**Geração parcial de código Java
a partir de especificações formais Z**

Alvaro Heiji Miyazawa

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Programa: Ciência da Computação
Orientador: Profa. Dra. Ana Cristina Vieira de Melo

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, 29 agosto de 2008

Geração parcial de código Java a partir de especificações formais Z

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Alvaro Heiji Miyazawa
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof. Dr. Flávio Soares Corrêa da Silva - IME-USP.
- Prof. Dr. Marco Dimas Gubitoso - IME-USP.
- Prof. Dr. Marcel Vinicius Medeiros Oliveira - DIMAP-UFRN.

Agradecimentos

Agradeço inicialmente à minha mãe, Inês Akiko Miyazawa, que sempre me ajudou e incentivou, e à minha orientadora, Profa. Dra. Ana Cristina Vieira de Melo, por todo o incentivo, ajuda e paciência.

Agradeço aos membros da banca examinadora da minha defesa de dissertação de mestrado, Prof. Dr. Flávio Soares Corrêa da Silva, Prof. Dr. Marco Dimas Gubitoso e Prof. Dr. Marcel Vinicius Medeiros Oliveira, pelos comentários e sugestões, ao grande amigo Márcio José de Medeiros pelas inúmeras discussões sobre os mais diversos assuntos, pelo companheirismo e apoio.

Agradeço ao amigo Paulo Salem por toda a ajuda e discussões durante o mestrado, aos amigos Jony Arrais e Patrícia Viana que se tornaram grandes amigos nesse pouco tempo que nos conhecemos, à minha prima Renata Matteoni pelas várias conversas que tivemos e ao Schummi pelas várias brigas que tivemos.

Agradeço ainda à minha família, aos amigos que fiz no Instituto de Matemática e Estatística (IME-USP) e na Faculdade de Filosofia, Letras e Ciências Humanas (FFLCH-USP), aos professores do IME e da FFLCH.

Resumo

Especificações formais são úteis para descrever o que um sistema deve fazer sem definir como, e, em virtude da sua natureza formal e da possibilidade de abstração, é possível analisá-las sistematicamente. No entanto, o uso de especificações formais como parte do desenvolvimento de software não constitui prática comum. Isso se dá, em parte, pelo fato de existirem apenas um pequeno número de metodologias e ferramentas adequadas que dêem suporte a esse desenvolvimento. O primeiro objetivo deste trabalho é propor uma metodologia de desenvolvimento que possibilite, a partir de uma especificação formal em notação Z, produzir uma implementação dessa especificação em Java. Essa metodologia centra-se na geração do esqueleto da aplicação Java e na instrumentação desse esqueleto com mecanismos de verificação de condições (invariantes, pré e pós-condições) e rastreamento de violações dessas condições. Através desses mecanismos, possibilita-se intercalar desenvolvimento formal e informal no processo global de desenvolvimento de software. O segundo objetivo é desenvolver uma ferramenta que implemente parte dessa metodologia, produzindo uma implementação parcial que deverá ser complementada pelo usuário.

Palavras-chave: desenvolvimento de software, especificação formal, desenvolvimento baseado em modelo, linguagem Java, notação Z, geração de código.

Abstract

Formal specifications are useful for describing what a system should do, without defining how, and, owing to its formal nature, it is possible to analyse them systematically. However useful formal specifications are, their usage as part of the software development process is rather rare. This is, in part, due to the scarcity of both methodologies and tools that support this development. The first goal of this work is to define a software development methodology that enables the developer to produce a Java application from a formal specification written in Z. This methodology will rely strongly on the generation of Java application skeletons and instrumentation of the generated code with means of verifying conditions (invariants, pre and post-conditions) e tracing violations of these conditions. Through this mechanisms, it is possible to mix formal and informal development in the global software development process. The second goal of this work is to develop a tool that will implement part of this methodology, producing a partial implementation that must be complemented by the developer.

Keywords: software development, formal specification, model-driven development, Java language, Z notation, code generation.

Sumário

Figuras	xi
1 Introdução	1
1.1 Objetivos	4
2 Metodologia	5
2.1 Desenvolvimento Baseado em Modelos	5
2.2 Especificação	6
2.3 Notação Z	7
2.4 Uma Metodologia de Desenvolvimento Baseado em Modelos Formais	14
2.4.1 Princípios	15
2.4.2 Arcabouço	16
2.4.3 Relacionamento entre Código e Especificação	18
2.4.4 Correção	19
3 Arcabouço Matemático	23
3.1 Conjuntos	25
3.1.1 Definição do Tipo de Dados	25
3.1.2 Definição da Interface	27
3.1.3 Implementação	27
3.2 Relações	30
3.2.1 Definição do Tipo de Dados	30
3.2.2 Definição da Interface	31
3.2.3 Implementação	34
3.3 Funções Parciais	38
3.3.1 Definição do Tipo	38
3.3.2 Definição da Interface	42
3.3.3 Implementação	42
4 Transformação de Estados	53
4.1 Representação de Estados	53
4.2 Transformação em Código Java	55

4.2.1	Esquemas Simples	55
4.2.2	Esquema com Restrição	56
4.2.3	Esquemas com Inclusão	57
4.2.4	Esquemas Compostos	58
4.3	Correção da Transformação de Esquemas Simples	61
4.3.1	Semântica de Esquemas	62
4.3.2	Semântica da Definição de Classe em Java	67
4.3.3	Comparação entre os Modelos	67
5	Transformação de Operações	69
5.1	Representação de Operações	70
5.1.1	Convenções	70
5.1.2	Cálculo de Esquemas	72
5.2	Transformação em Código Java	72
5.2.1	Operação Geral	73
5.2.2	Inclusão de Esquemas Δ	73
5.2.3	Variáveis	74
6	Tradução de Expressões e Predicados	77
6.1	Ferramenta	82
7	Conclusão	87
7.1	Trabalhos Futuros	88
A	Linguagem lógica	89
A.1	Lógica Proposicional	89
A.2	Lógica de Predicados	90
B	Linguagem de conjuntos	93
B.1	Tipos	94
B.2	Conjuntos em geral	95
B.3	Domínios	98
C	Linguagem Matemática	99
C.1	Relações	99
C.2	Funções	102
C.3	Números	104
C.4	Seqüências	105
D	Regras de Tradução	109
E	Implementação da classe BasicSet	119

F	Implementação da classe BasicRelation	123
G	Implementação da classe BasicPFunction	133
	Referências Bibliográficas	141

Figuras

2.1	Estrutura hierárquica do arcabouço matemático	8
2.2	Exemplo de estrutura hierárquica de um especificação	8
2.3	Visão geral da metodologia proposta ¹	15
3.1	Exemplo de uma especificação composta de diversas seções	23
3.2	Hierarquia de sub arcabouços do arcabouço matemática	24
3.3	Elementos do arcabouço matemática tratados	26
3.4	Interface Java para classes que implementam conjuntos.	28
3.5	Classe BasicSet que implementa a interface ZSet	28
3.6	Implementação das relações de conjuntos	29
3.7	Implementação das operações de união e intersecção	29
3.8	Implementação das versões generalizadas de união e intersecção	29
3.9	Implementação das outras operações	30
3.10	Classe BasicRelation que implementa a interface ZRelation	34
4.1	Exemplo de um domínio	54
4.2	Exemplo de um construtor de domínio	54
4.3	Exemplo de um esquema	54
4.4	Exemplo de um esquema horizontal	55
4.5	Exemplo de um esquema simples	55
4.6	Exemplo de esquemas	59
4.7	Conjunção de esquemas	59
4.8	Disjunção de esquemas	59
4.9	Implicação de esquemas	59
4.10	Equivalência de esquemas	59
4.11	Negação de esquemas	60
4.12	Quantificação universal de esquemas	60
4.13	Domínio simples, onde i é um nome e e uma expressão que denota um conjunto	62
4.14	Construtor de domínio equivalente ao esquema da Figura 4.13	62
4.15	Interpretação de construtores de domínio simples	62
4.16	Construtor de domínio composto	63
4.17	Construtor de domínio composto transformado	63

4.18	Interpretação de construtores de domínio compostos	63
4.19	Construtor de domínio simples 1	63
4.20	Construtor de domínio simples 2	63
4.21	Aplicação de $\mathcal{I}$ aos construtores de domínio das Figuras 4.19 e 4.20	63
4.22	Aplicação de $\mathcal{I}$ à conjunção dos construtores de domínio das Figuras 4.19 e 4.20	63
4.23	Função $\mathcal{I}$ estendida a construtores de domínio simples com declarações constantes	63
4.24	Aplicação da função $\mathcal{I}$ a um construtor de domínio simples com declaração constante	64
4.25	Aplicação da função $\mathcal{I}$ a construtor de domínio geral	64
4.26	Aplicação da função $\mathcal{I}$ a um domínio	64
4.27	Esquema Geral	64
4.28	Descrição axiomática equivalente a um esquema	65
4.29	Descrição axiomática simplificada	65
4.30	Exemplo de esquema descrevendo um ponto restrito a uma determinada região	65
4.31	Transformações sintáticas do esquema	66
4.32	<i>PontoNoroeste1</i>	67
4.33	<i>PontoNoroeste2</i>	67
4.34	Exemplo de esquema simples	68
4.35	Classe candidata a mapeamento do esquema da Figura 4.34.	68
5.1	Esquema de estado <i>Ponto</i> e esquema de operação <i>Esquerda</i>	70
5.2	Esquema que especifica uma reta	70
5.3	Esquema decorado <i>Reta'</i>	70
5.4	Resultado da expressão $\theta Reta$	71
5.5	Esquema de estado <i>Ponto</i> e esquema de operação <i>Esquerda</i>	73
6.1	Construção do mapeamento de tipos	83
6.2	Geração de código	83
6.3	Criação do projeto JZed	84
6.4	Seleção de especificações	84
6.5	Seleção da especificação <i>BirthdayBook</i>	85
6.6	Seleção dos tipos NAME e DATE	85
6.7	Detalhes da geração do código	86
6.8	Exemplo de código gerado	86

Capítulo 1

Introdução

Atualmente, cada vez mais produtos englobam, de uma forma ou de outra, computadores. Além disso, os mais diversos serviços (saúde, educação, etc) dependem de sistemas computacionais, e o software representa grande parte do custo desses sistemas. Assim, o desenvolvimento efetivo de software é um problema relevante para o mundo atual [49], o que coloca o processo de desenvolvimento de software em posição de destaque.

De forma bastante genérica, esse processo pode ser descrito pelas seguintes fases: definição do problema, análise de requisitos, especificação, *design*, implementação, teste, operação e manutenção [58]. A dependência existente entre as diferentes fases [49] faz com que erros inseridos em uma fase propaguem-se para as fases seguintes.

A fim de atenuar tanto a quantidade de trabalho necessária no desenvolvimento, quanto a quantidade de erros propagados e inseridos, pode-se usar processos automatizados de desenvolvimento. Porém, os processos conhecidos não abrangem todo o espectro de atividades envolvidas no desenvolvimento de software.

No entanto, as fases de especificação, *design*, implementação e teste são, em parte, suscetíveis a automatização. Em especial, se os produtos dessas fases forem de natureza formal.

Uma classe de processos de desenvolvimento que se aproveita desse fato são conhecidos, de forma genérica, como desenvolvimento baseado em modelos, que consiste na aplicação de modelos e tecnologias relacionadas para aumentar o nível de abstração no qual os desenvolvedores criam e evoluem o software, a fim de simplificar e formalizar as diversas atividades e tarefas de desenvolvimento [20]. Uma instância em particular desse tipo de processo é a arquitetura baseada em modelos que, entre outras características, distingue entre modelos dependentes de plataforma, isto é, que levam em conta aspectos da plataforma onde o sistema será executado, e independentes de plataforma [20]. Tradicionalmente, os modelos usados nesse processo são definidos em UML, uma linguagem visual de especificação para modelagem de objetos. Porém, existem diversas críticas ao uso dessa linguagem, dentre elas o tamanho e complexidade excessivos da linguagem e o fato da semântica de modelos descritos em UML não ser precisamente definida [56].

Esse processo de desenvolvimento baseado em modelos UML possibilita a automatização parcial de tarefas comuns, porém não aborda a inserção e propagação de erros, visto que não é possível analisar formalmente sistemas especificados em UML, devido à falta de semântica definida. Vale

notar que existem iniciativas que visam a dar semântica formal à notação UML [17]. Porém, essas iniciativas são apenas parciais.

Esse trabalho se propõe a definir uma metodologia de desenvolvimento de software inspirada no desenvolvimento baseado em modelos, usando como linguagem de especificação a notação Z [62] e como linguagem de implementação a linguagem Java [55]. Além da metodologia, pretendemos implementar uma ferramenta que dê suporte a ela, automatizando-a parcialmente.

A escolha da notação Z como linguagem de especificação foi feita pois, além de ser amplamente utilizada, ela possui um padrão internacional que define sua sintaxe, semântica e sistema de tipos e, devido a esse caráter formal da linguagem, é possível analisar as especificações descritas nela, evitando que eventuais erros introduzidos na fase de especificação se propaguem para as outras fases. Além disso, existem analisadores sintáticos para Z [1], verificadores de tipos [1, 2, 5], verificadores de modelos [48], simuladores [4], um cálculo de refinamento [10] e provadores de teoremas [3, 44]. A linguagem Java, por sua vez, foi escolhida para servir de base para a implementação devido ao seu amplo uso por desenvolvedores de software e grande quantidade de ferramentas de desenvolvimento.

Existem algumas iniciativas com objetivos semelhantes aos nossos, por exemplo, Ramkarthik e Zhang [43] propõem a existência de uma relação bijetora entre um esquema de classe em Object-Z [47] e o esqueleto de uma classe Java associado aos invariantes do estado da classe e às pré e pós-condições dos métodos. Essa abordagem difere da metodologia aqui proposta na medida em que se aplica a um subconjunto de Object-Z, o que impõe uma especificação estruturada segundo o paradigma da orientação a objetos. Vale salientar que desconhecemos ferramentas que dêem suporte a tradução automática de especificações Object-Z para código executável.

Além dessa abordagem, existem outras que visam, em geral, à prototipagem ou à animação de especificações formais. A animação de especificações consiste na construção de um sistema que permite ao usuário interagir com a especificação, possibilitando a validação de uma especificação antes de implementá-la [29].

Essas abordagens freqüentemente se baseiam em algum interpretador de Z, ou na tradução da especificação para alguma linguagem lógica ou funcional. Por exemplo, Abdallah et al. [6] descreve uma maneira formal de construir protótipos corretos em uma linguagem funcional, já Winikoff et al. [59], descreve a geração de protótipos em uma linguagem lógica (Mercury).

A animação de especificações, em geral, apresenta o problema de limitar a interação do usuário com o sistema. Por outro lado, a produção de protótipos em linguagem lógica ou funcional não sofre desse problema. Porém não é completamente adequada ao desenvolvimento prático de software, pois os protótipos são gerados em linguagens que não são amplamente usadas, de forma que a implementação final não pode se aproveitar diretamente desses protótipos, na medida em que, em geral, ela é desenvolvida em alguma linguagem procedural (C, Pascal) ou orientada a objeto (Java, Delphi, C++).

Outro tipo de abordagem existente consiste na definição de uma nova linguagem que visa a permitir a geração de código de forma mais simples. Por exemplo, Davies et al. [13] apresentam uma abordagem para a geração de código a partir de especificação formal escrita em uma linguagem

inspirada em Z, B [7] e no cálculo de refinamentos [9, 36]. Em [15], a abordagem descrita no artigo anterior é detalhada para a geração de bases de dados de objetos. Esses dois trabalhos diferem da nossa proposta, pois criam uma nova linguagem de especificação, além de se restringirem a domínios particulares.

Em geral, as abordagens citadas propõem a construção de parte da implementação ou assumem uma especificação bastante concreta, isto é, uma especificação que descreva como chegar a um estado particular e não apenas que estado se deseja alcançar. Propostas que não fazem essas suposições, em geral, baseiam-se em refinamento ou geração dedutiva de código. O refinamento [36] consiste na transformação, passo a passo, de uma especificação que introduz cada vez mais detalhes, visando a obter uma especificação suficientemente concreta que possa ser traduzida em código executável. A geração dedutiva de código [31, 32], consiste em converter uma especificação em um teorema matemático, tal que sua prova, dentro de uma lógica particular, poderia ser convertida em um programa que implementa a especificação inicial. Não é de nosso conhecimento trabalhos que proponham a implementação de uma especificação Z através de técnicas de geração dedutiva de código.

No entanto, existem algumas abordagens baseadas na cálculo de refinamento. Wood [60] propõe uma metodologia para o desenvolvimento de sistemas usando Z e cálculo de refinamentos. Porém esse trabalho propõe o uso de uma notação diferente de Z, composta do cálculo de refinamentos de Morgan [36] e um subconjunto de Z. Essa metodologia difere da apresentada neste trabalho, na medida em que impõe uma nova notação, o que impede a aplicação direta dela a especificações existentes. Por outro lado, Cavalcanti e Woodcock [10] propõem um cálculo de refinamentos baseado no trabalho de Morgan que incorpora a notação Z. Freitas et al. [16], a partir do trabalho anterior, apresenta uma ferramenta¹ para a mecanização do cálculo de refinamentos ZRC. Embora uma ferramenta que auxilie o refinamento de especificações formais seja desejável, ainda implica em uma grande quantidade de manipulações manuais. Assim, é desejável uma ferramenta que automatize ao máximo o processo de geração de código, minimizando os trechos de especificação que precisam ser refinados manualmente.

Vale notar que existem abordagens que tentam diminuir o trabalho necessário para refinar uma especificação através da definição de táticas de refinamento [38–40]. No entanto, nenhum deles lida com especificações Z padrão.

De acordo com [43], até sua data de publicação, não havia uma ferramenta que traduzisse diretamente um especificação em Object-Z para Java. Igualmente, através das busca efetuadas para esse trabalho, não há uma ferramenta que traduza especificação Z para Java. Além disso, uma pesquisa de ferramentas para notação Z, [53], efetuada em 1994 mostra que havia apenas duas ferramentas que proviam algum suporte para refinamento: ProofPower e CADiZ. Existem ainda uma série de ferramentas para animação da notação Z, em geral um subconjunto dela, tais como ZANS [26], Possum [21] e ZETA(Zap) [19], além de outras ferramentas citadas anteriormente, tais como Z/EVES e CZT.

¹Essa ferramenta não foi encontrada

Existem ainda outras linguagens de especificação que dão suporte à geração automática de código, tais como B [7], VDM [11, 27] entre outras. Nosso trabalho se distingue delas na medida em que permite e tenta dar suporte a utilização de técnicas de desenvolvimento formal junto ao desenvolvimento informal.

1.1 Objetivos

Este trabalho visa a definir uma metodologia de desenvolvimento baseado em modelos formais, em oposição às metodologias que se baseiam em modelos não formais. As principais características que distinguem essa proposta de outros trabalhos abordam o desenvolvimento baseado em modelos são:

- Os modelos que servem de base para essa metodologia são descritos na linguagem formal Z, ao contrário de grande parte das metodologias que se baseiam em modelos informais;
- As aplicações desenvolvidas através dessa metodologia serão implementadas na linguagem Java, uma linguagem de amplo uso tanto na indústria, quanto na academia. Grande parte das abordagens para o desenvolvimento de aplicações a partir de especificações formais produz código em linguagens lógicas ou funcionais, que não são amplamente utilizadas;
- A transformação das especificações em código Java regidas por princípios que visam a permitir a utilização de métodos de desenvolvimento informais, de forma a possibilitar a prototipagem rápida de sistemas;
- A aplicação obtida através dessa metodologia reproduz a estrutura da especificação original, possibilitando o rastreamento de eventos da aplicação de volta à especificação.

Um segundo objetivo deste trabalho é o desenvolvimento de uma ferramenta que implementa parte da metodologia proposta, facilitando, assim, o rápido desenvolvimento de aplicações.

Capítulo 2

Metodologia

A metodologia proposta neste trabalho se baseia no processo de desenvolvimento de software, denominado desenvolvimento baseado em modelos, associado à especificação formal de sistemas, atribuindo assim ao processo global um caráter formal, porém prático, que possibilita a aplicação de técnicas formais de desenvolvimento. Essa metodologia é pragmática na medida em que apenas faculta ao desenvolvedor o uso de métodos formais de desenvolvimento, não os impondo, e apresenta meios que permitem a automatização de parte do processo de desenvolvimento, livrando o desenvolvedor do ônus de processos repetitivos facilmente automatizáveis.

As principais características da metodologia proposta são o uso de especificações formais, as regras de transformação de especificações em código Java, o uso de uma linguagem de programação amplamente utilizada na academia e indústria para a implementação dos sistemas, e a possibilidade de rastreamento de eventos da implementação através da estrutura imposta pelas regras de transformação.

Na Seção 2.1 apresentamos uma visão geral sobre o desenvolvimento baseado em modelos e a arquitetura baseada em modelos. Na Seção 2.2, abordamos a especificação de sistemas de software, e, na Seção 2.3 apresentamos a notação Z, utilizada para especificar formalmente sistemas de software. Na Seção 2.4, descrevemos a metodologia, seus princípios e o arcabouço que dá suporte a ela.

2.1 Desenvolvimento Baseado em Modelos

A abordagem denominada desenvolvimento baseado em modelos (*Model-Driven Development* - MDD) consiste no uso de modelos e tecnologia relacionadas, com o objetivo de simplificar e formalizar o processo de desenvolvimento de software. Propõe-se atingir esses objetivos através da elevação do nível de abstração utilizado no processo de desenvolvimento [20].

Para o desenvolvimento baseado em modelos, um modelo é uma abstração de um produto de software relacionada de forma explícita a uma linguagem de modelagem [20]. Por exemplo, podemos modelar um sistema de software através de diagramas UML, ou através de código Java, dessa forma temos dois modelos distintos, cada um associado a uma linguagem em particular. A principal diferença entre os dois modelos é o nível de abstração, enquanto o modelo descrito em Java é executável, o modelo UML não o é.

Embora os dois modelos exemplificados acima sejam bastante diferentes, sobretudo quanto ao

nível de abstração, ambos são partes fundamentais do processo de desenvolvimento de software baseado em modelos, pois podemos partir de uma representação inicial bastante abstrata (UML) e obter uma implementação (Java) correspondente, através de uma série de transformações. Para que esse processo que aumenta o nível de concretude da representação seja interessante, é necessário que consigamos estabelecer um relacionamento entre o objeto de uma transformação e seu produto, mais precisamente entre os elementos deles [20].

Da forma como é especificado, o desenvolvimento baseado em modelos é um processo de desenvolvimento bastante geral, que não define as linguagens de modelagem, os processos de transformação de modelos etc. Uma instância particular desse processo de desenvolvimento é a chamada arquitetura baseada em modelos [33, 42], que adota como principal linguagem de modelagem a notação UML. A arquitetura baseada em modelos (MDA) foi definida pelo *Object Management Group, Inc.* (OMG) e propõe a separação de uma especificação em dois tipos de modelos, um independente de plataforma e outro específico. Esses dois modelos são aspectos diferentes de um mesmo sistema, a separação deles flexibiliza o processo de desenvolvimento na medida em que um mesmo modelo independente de plataforma pode ser usado com diversos modelos dependentes de plataforma. Porém, é necessário saber como esses modelos se relacionam, para isso o processo de desenvolvimento exige uma descrição de como o modelo base é implementado em uma dada plataforma específica [41].

Como mencionado anteriormente, a arquitetura baseada em modelos foca fortemente no uso de UML como linguagem de modelagem. Porém, na tentativa de tornar UML uma linguagem multi-uso, criou-se uma linguagem muito complexa, ao ponto de prejudicar sua adoção [20]. Além disso, segundo [57], UML 2.0 não possui uma implementação de referência, nem uma descrição (para o usuário) de sua semântica, o que torna difícil interpretar e implementar corretamente ferramentas de transformação de modelos UML.

2.2 Especificação

Como mencionado no Capítulo 1, uma das fases do processo de desenvolvimento de software é a especificação do sistema. Essa fase é precedida da análise de requisitos [28, 50] que proporciona a compreensão do problema a ser modelado [12].

O processo de especificação produz uma descrição do problema a ser implementado. É desejável que tal descrição seja feita de forma clara e precisa. Porém, é comum o uso de linguagem natural para descrever os problemas [12]. Tomemos o Exemplo 2.2.1 que descreve uma agenda de aniversários [52].

Exemplo 2.2.1

Desejamos um sistema que associa ao nome de uma pessoa, seu aniversário. Para manipular esse sistema, desejamos poder adicionar uma nova pessoa e seu aniversário, encontrar o aniversário de uma dada pessoa e ser lembrado das pessoas que fazem aniversário hoje.

Note que a descrição apresentada não define uma série de coisas, como por exemplo, o que

significa adicionar, encontrar ou lembrar. Existem diversas formas de descrever esse problema de forma mais clara e precisa. Podemos, por exemplo, usar a lógica clássica como linguagem de especificação [12]. Porém, isso dificulta a especificação de sistemas computacionais complexos. Para isso, podemos utilizar diversos métodos como Z [52, 62], B [7, 63] e VDM (Vienna Development Method) [27], que enriquecem a linguagem da lógica clássica com elementos que facilitam a especificação de sistemas mais complexos.

Ao especificar um sistema, precisamos, inicialmente, decidir como representar as informações contidas nele, isto é, o estado do sistema. No caso do Exemplo 2.2.1, precisamos decidir como representar as pessoas, os aniversários e a associação entre eles.

Note que, em nenhum momento, a especificação utiliza qualquer informação interna a pessoas e aniversários, de forma que podemos abstrair esses elementos sem nos preocupar com sua estrutura interna. No nosso exemplo, vamos abstrair uma pessoa pelo seu nome e representar o aniversário como uma data. Resta decidirmos como representar a associação entre nomes e datas. Uma forma possível é modelar essa associação como uma relação matemática. Porém, essa especificação pode ser reforçada, visto que ela permite associar mais de uma data a um determinado nome. Por isso, podemos modelar a associação como uma função parcial (*aniversario*), dessa forma, um nome está associado a no máximo uma data. É interessante representar também o conjunto de pessoas que estão registradas na agenda. Nessa caso, temos uma redundância de informação e será necessário adicionar a restrição de que o domínio da função que associa pessoas a aniversários seja igual ao conjunto das pessoas contidas na agenda.

Até aqui, definimos como desejamos representar o estado do sistema. Agora, podemos descrever as operações listadas no Exemplo 2.2.1. Por exemplo, adicionar uma novo nome e uma data à agenda é uma operação que recebe um nome e uma data e altera o estado de forma que a função parcial *aniversario* aplicada ao novo nome, devolva a data correspondente. Essa operação transforma o estado do sistema e pode ser descrita como a relação entre o estado do sistema antes e depois da operação.

2.3 Notação Z

A notação Z é uma linguagem de especificação formal baseada em teoria dos conjuntos e lógica matemática. Ela se insere no conjunto das linguagens orientadas a modelos, tais como B [7, 63] e VDM [27] e vem sendo estudada desde da década de 1970. Existe uma vasta literatura sobre a linguagem Z [25, 51, 52, 62] e trabalhos que abordam a notação Z têm sido publicados nas principais conferências de métodos formais e engenharia de software. Além disso, há uma série de conferências dedicadas à notação Z:

- *Z User Meeting* (ZUM): 1986-1988, 1990-1995, 1997-1999, 2006, 2007;
- *Formal Specification and Development in Z and B* (ZB): 2000, 2002, 2003, 2005;
- *Abstract State Machines, B and Z* (ABZ): 2008.

A notação Z pode ser dividida em quatro linguagens básicas: linguagem lógica, linguagem de conjuntos, linguagem matemática e linguagem de esquemas. As linguagens lógica e de conjuntos consistem basicamente na lógica proposicional e de predicados de primeira ordem e na teoria de conjuntos Zermello-Fraenkel e serão apresentadas quando necessário ao longo do texto (para uma visão geral da linguagem lógica vide Apêndice A e da linguagem de conjuntos vide Apêndice B). A linguagem matemática define uma série de elementos bastante usados em especificações, tais como relações, funções e seqüências e será apresentada parcialmente ao longo do Capítulo 3 (para uma visão geral da linguagem matemática vide Apêndice C). A linguagem de esquemas provê um mecanismo de estruturação de especificações, sendo uma das principais características da linguagem Z e será apresentada ao longo dos Capítulos 4 e 5.

Existe um padrão internacional que rege o uso da notação Z [24]. Esse padrão identifica dois níveis de estruturação de uma especificação, um com base na linguagem de esquemas e outro com base em uma hierarquia de seções.

Uma especificação consiste em um conjunto de seções que são relacionados hierarquicamente. Cada seção pode ser básica ou herdar as definições de alguma outra seção. O padrão internacional define o prelúdio como uma seção básica que é herdada implicitamente por qualquer outra seção, inclusive as básicas. Com base no prelúdio, o padrão internacional define ainda um conjunto de seções denominado arcabouço matemático que corresponde à linguagem matemática mencionada anteriormente. A estrutura do arcabouço matemático é apresentada na Figura 2.1, onde cada retângulo representa uma seção de especificação. A Figura 2.2 ilustra um exemplo de estruturação de especificação.

Figura 2.1: Estrutura hierárquica do arcabouço matemático

Figura 2.2: Exemplo de estrutura hierárquica de uma especificação

Note que é possível construir uma especificação composta de apenas uma seção não nomeada. Nesse caso, essa seção é denominada anônima e herda implicitamente o arcabouço matemático.

Cada seção é composta por um conjunto de parágrafos. Podemos identificar quatro tipos principais de parágrafos:

- Parágrafos que introduzem conjuntos abstratos;
- Parágrafos que introduzem tipos livres;
- Parágrafos que introduzem nomes globais e restrições;
- Parágrafos que introduzem nomes locais e restrições.

O primeiro tipo de parágrafo é denominado declaração de tipos dados e o segundo declaração de tipos livres. O terceiro tipo é dito descrição axiomática e o último esquema. Esses dois últimos tipos de parágrafos possuem, ainda, versões genéricas, as quais são parametrizadas.

O segundo nível de estruturação se dá através da linguagem de esquemas, que possibilita a definição de estados e operações. Um esquema, em Z, define um conjunto de nomes e restrições sobre os valores que podem ser atribuídos a esses nomes. Um esquema pode incluir outros esquemas (ver Seção 4.2.3), ou ser formado através de conectivos do cálculo de esquemas (ver Seção 4.2.4).

A rigor, não existem esquemas de estados e esquemas de operações. De forma que a representação de uma operação é feita através de uma série de convenções¹.

- Esquemas podem ser decorados com um apóstrofe, representando o estado final de uma operação, i.e, o resultado da aplicação da operação ao estado inicial;
- Esquemas podem ser prefixados com a letra grega maiúscula Delta (Δ), representando os estados inicial e final;
- Esquemas podem ser prefixados com a letra grega maiúscula Ksi (Ξ), representando os estados inicial e final e a restrição de que os valores deles são iguais, isto é, que não há alteração dos estados.
- Nomes de variáveis podem ser sufixados por um ponto de interrogação, indicando variáveis de entrada;
- Nomes de variáveis podem ser sufixados por um ponto de exclamação, indicando variáveis de saída;

Para facilitar a exposição vamos nos referir a um esquema como esquema de operação quando ele seguir alguma das convenções citadas e como esquema de estado caso contrário. Isto é, um esquema de operação inclui um ou mais esquemas decorados com apóstrofes ou prefixados pelas letras Δ ou Ξ , e/ou possui variáveis de entrada ou saída.

¹Para uma visão mais detalhada dessas convenções vide Capítulos 4 e 5

Embora essas sejam apenas convenções, elas são amplamente usadas e alguns operadores definidos no padrão internacional as utilizam. Elas serão mais bem detalhadas no Capítulo 5, que trata das operações. De forma bastante simplificada, uma operação modifica um ou mais esquemas de estados.

Exemplo 2.3.1

Nesse exemplo, apresentamos a especificação em Z do Exemplo 2.2.1. Essa especificação foi extraída de [52]. Ela é composta de um única seção anônima, e, portanto, herda implicitamente o arcabouço matemático. Ela declara, inicialmente, dois tipos dados: *NAME* e *DATE* que representam respectivamente nomes e datas de aniversário.

$$[NAME, DATE]$$

Essa declaração apenas introduz os nomes desses conjuntos, sem explicitar quais são seus elementos ou apresentar qualquer informação sobre a estrutura interna desses conjuntos.

Em seguida, a especificação define o estado principal do sistema. Esse estado é composto de um conjunto de nomes que armazena todos os nomes contidos na agenda de aniversários. Para armazenar as datas de aniversário, utiliza-se uma função parcial que mapeia nomes em datas. Seria possível utilizar uma relação para isso, porém essa opção permite associar mais que uma data de aniversário a cada nome, o que não é necessário, visto que cada pessoa possui apenas uma data de aniversário. Note que existe uma replicação de informação nessa definição de estado, pois todo nome que está no conjunto de nomes deve estar no domínio da função parcial e vice-versa, esse fato é contemplado por um invariante de estado que afirma que o conjunto de nomes (*known*) é igual ao domínio da função de aniversários (*birthday*). Esse estado é representado pelo seguinte esquema.

