp
      |  Exp "-" Exp
      |  Exp "*" Exp
      |  Exp "/" Exp
      |  "not" Exp
      |  "-" Exp
      |  Id
      |  bool
      |  num
Id    →  id
```

B.1.5 Domains of *Nano*

The domains of *Nano* are *Id*, *Value*, *Exp*, *Comm*, and *Progs*, which are defined from the abstract grammar G_η^A and token definitions by:

- $Id = L\,(\texttt{[a-zA-Z\_][a-zA-Z0-9\_]*})$;
- $Value = Int + Bool$;
- $\begin{aligned} Exp = \;& Id + Value \\ & + (\{\text{"}\texttt{-}\text{"}\} \times Exp \times Exp) + (\{\text{"}\texttt{not}\text{"}\} \times Exp \times Exp) \\ & + (Exp \times \{\text{"}\texttt{+}\text{"}\} \times Exp) + (Exp \times \{\text{"}\texttt{-}\text{"}\} \times Exp) \\ & + (Exp \times \{\text{"}\texttt{*}\text{"}\} \times Exp) + (Exp \times \{\text{"}\texttt{/}\text{"}\} \times Exp); \end{aligned}$
- $\begin{aligned} Comm = \;& (Id \times \{\text{":="}\} \times Exp) + (\{\text{"}\texttt{while}\text{"}\} \times Exp \times \times Comm) \\ & + Comm^+ + (\{\text{"}\texttt{ifthenelse}\text{"}\} \times Exp \times Comm \times Comm); \end{aligned}$
- $Prog = \{\text{"}\texttt{prog}\text{"}\} \times Id \times Comm \times Exp$.

```
1 module Sek.Kernel
2     element layoutchar = " " | "\n" | "\t" | "\r" | "\f" ;
3     element comment = "//" .* ;
4     ignore layoutchar | comment
5
6     Value = Int | Int*
7     Ans = {stop,error} | Ans(Value,Ans)
```

Listagem B.6: Definition of the Kernel of Sek

B.2 Example of Semantics Definition: Language Sek

Sek is an imperative language for manipulating of sequences of integer numbers, with assignment and printing.

B.2.1 Kernel Language

In *Sek*, blank symbols and comments are ignored. A comment start with two slashes (`//`) and goes until the end of the line. Values in *Sek* are integer numbers and sequences of integers.

Listing B.6 is the definition of a core *Sek* language. This module defines that blank symbols and comments are ignored by means of a ignore clause, and defines the semantic domain `Value`, which represents the values of *Sek*.

B.2.2 Memory Abstraction

Memory in *Sek* is a mapping from locations to values, which can be integer numbers and sequences of locations. There is a special position in the memory, named *counter*, which represents the next memory cell that can be allocated. Expressible values are integer numbers, locations, sequences of integers, and sequences of locations. Functions *assign*, *deref*, *cont*, *ref*, and *new* are used for memory manipulation.

```
1  module Sek.Memory
2      import Kernel
3
4      Loc = private ({location},Int)
5      Sv = Int | Loc*
6      Ev = Int | Loc | Int* | Loc*
7
8      State = (Loc | {counter}) -> (Sv | {unused})
9      Cc = State -> Ans
10     Ec = Ev -> Cc
11
12     // assign: assigns a value to a location
13     function assign : Loc -> Cc -> Sv -> Cc
14     assign loc cc sv state = cc state[loc <- value]
15
16     // Dereference a value if it is a location or passes its
17     // value to the continuation otherwise
18     function deref : Ec -> Ec
19     deref ec loc = cont ec loc
20     deref ec ev = ec ev
21
22     // Gives the content of a memory cell
23     function cont : Ec -> Loc -> Cc
24     cont ec loc state = let sv = state loc in ec sv
25
26     // Creates a reference to a value
27     function ref : Ec -> Ec
28     ref ec ev = evaluate new; \loc -> assign loc (ec loc) ev
29
30     // Allocs a new location in memory
31     function new : Ec -> Cc
32     new ec state = let Loc(location,int) = state counter
33                    in ec loc state[counter <- Loc(location,int+1)]
```

Listagem B.7: Definition of Sek Memory

```
1 module Sek.Variables
2     import Kernel, Memory
3
4     token v = [A−Z] is asCharacter    // sequence variables
5     Var = Char
6     Env = Var −> (Loc | {unbound})
```

Listagem B.8: Definition of Variables in Sek

B.2.3 Variables

Variables in *Sek* are single uppercase characters. The environment maps variables into locations or the special value *unbound*, which denotes that a variable is not bound in the environment.

B.2.4 Declarations

Variable declarations are a sequence of uppercase characters preceeded by the keyword **var** and separated by commas. Each variable can be listed only once. Function *dd* is the denotation of declarations and produces an environment binding variables to memory locations.

B.2.5 Basic Expressions

The simplest form of expression is a number, which is just a sequence of digits interpreted as an integer. Function *de* is the denotation of expressions. The denotation of an integer number is the number itself. Function *dr* is the denotation of right-values and simply tests whether a value is a location, dereferencing the location if necessary.

B.2.6 Commands

The commands in *Sek* are assignment and printing. Function *dc* is the denotation of commands.

The left-hand side of assignments may be sequence names, which must be declared variables. The denotation of the assignment is the evaluation of its left-hand side, which must

```
1  module Sek.Declarations
2      import Memory, Variables
3
4      // Variable declarations are preceeded by the keyword "var"
5      // and separated by commas
6      vars : Var* ::= "var" v+"," : v+
7
8      Dc = Env -> Cc // Continuation of declarations
9
10     // Denotation of Declarations
11     function dd : Var* -> Env -> Dc -> Cc
12     dd env <> dc = dc env
13     dd env var:var* dc =
14         let cc1 = \state -> error
15         in case env var of
16                 unbound => evaluate new; \loc -> dc env[var <- loc]
17                 _ => cc1
```

Listagem B.9: Definition of Declarations in Sek

produce a location where the value of the right-hand side is stored. Function *dl* denotes left-values.

For printing commands the expression to be printed is evaluated as a right-value and its value is appended before the output.

B.2.7 Sequences

Sequences in *Sek* are lists of integers which are represented as sequences of locations in the memory, so that it is possible to update individual positions. Indexed variables are sequence variables followed by an integer number representing its index. Functions *asIndexed*, *seqelem*, *locseq*, *subseq*, *start_seq*, and *finish_seq* are defined in module *SeqAuxiliary* in Listing B.12.

A sequence expression may be an aggregate, which is a sequence of values separated by commas and enclosed in <>, or a subsequence of a given sequence limited by its lower and upper index in the original sequence. Indexed variables may be used as values and as assignments left-hand sides.

