UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM SISTEMA DE TIPOS PARA UMA LINGUAGEM DE REPRESENTAÇÃO ESTRUTURADA DE CONHECIMENTO

por

LILIANA MARIA PASSERINO

Dissertação submetida como requisito parcial
para a obtenção do grau de Mestre em
Ciência da Computação

Prof. Paulo Azeredo
Orientador
Prof. Antônio Carlos da Rocha Costa
Co-orientador

Porto Alegre, 31 de julho de 1992.

CIP CATALOGAÇÃO NA PUBLICAÇÃO

Passerino,Liliana Maria

Um Sistema de Tipos para uma Linguagem de Representação Estruturada de Conhecimento / Liliana Maria Passerino.- Porto Alegre : CPGCC da UFRGS, 1992.

196 p. : il.

Dissertação (mestrado) - Universidade Federal de Rio Grande do Sul, Curso de Pós-Graducação em Ciência da Computação, Porto Alegre, 1992. Orientador: Azeredo, Paulo. Co-Orientador: Costa, Antonio Carlos da Rocha.

Dissertação : Um Sistema de Tipos para uma Linguagem de Representação Estruturada de Conhecimento.

Dedico este trabalho a meus pais

Ernesto Mario Passerino

e

Natalia C. M. Gili Borghet

por sua compreensão e carinho.

SUMARIO

LISTA DE ABREVIATURAS

Cap.	Capítulo
Fig.	Figura
IA	Inteligência Artificial
i. é	Isto é
$\mathcal{L}$	Linguagem
RC	Redes Conceituais
RS	Redes Semânticas
TAD	Tipo Abstrato de Dados
TMG	Tipo mais geral
$\mathcal{U}$	Universo de Discurso

LISTA DE FIGURAS

RESUMO

A noção de tipo é intrínseca ao raciocínio humano, na medida que os seres humanos tendem a "classificar" os objetos segundo seu uso e seu comportamento como parte do processo de resolução de problemas. Tal classificação dos objetos implica numa abstração das características irrelevantes dos mesmos,permitindo dessa maneira uma simplificação importante da complexidade do universo de discurso

Por outro lado, certos problemas são altamente complexos e requerem um tratamento diferenciado.Esses problemas exigem, para sua resolução, um grande conhecimento do universo de discurso. O ponto crítico nesta situação é que o domínio do problema não é exato como poderia ser um domínio matemático. Pelo contrário, ele inclui geralmente aspectos ambíguos e pouco formais que dificultam seu entendimento. Tal domínio é chamado de senso comum e é objeto de estudo de uma linha da computação, a Inteligência Artificial (IA). Para [KRA 87], entre outros, as soluções para muitos problemas de IA dependem mais da capacidade de adquirir e manipular conhecimento do que de algoritmos sofisticados. Por este motivo, existem na IA muitos tipos de linguagems que tentam, de diversas maneiras,facilitar a representação de conhecimentos sobre universos de discurso de problemas particulares. São as chamadas Linguagens de Representação de Conhecimento.

A noção de tipo é implícita nas linguagens de representação de conhecimento, uma vez que tal noção é natural no raciocínio humano e está intimamente ligada ao conceito de abstração.

Este trabalho visa explicitar a noção de tipo subjacente ao núcleo definido da linguagem RECON-II. Para isto, foi realizado um estudo semântico prévio para identificar os tipos semânticos da linguagem. A partir da noção semântica dos tipos foi possível definir a correspondente sintática e finalmente, descrever um Sistema de Tipos para RECON-II.

Um Sistema de Tipos consiste numa Linguagem de Tipos (tipos básicos + construtores de tipos) e num Sistema de Dedução que relaciona as expressões da linguagem objeto (linguagem de programação) com as expressões da linguagem de tipos.

Para a primeira etapa realizada neste trabalho, a

determinação da semântica da linguagem, foi utilizado o método algébrico. Nele toda expressão RECON-II é um termo de uma assinatura Σ, de modo que cada assinatura Σ determina um conjunto de expressões RECON-II.

Mas, por outro lado, uma assinatura também determina um conjunto de álgebras. Dessas álgebras-Σ só um subconjunto é significativo para as expressões RECON-II. As álgebras-Σ significativas são aquelas que satisfazem a assinatura-Σ mais um conjunto E de axiomas. A assinatura-Σ junto como o conjunto E de axiomas constituem o que se denomina Tipo Abstrato de Dados, T=(Σ, E), e as álgebras-Σ significativas são os chamados modelos-Σ do tipo T.

Assim, uma expressão RECON-II α é um elemento da algebra de termos W_{Σ}, que é uma álgebra gerada a partir de Σ. Essa álgebra, é o conjunto das expressões RECON-II significativas, e é o modelo inicial de tais expressões [GOG 78].

Dado um tipo abstrato T existe um único modelo para T, ou uma classe de modelos, não isomórficos, denominada M(T). No segundo caso, esses modelos constituem uma "quasi" ordem parcial com modelo inicial e terminal. A existência e unicidade do modelo inicial para qualquer tipo T foi demostrada por [GOG 77].

Com $\Sigma = (S, F)$, a $(W_{\Sigma})_s$ para $s \in S$, é o conjunto dos termos de "sort" s. Na RECON-II, são os termos de uma categoria sintática determinada. As categorias sintáticas principais são : Conceitos, Relações, Funções e Redes.

Um tipo semântico para $s \in S$ é um subconjunto $M(T)_s \subseteq M(T)$ que satisfaz os axiomas E_s exigidos de $(W_{\Sigma})_s$, constituindo o tipo abstrato T_s (por exemplo TConceitos, TRedes, etc.)

Por último foi definido o Sistema de Tipos, que consiste numa estrutura sintática adequada para os tipos semânticos de cada expressão-RECON e, para cada expressão de tipo, um conjunto de regras de inferências que permita, a partir de uma expressão-RECON inferir seu tipo mais geral.

Palavras-chave : tipos, tipos abstratos de dados, sistema de inferência de tipos, representação de conhecimento, linguagens de representação de conhecimento, semântica algébrica.

TITLE :" A Type Sistems for a Knowledge Structured Representation Language"

ABSTRACT

The notion of type is intrinsic to human reasoning, since human beings tend to classify objects according their use and behaviour as part of the problem solving process. By classifying objects, their irrevelant characteristics are abstrated; in this way, the complexity of the universe of discourse is much reduced.

On the other hand, certain problems are higly complex and require a differentiated treatament. In order to solve these problems, a great knowledge of de universe of discourse is needed. The critical proint in this situation is that the domain of the problem isn't as precise as a mathematic domain. On the contrary, it generally, includes ambiguous and not very formal aspects wich make its uderstanding difficult. Such a domains is known as common sense and this is the object of studies of one line of Computer Science, Artificial Intelligence (AI). For [KRA 87], among others, the solutions for many AI problems depend on the ability for acquiring and manipulating knowledge rather than on sophisticated algorithms. For this reason, there are in AI many type of languages that attemps in different ways, to represent the UD of a particular problem. These languagesare known as Knowledge Representation Languages.

The notion of type is implicit in Knowledge Representation Languages, since it is natural in human reasoning and closely rrelated to the concept of abstraction.

This work intends to make the notion of type intrinsic to the RECON-II's kernel language, explicity. In order to do this, a preliminary semantic stidy was carriedaut to identify the semantic types of the languages. From the semantic notion of the types it was possible to define the sintactic counterpart and finally to describe a Type System for RECON-II.

A Type System conssit of a type language (basic types + types constructors) end a deduction system that relattes

expressions in the language object (programming language) to the expressions in the type language.

In the first step of this work, language semantic determination, the algebric method was used. In it every RECON-II expression is one term of a signature Σ, so thet every signature Σ determines a RECON-II expressions set.

On the other hand, a signature also determines a set of algebras. Out of these Σ-algebras only one subset is significant to the RECON-II expressions. The significant Σ-algebras are those thet satisfy the Σ-signature and a set E of axioms. Together the Σ-siganture and the set E of axioms, constitute what is called Abstract Data Type T = (Σ, E) and the significant Σ-algebras are the so-called Σ-models of type T.

Therefore a RECON-II expressions a is an element of the world algebra W_{Σ} wich is an algebra generated from Σ. This algebra is the set of significant RECON-II expressions, and is the initia; model of such expressions [GOG 78].

Given an abstract type T there is one single model for T or one class of nonisomorphic models denominated M(T). In the second case, these models constitute a "quasi" partial order with an initial and terminal model. the exixstence nad uniqueness of the inititial model for any type T was shown at [GOG 77].

With Σ = (S, F), $(W_{\Sigma})_{s}$ for $s \in S$, is the set of terms of sort s. In RECON-II, those are the term of determinate sintactic category. The main sintactic categories are : Concepts, Relations, Functions and Nets.

A semantic type for $s \in S$ is s subset $M(T)_{s} \subseteq M(T)$ that satisfies the axioms E_{s} required from (W_{Σ}), constituting the abstract type T_{s} (for instance Tconcepts, Tnets, etc.).

Finally, the type systems was defined, consisting a syntatic structure suitable for the semantic types of each RECON-II expressions and for every type expressions, a set of inference rules wich allows infering its more general type from a RECON-II expressions.

Key-Words: Types, Abstract Data Types, types inference systems, knowledege representation, knowledege representation language, algebric semantic.

1 INTRODUÇÃO

Na última década, o conceito de tipo adquiriu grande relevância na definição de linguagens de programação mais poderosas. Esta situação origina-se no fato de que os tipos permitem caracterizar valores,organizando, dessa maneira, o universo de discurso impedindo a geração de expressões não significativas nas linguagens.

A noção de tipo é intrínseca ao raciocínio humano, na medida que os seres humanos tendem a "classificar" os objetos segundo seu uso e seu comportamento, como parte do processo de resolução de problemas. No desenvolvimento de linguagens de programação, também evidenciou-se essa necessidade de diferenciar objetos. Assim, no início tais linguagens possuiam intuitivamente a noção de tipo, como em FORTRAN, para posteriormente evoluir incorporando-os explicitamente. Exemplo dessa evolução são PASCAL, ADA, SIMULA, ML, etc. [CAR 85] [CAR 86] [DAN 88].

Paralelamente a essa classificação dos objetos do domínio do problema, observou-se que os tipos também permitiam uma abstração das características irrelevantes, permitindo dessa maneira uma simplificação importante da complexidade do problema. Tal abstração foi explicitada com o surgimento dos Tipos Abstratos de Dados, que revolucionaram a programação, marcando desta maneira sua importância dentro da Computação.

Por outro lado, certos problemas são altamente complexos e requerem um tratamento diferenciado. Esses problemas exigem, para sua resolução, um grande conhecimento do universo de discurso. O ponto crítico nesta situação é que o domínio do problema não é exato como poderia ser um domínio matemático, pelo contrário ele inclui geralmente aspectos ambíguos e pouco formais que dificultam seu entendimento. Tal domínio é chamado de senso comum. Uma linha da computação, a Inteligência Artificial (IA), surgiu como uma resposta ao estudo desses problemas. Para [KRA 87], entre outros, as soluções para muitos problemas de I.A. dependem mais da capacidade de adquirir e manipular conhecimento do que de algoritmos de controle sofisticados. Por este motivo,

existem na IA muitos tipos de linguagems que tentam, de diversas maneiras, facillitar a representação de conhecimentos sobre universos de discurso de problemas particulares. São as chamadas Linguagens de Representação de Conhecimento.