<i>BirthdayBook</i>
<i>known</i> : $\mathbb{P} NAME$
<i>birthday</i> : $NAME \rightarrow DATE$
<i>known</i> = dom <i>birthday</i>

Antes de podermos manipular a agenda de aniversários é necessário saber como inicializá-la. Caso contrário, ela pode conter qualquer coisa inicialmente. O esquema *InitBirthdayBook* especifica o estado inicial do sistema de forma que o conjunto de nomes conhecidos seja vazio. Isso implica, devido ao invariante do estado, que a função *birthday'* também é vazia.

<i>InitBirthdayBook</i>
<i>BirthdayBook</i> '
<i>known'</i> = {}

Esse esquema inclui uma versão decorada do esquema *BirthdayBook*. Note que a decoração do esquema implica na decoração de todas as variáveis do esquema, tanto na declaração, quanto no

predicado.

Agora, é possível definir as operações que manipulam o sistema. O esquema *AddBirthday* define a operação que altera o estado do sistema, indicado pela inclusão do nome do esquema *BirthdayBook* prefixado pela letra Δ . Ela recebe duas variáveis de entrada, indicada pelo ponto de interrogação após o nome.

<i>AddBirthday</i>
$\Delta \textit{BirthdayBook}$
$name? : NAME$
$date? : DATE$
$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

O predicado dessa operação afirma que o nome recebido pela operação não pode estar no conjunto de nomes conhecidos e que, após a operação, a variável *birthday'*, i.e. *birthday'*, deve ser igual à variável *birthday* antes da operação adicionada da associação entre o nome e a data recebidos. Esse predicado pode ser particionado em duas partes, uma que faz referência apenas a variáveis de entrada ou anteriores à operação ($name? \notin known$) e outra que contém referências a variáveis de saída ou variáveis finais ($birthday' = birthday \cup \{name? \mapsto date?\}$). A parte do predicado que se refere apenas a variáveis iniciais ou de entrada é denominada pré-condição, e a parte que se refere a variáveis finais ou de saída é denominada pós-condição.

O esquema *FindBirthday* não modifica o estado do sistema, fato indicado pela inclusão do nome do esquema *BirthdayBook* prefixado pela letra Ξ . Ele recebe um nome e devolve uma data.

<i>FindBirthday</i>
$\Xi \textit{BirthdayBook}$
$name? : NAME$
$date! : DATE$
$name? \in known$
$date! = birthday(name?)$

Nesse caso, a pré-condição exige que o nome recebido pela operação esteja no conjunto de nomes conhecidos e a pós-condição afirma que a data a ser devolvida pela operação é igual ao valor da função *birthday* aplicada ao nome recebido.

O esquema *Remind*, assim como o esquema *FindBirthday* não modifica o estado do sistema. Ele recebe uma data e devolve o conjunto de nomes que está associados a essa data pela função *birthday*. Esse comportamento é definido pela pós-condição contida no esquema.

<i>Remind</i>
$\Xi BirthdayBook$
$today? : DATE$
$cards! : \mathbb{P} NAME$
$cards! = \{n : known \mid birthday(n) = today?\}$

Note que essa operação não possui pré-condição². Exceto por ela, as outras operações possuem pré-condições, e, portanto, são aplicáveis apenas em determinadas situações. Seria interessante termos versões dessas operações que funcionasse corretamente em qualquer situação, isto é, versões que não tivessem pré-condições. Caso as pré-condições não sejam satisfeitas, o comportamento da operação não está definido. Porém, o arcabouço desenvolvido para dar suporte a essa metodologia lança uma exceção para indicar a falha de uma pré-condição.

Para isso, podem ser definidas algumas mensagens através da declaração de tipos livres que indicam se uma operação foi bem sucedida ou se ela não foi aplicada.

$REPORT ::= ok \mid already_known \mid not_known$

Vamos definir inicialmente alguns esquema auxiliares. A operação que simplesmente devolve a mensagem *ok* indicando o sucesso da operação.

<i>Success</i>
$result! : REPORT$
$result! = ok$

A operação *AlreadyKnown* indica que um nome já é conhecido devolvendo a mensagem *already_known*.

<i>AlreadyKnown</i>
$\Xi BirthdayBook$
$name? : NAME$
$result! : REPORT$
$name? \in known$
$result! = already_known$

A operação *NotKnown* indica que um nome não é conhecido e devolve a mensagem *not_known*.

²Podemos dizer que a pré-condição dessa operação é o predicado verdade (true)

<i>NotKnown</i>
$\exists \text{BirthdayBook}$
$\text{name?} : \text{NAME}$
$\text{result!} : \text{REPORT}$
$\text{name?} \notin \text{known}$
$\text{result!} = \text{not_known}$

Com essas operações auxiliares, podemos definir novas operações robustas através dos conectivos do cálculo de esquemas.

A versão robusta, isto é, sem pré-condição, da operação *AddBirthday* é definida pelo esquema horizontal apresentado a seguir.

$$R\text{AddBirthday} == (\text{AddBirthday} \wedge \text{Success}) \vee \text{AlreadyKnown}$$

Note que a pré-condição de *AddBirthday* é que o nome recebido por ela não pode ser conhecido, e a pré-condição de *AlreadyKnown* é que o nome recebido deve ser conhecido. Assim, a pré-condição da operação composta é que o nome é conhecido ou não, ou seja, ela é sempre verdade, tornando a operação robusta. A conjunção das operações *AddBirthday* e *Success* permite que a execução da operação *AddBirthday* devolva uma mensagem, assim como a operação *AlreadyKnown*.

A seguir, apresentamos a versão expandida dessa operação.

<i>XRAddBirthday</i>
$\text{known}, \text{known}' : \mathbb{P} \text{NAME}$
$\text{birthday}, \text{birthday}' : \text{NAME} \leftrightarrow \text{DATE}$
$\text{name?} : \text{NAME}$
$\text{date?} : \text{DATE}$
$\text{result!} : \text{REPORT}$
$(\text{name?} \notin \text{known} \wedge \text{birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \wedge \text{result!} = \text{ok}) \vee$
$(\text{name?} \in \text{known} \wedge \text{result!} = \text{already_known})$

A versão robusta da operação *FindBirthday* é dada pela expressão abaixo.

$$R\text{FindBirthday} == (\text{FindBirthday} \wedge \text{Success}) \vee \text{NotKnown}$$

Embora a operação *Remind* já seja robusta, podemos definir uma nova versão que devolve uma mensagem, de forma a torná-la consistente com as outras operações. Essa nova operação é apresentada abaixo.

$$R\text{Remind} == \text{Remind} \wedge \text{Success}$$

2.4 Uma Metodologia de Desenvolvimento Baseado em Modelos Formais

A partir de uma especificação como a apresentada no Exemplo 2.3.1, desejamos obter uma implementação correspondente. Existem diversas abordagens possíveis. Porém, em geral, elas focam na correção da implementação produzida. Devido a essa escolha, as implementações produzidas são, freqüentemente, feitas em alguma linguagem lógica ou funcional, a fim de facilitar o processo de transformação da especificação em código.

No entanto, essas linguagens não são amplamente usadas pela indústria de software, e, muitas vezes, não é viável ao usuário comum, utilizar sistemas que dependam dessas linguagens, devido, principalmente, à dificuldade de instalação e configuração dos sistemas que são suporte ao uso delas.

Além disso, é muito difícil produzir automaticamente um sistema completo a partir de uma especificação, visto que, em geral, as especificações são bastante abstratas e não se conhece uma forma robusta de se produzir programas que resolvam problemas a partir de especificações abstratas. Em geral, as ferramentas e metodologias exigem que a especificação seja refinada, de forma a torná-la suficientemente concreta para que possa ser transformada em código. Esse processo de refinamento, por sua vez, é bastante dispendioso, em especial quando desenvolvido formalmente.

Por esses motivos, nosso objetivo é definir uma metodologia que, a partir de uma especificação qualquer, produza uma implementação Java. Porém, não é possível desenvolver, de forma automática, uma implementação completa. Por isso, a implementação produzida deve ser complementada pelo usuário. Essa intervenção do usuário, embora necessária, possibilita a introdução de erro, que é, tradicionalmente, evitada através do refinamento formal da especificação. Para lidar com esse problema, a implementação gerada é equipada com mecanismos que permitem a verificação em tempo de execução dos invariantes de estados e das pré e pós-condições das operações. Dessa forma, um erro introduzido pelo usuário pode, eventualmente, ser percebido.

A percepção de que um erro ocorreu no sistema é uma propriedade importante, porém perde sua eficiência sem um mecanismo de rastreamento de erros. A metodologia aqui proposta procura facilitar o rastreamento através de um princípio bastante simples:

- Produzir uma implementação cuja estrutura reflita a estrutura da especificação.

A Figura 2.3 apresenta uma visão geral da metodologia. A partir dos requisitos do sistema, produz-se uma especificação em Z. Essa especificação é processada por uma ferramenta que implementa parte dessa metodologia (ver Capítulo 6.1), produzindo a estrutura da aplicação Java correspondente à especificação. Os predicados da especificação são traduzidos, manual ou automaticamente³, e o código funcional da aplicação é inserido. Dessa forma, obtemos um aplicação concreta. A execução dessa aplicação pode gerar eventos, tais como erros, que podem ser rastreados de volta à especificação, levando o desenvolvedor a corrigir a especificação. Nesse caso, o código

³A ferramenta implementada não dá suporte à tradução automática dos predicados.

Figura 2.3: Visão geral da metodologia proposta⁴

é gerado novamente, e qualquer mudança efetuada pelo usuário no código é perdida. Pretendemos melhorar a ferramenta desenvolvida de forma a guardar as alterações do usuário e facilitar a correção da especificação.

Na Seção 2.4.1 apresentamos os princípios gerais que regem a metodologias e, na Seção 2.4.2 os mecanismos providos pelo arcabouço implementado em Java que serve de base para a aplicação gerada. A Seção 2.4.3 aborda a forma como o código gerado pode ser mapeado em elementos da especificação e porque isso é importante. Na Seção 2.4.4 descrevemos como o arcabouço ajuda a impor a semântica da especificação.

2.4.1 Princípios

O objetivo da metodologia é a transformação rápida de uma especificação formal na notação Z em código executável. Para tanto, propomos um arcabouço que implementa a semântica de um subconjunto da notação Z, e serve de base para a criação da estrutura da aplicação que deve ser complementada pelo programador. O subconjunto tratado consiste nas especificações compostas apenas por esquemas e expressões do cálculo de esquemas, e cujas definições do arcabouço matemático utilizadas se restringem a conjuntos finitos, relações finitas e funções parciais finitas. O código complementar a ser inserido pelo programador é de dois tipos: código de verificação e código funcional. Os trechos de código de verificação implementam as verificações de invariantes, pré e pós-condições, enquanto os trechos de código funcional implementam as operações descritas. A ferramenta desenvolvida não gera código de verificação, nem código funcional. No entanto, pretendemos estendê-la para que ela gere ao menos parte do código de verificação necessário.

Vale notar que seria possível gerar parte do código de verificação de forma automática. Porém,

⁴Essa figura foi criada por Paulo Salem e uma versão dela está incluída em [35].

a proposta inicial deste trabalho não contempla essa geração de código.

O arcabouço impõe uma série de verificações que possibilitam a identificação de código incorreto introduzido pelo usuário. Além disso, eventuais erros ocorridos que não necessariamente se devem a código incorreto, podem ser mapeados em elementos da especificação, possibilitando a análise também da especificação, contribuindo para a evolução tanto da implementação, quanto da especificação.

O código gerado define um estado que deve ser atualizado sempre que houverem mudanças no ambiente ou operações forem executadas. Ele pode ser visto como um módulo a ser inserido em um sistema maior que utiliza seus serviços.

2.4.2 Arcabouço

O arcabouço proposto define um conjunto de classes abstratas associadas a elementos de Z. Ao traduzir uma especificação em uma aplicação concreta, analisamos cada elemento da especificação e criamos uma nova classe que estende a classe abstrata correspondente fornecida pelo arcabouço. Os elementos que a ferramenta associada à metodologia é capaz de tratar atualmente são: definições de esquema, distinguindo entre estados e operações, e alguns conectivos do cálculo de esquemas.

Devido à relação hierárquica entre as classes abstratas e concretas, a semântica embutida nas classes abstratas do arcabouço são passadas para as classes concretas da aplicação atribuindo-lhes comportamento adequado. A partir dessas classes concretas, o programador pode preenchê-las de forma a tornar a aplicação executável.

A abordagem de desenvolvimento do arcabouço visou balancear a consistência do código à especificação e a possibilidade de personalização do código complementar. Para isso, foram definidos três níveis de implementação:

- *Requisitos da especificação.* Contém a implementação dos elementos da especificação e só deles.
- *Requisitos de usuário.* Contém invariantes que a especificação não impõe, mas que o usuário deseja impor.
- *Código personalizado.* Código complementar que torna a aplicação executável.

Esse níveis direcionam a implementação do arcabouço, se caracterizando por métodos abstratos específicos a cada nível e que devem ser implementados.

As principais classes do arcabouço são as relacionadas a definições de esquema e ao cálculo de esquemas. A seguir, apresentamos a estrutura dessa classe e explicamos como as verificações de invariantes, pré e pós-condições são feitas, e como o sistema é executado. Apresentamos também a forma de inserir o código complementar.

Esquema e Cálculo de Esquemas

Os esquemas são usados para representar estados do sistema e operações sobre esses estados. O arcabouço define duas classes que representam esses elementos: *Schema* e *Operation*.

A classe *Schema* representa uma definição de esquema que descreve um estado, enquanto a classe *Operation* representa uma definição de esquema que, convencionalmente, descreve uma operação. Embora operações sejam esquemas e a distinção entre eles seja feita apenas por convenção, decidimos adotar a distinção devido à semântica especial associada a operações, em especial quando mapeadas em uma linguagem como Java. A classe *Operation* é uma subclasse de *Schema*.

Além de definir estados e operações através de definições de esquema, é possível construir elementos mais complexos a partir de elementos simples. Em Z, isso é feito através do cálculo de esquemas que provê uma série de conectivos, tais como conjunção, disjunção, negação, etc.

Esse conectivos são representados, no arcabouço, por subclasses da classe abstrata *SchemaCalc-Connective* que é uma subclasse da classe *Operation*. Dessa forma, eles podem ser aplicados tanto a definições de esquema que representa estados, quanto àquelas que representam operações.

Cada conectivo recebe uma ou duas subclasses da classe *Schema*, dependendo da sua aridade. Chamadas a conectivos são interpretadas em termos da semântica dos conectivos e das classes contidas. Por exemplo, sejam *A* e *B* duas classes que representam definições de esquema, para verificar o invariante da expressão $A \wedge B$, basta verificarmos que os invariantes de *A* e de *B* são ambos verdadeiros.

Invariantes

A classe *Schema* define métodos abstratos para a verificação de invariantes e uma subclasse concreta deve implementá-los. Se um invariante é violado, esses métodos lançam uma exceção. Tendo em vista a separação de níveis adotada, existem dois métodos de checagem de invariantes: *checkSpecificationInvariants()* e *checkUserInvariants()*, o primeiro é responsável por verificar os invariantes definidos na especificação e o segundo, os invariantes suplementares impostos pelo programador.

Para verificar os invariantes de uma expressão do cálculo de esquemas, basta chamar os métodos das classes associadas de acordo com a semântica dos conectivos. A implementação da semântica dos conectivos está embutida no arcabouço proposto.

Pré-condições e Pós-condições

Diferentemente das definições de esquema de estado, os invariantes das operações são decompostos em pré e pós condições. Por isso, a classe *Operation* exige a implementação dos seguintes métodos: *checkSpecificationPreconditions()*, *checkUserPreconditions()*, *checkSpecificationPostconditions()* e *checkUserPostconditions()*, responsáveis pela verificação, respectivamente, das pré-condições impostas pela especificação, das pré-condições impostas pelo usuários, das pós-condições impostas pela especificação e das pós-condições impostas pelo usuário.

Assim como no caso de definições de esquema de estado, essas verificações para expressões do cálculo de esquema que contêm operações são feitas através da semântica dos conectivos e das operações subjacentes.

Execução de código complementar

Para cada classe concreta que estende a classe *Schema*, o desenvolvedor pode prover código que será executado sempre que o estado do sistemas for atualizado e o invariante continuar valendo. Esse código deve ser inserido no método *execute()*.

No caso das operações, um método *execute()* diferente deve ser implementado, *execute(Map schemas, Map input)*. Esse método recebe como parâmetros os estados a serem modificados pela operação e as entradas da operação, se existirem, e devolve as saídas, se existirem. Antes da execução da operação é necessário a verificação das pré-condições e após, a verificação das pós-condições. Em ambos os casos, é necessário verificar que os invariantes dos estados relevantes se mantêm verdadeiros.

Atualização do estado do sistema

A verificação freqüente de invariantes, pré e pós-condições deve ser feita pelo arcabouço, não pelo programador, visto que ela é necessária apenas em pontos bem definidos. Para que isso seja possível, a classe *Schema* define um método *update()*, que implementa o seguinte pseudo algoritmo:

1. Chama *checkSpecificationInvariants()* and *checkUserInvariants()* para cada *Schema* importado e para o *Schema* atual;
2. Se nenhuma exceção ocorre, chama *execute()* para cada *Schema* importado e para o *Schema* atual;
3. Repete o passo 1;
4. Se nenhuma exceção ocorre, a atualização foi bem sucedida. Caso contrário, o sistema está em um estado inconsistente e o processo de atualização é interrompido.

Dessa forma, para atualizar o estado do sistema, basta chamar o método *execute()* para cada *Schema* contido no sistema. Se a atualização de algum deles provar uma exceção, o sistema estará inconsistente.

O arcabouço foi estruturado tendo em vista o uso do código gerado como um módulo separado, cujo estado é mudado em passos discretos. Portanto, quando desejamos mudar o estado do sistema, devemos chamar o método relevante do módulo, que por sua vez chama *update()* para cada *Schema* concreto.

As operações exigem outras considerações. Elas são equipadas com um método *executeOperation(Map schemas, Map input)*, que checa as pré-condições antes de chamar *execute(Map schemas, Map input)* e as pós-condições depois dessa chamada. Seu funcionamento é análogo ao do método *update()*.

2.4.3 Relacionamento entre Código e Especificação

O arcabouço foi desenvolvido objetivando preservar a estrutura encontrada na especificação, a fim de mapear eventos da implementação em elementos da especificação. Por exemplo, se a

especificação contém uma definição de esquema que é composta por várias outras definições de esquema, nossa tradução preserva essas definições, criando uma classe para cada um. Se uma delas violar um invariante, nós podemos identificar exatamente qual definição de esquema está relacionada ao problema. Dessa forma, é possível relacionar a implementação à especificação.

As principais utilidades desse relacionamento são:

- se algo não funciona adequadamente, pode-se verificar na especificação as definições associadas ao mau funcionamento. Pode ser que o problema esteja na especificação, e através dessa identificação, poder-se-ia corrigi-la;
- a execução do sistema pode ser relacionada a uma teoria subjacente, de forma que uma interpretação possa ser dada a ela.

Esse último uso pode ser bastante importante em simulação de agentes. Se definirmos um modelo formal de comportamento de agentes, como por exemplo em [45], e o implementamos para simulá-lo, é necessário relacionar os resultados da simulação ao modelo formal. Caso contrário, pode ser difícil de interpretar tais resultados, o que os torna de pouca utilidade. Por exemplo, pode ser interessante identificar exatamente que definições de esquema foram usadas durante uma simulação, a fim de identificar as suposições relevantes. O relacionamento entre a implementação e a especificação é, portanto, de suma importância e o fato de ele ser implementado como parte do arcabouço livra o usuário do trabalho de implementá-lo.

2.4.4 Correção

Ao complementar a estrutura da aplicação gerada, o programador pode inserir código que contém erros, se ele não tiver garantias de correção do código inserido. Essa possibilidade foi mantida, pois tem a vantagem de agilizar o processo de desenvolvimento. Porém, para limitar os efeitos prejudiciais de um suposto código errado, impomos a verificação, pelo arcabouço, dos invariantes, pré e pós-condições.

Como os invariantes de Z devem ser sempre verdade, e em Java não é possível que eles sempre o sejam, é necessário que as verificações sejam feitas em pontos estratégicos da execução. Por isso, é preciso um controle rígido sobre a execução do sistemas. Esse controle é obtido através da suposição de que o sistema só muda quando é explicitamente alterado, e essa suposição é imposta pelo arcabouço.

A verificação freqüente de invariantes, pré e pós-condições tem o efeito negativo de prejudicar a eficiência do sistema. Uma opção para contornar esse problema, é utilizar os mecanismos de verificação apenas em uma fase de teste, desligando-os quando o sistema é efetivamente utilizado.

Vale notar que essa metodologia não se aplica a sistemas concorrentes, visto que a notação Z não visa a especificação desse tipo de sistema. Embora haja a possibilidade de especificá-los em Z [25], existem outras linguagens mais adequadas, tais como CSP [22], CCS [34] e *Circus* [37,61].

As regras de transformação que norteiam o processo de geração da aplicação serão apresentadas nos capítulos que seguem. Note que especificações Z podem ser escritas em duas marcações principais: $\text{\LaTeX}$ e ASCII. Segundo o padrão internacional [24], a marcação $\text{\LaTeX}$ é adequada para a

apresentação da especificação em forma matemática, enquanto a marcação ASCII visa a permitir a exibição da especificação em dispositivos de baixa resolução. Em sua maior parte, as especificações são escritas em $\text{\LaTeX}$, e por esse motivo apresentamos, junto às regras de tradução contidas no Apêndice D, a expressão $\text{\LaTeX}$ correspondente aos elementos Z mencionados. Para uma descrição completa das marcações $\text{\LaTeX}$ e ASCII, vide o padrão internacional.

O Capítulo 3 aborda as regras referentes à transformação de conjuntos e funções parciais, o Capítulo 4 apresenta as regras de transformação de estados e o Capítulo 5 descreve as regras de transformação de operações.

,

Capítulo 3

Arcabouço Matemático

Uma especificação Z pode ser constituída de uma seqüência de seções ou de uma única seção anônima, e uma seção pode herdar outras seções. Esse mecanismo de herança permite a construção estruturada de especificações, reutilizando seções existentes. A figura 3.1 apresenta um exemplo de especificação composta por diversas seções relacionadas entre si e relacionadas também a seções externas através das setas tracejadas.

Figura 3.1: Exemplo de uma especificação composta de diversas seções

Todo elemento usado em uma seção deve pertencer ao núcleo da linguagem Z , ser definido na própria seção ou ser herdado de alguma outra seção. Porém, o núcleo da linguagem Z não define alguns elementos muito usados em especificações, tais como relações e números. Se esses elementos não estão no núcleo da linguagem, eles devem ser definidos na seção que os usa ou herdados de

outra seção. Como alguns elementos são de uso comum, seria repetitivo defini-los em cada seção que os usa, por isso eles foram definidos em uma seção básica denominada arcabouço matemático. Embora essa seção defina elementos bastante básicos, ela assume a existência de elementos não pertencentes ao núcleo da linguagem, tal como o conjunto dos números naturais. Para contornar essa situação é definido uma seção básica denominada prelúdio que é herdada implicitamente por toda seção. Ela introduz um tipo dado denominado “*arithmos*” ($\mathbb{A}$), define o conjunto dos números naturais, os números 0 e 1, e a soma de naturais.

Enquanto o prelúdio é herdado implicitamente e, portanto, de forma obrigatória por toda seção, o arcabouço matemático é opcional para todas elas, exceto para as anônimas, que o herdam implicitamente. Isso ocorre pois as seções anônimas não podem herdar explicitamente outras seções.

O arcabouço matemático se divide em cinco seções (sub arcabouços), cada uma delas relacionada a uma classe específica de elementos (conjuntos, relações, funções, números e seqüências). Os elementos de um sub arcabouço são definidos em termos de elementos do núcleo da linguagem, do prelúdio ou de outro sub arcabouço. A hierarquia de sub arcabouços determinada pela estrutura de dependência forma uma árvore, como pode-se observar na figura 3.2.

Figura 3.2: Hierarquia de sub arcabouços do arcabouço matemática

Note que a modularização de especificações em seções é apresentada no padrão internacional [24]. Porém, não encontramos exemplos de especificações que utilizem desse artifício. Esse tipo de recurso pode ser muito útil, por exemplo, para estender uma especificação existente.

Nesse capítulo apresentamos a estratégia de tradução de trechos de especificação que contém elementos do arcabouço matemático. Além disso, alguns elementos do prelúdio também devem ser considerados e serão tratados junto a um sub arcabouço adequado, por exemplo, o conjunto dos números naturais pode ser tratado junto ao sub arcabouço de números.

A estratégia de tradução se concretiza na construção de um mapeamento entre elementos (tipos e operadores) definidos no arcabouço matemático e elementos definidos em Java. Para tanto, identificamos, a partir da descrição do arcabouço matemático contida no padrão da notação Z [24], os operadores relevantes para cada tipo, caracterizando tipos abstratos de dados que, em Java, podem ser descritos na forma de interfaces.

No entanto, apenas as interfaces não são suficientes para concretizar a tradução. São necessários tipos concretos de dados que efetivamente as implementam. Portanto, apresentamos um conjunto de classes que implementam essas interfaces. Embora não provemos a correção das implementações propostas com relação à definição formal, argumentamos, com base na definição formal, a conformidade das implementação a elas.

Nas seções seguintes apresentamos a construção do mapeamento para conjuntos, relações e funções. Vale notar que as implementações sugeridas são finitas. Portanto, elementos não finitos não poderão ser traduzidos para essas classes. Cada seção se compõe de três subseções: definição do tipo, definição da interface e implementação. Os elementos do arcabouço matemático tratado neste trabalho são indicados na Figura 3.3 por caixa com fundo de cor sólida, os elementos não implementados são identificados por linhas horizontais e os arcabouços parcialmente implementados são marcados por linhas inclinadas.

O mapeamento dos operadores dos tipos tratados, devido à definição da interface, se dá de forma direta, pois cada operador possui um método equivalente definido na interface do tipo correspondente. Assim, traduz-se um operador pela chamada do método correspondente com os parâmetros adequados.

3.1 Conjuntos

Conjuntos são tipos primitivos da notação Z e o núcleo da linguagem define formas de construir conjuntos e a relação de pertinência. Porém, outras relações e os operadores de conjuntos são definidos apenas no arcabouço matemático. Como estamos considerando o arcabouço matemático parte integrante da notação Z e para uniformizar a descrição desse tipo, vamos considerar aqui também os elementos definidos no núcleo da linguagem.

Um conjunto é parametrizado por um tipo, o tipo dos seus elementos. No que segue, esse tipo será representado por uma letra maiúscula ($X, Y, \dots$) e será identificado como tipo, exceto onde for inferível pelo contexto.

3.1.1 Definição do Tipo de Dados

Em Z, conjuntos e tipos são a mesma coisa, e os conjuntos mais básicos encontrados numa especificação são os tipos dados. A partir desses conjuntos básicos é possível construir novos conjuntos. Algumas formas são:

Figura 3.3: Elementos do arcabouço matemática tratados

Conjunto das partes

Consiste no conjunto de todos os subconjuntos de um dado conjunto. Por exemplo, $\mathbb{P}\mathbb{N}$ é o conjunto de todos os subconjuntos dos naturais.

Compreensão¹

Consiste no conjunto de valores de uma expressão para todos os elementos de um tipo. Por exemplo, $\{x, y : \mathbb{N} \mid x > y \bullet (x, y)\}$ é o conjunto de todos os pares ordenados de naturais cujo primeiro elemento é maior que o segundo.

Extensão

Consiste no conjunto que contém os valores das expressões contidas na extensão. Por exemplo, $\{1, 1 + 1, 1 + 1 + 1\}$ é o conjunto composto por 1, 2 e 3.

Nesses casos o tipo de conjunto obtido não é necessariamente o mesmo que um dos conjuntos originais. Existem, no entanto, operadores de conjuntos que produzem novos conjuntos do mesmo tipo do original:

- União ($\cup$)

¹O dicionário Houaiss define o termo “compreensão”, em lógica clássica, como “conjunto formado pelas características ou qualidades que estão contidas na definição de um conceito (como p.ex., no caso do conceito homem, cuja compreensão engloba sua racionalidade, animalidade, bipedismo, mortalidade etc.) [23]”.

- Intersecção ($\cap$)
- Diferença ($\setminus$)
- Diferença simétrica ($\ominus$)
- União generalizada ($\cup$)
- Intersecção generalizada ($\bigcap$)

Esses operadores são definidos no sub arcabouço de conjuntos, junto com as relações de não pertinência ($\notin$), continência ($\subseteq$) e continência própria ($\subset$), o conjunto vazio ($\emptyset$) e os subconjuntos não vazios ($\mathbb{P}_1$). São definidos também os subconjuntos finitos ($\mathbb{F}$) e os subconjuntos finitos não vazios ($\mathbb{F}_1$). Vale lembrar que a relação de pertinência ($\in$) foi deslocada para essa seção para uniformizar o tratamento dos tipos de dados.

3.1.2 Definição da Interface

Note que os operadores relevantes para o tipo conjunto de elementos do tipo X são apenas aqueles que preservam o tipo. Outros operadores precisam ser tratados em separado. Portanto, a interface Java que representa o tipo de dado abstrato definido acima deve conter métodos para cada relação e operador que preserva tipos definidos.

Na definição da interface a finitude dos conjuntos é irrelevante, visto que os operadores e relações se mantêm inalterados, igualmente irrelevante é a propriedade de não vacuidade, pois esta também não altera os operadores e relações.

As interfaces adicionais que essa interface estende são necessárias para a manipulação conveniente de conjuntos e foram incluídas para aumentar a flexibilidade das classes e facilitar a implementação de algoritmos que as utilizem.

3.1.3 Implementação

Na implementação do tipo abstrato de dado, além dos operadores e relações definidos é necessário levar em conta os elementos peculiares (p. ex. conjunto vazio). Além disso, ao contrário do que ocorre no caso das interfaces, as restrições de finitude e vacuidade são relevantes para a implementação, visto que podem alterar o comportamento de um operador. Por exemplo, a diferença de um conjunto por ele mesmo gera um conjunto vazio que violaria uma restrição de não vacuidade. O conjunto vazio, como elemento particular do tipo conjunto, será tratado, quando necessário, em termos da classe que representa conjuntos.

A implementação da interface *Set* (Figura 3.5) aqui proposta se baseia na classe `HashSet` da Java API. As relações de pertinência, compreensão e compreensão própria são definidas em termos das operações `contains`, `containsAll` e `equals` da classe `HashSet` (Figura 3.6). As operações de união e intersecção (Figura 3.7) são definidas em termos das operações `add` e `remove` da classe `HashSet` e as outras operações (Figuras 3.8 e 3.9) são definidas em termos dessas operações ou das operações da própria classe. A implementação completa da classe `BasicSet` pode ser encontrada no Apêndice E.

```

public interface ZSet<X> extends Iterable<X>, Collection<X>, Set<X>, Cloneable {
    /* this methods checks whether this set is equal to set S */
    public Boolean equals(ZSet<X> S);

    /* this method checks whether this set contains element e */
    public Boolean contains(X e);

    /* this method checks whether this set contains set S */
    public Boolean contains(ZSet<X> S);

    /* this method checks whether this set contains set S properly */
    public Boolean contains_properly(ZSet<X> S);

    /* this method turns this set into the union of S1 and S2 */
    public ZSet<X> union(ZSet<X> S1, ZSet<X> S2);

    /* this method turns this set into the intersection of S1 and S2 */
    public ZSet<X> intersection(ZSet<X> S1, ZSet<X> S2);

    /* this method turns this set into the difference of S1 and S2 */
    public ZSet<X> difference(ZSet<X> S1, ZSet<X> S2);

    /* this method turns this set into the symmetric difference of S1 and S2 */
    public ZSet<X> symmetric_difference(ZSet<X> S1, ZSet<X> S2);

    /* this method turns this set into the generalized union of the sets in S */
    public ZSet<X> generalized_union(ZSet<ZSet<X>> S);

    /*
     * this method turns this set into the generalized intersection of the sets
     * in S
     */
    public ZSet<X> generalized_intersection(ZSet<ZSet<X>> S);
}

```

Figura 3.4: Interface Java para classes que implementam conjuntos.