```
1  module Sek.Expressions
2      import Kernel, Variables, Memory
3
4      token n : E = [0−9]+ is asInteger        // numbers
5
6      e ::= val : val // Expressions are values
7      val: E ::= n // Values are numbers
8
9      // DENOTATION OF EXPRESSIONS
10     function de : E −> Env −> Ec −> Cc
11
12     // Numbers are just passed to the continuation
13     de int env ec = ec n
14
15     // DENOTATION OF RIGHT−VALUES
16     function dr : E −> Env −> Ec −> Cc
17
18     // if an expression in rhs is a location then it is dereferenced
19     dr e env ec = evaluate de e env; deref ec
```

Listagem B.10: Definition of Expressions in Sek

```
1  module Sek.Commands
2      import Variables, Expressions, Memory
3
4      // Commands are assignment and printing
5      c ::= lhs "=" e | "print" e
6
7      // Denotation of Commands
8      function dc : C -> Dc -> Cc
9
10     // Left-hand sides are sequence names
11     lhs: E ::= s
12
13     // Assignment: assigns the value of e2 to the location defined by e1
14     dc [e1 "=" e2] env cc = evaluate dl e1 env
15                                 \loc -> dr e2 env
16                                 assign loc cc
17
18     // Printing: preppends the value of e in the output
19     dc ["print" e] env dc = evaluate dr e env; print; dc env
20
21     // Sequence names must be bound to locations in the environment
22     de s env ec = let loc = env s in loc
23
24     // DENOTATION OF LEFT-VALUES
25     function dl : E -> Env -> Ec -> Cc
26
27     // expression in lhs must be a location and it is checked in
28     // the pattern matching in lambda abstraction
29     dl e env ec = evaluate de e env; \loc -> ec loc
30
31     // preppends a value in the output
32     function print : Ev -> Cc -> Cc
33     print int cc state = (int, cc state)
```

Listagem B.11: Definition of Commands in Sek

```
1  module Sek.SeqAuxiliary
2
3      Indexed = (Char,Int)
4      // asIndexed: separates identifier name and index from token i
5      function asIndexed: String -> Indexed
6      asIndexed c:s = Indexed(c,asInteger s)
7
8      function seqelem : Loc -> Int -> Ec -> Cc
9      seqelement loc:_ 0 ec = ec loc
10     seqelement _:loc* int = seqelem loc* (int - 1)
11
12     // Appends a location to a location list
13     function locseq : Ec -> Loc -> Loc* -> Cc
14     locseq ec loc loc* = ec (loc:loc*)
15
16     // Subsequence by index
17     function subseq : Loc* -> Int -> Ec -> Int -> Cc
18     subseq loc* int1 ec int2 =
19             if lt int1 0 then err
20             elseif lt int2 int1 then err
21             else start_seq loc* int1 ec int2
22
23     // Finds the start of the subsequence
24     private function start_seq : Loc* -> Int -> Ec -> Int -> Cc
25     start_seq <> 0 ec int2 = finish_seq <> ec int2
26     start_seq <> int1 ec int2 = err
27     start_seq loc* 0 ec int2 = finish_seq loc* ec int2
28     start_seq loc:loc* int1 ec int2 = start_seq loc* (int1 - 1) ec (int2 - 1)
29
30     // Finds the end of the subsequence
31     private function finish_seq : Loc* -> Ec -> Int -> Cc
32     finish_seq <> ec 0 = ec <>
33     finish_seq <> ec int = err
34     finish_seq loc:loc* ec int = finish_seq loc* ec1 (int - 1)
35                                 where ec1 = \loc* -> ec (loc:loc*)
```

Listagem B.12: Definition of Sequence Expressions in Sek

The denotation of a sequence aggregate is the evaluation of each element, and store each value in independent positions of the memory; the sequence is then a reference to the list of locations in the memory.

The denotation of a subsequence is the evaluation of its limiting indices and the extraction of a sequence of locations containing the elements of the original list. This sequence of location is stored in a new memory cell, but no new cell is allocated for each elements in the subsequence, and thus they are alias to the elements in the original list.

The denotation of a sequence element is a value stored in the sequence in the given position. Positions in sequences start in zero.

B.2.8 Programs

A program in *Sek* is a sequence of variable declarations followed by a sequence of commands. The denotation of a program is the sequential execution of each command in the environment produced by the evaluation of the variable declarations. After the execution of all commands, the execution stops.

B.2.9 Main Module

The main module defines the start symbol of the language's grammar, the semantic function for programs, and input and output streams.

```
1  module Sek.Sequences
2      import Memory, Expressions, Variables, Kernel, SeqAuxiliary
3
4      token i = [A−Z][0−9]+ is asIndexed // indexed sequence elements
5
6      // Expressions may be sequences
7      extend e with seq : seq
8
9      // Sequences are lists of values in <> or
10     // subsequences defined by sequence and boundaries
11     seq: E ::= "<" val*−"," ">" : ["<" val* ">"]
12              | v "[" val ".." val "]"
13
14     // Values may be sequence elements
15     extend val with i
16     // Left−hand sides may be indexed elements
17     extend lhs with i
18
19     // Sequences of values
20     de ["<" e* ">"] env ec = dseq e* env; ref ec
21
22     // Subsequences
23     de [s "[" e1 ".." e2 "]"] env ec =
24             evaluate dr s env; \loc* −> dr e1 env;
25                     \int1 −> dr e2 env;
26                     subseq loc* int1; ref ec
27
28     // Sequence elements are checked if the sequence exists in the environment, and
29     // then the corresponding location in the sequence is passed to the continuation
30     de Indexed(char,int) env ec = let loc = env char
31                                     in seqelement loc int ec
32
33     // DENOTATION OF SEQUENCES
34     function dseq : E* −> Env −> Ec −> Cc
35     dseq <> env ec = ec <>
36     dseq e:e* env ec = evaluate dr e env; ref;
37                                 \loc −> dseq e* env;
38                                 locseq ec loc
```

Listagem B.13: Definition of Sequence Expressions in Sek

```
1  module Sek.Programs
2      import Declarations, Commands
3
4      // a program is a declaration of variables
5      // followed by a sequence of commands
6      public p ::= vars c*
7
8      // Denotation of Programs
9      public function dp : P -> Ans
10     dp [var* c*] =
11         let env0 = \char -> unbound
12             state0 = \loc -> unused
13             cc0 = \state -> stop
14             dc0 = \env -> dcs c* env cc0
15         in dd var* env0 dc0 state0
16
17     // Denotation of Command Sequences
18     function dcs : C* -> Dc -> Cc
19     dcs <> _ cc _ = stop
20     dcs c:c* env = evaluate dc c env; dcs c* env cc
```

Listagem B.14: Definition of Programs in Sek

```
1  module Main
2      import Sek.Programs
3
4      syntax p // p is the start symbol of the grammar of Sek
5
6      semantics dp // dp is the evaluation function of Sek
7
8      input stdin
9
10     output stdout
```

Listagem B.15: Main Module of Sek Definition

B.3 Example of Incremental Definition: Language Tiny

B.3.1 Introduction

Consider Tiny is a programming language containing expressions, commands, variable declarations, and sequencers *break* and *continue*. This section shows the incremental definition of Tiny using Notus. This language is defined in two specification, so that the first specification (*Tiny0*) defines all constructs but the sequencers, which are defined in the second specification (*Tiny1*).

B.3.2 Initial Specification

The initial specification defines all constructs but the sequencers *break* and *continue*, which will be defined in the second specification.

B.3.2.1 Kernel

The first module in the description of Tiny is *Kernel* (Listing B.16, page 187), which defines that comments and white spaces must be ignored. It also defines identifiers, semantic domains, and auxiliary functions to manipulate memory.

B.3.2.2 Expressions

The simplest constructs of Tiny consists of expressions involving integral numbers and boolean values. Numbers consist of any non-empty sequence of digits from `0` to `9`, and boolean values are `true` and `false`.

Operators are divided into three categories: (i) arithmetic: `+`, unary and binary `-`, `*`, `div`, and `mod`; (ii) comparative: `=`, `!=`, `<`, `>`, `<=`, and `>=`; (iii) logical: `not`, `and`, and `or`. Precedence and associativity are defined in Table B.1, and parenthesis can be used to cast precedence. Values and operators have the usual meaning.

Modules *Expressions*, *Arithmetics*, *Relationals*, and *Booleans* define lexis, syntax, and semantics of these features.