Como nas primeiras linguagens de programação, a noção de tipo é implícita nas atuais linguagens de representação de conhecimento, uma vez que tal noção é natural no raciocínio humano e está intimamente ligada ao conceito de abstração.

Este trabalho visa explicitar a noção de tipo subjacente numa linguagem de representação de conhecimento do senso comum. Com esse fim escolheu-se uma linguagem baseada em redes semânticas por considerar esse tipo de linguagem mais flexível na representação de conhecimento e consequentemente mais interessante para a explicitação da noção de tipo.

A linguagem escolhida foi a RECON-II (REdes CONceituais) que está sendo desenvolvida por Oliveira [OLI 89] como uma ferramenta para o estudo da aprendizagem automática no nível do meta-conhecimento. Uma descrição desta linguagem, assim como dos modelos de representação de conhecimento que dão embasamento à RECON-II, é apresentada no capítulo 2.

Como foi dito antes, o objetivo principal do presente trabalho é explicitar a noção de tipos na linguagem RECON-II. Para concretizar isto, foi necessário um estudo semântico prévio que permitisse identificar os tipos semânticos da linguagem. A partir dessa noção semântica foi possível definir a correspondente sintática e finalmente, descrever um Sistema de Tipos para RECON-II.

A explicitação da noção de tipo é feita através da definição de um Sistema de Tipos, que consiste de três elementos. O primeiro é um conjunto de tipos básicos também chamados tipos primitivos. O segundo é um conjunto de construtores de tipos, que permitem obter tipos estruturados a partir dos primitivos. Esses dois elementos determinam o que se conhece como Linguagem de Tipo. O terceiro elemento consiste de um sistema de dedução que relaciona as expressões da linguagem objeto (linguagem de

programação) com as expressões da linguagem de tipo.

Logo, definir um Sistema de Tipos para uma linguagem requer conhecer as expressões significativas que podem ser geradas a partir dos construtores da linguagem, e mais ainda, precisa-se reconhecer quando duas expressões determinadas são iguais, i. é, possuem o mesmo significado.

Assim, para se determinar o Sistema de Tipos de RECON-II foi necessário, primeiro, determinar seus tipos semânticos.

Os tipos semânticos organizam o universo de discurso da linguagem objeto, enquanto que os tipos sintáticos estruturam a própria linguagem objeto. A figura 1.1 mostra os relacionamentos entre linguagem objeto, linguagem de tipos e universo de discurso. As numerações nas setas mostram os passos seguidos para obter o Sistema de Tipos de RECON-II.

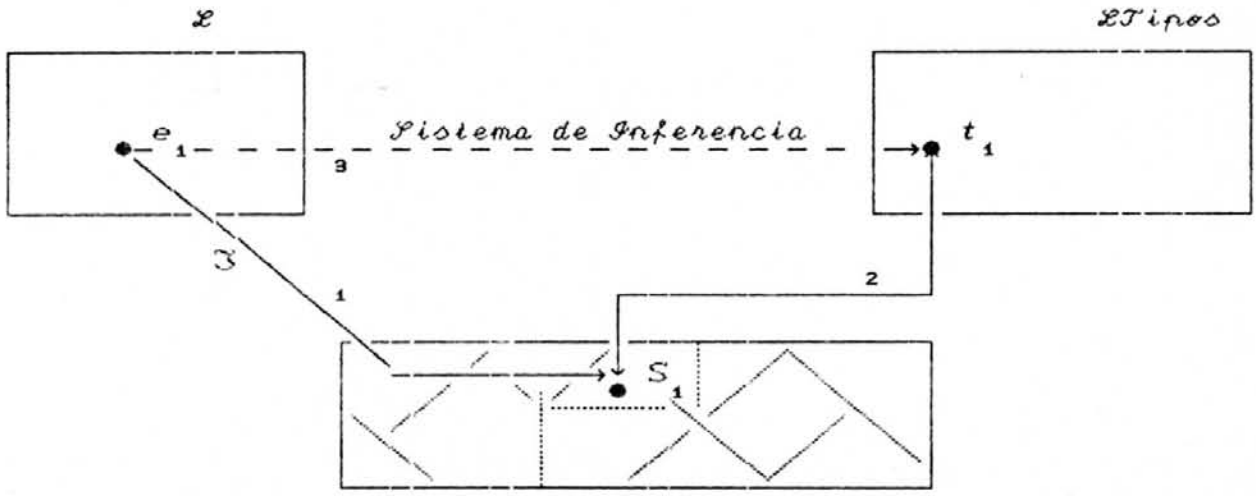


Fig. 1.1 : Esquema do trabalho

Na primeira etapa deste trabalho, a determinação da semântica da linguagem, foi utilizado o método algébrico. Os conceitos básicos relacionados com o método utilizado, são apresentados no capítulo 4, enquanto que a especificação semântica de RECON-II é descrita no capítulo 5.

As duas etapas seguintes, determinação de uma estrutura sintática adequada para os tipos semânticos definidos na etapa anterior e construção do Sistema de Inferência, são descritas no capítulo 6.

Por último, todos os conceitos relacionados com a noção de tipo são tratados no capítulo 3.

Dado que a linguagem RECON-II está atualmente em fase de desenvolvimento, outro objetivo procurado foi colaborar no melhoramento das características e da expressividade da linguagem assim como na determinação de uma semântica para a mesma. Estes objetivos são consequência do objetivo principal.

2 LINGUAGENS DE REPRESENTAÇÃO DO CONHECIMENTO

2.1 Representação de Conhecimento

Na inteligência artificial um ponto de fundamental importância que preocupa seus pesquisadores é o problema da representação do conhecimento. Mais concretamente, o problema de encontrar uma representação adequada para expressar o conhecimento inerente a um domínio de aplicação. As soluções para muitos problemas de I.A. dependem mais da capacidade de adquirir e manipular conhecimento do que de algoritmos de controle sofisticados [KRA 87].

Por *conhecimento* entende-se a soma de percepções de um indivíduo acerca de aspectos de algum universo de discurso num momento determinado [MAT 89], ou seja, tudo o que um sujeito "conhece" acerca de um domínio específico num instante dado. A expressão Universo de Discurso, ou Domínio de Aplicação, refere-se a qualquer parte do mundo real ou não real (mundo possível)

Por *representação* entende-se um conjunto de convenções acerca de como descrever um domínio de aplicação (objetos tangíveis e intangíveis). Diz-se que uma representação é *adequada* quando ela é explícita, completa, concisa, transparente, computável e eficiente do ponto de vista computacional [WIN 84].

Para [WIN 84], numa representação determinada é possível distinguir duas partes: uma sintática e outra semântica. A sintaxe determina os símbolos que podem ser usados e a maneira de combina-los. A semântica associa um *significado* a cada símbolo ou grupos de símbolos.

A *Representação do Conhecimento* trata dos métodos e técnicas utilizadas para incorporar representações de um dado conhecimento, de maneira adequada, nos sistemas de computação. O termo *Base de Conhecimento* é usado para referir-se ao "corpo" do conhecimento explicitado num sistema, e consiste usualmente de uma coleção de *fatos* e *regras* acerca dos *objetos* de interesse no sistema. Usa-se o termo *objeto* para designar às estruturas de

dados na base de conhecimento que denotam entidades no universo de discurso. Um *fato* é uma sentença incondicional expressando algum relacionamento entre as entidades denotadas pelos objetos. [KRA 87]. Uma regra é uma sentença condicional.

Nos primeiros sistemas de IA, no entanto, a representação de conhecimento não foi reconhecida explicitamente como um aspecto importante para a resolução de problemas, estando a ênfase nos algorimtos de controle, embora a maioria dos sistemas incorporassem conhecimento, por exemplo o sistema GPS [NEW 72].

Um dos primeiros formalismos propostos para representar o conhecimento foi a lógica matemática. A utilidade da lógica de primeira ordem na representação de conhecimento surgiu durante a década de 1960. Primeiramente como um resultado de pesquisas na área de provas automáticas de teoremas. Na época muitos pesquisadores se orientaram no estudo do princípio da resolução de Robinson [ROB 65] como uma técnica de inferência, enquanto que outros nos formalismos lógicos, a fim de utilizá-los na representação do conhecimento. Por outro lado, vêm sendo pesquisadas extensões da lógica clássica que superem suas limitações representacionais [GEN 86][CAR 91][PEQ 91][PEQa 91].

Por outro lado, em 1968, Minsky [MIN 68] edita um livro que dá uma argumentação forte para a necessidade de representar o conhecimento para a IA. Esta publicação contém uma série de artigos entre os quais figura o de Quillian [QUI 68] que introduz a noção de rede semântica. Embora existam muitos tipos de redes diferentes, na atualidade, todas consistem de um conjunto de estruturas de dados e um conjunto de procedimentos de inferência que opera sobre essas estruturas. Talvez a rede semântica mais popular seja aquela cuja estrutura de dado é uma hierarquia de nodos conectados por uma relação IS-A [BRA 83].

IS-A é um termo usado para descrever a existência de uma relação de generalização entre os conceitos representados pelos nodos. Com essa estrutura se popularizou também um mecanismo de inferência baseado na noção de hierarquização de conceitos,

chamado *herança de atributos* [MIN 75].

Um outro esquema utilizado para representar conhecimento é o sistema de produções, introduzido no início da década de setenta por Newell [NEW 72]. Os sistemas de produções foram originalmente apresentados como modelos do raciocínio humano. Num sistema de produções, o conhecimento é expresso através de um conjunto de regras, chamadas *regras de produção*. Cada uma destas regras especifica um conjunto de ações que deveriam ser executadas sempre que un conjunto de condições é satisfeito (ver seção 2.2.2).

Em meados da década de setenta, Minsky [MIN 75] introduziu a noção de "frame" como uma forma de representação de conhecimento. Um "frame" é uma estrutura de dados que contém uma agrupamento de fatos acerca de um objeto. O trabalho de Minsky teve uma grande repercussão nos posteriores métodos para representação de conhecimento, dando origem às chamadas representações estruturadas.

Atualmente há um crescente interesse na integração das diferentes metodologias de representação.

Um outro problema enfrentado pela IA na atualidade,é a falta de um embasamento formal para alguns dos esquemas de representação do conhecimento. O ataque a este problema se produz desde várias áreas diferentes. Assim, através da *lógica não monotônica*, tenta-se estender o cálculo de predicados para poder manipular uma variedade maior de fenômenos, conservando, entretanto, o núcleo de sua semântica formal. Por outro lado, as pesquisas sobre redes semânticas tentam formalizá-las, mantendo sua capacidade de estruturação e inferência. As linguagens de representação de conhecimento como KRL, KOALA, RECON-II, etc [BOB 77] [MAT 89] [OLI 90], tentam definir o significado de suas construções através de um intérprete.

Por último, o chamado processo de aquisição de conhecimentos é s dúvida um dos
maiores problemas que enfrenta a IA, o qual está sendo objeto de importantes pesquisas [OLI 91] [WER 91][BRA 91].