Note que a implementação de operadores binários foi feita utilizando um métodos com dois argumentos. Por exemplo, para unir dois conjuntos, devemos tomar um conjunto vazio e chamar o métodos union passando como parâmetros os dois conjuntos que devem ser unidos. Dessa forma, evitamos alterar um conjunto que possivelmente será usado posteriormente.

```

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class BasicSet<X> extends HashSet<X> implements ZSet<X> {
    ...
}

```

Figura 3.5: Classe BasicSet que implementa a interface ZSet

```

public Boolean contains(X e) {
    return super.contains(e);
}
public Boolean contains(ZSet<X> S) {
    return super.containsAll(S);
}
public Boolean contains_properly(ZSet<X> S) {
    return !(contains(S) || equals(S));
}

```

Figura 3.6: Implementação das relações de conjuntos

```

public ZSet<X> union(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    addAll(S2);
    return this;
}
public ZSet<X> intersection(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    for (Iterator<X> it = iterator(); it.hasNext();) {
        if (!S2.contains(it.next()))
            it.remove();
    }
    return this;
}

```

Figura 3.7: Implementação das operações de união e intersecção

```

public ZSet<X> generalized_intersection(ZSet<ZSet<X>> S) {
    generalized_union(S);
    for (Iterator<X> it = iterator(); it.hasNext();) {
        X x = it.next();
        for (ZSet<X> s: S) {
            if (!s.contains(x))
                it.remove();
        }
    }
    return this;
}

public ZSet<X> generalized_union(ZSet<ZSet<X>> S) {
    clear();
    for (ZSet<X> s: S) {
        addAll(s);
    }
    return this;
}

```

Figura 3.8: Implementação das versões generalizadas de união e intersecção

```

public ZSet<X> difference(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    for (X i : S2) {
        this.remove(i);
    }
    return this;
}
public ZSet<X> symmetric_difference(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    for (X i : S2) {
        if (contains(i))
            remove(i);
        else
            add(i);
    }
    return this;
}

```

Figura 3.9: Implementação das outras operações

3.2 Relações

Relações são associações entre elementos de dois conjuntos. Por exemplo, a relação de pertinência relaciona elementos de um conjunto X a elementos do conjunto $\mathbb{P}X$.

3.2.1 Definição do Tipo de Dados

As relações entre elementos do tipo X e do tipo Y são definidas em termos de conjuntos de pares ordenados. Os únicos operadores sobre pares ordenados são *first* e *second*, e os operadores definidos especificamente sobre relações são:

- Domínio (dom)
- Imagem (ran)
- Composição (relacional e funcional) ($\circ$, $\circ$)
- Restrição de domínio ($\triangleleft$)
- Restrição de imagem ($\triangleright$)
- Subtração de domínio ($\triangleleft$)
- Subtração de imagem ($\triangleright$)
- Inversão relacional ($\sim$)
- Imagem relacional ($\|$)
- *Overriding* ($\oplus$)

- Fecho transitivo (+)
- Fecho transitivo reflexivo. (*)

Apenas uma relação particular é definida, a relação identidade (*id*).

3.2.2 Definição da Interface

Como relações são definidas em termos de conjuntos, os operadores de conjuntos também se aplicam a relações. Dessa forma, a interface Java que deve ser implementada por classes que representam uma relação (de Z) e deve conter métodos para cada operador mencionado acima, além dos operadores de conjuntos, adaptados para a manipulação de relações. Os métodos referentes a conjuntos deveriam ser herdados da interface `ZSet`. Porém, isso implicaria em manter a assinatura dos métodos e representar as relações explicitamente como conjuntos. Por esse motivo, decidimos reproduzir explicitamente os métodos referentes a conjuntos na interface de relações.

A seguir apresentamos a interface `ZRelation` que serve de base para a implementação de relações, funções (totais, parciais, finitas, injetivas, sobrejetivas etc) e seqüências. Nós não definimos interfaces particulares para funções pois esses tipos de dados não apresentam operadores novos, além disso, uma operação como união de funções pode resultar numa relação que não é função, de forma que é necessário manter o tipos do resultado do métodos correspondente como relação e não função.

```
package jzed.mtoolkit;

public interface ZRelation<X, Y> extends Cloneable {
    /*
     * methods that must be implemented because of the set aspect of partial
     * functions
     */
    /* this methods checks whether this relation is equal to relation R */
    public Boolean equals(ZRelation<X, Y> R);

    /* this method checks whether this relation contains the pair x, y */
    public Boolean contains(X x, Y y);

    /* this method checks whether this relation contains relation R */
    public Boolean contains(ZRelation<X, Y> R);

    /* this method checks whether this relation contains relation R properly */
    public Boolean contains_properly(ZRelation<X, Y> R);

    /* this method turns this relation into the union of R1 and R2 */
    public ZRelation<X, Y> union(ZRelation<X, Y> R1, ZRelation<X, Y> R2);

    /* this method turns this relation into the intersection of R1 and R2 */
    public ZRelation<X, Y> intersection(ZRelation<X, Y> R1, ZRelation<X, Y> R2);
```

```

/* this method turns this relation into the difference of R1 and R2 */
public ZRelation<X, Y> difference(ZRelation<X, Y> R1, ZRelation<X, Y> R2);

/* this method turns this relation into the symmetric difference of R1 and R2 */
public ZRelation<X, Y> symmetric_difference(ZRelation<X, Y> R1,
                                           ZRelation<X, Y> R2);

/* this method turns this relation into the generalized union of the relations in R */
public ZRelation<X, Y> generalized_union(ZSet<ZRelation<X, Y>> R);

/*
 * this method turns this relation into the generalized intersection of the relations in
public ZRelation<X, Y> generalized_intersection(ZSet<ZRelation<X, Y>> R);

/* methods particular to relations */
/* this method returns the domain of this relation */
public ZSet<X> domain();

/* this method returns the range of this relation */
public ZSet<Y> range();

/*
 * this method turns this relation into the relational composition of
 * relations R1 e R2
*/
public <T> ZRelation<X, Y> relational_composition(ZRelation<X, T> R1,
                                                  ZRelation<T, Y> R2);

/*
 * this method turns this relation into the functional composition of
 * relations R1 e R2
*/
public <T> ZRelation<X, Y> functional_composition(ZRelation<X, T> R1,
                                                  ZRelation<T, Y> R2);

/*
 * this method turns this relation into relation R with domain restricted to
 * set S
*/
public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, ZSet<X> S);

/*
 * this method turns this relation into relation R with range restricted to
 * set S
*/
public ZRelation<X, Y> range_restriction(ZRelation<X, Y> R, ZSet<Y> S);

/*

```

```

    * this method turns this relation into relation R with domain subtracted by
    * set S
    */
public ZRelation<X, Y> domain_subtraction(ZRelation<X, Y> R, ZSet<X> S);

/*
    * this method turns this relation into relation R with range subtracted by
    * set S
    */
public ZRelation<X, Y> range_subtraction(ZRelation<X, Y> R, ZSet<Y> S);

/* this method turns this relation into the inverse of relation R */
public ZRelation<Y, X> relational_inversion();

/*
    * this method returns the set equivalent to the relational image of this
    * relation and set S
    */
public ZSet<Y> relational_image(ZSet<X> S);

/* this method turns this relation into relation R1 overridden by R2 */
public ZRelation<X, Y> overriding(ZRelation<X, Y> R1, ZRelation<X, Y> R2);

/* this method turns this relation into the transitive closure of relation R */
public ZRelation<X, X> transitive_closure(ZRelation<X, X> R);

/*
    * this method turns this relation into the transitive reflexive closure of
    * relation R
    */
public ZRelation<X, X> reflexive_transitive_closure(ZRelation<X, X> R);

/*
    * this method implements application of this relation to value X and
    * returns the sole range value i such that the pair (x,i) is in the set of
    * pairs that is R. If there is no value or more than one value associated
    * with x, then the application has a value but what it is is not specified
    * (from ISO/IEC 13568)
    */
public Y apply(X x);
/* these methods are auxiliary methods */
/* this method return the set of elements associated with element y */
public ZSet<X> yapply(Y y);
/* this method return the set of elements associated with element x */
public ZSet<Y> xapply(X x);

public void clear();

```

```

    /* this method puts a pair in a relation */
    public Y put (X x, Y y);
    /* this method removes a pair from a relation */
    public Y remove(X x, Y y);

    public Object clone();
}

```

3.2.3 Implementação

A implementação de relações foi desenvolvida de forma a priorizar o desempenho de operações de consulta, por esse motivo, utilizamos dois mapeamentos. O primeiro leva elementos do domínio em conjuntos de elementos da imagem, e o segundo leva elementos da imagem em conjuntos de elementos do domínio. A Figura 3.10 apresenta uma classe básica que implementa a interface `ZRelation`, denominada `BasicSet`. A implementação completa da classe `BasicSet` pode ser encontrada no Apêndice F.

```

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

public class BasicRelation<X, Y> implements ZRelation<X, Y> {
    public HashMap<X, ZSet<Y>> mapx;
    public HashMap<Y, ZSet<X>> mapy;
    ...
}

```

Figura 3.10: Classe `BasicRelation` que implementa a interface `ZRelation`

A seguir apresentamos alguns dos métodos implementados. O resultado da aplicação de uma relação a um valor do domínio e a diversos elementos do domínio (imagem relacional) é dado, respectivamente, pelos métodos `apply` e `relational_image`.

```

@Override
public Y apply(X x) {
    ZSet<Y> sety = xapply(x);
    Iterator<Y> it = sety.iterator();
    if (it.hasNext()) return it.next();
    else return null;
}

@Override
public ZSet<Y> relational_image(ZSet<X> S) {
    ZSet<Y> ring = new BasicSet<Y>();
    for (X x: S) {
        ring.addAll(xapply(x));
    }
    return ring;
}

```



```
}
```

Os operadores de domínio e imagem são dados pelos métodos `domain` e `range` apresentados a seguir.

```
@Override
public ZSet<X> domain() {
    return new BasicSet<X>(mapx.keySet());
}

@Override
public ZSet<Y> range() {
    return new BasicSet<Y>(mapy.keySet());
}
```

Os operadores de restrição e subtração de domínio e imagem são implementados nos métodos `domain_restriction`, `domain_subtraction`, `range_restriction` e `range_subtraction`

```
@Override
public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x: R.domain()) {
        if (S.contains(x)) {
            for (Y y: R.xapply(x)) {
                this.put(x,y);
            }
        }
    }
    return this;
}

@Override
public ZRelation<X, Y> domain_subtraction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x: R.domain()) {
        if (!S.contains(x)) {
            for (Y y: R.xapply(x)) {
                this.put(x,y);
            }
        }
    }
    return this;
}

@Override
public ZRelation<X, Y> range_restriction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (Y y: R.range()) {
        if (S.contains(y)) {
            for (X x: R.yapply(y)) {

```

```

        this.put(x,y);
    }
}

return this;
}

@Override
public ZRelation<X, Y> range_subtraction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (Y y: R.range()) {
        if (!S.contains(y)) {
            for (X x: R.yapply(y)) {
                this.put(x,y);
            }
        }
    }
    return this;
}

```

Os operadores de sobrescrita , composição e inversão são dados, a seguir, pelos métodos `overriding`, `functional_composition` e `relational_inversion`.

```

@Override
public <T> ZRelation<X, Y> functional_composition(ZRelation<X, T> R1,
    ZRelation<T, Y> R2) {
    clear();
    for (X x: R1.domain()) {
        for (T t: R1.xapply(x)) {
            if (R2.domain().contains(t)) {
                for (Y y: R2.xapply(t)) {
                    this.put(x,y);
                }
            }
        }
    }
    return this;
}

@Override
public ZRelation<X, Y> overriding(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R2);
    ZSet<X> dom2 = R2.domain();
    for (X x: R1.domain()) {
        if (!dom2.contains(x))
            for (Y y: R1.xapply(x)) {
                put(x,y);
            }
    }
}

```

```

    }
    return this;
}

@Override
public ZRelation<Y, X> relational_inversion() {
    ZRelation<Y,X> r = new BasicRelation<Y,X>();
    for (X x: domain()) {
        for (Y y: xapply(x)) {
            r.put(y,x);
        }
    }
    return r;
}

```

O fecho transitivo é calculado através do algoritmo de Warshall [46] no método `transitive closure` apresentado abaixo. O fecho transitivo reflexivo não foi implementado visto que essa implementação corresponde a uma relação finita e o fecho transitivo reflexivo de uma relação pode ser infinito. Por exemplo, tomemos as relação de naturais em naturais formada pelos pares $(1, 10), (2, 20), (3, 30)$, seu fecho transitivo reflexivo deve incluir pelo menos os pares $\{n : \mathbb{N} \bullet (n, n)\}$, logo o fecho transitivo reflexivo de uma relação finita, pode ser infinito.

```

@Override
public ZRelation<X, X> transitive_closure(ZRelation<X, X> R) {
    ArrayList<X> elements = new ArrayList<X>();
    elements.addAll((new BasicSet<X>()).union(R.domain(), R.range()));
    int n = elements.size();
    int[][] matrix = new int[n][n];
    for (X x: R.domain()) {
        for (X y: R.xapply(x)) {
            matrix[elements.indexOf(x)][elements.indexOf(y)] = 1;
        }
    }

    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            if (matrix[i][j] == 1) {
                for (int k = 0; k < n; k++) {
                    if (matrix[j][k] == 1)
                        matrix[i][k] = 1;
                }
            }
        }
    }

    ZRelation<X,X> r = new BasicRelation<X,X>();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {

```

```

        if (matrix[i][j] == 1) {
            r.put(elements.get(i), elements.get(j));
        }
    }
    return r;
}

```

3.3 Funções Parciais

As funções parciais são relações que associam um elemento de um conjunto a no máximo um elemento de outro conjunto. Por exemplo, uma relação que associa o nome de uma pessoa à sua data de aniversário é uma função parcial. A parcialidade da função indica que nem todo elemento do conjunto de origem tem um elemento do conjunto de chegada associado.

3.3.1 Definição do Tipo

Sejam X e Y conjuntos quaisquer, uma função parcial f de X para Y é uma relação entre X e Y , tal que para cada elemento x de X , existe no máximo um elemento y de Y relacionado a x . Formalmente, o conjunto das funções parciais de X para Y ($X \rightarrow Y$) é dado por:

$$X \rightarrow Y == \{f : X \leftrightarrow Y \mid \forall p, q : f \mid p.1 = q.1 \bullet p.2 = q.2\}$$

Utilizando a função de domínio de uma relação, podemos reescrever a definição acima da seguinte forma:

$$X \rightarrow Y == \{f : X \leftrightarrow Y \mid \forall x, y : \text{dom } f \mid x = y \bullet f(x) = f(y)\}$$

Observemos que a relação vazia ($f_\emptyset$) entre X e Y é, de fato, uma função parcial.

Se $f_\emptyset \in X \rightarrow Y$, então $\forall x, y : \text{dom } f_\emptyset \mid x = y \bullet f_\emptyset(x) = f_\emptyset(y)$, e como $\text{dom } f_\emptyset = \emptyset$, temos que $\forall x, y : \emptyset \mid x = y \bullet f_\emptyset(x) = f_\emptyset(y)$ é trivialmente verdade.

Os operadores que manipulam funções parciais são aqueles definidos para conjuntos e relações.

- Domínio (dom)
- Imagem (ran)
- Composição (relacional e funcional) ($\circ, \circ$)
- Restrição de domínio ($\triangleleft$)
- Restrição de imagem ($\triangleright$)
- Subtração de domínio ($\triangleleft$)
- Subtração de imagem ($\triangleright$)
- Inversão relacional ($\sim$)

- Imagem relacional ($\llbracket \rrbracket$)
- *Overriding* ($\oplus$)
- Fecho transitivo ($^+$)
- Fecho transitivo reflexivo. (*)

Podemos observar que todos esses operadores podem ser definidos em termos de duas operações básicas, inserção e eliminação. Inserção, como o próprio nome diz insere um elemento em um conjunto, e a eliminação remove um elemento de um conjunto. Abaixo definimos as operações de inserção e eliminação (respectivamente, *insert* e *delete*).

$$\frac{[X]}{\text{insert} : \mathbb{P} X \times X \rightarrow \mathbb{P} X} \quad \frac{}{\forall A : \mathbb{P} X; b : X; \bullet \text{insert}(A, b) = \{x : X \mid x \in A \vee x = b\}}$$

$$\frac{[X]}{\text{delete} : \mathbb{P} X \times X \rightarrow \mathbb{P} X} \quad \frac{}{\forall A : \mathbb{P} X; b : X; \bullet \text{delete}(A, b) = \{x : X \mid x \in A \wedge \neg (x = b)\}}$$

Como uma função parcial é uma relação, ela só pode ser modificada por operações que podem ser definidas em termos de inserções e eliminações de elementos. Vamos verificar que a eliminação de um elemento de uma função parcial produz uma função parcial e que a inserção só produz uma função parcial sob certas condições.

Teorema 3.1. *Seja f uma função parcial de X em Y e a um elemento do domínio de f , então $\text{delete}(f, (a, f(a)))$ também é uma função parcial.*

Prova do teorema 3.1. Seja $g = \text{delete}(f, (a, f(a)))$, vamos demonstrar que $g \in X \leftrightarrow Y$.

$$\begin{aligned} g \in X \leftrightarrow Y &\Leftrightarrow \\ g \in \{f : X \leftrightarrow Y \mid \forall x, y : \text{dom } f \mid x = y \bullet f(x) = f(y)\} &\Leftrightarrow \\ \forall x, y : \text{dom } g \mid x = y \bullet g(x) = g(y) & \end{aligned}$$

Basta então demonstrar que $\forall x, y : \text{dom } g \mid x = y \bullet g(x) = g(y)$.

Da definição de domínio e de *delete* temos que

$$\begin{aligned} \text{dom } g &= \text{dom } \text{delete}(f, (a, f(a))) \\ &= \{p : \text{delete}(f, (a, f(a))) \bullet p.2\} \end{aligned}$$

$$\begin{aligned}
&= \{p : \{q : X \times Y \mid q \in f \wedge q \neq (a, f(a))\} \bullet p.2\} \\
&= \{p : X \times Y \mid p \in f \wedge p \neq (a, f(a)) \bullet p.2\} \\
&= \{c : X \mid c \in \text{dom } f \wedge c \neq a\}
\end{aligned}$$

Portanto, $\text{dom } g \subset \text{dom } f$. Como f é função parcial, sabemos que $\forall x, y : \text{dom } f \mid x = y \bullet f(x) = f(y)$ e dado que $\text{dom } g \subset \text{dom } f$, vale que $\forall x, y : \text{dom } g \mid x = y \bullet f(x) = f(y)$, e como f coincide com g em todos os elementos do domínio de f exceto a , vale que $\forall x, y : \text{dom } g \mid x = y \bullet g(x) = g(y)$. ■

Teorema 3.2. *Seja f uma função parcial de X em Y , a um elemento de X e b um elemento de Y , então:*

1. *Se a pertence ao domínio de f e $f(a) \neq b$, $\text{insert}(f, (a, b))$ não é função parcial;*
2. *Caso contrário, $\text{insert}(f, (a, b))$ é função parcial.*

Prova do teorema 3.2. Seja $g = \text{insert}(f, (a, b))$,

1. suponha que a pertence ao domínio de f e que $f(a) \neq b$, então $(a, f(a)) \in g$ e $(a, b) \in g$, i.e. $\exists x, y : X \mid x = y \bullet g(x) \neq g(y)$, portanto g não é função parcial.
2. O segundo item pode ainda ser dividido em dois casos
 - (a) suponha que a pertence ao domínio de f e $f(a) = b$, nesse caso o par (a, b) já pertencia a f , e $f = g$, logo g é função parcial.
 - (b) suponha que a não pertence ao domínio de f , nesse caso devemos provar que $g \in X \leftrightarrow Y$.

$$\begin{aligned}
g \in X \leftrightarrow Y &\Leftrightarrow \\
g \in \{f : X \leftrightarrow Y \mid \forall x, y : \text{dom } f \mid x = y \bullet f(x) = f(y)\} &\Leftrightarrow \\
\forall x, y : \text{dom } g \mid x = y \bullet g(x) = g(y) &
\end{aligned}$$

Basta então demonstrar que $\forall x, y : \text{dom } g \mid x = y \bullet g(x) = g(y)$.

Da definição de domínio e de insert temos que

$$\begin{aligned}
\text{dom } g &= \text{dom } \text{insert}(f, (a, b)) \\
&= \{p : \text{insert}(f, (a, b)) \bullet p.2\} \\
&= \{p : \{q : X \times Y \mid q \in f \vee q = (a, b)\} \bullet p.2\} \\
&= \{p : X \times Y \mid p \in f \vee p = (a, b) \bullet p.2\} \\
&= \{c : X \mid c \in \text{dom } f \vee c = a\} \\
&= \text{dom } f \cup \{a\}
\end{aligned}$$

Então temos que

$$\begin{aligned} \forall x, y : \text{dom } g \mid x = y \bullet g(x) = g(y) &\Leftrightarrow \\ \forall x, y : \text{dom } f \cup \{a\} \mid x = y \bullet g(x) = g(y) \end{aligned}$$

Note que x pode pertencer a $\text{dom } f$ ou a $\{a\}$, e o mesmo vale para y , então podemos separar em quatro casos:

- i. $x \in \text{dom } f \wedge y \in \text{dom } f$
- ii. $x \in \text{dom } f \wedge y \in \{a\}$
- iii. $x \in \{a\} \wedge y \in \text{dom } f$
- iv. $x \in \{a\} \wedge y \in \{a\}$

No caso **2(b)i**, temos $\forall x, y : \text{dom } f \mid x = y \bullet g(x) = g(y)$, como restrito ao domínio de f , $g = f$, obtemos $\forall x, y : \text{dom } f \mid x = y \bullet f(x) = f(y)$, que é verdade visto que f é função parcial.

No caso **2(b)ii**, temos $\forall x : \text{dom } f; y : \{a\} \mid x = y \bullet g(x) = g(y)$, que equivale a $\forall x : \text{dom } f; \mid x = a \bullet g(x) = g(a)$, e como $a \notin \text{dom } f$, $\forall x : \text{dom } f; \mid x = a \bullet g(x) = g(a)$ é verdadeiro.

No caso **2(b)iii**, temos $\forall x : \{a\}; y : \text{dom } f \mid x = y \bullet g(x) = g(y)$, que equivale a $\forall y : \text{dom } f \mid a = y \bullet g(a) = g(y)$, de forma similar ao caso **2(b)ii**, como $a \notin \text{dom } f$, vale que $\forall y : \text{dom } f \mid a = y \bullet g(a) = g(y)$.

No caso **2(b)iv**, temos $\forall x : \{a\}; y : \{a\} \mid x = y \bullet g(x) = g(y)$, que equivale a $g(a) = g(a)$, que é trivialmente verdade.

Logo $\forall x, y : \text{dom } f \cup \{a\} \mid x = y \bullet g(x) = g(y)$ é verdade e portanto g é função parcial.

■

Observamos que a operação de inserção como definida acima não produz sempre funções parciais a partir de funções parciais, então vamos definir uma nova função insert_p específica para funções parciais que produz apenas funções parciais, baseados no teorema **2**.

$$\begin{array}{|l} \hline [X] \\ \hline \text{insert}_p : (X \rightarrow Y) \times (X \times Y) \rightarrow X \rightarrow Y \\ \hline \forall f : X \rightarrow Y; a : X; b : Y \bullet \text{insert}_p(f, a, b) = \text{if}(a \notin \text{dom } f) \text{ then } \text{insert}(f, (a, b)) \\ \text{else } \text{insert}(\{a\} \triangleleft f, (a, b)) \\ \hline \end{array}$$

Propomos como estrutura de dados para a função parcial a class `HashMap` de Java, e justificamos essa escolha com base na descrição dos métodos **put** e **remove**.

Segundo a documentação da Java API, o método **public V put(K key, V value)**

associates the specified value with the specified key in this map. If the map previously contained a mapping for this key, the old value is replaced.

Note que essa operação se comporta como a operação $insert_p$, quando o primeiro elemento do par a ser inserido não pertence ao domínio da função (mapeamento), de forma que o mapeamento resultante é uma função parcial (ver **Teorema 2**). Caso contrário ele substitui o par antigo pelo novo, dessa forma o mapeamento continua sendo uma função parcial.

Ainda segundo a documentação da Java API, o método **public V remove(Object key)**

removes the mapping for this key from this map if present.

Isto é, ele se comporta como a operação $delete$, e de acordo com o teorema 1 o mapeamento resultante é uma função parcial.

As funções parciais podem ser construídas a partir da relação vazia ($f_\emptyset$), através da aplicação das funções de inserção e eliminação, e funções derivadas. Assim, a função de tradução $\llbracket \cdot \rrbracket^{Java, Tipos}$ é definida sobre funções parciais da seguinte forma

$$\begin{aligned}\llbracket f_\emptyset \rrbracket^{Java, X, Y} &= (f = \text{new HashMap}()), \text{ where } f_\emptyset \in X \leftrightarrow Y \\ \llbracket insert_p(f, a, b) \rrbracket^{Java, X, Y} &= f.\text{put}(a, b), \text{ where } a \in X, b \in Y, f \in X \leftrightarrow Y \\ \llbracket delete(f, (a, b)) \rrbracket^{Java, X, Y} &= f.\text{remove}(a), \text{ where } a \in X, b \in Y, f \in X \leftrightarrow Y\end{aligned}$$

3.3.2 Definição da Interface

A interface do tipo função parcial é a mesma do tipo relação (ver Seção 3.2.2). A princípio, imaginamos que seria necessário alterar os tipos dos valores devolvidos por essas funções, porém isso não é desejável, como mencionado anteriormente, pois o resultado de algumas operações a funções não são necessariamente funções. Mantendo o tipos dos resultados como a interface **ZRelation**, possibilitamos a implementação de versões robustas dos métodos que produzam relações, ao invés de funções, quando necessário.

3.3.3 Implementação

A seguir apresentamos a definição das operações que podem manipular funções parciais de acordo com o padrão internacional [24]. Apresentamos ainda uma classe que implementa a interface **ZRelation** segundo as definições apresentadas no padrão internacional [24], levando em conta as restrições associadas a funções. Os algoritmos que implementam as operações foram definidos em termos das operações de inserção e eliminação propostas.

A implementação apresentada neste trabalho, utiliza o comando de iteração **for** da linguagem Java 5.0 [54] que permite iterar sobre coleções e vetores sem utilizar iteradores ou índices. A cada passo da iteração esse comando escolhe um elemento da coleção, e essa escolha pode ser descrita pela função *choose* definida a seguir.

$$\begin{array}{|l} \hline [X, Y] \\ \hline choose : \mathbb{P} X \rightarrow X \\ \hline \forall A : \mathbb{P} X \mid A \neq \emptyset \bullet choose(A) \in A \\ \hline \end{array}$$

A seguir apresentamos alguns dos operadores de funções parciais e suas respectivas implementações. A implementação completa da classe `BasicPFunction` pode ser encontrada no Apêndice G.

Pertinência e Continência

As funções parciais herdam dos conjuntos três relações básicas: pertinência, continência e continência própria. A seguir, apresentamos a implementação dos três métodos correspondentes a essas relações. A pertinência verifica se um par (x, y) pertence à função parcial, a continência verifica se uma função parcial está contida em outra e a continência própria verifica se, além de uma função parcial estar contida em outra, elas são diferentes.

```
@Override
public Boolean contains(X x, Y y) {
    Y aux = apply(x);
    if (aux == null)
        return false;
    if (aux.equals(y))
        return true;
    else
        return false;
}

@Override
public Boolean contains(ZRelation<X, Y> R) {
    for (X x : R.domain()) {
        Y y = R.apply(x);
        if (!contains(x, y))
            return false;
    }
    return true;
}

@Override
public Boolean contains_properly(ZRelation<X, Y> R) {
    return (!(contains(R) || equals(R)));
}
```

Domínio e Imagem de Funções Parciais

O domínio de uma função parcial consiste no conjunto dos elementos que são primeiro termo de algum par contido na função parcial e a imagem consiste no conjunto dos elementos que são segundo termo desses pares. A implementação desses operadores é dada em termos de métodos da classe `HashMap`, como a seguir.

```
@Override
public ZSet<X> domain() {
    return new BasicSet<X>(keySet());
}
```

```

    }

    @Override
    public ZSet<Y> range() {
        return new BasicSet<Y>(values());
    }

```

União de Funções Parciais

Como funções parciais são conjuntos de pares ordenados, podemos aplicar a elas o operador de união. A definição formal dessa operação é apresentada abaixo [24].

$$\frac{[X]}{- \cup - : \mathbb{P} X \times \mathbb{P} X \rightarrow \mathbb{P} X} \\ \forall a, b : \mathbb{P} X \bullet a \cup b = \{x : X \mid x \in a \vee x \in b\}$$

Podemos reescrevê-la substituindo o conjunto X da definição pelo conjunto das funções parciais de X em Y . Dessa forma, obtemos a seguinte definição.

$$\frac{[X, Y]}{- \cup - : (X \rightharpoonup Y) \times (X \rightharpoonup Y) \rightarrow (X \rightharpoonup Y)} \\ \forall a, b : X \rightharpoonup Y \bullet a \cup b = \{p : X \times Y \mid p \in a \vee p \in b\}$$

Note que se a intersecção dos domínios das funções parciais a serem unidas não for vazio, existem elementos de X aos quais tanto uma função parcial, quanto a outra, atribui valor. Se os valores atribuídos por essas funções parciais a pelo menos um desses elementos forem diferentes entre si, a união dessas funções parciais não será uma função parcial. Para que o operador produza apenas funções parciais, precisamos contemplar essa possibilidade. O algoritmo proposto para esse operador lida com essa possibilidade sobrescrevendo um valor já associado. Esse algoritmo é apresentado abaixo.

```

@Override
public ZRelation<X, Y> union(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R1);
    union(R2);
    return this;
}

public ZRelation<X, Y> union(ZRelation<X, Y> R) {
    for (X x : R.domain()) {
        for (Y y : R.apply(x)) {
            put(x, y);
        }
    }
}

```

```

    }
    return this;
}

```

Intersecção de Funções Parciais

A intersecção de funções, assim como a união, é aplicável devido ao fato de funções parciais serem conjuntos. Porém, essa operação não sofre do mesmo problema que a união, pois a intersecção de duas funções parciais é sempre uma função parcial.

$$\begin{array}{|l}
 \hline
 [X, Y] \\
 \hline
 _ \cap _ : (X \rightarrow Y) \times (X \rightarrow Y) \rightarrow (X \rightarrow Y) \\
 \hline
 \forall a, b : X \rightarrow Y \bullet a \cap b = \{p : X \times Y \mid p \in a \wedge p \in b\} \\
 \hline
 \end{array}$$

O algoritmo correspondente a essa operação é o seguinte.

```

@Override
public ZRelation<X, Y> intersection(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R1);
    intersection(R2);
    return this;
}

public ZRelation<X, Y> intersection(ZRelation<X, Y> R) {
    for (X x : domain()) {
        Y y = apply(x);
        if (!R.contains(x, y)) {
            remove(x, y);
        }
    }
    return this;
}

```

União e Intersecção Generalizadas de Funções Parciais

As versões generalizadas da união e intersecção consistem basicamente na iteração dos respectivos métodos sobre um conjunto de funções parciais. Note que, assim como os outros métodos, ela altera a instância atual da classe, de forma que se temos um conjunto de funções parciais e desejamos calcular, por exemplo, a união generalizada desse conjunto, devemos instanciar uma função parcial nova e chamar o método correspondente com o parâmetro adequado. Os métodos correspondentes a esses operadores são apresentados a seguir.

```

}

@Override
public ZRelation<X, Y> generalized_intersection(ZSet<ZRelation<X, Y>> R) {

```

```

    clear();
    Iterator<ZRelation<X, Y>> it = R.iterator();
    if (!it.hasNext())
        return this;
    else
        union(it.next());
    while (it.hasNext()) {
        intersection(it.next());
    }
    return this;
}

@Override
public ZRelation<X, Y> generalized_union(ZSet<ZRelation<X, Y>> R) {
    clear();
    for (ZRelation<X, Y> r : R) {
        union(r);
    }
    return this;
}

```

Diferença de Funções Parciais

Assim como a intersecção, a diferença de funções parciais sempre produz funções parciais, visto que apenas eliminamos pares de uma função parcial.

$$\begin{array}{|l}
 \hline
 [X, Y] \\
 \hline
 - \setminus - : (X \rightarrow Y) \times (X \rightarrow Y) \rightarrow (X \rightarrow Y) \\
 \hline
 \forall a, b : X \rightarrow Y \bullet a \setminus b = \{p : X \times Y \mid p \in a \wedge p \notin b\} \\
 \hline
 \end{array}$$

O algoritmo correspondente é:

```

@Override
public ZRelation<X, Y> difference(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    for (X x : R1.domain()) {
        Y y = R1.apply(x);
        if (!R2.contains(x, y))
            put(x, y);
    }
    return this;
}

```

Diferença Simétrica de Funções Parciais

A diferença simétrica entre duas funções parciais é composta pelos pares que estão em apenas uma das duas funções, se uma função contém o par (x, a) e a outra contém o par (x, b) e $a \neq b$.

diferente de b , então os pares são diferentes e, portanto, pertencem à diferença simétrica das duas funções. Assim, a diferença simétrica entre tais funções parciais não será uma função parcial.