```
1  module Basics.Kernel
2
3      ignore layoutchar | comment ;
4
5      element layoutchar = " " | "\n" | "\t" | "\r" | "\f" ;
6      element comment = "//" .* ;
7
8      public syntactic Id ;
9      public token id : Id = [A-Za-z]([A-Za-z]|[0-9])* ;
10
11     public Loc = ({location},Int);
12     public Val = Int | Bool | Loc ;
13
14     public Store = Loc -> (Val | {unused}) ;
15     public Env = Id -> (Loc | {unbound}) ;
16
17     public Ans = String ;
18
19     public Cc = State -> Ans ;
20     public Ec = Val -> Cc ;
21     public Dc = Env -> Cc ;
22
23     // assigns a value to a location
24     function assign : Loc -> Cc -> Val -> Cc ;
25     // dereference a value if it is a location or passes its
26     // value to the continuation otherwise
27     function deref : Ec -> Ec ;
28     // gives the content of a memory cell
29     function cont : Ec -> Loc -> Cc ;
30     // creates a reference to a value
31     function ref : Ec -> Ec ;
32     // allocs a new location in memory
33     function new : Ec -> Cc ;
34     // issues an error message
35     function err : Cc ;
```

Listagem B.16: Module Kernel in the Definition of Tiny

Operator	Precedence	Associativity
`not`, unary `-`	*highest precedence*	*right*
`*`, `div`, `mod`		*left*
`+`, binary `-`		*left*
`<`, `>`, `<=`, `>=`		*none*
`=`, `!=`		*none*
`and`		*left*
`or`	*lowest precedence*	*left*

Tabela B.1: Precedence and Associativity in Tiny

```
1  module Basics.Expressions
2
3      import Kernel ;
4
5      // Syntactic domain of expressions
6      public syntactic Exp ;
7
8      // Denotation of expressions
9      public de : Exp -> Env -> Ec -> Cc ;
```

Listagem B.17: Module Expressions in Tiny Definition

B.3.2.3 Module Expressions

Module *Expressions* (Listing B.17, page 188) defines: (i) the initially empty syntactic domain *Exp*, which represents syntax of expressions; (ii) the semantic function for expressions, *de*. These elements are initially set to empty in order to represent only the interface necessary to define new expression constructs.

B.3.2.4 Module Arithmetics

Module *Arithmetics* (Listing B.18, page 190) defines syntax and semantics of expressions involving numbers. In the lexical part, token *num* is defined as a sequence of digits, and since it may be necessary in other modules it is declared public. Since *num* is a token it is automatically included in the lexis definition. In addition, variable *digit* is defined to be private to the module because it is auxiliary in the definition of token *num*.

New forms of expressions are defined in the language by adding new syntactic clauses having syntactic domain *Exp* as left-hand side. Furthermore, each concrete grammar production is associated with some abstract grammar rule.

Besides, using symbols `"+"`, `"-"`, `"*"`, `"div"`, `"mod"`, `"("`, and `")"` in the syntactic clauses indicates they are tokens of the language and therefore must be automatically included in the lexis of the language.

Furthermore, the semantics of each new construct is defined as new definition of function *de*.

B.3.2.5 Module Relationals

Module *Relationals* (Listing B.19, page 191) initially extends the concrete grammar of Tiny to contain new production rules corresponding to relational operations. These inclusions cause the extension of domain *Exp* and the inclusion of tokens `"="`, `"!="`, `"<"`, `">"`, `"<="`, and `">="` in the lexis of the language.

B.3.2.6 Module Booleans

Module *Booleans* (Listing B.20, page 192) initially defines token *bool* to represent boolean constants, and new syntactic clauses to boolean operations, by defining rules having syntactic domain *Exp* as left-hand side, and their denotations by extending function *de*.

B.3.2.7 Commands

Module *Commands* (Listing B.21, page 193) defines syntactic domain *Comm* to represent commands and semantic function *dc* to compute their denotations.

B.3.2.8 Declarations

Module *Declarations* (Listing B.22, page 193) defines syntactic domain *Decl*, to represent declarations, grammar rules for variable declaration and declaration sequences, and semantic function *dd* for declarations.

```
1  module Basics.Arithmetics
2
3      import Expressions, Kernel;
4
5      public token num : Num ::= digit+ is asInteger ; // defines numbers
6      digit  ::= [0−9] ;
7
8      addexp : Exp ::=
9          addexp "+" multexp | addexp "−" multexp
10         | multexp;
11
12     multexp : Exp ::=
13         multexp "∗" unary | multexp "div" unary | multexp "mod" unary
14         | unary;
15
16     unary : Exp ::=
17         "−" primary | primary;
18
19     primary ::=
20         "(" exp ")" : exp | num ;
21
22     de [num] env ec = ec num;
23
24     de ["−" exp] env ec = de exp env (\int −> ec (−int))
25
26     de [exp1 "+" exp2] env ec =
27         de exp1 env (\int1 −> de exp2 env (\int2 −> ec (int1 + int2))) ;
28
29     de [exp1 "−" exp2] env ec =
30         de exp1 env (\int1 −> de exp2 env (\int2 −> ec (int1 − int2))) ;
31
32     de [exp1 "∗" exp2] env ec =
33         de exp1 env (\int1 −> de exp2 env (\int2 −> ec (int1 ∗ int2))) ;
34
35     de [exp1 "div" exp2] env ec =
36         de exp1 env (\int1 −> de exp2 env (
37             \int2 −> if int2 == 0 then err else ec (int1 / int2)))  ;
38
39     de [exp1 "mod" exp2] env ec =
40         de exp1 env (\int1 −> de exp2 env (
41             \int2 −> if int2 == 0 then err else ec (int1 mod int2))) ;
```

Listagem B.18: Module Arithmetics in Tiny Definition

```
1  module Basics.Relationals
2
3      import Kernel, Expressions, Arithmetics;
4
5      eqexp ::=
6          ineqexp "==" ineqexp
7          | ineqexp "'=" ineqexp
8          | ineqexp ;
9
10     ineqexp ::=
11         addexp "<" addexp
12         | addexp ">" addexp
13         | addexp "<=" addexp
14         | addexp ">=" addexp
15         | addexp ;
16
17     de [exp1 "==" exp2] env ec =
18         de exp1 env (\int1 -> de exp2 env (\int2 -> ec (int1 == int2))) ;
19
20     de [exp1 "'=" exp2] env ec =
21         de exp1 env (\int1 -> de exp2 env (\int2 -> ec (int1 '= int2))) ;
22
23     de [exp1 "<" exp2] env ec =
24         de exp1 env (\int1 -> de exp2 env (\int2 -> ec (int1 < int2))) ;
25
26     de [exp1 ">" exp2] env ec =
27         de exp1 env (\int1 -> de exp2 env (\int2 -> ec (int1 > int2))) ;
28
29     de [exp1 "<=" exp2] env ec =
30         de exp1 env (\int1 -> de exp2 env (\int2 -> ec (int1 <= int2))) ;
31
32     de [exp1 ">=" exp2] env ec =
33         de exp1 env (\int1 -> de exp2 env (\int2 -> ec (int1 >= int2))) ;
```

Listagem B.19: Module Relationals in Tiny Definition

```
1  module Basics.Booleans
2
3      import Kernel, Expressions, Arithmetics, Relationals;
4
5      token bool : Bool ::= "true" | "false" is asBoolean ; // defines numbers
6
7      extend primary with bool ;
8
9      exp : Exp ::=
10         exp "or" andexp
11         | andexp ;
12
13     andexp : Exp ::=
14         andexp "and" eqexp
15         | eqexp ;
16
17     extend unary with "not" exp ;
18
19     de [bool] env ec = ec bool;
20
21     de ["not" exp] env ec = de exp env (\bool -> ec (~ bool))
22
23     de [exp1 "and" exp2] env ec =
24         de exp1 env (
25             \bool1 -> case bool1 of
26                           true  => de exp2 env ec
27                           false => ec false
28         ) ;
29
30     de [exp1 "or" exp2] env ec =
31         de exp1 env (
32             \bool1 -> case bool1 of
33                           false => de exp2 env ec
34                           true  => ec true
35         ) ;
```

Listagem B.20: Module Booleans in Tiny Definition