2.2 Organização do Conhecimento e Linguagens de Representação

Na seção anterior apresentou-se uma visão simplificada dos principais paradigmas que existem para representação de conhecimento (redes semânticas, sistemas de produções, lógica matemática e frames) e de como surgiram. Certos esquemas de representação enfatizam as características declarativas do conhecimento e são, portanto, apropriados para descrição das partes passivas do domínio de conhecimento; enquanto que outros reforçam os aspectos procedurais do conhecimento. Além disso, existem também os esquemas chamados estruturais que priorizam aspectos de estruturação da representação. Exemplos de representações declarativas são as redes semânticas e a lógica formal. O esquema de regras de produção é considerado procedural e o de frames, estrutural.

Nesta seção serão apresentados detalhes dos dois paradigmas mais relevantes para a comprteensão da linguagem de representação que é objeto de estudo deste trabalho, que são as redes semânticas e os frames.

2.2.1 Redes Semânticas

As Redes Semânticas e suas linguagens [WOO 75] surgiram como um paradigma de representação do conhecimento do senso comum, constituindo uma alternativa em relação às representações em lógica formal e em regras de produção. As linguagens de representação em lógica possuem um alto poder dedutivo; porém, suas construções são semanticamente pobres quando comparadas com a riqueza dos domínios de conhecimento representados em algumas aplicações de Inteligência Artificial. Por outro lado, as linguagens de regras de produção modelam com facilidade conhecimento orientado a procedimentos, mas não são adequadas para representação da estrutura de objetos complexos e suas interrelacões. As linguagens de redes semânticas representam um esforço no sentido de preencher estas lacunas.

Mas, o modelo de redes semânticas não foi o único esforço no sentido de resolver esses problemas. Na área de linguagens de

programação, o paradigma de Programação Orientada a Objetos [TAK 89] procura representações computacionais mais próximas da visão que o pensamento do senso comum possui da realidade, ou seja, representações baseadas em objetos, suas interrelações e comportamentos, organizados em sistemas taxonômicos. É interessante observar como, em áreas diferentes (IA, Linguagens de Programação e Bancos de Dados), as soluções adotadas foram semelhantes.

Uma rede semântica pode ser vista como um grafo onde os nodos representam objetos ou conceitos. Um nodo pode estar conectado a qualquer outro através de um arco dirigido. Esse arco denota uma relação entre esses objetos. Os arcos são nomeados devido à necessidade de modelar formas diversas de relações entre os nodos. Dentre essas formas, a mais importante é a relação de *taxonomia* ou *hierarquia*, que relaciona indivíduos com as classes a que pertencem. É esta relação que permite um tipo fundamental de raciocínio em redes semânticas: a dedução da estrutura e comportamento dos indivíduos através da herança de atributos.

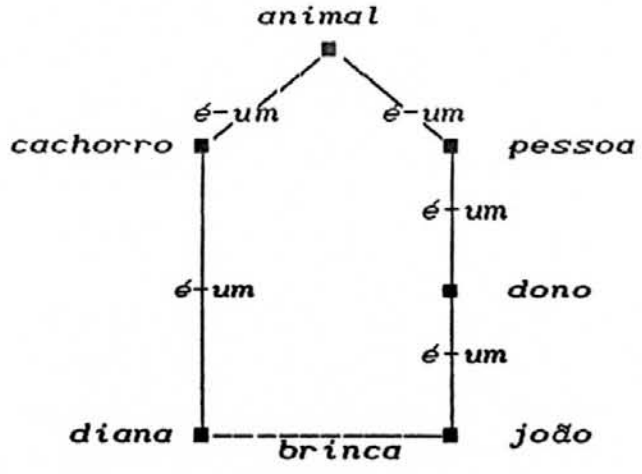


Fig. 2.1 : Exemplo de Rede Semântica

Na figura 2.1 apresenta-se um exemplo de rede semântica. A rede tenta modelar o fato de a cachorra Diana brincar com seu dono João.

A partir da figura é possível notar a existência de dois tipos de nodos: os nodos individuais e os genéricos. Os nodos

genéricos representam uma classe ou categoria de objetos (*pessoa, cachorra, dono, etc.*) enquanto que os individuais são descrições ou afirmações a respeito de uma instáncia individual de um objeto (*joão e diana*).

Uma das maiores vantagens deste esquema de representação é que toda a informação acerca de um conceito determinado encontra-se integrada em torno do nodo respectivo e é diretamente acessada a partir dele.

Uma outra vantagem é a de permitir a representação de relacionamentos entre objetos de maneira simples e explícita. E é a partir destas representações que surge o poder de estruturação e de dedução das redes semânticas. Embora existam muitos modelos de redes há algum consenso quanto aos tipos de relacionamentos básicos que toda rede deve possuir. Eles são três: *generalização, classificação e agregação* [MAT 89] [TAK 89].

A relação de *generalização* aplica-se entre objetos genéricos, estabelecendo assim uma hierarquia de objetos. Considere-se por exemplo, na figura 2.1, a relação entre *pessoa* e *animal*. Essa relação determina que *animal* é um conceito mais genérico que *pessoa*. A semântica intuitiva deste relacionamento é que os conceitos menos genéricos (pessoa) *herdam* as características dos mais genéricos (animal). O nome desta relação numa rede é geralmente: *é-um*.

No relacionamento chamado de *classificação* participam objetos individuais e genéricos. Continuando com o exemplo da figura 2.1, pode-se notar que a relação entre *diana* e *cachorro* é de classificação, assim como a de *joão* e *dono*. O significado intuitivo desta relação é basicamente o mesmo que o dado à relação de generalização, com a exceção de que aqui só se tem um nodo genérico a partir do qual o nodo individual, que é instáncia daquele, *herda* suas propriedades. No exemplo, *diana* herda todas as características de *cachorro* e *joão* de *dono*. Os nomes dados a este relacionamento numa rede semântica geralmente são: *é-um, membro-de* ou *instância-de*. Mas, usualmente, o nome *é-um* é reservado para indicar relações de generalização. Apesar disto,

no exemplo da figura 2.1, foi utilizado o mesmo rótulo (é-um) tanto para indicar as relações de generalização como as de classificação, porque informalmente ambas relações seconfundem. Este fato foi feito proprositalmente visando ajudar no entendimento preliminar do exemplo ,no entanto cabe, agora, clarificar o mesmo diferenciando perfeitamente cada relacionamento (figura 2.2).

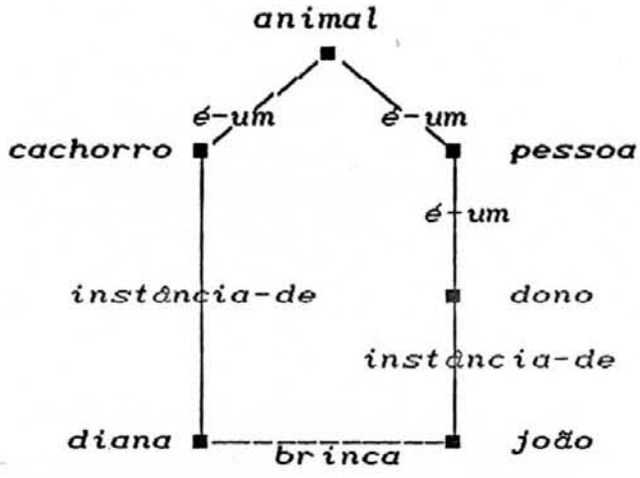


Fig. 2.2 : Exemplo de Rede Semântica

Por último, a relação de *agregação* é usada para relacionar um objeto aos seus componentes. Por exemplo, as partes de um automóvel (objeto genérico) são: rodas , motor, carroceria, etc.

O maior problema que enfrenta este esquema de representação é a falta de uma semântica formal e uma terminologia padronizada. Nota-se esse problema, por exemplo, nos muitos significados existentes para o relacionamento *é-um* (ver [BRA 83]). Isto deve-se, em parte, ao fato de que as redes semânticas tem sido usadas para a representação de conhecimento a partir de fontes muito diferentes,por exemplo: na representação de fórmulas lógicas, para expressar o significado de sentenças em linguagem natural, etc.

Uma outra desvantagem que surge devido à falta de semântica formal, é a dificultade em verificar a correção do processo de inferência [MAT 89].

A pesar da falta de uma semântica formal, alguns esforços foram realizados, como o caso do modelo matemático baseado na

teoría de grafos dado por João Gluz [GLU 91] para uma linguagem de redes semânticas desenvolvida por Nelson Mattos [MAT 89]. Um outro intento é o presente trabalho que visa fornecer um sistema de tipos para a linguagem de representação estruturada de conhecimentos RECON-II [OLI 90] que, entre outras coisas, exigirá definir um modelo matemático para a mesma.

2.2.2. Frames

Este esquema de representação deve sua origem a Minsky [MIN 75] que tentava através dele conseguir representar a percepção visual, diálogos em linguagem natural e outros comportamentos complexos.

Segundo Barr [BAR 86] existe evidência psicológica de que as pessoas usam parte do conhecimento de experiências prévias na forma conceitual de *protótipos*. Por exemplo, quando alguém fala sobre um livro as pessoas imediatamente formam uma imagem de um livro que condiz com o estereótipo do livro, i. é, imagina-se provavelmente um objeto retangular, com capas, um conjunto de folhas numeradas e escritas, um índice, um ou mais autores, um editorial, em fim uma série de expectativas a respeito desse objeto, geradas pela experiência. Assim, pode-se dizer que o cérebro humano está menos relacionado com definições exatas e exaustivas dos objetos e sim com conjuntos de propriedades *essenciais* associadas a objetos que são *típicos* em sua classe. Tais objetos denominam-se *objetos prototípicos*.

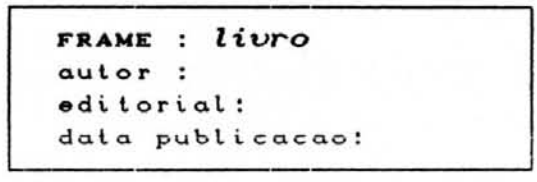
Minsky define um "frame" como uma estrutura de dados, que ele chama de objetos prototípicos, para representar situações estereotipadas [MIN 75]. Cada "frame" é identificado por seu nome e composto por um agregado de "slots" (o conteúdo de um "frame") em número variável. Cada "slot" (ou campo) é composto por valores e por ações. Os "slots" são os elementos que permitem definir relacionamentos entre "frames", i. é, estruturar os "frames" em hierarquias baseadeas na relação de generalização (tratada na seção 2.2.1).

Os "frames" foram concebidos como uma variedade restrita de redes semânticas em que conceitos são agrupados em hierarquias de generalização/especialização e cada conceito é representado como

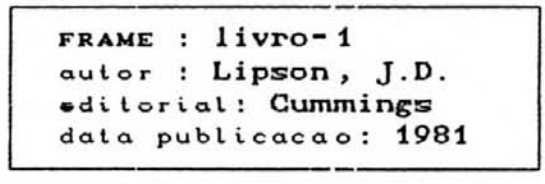
um agregado de atributos estáticos (valores) e dinâmicos (operações) [TAK 90].

Por isto, os "frames" além de gozar das vantagens das redes semânticas, têm uma outra vantagem que é a de fornecer uma representação estrutural concisa.

Na figura 2.3(a) apresenta-se um exemplo de "frame", e na 2.3(b) uma instância particular do mesmo com seus "slots" preenchidos.