$[X, Y]$
$- \ominus - : (X \multimap Y) \times (X \multimap Y) \rightarrow (X \multimap Y)$
$\forall a, b : X \multimap Y \bullet a \ominus b = \{p : X \times Y \mid \neg (p \in a \Leftrightarrow p \in b)\}$

O algoritmo correspondente à diferença simétrica utiliza o operador de união e, portanto, converte esse problema adotando a mesma solução adotada para a união de funções parciais. Esse algoritmo é apresentado a seguir.

```
@Override
public ZRelation<X, Y> symmetric_difference(ZRelation<X, Y> R1,
                                           ZRelation<X, Y> R2) {
    clear();
    union(R1);
    for (X x: R2.domain()) {
        Y y = R2.apply(x);
        if (contains(x,y))
            remove(x,y);
        else
            put(x,y);
    }
    return this;
}
```

Restrição de Domínio de Funções Parciais

A restrição de domínio recebe uma função parcial e um conjunto de elementos do mesmo tipo que os elementos do domínio da função e produz uma nova função cujo domínio contém os elementos que estão no domínio da função parcial dada e no conjunto dado. Os valores associados a esses elementos são exatamente os valores que a função parcial dada lhes associa.

$[X, Y] =$
$- \triangleleft - : \mathbb{P}X \times (X \multimap Y) \rightarrow (X \multimap Y)$
$\forall a : \mathbb{P}X; r : X \multimap Y \bullet a \triangleleft r = \{p : r \mid p.1 \in a\}$

O algoritmo correspondente é:

```
@Override
public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x : R.domain()) {
        Y y = R.apply(x);
        if (S.contains(x))
```

```

        put(x, y);
    }
    return this;
}

```

Restrição de Imagem de Funções Parciais

A restrição de imagem é um operador semelhante à restrição de domínio, porém a função parcial obtida através desse operador contém apenas os pares cujo segundo elemento pertence a um dado conjunto.

$$\begin{array}{l}
 \text{[X, Y]} \\
 \hline
 \text{--} \triangleright \text{--} : (X \rightarrowtail Y) \times \mathbb{P} Y \rightarrow (X \rightarrowtail Y) \\
 \hline
 \forall r : X \rightarrowtail Y; b : \mathbb{P} Y \bullet r \triangleright b = \{p : r \mid p.2 \in b\}
 \end{array}$$

O algoritmo correspondente a esse operador é o seguinte:

```

@Override
public ZRelation<X, Y> range_restriction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (X x: R.domain()) {
        Y y = R.apply(x);
        if (S.contains(y))
            put(x,y);
    }
    return this;
}

```

Subtração de Domínio de Funções Parciais

A subtração de domínio tem um efeito contrário à restrição de domínio. Os únicos pares mantidos na função parcial são aqueles cujo primeiro elemento não pertence a um conjunto dado.

$$\begin{array}{l}
 \text{[X, Y]} \\
 \hline
 \text{--} \triangleleft \text{--} : \mathbb{P} X \times (X \rightarrowtail Y) \rightarrow (X \rightarrowtail Y) \\
 \hline
 \forall a : \mathbb{P} X; r : X \rightarrowtail Y \bullet a \triangleleft r = \{p : r \mid p.1 \notin a\}
 \end{array}$$

O algoritmo correspondente é:

```

@Override
public ZRelation<X, Y> domain_subtraction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x : R.domain()) {
        Y y = R.apply(x);
        if (!S.contains(x))
            put(x, y);
    }
}

```

```

    }
    return this;
}

```

Subtração de Imagem de Funções Parciais

A subtração de imagem produz uma função parcial composta pelos pares da função parcial dada cujo segundo elemento não pertence ao conjunto dado.

$$\begin{array}{c}
 \hline \hline
 [X, Y] \\
 \hline
 - \triangleright - : (X \rightarrow Y) \times \mathbb{P} Y \rightarrow (X \rightarrow Y) \\
 \hline
 \forall r : X \rightarrow Y; b : \mathbb{P} Y \bullet r \triangleright b = \{p : r \mid p.2 \notin b\} \\
 \hline
 \end{array}$$

O algoritmo correspondente é:

```

@Override
public ZRelation<X, Y> range_subtraction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (X x: R.domain()) {
        Y y = R.apply(x);
        if (!S.contains(y))
            put(x,y);
    }
    return this;
}

```

Sobrescrita de Funções Parciais

O operador de sobrescrita recebe duas funções parciais e produz uma nova função que contém todos os pares contidos na segunda função parcial dada, e os pares da primeira função parcial dada cujo primeiro elemento não está contido no domínio da segunda. Note que a função parcial obtida através desse operador é a mesma função parcial obtida através da implementação sugerida para a união de funções parciais.

$$\begin{array}{c}
 \hline \hline
 [X, Y] \\
 \hline
 - \oplus - : (X \rightarrow Y) \times (X \rightarrow Y) \rightarrow (X \rightarrow Y) \\
 \hline
 \forall r, s : X \rightarrow Y \bullet r \oplus s = ((\text{dom } s) \triangleleft r) \cup s \\
 \hline
 \end{array}$$

O algoritmo correspondente é:

```

@Override
public ZRelation<X, Y> overriding(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R2);
    ZSet<X> dom2 = R2.domain();
}

```

```

    for (X x : R1.domain()) {
        if (!dom2.contains(x)) {
            Y y = R1.apply(x);
            put(x, y);
        }
    }
    return this;
}

```

Composição Relacional e Funcional

A composição relacional de duas funções parciais recebe duas funções parciais f e g e produz uma terceira função que corresponde à função $g(f())$. A seguir apresentamos a definição formal da composição relacional.

$$\begin{array}{c}
 \hline [X, Y, Z] \hline \hline
 \text{-- } \circ \text{ --} : (X \rightarrow Y) \times (Y \rightarrow Z) \rightarrow (X \rightarrow Z) \\
 \hline
 \forall r : X \rightarrow Y; s : Y \rightarrow Z \bullet r \circ s = \{p : r; q : s \mid p.2 = q.1 \bullet p.1 \mapsto q.2\}
 \end{array}$$

A composição funcional é definida de forma idêntica. A implementação proposta para esses operadores é apresentada a seguir.

```

@Override
public <T> ZRelation<X, Y> functional_composition(ZRelation<X, T> R1,
    ZRelation<T, Y> R2) {
    clear();
    for (X x : R1.domain()) {
        Y y = R2.apply(R1.apply(x));
        put(x, y);
    }
    return this;
}

@Override
public <T> ZRelation<X, Y> relational_composition(ZRelation<X, T> R1,
    ZRelation<T, Y> R2) {
    return functional_composition(R1, R2);
}

```

Inversão de Funções Parciais

O operador de inversão produz uma nova função parcial correspondente à inversa da função original.

$$\begin{array}{c}
 \hline [X, Y] \hline \hline
 \text{-- } \sim \text{ --} : (X \rightarrow Y) \rightarrow (Y \rightarrow X) \\
 \hline
 \forall r : X \rightarrow Y \bullet r \sim = \{p : r \bullet p.2 \mapsto p.1\}
 \end{array}$$

Note que caso a função original não seja injetora, a sua inversa não será uma função e, a princípio, esse operador não poderia ser aplicado. Porém, a implementação proposta contorna esse problema, atribuindo a um elemento inserido múltiplas vezes no domínio da função inversa apenas um dos valores associados a ele.

```
@Override
public ZRelation<Y, X> relational_inversion() {
    ZRelation<Y, X> f = new BasicFFunction<Y,X>();
    for (X x: domain()) {
        Y y = apply(x);
        if (f.domain().contains(y))
            throw new NotInjectionException();
        else
            f.put(y,x);
    }
    return f;
}
```

Imagem Relacional e Aplicação de Funções Parciais

A imagem relacional de uma função parcial recebe um conjunto de elementos e produz um novo conjunto contendo os valores da função parcial calculados em cada um dos elemento do conjunto recebido.

$$\begin{array}{|l} [X, Y] \\ \hline - (\mid - \mid) : (X \rightarrowtail Y) \times \mathbb{P} X \rightarrow \mathbb{P} Y \\ \hline \forall r : X \rightarrowtail Y; a : \mathbb{P} X \bullet r(\mid a \mid) = \{p : r \mid p.1 \in a \bullet p.2\} \end{array}$$

A implementação correspondente é a seguinte.

```
@Override
public ZSet<Y> relational_image(ZSet<X> S) {
    ZSet<Y> ring = new BasicSet<Y>();
    for (X x: S) {
        ring.add(apply(x));
    }
    return ring;
}
```

A aplicação de uma função parcial a um elemento do seu domínio devolve o elemento da imagem correspondente. O método que implementa essa operação é dado a seguir.

```
@Override
public Y apply(X x) {
    return get(x);
}
```

Fecho Transitivo e Fecho Reflexivo Transitivo

Assim como no caso de relações, o fecho reflexivo transitivo não foi implementado. O fecho transitivo foi implementado convertendo a função passada como parâmetro para uma relação e calculando o fecho transitivo dela. O método correspondente é apresentado abaixo.

```
@Override
public ZRelation<X, X> transitive_closure(ZRelation<X, X> R) {
    ZRelation<X,X> r = new BasicRelation<X,X>();
    for (X x: R.domain()) {
        X y = R.apply(x);
        r.put(x, y);
    }
    return r.transitive_closure(r);
}
```

Neste capítulo apresentamos as interfaces e implementações básicas para alguns tipos de dados definidos no arcabouço matemático [24]. Vale ressaltar que escolhemos não manter a hierarquia imposta pelo arcabouço matemático, pois acreditamos com isso flexibilizar a implementação dos tipos de dados. Por exemplo, poderíamos implementar o tipo *Relation* utilizando um banco de dados relacional, e a manter a implementação do tipo *Set* como sugerido. Caso tivessemos mantido a estrutura hierárquica, seria necessário se preocupar com o comportamento de todos os elementos relacionados hierarquicamente.

Capítulo 4

Transformação de Estados

Como visto anteriormente, uma das formas de se especificar um sistema é através da descrição das configurações ou estados que esse sistema admite, e das transições entre estados. Nesse capítulo, examinaremos de forma mais detalhada os estados, os mecanismos providos pela notação *Z* para representação de estados e como transformar trechos de especificação *Z* que descrevem estados em código Java.

Em um sistema simples ou programa que recebe alguns dados de entrada, processa-os e devolve alguns dados de saída, podemos definir o estado inicial do sistema como sendo o conjunto das suas variáveis e os valores associados a elas, quando a computação do programa é iniciada. Ao final da execução do programa obtemos o estado final caracterizado pelas variáveis e os valores associados a ela nesse ponto do programa [12]. Podemos ainda identificar estados intermediários associados a pontos específicos da execução do programa.

Um exemplo de sistema simples é aquele que recebe um conjunto de nomes e os devolve em ordem alfabética. Um possível sistema mais complexo poderia manter um conjunto de nomes e permitir ao usuário adicionar e apagar nomes desse conjunto. Nesse caso, temos um sistema interativo, que possivelmente não termina, isto é, permanece ativo, esperando a interação do usuário, que é composto por diversos programas, possivelmente acionados pelo usuário, que modificam esse estado. Um sistema simples, em geral, termina em um tempo finito e determinado.

O estado de sistemas simples freqüentemente é caracterizado pelas variáveis de entrada, de saída e por outras variáveis intermediárias declaradas pelo programa para armazenar valores a serem computados ao longo da execução do programa [12]. Nesse caso, ele deixa de existir após a execução do programa. Em um sistema complexo como o exemplificado acima, o estado do sistema tem um caráter mais permanente, ele não deixa de existir após uma modificação.

Nesse trabalho, estamos lidando com a especificação de sistemas complexos, composto de um ou mais estados e operações que modificam esses estados. Por esse motivo, a representação de estados é de suma importância para nós.

4.1 Representação de Estados

Em *Z*, a forma mais básica de declarar um estado é através de um elemento que o padrão internacional da linguagem denomina esquema. Porém, essa denominação conflita com a nomenclatura mais tradicional. Por esse motivo, referiremo-nos, nesse trabalho, a esse elemento como domínio.

Um domínio declara nomes e restrições sobre os valores associados a esses nomes. A Figura 4.1 apresenta um exemplo de domínio que declara os nomes x e y cujos valores associados pertencem ao conjunto dos números inteiros e cuja soma dos seus quadrados deve ser menor ou igual a 100.

$$x, y : \mathbb{Z} \mid x^2 + y^2 \leq 100$$

Figura 4.1: Exemplo de um domínio

O padrão internacional [24] define também um elemento denominado construtor de domínio que consiste em um domínio entre colchetes (Figura 4.2). Assim como o domínio, esse elemento não é utilizado por si só em especificações, sendo utilizado como parte de outros elementos. Um domínio e um construtor de domínio são semanticamente equivalentes [24].

$$[x, y : \mathbb{Z} \mid x^2 + y^2 \leq 100]$$

Figura 4.2: Exemplo de um construtor de domínio

Embora o domínio seja a forma mais básica de se declarar um estado, não é a mais usada. O elemento mais usado para descrever estados é tradicionalmente denominado "esquema" [51, 62], enquanto o padrão internacional o denomina "definição de esquema" [24]. Nesse trabalho, referiremo-nos a esse elemento segundo a denominação tradicional. Um esquema é composto por um nome e um domínio [24]. A principal diferença entre um domínio e um esquema é o contexto em que os nomes e restrições declarados por cada elemento são introduzidos.

Os nomes e restrições declarados por um domínio são introduzidos globalmente ao contexto onde eles são declarados. O esquema por outro lado, cria um novo contexto associado ao seu nome, e introduz os nomes e restrições declarados por seu domínio nesse novo contexto [24]. Note que o nome do esquema é introduzido no mesmo contexto em que ele é declarado. Um exemplo de esquema é apresentado na Figura 4.3, esse esquema possui as mesmas variáveis e restrições que o domínio exemplificado na Figura 4.1, enfatizando que a diferença entre eles é o contexto em que esses nomes são inseridos.

<i>Ponto</i>
$x, y : \mathbb{Z}$
$x^2 + y^2 \leq 100$

Figura 4.3: Exemplo de um esquema

Existe uma segunda forma de se declarar um esquema, tradicionalmente, denominada esquema horizontal. A notação comum para a representação de um esquema horizontal utiliza um sinal de igual com um acento circunflexo sobre ele ($\hat{=}$) [51]. Porém, o padrão internacional utiliza dois

símbolos de igual ($==$) [24]. A Figura 4.4 mostra o mesmo esquema apresentado na Figura 4.3 na notação horizontal. Nessa notação, o fato de um esquema incluir um domínio é mais patente.

$$Ponto == [x, y : \mathbb{Z} \mid x^2 + y^2 \leq 100]$$

Figura 4.4: Exemplo de um esquema horizontal

4.2 Transformação em Código Java

Nessa seção apresentaremos as regras de transformação de esquemas em código Java. Iniciaremos descrevendo como transformar esquemas muito simples em classes Java. A partir da transformação de esquema simples, prosseguiremos apresentando a transformação de esquemas mais complexos.

4.2.1 Esquemas Simples

Vamos considerar um esquema simples como aquele composto apenas por declarações de variáveis e constantes. Uma declaração de variável é composta por um ou mais nomes e de um conjunto, e os elementos que podem ser associados a esses nomes declarados devem pertencer ao conjunto. Uma declaração de constante pode ser reduzida a uma declaração de variável cujo conjunto é unitário. Assumimos a existência de um mapeamento $\mathcal{M}$ entre os conjuntos associados a variáveis e tipos Java, e outro $\mathcal{T}$ entre elementos e seus tipos Java.

A Figura 4.5 apresenta um esquema simples composto por duas declarações de variáveis e uma declaração de constante.

Figura 4.5: Exemplo de um esquema simples

Para cada esquema simples S definiremos uma classe Java S_{impl} , tal que:

- para cada declaração da forma " $v_1, \dots, v_n : \mathbb{C}$ " contida em S , existe uma declaração de variável na classe S_{impl} da forma " $\mathcal{M}(\mathbb{C}) \ v_1, \dots, v_n;$ ", anotada com a anotação *AVariableDeclaration* definida no arcabouço;
- existe um construtor que permite inicializar as variáveis declaradas;
- para cada declaração da forma " $c == n$ " contida em S , existe uma declaração de variável estática na classe S_{impl} da forma "**static** $\mathcal{T}(n) \ c = n;$ ".

- A declaração " $x, y : \mathbb{Z}$ " será transformada na seguinte declaração de variáveis na classe Java correspondente: "`Integer x, y;`", onde $\mathcal{M}(\mathbb{Z}) = \text{Integer}$.
- A declaração " $z == 1$ " será transformada na seguinte declaração de variável estática na classe Java correspondente: "`static final Integer z = 1;`", onde $\mathcal{T}(1) = \text{Integer}$.

A partir do esquema apresentado na Figura 4.5, obtemos a seguinte classe Java:

```
public class Tripla {
    @AVariableDeclaration
    Integer x;
    @AVariableDeclaration
    Integer y;
    static final Integer z = 1;
    public Tripla(Integer x, Integer y) {
        this.x = x;
        this.y = y;
    }
}
```

Na prática, esse tipo de esquema quase não ocorre, e ele pode ser considerado da mesma forma que um esquema mais complexo. Ele foi explicitamente mencionado aqui, pois servirá de base para argumentarmos que o esquema e sua classe correspondente são aproximadamente equivalentes.

4.2.2 Esquema com Restrição

Podemos estender um esquema simples adicionando um predicado que restringe os valores que podem ser associados às variáveis declaradas. A Figura 4.3 corresponde a um esquema desse tipo.

Sua transformação em código Java segue o mesmo modelo apresentado na Seção 4.2.1 para esquemas simples. Porém, devido à existência do predicado é necessário que a classe correspondente ao esquema estenda a classe `Schema` definida no arcabouço JZed (ver Seção 2.4.2). Essa classe abstrata exige que suas sub classes implementem o método `checkSpecificationInvariants()`. A implementação desse método deve ser feita de forma a verificar o predicado contido no esquema.

Exemplo 4.2.2

O esquema apresentado na Figura 4.3 seria transformado no seguinte código:

```
public class Ponto {
    @AVariableDeclaration
    Integer x;
    @AVariableDeclaration
    Integer y;
    public Ponto (Integer x, Integer y) {
        this.x = x;
        this.y = y;
    }
    @Override
```

```

protected void checkSpecificationInvariants()
    throws InvariantViolation {
    Assert.isTrue(x*x + y*y <= 100);
}
}

```

4.2.3 Esquemas com Inclusão

Um esquema pode incluir outros esquema, nesse caso ele declara o nome do esquema a ser incluído junto às outras declarações. Esse recurso é bastante semelhante à herança em linguagens orientadas a objetos, e seu efeito consiste na inclusão de todas as declarações do esquema incluído e conjunção de seu predicado ao esquema que o inclui.

Exemplo 4.2.3

Tomemos os esquemas A e B definidos abaixo.

A
$x : \mathbb{Z}$
$x > 0$

B
A
$y : \mathbb{Z}$
$y > 0$

O esquema C apresentado a seguir é equivalente ao esquema B e consiste na expansão do esquema B .

C
$x : \mathbb{Z}$
$y : num$
$x > 0 \wedge y > 0$

A transformação de um esquema que inclui esquemas em uma classe Java poderia ser feita de forma trivial, expandindo o esquema como no Exemplo 4.2.3. Porém, essa abordagem pode prejudicar o objetivo de rastreamento de eventos. Por esse motivo, decidimos adotar uma estratégia diferente para lidar com inclusões de esquemas. Essa estratégia procura manter a estrutura da especificação na implementação de forma explícita.

A transformação de um esquema com inclusão é feita da mesma forma que para um esquema com restrição aplicando-se uma regra extra.

- Para cada esquema S_i incluído por um esquema S , o construtor da classe S_{impl} deve conter uma chamada ao método `importSchema(new  $S_i$ (..))`.

O métodos `importSchema` guarda a instância passada como parâmetro em uma lista de esquemas importados.

Em Z, se a mesma variável for declarada em diferentes níveis da estrutura de importação, elas deve ter o mesmo valor. Se usássemos o tratamento da inclusão por expansão não teríamos problema com esse fato. Porém, na abordagem escolhida, devido à replicação da estrutura de inclusão, podem ocorrer múltiplas declarações de variáveis que representam a mesma variável da especificação. Para contornar esse problema, nós exigimos que as definições de esquema a serem traduzidas estejam numa forma normal, tal que para cada esquema S , toda variável que foi declarada nas definições de esquema importadas por S deve também ser declarada em S . Essa normalização pode ser facilmente automatizada e, portanto, não precisa, necessariamente, ser aplicada no processo de especificação.

Exemplo 4.2.4

O esquema B apresentado no Exemplo 4.2.3 seria transformado no seguinte código Java.

```
public class B {
    @AVariableDeclaration
    Integer x;
    @AVariableDeclaration
    Integer y;
    public B(Integer x, Integer y) {
        this.x = x;
        this.y = y;
        importSchema(new A());
    }
    @Override
    protected void checkSpecificationInvariants()
        throws InvariantViolation {
        Assert.isTrue(y > 0);
    }
}
```

4.2.4 Esquemas Compostos

Podemos formar novos esquemas através de esquemas já definidos através de conectivos e quantificadores definidos pelo cálculo de esquemas. Os conectivos disponíveis são: negação, conjunção, disjunção, implicação e equivalência. Os quantificadores são o universal e o existencial. Embora esses conectivos e quantificadores sejam denotados pelos mesmos símbolos e tenham os mesmos nomes que os conectivos da lógica proposicional e quantificadores da lógica de predicados, sua semântica é diferente. Além desse elementos, são definidos os operadores de composição e encadeamento, voltados especificamente para operações. Por esse motivo, eles serão abordados no Capítulo 5.

O esquema formado pela aplicação de um conectivo binário $\odot$ a dois esquemas X e Y é denotado por $X \odot Y$ e consiste no esquema composto pelas união das declarações de X e Y , e pelo predicado formado pela aplicação do conectivo lógico equivalente aos predicados de X e Y .

Exemplo 4.2.5

Sejam dois esquemas A e B como definidos na Figura 4.6.

Figura 4.6: Exemplo de esquemas

Os esquemas resultantes da aplicação dos conectivos binários: conjunção, disjunção, implicação e equivalência a esses esquemas estão representados nas Figuras 4.7, 4.8, 4.9 e 4.10.

Figura 4.7: Conjunção de esquemas

Figura 4.8: Disjunção de esquemas

Figura 4.9: Implicação de esquemas

Figura 4.10: Equivalência de esquemas

A negação toma apenas um esquema e corresponde ao conjunto de associações que tem a mesma assinatura, isto é, declarações do esquema, porém não pertencem a ele. O seu efeito é negar o predicado, como mostra a Figura 4.11. Note que para aplicar o conectivo de negação é necessário expandir todos os esquemas incluídos [25].

$\neg A$
$a : \mathbb{N}$
$b : \mathbb{Z}$
$\neg (a < 10 \wedge b > -10 \wedge b < 10)$

Figura 4.11: Negação de esquemas

A quantificação de esquema tem a seguinte estrutura:

Q Declarações • Esquema,

onde Q corresponde ao quantificador. Ela consiste em eliminar as declarações do esquema quantificado que aparecem nas declarações da estrutura de quantificação e quantificar o predicado do esquema sobre elas. Por exemplo, a quantificação universal do esquema A definido na Figura 4.6 sobre a declaração $a : \mathbb{N}$ corresponde ao esquema apresentado na figura 4.12.

$\forall a : \mathbb{N} \bullet A$
$b : \mathbb{Z}$
$\forall a : \mathbb{N} \bullet a < 10 \wedge b > -10 \wedge b < 10$

Figura 4.12: Quantificação universal de esquemas

Nesse trabalho, não tratamos os quantificadores. Nos restringimos aos conectivos: negação, conjunção, disjunção, implicação e equivalência.

As regras de transformação de uma expressão do cálculo de esquemas com conectivos unários e binários é dada a seguir. Sejam S e T dois esquemas e S_{impl} e T_{impl} suas classes Java correspondentes. Então:

- $\neg S = \text{new SchemaCalcNOT}(\text{new } S_{impl}(\dots));$
- $S \vee T = \text{new SchemaCalcOR}(\text{new } S_{impl}(\dots), \text{new } T_{impl}(\dots));$
- $S \wedge T = \text{new SchemaCalcAND}(\text{new } S_{impl}(\dots), \text{new } T_{impl}(\dots));$
- $S \Rightarrow T = \text{new SchemaCalcIMPLIES}(\text{new } S_{impl}(\dots), \text{new } T_{impl}(\dots));$

- $S \Leftrightarrow T = \text{new SchemaCalcIFF}(\text{new } S_{\text{impl}}(\dots), \text{new } T_{\text{impl}}(\dots));$

As classes `SchemaCalcNOT`, `SchemaCalcOR`, `SchemaCalcAND`, `SchemaCalcIMPLIES` e `SchemaCalcIFF` implementam a semântica dos respectivos conectivos e são definidas no arcabouço `JZed`.

Na seção seguinte argumentaremos, para esquemas simples, que, embora o conjunto de estados determinados por um esquema e sua classe Java correspondente não sejam os mesmos, o conjunto de estados da classe Java está contido no conjunto descrito pelo esquema.

Exemplo 4.2.6

Para finalizar essa seção, vamos apresentar o código referente ao esquema *BirthdayBook* apresentado no Exemplo 2.3.1.

```
public class BirthdayBook extends Schema {
    @AVariableDeclaration
    protected BasicSet<String> known;
    @AVariableDeclaration
    protected BasicPFunction<String, GregorianCalendar> birthday;
    public BirthdayBook(BasicSet<String> known, BasicPFunction<String,
        GregorianCalendar> birthday) {
        // código do construtor
    }
    @Override
    protected void checkSpecificationInvariants() throws InvariantViolation {
        // código de verificação
    }
    @Override
    protected void checkUserInvariants() throws InvariantViolation {
        // código de verificação
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        // código de clonagem
    }
    @Override
    protected void execute() {
        // código funcional
    }
}
```

4.3 Correção da Transformação de Esquemas Simples

Essa argumentação foi feita com base na semântica formal de `Z` apresentada em [24] e em uma semântica informal de Java baseada em [18]. Vale notar que existem propostas de semântica formal para Java, tais como [14]. Porém, em geral, elas se baseiam em alguma descrição informal, como [18].

Embora apresentemos um argumento informal para o fato de que toda instância de uma classe pertence ao conjunto de estados descrito pelo esquema associado, não pretendemos demonstrar que

o mesmo vale para a transformação de sistemas mais complexos. Essa demonstração fica como um trabalho futuro.

Nas seções que seguem apresentamos a semântica de esquemas segundo o padrão internacional [24] (4.3.1), uma possível semântica bastante simples da definição de classes Java (4.3.2) e a comparação entre os modelos (4.3.3).

4.3.1 Semântica de Esquemas

Um esquema é um domínio associado a um nome, seu efeito é introduzir esse nome e associá-lo ao conjunto de nomes declarados no domínio. Apresentaremos a semântica de esquemas, iniciando pela interpretação de domínios simples e aumentando a complexidade até chegarmos aos esquemas propriamente ditos.

No que segue, assumimos um universo semântico $\mathbb{U}$ e um conjunto de nomes $NOME$. Um modelo $\mathcal{M}$ é uma função finita de $NOME$ em $\mathbb{U}$.

Consideremos o domínio mais simples possível que é aquele constituído por uma única declaração de variável, como na Figura 4.13.

$$i : e$$

Figura 4.13: Domínio simples, onde i é um nome e e uma expressão que denota um conjunto

Ele pode ser reescrito, segundo regras de transformação descritas no padrão internacional [24], na forma de um esquema, como na figura 4.14.

$$[i : e]$$

Figura 4.14: Construtor de domínio equivalente ao esquema da Figura 4.13

O significado desse construtor de domínio é o conjunto de todos os modelos $\mathcal{M}$, tais que o único nome contido no domínio de $\mathcal{M}$ é i e o valor associado a i está no conjunto denotado por e [24]. Suponha que podemos calcular o conjunto definido por uma expressão e através da função $\mathcal{V}$. Nesse caso, podemos definir uma função de interpretação, $\mathcal{I}$, que toma um construtor de domínio simples e devolve o seu significado como na figura 4.15.

$$\mathcal{I}([i : e]) = \{\mathcal{M} : NOME \mapsto \mathbb{U} \mid \text{dom } \mathcal{M} = \{i\} \wedge \mathcal{M}(i) \in \mathcal{V}(e) \bullet \mathcal{M}\}$$

Figura 4.15: Interpretação de construtores de domínio simples

Um construtor de domínio composto por diversas declarações de variáveis, como na figura 4.16, pode ser transformado, segundo as regras de transformação descrita no padrão internacional [24], no construtor de domínio apresentado na figura 4.17. O significado desse novo domínio corresponde à união das associações em cada um dos construtores de domínio.

Podemos estender a função de interpretação, $\mathcal{I}$, para construtores de domínio compostos como na figura 4.18.

$$[i_1, i_2 : e_1; i_3 : e_3]$$

Figura 4.16: Construtor de domínio composto

$$[i_1 : e_1] \wedge [i_2 : e_1] \wedge [i_3 : e_3]$$

Figura 4.17: Construtor de domínio composto transformado

$$\mathcal{I}([i_1 : e_1] \wedge [i_2 : e_2]) = \{t_1 : \mathcal{I}([i_1 : e_1]); t_2 : \mathcal{I}([i_2 : e_2]) \bullet t_1 \cup t_2\}$$

Figura 4.18: Interpretação de construtores de domínio compostos

Exemplo 4.3.1

Sejam os construtores de domínio dados pelas Figuras 4.19 e 4.20. O resultado da aplicação da função $\mathcal{I}$ a cada um deles é dado na figura 4.21 e o resultado da aplicação ao esquema composto por eles é dado na figura 4.22.

$$[i_1 : \{1, 2, 3\}]$$

Figura 4.19: Construtor de domínio simples 1

$$[i_2 : \{a, b, c\}]$$

Figura 4.20: Construtor de domínio simples 2

$$\begin{aligned} \mathcal{I}([i_1 : \{1, 2, 3\}]) &= \{\{i_1 \mapsto 1\}, \{i_1 \mapsto 2\}, \{i_1 \mapsto 3\}\} \\ \mathcal{I}([i_2 : \{a, b, c\}]) &= \{\{i_2 \mapsto a\}, \{i_2 \mapsto b\}, \{i_2 \mapsto c\}\} \end{aligned}$$

Figura 4.21: Aplicação de $\mathcal{I}$ aos construtores de domínio das Figuras 4.19 e 4.20

$$\begin{aligned} \mathcal{I}([i_1 : \{1, 2, 3\}] \wedge [i_2 : \{a, b, c\}]) &= \{\{i_1 \mapsto 1, i_2 \mapsto a\}, \{i_1 \mapsto 1, i_2 \mapsto b\}, \{i_1 \mapsto 1, i_2 \mapsto c\}, \\ &\quad \{i_1 \mapsto 2, i_2 \mapsto a\}, \{i_1 \mapsto 2, i_2 \mapsto b\}, \{i_1 \mapsto 2, i_2 \mapsto c\}, \\ &\quad \{i_1 \mapsto 3, i_2 \mapsto a\}, \{i_1 \mapsto 3, i_2 \mapsto b\}, \{i_1 \mapsto 3, i_2 \mapsto c\}\} \end{aligned}$$

Figura 4.22: Aplicação de $\mathcal{I}$ à conjunção dos construtores de domínio das Figuras 4.19 e 4.20

Um construtor de domínio pode ainda ter uma declaração constante $i == e$. Nesse caso ela pode ser transformada em uma declaração de variável, $i : \{e\}$, e seu significado será exatamente o significado da declaração de variável. Podemos estender a função $\mathcal{I}$ para construtores de domínio com declarações constantes como na Figura 4.23.

$$\mathcal{I}([i == e]) = \mathcal{I}([i : \{e\}])$$

Figura 4.23: Função $\mathcal{I}$ estendida a construtores de domínio simples com declarações constantes

Por exemplo, tomemos o construtor de domínio simples com declaração constante $[i == 1]$, a aplicação da função $\mathcal{I}$ é dada na figura 4.24.

$$\mathcal{I}([i == 1]) = \mathcal{I}([i : \{1\}]) = \{\{i \mapsto 1\}\}$$

Figura 4.24: Aplicação da função $\mathcal{I}$ a um construtor de domínio simples com declaração constante

O último tipo de declaração que um construtor de domínio pode ter é uma expressão, embora, usualmente, esse uso se restrinja a nomes de esquemas. No tratamento desse tipo de declaração, nos restringiremos a esses casos. Sua interpretação dependerá da interpretação de um nome de esquema, cujo significado será descrito mais à frente.