```
1  module Basics.Commands
2
3      import Kernel, Expressions ;
4
5      public syntactic Comm ;
6
7      public function dc : Comm -> Env -> Cc -> Cc ;
```

Listagem B.21: Module Commands in Tiny Specification

```
1  module Basics.Declarations
2
3      import Kernel, Expressions ;
4
5      public syntactic Decl ;
6
7      public decl : Decl ::=
8          "var" id ";" : ["var" id]
9          | decl* ;
10
11     public function dd : Decl -> Env -> Dc -> Cc ;
```

Listagem B.22: Module Declarations in Tiny Definition

```
1  module Basics.InputOutput
2
3      import Kernel, Commands;
4
5      extend Loc with {input,output} ;
6      extend Val with String ;
7
8      function read :: Ec -> Cc ;
9      function write :: Cc -> Val -> Cc ;
10
11     extend comm with "write" exp;
12     extend exp with "read" ;
13
14     de ["read"] env ec = read ec ;
15
16     dc ["write" exp] env cc = de exp env (write cc) ;
```

Listagem B.23: Module InputOutput in Tiny Definition

B.3.2.9 Input and Output

Module *InputOutput* (Listing B.23, page 194) initially extends domain *Loc* to allow input and output sequences be stored in the memory; also, domain *Val* is extended to contain strings, which represent input and output streams. Auxiliary functions to read and write values are defined. The syntactic domain of commands is extended to include writing, and the syntactic domain of expressions is extended to include reading. The semantics of both constructs are given by defining new clauses of functions *dc* and *de*.

B.3.2.10 Variables and Assignment

Module *Variables* (Listing B.24, page 195) defines identifiers as expression by extending grammar variable *primary*. Then, the rule for commands is extended to include assignment statement. The semantics for declarations, identifier expression, and assignment are given by defining new clauses in functions *dd*, *de*, and *dc*.

```
1  module Basics.Variables
2
3      import Declarations, Expressions, Commands, Kernel ;
4
5      extend primary with id ;
6
7      extend comm with id ":=" exp ";" : [id ":=" exp] ;
8
9      dd ["var" id] env dc =
10         new (\loc -> dc (env[id <- loc])) ;
11
12     dd [decl*] env dc =
13         case decl* of
14             ()                  => dc ;
15             decl' :: decl*' => dd decl' env (\env' -> dd decl*' env' dc)
16
17     de [id] env ec =
18         case env id of
19             unbound => err
20             loc     => deref ec loc
21
22     dc [id ":=" exp] env cc =
23         case env id of
24             unbound => err
25             loc     => de exp env (assign loc cc)
```

Listagem B.24: Module Variables in Tiny Definition

B.3.2.11 Control Flow

Module *ControlFlow* (Listing B.25, page 197) defines syntax and semantics for conditional, repetition, and blocks. Auxiliary function *dcs* is used to iterate over sequences of commands.

B.3.2.12 Programs

Module *Programs* (Listing B.26, page 198) defines syntax and semantics for programs, which consist of commands. Semantic function *dp* executes the command corresponding to a program in an empty environment, an store which only maps *input* to the input stream and *output* to an empty sequence, and in a continuation, for which, given a final store, returns the sequence associated with the output location.

B.3.2.13 Main Module

Module *Main* (Listing B.27, page 198) defines the start symbol of the grammar to be *prog*, the semantic function to be *dp*, and uses standard input and output.

B.3.3 Specification Tiny1

B.3.3.1 Main Module

Module *Main* of specification *Tiny1* (Listing B.28, page 198) is an extension module, which applies transformation *IncludeSequencers* to specification *Tiny0*.

B.3.3.2 Transformer Module

Transformer *IncludeSequencer* (Listing B.29, page 199) decorates the semantics of commands inside semantic equation for *while*, by indicating in the environment the continuations for *break* and *continue*.

```
1  module Basics.ControlFlow
2
3      import Kernel, Expressions, Declarations, Commands ;
4
5      public comm : Comm ::=
6          "if" exp "then" comm1 "else" comm2
7          | "while" exp "do" comm
8          | "begin" decl comm* "end" ;
9
10     dc ["if" exp "then" comm1 "else" comm2] env cc =
11         de exp env (\bool ->
12             case bool of true  => dc comm1 env cc
13                          false => dc comm2 env cc
14         ) ;
15
16     dc ["while" exp "do" comm] env =
17         fix (\f cc -> de exp env (\bool ->
18             case bool of true  => dc comm env (f cc)
19                          false => cc
20         )) ;
21
22     dc ["begin" decl comm* "end"] env cc =
23         dd decl env (\env' -> dcs comm* env' cc) ;
24
25     function dcs : Comm* -> Env -> Cc -> Cc ;
26     dcs () env cc = cc
27     dcs comm:comm* env cc =
28         dc comm env (dcs comm* env cc) ;
```

Listagem B.25: Module ControlFlow in Tiny Definition

```
1  module Programs
2
3      import Kernel, Commands;
4
5      public syntactic Prog;
6
7      public prog : Prog ::= comm ;
8
9      public dp : Prog -> String -> String ;
10
11     dp [comm] string =
12         string' where store   = dc comm env0 cc0 store0
13                       string' = store output
14                       env0    = \id -> unbound
15                       cc0     = \store -> store output
16                       store0  = (\loc -> unused)[input <- string, output <- ""];
```

Listagem B.26: Module Programs in Tiny Definition

```
1  module Main
2
3      import Kernel, AuxiliaryImplementations, Programs;
4
5      syntax prog;
6      semantics dp;
7      input stdin;
8      output stdout;
```

Listagem B.27: Main Module of Specification *Tiny0*

```
1  extension IncludeSequencer(Tiny0)
2      contains Sequencers
```

Listagem B.28: Main Module of Specification *Tiny1*

```
1  transformation IncludeSequencer
2      import Basics.Kernel, Basic.Commands, Basic.ControlFlow,
3             Basic.Expressions, Sequencers.Sequencers ;
4
5      replace dc [com] env cc within dc ["while" exp "do" comm] env' cc'
6                  by dc [com] (env[cc'/break, cc/continue])
```

Listagem B.29: Transformer *IncludeSequencer*