(a)



(b)

Fig. 2.3 : Exemplos de "Frames"

Analisando a figura 2.3 observa-se que os "frames" suportam os mesmos relacionamentos básicos que as redes semânticas, embora, alguns de maneira implícita. Como o relacionamento de agregação que pode ser definido atraves dos "slots", i. é, o conteúdo de um "slot" pode ser um valor simples mas também outro "frame". Quanto à classificação, ela resulta obvia pela simples observação do exemplo apresentado acima.

Por último, a generalização é obtida no modelo da mesma maneira que a classificação. Para exemplificar este ponto considere-se a figura 2.4.

Em resumo os sistemas de "frames" tentam raciocinar sobre classes de objetos utilizando representações prototípicas de conhecimento que valem para a maioria dos casos. Mas, por causa

das exceções, os "frames" incorporam mecanismos para tal tratamento (campos com valores "default", com valores limites, etc.). É justamente por causa disto que os "frames" são mais criticados pois não oferecem uniformidade na representação. Além desta desvantagem, encontra-se nos "frames" as mesmas desvantagens mencionadas nas redes semânticas, i. é, a falta de uma semântica formal e de uma notação padronizada.

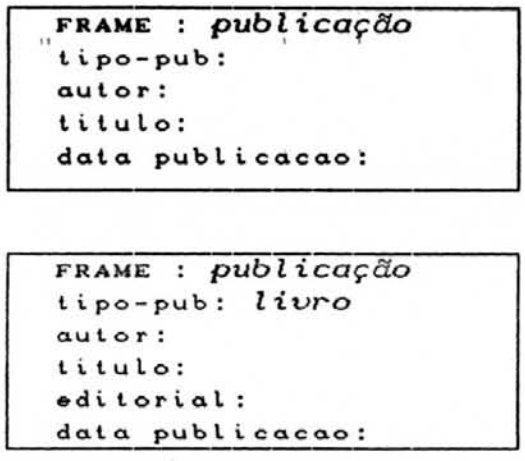


Fig. 2.4 : Outros exemplos de "Frames"

2.3 Uma Linguagem de Representação do Conhecimento: RECON-II

RECON-II (REdes CONceituais) é uma linguagem de representação de conhecimento baseada numa combinação dos esquemas de redes semânticas e "frames", em desenvolvimento no CPGCC/UFRGS [OLI 90].

Antes de iniciar a descrição propriamente dita da linguagem RECON-II é necessário aprofundar algumas características, já mencionadas nas seções 2.2.1 e 2.2.2, do esquema utilizado para a representação do conhecimento.

2.3.1 Redes Conceituais

Uma *rede conceitual* (RC) é um grafo orientado onde os nodos representam *conceitos* e os arcos *relações* entre conceitos.

Os nodos têm uma estrutura interna associada que consiste num conjunto de *atributos*.

Entende-se por *conceito* todo objeto tangível ou intangível assim como, também, qualquer situação do universo de discurso. Um conceito pode possuir atributos que o determinam, i. é, um conceito diferencia-se de outro através de sua *descrição* (definição dos atributos).

Um *atributo*, refere-se, geralmente a um valor tendo o mesmo papel que os "slots" na determinação de um conceito.

2.3.2 Relacionamentos nas RC

Na seção 2.2.1 mencionou-se brevemente os relacionamentos chamados básicos para uma rede semântica e posteriormente viu-se que esses relacionamentos estão presentes também no modelo de "frames".

O objetivo agora é mostrar como eles surgem naturalmente a partir de um processo de abstração e aprofundar mais no significado que cada relacionamento tem no modelo de rede conceitual.

Antes de ser possível qualquer representação sobre um domínio de aplicação é necessário considerar um processo prévio: a *abstração*, que diz respeito às operações mentais executadas ao observar-se um domínio e capturar sua estrutura num modelo de representação [TAK 90]. Isto é válido para qualquer tipo de domínio e qualquer tipo de esquema de representação utilizado. Afinal de contas uma representação, como já foi dito, não é mais que um conjunto de convenções adotadas para falar sobre algo (ver seção 2.1).

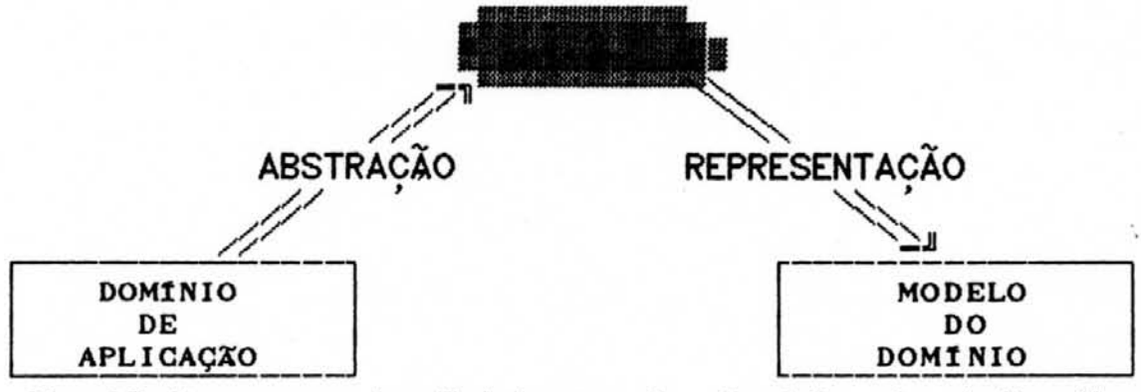


Fig. 2.5: Processo de Modelagem do Domínio de Aplicação

A figura 2.5 ilustra o papel da abstração e representação numa modelagem e o fato de que não existe abstração sem representação e vice-versa.

Uma vez que a abstração é o processo mental de identificar as qualidades ou propriedades importantes do fenômeno sendo modelado, é evidente que o resultado de tal processo depende não tanto do fenômeno observado quanto do interesse do observador. Reconhecido este aspecto da abstração como regulador da atenção de um indivíduo é necessário determinar quais são as operações mais importantes para a modelagem de universos. Pesquisas na linha de IA e também de Bancos de Dados identificaram três operações básicas:classificação, generalização e agregação, fato que se reflitiu nas linguagens de representação estruturada de conhecimentos, como referido nas seções 2.2.1. e 2.2.2.

A utilização dessas operações num universo de discurso determinado e com um esquema de representação como o de redes conceituais dá como resultado uma *hierarquia de abstrações*. A hierarquia é composta por categorias ou classes, da mais genérica à mais particular (interconectadas por arcos denotando especialização/generalização), por instâncias dessas categorias (interconectando uma instância a uma categoria por arcos denotando instanciação/classificação) e por informações de agregação entre categorias (através da estrutura interna de cada nodo na rede conceitual)

As redes conceituais, da mesma forma que as redes semânticas e os "frames", conseguem criar, através do uso das relações de generalização e classificação, uma *taxonomia* de conceitos. A partir dessa taxonomia e das propriedades estruturais dos nodos que constituem as RC, é possível realizar inferências sobre as redes. Um dos métodos de inferência é o referido anteriormente: *herança*. Outro método é o que usa as informações internas dos conceitos, como por exemplo valores possíveis, valores limites, etc. Existe um terceiro chamado *reconhecimento de padrões* que consiste em derivar informação de uma rede através da comparação com outra estrutura de rede, mas incompleta. Por exemplo, no caso

da figura 2.2 a pergunta "quem *brinca* com *diana*?" é respondida comparando a rede da figura com a seguinte rede:

diana ⟶ ?

o nodo que corresponde a "?" nesse "esquema" de rede é a resposta à pergunta formulada.

2.3.3 A linguagem RECON-II

RECON-II é uma linguagem de representação de conhecimento baseada no modelo de Redes Conceituais. Através de RECON-II é possível, então, modelar o domínio de aplicação num modelo que inclui uma ou mais Redes Conceituais.

A definição de uma RC consiste de um identificador da rede mais um conjunto de declarações como aparece na figura 2.6

As declarações que aparecem no corpo da rede consistem na definição dos conceitos e relacionamentos que a compoem, e a definição de abstrações-λ.

```
{ conceptual-net  : identifier  }
      declarations
{ end }
```

Fig. 2.6 : Definição de uma RC

Define-se os conceitos em RECON-II através da especificação de sua estrutura interna, a qual está associada a um nome que identifica o conceito. A estrutura interna de um conceito é um conjunto de propriedades nomeadas. Tais propriedades são representadas por um atributo composto de um par (nome : valor). Os valores dos atributos podem ser *atômicos* (numéricos, "strings", etc.) ou *compostos*, i. é, ter um valor que por sua vez tem atributos e assim por diante.

Uma declaração de conceito tem a forma geral:

```
{ concept  : identifier
    attribute-name1 : value1 ;
    attribute-name2 : value2 ;
        :
    attribute-namen : valuen ;   }
```

Fig. 2.7 : Declaração de Conceito

Para exemplificar, considere-se o livro apresentado na figura 2.3 da seção de "frames". Utilizando RECON-II o conceito livro-1 ficaria expresso como:

```
{ concept : livro-1
    autor : Lipson, J. D. ;
    editorial : Cummings ;
    data-public.: 1981 ;        }
```

As relações em RECON-II são representadas, como já foi mencionado, pelos arcos das redes conceituais e definem-se da seguinte maneira:

```
{ relation : identifier
    properties :
    [ p1, p2, ... ,pn ] ;
    elements :
              [ x1, y1 ] ,
              [ x2, y2 ] ,
                  :
              [ xk, yk ] ,      }
```

Fig. 2.8 : Declaração de uma Relação

onde os pares $[x_i, y_i]$ denotam componentes dos arcos da relação; x_i e y_i são os nomes dos nodos (conceitos) origem e destino respectivamente. Os p_j são propriedades da relação (se verá mais adiante como se definem as mesmas)

Uma relação pode ser definida extensionalmente através da lista de pares, ou combinando-os intensionalmente com as propriedades.

Existem três relações pré-definidas em RECON-II correspondente aos três tipos de abstrações mais típicas em modelos semânticos que foram definidas na seção 2.3.1. Elas são:

- *especialização/generalização.*
- *classificação/instanciação.*
- *agregação/decomposição.*

Na figura 2.9 apresenta-se uma definição de relação de especialização para o exemplo dos livros dado antes.

```
{ relation : especialização
properties :
        [transitiva, reflexiva ] ;
elements :
        [publicação, livro ] ;
        [livro, livro-1 ] ;
}
```

Fig. 2.9 : Exemplo de Relação

Um outro elemento importante de RECON-II são as funções para descrever processos e transformações. A forma geral de uma função é a seguinte:

```
{ function : identifier
    domains :
    { dom1, dom2, ... ,domn → codomain }
    body : lambda { arg1, arg2, ... ,argn)
                [ ( operator expressio ]
    < attributes >
```

Fig 2.10 : Declaração de uma Função

Duas funções, em particular, são interessantes : transfomações de redes e predicados. Estes últimos são funções do tipo *D → Bool*, onde *D* é qualquer domínio e *Bool = {true, false}*. As transformações de redes são funções do tipo *rede-conceitual → rede-conceitual*. Desde que redes conceituais representam bases de conhecimento, as transformações sobre redes podem ser usadas para

modelar mecanismos de inferência em geral e processos de aprendizagem (transformações de conhecimento), em particular [OLI 91].