O significado de um construtor de domínio geral [Declarações | Predicado] é o conjunto dos modelos do construtor de domínio [Declarações] que satisfazem Predicado. Novamente, podemos estender a função $\mathcal{I}$ para construtor de domínio gerais como na figura 4.25. Para isso, precisamos definir o efeito de um modelo sobre um predicado.

Seja $\mathcal{M}$ um modelo e p um predicado, o efeito de $\mathcal{M}$ sobre p , denotado neste trabalho por $p[\mathcal{M}]$, é o predicado obtido substituindo-se todas as variáveis livres de p que estão no domínio de $\mathcal{M}$ pelos seus valores correspondentes.

$$\mathcal{I}([Declarações | Predicado]) = \{t : \mathcal{I}([Declarações]) \mid Predicado[t]\}$$

Figura 4.25: Aplicação da função $\mathcal{I}$ a construtor de domínio geral

Vale lembrar que um domínio é equivalente ao construtor de domínio obtido a partir dele, dessa forma a função de interpretação, $\mathcal{I}$, pode ser estendida para domínio, como mostra a figura 4.26.

$$\mathcal{I}(\text{Domínio}) = \mathcal{I}([\text{Domínio}])$$

Figura 4.26: Aplicação da função $\mathcal{I}$ a um domínio

Consideremos um esquema qualquer, ele tem um nome e um domínio, como na figura 4.27.

Figura 4.27: Esquema Geral

Pelas regras de transformação apresentadas no padrão internacional [24], esse esquema pode ser reescrito na forma de uma descrição axiomática, como na figura 4.28. Uma descrição axiomática consiste basicamente de um domínio, de forma que os nomes e restrições declarados são introduzidos no contexto global.

Essa descrição axiomática contém uma declaração de constante que pode ser simplificada, obtendo a descrição axiomática apresentada na figura 4.29.

$$\mid \text{ Nome } == \text{ Domínio }$$

Figura 4.28: Descrição axiomática equivalente a um esquema

$$\mid \text{ Nome } : \{ \text{Domínio} \}$$

Figura 4.29: Descrição axiomática simplificada

A interpretação do domínio simples $\text{Nome} : \{ \text{Domínio} \}$, como definida na figura 4.15, é o conjunto de todos modelos $\mathcal{M}$ tais que $\text{dom } \mathcal{M} = \{ \text{Nome} \}$ e $\mathcal{M}(\text{Nome}) \in \{ \text{Domínio} \}$. No entanto, $\{ \text{Domínio} \}$ denota um conjunto unitário. Logo, o único valor passível de associação a Nome é Domínio , cuja interpretação é dada pela função $\mathcal{I}$, como descrita na figura 4.26.

Note que o modelo obtido de uma esquema contém apenas um nome e este nome está associado a um conjunto de modelos.

Exemplo 4.3.2

Seja o esquema apresentada na Figura 4.30. O processo de interpretação consiste em, inicialmente, transformar sintaticamente o esquema em uma descrição axiomática, como mostra a figura 4.31.

Em seguida, calcula-se a interpretação da descrição axiomática.

$$\begin{aligned} \mathcal{I}(\text{PontoNoroeste} : \{ [x : \{1, 2, 3\}; y : \{-1, -2, -3\} \mid x + y \neq 0] \}) = \\ \{ \text{PontoNoroeste} \mapsto \mathcal{I}([x : \{1, 2, 3\}; y : \{-1, -2, -3\} \mid x + y \neq 0]) \} = \\ \{ \text{PontoNoroeste} \mapsto \{ t : \mathcal{I}([x : \{1, 2, 3\}; y : \{-1, -2, -3\}]) \mid (x + y \neq 0)[t] \} \} \end{aligned}$$

Nesse ponto, precisamos calcular a interpretação de um construtor de domínio intermediário, $[x : \{1, 2, 3\}; y : \{-1, -2, -3\}]$.

$$\mathcal{I}([x : \{1, 2, 3\}; y : \{-1, -2, -3\}]) =$$

<i>PontoNoroeste</i>	
$x : \{1, 2, 3\}$	
$y : \{-1, -2, -3\}$	
$x + y \neq 0$	

Figura 4.30: Exemplo de esquema descrevendo um ponto restrito a uma determinada região

<i>PontoNoroeste</i>
$x : \{1, 2, 3\}$
$y : \{-1, -2, -3\}$
$x + y \neq 0$

$$\Downarrow$$

$$| \text{PontoNoroeste} == [x : \{1, 2, 3\}; y : \{-1, -2, -3\} \mid x + y \neq 0]$$

$$\Downarrow$$

$$| \text{PontoNoroeste} : \{[x : \{1, 2, 3\}; y : \{-1, -2, -3\} \mid x + y \neq 0]\}$$

Figura 4.31: Transformações sintáticas do esquema

$$\begin{aligned}
& \mathcal{I}([x : \{1, 2, 3\}] \wedge [y : \{-1, -2, -3\}]) = \\
& \mathcal{I}([x : \{1, 2, 3\}] \wedge [y : \{-1, -2, -3\}]) = \\
& \{t_1 : \mathcal{I}([x : \{1, 2, 3\}]); t_2 : \mathcal{I}([y : \{-1, -2, -3\}]) \bullet t_1 \cup t_2\} = \\
& \{t_1 : \{\{x \mapsto 1\}, \{x \mapsto 2\}, \{x \mapsto 3\}\}; t_2 : \{\{y \mapsto -1\}, \{y \mapsto -2\}, \{y \mapsto -3\}\} \bullet t_1 \cup t_2\} = \\
& \{\{x \mapsto 1, y \mapsto -1\}, \{x \mapsto 1, y \mapsto -2\}, \{x \mapsto 1, y \mapsto -3\}, \\
& \{x \mapsto 2, y \mapsto -1\}, \{x \mapsto 2, y \mapsto -2\}, \{x \mapsto 2, y \mapsto -3\}, \\
& \{x \mapsto 3, y \mapsto -1\}, \{x \mapsto 3, y \mapsto -2\}, \{x \mapsto 3, y \mapsto -3\}\}
\end{aligned}$$

A partir desse cálculo, podemos retomar o cálculo da interpretação da descrição axiomática do ponto onde paramos.

$$\begin{aligned}
& \{\text{PontoNoroeste} \mapsto \{t : \mathcal{I}([x : \{1, 2, 3\}]; y : \{-1, -2, -3\}]) \mid (x + y \neq 0)[t]\}\} = \\
& \{\text{PontoNoroeste} \mapsto \{t : \{\{x \mapsto 1, y \mapsto -1\}, \{x \mapsto 1, y \mapsto -2\}, \{x \mapsto 1, y \mapsto -3\}, \\
& \{x \mapsto 2, y \mapsto -1\}, \{x \mapsto 2, y \mapsto -2\}, \{x \mapsto 2, y \mapsto -3\}, \\
& \{x \mapsto 3, y \mapsto -1\}, \{x \mapsto 3, y \mapsto -2\}, \{x \mapsto 3, y \mapsto -3\}\} \mid (x + y \neq 0)[t]\}\} = \\
& \{\text{PontoNoroeste} \mapsto \{\{x \mapsto 1, y \mapsto -2\}, \{x \mapsto 1, y \mapsto -3\}, \\
& \{x \mapsto 2, y \mapsto -1\}, \{x \mapsto 2, y \mapsto -3\}, \\
& \{x \mapsto 3, y \mapsto -1\}, \{x \mapsto 3, y \mapsto -2\}\}\}
\end{aligned}$$

Finalmente, a interpretação do esquema *PontoNoroeste* é o modelo que associa o nome *PontoNoroeste* ao conjunto de modelos $\{\{x \mapsto 1, y \mapsto -2\}, \{x \mapsto 1, y \mapsto -3\}, \{x \mapsto 2, y \mapsto -1\}, \{x \mapsto 2, y \mapsto -3\}, \{x \mapsto 3, y \mapsto -1\}, \{x \mapsto 3, y \mapsto -2\}\}$, como descrito acima.

Note que poderíamos especificar esse conjunto de modelos de formas diferentes, como feito nas

Figura 4.32: *PontoNoroeste1* Figura 4.33: *PontoNoroeste2*

Figuras 4.32 e 4.33.

4.3.2 Semântica da Definição de Classe em Java

Uma classe Java é composta de um identificador e um conjunto de declarações possivelmente vazio. Essas declarações podem ser de campos, métodos, classes e interfaces. Os campos declarados são variáveis associadas a tipos da linguagem. Uma classe determina um conjunto de objetos identificados, dentre os quais aqueles que são instâncias diretas da classe possuem todos os elementos declarados pela classe.

Podemos dizer que duas instâncias de uma classe são iguais¹ se seus métodos são iguais e seus campos são iguais e possuem os mesmos valores. Se as instâncias são instâncias diretas da mesma classe, os métodos delas são necessariamente iguais, assim como os campos. Porém os valores associados aos campos podem diferir, então podemos dizer que duas instâncias diretas de uma mesma classe são iguais se os valores associados aos campos declarados pela classe são iguais. A partir dessa definição de igualdade, podemos determinar um conjunto de instâncias diretas de uma classe, em que todos os elementos dele são diferentes de acordo com essa definição de igualdade.

A definição de igualdade adotada na construção do conjunto de instâncias permite que uma instância seja caracterizada por uma função finita entre seus nomes e valores e, assim como no caso da semântica de Z, designaremos essas funções finitas por modelos.

Dessa forma, tanto o esquema em Z, quanto a definição de classe em Java, são definidos em termos de conjuntos de modelos associados a um nome.

4.3.3 Comparação entre os Modelos

A semântica da definição de classe, como apresentada acima, nos permite comparar definições de esquemas em Z a definições de classes em Java. Seria interessante que a interpretação da definição de classe obtida pela aplicação do mapeamento a um esquema fosse o mesmo conjunto de modelos

¹Essa não é a definição de igualdade de Java

que a interpretação do esquema correspondente. Porém, isso nem sempre é válido. Tomemos, por exemplo, o esquema apresentada na Figura 4.34, uma definição de classe candidata a ser o mapeamento dele seria a apresentada na Figura 4.35.

```
public class Exemplo {
    int x;
}
```

Figura 4.34: Exemplo de esquema simples

Figura 4.35: Classe candidata a mapeamento do esquema da Figura 4.34.

O conjunto dos modelos do esquema *exemplo* pode ser descrito como o conjunto de todas as funções finitas com domínio unitário contendo o nome x e o associando a algum número inteiro (Equação 4.1).

$$\{v : \mathbb{Z} \bullet \{x \mapsto v\}\} \quad (4.1)$$

O conjunto de modelos da definição de classe **Exemplo**, por sua vez, pode ser descrito como o conjunto de todas as funções finitas com domínio unitário contendo o nome x e o associando a algum número inteiro menor que $2^{31} - 1$. Portanto, o conjunto de modelos da definição de esquema não é igual ao conjunto de modelos da definição de classe. Embora não possamos dizer que eles são iguais, eles estão relacionados de uma forma interessante, o conjunto de modelos da definição de classe está contido no conjunto de modelos do esquema, isto é, todo modelo da definição de classe é também um modelo do esquema.

Se interpretarmos um modelo como um estado possível, isso significa que uma instância da classe **Exemplo** nunca entra em um estado que não seja compatível com os estados válidos para o esquema *Exemplo*. A propriedade de produzir definições de classe cujos modelos são modelos do esquema é desejável, visto que nem sempre é possível que os modelos do esquema sejam modelos da definição de classe.

Neste capítulo apresentamos a estratégia de tradução de esquemas de estado. Note que a implementação de um esquema de estado deve prover métodos de verificação de invariantes de estado e, opcionalmente, um métodos funcional que pode manipular o estado. Vale frisar que o usuário não precisa se preocupar em chamar os métodos de verificação e execução, pois o arcabouço provido se encarrega disso.

Capítulo 5

Transformação de Operações

Em notações como Z, operações são transformadores de estado. Elas recebem um conjunto de variáveis de estado e alteram seus valores de forma a atingir um objetivo. A especificação de uma operação deve descrever que objetivo deseja-se atingir. Uma forma bastante comum de especificar o comportamento de uma operação é apresentando um predicado que, dado o estado inicial, declara a relação entre este e o estado final [12]. Por exemplo, a operação raiz quadrada (*sqr*t) aplicada a um número inteiro n pode ser descrita pelo predicado: $sqr(t(n) * sqr(t(n) = n$. Porém, essa especificação contém um erro. A operação raiz quadrada não está definida sobre números negativos, isto é, existe uma condição que deve valer para que ela possa ser aplicada. Essa condição pode ser descrita por outro predicado: $n \geq 0$. Note que nessa especificação, o estado inicial é a variável n e o estado final é a operação *sqr*t aplicada a n . Em Z, temos mecanismos convencionais que permitem a distinção entre um estado inicial e final, além de meios para identificar variáveis de entrada (como n) e variáveis de saída.

Ao predicado que determina as condições que devem valer para que a operação possa ser aplicada denominamos pré-condição da operação, e ao predicado que relaciona o estado inicial e o estado final denominamos pós-condição da operação. Para que uma operação seja aplicada com sucesso, é necessário que os dois predicados valham. Porém, freqüentemente, o comportamento da operação quando a pré-condição falha é considerado indeterminada. Nesse trabalho, em particular, consideramos que quando uma pré-condição falha, a operação associada não é executada e uma exceção é lançada.

Vale notar que, ao contrário de notações como VDM, Z não apresenta meios de declarar, separadamente, a pré e a pós-condição de uma operação. Porém, é possível identificar que partes de um predicado fazem parte da pré-condição da operação, e que partes fazem parte da pós-condição.

Dada a especificação de uma operação, a pré-condição dela pode ser calculada escondendo-se os componentes correspondentes aos estados finais da operação e às variáveis de saída [62]. Como assumimos que a operação não é executada quando a pré-condição falha, só faz sentido calcular a pós-condição nas situações em que a pré-condição vale. Dessa forma, é possível considerar o predicado todo como pós-condição, visto que a pré-condição é verdadeira¹.

¹Note que isso só é possível, porque em Z o estado inicial e o estado final do sistema são representados de forma independente.

5.1 Representação de Operações

Como mencionado na Seção 2.3, a rigor, não existem formas específicas de se descrever uma operação em $\mathbb{Z}$. O que se convencionou descrever uma operação é na verdade um esquema que inclui esquemas duplicados, um representando o estado inicial e outro o estado final. Por exemplo, a figura 5.1 define um esquema *Ponto* composto de dois números inteiros, x e y , e um esquema que especifica uma operação que desloca um ponto à esquerda.

Figura 5.1: Esquema de estado *Ponto* e esquema de operação *Esquerda*

O esquema *Ponto* representa o estado inicial e o esquema *Ponto'* representa o estado final. O predicado do esquema determina a relação entre o estado inicial e final.

5.1.1 Convenções

Como foi visto na Seção 2.3, existem algumas convenções que regem a especificação de operações em $\mathbb{Z}$. A seguir, as detalharemos.

Decoração de esquemas

Um esquema decorado é construído concatenando um apóstrofe ao final do nome do esquema original. Por exemplo, tomemos o esquema *Reta* apresentado na Figura 5.2. O esquema decorado *Reta'* pode ser escrito em sua forma completamente expandida como na Figura 5.3

Figura 5.2: Esquema que especifica uma reta

Figura 5.3: Esquema decorado *Reta'*

Esquema Δ

Um esquema Δ é formada concatenando-se a letra grega Δ ao início do nome do esquema original. Seu efeito é a formação de um novo esquema composto pelo esquema original e por uma versão decorada dele. Por exemplo, o esquema $\Delta Reta$ pode ser reescrito da seguinte forma:

$\Delta Reta$
$Reta$
$Reta'$

Uma operação que modifica uma reta, em geral, inclui o esquema $\Delta Reta$, e estabelece a relação entre os esquemas $Reta$ e $Reta'$.

Esquema Ξ

O esquema Ξ é semelhante ao esquema Δ na medida em que é formado concatenando-se a letra Ξ ao início do nome do esquema original e forma um novo esquema composto pelo esquema original e pela versão decorada dele. A diferença entre eles é que o esquema Ξ introduz restrições sobre as variáveis². Por exemplo, o esquema $\Xi Reta$ pode ser expandido da seguinte forma:

$\Xi Reta$
$Reta$
$Reta'$
$p_1 = p'_1$
$p_2 = p'_2$

Existe uma notação que permite representar essa restrição de forma mais concisa denominada expressão Theta (θ). Uma expressão Theta composta por um determinado esquema constrói a associação entre os nomes da assinatura (declarações) do esquema e os valores das variáveis de mesmo nome no escopo da expressão. Por exemplo, a expressão $\theta Reta$ pode ser reescrita como $\langle p_1 == p_1, p_2 == p_2 \rangle$. A Figura 5.4 adaptada de [62] pode deixar mais claro o significado dessa expressão.

$$\langle p_1 == p_1, \underbrace{p_2}_{\text{nome do componente}} == \overbrace{p_2}^{\text{valor da variável}} \rangle$$

Figura 5.4: Resultado da expressão $\theta Reta$

A partir da Figura 5.4 fica claro que a segunda ocorrência do nome p_2 denota o valor da variável p_2 , e, portanto, essa variável deve ter sido definida anteriormente.

Com essa expressão podemos definir de forma genérica o esquema Ξ correspondente. Para algum esquema A , o esquema ΞA pode ser reescrito assim:

²O arcabouço provido não garante essa propriedade, um possível tratamento seria a introdução de um invariante implícito

ΞA	
A	
A'	
$\theta A = \theta A'$	

O esquema Ξ é bastante utilizado quando descrevemos operações que não alteram o estado do sistema. Por exemplo, uma consulta a algum valor do estado do sistema.

Variáveis de entrada e saída

Variáveis de entrada e saída são como quaisquer variáveis declarada em um esquema. A diferença entre elas é a concatenação de símbolos que explicitam o ato de serem variáveis de entrada ou saída. O nome de uma variável de entrada contém um ponto de interrogação concatenado ao seu final e uma variável de saída contém um ponto de exclamação no mesmo lugar. A declaração dessas variáveis é feita da forma tradicional. Por exemplo, uma operação que devolve o valor de um dos pontos extremos de uma reta pode ser descrita da seguinte forma:

<i>PegaPonto</i>	
$\Xi Reta$	
$p! : Ponto$	
$p! = p_1 \vee p! = p_2$	

5.1.2 Cálculo de Esquemas

Os operadores do cálculo de esquema também se aplicam a esquemas de operação, visto que esse não são diferentes de esquemas de estado. Enquanto essa uniformidade de tratamento dos operadores do cálculo de esquemas facilita a manipulação matemática de especificações, na transformação de especificações em código isso pode gerar alguns problema. Por esse motivo, a semântica dos operadores é adaptada de acordo com o tipo de esquema a que eles se aplicam. O arcabouço JZed provê classes que implementam alguns dos operadores, e essas classes atribuem a semântica apropriada ao operador.

Os operadores de composição ($\circ$) e encadeamento ($\gg$), assim como os quantificadores não foram implementados na versão atual do arcabouço. Ficando, portanto, como trabalho futuro.

5.2 Transformação em Código Java

Um esquema de operação pode incluir esquemas tradicionais, esquemas decorados, esquemas Δ e Ξ , além de variáveis de entrada e saída. A seguir apresentamos as regras de transformação para esquemas de operação, nos restringindo a esquema que incluem Δ esquemas e variáveis de entrada e saída.

5.2.1 Operação Geral

Para cada esquema de operação O definiremos uma classe Java O_{impl} que estende a classe abstrata `Operation` definida no arcabouço JZed. Essa classe deve implementar seu construtor, o método `checkSpecificationPreconditions()` de forma a verificar a pré-condição da operação, o método `checkSpecificationPostconditions()` de forma a verificar a pós-condição da operação e o método `execute(Map initial, Map input)` de forma a implementar efetivamente a operação. O mapeamento `initial` contém os esquemas a serem modificados pela operação, o mapeamento `input` contém as variáveis de entrada. O método `execute(..)` devolve um mapeamento contendo as variáveis de saída.

5.2.2 Inclusão de Esquemas Δ

A inclusão de um esquema ΔS_i implica a inclusão dos esquema S_i e S'_i , mas esse fato não é representado explicitamente na classe. Ao invés disso, o método `execute(Map<String, Schema> initial, Map<String, Object> input)` recebe cada S_i no mapeamento `initial` e o transforma em S'_i . Isso pode gerar problema para a verificação das pós-condições. Porém, esse problema é contornado pelo arcabouço produzindo uma cópia de cada S_i , permitindo, assim, a verificação das pós-condições após a execução do operação.

Exemplo 5.2.1

Tomemos os esquemas *Ponto* e *Esquerda* definidos no início deste capítulo e reproduzido na Figura 5.5

Figura 5.5: Esquema de estado *Ponto* e esquema de operação *Esquerda*

A operação *Esquerda* será transformada na seguinte classe Java:

```
public class Esquerda extends Operation {
    public Esquerda(String name) {
        super(name);
    }
    @Override
    public void checkSpecificationPostconditions(Map<String, Schema> before,
        Map<String, Schema> after, Map<String, Object> output)
        throws InvariantViolation {
        // código de verificação
    }
}
```

```

@Override
public void checkSpecificationPreconditions(Map<String, Schema> before,
    Map<String, Object> input) throws InvariantViolation {
    // código de verificação
}
protected Map<String, Object> execute(Map<String, Schema> initial,
    Map<String, Object> input) {
    // código funcional
}
}

```

5.2.3 Variáveis

Assim como no caso da transformação de estados, vamos assumir a existência de um mapeamento $\mathcal{M}$ entre os conjuntos associados a variáveis e tipos Java. Para cada variável de estado³ v_i de O , um campo de mesmo nome deve ser declarado na classe O_{impl} e anotado com a anotação *AVariableDeclaration* definida no arcabouço. O tipo de cada uma das variáveis é dado pelo mapeamento $\mathcal{M}$. As variáveis de entrada in_i de O são passadas ao método `execute` através do mapeamento `input`. As variáveis de saída são devolvidas através de um mapeamento pelo método `execute`.

Exemplo 5.2.2

Retomando o exemplo 2.3.1, apresentamos a seguir a classe Java que é obtida a partir do esquema *FindBirthday*.

```

public class FindBirthday extends Operation {
    public FindBirthday(String name) {
    }
    @Override
    public void checkSpecificationPostconditions(Map<String, Schema> before,
        Map<String, Schema> after, Map<String, Object> output)
        throws InvariantViolation {
    }
    @Override
    public void checkSpecificationPreconditions(Map<String, Schema> before,
        Map<String, Object> input) throws InvariantViolation {
    }
    @Override
    public void checkUserPostconditions(Map<String, Schema> before,
        Map<String, Schema> after, Map<String, Object> output)
        throws InvariantViolation {
    }
    @Override
    public void checkUserPreconditions(Map<String, Schema> before,

```

³Uma variável de estado em uma operação pode ser vista como uma variável auxiliar utilizada para guardar valores a serem computados ao longo da execução da operação


```
        Map<String, Object> input) throws InvariantViolation {
    }
    @Override
    protected Map<String, Object> execute(Map<String, Schema> initial,
        Map<String, Object> input) {
    }
    @Override
    protected void execute() {
    }
    protected Object clone() throws CloneNotSupportedException {
    }
}
```

Neste capítulo apresentamos a estratégia de tradução para esquemas de operações. Alguns detalhes merecem ser mencionados, assim como no caso de esquemas de estado, a implementação de esquemas de operação deve prover a implementação de alguns métodos: verificação de pré e pós-condições e execução da operação. Note que existem dois métodos `execute`, um é herdado da classe `State` e outro recebe como parâmetros os estados iniciais e as variáveis de entrada da operação. O usuário não precisa se preocupar com a verificação de pré-condições e pós-condições, pois o arcabouço subjacente se encarrega de verificar essas condições, além de checar o invariantes dos estados modificados.

Capítulo 6

Tradução de Expressões e Predicados

A partir da regras de tradução apresentadas no Capítulo 3 é possível traduzir expressões mais complexas encontradas, por exemplo, no predicado de um esquema.

Tomemos como exemplo o esquema *BirthdayBook* apresentado no Capítulo 2 e repetido aqui.

<i>BirthdayBook</i>
$known : \mathbb{P} NAME$
$birthday : NAME \leftrightarrow DATE$
$known = \text{dom } birthday$

Os conjuntos *NAME* e *DATE* são conjuntos dos quais não temos informação (vide Apêndice B) e, por esse motivo, devem ser associados, pelo usuário, a alguma classe Java. Nesse exemplo, vamos associá-los à classe `String`. Dessa forma, podemos traduzir os conjuntos associados às variáveis *known* e *birthday*, respectivamente, às classes Java `BasicSet<String>` e `BasicPFunction<String,String>`.

O predicado contido nesse esquema pode ser traduzido para a seguinte expressão:

`known.equals(birthday.domain())`

Assim, a partir do esqueleto gerado pelas regras de tradução de estados, obtemos a seguinte classe Java.

```
public class BirthdayBook extends Schema {
    @AVariableDeclaration
    public BasicSet<String> known;
    @AVariableDeclaration
    public BasicPFunction<String, String> birthday;

    public BirthdayBook(BasicSet<String> known,
        BasicPFunction<String, String> birthday) {
        super("BirthdayBook");

        this.known = known;
        this.birthday = birthday;
    }
}
```

```

@Override
protected void checkSpecificationInvariants() throws InvariantViolation {
    Assert.isTrue(known.equals(birthday.domain()));
}

@Override
protected void checkUserInvariants() throws InvariantViolation {
    // nothing
}

@Override
protected Object clone() throws CloneNotSupportedException {
    return new BirthdayBook((BasicSet<String>) known.clone(),
        (BasicPFunction<String, String>) birthday.clone());
}

@Override
protected void execute() {
    // nothing
}
}

```

Um exemplo um pouco mais complexo pode ser visto na operação *AddBirthday* reproduzida abaixo.

$AddBirthday$ $\Delta BirthdayBook$ $name? : NAME$ $date? : DATE$
$name? \notin known$ $birthday' = birthday \cup \{name? \mapsto date?\}$

Podemos identificar a pré-condição dessa operação como $name? \notin known$ e a pós-condição como $birthday' = birthday \cup \{name? \mapsto date?\}$. Com base nas regras de tradução apresentadas no Capítulo 3, podemos traduzir essas condições, respectivamente, para as seguintes expressões:

Pré-condição: `!known.contains(name)`

Pós-condição:

```

BasicPFunction<String,String> pair = new BasicPFunction<String,String>();
pair.put(name, date);
bbook_after.equals(bbook_before.birthday.union(pair));

```

Formalmente a pré-condição de uma operação é dada pela seguinte fórmula:

$$\text{pre } Operation = \exists State' \bullet Operation \text{ outputs}$$

Essa definição envolve a ocultação das variáveis de saída e do estado final da operação. A estratégia aplicada neste trabalho consiste em tomar o predicado da operação na forma normal conjuntiva e eliminar os termos que contém variáveis decoradas e de saída¹.

Assim, complementando o esqueleto Java gerado pelas regras de tradução de operações, obtemos a seguinte classe Java.

```
public class AddBirthday extends Operation {
    public AddBirthday(String name) {
        super(name);
    }

    @Override
    public void checkSpecificationPostconditions(Map<String, Schema> before,
        Map<String, Schema> after, Map<String, Object> input,
        Map<String, Object> output) throws InvariantViolation {
        BirthdayBook bbook_before = (BirthdayBook)before.get("BirthdayBook");
        BirthdayBook bbook_after = (BirthdayBook)after.get("BirthdayBook");
        String name = (String)input.get("name");
        String date = (String)input.get("date");
        BasicPFunction<String,String> pair = new BasicPFunction<String,String>();
        BasicPFunction<String,String> f = new BasicPFunction<String,String>();
        pair.put(name, date);
        f.union(bbook_before.birthday,pair);
        Assert.isTrue(bbook_after.equals(f));
    }

    public void checkSpecificationPreconditions(Map<String, Schema> before,
        Map<String, Object> input) throws InvariantViolation {
        BirthdayBook bbook = (BirthdayBook)before.get("BirthdayBook");
        String name = (String)input.get("name");
        Assert.isTrue(!bbook.known.contains(name));
    }

    @Override
    public void checkUserPostconditions(Map<String, Schema> before,
        Map<String, Schema> after, Map<String, Object> input,
        Map<String, Object> output) throws InvariantViolation {
        // nothing
    }

    @Override
```

¹Note que nós não verificamos formalmente se essas duas definições são equivalentes

```

public void checkUserPreconditions(Map<String, Schema> before,
    Map<String, Object> input) throws InvariantViolation {
    // TODO
}

@Override
protected Map<String, Object> execute(Map<String, Schema> initial,
    Map<String, Object> input) {
    // Put the output variables in this map (name -> value)
    Map<String, Object> output = new HashMap<String, Object>();
    BirthdayBook bbook = (BirthdayBook)initial.get("BirthdayBook");
    String name = (String)input.get("name");
    String date = (String)input.get("date");
    bbook.birthday.put(name, date);
    return output;
}

@Override
protected void execute() {
    // nothing
}

protected Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException("Not a state schema.");
}
}

```

Para cada estado e operação contido na especificação, geramos uma classes correspondente de acordo com a metodologia proposta, como foi feito acima para uma estado e uma operação. Após gerarmos todas as classes, podemos criar uma classe que representa a especificação como um todo. Essa classe deve estender a classe `Specification` provida pelo arcabouço JZed. Para cada classe correspondente a um esquema, devemos chamar o método `add` com uma instância dela. Para cada expressão do cálculo de esquema, devemos construir os esquema correspondentes através dos conectivos providos pelo arcabouço JZed. A seguir apresentamos a classe correspondente à especificação *BirthdayBook*.

```

package com.jzed.BBook;

import jzed.Specification;
import jzed.paragraph.MalformedSchemaCalculusFormula;
import jzed.paragraph.Schema;
import jzed.paragraph.SchemaCalcAND;
import jzed.paragraph.SchemaCalcOR;

import com.jzed.BBook.operation.AddBirthday;
import com.jzed.BBook.operation.AlreadyKnown;
import com.jzed.BBook.operation.FindBirthday;
import com.jzed.BBook.operation.InitBirthdayBook;

```

```

import com.jzed.BBook.operation.NotKnown;
import com.jzed.BBook.operation.Remind;
import com.jzed.BBook.operation.RemindOne;
import com.jzed.BBook.operation.Success;
import com.jzed.BBook.state.BirthdayBook;

public class BBook extends Specification {
    /**
     * A constructor that initializes all the necessary resources.
     */
    public BBook() {
        init();
    }

    public void init() {

        BirthdayBook aBirthdayBook = new BirthdayBook();
        add(aBirthdayBook);

        InitBirthdayBook aInitBirthdayBook = new InitBirthdayBook(
            "InitBirthdayBook");
        add(aInitBirthdayBook);

        AddBirthday aAddBirthday = new AddBirthday("AddBirthday");
        add(aAddBirthday);

        FindBirthday aFindBirthday = new FindBirthday("FindBirthday");
        add(aFindBirthday);

        Remind aRemind = new Remind("Remind");
        add(aRemind);

        RemindOne aRemindOne = new RemindOne("RemindOne");
        add(aRemindOne);

        Success aSuccess = new Success("Success");
        add(aSuccess);

        AlreadyKnown aAlreadyKnown = new AlreadyKnown("AlreadyKnown");
        add(aAlreadyKnown);

        NotKnown aNotKnown = new NotKnown("NotKnown");
        add(aNotKnown);

        // Schema calculus expressions
        try {
            Schema RAddBirthday;
            RAddBirthday = new SchemaCalcOR(new SchemaCalcAND(

```

```

                                aAddBirthday, aSuccess), aAlreadyKnown);
add(RAddBirthday);

Schema RFindBirthday = new SchemaCalcOR(new SchemaCalcAND(
                                aFindBirthday, aSuccess), aNotKnown);
add(RFindBirthday);

Schema RRemind = new SchemaCalcAND(aRemind, aSuccess);
add(RRemind);
} catch (MalformedSchemaCalculusFormula e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
}

```

Note que o código funcional e de verificação apresentado nos exemplo acima não foram gerados automaticamente. A seguir, apresentamos um visão bastante superficial das ferramentas desenvolvidas para dar suporte à metodologia descrita neste trabalho.

6.1 Ferramenta

A ferramenta JZed-Gen foi desenvolvida em Java como uma implementação parcial da metodologia proposta nesse trabalho. Ela utiliza como base as ferramentas do *Community Z Tools* (CZT) [30] para analisar especificações em Z, o arcabouço *Velocity* [8] para auxiliar a geração do código.

A interface gráfica da ferramenta possui três formulários a serem preenchidos:

- *Specification*: permite a seleção dos arquivos de especificação, a partir dos quais a aplicação será gerada;
- *Type Translation*: permite construir um mapeamento que associa os tipos dados contidos na especificação a classes Java (ver Figura 6.1);
- *Code Generation*: permite a escolha da pasta em que a aplicação final será gerada, o nome da aplicação e o pacote principal (ver Figura 6.2), e o acionamento do processo de geração de código.