```
1  module Sequencers.Sequencers
2
3      import Commands, Kernel;
4
5      extend comm with "break" | "continue" ;
6
7      extend Id with public {break,continue}
8
9      dc ["break"] env cc = env break;
10
11     dc ["continue"] env cc = env continue;
```

Listagem B.30: Module *Sequencers*

B.3.3.3 Sequencers

Module *Sequencers* (Listing B.30, page 199) includes new grammar rules for sequencers *break* and *continue*, defines special positions for both sequencer as identifiers, and defines the their semantics.

Apêndice C

Auxiliary Algorithms

C.1 Transforming Notus Grammars into BNF

Function ζ takes a grammar G defined in Notus and produces a new grammar G' in BNF.

The auxiliary function $\zeta' : \mathcal{C} \rightarrow (\mathcal{C} \times 2^{\mathcal{R}} \times 2^{\mathcal{V}})$, where $\mathcal{C}$ is the set of all rule constituents, $\mathcal{V}$ is the set of all grammar variables, and $\mathcal{R}$ is the set of all production rules[1]. Function ζ' applied to a rule constituent α is a triple $(\beta, R_\alpha, V_\alpha)$, where β is a rule constituent in conformity with usual parser generators syntax, R_α is a (possibly empty) set of grammar rules which must be included in R, and V_α is a (possibly empty) set of grammar variables which must be included in V.

Function ζ' is recursively defined as follows:

- $\zeta'(\lambda) = (\lambda, \emptyset, \emptyset)$;
- $\zeta'(\textbf{empty}) = (\lambda, \emptyset, \emptyset)$;
- $\zeta'(v) = (v, \emptyset, \emptyset)$, if $v \in V$;
- $\zeta'(t) = (t, \emptyset, \emptyset)$, if $t \in T$;
- $\zeta'(\alpha_1\alpha_2 \cdots \alpha_m) = (\beta_1\beta_2 \cdots \beta_m, R_\alpha, V_\alpha)$, where:
  - $(\beta_i, R_{\alpha_i}, V_{\alpha_i}) = \zeta'(\alpha_i)$, for $1 \leq i \leq m$;

[1]The notation 2^A represents the power-set of A.

  - $R_\alpha = \bigcup_{i=1}^{m} R_{\alpha_i}$;
  - $V_\alpha = \bigcup_{i=1}^{m} V_{\alpha_i}$;

- $\zeta'(\alpha*) = (v, \{v \to \beta\, v, v \to \lambda\} \cup R_\alpha, \{v\} \cup V_\alpha)$, where:
  - $(\beta, R_\alpha, V_\alpha) = \zeta'(\alpha)$;
  - v is a new variable;
- $\zeta'(\alpha+) = (v, \{v \to \beta\, v, v \to \beta\} \cup R_\alpha, \{v\} \cup V_\alpha)$, where:
  - $(\beta, R_\alpha, V_\alpha) = \zeta'(\alpha)$;
  - v is a new variable;
- $\zeta'(\alpha *\text{-} t) = (v, R' \cup R_\alpha, \{v, v_1\} \cup V_\alpha)$, where:
  - $R' = \{v \to v_1, v \to \lambda, v_1 \to \beta\, t\, v_1, v_1 \to \beta\}$;
  - $(\beta, R_\alpha, V_\alpha) = \zeta'(\alpha)$;
  - v, v_1 are new variables;
- $\zeta'(\alpha +\text{-} t) = (v, R' \cup R_\alpha, \{v\} \cup V_\alpha)$, where:
  - $R' = \{v \to \beta\, t\, v_1, v \to \beta\}$;
  - $(\beta, R_\alpha, V_\alpha) = \zeta'(\alpha)$;
  - v is a new variable;

If $G = (V, T, R, S)$ is a grammar, then $G' = \zeta(G) = (V \cup V', T, R', S)$, where:

- $V' = \{v \in V_\alpha \mid A \to \alpha \in R \wedge (\beta, R_\alpha, V_\alpha) = \zeta'(\alpha)\}$;
- $R' = R_1 \cup R_2$, where:
  - $R_1 = \{A \to \beta \mid A \to \alpha \in R \wedge (\beta, R_\alpha, V_\alpha) = \zeta'(\alpha)\}$;
  - $R_2 = \{p \in R_\alpha \mid A \to \alpha \in R \wedge (\beta, R_\alpha, V_\alpha) = \zeta'(\alpha)\}$.

For instance, consider grammar $G = (\{E\}, \{x, f, \texttt{","}, \texttt{"("}, \texttt{")"}\}, R, E)$, where

$$R: \; E \;\rightarrow\; x \;\mid\; f \;\texttt{"("}\;\; E\texttt{*-","}\;\; \texttt{")"}$$

The conversion of this grammar via function ζ produces grammar

$$G' = (\{E, v, v_1\}, \{x, f, \texttt{","}\;\;, \texttt{"("}\;\;, \texttt{")"}\;\;\}, R', E),$$

where

$$\begin{array}{rcl} R': \; E & \rightarrow & x \;\mid\; f \;\texttt{"("}\;\; v \;\texttt{")"} \\ v & \rightarrow & v_1 \;\mid\; \lambda \\ v_1 & \rightarrow & v_1 \;\texttt{","}\;\; E \;\mid\; E \end{array}$$

C.2 Simplification of Abstract Grammars

Given an abstract grammar $G = (V, T, R, S)$, it is possible to obtain a simplified, but equivalent, version of G, using algorithms for elimination of unused symbols. These algorithms are adapted from *Sipser* (1996).

Algorithm I: Elimination of Variables Which Do Not Generate Strings. The first algorithm consists of eliminating variables from which no string in T^* is generated. Given a grammar $G = (V, T, R, S)$, this algorithm initially creates a mapping $m_0 : V \rightarrow 2^{2^V}$ from variables to sets of sets of variables. For each rule $X \rightarrow \alpha \in R$, $\{X_1, \cdots, X_k\} \in m_0\; X$, if each X_i appears in α.

Then function $\varphi : (V \rightarrow 2^{2^V}) \rightarrow (V \rightarrow 2^{2^V})$ is defined as:

$$\varphi\; m\; v = \{A - Z_m \mid A \in m_0\; v\},$$

where $Z_m = \{w \in V \mid \emptyset \in m\; w\}$. From function φ, define a sequence of mappings $M = m_0, m_1, m_2, \cdots$, such that $m_{i+1} = \varphi\; m_i$, for $i \geq 0$. Then a new grammar equivalent to G is given by $G' = (Z_{\overline{m}}, T, R', S)$, where:

- $\overline{m} = \textbf{lub}\; M$, i.e the least upper bound[2] of M, and therefore a fix-point of φ.
- $R' = \{v \rightarrow \alpha \in R \mid v \in Z_{\overline{m}} \wedge \alpha \in (T \cup Z_{\overline{m}})^*\}$;

[2]The existence of $\overline{m}$ is a direct consequence from the fact that the sets are finite, and all elements in $Z_{m_i} \subset Z_{m_{i+1}}$, for all $i \geq 0$.

For instance, consider grammar $G = (V, T, R, S)$, where $V = \{A, B, C, D, E, F, S\}$, $T = \{a, b, c, d, e, f, g\}$, and R given by:

$$\begin{array}{rlcl} R: & S & \rightarrow & AbB \mid Cd \mid Df \\ & A & \rightarrow & Ac \mid d \\ & B & \rightarrow & Cd \mid Bc \mid A \\ & C & \rightarrow & De \mid Cf \\ & D & \rightarrow & Dg \mid Ee \\ & E & \rightarrow & Ce \mid Db \mid Ed \\ & F & \rightarrow & Cb \mid Fab \mid c \end{array}$$

Mapping m_0 created from G, mappings m_1 and m_2 defined using function φ are given by:

m_0	m_1	m_2
$S \mapsto \{\{A, B\}, \{C\}, \{D\}\}$	$S \mapsto \{\{B\}, \{C\}, \{D\}\}$	$S \mapsto \{\emptyset, \{C\}, \{D\}\}$
$A \mapsto \{\{A\}, \emptyset\}$	$A \mapsto \{\emptyset\}$	$A \mapsto \{\emptyset\}$
$B \mapsto \{\{C\}, \{B\}, \{A\}\}$	$B \mapsto \{\{C\}, \{B\}, \emptyset\}$	$B \mapsto \{\{C\}, \emptyset\}$
$C \mapsto \{\{D\}, \{C\}\}$	$C \mapsto \{\{D\}, \{C\}\}$	$C \mapsto \{\{D\}, \{C\}\}$
$D \mapsto \{\{D\}, \{E\}\}$	$D \mapsto \{\{D\}, \{E\}\}$	$D \mapsto \{\{D\}, \{E\}\}$
$E \mapsto \{\{C\}, \{D\}, \{E\}\}$	$E \mapsto \{\{C\}, \{D\}, \{E\}\}$	$E \mapsto \{\{C\}, \{D\}, \{E\}\}$
$F \mapsto \{\{C\}, \{F\}, \emptyset\}$	$F \mapsto \{\{C\}, \emptyset\}$	$F \mapsto \{\{C\}, \emptyset\}$
$Z_{m_0} = \{A, F\}$	$Z_{m_1} = \{A, B, F\}$	$Z_{m_2} = \{A, B, F, S\}$

Since $\varphi\ m_2 = m_2$, $\overline{m} = m_2$, and from this mapping, the only variables which generate strings in T^* are A, B, F, and S. Then grammar $G' = (\{A, B, F, S\}, T, R', S)$, where:

$$\begin{array}{rlcl} R': & S & \rightarrow & AbB \\ & A & \rightarrow & Ac \mid d \\ & B & \rightarrow & Bc \mid A \\ & F & \rightarrow & Fab \mid c \end{array}$$

Note: terminals e, f, and g are not used in this grammar; however, it is not necessary to eliminate them in this algorithm, since they are unreachable symbols and, for this reason, will be eliminated in Algorithm II.

Algorithm II: Elimination of Unreachable Symbols. The second algorithm consists of eliminating terminals and variables unreachable from the starting symbol. This problem is solved by modelling the resulting grammar of the first algorithm, $G' = (V', T, R', S)$, as

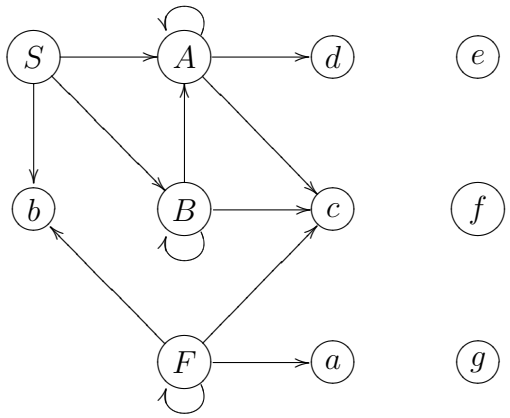

Figura C.1: Example of Graph Generated by Algorithm II for Grammar G'.

a directed graph $H = (N, E)$, where $N = V' \cup T$, and for each $n_1, n_2 \in N$, $(n_1, n_2) \in E$ if $n_1 \in V'$ and there exists at least one rule $n_1 \rightarrow \alpha n_2 \beta \in R'$, for some $\alpha, \beta \in (V' \cup T)^*$. From this graph, it is possible to create a new grammar $G'' = (V'', T'', R'', S)$, where:

- $V'' = V' \cap \hat{\Gamma}^+(S)$, where $\hat{\Gamma}^+(S)$ is the transitive closure of vertex S in graph H; V'' represents all variables which can be present in some sentential form generated by S;
- $T'' = T \cap \hat{\Gamma}^+(S)$; this set represents all terminals which can be present in some sentential form generated by S;
- $R'' = \{A \rightarrow \alpha \in R' \mid A \in V'' \wedge \alpha \in (V'' \cup T'')^* \wedge \alpha \neq A\}$; this set represents all useful rules of G.

For instance, consider grammar G' produced in the example for Algorithm I. Algorithm II creates the graph shown in Figure C.1, and in this graph, $\hat{\Gamma}^+(S) = \{A, B, S, b, c, d\}$. Then, this algorithm produces grammar $G'' = (\{A, B, S\}, \{b, c, d\}, R'', S)$, where:

$$\begin{array}{rlcl} R'': & S & \rightarrow & AbB \\ & A & \rightarrow & Ac \mid d \\ & B & \rightarrow & Bc \mid A \end{array}$$

C.3 Automatic Generation of Abstract Grammars

In order to derive abstract grammar G' from a concrete grammar G, consider the auxiliary function $\chi' : \mathcal{C} \rightarrow (\mathcal{C} \times 2^{\mathcal{R}} \times 2^{\mathcal{V}} \times 2^{\mathcal{T}})$, where $\mathcal{C}$ is the set of all rule constituents, $\mathcal{V}$ is the

set of all grammar variables, $\mathcal{R}$ is the set of all production rules, and $\mathcal{T}$ is the set of all grammar terminals. Function χ' is recursively defined as:

- $\chi'(\lambda) = (\textbf{empty}\ , \emptyset, \emptyset, \{\textbf{empty}\ \})$;
- $\chi'(\textbf{empty}) = (\textbf{empty}, \emptyset, \emptyset, \{\textbf{empty}\})$;
- $\chi'(v) = (d, \emptyset, \{d\}\ , \emptyset)$, where d is the name of the domain of variable v;
- $\chi'(t) = (d, \{d \rightarrow t\}\ , \{d\}\ , \{t\})$, where d is the name of the domain of token t, if t is defined to be in some domain;
- $\chi'(t) = (t, \emptyset, \emptyset, \{t\})$, if token t is defined to be in no domain;
- $\chi'\left(\alpha_1\alpha_2\cdots\alpha_m\right) = (\beta_1\beta_2\cdots\beta_m, R_\alpha, V_\alpha, T_\alpha)$, where:
  - $(\beta_i, R_{\alpha_i}, V_{\alpha_i}, T_{\alpha_i}) = \chi'\left(\alpha_i\right)$, for $1 \leq i \leq m$
  - $R_\alpha = \bigcup_{i=1}^{m} R_{\alpha_i}$; $\quad V_\alpha = \bigcup_{i=1}^{m} V_{\alpha_i}$; $\quad T_\alpha = \bigcup_{i=1}^{m} T_{\alpha_i}$;
- $\chi'\left(\alpha*\right) = (\beta*, R_\alpha, V_\alpha, T_\alpha)$, where $(\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)$;
- $\chi'\left(\alpha+\right) = (\beta+, R_\alpha, V_\alpha, T_\alpha)$, where $(\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)$;
- $\chi'\left(\alpha*\text{-}t\right) = (\beta*, R_\alpha, V_\alpha, T_\alpha)$, where $(\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)$;
- $\chi'\left(\alpha+\text{-}t\right) = (\beta+, R_\alpha, V_\alpha, T_\alpha)$, where $(\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)$.

The abstract grammar $G' = \chi(G) = (V', T', R', d_S)$ is defined as follows:

- $T' = \{t \in T_\alpha \mid A \rightarrow \alpha \in R \wedge (\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)\}$;
- $V' = \{v \in V_\alpha \mid A \rightarrow \alpha \in R \wedge (\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)\}$;
- $R' = R_1 \cup R_2$, where:
  - $R_1 = \{A \rightarrow \beta \mid A \rightarrow \alpha \in R \wedge (\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)\}$;
  - $R_2 = \{p \in R_\alpha \mid A \rightarrow \alpha \in R \wedge (\beta, R_\alpha, V_\alpha, T_\alpha) = \chi'(\alpha)\}$.
- d_S is the domain of variable S.

Bibliografia

Aldrich, J., Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming, *Tech. rep.*, Carnegie Mellon, 2004.

Andrade, C. A. R., A. L. M. Santos, and P. H. M. Borba, AspectH: Uma Extensão Orientada a Aspectos de Haskell, in *WASP - Workshop on Aspect Oriented Programming*, Brasília, DF, Brazil, 2004.

AspectWerks, http://aspectwerkz.codehaus.org/, 2005, Último acesso: 27 de novembro de 2005.

Atkinson, C., and T. Kühne, Aspect-Oriented Development with Stratified Frameworks, *IEEE Software*, *20*, 81–89, 2003.

Bigonha, R. S., Notas de Aulas de Semântica Denotacional, 2003, Último acesso: 27 de novembro de 2005.

Börger, E., and W. Schulte, Programmer Friendly Modular Definition of the Semantics of Java, in *Formal Syntax and Semantics of Java*, edited by J. Alves-Foss, vol. 533 of *Lecture Notes in Computer Science*, Springer, 1998.

Boute, R., Functional Declarative Language Design and Predicate Calculus: a Practical Approach, *ACM Trans. Program. Lang. Syst.*, *27*, 988–1047, 2005.

Bravenboer, M., and E. Visser, Parse Table Composition. Separate Compilation and Binary Extensibility of Grammars, in *Software Language Engineering (SLE 2008)*, edited by D. Gasevic and E. van Wyk, Lecture Notes in Computer Science, Springer, Toulouse, France, 2009, (to appear).

Cardelli, L., and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *ACM Comput. Surv.*, *17*, 471–523, 1985.

Cenciarelli, P., A. Knapp, B. Reus, and M. Wirsing, An Event-Based Structural Operational Semantics of Multi-Threaded Java, *Lecture Notes in Computer Science*, *1523*, 157–200, 1999.

Clifton, C., and G. Leavens, Observers and Assistants: A Proposal for Modular Aspect-oriented Reasoning, in *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Enschede, The Netherlands, 2002.

Coady, Y., G. Kiczales, M. Feeley, and G. Smolyn, Using AspectC to Improve the Modularity of Path-specific Customization in Operating System Code, in *Proceedings of the 8th European software engineering conference*, pp. 88–98, ACM Press, Vienna, Austria, 2001.