Por último, RECON-II permite a definição de operações sobre conceitos através de um conjunto de operadores pré-definidos: intersection, union, minus, project, selet e aplicação de funções.

Mais detalhe da sintaxe de RECON-II pode ser encontrado no Anexo A. Cabe mencionar, não obstante, que RECON-II ainda está em desenvolvimento e portanto sujeita a modificações, pelo qual este trabalho desenvolver-se-á sobre um sub-conjunto estável da mesma.

3 OS TIPOS NAS LINGUAGENS DE PROGRAMAÇÃO

3.1 A noção de tipo

A noção de tipo é muito familiar para matemáticos, lógicos e programadores e surge como resposta à necessidade de organizar seus universos de discurso ($\mathcal{U}$).

Costa em [COS 90] expressa: "...a noção de tipo visa organizar tanto $\mathcal{U}$ quanto $\mathcal{L}$ (linguagem), pela aglutinação de elementos de $\mathcal{U}$ e de expressões de $\mathcal{L}$ em classes ... de entidades afins. Embora o critério de afinidade seja variável, a classificação em $\mathcal{U}$ é sempre de cunho ontológico (categorias ontológicas) ... a classificação em $\mathcal{L}$ sempre é de cunho sintático (categorias sintáticas, classes gramaticais) e de cunho semântico (categorias semânticas, tipos propriamente ditos). Nas teorias semânticas em que o espaço de significados $\mathcal{SI} \equiv \mathcal{U}$, os tipos semânticos de $\mathcal{L}$ em $\mathcal{SI}$ se confundem com as categorias ontológicas de $\mathcal{U}$ ..."

Especificamente em computação, o conceito de tipo adquiriu grande relevância, a partir da última década, na definição de linguagens mais rigorosas. Esta situação origina-se não somente no fato de que os tipos permitem caracterizar valores, organizando o universo de discurso, senão também, devido a que impedem a geração de expressões não significativas nas linguagens.

Um universo de valores onde objetos são indistinguíveis, do ponto de vista de suas caracterísitcas gerais, i.é, todos os objetos ou valores são tratados da mesma maneira e têm o mesmo comportamento, é chamado *universo não tipado*, e pode-se dizer que possui portanto um único tipo.

Em computação, por exemplo, pode-se considerar como um domínio não tipado a memória do computador, a qual está composta de um conjunto de seqüências de bits de tamanho fixo. O significado de um fragmento de memória é determinado pela interpretação externa que se dá a seu conteúdo.

Mas, quando se trabalha com um universo não tipado surge naturalmente a necessidade de *organizá-lo* em diferentes formas, para poder distinguir grupos de objetos com características e/ou comportamento comuns. Tal categorização leva à definição de um Sistema de Tipos (ver seção 3.3) [CAR 85].

Assim um universo não tipado decompõe-se naturalmente em subconjuntos com comportamento uniforme, os tipos, e denomina-se *universo tipado*.

Continuando com o exemplo anterior, em certos casos pode ser necessário pensar a memória como dividida em dois conjuntos de objetos, por exemplo, entre dados e operações, diferenciando assim o comportamento e a utilidade dessas categorias.

Dado que os *tipos* surgem informalmente em qualquer domínio, para *categorizar* os objetos segundo seu *uso* e *comportamento*, é razoável afirmar que tal noção é intrínseca ao raciocínio humano, na medida em que os seres humanos tendem a "classificar" os objetos como parte do processo de resolução de problemas.

Por outro lado, os tipos também permitem realizar uma abstração de $\mathcal{U}$, "esquecendo" as características irrelevantes dos objetos e concentrando-se nos aspectos essenciais, conseguindo, assim, uma simplificação importante da complexidade do problema.

Do ponto de vista computacional, um tipo pode ser pensado como um *conjunto de camadas* que protege uma representação interna não tipada, desconhecida, de um uso inadequado. Assim, com o ocultamento de tal informação, restringe-se a maneira em que os diversos objetos podem interagir entre si.

Evidentemente, a escolha da representação interna do tipo é feita considerando as operações que serão executadas sobre ele visando respeitar suas propriedades.

Os tipos, na área de computação, além de categorizar o domínio de valores de uma linguagem e assegurar a construção de expressões corretas, são importantes porque garantem a evolução ordenada dos grandes sistemas de software [CAR 89]. Essa garantia existe porque eles fornecem uma maneira de controlar o desenvolvimento de tais sistemas, através da verificação parcial

dos programas que os compõem. Tal verificação, chamada *verificação de tipos* (ver seção 3.5), é mecânica e assegura que certa classe de erros não podem aparecer durante a execução dos programas. Logo, os tipos incrementam a confiabilidade[1] dos sistemas de software [CAR 89]. Por isto a informação de tipo é vista como uma especificação parcial do programa.

Outro motivo pelo qual o conceito de tipo é fundamental nas linguagens de programação de alto nível, é que ele determina a representação dos valores, sua forma de armazenamento dentro do computador e a maneira como os operadores serão interpretados.

Segundo [HOA 72] as características mais importantes do conceito de tipo são:

- Um tipo determina a classe de valores que podem ser assumidos por uma expressão que possui tal tipo.
- Cada valor ou elemento do universo de discurso tem pelo menos um tipo.
- O tipo de uma expressão pode ser deduzido do contexto sem conhecimento do valor que ela denota em tempo de execução.
- As propriedades dos valores de um tipo e das operações primitivas definidas sobre ele são especificadas por meio de axiomas.
- A informação de tipo numa linguagem de programação de alto nível é usada com dois fins: para detectar construções sem sentido no programa e para determinar o método de representação e manipulação do dado no computador.

Um tipo é uma caracterização precisa de propriedades comportamentais ou estruturais que um conjunto de objetos compartilham. Uma instância de um tipo é um objeto que tem as propriedades características do tipo.

---

(1) Se diz que um sistema é confiável quando, primeiro, não produz erros frequentemente e, segundo, quando as mudanças nele não são muito custosas.

Como já foi mencionado, um tipo determina um conjunto de valores. O número de elementos que compõe esse conjunto chama-se *cardinalidade do tipo*.

A cardinalidade pode ser finita ou infinita. Se é finita, então, pode ser computada por alguma fórmula simples. Senão ela é infinita enumerável, já que cada valor do tipo deve ser construível num número finito de operações computáveis e representável numa quantidade também finita de armazenamento.

Do ponto de vista implementativo, uma linguagem de programação de alto nível tem um conjunto de tipos básicos chamados *tipos primitivos*, os quais podem ser pensados como sendo definidos por axiomas, e tem, também, um conjunto de construtores que permite obter tipos chamados *estruturados* ou *compostos* a partir dos primitivos.

O aspecto mais importante (do ponto de vista prático) é a forma com que os dados podem ser manipulados e o alcance dos operadores básicos aplicáveis.

Assim, cada tipo, está associado a um conjunto de *operadores básicos*[1] que asegura que qualquer outra operação requerida pode ser definida em termos deles.

O conceito de tipo também pode ser pensado como um predicado tal que os objetos que pertencem a um tipo o "satisfazem" (no sentido lógico).

É possível estabelecer formalmente que existe uma correspondência entre a noção de *tipos-como-conjuntos-de-valores* e a de *tipos-como-fórmulas*, já que um predicado da lógica define o conjunto daqueles elementos que o fazem verdadeiro. E inversamente, um conjunto pode ser definido por compreensão através de alguma propriedade expressa como uma fórmula lógica.

---

(1) Na prática, um operador é básico quando sua implementação depende muito da representação do dado.

Além dessas semânticas para a noção de tipo, existem outras como *tipos-como-valores*, *tipos-como-conjunto-de- operações*, que não serão tratadas no presente trabalho. Um estudo informal sobre as diferentes linguagens e os quatro tipo de noções (conjuntos, predicados, valores e operações) pode ser visto em [PAS 90].

3.2 Evolução dos tipos nas linguagens de programação

Esta seção não pretende ser uma resenha histórica exata nem completa da evolução do conceito de tipo nas linguagens de programação, mas sim dar uma aproximação dessa evolução em termos globais

No início, a computação era pensada como sendo essencialmente numérica e os valores tinham um único tipo aritmético.

Não obstante, em FORTRAN se considerou conveniente distinguir entre número inteiro e de ponto flutuante por questões de representação e eficiência. Esta distinção foi feita pela primeira letra do nome das variáveis.

Posteriormente com o surgimento de ALGOL 60, a diferenciação é explícita através de declarações de identificadores. ALGOL 60 foi a primeira linguagem que teve uma noção explícita de tipo.

A partir dela surgiram outras linguagens que enriqueceram a noção de tipo. Entre elas estão PL/1, PASCAL, ALGOL 68 e SIMULA só para citar as mais importantes.

PASCAL fornece uma clara extensão de tipos para arranjos, registros e ponteiros, como também tipos definidos pelo usuário. Não obstante, PASCAL não define equivalência de tipos. Tem, além disso, problemas de granularidade de tipos já que, por exemplo, a noção de tipo de arranjos inclui os limites dos mesmos e isto é demasiado restritivo, porque procedimentos que operam uniformente sobre arranjos de diferentes dimensões não podem ser definidos.

ALGOL 68 é mais rigoroso na noção de tipo. Define uma equivalência estrutural de tipo e considera os procedimentos como

valores de primeira classe. Inclui como tipos primitivos: *int*, *real*, *char*, *bool*, *string*, *bits*, *bytes*, *format*, e *file*; como construtores de tipos: *array*, *struct*, *proc*, *union* e *ref*, para tipos de arranjos, registros, procedimentos, uniões, e tipos ponteiros, respectivamente.

As linguagens citadas acima tem também definidas regras de coerção[1] de tipos. Nelas a verificação de tipos é decidível mas o algoritmo de verificação de tipos é tão complexo que nem sempre podem ser verificadas algumas questões de equivalência de tipos e coerção.

Com o aparecimento de SIMULA, primeira linguagem orientada a objeto, a noção de tipo incorpora a idéia de classe cujas instâncias podem ser atribuídas como valores persistentes[2] às variáveis de classe.

Os procedimentos e declarações de dados de uma classe SIMULA são sua interface e são acessíveis ao usuário.

As subclasses SIMULA herdam as entidades declaradas na interface da classe e podem definir operações adicionais e dados que especializam seu comportamento.

As instâncias de uma classe SIMULA são análogas às abstrações de dados, por terem uma interface declarativa e um estado que persiste entre as invocações das operações, mas falta o mecanismo de ocultamento da informação daquelas.

Linguagens posteriores a SIMULA, tais como SMALTALK e LOOPS, combinam os conceitos de classe com uma noção de ocultamento da informação.

Segundo Cardelli, uma linguagem orientada a objetos é uma extensão de uma orientada a procedimentos que inclui as noções de

---

(1) Coerção de tipo é uma operação sintática que consiste em converter o tipo de um valor determinado em outro tipo.

(2) Persistência é o tempo de vida de um objeto num sistema determinado. Se diz que um valor é persistente quando ele permanece acessível após a execução do procedimento que o gera ou atualiza.

abstração de tipos de dados e herança [CAR 85].