O *plug-in* Eclipse desenvolvido permite a criação de um projeto denominado “Projeto JZed” e apresenta os mesmos três formulários da ferramenta mencionada acima.

Para utiliza o *plug-in* devemos clicar inicialmente criar um novo projeto JZed como na Figura 6.3. Ao selecionar o tipo do projeto, uma nova janela é aberta permitindo a seleção dos arquivos de especificação (Figura 6.4). No nosso exemplo, selecionamos o arquivo contendo a especificação *BirthdayBook* (Figura 6.5).

Após selecionarmos os arquivos de especificação, passamos à tela de seleção de tipos (Figura 6.6) e, em seguida, à tela de geração de código (Figura 6.7). Um exemplo do código gerado pode ser visto na Figura 6.8.

Figura 6.1: Construção do mapeamento de tipos

Figura 6.2: Geração de código

Neste capítulo apresentamos de forma bastante sucinta a sugestão de estratégia de tradução de expressões e predicados. Além disso, apresentamos algumas informações sobre as ferramentas desenvolvidas, entre elas um *plug-in* Eclipse que possibilita a criação de um projeto Java a partir de uma especificação Z, de acordo com a metodologia proposta.

Figura 6.3: Criação do projeto JZed

Figura 6.4: Seleção de especificações

Figura 6.5: Seleção da especificação *BirthdayBook*

Figura 6.6: Seleção dos tipos NAME e DATE

Figura 6.7: Detalhes da geração do código

Figura 6.8: Exemplo de código gerado

Capítulo 7

Conclusão

Nesse trabalho propomos uma metodologia prática para o desenvolvimento de sistemas computacionais a partir de especificações formais.

A proposta delineada nessa dissertação sugere o uso da notação Z como linguagem de especificação e utiliza algumas idéias do desenvolvimento baseado em modelos (MDD) [20]. Dentro das noções da arquitetura baseada em modelos (MDA) [42], uma instância do desenvolvimento baseado em modelos, podemos interpretar alguns elementos dessa metodologia da seguinte forma: a especificação Z é um modelo independente de plataforma, que será implementado sobre a API da linguagem Java. A metodologia proposta contempla o mapeamento entre o modelo base e a implementação na plataforma Java.

Nós propomos uma estratégia de transformação de esquemas de estado e operação para código Java e um artigo contendo parte desse trabalho foi aceito para publicação no *Simpósio Brasileiro de Métodos Formais* [35]. Apresentamos também estratégias de tradução de elementos definidos no arcabouço matemático.

Associada à metodologia proposta, foi desenvolvida uma ferramenta que implementa parte dela e um *plug-in* Eclipse que possibilita a criação de um projeto Java a partir de uma especificação Z segundo a metodologia proposta. Vale notar que as especificações tratadas, até agora, são aquelas compostas exclusivamente de definições de esquema que representam estados ou operações e expressões do cálculo de esquemas. Descrições axiomáticas, entre outros elementos de Z, não são tratados ainda. Além disso, os conjuntos associados às variáveis da especificação devem ser tipos simples (tipo dados e tipos definidos na especificação da notação Z, por exemplo, $\mathbb{N}$), tipos construídos a partir do conjunto das partes, relações ou funções parciais.

Alguns aspectos do trabalho desenvolvido poderiam ter sido trabalhos de forma diferente. Outros mereceriam um estudo mais profundo que não foi possível devido às limitações de tempo. A implementação dos elementos do arcabouço matemático poderia ter sido feita mantendo-se a hierarquia imposta pela especificação. Isso não foi feito, pois acreditamos que dessa forma flexibilizamos a implementação dos tipos de dados. Porém, talvez um estudo mais profundo do tema, nos mostre que a escolha feita não foi a mais correta, nem adequada. Outros aspectos que merecem um estudo mais aprofundado são a geração de código de verificação, a geração parcial de código funcional (através de técnicas de geração dedutiva de código ou técnicas de refinamento) e a verificação

formal das transformações propostas.

Devido às limitações de tempo e ao fato da metodologia proposta ser ainda bastante incompleta, não conseguimos testá-la com uma grande variedade de exemplo, nem com especificações de grande porte. Esperamos poder desenvolver este trabalho e aplicá-lo a um grande número de especificações, inclusive especificações reais, de forma a avaliar as vantagens e desvantagens da aplicação dessa metodologia.

7.1 Trabalhos Futuros

Uma extensão natural desse trabalho consiste na definição das estratégias de transformação de partes da notação Z que não foram contempladas nesse trabalho, tais como as descrições axiomáticas, as versões genéricas dos parágrafos, os conjuntos infinitos etc. Além disso, seria interessante conduzir um estudo formal acerca da correção das transformações propostas, e expandir a ferramenta desenvolvida de forma a contemplar partes da metodologia não implementadas e possibilitar a integração com outras ferramentas que visam ao desenvolvimento formal e geração de código.

Durante o desenvolvimento deste trabalho, percebemos que seria possível produzir código mais bem estruturado se nos restringíssemos a uma classe bastante restrita de especificações. Seria interessante analisar como integrar a metodologia aqui proposta à metodologia restrita. Outro ponto de extensão possível deste trabalho seria a utilização de aspectos para promover a separação entre as verificações de condições da especificação e o código funcional da aplicação.

Apêndice A

Linguagem lógica

As restrições sobre variáveis são definidas através de fórmulas da linguagem lógica de Z compostas por elementos das outras linguagens. Para fins de exposição, essa linguagem lógica será dividida em lógica proposicional e lógica de predicados, uma extensão da lógica proposicional que inclui predicados e quantificadores.

A.1 Lógica Proposicional

A lógica proposicional permite expressar proposições complexas a partir de outras mais simples. Nela, as proposições mais simples são as atômicas, isto é, que são indivisíveis, e são ou verdadeiras ou falsas.

Por exemplo, a proposição “João ama Maria” pode ser verdadeira ou falsa, e nessa linguagem ela é indivisível. Para construir proposições mais complexas, podemos utilizar cinco conectivos lógicos, um unário: negação ($\neg$), e quatro binários: conjunção ($\wedge$), disjunção ($\vee$), implicação ($\Rightarrow$) e equivalência ($\Leftrightarrow$). Por exemplo, podemos desejar afirmar que “João ama Maria” é falso, para tanto podemos escrever “ $\neg$ João ama Maria”. Podemos ainda querer expressar o fato de que o sentimento é mútuo, isso é possível através da fórmula “João ama Maria $\wedge$ Maria ama João”.

A interpretação de uma proposição complexa é dada em termos de suas subproposições. Por exemplo, “João ama Maria $\wedge$ Maria ama João” é verdade se e somente se “João ama Maria” é verdade e “Maria ama João” é verdade. A interpretação de cada um dos conectivos com relação à interpretação de suas subproposições é dada na Tabela A.1.

Em marcação L^AT_EX os conectivos são escritos da seguinte forma:

- conjunção ($\wedge$): `\land`;
- disjunção ($\vee$): `\lor`;

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

Tabela A.1: Tabela verdade para os conectivos lógicos: negação, conjunção, disjunção, implicação e equivalência

- negação ($\neg$): `\lnot`;
- implicação ($\Rightarrow$): `\implies`;
- equivalência ($\Leftrightarrow$): `\iff`.

O conectivo e ($\wedge$) pode também ser escrito como ponto-e-vírgula ($;$) ou como uma quebra de linha.

A.2 Lógica de Predicados

A lógica de predicados permite expressar afirmações mais complexas que aquelas expressas pela lógica proposicional. Nessa extensão, a sentença “João ama Maria” não é necessariamente indivisível como no caso anterior. Podemos subdividir essa sentença em uma relação “ama” aplicada a dois nomes “João” e “Maria”. Note que nenhum dos componentes da sentença tem valor lógico (verdadeiro ou falso), a aplicação da relação “ama” aos dois nomes, por outro lado, o tem. Assim, podemos construir sentenças a partir de elementos ainda mais simples, através do uso de relações que podem tomar um número finito de parâmetros. Ao aplicar uma relação $\mathcal{R}$ que toma n parâmetros $(p_1, \dots, p_n)$ a n valores $(v_1, \dots, v_n)$, substituímos os parâmetros pelos valores correspondentes. O número de parâmetros que uma relação toma é denominado aridade da relação, por exemplo a relação de igualdade possui aridade 2.

Formalmente, uma relação $\mathcal{R}$ de aridade n é um conjunto de tuplas de tamanho n e o resultado de sua aplicação a uma tupla $(e_1, \dots, e_n)$ é dado pela seguinte condição:

$$\mathcal{R}(e_1, \dots, e_n) = \begin{cases} \text{true} & \text{se } (e_1, \dots, e_n) \in \mathcal{R}; \\ \text{false} & \text{se } (e_1, \dots, e_n) \notin \mathcal{R}; \end{cases} \quad (\text{A.1})$$

Na representação gráfica da aplicação de uma relação, o símbolo relacional pode aparecer antes dos parâmetros, após eles ou entre eles. Nesses casos, a relação é dita, respectivamente, prefixa, posfixa e infix. A relação de divisibilidade é infix de aridade dois (binária) representada, a seguir, por $(- \mid -)$, e pode ser caracterizada como o conjunto dos pares ordenados x e y , tais que x é inteiro, y é inteiro diferente de zero e o resto da divisão de x por y é igual a zero. Em notação matemática, essa definição é representada assim:

$$(- \mid -) = \{x : \mathbb{Z}; y : \mathbb{Z}^+ \mid x \bmod y = 0 \bullet (x, y)\} \quad (\text{A.2})$$

Note que os parâmetros de uma relação não precisam ser elementos simples. Podemos, por exemplo, aplicar relações a resultados de aplicação de uma função. Tomemos o exemplo da relação de divisibilidade, ela pode ser aplicada à seguinte tupla $(f(4), 4)$, onde $f(x) = x^2$.

Nessa linguagem as sentenças mais simples que possuem valor lógico são as proposições indivisíveis da lógica proposicional e a aplicação de uma relação. Esses elementos são denominados predicados e podem formar predicados mais complexos através dos mesmos conectivos da lógica

proposicional.

Exemplo A.2.1

Suponha que desejamos especificar uma máquina de café que registra o número de bebidas disponíveis para serem vendidas, e que esse número pode ser obtido através da função $numero(n)$, onde n representa o número da bebida (no caso dessa máquina, café seria representado pelo número 1). A máquina só pode vender tantos cafés quanto houver disponibilidade, por isso o valor de $numero(1)$ nunca pode ser negativo. Podemos especificar essa restrição usando a relação “maior ou igual a” ($\geq$) da seguinte forma:

$$numero(1) \geq 0 \quad (\text{A.3})$$

Posteriormente, a máquina é modificada para servir chá (número 2), além do café. A mesma restrição se aplica a essa bebida, e a restrição conjunta pode ser especificada assim:

$$numero(1) \geq 0 \wedge numero(2) \geq 0 \quad (\text{A.4})$$

Suponha que em algum momento essa máquina seja modificada para vender n produtos. Nesse caso, novamente a restrição se aplica a cada produto. Como poderíamos escrever a restrição associada a esses produtos? Uma forma possível seria afirmar que para todo produto identificado pelos números de 1 a n , a função $numero()$ aplicada a ele deve ser maior ou igual a zero. Porém, a linguagem apresentada aqui não nos permite fazer essa afirmação. A seguir, apresentamos novos elementos que nos permitem expressar essa restrição.

Outra diferença entre a lógica de predicados e a lógica proposicional é a existência de quantificadores, isto é, conectivos que nos permitem fazer afirmações do tipo “existe” ($\exists$) e “para todo” ($\forall$). Esses novos conectivos apresentam uma estrutura diferente dos anteriores, ela está representada abaixo:

$$\mathcal{Q}x : \mathcal{D} \mid p \bullet q \quad (\text{A.5})$$

Onde $\mathcal{Q}$ é o quantificador, x a variável quantificada, $\mathcal{D}$ o conjunto a que x pertence, p o predicado que restringe o domínio de x e q o predicado propriamente dito. Note que não é necessário quantificar sobre apenas uma variável, podemos fazê-lo sobre várias variáveis do mesmo conjunto ou de conjuntos diferentes, por exemplo:

$$\forall x, y : \mathbb{N} ; z : \mathbb{Z} \mid p \bullet q \quad (\text{A.6})$$

Com esses novos elementos podemos expressar fatos que não eram expressáveis apenas com a lógica proposicional ou com as relações. Por exemplo, podemos expressar a restrição deixada em aberto no Exemplo A.2.1, da seguinte forma:

$$\underbrace{\forall i : \mathbb{N} \mid i \geq 1 \wedge i \leq n}_{\text{domínio}} \bullet numero(i) \geq 0 \quad (\text{A.7})$$

Isto é, para todo número natural i , tal que i é maior ou igual a 1 e menor ou igual a n , vale que $\text{numero}(i)$ é maior ou igual a 0.

Como visto acima, um quantificador é composto por um domínio¹, isto é, um conjunto restrito ou não e outro predicado que será avaliado contra os valores do domínio. O quantificador “para todo” ($\forall$) afirma que para todo elemento do domínio, para o qual o predicado com a variável quantificada substituída por ele é verdade. O quantificador “existe” ($\exists$) afirma que existe ao menos um elemento do domínio, tal que o predicado com a variável quantificada substituída por ele é verdade.

Temos ainda um quantificador auxiliar “existe exatamente um” ($\exists_1$) que afirma que existe apenas um elemento do domínio, tal que o predicado com a variável quantificada substituída por ele é verdade.

Note que a expressão $\text{numero}(i) \geq 0$ não tem valor de verdade, pois não sabemos o valor de x para podermos calcular a função $\text{numero}()$ nele e comparar o resultado a 0. Nesse caso, dizemos que x é uma variável livre. Ao quantificarmos essa expressão sobre a variável i , ela deixa de ser livre pois ela pode ser qualquer uma do domínio definido pelo quantificador, e portanto conhecemos passando a ser denominada variável ligada e a ter valor de verdade.

¹No padrão internacional essa estrutura, determinada por um conjunto de declarações de variável e um predicado, é denominado Esquema (*Schema*) e constitui um dos principais elementos da notação. Esse elemento é abordado no Capítulo 4

Apêndice B

Linguagem de conjuntos

Quando falamos de quantificação, consideramos o conjunto dos elementos que podem ser atribuídos à variável quantificada. Quando estendemos a máquina de café para conter n produtos, consideramos o conjunto dos números naturais e o restringimos aos números maiores ou iguais a 1 e menores ou iguais a n . Quando definimos relação, falamos em conjunto de pares ordenados. Como pode-se perceber, o conceito de conjunto é recorrente tanto em definições matemáticas, quanto em especificações de sistemas. Por esse motivo, é necessário que a linguagem Z dê conta de expressar a manipulação de conjuntos.

A forma mais simples de se usar um conjunto em Z é através dos conjuntos definidos pela própria linguagem. A linguagem Z foi descrita em um padrão internacional [24]. Nesse padrão, ela é dividida em três partes: núcleo, prelúdio e arcabouço matemático. A linguagem lógica (proposicional e de predicados) vista anteriormente está definida no núcleo, a linguagem de conjuntos, por sua vez, está espalhada pelas três partes. No núcleo, define-se formas de construir novos conjuntos (conjunto das partes, extensão e compreensão de conjuntos) e a relação de pertinência. No prelúdio, é definido o conjunto dos números naturais e no arcabouço matemático, são definidos operadores de conjuntos (união, intersecção, diferença etc), relações que envolvem conjuntos (não pertinência, subconjunto e subconjunto próprio), além do conjunto vazio, conjunto dos subconjuntos não vazios etc).

Para facilitar a exposição, todos os aspectos relevantes para a expressão de fatos envolvendo conjuntos serão tratados uniformemente, sem fazer referência à estruturação imposta pelo padrão internacional.

Outro ponto particular à linguagem Z que vale ser mencionado é o fato de que Z utiliza um sistema de tipos baseado na noção de conjuntos maximais. Um conjunto maximal é aquele tal que não existe nenhum outro conjunto diferente dele que o contenha, dessa forma cada elemento da linguagem está relacionado a exatamente um tipo. Por exemplo, o conjunto dos números naturais é definido em termos de um tipo denominado *arithmos* ($\mathbb{A}$) declarado no prelúdio. Portanto, o conjunto dos números naturais não é um maximal, e conseqüentemente não é um tipo.

Devido a essa natureza limítrofe dos tipos, eles são os conjuntos mais básicos existentes numa especificação.

B.1 Tipos

Existem duas formas de introduzir tipos novos em $\mathbb{Z}$, a primeira através da declaração de um tipo dado e a segunda através da declaração de tipos livres.

A declaração de tipos dados é feita através de colchetes contendo o nomes dos tipos a serem declarados separados por vírgula. Esse tipo de declaração introduz tipo, porém não explicita nem seus elementos, nem sua estrutura interna. Por exemplo, suponha que, numa especificação de um sistema de alocação de sala de aula, alunos e sala são elementos básico. Podemos declarar os tipos *Aluno* e *Sala* da seguinte forma:

$$[Aluno, Sala] \quad (B.1)$$

A declaração de tipos livres, por outro lado, introduz explicitamente os elementos desse tipo através de uma estrutura semelhante às enumerações existentes em muitas linguagens de programação. Podemos, por exemplo, declarar o tipo *Cores* que contém as cores primárias da seguinte forma:

$$Cores ::= vermelho \mid azul \mid amarelo \quad (B.2)$$

No entanto, a declaração de tipos livres não se limite a casos simples como o exemplo acima, ela pode ser usada para declarar tipos associados com estruturas de dados, tais como listas e árvores. Por exemplo, podemos declarar o tipo das árvores binárias como a seguir [62]:

$$\acute{A}rvore ::= folha \langle \langle \mathbb{Z} \rangle \rangle \mid ramo \langle \langle \acute{A}rvore \times \acute{A}rvore \rangle \rangle \quad (B.3)$$

Note que essa declaração lembra bastante a especificação de gramáticas livres de contexto através da forma normal de Backus-Naur e introduz uma definição recursiva. Os elementos *folha* e *ramo* da definição acima são funções injetoras de, respectivamente, $\mathbb{Z}$ e $\acute{A}rvore \times \acute{A}rvore$ para $\acute{A}rvore$. Nesse exemplo, em particular, define-se uma árvore como uma folha ou um ramo contendo duas árvores.

A partir de um tipo livre como $\acute{A}rvore$, podemos definir uma função que toma elemento desse tipo especificando o comportamento da função para cada parte da definição do tipo. Por exemplo, uma função que produz uma seqüência de inteiros percorrendo a árvore em profundidade da esquerda para a direita pode ser definida como a função que toma uma árvore A e produz, se $A = folha \ n$, uma seqüência contendo o número n , e, se $A = ramo(B, C)$, uma nova seqüência concatenando as seqüências obtidas pela aplicação da função recursivamente às árvores B e C .

Essa função pode ser especificada em $\mathbb{Z}$ da seguinte forma:

$$\left| \begin{array}{l} \text{percorre} : \acute{A}rvore \rightarrow \text{seq } \mathbb{N} \\ \hline \forall n : \mathbb{N} \bullet \text{percorre } folha \ n = \langle n \rangle \\ \forall A_{esq}, A_{dir} : \acute{A}rvore \bullet \text{percorre } ramo(A_{esq}, A_{dir}) = \text{percorre } A_{esq} \frown \text{percorre } A_{dir} \end{array} \right|$$

A estrutura usada para especificar essa função é denominada descrição axiomática e introduz nome globais, nesse caso *percorre* e restrições sobre seus valores. Nesse exemplo, $\langle n \rangle$ representa a

seqüência formada pelo número n e $S_1 \cap S_2$ a concatenação das seqüências S_1 e S_2 .

A partir dos tipos introduzidos através da declaração de tipos dados ou livres é possível construir novos tipos usando os operadores $\mathbb{P}$ e $\times$, chamados, respectivamente, conjunto das partes e produto cartesiano. O primeiro toma um conjunto C e gera o conjunto que contém todos os subconjuntos de C . O segundo toma dos conjuntos C_1 e C_2 e produz o conjunto dos pares ordenados cujos primeiro e segundo termos pertencem, respectivamente, a C_1 e C_2 .

Quando aplicados a tipos, esses operadores produzem tipos. Por exemplo, seja T um tipo, $\mathbb{P} T$ produz um conjunto maximal que contém os subconjuntos de T , isto é, não existe nenhum conjunto maior que $\mathbb{P} T$ que contenha um subconjunto de T . Esse fato é facilmente demonstrável.

Teorema B.3. *Seja T um conjunto maximal, então $\mathbb{P} T$ também é maximal.*

Prova. Suponha que exista um conjunto K de subconjuntos de T maior que $\mathbb{P} T$, por definição $\mathbb{P} T$ contém todos os subconjuntos de T , então existe algum elemento de K que não é subconjunto de T . E portanto K não é composto por subconjuntos de T . ■

Note, porém, que se tomarmos um subconjunto T_1 de T , $\mathbb{P} T_1$ não é maximal, visto que todos seus elementos pertencem a $\mathbb{P} T$.

B.2 Conjuntos em geral

A declaração de tipos dados e livres introduz conjuntos maximais, porém podemos especificar conjuntos não necessariamente maximais. Para tanto, existem duas formas básicas de declarar conjuntos: extensão e compreensão.

A **extensão** de um conjunto é semelhante ao uso básico da declaração de tipos livres, na medida em que devemos enumerar os elementos pertencentes ao conjunto. Porém, ela possui uma sintaxe diferente e não produz tipos. O conjunto das cores primárias, por exemplo, pode ser definido da seguinte forma:

$$Cores == \underbrace{\{verde, azul, amarelo\}}_{\text{extensão}} \quad (\text{B.4})$$

Note que os nomes *verde*, *azul* e *vermelho* deve ter sido previamente definidos, visto que essa construção, ao contrário da declaração de tipos livres, não os introduz. A ligação da extensão a um nome por $==$ é denominado abreviação e introduz um novo nome, no caso do exemplo, introduz o nome *Cores*. A abreviação introduz um nome global (lado esquerdo) e associa a ele o elemento à direita. Ela será usada em outros contextos também.

A **compreensão** toma um domínio, assim como a quantificação, e uma expressão que contém (ou não) as variáveis declaradas no domínio. Ela declara um conjunto que contém os valores da expressão calculada sobre os valores do domínio. Por exemplo, podemos declarar o conjunto de pares ordenados cujos primeiro e segundo membros são, respectivamente, par e ímpar, da seguinte forma.

$$Par\acute{I}mpar == \underbrace{\{x, y : \mathbb{Z} \mid x \bmod 2 = 0 \wedge y \bmod 2 = 1 \bullet (x, y)\}}_{\text{domínio}} \quad (\text{B.5})$$

Tomemos, por exemplo, x valendo 10 e y valendo 21, eles pertencem ao domínio pois 10 é par e 21 é ímpar. A expressão (x, y) calculada sobre esses valores produz o par $(10, 21)$. Portanto, o par $(10, 21)$ pertence ao conjunto *ParÍmpar*.

Note que também na compreensão usamos a abreviação para associar o conjunto descrito a um nome.

Uma restrição de suma importância para a teoria dos conjuntos é que todos os elementos de um determinado conjunto tenham o mesmo tipo. Por exemplo, um conjunto não pode conter o elemento *azul* do tipo livre *Cores* definido na Seção B.1 e o número 2, pois o número 2 possui tipo $\mathbb{A}$ (definido no prelúdio) que é diferente do tipo *Cores*. De fato, todo tipo, devido ao caráter maximal, é único, isto é, não existe um elemento que tenha dois tipos. Assim, um conjunto não pode ter elemento dos tipos T e $\mathbb{P} T$, pois esses são tipos (ver **Teorema B.3**).

Assim como para os tipos, os operadores $\mathbb{P}$ e $\times$ se aplicam para conjuntos gerais.

Exemplo B.2.1

Sejam os conjuntos $A == \{1, 2, 3\}$ e $B == \{x : \mathbb{N}_1 \mid x < 4 \bullet -x\}$, onde $\mathbb{N}_1$ representa os números naturais estritamente positivos. O conjunto representado por $\mathbb{P} A$ é o seguinte:

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\} \quad (\text{B.6})$$

O conjunto representado por $A \times B$ é o seguinte:

$$\{(1, -1), (1, -2), (1, -3), (2, -1), (2, -2), (2, -3), (3, -1), (3, -2), (3, -3), \} \quad (\text{B.7})$$

Alguns operadores de conjuntos derivados de $\mathbb{P}$ são: subconjuntos não vazios ($\mathbb{P}_1$), subconjuntos finitos ($\mathbb{F}$) e subconjuntos finitos não vazios ($\mathbb{F}_1$).

Podemos também encadear diversos produtos cartesianos produzindo conjuntos de tuplas de tamanho correspondente. Por exemplo, o conjunto $A \times B \times C$ conteria triplas (tuplas de tamanho 3). Dada uma tupla t de tamanho n , podemos acessar o valor na i -ésima posição da tupla da seguinte forma: $t.i$, onde $1 \leq i \leq n$.

Note que o conjunto descrito pelo produto cartesiano produz um conjunto de corresponde a uma relação que é verdade para todos valores dos parâmetros.

Podemos descrever uma relação mais restritiva através do uso do produto cartesiano e da compreensão de conjuntos. Por exemplo, a relação que associa cada número estritamente positivo ao seu inverso aditivo da seguinte forma:

$$\{t : \mathbb{Z} \times \mathbb{Z} \mid t.1 > 0 \wedge t.1 = -t.2 \bullet t\} \quad (\text{B.8})$$

Nesse caso, a expressão da compreensão de conjuntos é exatamente a variável do domínio. Por isso,

podemos omitir a última parte da compreensão.

$$\{t : \mathbb{Z} \times \mathbb{Z} \mid t.1 > 0 \wedge t.1 = -t.2\} \quad (\text{B.9})$$

Além desse dois operadores, existem mais seis operadores de conjuntos. Porém, enquanto os operadores conjunto das partes ($\mathbb{P}$) e produto cartesiano ($\times$) preservam o caráter maximal quando aplicados a conjuntos maximais, esses outros só podem ser aplicados a conjuntos de mesmo tipo, e, como dois conjuntos maximais não podem ter o mesmo tipo, não é possível aplicá-los a conjuntos maximais.

Dos seis operadores, quatro são binários e tomam dois conjuntos de mesmo tipo como parâmetros. Os dois último, são generalizações de dois operadores binários para um número variável de parâmetros. Os operadores binários são a união, a intersecção, a diferença e a diferença simétrica e os outros dois são versões generalizadas dos operadores de união e intersecção. São definidas também algumas relações, pertinência, não pertinência, subconjunto e subconjunto próprio.

O operador de **união** ($\cup$) representa um conjunto que contém todos os elementos dos dois conjuntos que toma como parâmetros, o operador **intersecção** ($\cap$) representa um conjunto que contém apenas os elementos que estão nos dois conjuntos tomados como parâmetro. O operador de **diferença** ($\setminus$) representa um conjunto que possui todos os elementos do primeiro parâmetro que não estão no segundo, e a **diferença simétrica** ($\ominus$) representa o conjunto cujos elementos estão apenas em um dos dois parâmetros. Esses operadores são definidos, respectivamente, pelas Equações B.10, B.11, B.12 e B.13 [24].

$$\forall a, b : \mathbb{P} X \bullet a \cup b = \{x : X \mid x \in a \vee x \in b\} \quad (\text{B.10})$$

$$\forall a, b : \mathbb{P} X \bullet a \cap b = \{x : X \mid x \in a \wedge x \in b\} \quad (\text{B.11})$$

$$\forall a, b : \mathbb{P} X \bullet a \setminus b = \{x : X \mid x \in a \wedge x \notin b\} \quad (\text{B.12})$$

$$\forall a, b : \mathbb{P} X \bullet a \ominus b = \{x : X \mid \neg (x \in a \Leftrightarrow x \in b)\} \quad (\text{B.13})$$

Os operadores generalizados tomam um conjunto de conjuntos e aplicam os operadores binários correspondentes aos elementos. Por exemplo,

$$\bigcup \{\{1\}, \{2, 3\}, \{4\}, \{5\}\} = \{1\} \cup \{2, 3\} \cup \{4\} \cup \{5\} = \{1, 2, 3, 4, 5\} \quad (\text{B.14})$$

A **união generalizada** é denotada por $\bigcup$ e a **intersecção generalizada** por $\bigcap$, elas são definidas, respectivamente, pelas Equações B.15 e B.16 [24].

$$\forall A : \mathbb{P} \mathbb{P} X \bullet \bigcup A = \{x : X \mid \exists a : A \bullet x \in a\} \quad (\text{B.15})$$

$$\forall A : \mathbb{P} \mathbb{P} X \bullet \bigcap A = \{x : X \mid \forall a : A \bullet x \in a\} \quad (\text{B.16})$$

As relações de **pertinência** ($\in$) e **não pertinência** ($\notin$) são complementares, a primeira afirma que um dado elemento pertence a um conjunto, e a segunda afirmam o contrário. As relações

subconjunto ($\subseteq$) afirma que um conjunto é subconjunto de outro ou igual a ele, e a relação **subconjunto próprio** ($\subset$) afirma que um conjunto é subconjunto de outro e diferente dele. As definições formais dessas duas últimas relações são dadas pelas Equações B.17 e B.18 [24].

$$\forall a, b : \mathbb{P} X \bullet a \subseteq b \Leftrightarrow (\forall x : a \bullet x \in b) \quad (\text{B.17})$$

$$\forall a, b : \mathbb{P} X \bullet a \subset b \Leftrightarrow a \subseteq b \wedge a \neq b \quad (\text{B.18})$$

Um conjunto particular é o **conjunto vazio** ($\emptyset$). Note que embora não possua elementos, o conjunto vazio é subconjunto de algum outro, e portanto é parametrizado por algum tipo. Isso é necessário, para que se possa aplicar operadores que impõem que os tipos dos parâmetros sejam os mesmos. Dessa forma, definimos o conjunto vazio de um conjunto X como na Equação B.19, onde a abreviação está parametrizada pelo conjunto X .

$$\emptyset[X] == \{x : X \mid \text{false}\} \quad (\text{B.19})$$

B.3 Domínios

Os domínios que são usados na construção de fórmulas quantificadas da lógica de predicados e na compreensão de conjuntos possuem uma estrutura particular que merece ser melhor descrita, pois os domínios permeiam todos os níveis de especificação em Z.

Eles são compostos de duas partes, declaração de variáveis e um predicado opcional. A declaração de variáveis a, b, c relaciona um conjunto de nomes a um conjunto X do qual ele tomam valores, isto é, os nomes das variáveis declaradas podem ser associados a elemento de X . Essa declaração tem a seguinte forma:

$$a, b, c : X \quad (\text{B.20})$$

O símbolo de dois pontos ($:$) usados na declaração fazem o papel da relação de pertinência, isto é, indicam que as variáveis pertencem ao conjunto mencionado à direita.

Pode-se declarar várias variáveis, cada uma pertencentes a um conjunto diferente, separando-as por ponto-e-vírgula, como a seguir:

$$x : \mathbb{N} ; y, z : \mathbb{Z} \quad (\text{B.21})$$

O predicado, em geral, envolve as variáveis declaradas, restringindo ainda mais os valores que elas podem assumir. Por exemplo, aproveitando a declaração anterior (Equação B.21), podemos adicionar um predicado que restrinja a variável x a tomar apenas valores pares e as outras a tomarem apenas valores ímpares.

$$\overbrace{x : \mathbb{N} ; y, z : \mathbb{Z} \mid x \bmod 2 = 0 \wedge y \bmod 2 = 1 \wedge z \bmod 2 = 1}^{\text{domínio}} \quad (\text{B.22})$$

$\underbrace{x : \mathbb{N} ; y, z : \mathbb{Z}}_{\text{declaração}}$
 $\underbrace{x \bmod 2 = 0 \wedge y \bmod 2 = 1 \wedge z \bmod 2 = 1}_{\text{predicado}}$

Apêndice C

Linguagem Matemática

Devido ao caráter fundamental da linguagem de conjuntos na notação Z, ela foi apresentada paralelamente à linguagem lógica. A partir das duas linguagens já expostas, é possível definir um conjunto de objetos, relações e operadores muito usados em especificações, tais como relações, funções, números, seqüências e suas relações e operadores tradicionais. Esses elementos são definidos, em sua maior parte, no arcabouço matemático descrito no padrão internacional [24]. A seguir, expomos, de maneira breve, cada uma desses elementos.

C.1 Relações

As relações já foram mencionadas anteriormente como forma de construir predicados. Sua definição também já foi exposta, porém vale retomá-la.