Costa, M. R., Compilação de um Cálculo Lambda Estendido para Supercombinadores, Master's thesis, Universidade Federal de Minas Gerais, 2000.

Crepalde, M. A., R. S. Bigonha, and F. Tirelo, Semântica Multidimensional de Java, *Tech. Rep. LLP-02/2008*, Universidade Federal de Minas Gerais, 2008a.

Crepalde, M. A., R. S. Bigonha, F. Tirelo, and T. C. do A. P. Soares, Manual do Usuário de Notus, *Tech. Rep. LLP-01/2008*, Universidade Federal de Minas Gerais, 2008b.

de Moor, O., K. Backhouse, and S. D. Swierstra, First-class Attribute Grammars, *Informatica*, *24*, 2000.

Dijkstra, E. W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.

Espinosa, D. A., *Semantic Lego*, Ph.D. thesis, Columbia University, New York, NY, USA, 1995.

Fokkinga, M., Calculate categorically!, *Formal Aspects of Computing*, *4*, 673–692, 1992.

Fokkinga, M. M., and L. Meertens, Adjunctions, *Tech. Rep. Memoranda Inf 94-31*, University of Twente, Enschede, Netherlands, 1994.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Gayo, J. E. L., Reusable Semantic Specifications of Programming Languages, in *Proceedings of the 6th Brazilian Symposium on Programming Languages*, Rio de Janeiro, Brazil, 2002.

Gordon, M. J. C., *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, 1979.

Gradecki, J. D., and N. Lesieck, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley Publishing, Inc., 2003.

Gurevich, Y., Evolving Algebras, in *IFIP 1994 World Computer Congress*, edited by B. Pehrson and I. Simon, vol. I: Technology and Foundations, pp. 423–427, Elsevier, Amsterdam, 1994.

Gurevich, Y., Evolving Algebras 1993: Lipari Guide, in *Specification and Validation Methods*, edited by E. Börger, pp. 9–36, Oxford University Press, 1995.

Hannemann, J., and G. Kiczales, Design Pattern Implementation in Java and aspectJ, in *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 161–173, ACM Press, Seattle, Washington, USA, 2002.

Harrison, W., and S. Kamin, Compilation as Metacomputation: Binding Time Separation in Modular Compilers, *Tech. rep.*, Department of Computer Science, University of Illinois, Urbana-Champaign, 2000.

Harrison, W. L., and S. N. Kamin, Modular compilers based on monad transformers, in *IEEE Computer Society International Conference on Computer Languages*, pp. 122–131, IEEE Computer Society, Chicago, USA, 1998.

Hoare, C. A. R., An Axiomatic Basis for Computer Programming, *Commun. ACM*, *12*, 576–580, 1969.

Ivanović, M., and V. Kuncak, Modular Language Specifications in Haskell, in *Proceedings of the Third International Conference on Theoretical Aspects of Computer Science with practical applications*, 2000.

Jones, M. P., and L. Duponcheel, Composing Monads, *Tech. rep.*, Yale University Computer Science Department, New Haven, Connecticut, 1993.

Jones, N. D., C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

Jones, S. L. P., and P. Wadler, Imperative Functional Programming, in *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 71–84, ACM Press, Charleston, South Carolina, United States, 1993.

Kay, A. C., The Early History of Smalltalk, in *HOPL-II: The second ACM SIGPLAN Conference on History of Programming Languages*, pp. 69–95, ACM, Cambridge, Massachusetts, United States, 1993.

Kiczales, G., Aspect-Oriented Programming – Keynote Talk at EclipseCon 2004, 2002, disponível em *http://www.cs.ubc.ca/~gregor/*. Último acesso: 23 de maio de 2004.

Kiczales, G., and E. Hilsdale, Aspect-Oriented Programming, in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, p. 313, ACM Press, Vienna, Austria, 2001.

Kiczales, G., and M. Mezini, Aspect-Oriented Programming and Modular Reasoning, in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 49–58, ACM, St. Louis, MO, USA, 2005.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, Aspect-Oriented Programming, in *Proceedings of the 11th European Conference on Object-Oriented Programming*, p. 220ff, Springer-Verlag, Jyväskylä, Finland, 1997.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, An Overview of AspectJ, in *Proceedings of the 15th European Conference on Object-Oriented Programming*, Springer-Verlag, Budapest, Hungary, 2001.

Kim, H., AspectC#: An AOSD Implementation for C#, 2002.

King, D. J., and P. Wadler, Combining Monads, in *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pp. 134–143, Springer-Verlag, London, UK, 1993.

Krishnamurthi, S., K. Fisler, and M. Greenberg, Verifying Aspect Advice Modularly, *SIGSOFT Softw. Eng. Notes*, *29*, 137–146, 2004.

Laddad, R., *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning, 2003.

Lafferty, D., and V. Cahill, Language-Independent Aspect-oriented Programming, in *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 1–12, ACM Press, Anaheim, California, USA, 2003.

Lämmel, R., Declarative Aspect-Oriented Programming, in *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, edited by O. Danvy, Technical report BRICS-NS-99-1, University of Aarhus, pp. 131–146, San Antonio, Texas, 1999.

Liang, S., Modular Monadic Semantics and Compilation, Ph.D. thesis, Yale University, 1997, adviser-Paul Hudak.

Liang, S., and P. Hudak, Modular Denotational Semantics for Compiler Construction, in *ESOP '96: Proceedings of the 6th European Symposium on Programming Languages and Systems*, pp. 219–234, Springer-Verlag, London, UK, 1996.

Liang, S., P. Hudak, and M. Jones, Monad Transformers and Modular Interpreters, in *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 333–343, San Francisco, California, United States, 1995.

Loredo, F. S., R. S. Bigonha, and F. Tirelo, Implementação de Semântica Vaga em Notus, *Tech. Rep. LLP-03/2008*, Universidade Federal de Minas Gerais, 2008.

Maia, M. A., Implementação Eficiente de uma Linguagem para Definição de Semântica, Master's thesis, Universidade Federal de Minas Gerais, 1994.

Menezes, P. B., and E. H. Haeusler, *Teoria das Categorias para Ciência da Computação*, no. 12 in Série Livros Didáticos, Instituto de Informática da UFRGS, Ed. Sagra Luzzato, 2001.

Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, New Jersey, 1997.

Moggi, E., Computational Lambda-Calculus and Monads, in *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pp. 14–23, IEEE Press, Pacific Grove, California, United States, 1989.

Moggi, E., Notions of Computation and Monads, *Inf. Comput.*, *93*, 55–92, 1991.

Mosses, P. D., Making denotational semantics less concrete, in *Proc. Int. Workshop on Semantics of Programming Languages, Bad Honnef*, no. 41 in Bericht, pp. 102–109, Abteilung Informatik, Universität Dortmund, 1977.

Mosses, P. D., The Modularity of Action Semantics, *Internal Report IR–75*, Dept. of Computer Science, Univ. of Aarhus, 1988, 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., *Action Semantics*, vol. 26 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1992.