Assim, uma linguagem é orientada a objeto quando satisfaz:

- Suporta objetos que são abstrações de dados com uma interface de operações nomeadas e um estado local oculto
- Todo objeto tem ao menos um tipo
- Os tipos podem herdar atributos dos supertipos.

A segunda exigência é aquela que assegura que todos os objetos sejam de primeira classe e assim possam ser utilizados tanto como estruturas de dados como para representar computações.

MODULA-2 foi a primeira linguagem a usar modularização como o maior princípio de estruturação. As interfaces tipadas especificam os tipos e as operações definidas dentro do módulo, e podem ser especificadas separadamente da sua implementação.

ML foi a linguagem que introduziu a noção de polimorfismo paramétrico (ver seção 3.6.1). Os tipos em ML podem conter variáveis de tipos que são instanciadas para tipos diferentes dependendo do contexto. Logo, é possível especificar tipos parcialmente e escrever programas que podem ser usados sobre todas as instâncias de tais tipos.

Em ML, uma maneira de especificar parcialmente os tipos é omitindo as declarações de tipos. Assim, os tipos mais gerais das expressões serão inferidos automáticamente pelo sistema de inferência de tipos do compilador.

ADA é uma linguagem de programação orientada a procedimentos que suporta o conceito de módulos, e o de abstração de dados através da estrutura *packages*.

Sua noção de tipos é relativamente fraca, excluindo procedimentos e *packages* do domínio dos objetos tipados, i.é, tanto procedimentos quanto *packages* não são tratados como objetos com tipos.

Por outro lado, a relação de equivalência escolhida em ADA é a equivalência por nome, que é uma noção mais fraca que a equivalência estrutural utilizada em ALGOL 68.

Além disso, apresenta algumas restrições nas coerções

implícitas diminuindo sua capacidade de prover operações polimórficas.

Os *packages* em ADA tem uma especificação de interface de componentes nomeados que podem ser variáveis, procedimentos, exceções e tipos. E além disso, podem ter um estado local oculto.

Eles são similares aos registros porque possuem componentes nomeados, mas diferem deles, porque os componentes dos registros devem ser valores tipados enquanto que os *packages* podem ter como componentes procedimentos, exceções, tipos e outras entidades nomeadas.

Como os *packages* não têm tipos, i. é, não são objetos computáveis, não podem ser passados como parâmetros, nem ser componentes de estruturas. A diferença em relação aos registros (instância de classes) em SIMULA e outras linguagens orientadas a objetos reside no fato de que tais registros são objetos de primeira classe, enquanto que os *packages* de ADA são objetos de segunda classe.

3.3 Sistema de Tipos

Foi apresentada nas seções prévias a importância dos tipos nas linguagens de programação e sua evolução dentro delas.

Mas, antes de definir o que se entende por um sistema de tipos, é necessário observar a diferença entre linguagens atipadas e linguagens tipadas e entre estas e as linguagens de tipo.

Uma linguagem onde todas suas expressões pertencem a um único tipo é chamada *linguagem atipada* ou *não tipada*.

Inversamente as linguagens que incorporam a idéia de universo de tipos são chamadas *linguagens tipadas*, mas dependendo da maneira como tal noção é introduzida se tem diferentes classes de linguagens tipadas.

Reynolds em [REY 85] define as *linguagens tipadas* como sendo aquelas que satisfazem dois requisitos:

- Cada expressão da linguagem, dada uma atribuição de tipos particular (ver seção 3.5), tem no máximo um tipo. Isto é,

existe uma função parcial chamada *função de tipagem* que mapeia uma expressão ℵ e uma atribuição de tipos Π no tipo de ℵ com Π.

- O tipo de qualquer expressão, dada uma atribuição de tipos particular, é função apenas dos tipos de suas sub-expressões imediatas com as suas atribuições de tipos respectivas (possivelmente diferentes).

A partir desta definição as linguagens tipadas podem-se dividir em *linguagens explicitamente tipadas*, que são aquelas onde os tipos devem aparecer explicitamente nas expressões, por exemplo PASCAL, e as *implicitamente tipadas*, onde os tipos estão implícitos e são inferidos do contexto, como no caso de ML. Existem também linguagens onde as duas noções são combinadas para dar maior flexibilidade. Exemplo deste último tipo de linguagem é QUEST desenvolvida por Cardelli [CAR 89] [PAS 90].

Por outro lado, uma linguagem tipada cujas expressões têm consistência de tipos, i. é, toda expressão tem algum tipo (conhecido ou não), é chamada *fortemente tipada* [CAR 85].

Uma outra categoria são as linguagens onde o tipo de cada expressão pode ser determinado estáticamente, i. é, todos os erros de tipo podem ser detectados em tempo de compilação. Elas recebem o nome de *linguagens estaticamente tipadas* [CAR 85] [ATK 87].

Logo, é possível deduzir que toda linguagem estaticamente tipada é fortemente tipada, mas não o inverso.

A exigência de que a linguagem seja estaticamente tipada garante que os programas compilados não terão erros de tipo na execução. Esta é uma exigência muito restritiva, já que exige que as variáveis e expressões do programa tenham assumido um tipo determinado em tempo de compilação. Por isto, é desejável ter uma linguagem com tipagem forte, porque esta caraterística outorga uma certa flexibilidade na construção das suas expressões, permitindo atribuições de tipo em tempo de execução. O grau de tal flexibilidade depende da capacidade do sistema de tipos e em particular do sistema de inferência do mesmo.

Com a tipagem forte a programação ganha em confiabilidade, já que por um lado, permite um desenvolvimento sistemático dos programas, o qual obviamente diminui a quantidade de erros que o programador comete nesse processo.E por outro lado, a existência de um algoritmo de decisão para verificação da boa tipagem dos programas permite detectar certa classe de erros estaticamente [MIT 78].

Em outro contexto, as linguagens são classificadas em polimórficas ou monomórficas segundo suportem ou não o conceito de polimorfismo. (ver seção 3.6.1).

Finalmente, com respeito à semântica dos programas sintáticamente válidos, Reynolds diz que o significado deles, numa linguagem tipada, não deveria depender da representação usada para implementar os tipos primitivos [REY 74].

Esta propriedade de *independência da representação* deve valer tanto para os tipos definidos pelo usuário como para os tipos primitivos.

A diferença fundamental entre as linguagens tipadas e as linguagens de tipo reside na maneira como elas tratam os tipos.

Nas linguagens tipadas, os tipos são propriedades das expressões da linguagem que podem ser inferidos, e geralmente não aparecem em tempo de execução, i. é, não são valores computáveis; as expressões denotam os verdadeiros objetos de computação da linguagem.

Contrariamente nas linguagens de tipos, os elementos do universo de discurso são os próprios tipos.

Em geral as linguagens tipadas incluem, como um sub-conjunto, uma linguagem de tipos.

O relacionamento entre as linguagens tipadas e as de tipo é dado através do sistema de tipos.

Por um *Sistema de Tipos* de uma linguagem, deve-se entender a maquinaria formal pela qual se associam tipos a expressões dessa linguagem [COS 90].

Um *Sistema de Tipos* é composto por :

- um conjunto de tipos básicos.
- um conjunto de construtores de tipos para construir novos tipos a partir dos básicos.
- um sistema de dedução de tipos para inferir os tipos das expressões da linguagem. (ver seção 3.5).

Os dois primeiros elementos do Sistema de Tipos determinam o que se denominou anteriormente de Linguagem de Tipo. Enquanto que o último elemento, o Sistema de Inferência de Tipos, é um conjuntos de regras que relacionam cada expressão da linguagem de programação com uma expressão da linguagem de tipo.

A utilidade do *Sistema de Tipos* reside não só na representação do conjunto de tipos, i. é, na possibilidade de particionar o segundo algum critério pré-definido, mas também na capacidade de expressar os diferentes relacionamentos existentes entre eles [CAR 85]. Essa capacidade implica também uma aptidão para executar computações e determinar se os tipos satisfazem os relacionamentos desejados.

Do ponto de vista computacional um *Sistema de Tipos* pode ser pensado como um conjunto de restrições impostas sobre uma linguagem de programação.

Esta visão é muito interessante, já que através da definição do sistema de tipo é possível excluir da linguagem determinadas expressões sintáticamente corretas mas sem significado computacional [CAR 89].

Do ponto de vista algébrico, um Sistema de Tipos pode ser visto como uma família de conjuntos (tipos) e uma coleção de funções sobre eles (regras de inferência).

É interessante destacar que, dependendo da linguagem de programação, o *Sistema de Tipos* pode ser monomórfico ou polimórfico.

Um Sistema é monomórfico quando cada valor denotado pelas expressões da linguagem, pertence no máximo a um tipo, ou seja um objeto (valor) só pode ter um único tipo.

Como do ponto de vista comportamental, um tipo pode ser pensado como uma especificação do comportamento dos valores a ele associados pode-se dizer que um sistema é monomórfico se cada valor só tem um único comportamento .

Contrariamente, um sistema é dito polimórfico quando é permitido que cada valor pertença a muitos tipos, i. é, tenha mais de um comportamento possível.

Uma caraterística desejável num *Sistema de Tipos* é a coerência. Um sistema é coerente se dado um programa que o sistema determina como bem-tipado, ele não possui nenhum erro de tipo na execução. Se diz que um programa é *bem tipado* se toda expressão do progrma tem um tipo e ele é correto.

A determinação da correção de um programa é o que chama-se de checagem ou verificação de tipo. Um *Sistema de Tipos* é dito bom quando fornece uma verificação estática de tipo, ou seja, em tempo de compilação.

Essa verificação estática não deveria impor restrições muito fortes, e fazer com que a linguagem tipada conserve a flexibilidade de uma linguagem não tipada. Isto é obtido quando a linguagem contém as noções de polimorfismo e herança.

Um ponto importante a considerar num Sistema de Tipos é a definição da relação de equivalência entre tipos.

3.4 Equivalência de Tipos

A importancia da equivalência de tipos reside no fato de que, no momento da verificação de tipo, é necessário determinar quando duas expressões denotam o mesmo tipo.

Nas linguagens de programação, a relação de equivalência determina a usabilidade dos valores de um tipo dado [BER 79], já que valores de tipos equivalentes podem ser utilizados tanto nas funções de um tipo como de outro.

Existem três formas principais de relação de equivalência que podem ser verificadas de maneira estática. Essas relações baseiam-se na forma que a definição de tipo é feita dentro do programa.

A primeira delas, e mais simples, é a *equivalência por nome*, onde duas definições de tipo são iguais se ambas têm o mesmo nome. O problema deste tipo de equivalência é que, no caso de tipos compostos, i. é, estruturados, a estrutura detalhada de tais tipos pode ser completamente diferente e eles ainda assim serem considerados equivalentes. Um refinamento desta relação seria só permitir como sendo equivalentes as ocorrências do mesmo nome dentro de um certo escopo, onde o nome preserva o mesmo significado.

A segunda relação é a chamada *equivalência por ocorrência*. Nesta conceituação, dois tipos são equivalentes quando suas definições ocorrem simultaneamente na mesma declaração. Assim a equivalência é inteiramente dependente da forma textual do programa.