Definição C.1.1

Uma **relação** $\mathcal{R}$ que toma n elementos, cada um deles pertencente, respectivamente, aos conjuntos $\mathcal{C}_1, \dots, \mathcal{C}_n$, é um conjunto de tuplas de tamanho n , isto é:

$$\mathcal{R} \subseteq \mathcal{C}_1 \times \dots \times \mathcal{C}_n \quad (\text{C.1})$$

O resultado de sua **aplicação** a uma tupla $(e_1, \dots, e_n) \in \mathcal{C}_1 \times \dots \times \mathcal{C}_n$ é dado pela seguinte condição:

$$\mathcal{R}(e_1, \dots, e_n) = \begin{cases} \text{true} & \text{se } (e_1, \dots, e_n) \in \mathcal{R}; \\ \text{false} & \text{se } (e_1, \dots, e_n) \notin \mathcal{R}; \end{cases} \quad (\text{C.2})$$

■

Como uma relação nada mais é que um conjunto, todos as relações e operadores de conjuntos se aplicam a ela. Porém, devido à maior complexidade de uma relação comparada a um conjunto simples, outros operadores podem ser definidos.

A rigor, relações são sempre binárias e o conjunto de todas as relações de X para Y é denotado pela Equação C.3. Lembre-se que uma relação é um subconjunto de $X \times Y$, e portanto pertence

ao conjunto das partes de $X \times Y$, isto é, $\mathbb{P} X \times Y$. Esse fato é ilustrado pela Equação C.4.

$$X \leftrightarrow Y \tag{C.3}$$

$$\mathcal{R} : X \leftrightarrow Y \Leftrightarrow \mathcal{R} : \mathbb{P}(X \times Y) \tag{C.4}$$

No caso das relações binárias, denominamos X o conjunto de origem da relação e Y o conjunto de chegada. Porém, podemos definir relações não binárias em termos das binárias escolhendo apropriadamente os conjuntos de chegada e origem, que podem ser eles mesmos relações (Equação C.6) ou produtos cartesianos (Equação C.5) de conjuntos.

$$(X \times Y) \leftrightarrow Z \tag{C.5}$$

$$(X \leftrightarrow Y) \leftrightarrow Z \tag{C.6}$$

Devido a essa natureza intrinsecamente binária da relações, os pares ordenados possuem um papel bastante importante. Por isso, definem-se formas auxiliares de manipular esses objetos, os operadores de projeção, *first* e *second*, que devolvem, respectivamente, o primeiro e o segundo componente do par. Outra definição auxiliar motivada pela importância dos pares ordenados é o operador de mapeamento ($\mapsto$) que toma dois valores e produz um par ordenado. Note porém que esses três operadores são, na verdade, nomes diferentes para operadores já existentes (Equações C.7, C.8, C.9).

$$\text{first } p = p.1 \tag{C.7}$$

$$\text{second } p = p.2 \tag{C.8}$$

$$x \mapsto y = (x, y) \tag{C.9}$$

Os operadores novos definidos para relações se dividem em dois tipos, operadores que necessariamente produzem relações e operadores que não o fazem. Os operadores que não produzem obrigatoriamente relações são: domínio (*dom*), imagem (*ran*) e imagem relacional ($_ \llbracket _ \rrbracket$). O **domínio** (Equação C.10) e a **imagem** (Equação C.11) de uma relação são os conjuntos dos valores que são, respectivamente, primeiro e segundo componentes dos pares ordenados contidos na relação. Isto é, se $\mathcal{R}$ é uma relação de X em Y , então:

$$\text{dom } \mathcal{R} = \{p : \mathcal{R} \bullet \text{first } p\} \tag{C.10}$$

$$\text{ran } \mathcal{R} = \{p : \mathcal{R} \bullet \text{second } p\} \tag{C.11}$$

A **imagem relacional** de um subconjunto x de X por uma relação de X em Y é um subconjunto y de Y , tal que todo elemento de y está relacionado a um elemento de x . A definição formal de imagem relacional é dada pela Equação C.12, adaptada de [24].

$$\forall \mathcal{R} : X \leftrightarrow Y; \ c : \mathbb{P} X \bullet \mathcal{R} \llbracket c \rrbracket = \{p : \mathcal{R} \mid \text{first } p \in c \bullet \text{second } p\} \tag{C.12}$$

Os outros operadores são: composição, restrição de domínio, restrição de imagem, subtração de domínio, subtração de imagem, inversão, sobrescrita, fecho transitivo e fecho transitivo reflexivo.

A **composição** de duas relações $\mathcal{R} : X \leftrightarrow Y$ e $\mathcal{S} : Y \leftrightarrow Z$, cujos conjuntos de chegada de $\mathcal{R}$ e de origem de $\mathcal{S}$ devem ser os mesmos, é uma relação de X em Z , denotada por $\mathcal{R} \circ \mathcal{S}$ que relaciona os elementos x de X e z de Z , tais que existe um elemento y de Y , tal que $\mathcal{R}$ relaciona x e y e $\mathcal{S}$ relaciona y e z . Outra forma de denotar a composição de relações é através do símbolo 'o'.

Uma definição alternativa, porém equivalente, da composição de relações é dada pela Equação C.13, adaptada do padrão internacional [24].

$$\forall \mathcal{R} : X \leftrightarrow Y; \mathcal{S} : Y \leftrightarrow Z \bullet \mathcal{R} \circ \mathcal{S} = \{p : \mathcal{R}; q : \mathcal{S} \mid p.2 = q.1 \bullet p.1 \mapsto q.2\} \quad (\text{C.13})$$

A **restrição de domínio** ($\triangleleft$) de uma relação $\mathcal{R} : X \leftrightarrow Y$ por um conjunto $c : \mathbb{P} X$ é uma relação composta pelos pares de $\mathcal{R}$, tais que o primeiro componente pertence a c . A **restrição de imagem** ($\triangleright$) de uma relação $\mathcal{R} : X \leftrightarrow Y$ por um conjunto $d : \mathbb{P} Y$ é uma relação composta pelos pares de $\mathcal{R}$, tais que o segundo componente pertence a d . A definição formal dos dois operadores é dada pelas Equações C.14 e C.15.

$$\forall \mathcal{R} : X \leftrightarrow Y; c : \mathbb{P} X \bullet c \triangleleft \mathcal{R} = \{p : \mathcal{R} \mid \text{first } p \in c\} \quad (\text{C.14})$$

$$\forall \mathcal{R} : X \leftrightarrow Y; d : \mathbb{P} Y \bullet \mathcal{R} \triangleright d = \{p : \mathcal{R} \mid \text{second } p \in d\} \quad (\text{C.15})$$

A **subtração de domínio** e a **subtração de imagem** funcionam de forma inversa às versões restritivas. Ao invés de conter apenas os pares cujos primeiro ou segundo elementos pertencem a um determinado conjunto, contêm os pares cujos primeiro ou segundo elementos não pertencem ao conjunto. Formalmente, elas são definidas, respectivamente, pelas Equações C.16 e C.17 [24].

$$\forall \mathcal{R} : X \leftrightarrow Y; c : \mathbb{P} X \bullet c \triangleleft \mathcal{R} = \{p : \mathcal{R} \mid \text{first } p \notin c\} \quad (\text{C.16})$$

$$\forall \mathcal{R} : X \leftrightarrow Y; d : \mathbb{P} Y \bullet \mathcal{R} \triangleright d = \{p : \mathcal{R} \mid \text{second } p \notin d\} \quad (\text{C.17})$$

A **inversão** de uma relação $\mathcal{R} : X \leftrightarrow Y$ é uma relação $\mathcal{R}^\sim : Y \leftrightarrow X$, tal que para todo par (p_1, p_2) pertencente a $\mathcal{R}$, o par (p_2, p_1) pertence a $\mathcal{R}^\sim$. A inversão é definida pela Equação C.18 [24].

$$\forall \mathcal{R} : X \leftrightarrow Y \bullet \mathcal{R}^\sim = \{p : \mathcal{R} \bullet \text{second } p \mapsto \text{first } p\} \quad (\text{C.18})$$

A **sobrescrita** consiste na junção de duas relações de forma que a segunda relação prevaleça, isto é, a nova relação é basicamente a segunda mais os pares da primeira, exceto aqueles cujo primeiro componente pertence ao domínio da segunda. Podemos definir a sobrescrita de duas relações através da Equação C.19 [24].

$$\forall \mathcal{R}, \mathcal{S} : X \leftrightarrow Y \bullet \mathcal{R} \oplus \mathcal{S} = ((\text{dom } \mathcal{S}) \triangleleft \mathcal{R}) \cup \mathcal{S} \quad (\text{C.19})$$

O fecho de uma relação consiste no menor conjunto que contém a própria relação e é fechado sobre alguma propriedade. Uma relação $\mathcal{R} : X \leftrightarrow X$ é dita transitiva se para todo $x, y, z : X$, se $\mathcal{R}(x, y)$ e $\mathcal{R}(y, z)$, então $\mathcal{R}(x, z)$. Dessa forma, o **fecho transitivo** de uma relação $\mathcal{R}$ é $\mathcal{R}^+$ e denota o menor conjunto que contém $\mathcal{R}$ e é uma relação transitiva. Essa relação é definida como a intersecção de todas as relações que contém $\mathcal{R}$ e são transitivas. Essa definição garante a restrição de minimalidade da relação através do operador de intersecção. Formalmente, podemos definir essa relação através da Equação C.20 [24].

$$\forall \mathcal{R} : X \leftrightarrow X \bullet \mathcal{R}^+ = \bigcap \{ \mathcal{S} : X \leftrightarrow X \mid \mathcal{R} \subseteq \mathcal{S} \wedge \mathcal{R} \circ \mathcal{S} \subseteq \mathcal{S} \} \quad (\text{C.20})$$

Uma definição alternativa é descrita pela Equação C.21 [62] e pode se mostrar mais intuitiva.

$$\forall \mathcal{R} : X \leftrightarrow X \bullet \mathcal{R}^+ = \bigcup \{ n : \mathbb{N} \mid n \geq 1 \bullet \mathcal{R}^n \}, \text{ onde } \mathcal{R}^n \text{ é dado por C.22.} \quad (\text{C.21})$$

$$\mathcal{R}^n = \begin{cases} \mathcal{R} & \text{se } n = 1 \\ \mathcal{R} \circ \mathcal{R}^{n-1} & \text{se } n > 1 \end{cases} \quad (\text{C.22})$$

Uma relação é dita reflexiva se ela relaciona todo elemento a ele mesmo, e a menor relação reflexiva de um conjunto X é a relação identidade, denotada por $\text{id } X$. A identidade é dada por $\{x : X \bullet x \mapsto x\}$. O **fecho transitivo reflexivo** de $\mathcal{R}$, $\mathcal{R}^*$, é a menor relação que contém $\mathcal{R}$ e é reflexiva e transitiva, isto é, ele é a união entre o fecho transitivo da relação e a relação identidade. O fecho transitivo reflexivo é definido formalmente pela Equação C.23 [24].

$$\forall \mathcal{R} : X \leftrightarrow X \bullet \mathcal{R}^* = \mathcal{R}^+ \cup \text{id } X \quad (\text{C.23})$$

C.2 Funções

Funções são tipos particulares de relações. Uma relação $\mathcal{F} : X \leftrightarrow Y$ é uma função se e somente se ela associa cada elemento de X a exatamente um elemento de Y . O conjunto de todas as funções de X em Y é denotado por $X \rightarrow Y$. Formalmente o conjunto das funções é dado pela Equação C.24.

$$\{ \mathcal{F} : X \leftrightarrow Y \mid \forall x : X \bullet \exists_1 y : Y \bullet (x, y) \in \mathcal{F} \} \quad (\text{C.24})$$

Como funções são relações, os operadores definidos para essas valem também para aquelas. Embora úteis, funções são bastante restritivas. Em especial, na medida em que exigem que todos os elementos do conjunto de origem estejam associados a algum elemento do conjunto de chegada. É possível enfraquecer e reforçar as restrições sobre os conjuntos de origem e chegada, obtendo-se diferentes tipos de funções. Por exemplo, podemos enfraquecer a restrição de que todo elemento do conjunto de origem seja associado a exatamente um elemento do conjunto de chegada, para que cada um daqueles elemento seja associado a no máximo um elemento do conjunto de chegada, isto é, se um dado elemento do conjunto de origem for associado a dois elementos do conjunto de

chegada, eles são iguais. Ao conjunto de relações que respeitam essa nova restrição denominamos **funções parciais**. Esse conjunto é denotado por $X \rightarrowtail Y$ e definido pela equação C.25 [24].

$$\{f : X \leftrightarrow Y \mid \forall p, q : f \mid p.1 = q.1 \bullet p.2 = q.2\} \quad (\text{C.25})$$

Para marcar a oposição entre funções como definidas pela Equação C.24 e funções parciais, denominamos as primeiras de **funções totais**.

A restrição que define as funções parciais, pode ser aplicada também com relação ao conjunto de chegada, isto é, podemos exigir que cada elemento do conjunto de chegada seja associado a no máximo um elemento do conjunto de origem. Com essas duas restrições, obtemos o conjunto das **funções parciais injetoras**, denotado por $X \rightarrowtail Y$ e definido pela Equação C.26 [24].

$$\{f : X \leftrightarrow Y \mid \forall p, q : f \bullet p.1 = q.1 \Leftrightarrow p.2 = q.2\} \quad (\text{C.26})$$

Essa restrição que define as funções parciais injetoras também pode ser aplicada às funções totais, resultando nas **funções totais injetoras**, cujo conjunto é denotado por $X \rightarrow Y$ e definido pela Equação C.27 [24], isto é, como toda função total é também uma função parcial, as funções totais injetoras são todas as funções totais que são também funções parciais injetoras.

$$(X \rightarrowtail Y) \cap (X \rightarrow Y) \quad (\text{C.27})$$

Outro tipo de restrição que podemos impor é que todo elemento do conjunto da chegada seja associado a algum elemento do conjunto de origem. Essa restrição produz algo parecido com as funções totais, porém em ordem contrária, do conjunto de chegada para o conjunto de origem. As funções parciais que possuem essa propriedade são denominadas **funções parciais sobrejetoras** e as totais que a apresentam são ditas **funções totais sobrejetoras**. O conjunto das funções parciais sobrejetoras é denotado por $X \twoheadrightarrowtail Y$ e o das funções totais sobrejetoras por $X \twoheadrightarrow Y$. Eles são definidos, respectivamente, pelas Equações C.28 e C.29 [24].

$$\{f : X \rightarrowtail Y \mid \text{ran } f = Y\} \quad (\text{C.28})$$

$$(X \twoheadrightarrowtail Y) \cap (X \rightarrow Y) \quad (\text{C.29})$$

Outra classe relevante de funções é aquela composta pelas funções que são, ao mesmo tempo, injetoras e sobrejetoras. Essa classe é definida apenas para as funções totais, é denotada por $X \rightarrowtail Y$ e definida pela Equação C.30 [24].

$$(X \rightarrowtail Y) \cap (X \twoheadrightarrowtail Y) \quad (\text{C.30})$$

Uma última forma de restringir as funções é quanto ao número de pares ordenados contidos nelas. A princípio, não há limite para esse valor. Então, podemos definir a classe de **funções finitas** e **funções injetoras finitas**. Note que não se definir a classe de funções sobrejetoras

finitas, pois elas só existem no caso de o conjunto de chegada ser finito e, nesse caso, essa classe é a mesma das funções parciais sobrejetoras. As funções finitas são aquelas relações finitas ($\mathbb{F}(X \times Y)$) que são também funções parciais, sua classe é denotada por $X \twoheadrightarrow Y$ e definida pela equação C.31, sua contraparte injetora é denotada por $X \rightarrowtail Y$ e definida pela Equação C.32 [24].

$$(X \twoheadrightarrow Y) \cap \mathbb{F}(X \times Y) \quad (\text{C.31})$$

$$(X \rightarrowtail Y) \cap (X \twoheadrightarrow Y) \quad (\text{C.32})$$

Uma das principais formas de descrever um função é descrevendo o conjunto de pares ordenados que a define. Porém, a linguagem Z permite a definição de funções de uma forma alternativa, através da abstração lambda (λ). A abstração lambda é formada por um domínio e uma expressão a ser calculada sobre os elementos desse domínio. Por exemplo, podemos definir a função quadrado ($_{}^2$) como na Equação C.33

$$_{}^2 == \lambda \underbrace{x : \mathbb{Z}}_{\text{domínio}} \bullet x * x \quad (\text{C.33})$$

C.3 Números

Os números definidos pela notação Z são os naturais e os inteiros. Os naturais são definidos no prelúdio e os inteiros no arcabouço matemático. Ambos são definidos em termos do tipo dado *arithmos* ($\mathbb{A}$). Esse conjunto serve apenas para prover valores para a especificação de sistemas numéricos [24].

Para a construção do conjunto dos números naturais são definidos o conjunto $\mathbb{N}$ como um subconjunto de $\mathbb{A}$, os números 1 e 2 como elementos de $\mathbb{N}$ e a soma de naturais ($_{} + _{}^{}$) e o sucessor de um número natural (*succ* $_{}^{}$). Para a construção do conjunto dos números inteiros são definidos o conjunto propriamente dito ($\mathbb{Z}$) como um subconjunto de $\mathbb{A}$ e a operação de negação ($- _{}^{}$). A operação de adição definida para os naturais é estendida para os inteiros. Define-se também a subtração de inteiros ($_{} - _{}^{}$), a multiplicação ($_{} * _{}^{}$), a divisão ($_{} \text{div } _{}^{}$) e o resto da divisão ($_{} \text{mod } _{}^{}$). Além disso, são definidas as relações: menor ou igual ($\leq$), menor ($<$), maior ou igual ($\geq$) e maior ($>$).

Alguns conjuntos especiais de números são também definidos: naturais estritamente positivos ($\mathbb{N}_1$) e inteiros não nulos ($\mathbb{Z}_1$).

Com a definição dos inteiros é possível definir uma escala de números e a iteração de uma relação, isto é, a composição de uma relação um número finito de vezes. Para $i, j : \mathbb{Z}$, podemos denotar o conjunto dos números inteiros maiores ou iguais a i e menores ou iguais a j por $i..j$. A iteração de uma relação foi definida anteriormente pela Equação C.22. Porém, naquele caso ela foi definida apenas para números estritamente positivos. Podemos redefini-la para qualquer n inteiro

e relação $\mathcal{R} : X \leftrightarrow X$ (Equação C.34, [24]).

$$\mathcal{R}^n = \begin{cases} (\mathcal{R}^\sim)^{-n} & \text{se } n < 0 \\ \text{id } X & \text{se } n = 0 \\ \mathcal{R} \circ \mathcal{R}^{n-1} & \text{se } n > 0 \end{cases} \quad (\text{C.34})$$

São definidos também a função $\# _$ que toma um conjunto e devolve o número de elementos contidos nele, o mínimo ($\min _$) e o máximo ($\max _$) de um conjunto de números.

C.4 Seqüências

A partir da definição de números e funções, são definidas as seqüências finitas. Uma seqüência finita de elementos de um conjunto X é definida como uma função finita f dos naturais em X , cujo domínio seja o conjunto dado por $1..\#f$. O conjunto de todas as seqüências finitas de elementos de X é denotado por $\text{seq } X$ e definido pela Equação C.35, e o conjuntos das seqüências não vazias por $\text{seq}_1 X$ e pela Equação C.36 [24].

$$\{f : \mathbb{N} \multimap X \mid \text{dom } f = 1..\#f\} \quad (\text{C.35})$$

$$\text{seq } X \setminus \{\emptyset\} \quad (\text{C.36})$$

Dado que seqüências são funções finitas, podemos nos ater àquelas que são também injetivas, isto é, àquelas seqüências finitas cujos elementos são todos diferentes entre si. Essa classe de seqüências é denominada **seqüências injetivas** e denotada por $\text{iseq } X$. Sua definição é dada pela Equação C.37 [24].

$$\text{seq } X \cap (\mathbb{N} \multimap X) \quad (\text{C.37})$$

Assim como no caso de conjuntos, é possível definir seqüências através de um tipo de enumeração, essa enumeração é delimitada pelos símbolos ' $\langle$ ' e ' $\rangle$ '. Por exemplo, a seqüência dos número de 1 a 4 pode ser descrita pela Equação C.38 [24].

$$\langle 1, 2, 3, 4 \rangle \quad (\text{C.38})$$

Como seqüências são definidas em termos de funções finitas, os operadores que se aplicam a essas, também se aplicam àquelas. Porém, devido às particularidades das seqüências, são definidos novos operadores. Dentre eles, concatenação, inversa, primeiro, último, começo, fim, extração, filtragem e concatenação distributiva. Uma nova forma de construir seqüências denominada esmagamento é definida e algumas relações novas são descritas: prefixo, sufixo e infixo.

O operador de **concatenação** permite construir uma nova seqüência a partir de outras duas justapondo-as. Ele é denotado por $_ \frown _$ e sua definição é dada pela Equação C.39 [24].

$$\forall s, t : \text{seq } X \bullet s \frown t = s \cup \{n : \text{dom } t \bullet n + \#s \mapsto tn\} \quad (\text{C.39})$$

A **inversa** de uma seqüência é obtida tomando-se os elementos dela em ordem inversa. Ela é denotada por $rev _$ e definida pela Equação C.40 [24].

$$\forall s : seq X \bullet rev s = \lambda n : dom s \bullet s(\#s - n + 1) \quad (C.40)$$

Os operadores **primeiro** e **último**, devolvem respectivamente o primeiro e o último elemento de uma seqüência. Eles são denotados, respectivamente, por $head$ e $last$ e definidos pelas Equações C.41 e C.42 [24].

$$\forall s : seq X \bullet head s = s 1 \quad (C.41)$$

$$\forall s : seq X \bullet last s = s(\#s) \quad (C.42)$$

Os operadores **começo** e **fim**, devolvem a seqüência, respectivamente, sem o primeiro elemento e sem o último elemento. Eles são denotados por $front$ e $tail$ e definidos, respectivamente, pelas Equações C.43 e C.44 [24].

$$\forall s : seq X \bullet front s = \{\#s\} \triangleleft s \quad (C.43)$$

$$\forall s : seq X \bullet tail s = \lambda n : 1..(\#s - 1) \bullet s(n + 1) \quad (C.44)$$

O operador de **extração** toma uma seqüência e um conjunto de índices e produz uma nova seqüência restrita a esse conjunto de índices. Note que simplesmente eliminar os pares cujo primeiro elemento não está no conjunto de índices produz uma função finita que não é seqüência, visto que o domínio dela não é um conjunto contíguo de naturais positivos iniciados por 1. Para obter uma seqüência da função finita obtida eliminando-se tais elementos, é necessário usar o operador de **esmagamento** que toma uma função finita de domínio inteiro e a renumera de forma a que o domínio passe a ser um conjunto contíguo de naturais positivos iniciados por 1. O operador de esmagamento é denotado por $squash _$ e definido pela Equação C.45, enquanto o operador de extração é denotado por $_ \upharpoonright _$ e definido pela Equação C.46 [24]. A renumeração efetuada pelo operador de esmagamento conta para cada par da função finita, quantos pares existem com índices menores que o dele, e associa seu segundo componente e esse número.

$$\forall f : \mathbb{Z} \multimap X \bullet squash f = \{p : f \bullet \#(i : dom f \mid i < p.1) \mapsto p.2\} \quad (C.45)$$

$$\forall c : \mathbb{P} \mathbb{Z} ; s : seq X \bullet c \upharpoonright s = squash(c \triangleleft s) \quad (C.46)$$

O operador de extração possui uma contra parte que toma uma seqüência de elementos de um conjunto X e um subconjunto de X e produz a seqüência cujos elementos são os originais exceto aqueles não contidos no conjunto dado. Ele é denominado **filtragem**, denotado por $_ \upharpoonright _$ e definido pela Equação C.47 [24].

$$\forall s : seq X ; c : \mathbb{P} X \bullet s \upharpoonright c = squash(s \triangleright c) \quad (C.47)$$

No caso dos conjuntos, nós generalizamos os operadores de união e intersecção para trabalharem com mais de dois conjuntos. Para seqüências, podemos fazer o mesmo para o operador de concatenação, denominamos essa generalização de **concatenação distributiva**, ela toma uma seqüência S de seqüências e produz uma nova seqüência composta pela justaposição de todas as seqüências contidas em S na ordem imposta por S . Esse operador é denotado por $\frown / _$ e definido recursivamente pelo conjunto de Equações C.48, C.49 e C.50 [24], onde $\langle \rangle$ denota a seqüência vazia.

$$\frown / \langle \rangle = \langle \rangle \quad (\text{C.48})$$

$$\forall s : \text{seq } X \bullet \frown / \langle s \rangle = s \quad (\text{C.49})$$

$$\forall q, r : \text{seq seq } X \bullet \frown / (q \frown r) = (\frown / q) \frown (\frown / r) \quad (\text{C.50})$$

As relações definidas para seqüência, **prefixo**, **sufixo** e **infixo**, identificam a existência de subsequência, respectivamente no começo, fim e meio da seqüência. Elas são denotadas, respectivamente, por *prefix*, *suffix* e *infix*, e definidas pelas Equações C.51, C.52 e C.53 [24].

$$\forall s, t : \text{seq } X \bullet s \text{ prefix } t \Leftrightarrow s \subseteq t \quad (\text{C.51})$$

$$\forall s, t : \text{seq } X \bullet s \text{ suffix } t \Leftrightarrow (\exists u : \text{seq } X \bullet u \frown s = t) \quad (\text{C.52})$$

$$\forall s, t : \text{seq } X \bullet s \text{ infix } t \Leftrightarrow (\exists u, v : \text{seq } X \bullet u \frown s \frown v = t) \quad (\text{C.53})$$

Apêndice D

Regras de Tradução

A seguir apresentamos uma tabela contendo as regras de tradução dos elementos da notação Z. A primeira coluna contém o elemento de Z propriamente dito, a segunda contém a marcação $\text{\LaTeX}$ associada a ele e a terceira coluna apresenta o código Java associado. A marcação $\text{\LaTeX}$ é apresentada, pois é necessária para a criação de especificações que possam ser manipuladas pela ferramenta. A tradução assume um mapeamento $\mathcal{M}$ entre os conjuntos e tipos Java, e outro $\mathcal{T}$ entre elementos e seus tipos Java.

Os elementos marcados com um asterisco não são usados diretamente para compor uma especificação. Eles são parte de elementos da notação Z e foram separados para facilitar a exposição das regras de tradução.

Regra 1 (Declaração de Variável*)

Z - $v : T$
 $\text{\LaTeX}$ - $v: T$
Java - $\mathcal{M}(T) \ v;$

Regra 2 (Declaração de Constante*)

Z - $v == k$
 $\text{\LaTeX}$ - $v == k$
Java - `static  $\mathcal{T}(k)$  v = k;`

Regra 3 (Inclusão de Esquema*)

Z - S
 $\text{\LaTeX}$ - S
Java - `importSchema(new S(..)`

Regra 4 (Esquema de Estado)

Z -

S	
S_1	
$\vdots$	
S_n	
$v_1 : T_1$	
$\vdots$	
$v_m : T_m$	
$c_1 == k_1$	
$\vdots$	
$c_p == k_p$	
Predicado	

L^AT_EX -

```
\begin{schema}{S}
S_1\\
\vdots\\
S_n\\
v_1: T_1\\
\vdots\\
v_m: T_m\\
c_1 == k_1\\
\vdots\\
c_p == k_p
\where
Predicado
\end{schema}
```

Java - Nessa tradução utilizaremos a seguinte abreviação: $\mathcal{M}(T_i) = T_i$, $\mathcal{T}(k_i) = T_{k_i}$ e $v_i = v_i$.

```
public class S extends Schema {
    T1 v1;
    ...
    Tm vm;
    static Tk1 c1 = k1;
    ...
    static Tkp cp = kp;
    public S(T1 v1,...,Tm vm) {
        importSchema(new S1(...));
        ...
        importSchema(new Sn(...));
    }
    @Override
    protected void checkSpecificationInvariants()
```

```

        throws InvariantViolation {
            ...
        }
        @Override
        protected void checkUserInvariants () throws InvariantViolation {
            ...
        }
        @Override
        protected Object clone () throws CloneNotSupportedException {
            ...
        }
        @Override
        protected void execute () {
            ...
        }
    }
}

```

Regra 5 (Conjunção de Esquemas)

$$\mathbf{Z} - S_1 \wedge S_n$$

TEX - $S_1 \ \wedge S_2$

Java - new SchemaCalcAND(new S1(..), new S2(..));

Regra 6 (Disjunção de Esquemas)

$$\mathbf{Z} - S_1 \vee S_n$$

TEX - $S_1 \ \vee S_2$

Java - new SchemaCalcOR(new S1(..), new S2(..));

Regra 7 (Implicação de Esquemas)

$$\mathbf{Z} - S_1 \Rightarrow S_n$$

TEX - $S_1 \ \Rightarrow S_2$

Java - new SchemaCalcIMPLIES(new S1(..), new S2(..));

Regra 8 (Equivalência de Esquemas)

$$\mathbf{Z} - S_1 \Leftrightarrow S_n$$

TEX - $S_1 \ \Leftrightarrow S_2$

Java - new SchemaCalcIFF(new S1(..), new S2(..));

Regra 9 (Negação de Esquemas)

$$\mathbf{Z} - \neg S$$

TEX - $\neg S$

Java - new SchemaCalcNOT(new S(..));

Regra 10 (Inclusão de Esquema Δ^*)

$$\mathbf{Z} - \Delta S$$

TEX - ΔS

Java - uma instância do esquema é passada como parâmetro no mapeamento *initial* do método *execute*.

Regra 11 (Inclusão de Esquema Ξ^*)

Z - ΞS

L^AT_EX - $\backslash \Xi S$

Java - uma instância do esquema é passada como parâmetro no mapeamento *initial* do método *execute*.

Regra 12 (Esquema de Operação)

Z -

O
ΔDS_1
$\vdots$
ΔDS_i
ΞXS_1
$\vdots$
ΞXS_j
$v_1? : T_1^{in}$
$\vdots$
$v_m? : T_m^{in}$
$v_1! : T_1^{out}$
$\vdots$
$v_n! : T_n^{out}$
$c_1 == k_1$
$\vdots$
$c_p == k_p$
Predicado

L^AT_EX -

```
\begin{schema}{0}
\Delta DS_1\\
\vdots\\
\Delta DS_i\\
\Xi XS_1\\
\vdots\\
\Xi XS_j\\
v_1?: T^{in}_1\\
\vdots\\
v_m?: T^{in}_m\\
v_1!: T^{out}_1\\
\vdots\\
v_n!: T^{out}_n\\
c_1 == k_1\\
\vdots\\
c_p == k_p
```

```

v_1!: T^{out}_1\\
\vdots\\
v_n!: T^{out}_n\\
c_1 == k_1\\
\vdots\\
c_p == k_p
\where
Predicado
\end{schema}

```

Java - As variáveis de entrada $v_1?, \dots, v_m?$ são passada aos métodos da classe através do mapeamento *input*, e as variáveis de saída $v_1!, \dots, v_n!$ psão passadas através do mapeamento *output*. Os esquemas modificados são passados através do mapeamento *initial*.

```

public class O extends Operation {
    public O(String name) {
        super(name);
    }
    @Override
    public void checkSpecificationPostconditions(Map<String, Schema> before,
        Map<String, Schema> after, Map<String, Object> input,
        Map<String, Object> output) throws InvariantViolation {
        ...
    }
    @Override
    public void checkSpecificationPreconditions(Map<String, Schema> before,
        Map<String, Object> input) throws InvariantViolation {
        ...
    }
    @Override
    public void checkUserPostconditions(Map<String, Schema> before,
        Map<String, Schema> after, Map<String, Object> input,
        Map<String, Object> output) throws InvariantViolation {
        ...
    }
    @Override
    public void checkUserPreconditions(Map<String, Schema> before,
        Map<String, Object> input) throws InvariantViolation {
        ...
    }
    @Override
    protected Map<String, Object> execute(Map<String, Schema> initial,
        Map<String, Object> input) {
        ...
    }
    @Override
    protected void execute() {

```

```

        \\não é implementado
    }
    protected Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException("Not a state schema.");
    }

```

Regra 13 (Igualdade)

Z - $a = b$

TEX - $a = b$

Java - `a.equals(b);`

Regra 14 (Conjunto)

Z - $\mathbb{P} X$

TEX - $\backslash power X$

Java - `new BasicSet<X>;`

Regra 15 (Pertinência)

Z - $x \in X$

TEX - $x \in X$

Java - `X.contains(x);`

Regra 16 (Continência)

Z - $X \subseteq Y$

TEX - $X \subseteq Y$

Java - `Y.contains(X);`

Regra 17 (Continência Própria)

Z - $X \subset Y$

TEX - $X \subset Y$

Java - `Y.contains_properly(X);`

Regra 18 (União de Conjuntos)

Z - $A \cup B$

TEX - $A \cup B$

Java - `C.union(A,B);`

Regra 19 (Intersecção de Conjuntos)

Z - $A \cap B$

TEX - $A \cap B$

Java - `C.intersection(A,B);`

Regra 20 (Diferença de Conjuntos)

Z - $A \setminus B$

LaTeX - $A \setminus B$

Java - `C.difference(A,B);`

Regra 21 (Diferença Simétrica de Conjuntos)

$$Z - A \oplus B$$

LaTeX - $A \mathrel{\mathop{\mathrm{symdiff}}} B$

Java - `C.symmetric_difference(A,B);`

Regra 22 (União Generalizada de Conjuntos)

$$Z - \bigcup C$$

LaTeX - $\bigcup C$

Java - `A.generalized_union(C)`

Obs.: Esse método deve ser chamado em uma instância vazia da classe BasicSet.