Mosses, P. D., An Introduction to Action Semantics, in *Logic and Algebra of Specification, Marktoberdorf Summer School 1991, Proceedings*, vol. 94 of *NATO ASI Series F*, pp. 247–288, Springer, 1993.

Mosses, P. D., AN-2: Revised Action Notation—Syntax and Semantics, 2000, available at http://www.brics.dk/ pdm/papers/Mosses-AN-2-Semantics/.

Mosses, P. D., The Varieties of Programming Language Semantics (and Their Uses), in *Perspectives of System Informatics, 4th Intl. Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, Revised Papers*, edited by D. Bjørner, M. Broy, and A. V. Zamulin, vol. 2244 of *LNCS*, pp. 165–190, 2001.

Mosses, P. D., Modular Structural Operational Semantics, *J. Logic and Algebraic Programming*, *60–61*, 195–228, 2004, special issue on SOS.

Mosses, P. D., A Constructive Approach to Language Definition, *Journal of Universal Computer Science*, *11*, 1117–1134, 2005.

Mosses, P. D., and D. A. Watt, The Use of Action Semantics, in *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pp. 135–166, North-Holland, 1987.

Mosses, P. D., and D. A. Watt, Pascal Action Semantics, Version 0.6, 1993, http://www.brics.dk/ pdm/papers/MossesWatt-Pascal-AS/.

Moura, H. P., An Overview of Action Semantics, in *I Simpósio Brasileiro de Linguagens de Programação*, Belo Horizonte, MG, Brazil, 1996.

Myers, G. J., *Reliable Software through Composite Design*, Petrocelli/Charter, New York, 1975.

Nygaard, K., and O.-J. Dahl, The Development of the SIMULA Languages, *History of Programming Languages I*, pp. 439–480, 1981.

Oliveira, F. F., *Compilação de uma Linguagem Funcional Orientada por Objetos para Definição de Semântica Denotacional*, Master's thesis, Universidade Federal de Minas Gerais, 1998.

Ossher, H., and P. Tarr, Hyper/J: Multi-Dimensional Separation of Concerns for Java, in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 734–737, ACM Press, Limerick, Ireland, 2000.

Ossher, H., and P. Tarr, Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software, *Communications of the ACM*, *44*, 43–50, 2001.

Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, *15*, 1053–1058, 1972.

Plotkin, G. D., A Structural Approach to Operational Semantics, *Tech. rep.*, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

Rajan, H., and K. Sullivan, Eos: Instance-Level Aspects for Integrated System Design, *SIGSOFT Softw. Eng. Notes*, *28*, 297–306, 2003.

Rajan, H., and K. J. Sullivan, Classpects: Unifying Aspect- and Object-Oriented Language Design, in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 59–68, ACM Press, St. Louis, MO, USA, 2005.

Schmidt, D. A., Programming Language Semantics, *ACM Comput. Surv.*, *28*, 265–267, 1996.

Scott, D., Data Type as Lattices, *SIAM J. on Comp.*, *5*, 522–587, 1976.

Scott, D., and C. Strachey, Toward a Mathematical Semantics for Computer Languages, *Programming Research Group Technical Monograph PRG-6*, Oxford Univ. Computing Lab., 1971.

Shutt, J. N., Monads for Programming Languages, *Tech. rep.*, Computer Science Department, Worcester Polytechnic Institute, Worcester, Massachussets, 2003.

Sipser, M., *Introduction to the Theory of Computation*, International Thomson Publishing, 1996.

Soares, T. C. A. P., Compilação de Semântica Denotacional Modular, Master's thesis, Universidade Federal de Minas Gerais, 2007.

Spinczyk, O., A. Gal, and W. Schröder-Preikschat, AspectC++: An Aspect-oriented Extension to the C++ Programming Language, in *Proceedings of the Fortieth International Conference on Tools Pacific*, pp. 53–60, Australian Computer Society, Inc., Sydney, Australia, 2002.

Steele Jr., G. L., Building Interpreters by Composing Monads, in *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 472–492, ACM Press, Portland, Oregon, United States, 1994.

Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, USA, 1981.

Strachey, C., and C. Wadsworth, Continuations: a Mathematical Semantics Which Can Deal with Full Jumps, *Tech. rep.*, Programming Research Group, Oxford University, Oxford, 1974.

Tennent, R. D., The Denotational Semantics of Programming Languages, *Communications of the ACM*, *19*, 437–453, 1976.

Tirelo, F., M. A. Maia, V. O. D. Iorio, and R. S. Bigonha, Máquinas de Estado Abstratas, in *Anexo do III Simpósio Brasileiro de Linguagens de Programação*, Porto Alegre, RS, Brazil, 1999.

Tirelo, F., R. da Silva Bigonha, M. da Silva Bigonha, and M. T. de Oliveira Valente, Desenvolvimento de Software Orientado por Aspectos, *Anexo da XXIII Jornada de Atualização em Informática, XXIV Congresso da Sociedade Brasileira de Computação*, 2004.

Tirelo, F., R. S. Bigonha, and J. Saraiva, Disentangling Denotational Semantics Specifications, *Journal on Universal Computer Science*, 2009, (a ser publicado).

Tucker, D. B., and S. Krishnamurthi, Pointcuts and Advice in Higher-Order Languages, in *Proceedings of the Aspect-Oriented Software Development Conference (AOSD 2003)*, pp. 158–167, ACM Press, Boston, MA, USA, 2003.

Turi, D., and G. Plotkin, Towards a Mathematical Operational Semantics, in *LICS '97: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, p. 280, IEEE Computer Society, Washington, DC, USA, 1997.

von Staa, A., *Programação Modular*, Editora Campus, Rio de Janeiro, 2000.

Wadler, P., Comprehending Monads, in *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pp. 61–78, ACM Press, Nice, France, 1990.

Wadler, P., The Essence of Functional Programming, in *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 1–14, ACM Press, Albuquerque, New Mexico, United States, 1992.

Wadler, P., Monads for Functional Programming, in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pp. 24–52, Springer-Verlag, London, UK, 1995.

Wadler, P., How to Declare an Imperative, *ACM Comput. Surv.*, *29*, 240–263, 1997.

Walker, D., S. Zdancewic, and J. Ligatti, A Theory of Aspects, in *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pp. 127–139, ACM Press, Uppsala, Sweden, 2003.

Wand, M., G. Kiczales, and C. Dutchyn, A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming, *ACM Trans. Program. Lang. Syst.*, *26*, 890–910, 2004.

Wansbrough, K., and J. Hamer, A Modular Monadic Action Semantics, in *Proceedings of the Conference on Domain-Specific Languages, Santa Barbara, California*, pp. 157–170, The USENIX Association, 1997.

Watt, D. A., An Action Semantics of Standard ML, in *Proceedings of the 3rd Workshop on Mathematical Foundations of Programming Language Semantics*, pp. 572–598, Springer-Verlag, London, UK, 1988.

Watt, D. A., JOOS Action Semantics, *Tech. rep.*, Department of Computer Science, University of Glasgow, Glasgow, Scotland, 1997.

Zhang, Y., and B. Xu, A Survey of Semantic Description Frameworks for Programming Languages, *SIGPLAN Not.*, *39*, 14–30, 2004.