A terceira relação de equivalência chama-se *equivalência estrutural*. Duas expressões são equivalentes estruturalmente se elas são idênticas, i.é., tem o mesmo tipo básico e estão formadas por aplicação dos mesmos contrutores na mesma ordem [AHO 86]. Logo, dois tipos são estruturalmente equivalentes se eles têm a mesma construção [COW 86].

Para exemplificar as relações apresentadas acima considere as sequintes definições de tipos numa linguagem de tipo de Pascal:

```
type vetor-int = array [1..10] of integer
var  a1, a2: vetor-int;
     b1, b2 : array [1..10] of integer;
     c1: vetor-int;
     d1, d2 : array [1..10] of integer;
```

Segundo o tipo de equivalência válida na linguagem as variáveis assim definidas resultarão equivalentes ou não.

Considerando cada uma das três relações resulta:

equivalência por nome: a1 ≡ a2 ≡ c1.

equivalência por ocorrência: a1 ≡ a2;
b1 ≡ b2 e
d1 ≡ d2.

equivalência estrutural: todas as variáveis são equivalentes.

Um dos inconvenientes que surgem com o uso da equivalência estrutural é que, dados dois tipos com estruturas idênticas que são vistos pelo usuário como semanticamente diferentes, para o sistema são equivalentes. Neste caso, se acontecer um erro de atribuição por parte do usuário, ele não sería detectado como tal pelo sistema.

Um outro problema, mais grave que o anterior, surge com a utilização de tipos abstratos de dados (TAD). Com os tipos abstratos de dados pretende-se conseguir justamente abstração (ver seção 3.6.3) mediante ocultamento da representação interna do tipo. Mas, como a equivalência estrutural necessita conhecer a estrutura do tipo para poder determinar se dados dois tipos eles são equivalentes, no caso dos tipos abstratos estar-se-ía violando o princípio da abstração.

Uma maneira de solucionar este problema é proibir essa relação para os TAD, e definir para eles uma equivalência somente por nome. Essa solução,às vezes, é chamada *encapsulação completa* [BER 79], onde a representacão do tipo fica oculta dentro de um módulo e só são visíveis o identificador do tipo e os identificadores das operações sobre o tipo.

Neste método, todo tipo é definido como um tipo abstrato. A vantagem disso é que a programação resultante é muito mais modular e independente, mas a desvantagem é a rigidez de trabalhar todos os tipos como encapsulados. Ovbiamente, existem linguagens que tentam evitar isto permitindo que a encapsulação seja opcional, logrando assim um compromisso entre encapsulação e equivalência estrutural.

Por outro lado, a grande vantagem da equivalência estrutural é que ela é muito adequada às linguagens polimórficas, enquanto

que as outras duas relações definidas não são aplicáveis [COW 86]. Isto deve-se a que, nas linguagens polimórficas, as expressões de tipos contém variáveis de tipos[1] e portanto a verificação da equivalência entre elas precisa de uma computação a mais, já que é necessário realizar um processo prévio de "unificação" das variáveis de tipo para determinar se elas estão representando os mesmos tipos para logo, determinar a equivalência das expressões.

Disto se deduz que as outras classes de equivalências não são aplicáveis, já que elas não consideram a estrutura interna do tipo e logo também não contemplam as variáveis de tipos que compõe tal estrutura.

Por outro lado, além das relações de equivalência tradicionais apresentadas existem outras menos conhecidas.

Uma delas, por exemplo, é apresentada em [BER 79], e é chamada *Euclid*, consistindo em :

- cada definição de tipo encapsulado introduz um tipo diferente.
- para tipos introduzidos pelo uso de construtores, ou pelo uso de equações de definição de tipos, a equivalência é estrutural.

Uma outra relação, definida por [HAR 84],está baseada na idéia dos tipos como álgebras. Esta *equivalência algébrica* diz que dois tipo são iguais quando tanto seu conjunto de valores (sort) como o conjunto de operações definidas sobre ele são iguais.

Com este tipo de relação se apresentam dois problemas: primeiro, determinar a igualdade de dois conjuntos de valores infinitos, e segundo determinar quando duas funções (operações) são iguais.

---

(1) Variável de tipos denota tipos, seu valor depende do contexto. Por este motivo se diz que as variáveis de tipos denotam tipos genéricos ou universais.

Esta última questão é insolúvel no caso geral, mas na implementação pode-se comparar os fechos de cada função.

Para resolver o primeiro problema poderia restringir-se a noção de tipo, considerando os valores como associados apenas a um único tipo. Assim eles podem ter então uma marca do tipo que possuem, a qual seria única, e logo a verificação da equivalência consistiria em testar se ambas expressões tem as mesmas marcas, o que poderia ser feito dinamicamente.

3.5 Inferência e Verificação de Tipo

Dado um programa P, escrito numa linguagem de programação tipada $\mathcal{L}$, é preciso definir quais expressões de P têm significado e quais não.

Se se consideram os tipos como um meio de determinar propriedades dos objetos do universo $\mathcal{U}$ através de sua classificação, então se pode dizer que os tipos participam no processo de determinação do significado das expressões de P.

Como já foi mencionado anteriormente, uma linguagem tipada $\mathcal{L}$ define implicitamente uma sublinguagem, a linguagem de tipos. Uma linguagem de tipos fica determinada quando são definidos um conjunto finito de tipos chamados *tipos básicos* e um conjunto, também finito, de *construtores de tipos*.

Os contrutores de tipo são operadores que constroem novos tipos a partir dos básicos. Estes tipos assim contruídos são chamados geralmente *tipos compostos* ou *estruturados*.

Algumas vezes utiliza-se o termo *expressões de tipos* para referir-se tanto aos tipos básicos quanto aos compostos [AHO 86].

Uma *expressão de tipo* é definida indutivamente como segue:

Seja $\mathcal{B}$ o conjunto dos tipos básicos de uma linguagem $\mathcal{L}$.

Seja $\mathcal{C}$ o conjunto dos construtores de tipos de $\mathcal{L}$ e seja $\mathcal{X}$ um conjunto infinito enumerável de *variáveis de tipo*.

Define-se $\mathfrak{T}$, o conjunto das expressões de tipos de $\mathfrak{L}$, como:

- Se $e \in \mathfrak{B}$ então $e \in \mathfrak{T}$.
- Se $x \in \mathfrak{X}$ então $x \in \mathfrak{T}$.
- Se $f_n \in \mathfrak{C} \wedge x_1, x_2, \ldots, x_n \in \mathfrak{T}$ então $f_n(x_1, x_2, x_3, \ldots, x_n) \in \mathfrak{T}$.
- nada mais pertence a $\mathfrak{T}$.

Da mesma maneira, uma linguagem $\mathfrak{L}$ permite computar valores (objetos), os quais podem ser simples (atómicos) ou compostos (estruturados).

Como no caso dos tipos, os objetos compostos também são obtidos a partir dos atómicos pela aplicação de construtores de objetos da linguagem.

Ás vezes utiliza-se o termo *expressões de objetos* ou simplesmente *expressões* para referir-se tanto aos objetos simples quanto aos compostos.

Como um exemplo do anterior considere-se a linguagem de calculo-λ tipada de primeira ordem. Para tal linguagem, se tem por um lado uma definição indutiva para a construção de tipos (expressões de tipos) e por outro uma definição, também indutiva, para a construção de termos da linguagem (expressões de objetos).

As expressões da linguagem são definidas como:

- Se $x \in X$ então x é termo-λ[1] , onde X é o conjunto infinito enumerável dos identificadores;
- Se M, N são termos-λ então (MN) é termo-λ;
- Se $x \in X$ e M é termo-λ então (λx. M) é termo-λ;
- Nada mais é termo-λ.

E a definição das expressões de tipos é a seguinte :

- Se $\tau \in \mathfrak{B}$ então τ é um tipo, onde $\mathfrak{B}$ é o conjunto dos tipos básicos;
- Se σ, τ são tipos então $\sigma \rightarrow \tau$ é um tipo;
- Nada mais é um tipo.

---

(1) Termo-λ é um termo que pertence à linguagem do cálculo-λ de primeira ordem.

Agora, dadas as definições das expressões da linguagem $\mathcal{L}$ e de tipo é necessário determinar quando uma expressão de $\mathcal{L}$ dada tem um determinado tipo.

Para isto é preciso definir a maneira como são atribuídos os tipos ás expressões de objetos de $\mathcal{L}$.

Segundo [REY 74] o significado de uma expressão numa linguagem $\mathcal{L}$ depende dos valores que denotam suas variáveis livres(1), tanto de objetos, como de tipos.

Então determinar se uma expressão de $\mathcal{L}$ tem um tipo, consiste em *atribuir tipos* ás variáveis de tipos da expressão.

Essa atribuição é feita através da definição de *Atribuição de tipos* ($\mathcal{A}$), função que vai do conjunto das expressões de $\mathcal{L}$ ao conjunto das expressões de tipo, i. é: $\mathcal{A} : \mathcal{EL} \dashrightarrow \mathcal{T}$;onde $\mathcal{EL}$ é o conjunto formado pelas expressões da linguagem $\mathcal{L}$.

Em geral a definição da função de Atribuição de tipos é dada por um conjunto de *regras de atribuição de tipo* e de esquemas de regras. Uma regra de atribuição de tipo ou simplesmente regra de tipo é uma regra de inferência que permite inferir os tipos das expressões de $\mathcal{L}$.

Para o exemplo acima do cálculo-λ, a função de atribuição seria dada pelas seguintes regras de tipo :

•para cada variável $x \in \mathcal{X}$ se tem uma regra da forma:

i) $\dfrac{}{x : \sigma}$ onde σ é um tipo;

•para os termo-λ (exceto as variáveis) se tem os esquemas de regras seguintes:

ii) $\dfrac{x : \sigma \qquad M : \tau}{(\lambda x.\ M) : \sigma \to \tau}$

iii) $\dfrac{M : \sigma \to \tau \qquad N : \sigma}{(MN) : \tau}$

---

(1) Uma variável é livre do ponto de vista semântico, quando seu significado (valor denotado) depende do contexto.

A partir da definição da função $\mathcal{A}$ surge a noção de *expressões bem-tipadas* [MIL 78].

Uma expressão é *bem-tipada*, com respeito ao contexto de tipos, se a expressão tem pelo menos um tipo. Em outras palavras, é bem tipada quando satisfaz uma *condição sintática suficiente*: inferir os tipos das expressões e sub-expressões através das regras de atribuição de tipo.

Continuando com o exemplo anterior, um termo M do cálculo-λ é bem tipado se e somente se a atribuição $M : \sigma$ pode ser deduzida pelas regras de atribuição i),ii) e iii) para algum tipo σ.

Definida a atribuição de tipos, é possível demostrar que uma expressão, com uma atribuição de tipos válida, não pode ser mal-tipada, e que dado um programa com informação de tipo incompleta pode-se inferir a atribuição de tipo válida de suas expressões. Este processo denomina-se *inferência de tipo*.

Dado que uma expressão pode ter mais de um tipo, no processo de inferência é necessario obter seu *tipo mais geral*. Milner em [MIL 78] demostrou que o *algoritmo de unificação* (U) de Robinson pode ser utilizado para este fim.