Regra 23 (Disjunção Generalizada de Conjuntos)

$$Z - \bigcap C$$

LaTeX - $\bigcap C$

Java - `A.generalized_intersection(C)`

Obs.: Esse método deve ser chamado em uma instância vazia da classe BasicSet.

Regra 24 (Relações)

$$Z - X \leftrightarrow Y$$

LaTeX - $X \mathrel{\mathop{\mathrm{rel}}} Y$

Java - `new BasicRelation<X,Y>();`

Regra 25 (Funções Parciais)

$$Z - X \mapsto Y$$

LaTeX - $X \mathrel{\mathop{\mathrm{pfun}}} Y$

Java - `new BasicPFunction<X,Y>();`

Regra 26 (Aplicação (Relação))

$$Z - R(a, b)$$

LaTeX - $R(a, b)$

Java - `f.contains(a,b);`

Regra 27 (Aplicação (Função))

$$Z - f(a)$$

LaTeX - $f(a)$

Java - `f.apply(a);`

Regra 28 (Pertinência)

$$Z - (a \mapsto b) \in f$$

LaTeX - $(a \mathrel{\mathop{\mathrm{mapsto}}} b) \in f$

Java - `f.contains(a,b);`

Regra 29 (Continência)

$$\mathbf{Z} - f \subseteq g$$

TEX - `f \subsepeq g`

Java - `g.contains(f);`

Regra 30 (Continência Própria)

$$\mathbf{Z} - f \subset g$$

TEX - `f \subset g`

Java - `g.contains_properly(f);`

Regra 31 (União de Relações e Funções Parciais)

$$\mathbf{Z} - f \cup g$$

TEX - `f \cup g`

Java - `h.union(f,g);`

Regra 32 (Intersecção de Relações e Funções Parciais)

$$\mathbf{Z} - f \cap g$$

TEX - `f \cap g`

Java - `h.intersection(f,g);`

Regra 33 (Diferença de Relações e Funções Parciais)

$$\mathbf{Z} - f \setminus g$$

TEX - `f \setminusminus g`

Java - `h.difference(f,g);`

Regra 34 (Diferença Simétrica de Relações e Funções Parciais)

$$\mathbf{Z} - f \ominus g$$

TEX - `f \symdiff g`

Java - `h.symmetric_difference(f,g);`

Regra 35 (União Generalizada de Relações e Funções Parciais)

$$\mathbf{Z} - \bigcup G$$

TEX - `\bigcup G`

Java - `f.generalized_union(G);` Obs.: Esse método deve ser chamado em uma instância vazia da classe `BasicPFunction`.

Regra 36 (Intersecção Generalizada de Relações e Funções Parciais)

$$\mathbf{Z} - \bigcap G$$

TEX - `\bigcap G`

Java - `f.generalized_intersecction(G);`

Regra 37 (Domínio de Relações e Funções Parciais) $Z - \text{dom } f$ **L^AT_EX** - `\dom f`**Java** - `f.domain();`**Regra 38 (Imagem de Relações e Funções Parciais)** $Z - \text{ran } f$ **L^AT_EX** - `\ran f`**Java** - `f.range();`**Regra 39 (Composição Relacional de Relações e Funções Parciais)** $Z - f \circ g$ **L^AT_EX** - `f \comp g`**Java** - `h.relational_composition(f,g);`**Regra 40 (Composição Funcional de Relações e Funções Parciais)** $Z - f \circ g$ **L^AT_EX** - `f \circ g`**Java** - `h.functional_composition(f,g);`**Regra 41 (Restrição de Domínio de Relações e Funções Parciais)** $Z - a \triangleleft f$ **L^AT_EX** - `a \dres f`**Java** - `g.domain_restriction(f,a);`**Regra 42 (Restrição de Imagem de Relações e Funções Parciais)** $Z - f \triangleright a$ **L^AT_EX** - `f \rres a`**Java** - `g.range_restriction(f,a);`**Regra 43 (Subtração de Domínio de Relações e Funções Parciais)** $Z - a \triangleleft f$ **L^AT_EX** - `a \ndres f`**Java** - `g.domain_subtraction(f,a);`**Regra 44 (Subtração de Imagem de Relações e Funções Parciais)** $Z - f \triangleright a$ **L^AT_EX** - `f \nrres a`**Java** - `g.range_subtraction(f,a);`**Regra 45 (Inversão Relacional de Relações e Funções Parciais)** $Z - f^\sim$

L^AT_EX - $f \backslash \text{inv}$

Java - `f.relational_inversion();`

Regra 46 (Imagem Relacional de Relações e Funções Parciais)

Z - $f(C)$

L^AT_EX - $f \backslash \text{img } C \backslash \text{ring}$

Java - `f.relational_image(C);`

Regra 47 (Sobrescrita de Relações e Funções Parciais)

Z - $f \oplus g$

L^AT_EX - $f \backslash \text{oplus } g$

Java - `h.overriding(f,g);`

Regra 48 (Fecho transitivo de Relações e Funções Parciais)

Z - f^+

L^AT_EX - $f \backslash \text{plus}$

Java - `f.transitive_closure(f);`

Apêndice E

Implementação da classe BasicSet

Neste capítulo apresentamos a implementação completa da classe `BasicSet`. Para a especificação formal de cada operador de conjuntos, vide o Capítulo [B](#).

```
package jzed.mtoolkit;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

@SuppressWarnings("serial")
public class BasicSet<X> extends HashSet<X> implements ZSet<X> {
    public BasicSet() {
        super();
    }
    public BasicSet(Collection<X> S) {
        super();
        addAll(S);
    }

    @Override
    public Boolean contains(X e) {
        return super.contains(e);
    }

    @Override
    public Boolean contains(ZSet<X> S) {
        return super.containsAll(S);
    }

    @Override
    public Boolean contains_properly(ZSet<X> S) {
        return !(contains(S) || equals(S));
    }

    @Override
    public ZSet<X> difference(ZSet<X> S1, ZSet<X> S2) {
        clear();
```

```

        addAll(S1);
        for (X i : S2) {
            this.remove(i);
        }
        return this;
    }

    @Override
    public Boolean equals(ZSet<X> S) {
        return super.equals(S);
    }

    @Override
    public ZSet<X> generalized_intersection(ZSet<ZSet<X>> S) {
        generalized_union(S);
        for (Iterator<X> it = iterator(); it.hasNext();) {
            X x = it.next();
            for (ZSet<X> s: S) {
                if (!s.contains(x))
                    it.remove();
            }
        }
        return this;
    }

    @Override
    public ZSet<X> generalized_union(ZSet<ZSet<X>> S) {
        clear();
        for (ZSet<X> s: S) {
            addAll(s);
        }
        return this;
    }

    @Override
    public ZSet<X> intersection(ZSet<X> S1, ZSet<X> S2) {
        clear();
        addAll(S1);
        for (Iterator<X> it = iterator(); it.hasNext();) {
            if (!S2.contains(it.next()))
                it.remove();
        }
        return this;
    }

    @Override
    public ZSet<X> symmetric_difference(ZSet<X> S1, ZSet<X> S2) {
        clear();

```

```

        addAll(S1);
        for (X i : S2) {
            if (contains(i))
                remove(i);
            else
                add(i);
        }
        return this;
    }

    @Override
    public ZSet<X> union(ZSet<X> S1, ZSet<X> S2) {
        clear();
        addAll(S1);
        addAll(S2);
        return this;
    }
}

```


Apêndice F

Implementação da classe BasicRelation

Neste capítulo apresentamos a implementação completa da classe `BasicRelation`. Para a especificação formal de cada operador de relações, vide o Capítulo [C](#).

```
package jzed.mtoolkit;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

public class BasicRelation<X, Y> implements ZRelation<X, Y> {
    public HashMap<X, ZSet<Y>> mapx;
    public HashMap<Y, ZSet<X>> mapy;

    public BasicRelation() {
        mapx = new HashMap<X, ZSet<Y>>();
        mapy = new HashMap<Y, ZSet<X>>();
    }

    public BasicRelation(HashMap<X, ZSet<Y>> mapx, HashMap<Y, ZSet<X>> mapy) {
        this.mapx = mapx;
        this.mapy = mapy;
    }

    @Override
    public Y apply(X x) {
        ZSet<Y> sety = xapply(x);
        Iterator<Y> it = sety.iterator();
        if (it.hasNext()) return it.next();
        else return null;
    }

    @Override
    public Boolean contains(X x, Y y) {
        ZSet<Y> sety = xapply(x);
        return sety.contains(y);
    }

    @Override
```

```

public Boolean contains(ZRelation<X, Y> R) {
    if (!domain().contains(R.domain())) return false;
    for (X x: R.domain()) {
        if (!xapply(x).contains(R.xapply(x)))
            return false;
    }
    return true;
}

@Override
public Boolean contains_properly(ZRelation<X, Y> R) {
    return !(contains(R) || equals(R));
}

@Override
public ZRelation<X, Y> difference(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    for (X x: R1.domain()) {
        for (Y y: R1.xapply(x)) {
            if (!R2.contains(x, y))
                this.put(x,y);
        }
    }
    return this;
}

@Override
public ZSet<X> domain() {
    return new BasicSet<X>(mapx.keySet());
}

@Override
public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x: R.domain()) {
        if (S.contains(x)) {
            for (Y y: R.xapply(x)) {
                this.put(x,y);
            }
        }
    }
    return this;
}

@Override
public ZRelation<X, Y> domain_subtraction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x: R.domain()) {

```

```

        if (!S.contains(x)) {
            for (Y y: R.xapply(x)) {
                this.put(x,y);
            }
        }
    }
    return this;
}

@Override
public Boolean equals(ZRelation<X, Y> R) {
    if (!domain().equals(R.domain())) return false;
    for (X x: R.domain()) {
        if (!xapply(x).equals(R.xapply(x)))
            return false;
    }
    return true;
}

@Override
public <T> ZRelation<X, Y> functional_composition(ZRelation<X, T> R1,
        ZRelation<T, Y> R2) {
    clear();
    for (X x: R1.domain()) {
        for (T t: R1.xapply(x)) {
            if (R2.domain().contains(t)) {
                for (Y y: R2.xapply(t)) {
                    this.put(x,y);
                }
            }
        }
    }
    return this;
}

@Override
public ZRelation<X, Y> generalized_intersection(ZSet<ZRelation<X, Y>> R) {
    clear();
    Iterator<ZRelation<X,Y>> it = R.iterator();
    if (!it.hasNext()) return this;
    else union(it.next());
    while (it.hasNext()) {
        intersection(it.next());
    }
    return this;
}

@Override

```

```

public ZRelation<X, Y> generalized_union(ZSet<ZRelation<X, Y>> R) {
    clear();
    for (ZRelation<X,Y> r: R) {
        union(r);
    }
    return this;
}

@Override
public ZRelation<X, Y> intersection(ZRelation<X, Y> R1,
    ZRelation<X, Y> R2) {
    union(R1);
    intersection(R2);
    return this;
}

public ZRelation<X, Y> intersection(ZRelation<X, Y> R) {
    for (X x: domain()) {
        for (Y y: xapply(x)) {
            if (!R.contains(x, y)) {
                remove(x, y);
            }
        }
    }
    return this;
}

@Override
public ZRelation<X, Y> overriding(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R2);
    ZSet<X> dom2 = R2.domain();
    for (X x: R1.domain()) {
        if (!dom2.contains(x))
            for (Y y: R1.xapply(x)) {
                put(x,y);
            }
    }
    return this;
}

@Override
public ZSet<Y> range() {
    return new BasicSet<Y>(mapy.keySet());
}

@Override
public ZRelation<X, Y> range_restriction(ZRelation<X, Y> R, ZSet<Y> S) {

```

```

        clear();
        for (Y y: R.range()) {
            if (S.contains(y)) {
                for (X x: R.yapply(y)) {
                    this.put(x,y);
                }
            }
        }
        return this;
    }

    @Override
    public ZRelation<X, Y> range_subtraction(ZRelation<X, Y> R, ZSet<Y> S) {
        clear();
        for (Y y: R.range()) {
            if (!S.contains(y)) {
                for (X x: R.yapply(y)) {
                    this.put(x,y);
                }
            }
        }
        return this;
    }

    @Override
    public ZRelation<X, X> reflexive_transitive_closure(ZRelation<X, X> R) {
        throw new MethodNotImplementedException(
            "reflexive_transitive_closure");
    }

    @Override
    public <T> ZRelation<X, Y> relational_composition(ZRelation<X, T> R1,
        ZRelation<T, Y> R2) {
        return functional_composition(R1,R2);
    }

    @Override
    public ZSet<Y> relational_image(ZSet<X> S) {
        ZSet<Y> ring = new BasicSet<Y>();
        for (X x: S) {
            ring.addAll(xapply(x));
        }
        return ring;
    }

    @Override
    public ZRelation<Y, X> relational_inversion() {
        ZRelation<Y,X> r = new BasicRelation<Y,X>();

```

```

        for (X x: domain()) {
            for (Y y: xapply(x)) {
                r.put(y,x);
            }
        }
        return r;
    }

    @Override
    public ZRelation<X, Y> symmetric_difference(ZRelation<X, Y> R1,
        ZRelation<X, Y> R2) {
        clear();
        union(R1);
        for (X x: R2.domain()) {
            for (Y y: R2.xapply(x)) {
                if (contains(x,y))
                    remove(x,y);
                else
                    put(x,y);
            }
        }
        return this;
    }

    @Override
    public ZRelation<X, X> transitive_closure(ZRelation<X, X> R) {
        ArrayList<X> elements = new ArrayList<X>();
        elements.addAll((new BasicSet<X>()).union(R.domain(), R.range()));
        int n = elements.size();
        int[][] matrix = new int[n][n];
        for (X x: R.domain()) {
            for (X y: R.xapply(x)) {
                matrix[elements.indexOf(x)][elements.indexOf(y)] = 1;
            }
        }

        for (int j = 0; j < n; j++) {
            for (int i = 0; i < n; i++) {
                if (matrix[i][j] == 1) {
                    for (int k = 0; k < n; k++) {
                        if (matrix[j][k] == 1)
                            matrix[i][k] = 1;
                    }
                }
            }
        }

        ZRelation<X,X> r = new BasicRelation<X,X>();

```

```

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 1) {
                    r.put(elements.get(i), elements.get(j));
                }
            }
        }
        return r;
    }

    @Override
    public ZRelation<X, Y> union(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
        clear();
        union(R1);
        union(R2);
        return this;
    }

    public ZRelation<X, Y> union(ZRelation<X, Y> R) {
        for (X x: R.domain()) {
            for (Y y: R.xapply(x)) {
                put(x,y);
            }
        }
        return this;
    }

    @Override
    public Y put(X x, Y y) {
        ZSet<Y> sety = xapply(x);
        ZSet<X> setx = yapply(y);
        sety.add(y);
        setx.add(x);
        mapx.put(x, sety);
        mapy.put(y, setx);
        return y;
    }

    @Override
    public Y remove(X x, Y y) {
        ZSet<Y> sety = xapply(x);
        ZSet<X> setx = yapply(y);
        if (sety != null && setx != null) {
            setx.remove(x);
            sety.remove(y);
            mapx.put(x, sety);
            mapy.put(y, setx);
        }
    }

```

```

        if (sety.isEmpty()) {
            mapx.remove(x);
        }
        if (setx.isEmpty()) {
            mapy.remove(y);
        }
    }
    return y;
}

@Override
public ZSet<Y> xapply(X x) {
    ZSet<Y> sety = mapx.get(x);
    if (sety != null) return sety;
    else return new BasicSet<Y>();
}

@Override
public ZSet<X> yapply(Y y) {
    ZSet<X> setx = mapy.get(y);
    if (setx != null) return setx;
    else return new BasicSet<X>();
}

@Override
public void clear() {
    mapx.clear();
    mapy.clear();
}

public String toString() {
    String s = "";
    for (X x: domain()) {
        for (Y y: xapply(x)) {
            s = s + x + " → " + y + "\n";
        }
    }
    return s;
}

@Override
public Object clone() {
    return new BasicRelation<X,Y>((HashMap<X,ZSet<Y>>)mapx.clone(),
        (HashMap<Y,ZSet<X>>)mapy.clone());
}

```


}

Apêndice G

Implementação da classe BasicPFunction

Neste capítulo apresentamos a implementação completa da classe `BasicPFunction`. Para a especificação formal de cada operador de funções parciais, vide o Capítulo [C](#).

```
package jzed.mtoolkit;

import java.util.HashMap;
import java.util.Iterator;

@SuppressWarnings("serial")
public class BasicPFunction<X, Y> extends HashMap<X, Y> implements
    ZRelation<X, Y> {

    @Override
    public Y apply(X x) {
        return get(x);
    }

    @Override
    public Boolean contains(X x, Y y) {
        Y aux = apply(x);
        if (aux == null)
            return false;
        if (aux.equals(y))
            return true;
        else
            return false;
    }

    @Override
    public Boolean contains(ZRelation<X, Y> R) {
        for (X x : R.domain()) {
            Y y = R.apply(x);
            if (!contains(x, y))
                return false;
        }
        return true;
    }
}
```

```

@Override
public Boolean contains_properly(ZRelation<X, Y> R) {
    return !(contains(R) || equals(R));
}

@Override
public ZRelation<X, Y> difference(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    for (X x : R1.domain()) {
        Y y = R1.apply(x);
        if (!R2.contains(x, y))
            put(x, y);
    }
    return this;
}

@Override
public ZSet<X> domain() {
    return new BasicSet<X>(keySet());
}

@Override
public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x : R.domain()) {
        Y y = R.apply(x);
        if (S.contains(x))
            put(x, y);
    }
    return this;
}

@Override
public ZRelation<X, Y> domain_subtraction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x : R.domain()) {
        Y y = R.apply(x);
        if (!S.contains(x))
            put(x, y);
    }
    return this;
}

@Override
public Boolean equals(ZRelation<X, Y> R) {
    if (!domain().equals(R.domain()))
        return false;

```

```

        for (X x : R.domain()) {
            if (!apply(x).equals(R.apply(x)))
                return false;
        }
        return true;
    }

    @Override
    public <T> ZRelation<X, Y> functional_composition(ZRelation<X, T> R1,
        ZRelation<T, Y> R2) {
        clear();
        for (X x : R1.domain()) {
            Y y = R2.apply(R1.apply(x));
            put(x, y);
        }
        return this;
    }

    @Override
    public ZRelation<X, Y> generalized_intersection(ZSet<ZRelation<X, Y>> R) {
        clear();
        Iterator<ZRelation<X, Y>> it = R.iterator();
        if (!it.hasNext())
            return this;
        else
            union(it.next());
        while (it.hasNext()) {
            intersection(it.next());
        }
        return this;
    }

    @Override
    public ZRelation<X, Y> generalized_union(ZSet<ZRelation<X, Y>> R) {
        clear();
        for (ZRelation<X, Y> r : R) {
            union(r);
        }
        return this;
    }

    @Override
    public ZRelation<X, Y> intersection(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
        clear();
        union(R1);
        intersection(R2);
        return this;
    }

```

```

public ZRelation<X, Y> intersection(ZRelation<X, Y> R) {
    for (X x : domain()) {
        Y y = apply(x);
        if (!R.contains(x, y)) {
            remove(x, y);
        }
    }
    return this;
}

@Override
public ZRelation<X, Y> overriding(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R2);
    ZSet<X> dom2 = R2.domain();
    for (X x : R1.domain()) {
        if (!dom2.contains(x)) {
            Y y = R1.apply(x);
            put(x, y);
        }
    }
    return this;
}

@Override
public ZSet<Y> range() {
    return new BasicSet<Y>(values());
}

@Override
public ZRelation<X, Y> range_restriction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (X x: R.domain()) {
        Y y = R.apply(x);
        if (S.contains(y))
            put(x,y);
    }
    return this;
}

@Override
public ZRelation<X, Y> range_subtraction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (X x: R.domain()) {
        Y y = R.apply(x);
        if (!S.contains(y))
            put(x,y);
    }
}

```

```

        }
        return this;
    }

    @Override
    public ZRelation<X, X> reflexive_transitive_closure(ZRelation<X, X> R) {
        throw new MethodNotImplementedException("This class does not "+
            "implement method reflexive_transitive_closure");
    }

    @Override
    public <T> ZRelation<X, Y> relational_composition(ZRelation<X, T> R1,
        ZRelation<T, Y> R2) {
        return functional_composition(R1,R2);
    }

    @Override
    public ZSet<Y> relational_image(ZSet<X> S) {
        ZSet<Y> ring = new BasicSet<Y>();
        for (X x: S) {
            ring.add(apply(x));
        }
        return ring;
    }

    @Override
    public ZRelation<Y, X> relational_inversion() {
        ZRelation<Y, X> f = new BasicPFunction<Y,X>();
        for (X x: domain()) {
            Y y = apply(x);
            if (f.domain().contains(y))
                throw new NotInjectionException();
            else
                f.put(y,x);
        }
        return f;
    }

    @Override
    public Y remove(X x, Y y) {
        Y aux = apply(x);
        if (y.equals(aux))
            return remove(x);
        else
            return null;
    }

    @Override

```

```

public ZRelation<X, Y> symmetric_difference(ZRelation<X, Y> R1,
                                           ZRelation<X, Y> R2) {
    clear();
    union(R1);
    for (X x: R2.domain()) {
        Y y = R2.apply(x);
        if (contains(x,y))
            remove(x,y);
        else
            put(x,y);
    }
    return this;
}

@Override
public ZRelation<X, X> transitive_closure(ZRelation<X, X> R) {
    ZRelation<X,X> r = new BasicRelation<X,X>();
    for (X x: R.domain()) {
        X y = R.apply(x);
        r.put(x, y);
    }
    return r.transitive_closure(r);
}

@Override
public ZRelation<X, Y> union(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R1);
    union(R2);
    return this;
}

public ZRelation<X, Y> union(ZRelation<X, Y> R) {
    if (!(new BasicSet<X>()).intersection(domain(), R.domain()).isEmpty()) {
        // the result is not a function
        throw new NotFunctionException();
    } else {
        for (X x : R.domain()) {
            for (Y y : R.xapply(x)) {
                put(x, y);
            }
        }
    }
    return this;
}

@Override
public ZSet<Y> xapply(X x) {

```



```

        ZSet<Y> set = new BasicSet<Y>();
        set.add(apply(x));
        return set;
    }

    @Override
    public ZSet<X> yapply(Y y) {
        ZSet<X> set = new BasicSet<X>();
        if (range().contains(y)) {
            for (X x : domain()) {
                if (y.equals(apply(x)))
                    set.add(x);
            }
        }
        return set;
    }
}

```


Referências Bibliográficas

- [1] *CZT*, <http://czt.sourceforge.net/>.
- [2] *fuzz*, <http://spivey.oriel.ox.ac.uk/mike/fuzz/>.
- [3] *Proofpower*, <http://www.lemma-one.com/ProofPower/index/>.
- [4] *ZANS*, <http://venus.cs.depaul.edu/fm/zans.html>.
- [5] *ZTC*, <http://venus.cs.depaul.edu/fm/ztc.html>.
- [6] Ali E. Abdallah, Jonathan P. Bowen, Alexandra Barros, and José B. Barros, *A provably correct functional programming approach to the prototyping of formal Z specifications*, Proc. ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'03), Tunis, Tunisia, IEEE Computer Society, 14-18 July 2003.
- [7] J.R. Abrial, *The B-book: Assigning programs to meanings*, Cambridge University Press, 1996.
- [8] Apache Software Foundation, *The apache velocity project*, 2008, <http://velocity.apache.org/>.
- [9] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright, *Refinement calculus: A systematic introduction*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
- [10] A. L. C. Cavalcanti and J. C. P. Woodcock, *ZRC—A Refinement Calculus for Z*, Formal Aspects of Computing **10** (1999), no. 3, 267—289.
- [11] Quentin Charatan and Aaron Kans, *Formal software development: From VDM to java*, Palgrave, 2003.
- [12] Flávio Soares Corrêa da Silva, Marcelo Finger, and Ana Cristina Vieira de Melo, *Lógica para computação*, Thomson Learning, São Paulo, 2006.
- [13] Jim Davies, Charles Crichton, Edward Crichton, David Neilson, and Ib Holm Sørensen, *Formality, evolution, and model-driven software engineering*, Electronic Notes in Theoretical Computer Science (2005).
- [14] Sophia Drossopoulou and Susan Eisenbach, *Towards an Operational Semantics and Proof of Type Soundness for Java*, Formal Syntax and Semantics of Java, no. 3, Springer-Verlag, 1999.
- [15] David Faitelson, James Welch, and Jim Davies, *From predicates to programs: The semantics of a method language*, Electronic Notes in Theoretical Computer Science **184** (2007), 171–187.
- [16] A. F. Freitas, C. M. P. Nascimento, and A. L. C. Cavalcanti, *A Refinement Tool for Z*, Formal Methods and Software Engineering: 5th International Conference on Formal Engineering Methods, ICFEM 2003 (J. S. Dong and J. C. P. Woodcock, eds.), Lecture Notes in Computer Science, vol. 2885, Springer-Verlag, 2003, pp. 396 – 415.

- [17] Robert Geisler, Marcus Klar, and Stefan Mann, *Precise UML Semantics Through Formal Metamodeling*, Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How? (Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, eds.), 1998.
- [18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The java language specification*, Prentice Hall, 2005.
- [19] Wolfgang Grieskamp, *A computation model for Z based on concurrent constraint resolution*, Lecture Notes in Computer Science **1878** (2000), 414–432.
- [20] B. Hailpern and P. Tarr, *Model-driven development: the good, the bad, and the ugly*, IBM Syst. J. **45** (2006), no. 3, 451–461.
- [21] Dan Hazel, Paul A. Strooper, and Owen Traynor, *Requirements engineering and verification using specification animation*, Automated Software Engineering, 1998, pp. 302–305.
- [22] C. A. R. Hoare, *Communicating sequential processes*, Commun. ACM **21** (1978), no. 8, 666–677.
- [23] Antonio Houaiss, Mauro de Salles Villar, and Francisco Manoel de Mello Fraco, *Dicionário houaiss da língua portuguesa*, Objetiva, Rio de Janeiro, 2001.
- [24] ISO/IEC, *Information technology – Z formal specification notation – Syntax, type system and semantics*, Tech. Report ISO/IEC 13568:2002(E), ISO/IEC, 2002.
- [25] Jonathan Jacky, *The way of Z: practical programming with formal methods*, Cambridge University Press, New York, NY, USA, 1996.
- [26] Xiaoping Jia, *An approach to animating Z specifications*, COMPSAC '95: Proceedings of the 19th International Computer Software and Applications Conference (Washington, DC, USA), IEEE Computer Society, 1995, p. 108.
- [27] C. B. Jones, *Systematic software development using VDM*, Prentice Hall, 1990.
- [28] Gerald Kotonya and Ian Sommerville, *Requirements engineering: Processes and techniques*, John Wiley and Sons, 1998.
- [29] Shaoying Liu and Hao Wang, *An automated approach to specification animation for validation*, J. Syst. Softw. **80** (2007), no. 8, 1271–1285.
- [30] Petra Malik and Mark Utting, *CZT: A Framework for Z Tools*, ZB, 2005, pp. 65–84.
- [31] Zohar Manna and Richard Waldinger, *A deductive approach to program synthesis*, ACM Trans. Program. Lang. Syst. **2** (1980), no. 1, 90–121.
- [32] ———, *Fundamentals of deductive program synthesis*, IEEE Trans. Softw. Eng. **18** (1992), no. 8, 674–704.
- [33] Joaquin Miller and Jishnu Mukerji, *Mda guide version 1.0.1*, Tech. Report omg/2003-06-01, The Object Management Group, 2003, See also <http://www.omg.org/mda/>.
- [34] Robin Milner, *A calculus of communicating systems*, Lecture Notes in Computer Science, vol. 92, Springer, 1980.

- [35] Alvaro Miyazawa, Paulo Salem da Silva, and Ana C. V. de Melo, *Jzed-gen: Towards pragmatical generation of software from Z specifications*, Brazilian Symposium on Formal Methods Special Tracks, 2008.
- [36] Carroll Morgan, *Programming from specifications (2nd ed.)*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.
- [37] M. V. M. Oliveira, *Formal derivation of state-rich reactive programs using circus*, Ph.D. thesis, University of York, 2005.
- [38] M. V. M. Oliveira and A. L. C. Cavalcanti, *Tactics of Refinement*, 14th Brazilian Symposium on Software Engineering, 2000, pp. 117 – 132.
- [39] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock, *ArcAngel: A Tactic Language for Refinement*, Formal Aspects of Computing **15** (2003), no. 1, 28 – 47.
- [40] M. V. M. Oliveira, M. A. Xavier, and A. L. C. Cavalcanti, *Refine and Gabriel: Support for Refinement and Tactics*, 2nd IEEE International Conference on Software Engineering and Formal Methods (J. R. Cuellar and Z. Liu, eds.), IEEE Computer Society Press, September 2004, pp. 310 – 319.
- [41] OMG, *Model driven architecture (MDA) FAQ*, http://www.omg.org/mda/faq_mda.htm.
- [42] OMG Architecture Board ORMSC, *Model driven architecture (MDA)*, July 2001, OMG document number ormsc/2001-07-01, available from www.omg.org.
- [43] Sowmiya Ramkarthik and Cui Zhang, *Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z*, ICIS-COMSAR '06: Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (Washington, DC, USA), IEEE Computer Society, 2006, pp. 405–411.
- [44] Mark Saaltink and Ora Canada, *The z/eves 2.0 user's guide*, 1999.
- [45] Paulo Salem and Ana C. V. de Melo, *A simulation-oriented formalization for a psychological theory*, 10th International Conference, FASE 2007 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007 Proceedings (Matthew B. Dwyer and Antonia Lopes, eds.), Lecture Notes in Computer Science, vol. 4422, Springer-Verlag, 2007, pp. 42–56.
- [46] Robert Sedgewick, *Algorithms in C, part 5: graph algorithms*, 3rd ed., Addison-Wesley Professional, 2001.
- [47] Graeme Smith, *The Object-Z specification language*, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
- [48] Graeme Smith and Luke Wildman, *Model checking z specifications using sal*, International Conference of Z and B Users (ZB 2005), volume 3455 of LNCS, Springer-Verlag, 2005, pp. 87–105.
- [49] Ian Sommerville, *Software engineering (5th ed.)*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

- [50] Ian Sommerville and Pete Sawyer, *Requirements Engineering: A Good Practice Guide*, John Wiley and Sons, 1997.
- [51] J. M. Spivey, *Understanding Z: a specification language and its formal semantics*, Cambridge University Press, 1988.
- [52] ———, *The Z notation: a reference manual*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.
- [53] P. Steggles and J. Hulance, *Z tools survey*, 1994.
- [54] Sun Microsystems, *The All-New Java 2 platform, Standard Edition (J2SE) 5.0 Platform: Programming with the New Language Features in J2SE 5.0*, 2004, <http://java.sun.com/developer/technicalArticles/releases/j2se15langfeat/>.
- [55] ———, *Java technology*, 2007, <http://java.sun.com/>.
- [56] Marcin Szlenk, *Formal Semantics and Reasoning about UML Class Diagram*, DEPCOS-RELCOMEX '06: Proceedings of the International Conference on Dependability of Computer Systems (Washington, DC, USA), IEEE Computer Society, 2006, pp. 51–59.
- [57] D. Thomas, *MDA: revenge of the modelers or UML utopia?*, IEEE Softw. **21** (2004), no. 3, 15–17.
- [58] Anneliese von Mayrhauser, *Software engineering: methods and management*, Academic Press Professional, Inc., San Diego, CA, USA, 1990.
- [59] M. Winikoff, P. Dart, and E. Kazmierczak, *Rapid prototyping using formal specifications*, 1998.
- [60] Kenneth R. Wood, *A practical approach to software engineering using Z and the refinement calculus*, SIGSOFT Softw. Eng. Notes **18** (1993), no. 5, 79–88.
- [61] J. C. P. Woodcock and A. L. C. Cavalcanti, *The semantics of circus*, ZB 2002: Formal Specification and Development in Z and B (D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, eds.), Lecture Notes in Computer Science, vol. 2272, Springer-Verlag, 2002, pp. 184–203.
- [62] Jim Woodcock and Jim Davies, *Using Z: Specification, refinement, and proof*, Prentice Hall, 1996.
- [63] J. B. Wordsworth, *Software engineering with B*, Addison-Wesley, 1996.