Tal algoritmo diz que, dadas duas expressões, é possível obter uma substituição[1] tal que essas expressões unifiquem[2], i.é, para cada par de expressões σ e τ se tem que :

- Se U (σ, τ) sucede com U, então U unifica σ e τ, i.é, $U\sigma = U\tau$;
- Se R unifica σ e τ então U (σ, τ) sucede dando U tal que para alguma substituição S se tem : $R = SU$.

Na definição do seu algoritmo de inferência de tipos, Milner utiliza a idéia de *unificador mais geral* para cláusulas de Horn,

---

(1) Uma substituição é uma função que mapeia variáveis de tipos em tipos [MIL 78].

(2) Duas expressões m e ω unificam quando, dada uma substituição R para ambas, se tem que $Rm = R\omega$, onde Rm significa a substituição R aplicada em m.

nas expressões de tipos. Assim, o tipo mais geral (TMG) de uma expressão é aquele que satisfaz o seguinte:

Dada uma expressão $e \in \mathcal{EL}$, com $e : \sigma$, i. é e é bem-tipada e seu tipo é σ, se diz que σ é TMG se e somente se $\forall \alpha \in \mathcal{T}$ $(\mathcal{A}(e) = \alpha \wedge \alpha = \eta\sigma)$, onde η é uma substituição.

Em [REY 85] define-se o tipo mais geral ou tipo principal como sendo um esquema de tipo[1], tal que os tipos válidos da expressão são exatamente aqueles que podem ser obtidos a partir do tipo principal pela substituição das variáveis de tipos por tipos arbitrários. Um exemplo de tipo principal seria :

$x : \alpha \qquad f : \alpha \rightarrow \alpha \not\vdash f\ (f\ x) : \alpha$; sendo α uma variável de tipo. Com a substituição [string/α] ter-se-ia um tipo válido, i. é :

x : string $\quad f$: string $\rightarrow$ string $\quad \vdash \quad f\ (f\ x)$: string.

[MIL 78] demostrou que uma expressão tem tipo principal se ela tem pelo menos um tipo e se o tipo principal existe, ele é único. Mais ainda, demonstrou que o tipo principal pode ser computado a partir dos tipos principais das sub-expressões utilizando o algoritmo de unificação de Robinson.

Antes de começar a falar sobre o algoritmo de inferência de tipos é importante deixar bem claro o que se entende por inferência de tipos.

Segundo [AHO 86], *inferência de tipos* é o processo de determinar o tipo de uma construção de uma linguagem a partir da forma como esta é utilizada.

[CAR 85] define *inferência de tipos* como o processo de dedução de tipos de uma expressão através do contexto, o que é feito numa forma ascendente (bottom up) sobre as árvores das expressões.

---

(1) Um esquema de tipo é uma expressão de tipo contendo variáveis de tipo.

Tais árvores são construídas tomando a expressão como raíz e suas sub-expressões como filhos, recursivamente até chegar às folhas onde encontram-se as expressões mais simples, i. é, valores atómicos (variáveis e constantes).

O processo de dedução começa nas folhas das árvores, cujos tipos são dados pela função de atribuição, a qual determina o contexto.

A partir desse contexto, e com as regras de inferência de tipos, é possível determinar o tipo de uma expressão qualquer, sempre que ela seja bem-tipada, caso contrário detectar-se-ia um *erro de tipo*.

Alguns algoritmos de inferência precisam que o contexto, i. é, a atribuição de tipos às variáveis e constantes normais , seja feito em forma explícita na linguagem de programação , como por exemplo no caso da linguagem PASCAL; mas outros são capazes de inferir todos os tipos das expressões da linguagem sem nenhuma declaração explícita,como no caso de ML cujo algoritmo de inferência de tipos foi desenvolvido por Milner [MIL 78]. Tal algoritmo, chamado também de algoritmo da boa-tipagem, é sintaticamente completo e coerente.

Sintaticamente coerente significa que sempre que o algoritmo sucede então a expressão é bem-tipada e sintaticamente completo significa que sempre que uma expressão é bem-tipada o algoritmo sucede, obtendo o tipo mais geral da expressão (TMG).

O algoritmo de ML trabalha em forma ascendente, baseado na idéia de unificação de variáveis de tipo (TMG), não precisa de nenhuma declaração de tipos. Atribui às variáveis-ML (variáveis de objetos) um tipo genérico, i. é, uma variável de tipo.

Mais tarde as variáveis de tipos são instanciadas com os tipos que, segundo o contexto, elas estão denotando. Isto é feito pelo algoritmo de unificação de Robinson, o qual faz uma propagação da informação de tipo através de todas as instâncias da mesma variável.

Apesar deste algoritmo de inferência ser adequado em linguagens polimórficas, também poderia ser utilizado nas linguagens monomórficas onde cada variável ou expressão tem um único tipo possível, para justamente testar que todas as variáveis de tipos atribuídas desaparecem ao final do processo de inferência [CAR 85].

Embora, o algoritmo de inferência de ML seja o mais eficiente, ele é restrito, não considerando a noção de herança e limitando a quantificação universal (ver polimorfismo em seção 3.6.1) por questões de decibilidade.

Em muitas extensões do Sistema de Tipos de ML o problema da inferência de tipo foi demostrado que é indecidível.

Quando se define um Sistema de Tipos, define-se um conjunto de regras de inferência de tipos que são as que determinam o algoritmo de inferência. Tal algoritmo é implementado pelo processo de verificação de tipos [AHO 86]. Dependendo da estratégia de implementação, a verificação pode ser estática ou dinâmica.

A verificação dinâmica é feita durante a execução do programa enquanto que a estática é feita totalmente em tempo de compilação e tem como propósito fundamental a deteção de certos erros de programação, entre eles os erros de tipos permitindo assim determinar as inconsistencias de tipos e garantir que os programas executáveis são consistentes em tipo (type-consistent).

Logo a verificação estática facilita a deteção mais cedo de erros e dispensa a necessidade dos tipos em tempo de execução, incrementando assim, a eficiência e impondo uma disciplina de programação [CAR 85] [MIL 78].

O verificador de tipos testa que o tipo de uma construção unifique com o tipo esperado pelo contexto e o projeto de tal verificador está baseado nas regras de construção sintática da linguagem, na noção de tipo e nas regras de inferência de tipos para cada construtor da linguagem.

Também se deve ter em conta no projeto de um verificador outras caraterísticas da linguagem, como por exemplo, se a linguagem suporta polimorfismo (ver seção 3.6.1).

Neste caso é preciso que o verificador seja capaz de manipular valores simbólicos, já que é a única maneira como o polimorfismo pode ser verificado estáticamente.

A necessidade de tal manipulação simbólica surge quando precisa-se conhecer se duas expressões de tipo polimórficas[1] são equivalentes.

[MIL 78] demostrou que para qualquer programa válido[2], é possível derivar uma única atribuição de valores para as variáveis de tipo, como parte da determinação da equivalência entre tipos.

Logo, a verificação de tipos poderia ser pensada como um processo que consta de duas partes: primeiro a derivação da atribuição de tipos das expressões, independentemente de qualquer relação de equivalência e por último, a utilização dessa relação entre os tipos para unificar expressões.

Assim para cada expressão da linguagem precisa-se:

- uma regra de verificação de tipo que indique que a expressão e tem o tipo σ .
- uma regra de avaliação que permita obter o valor de uma expressão qualquer, i.é que permita concluir que "e tem o valor E ".

A regra de verificação pode invocar ás regras de avaliação, mas a inversa não é necessária.

---

(1) Uma expressão de tipo é polimórfica quando denota muitos tipos, i. é contém variáveis de tipos.

(2) Programa válido, do ponto de vista do sistema de tipos, é aquele que tem todas suas expressões bem tipadas.

3.6 Aspetos específicos dos tipos nas linguagens de programação

Nesta seção serão apresentadas três características desejáveis na definição de linguagens poderosas. Essas características estão intimamente relacionadas com a noção de tipo e são: polimorfismo, herança e abstração.

3.6.1 Polimorfismo, Coerção e Overloading

O termo *polimorfismo* significa literalmente "muitas formas". Assim, algo é polimórfico quando "possui mais de uma forma". Em computação tal termo é, geralmente, aplicado às funções ou procedimentos, que tem muitos tipos ou operam sobre muitos tipos.

Uma função é dita *polimórfica* quando aceita argumentos de distintos tipos.

Logo, a noção de polimorfismo está intimamente relacionada com a de tipo, mais ainda, o conceito de tipo é condição necessária para definir polimorfismo.

Segundo [AHO 86] a palavra "polimórfico" pode ser aplicada a qualquer fragmento de código que admita argumentos de tipos diferentes.

Uma linguagem que suporta polimorfismo é dita polimórfica. Nela tanto as variáveis como os valores podem ter mais de um tipo e as funções cujos operandos têm essa característica são chamadas *funções polimórficas* [CAR 85].

Strachey definiu duas classes de polimorfismo: *universal* ou *genérico* e *ad-hoc*.

Se tem polimorfismo universal quando as funções trabalham *uniformemente* com argumentos que podem ter *infinitos tipos*.

Por outro lado, o polimorfismmo ad-hoc, ocorre quando se tem funções que somente operam sobre um conjunto finito de tipos diferentes, os quais, geralmente, não têm nenhuma estrutura em comum. Além disso, tais funções comportam-se em forma não relacionada para cada tipo, i. é têm comportamento não uniforme.
Uma função polimórfica ad-hoc pode ser pensada como um conjunto

finito de funções monomórficas.

Do ponto de vista da implementação, uma função com polimorfismo ad-hoc costuma ter diferentes códigos para cada tipo de argumento, enquanto que no caso do polimorfismo universal se tem o mesmo código que se executa com argumentos de qualquer tipo.

O polimorfismo universal segundo Cardelli em [CAR 85] é formado pelo *polimorfismo de inclusão* e o *paramétrico* (ver figura 3.1)

No polimorfismo de inclusão um objeto pode pertencer a muitas classes (tipos) não necessariamente disjuntas.

O polimorfismo ocorre porque uma função que admite argumentos de um determinado tipo (classe), admite também argumentos de qualquer subtipo (subclasse) daquele tipo. Através desta hierarquia de inclusão de tipos consegue-se um comportamento uniforme das funções universais.

O polimorfismo paramétrico é uma propriedade dos programas que admitem tanto parâmetros de objetos como de tipos [CAR 87]. Existe duas classes de polimorfismo paramétrico: *implícito* e *explícito*.

O polimorfismo paramétrico é chamado explícito, quando a parametrização é obtida através de parâmetros de tipo explícitados na definição e chamada de procedimentos.

Neste polimorfismo, então, uma função paramétrica ou *genérica* tem um parâmetro de tipo o qual denota o tipo do argumento e através dele, consegue-se que a função tenha um comportamento uniforme quando aplicada a valores de diferentes tipos.

Um exemplo de polimorfismo paramêtrico explícito é a definição da função identidade na linguagem QUEST [CAR 89]:
let Id = **fun** (t: *type*) **fun** (a:t) a; onde **fun** denota a abstração-λ.

Aplicando a função para um inteiro ficaria: Id(int)(3), e para um boleano: Id(bool)(true).

O polimorfismo paramétrico é chamado implícito quando os tipos podem conter *variáveis de tipos*. Se um parâmetro de procedimento tem uma variável de tipo ou uma expressão contendo variáveis de tipo, então o procedimento pode ser aplicado a argumentos de diferentes tipos.

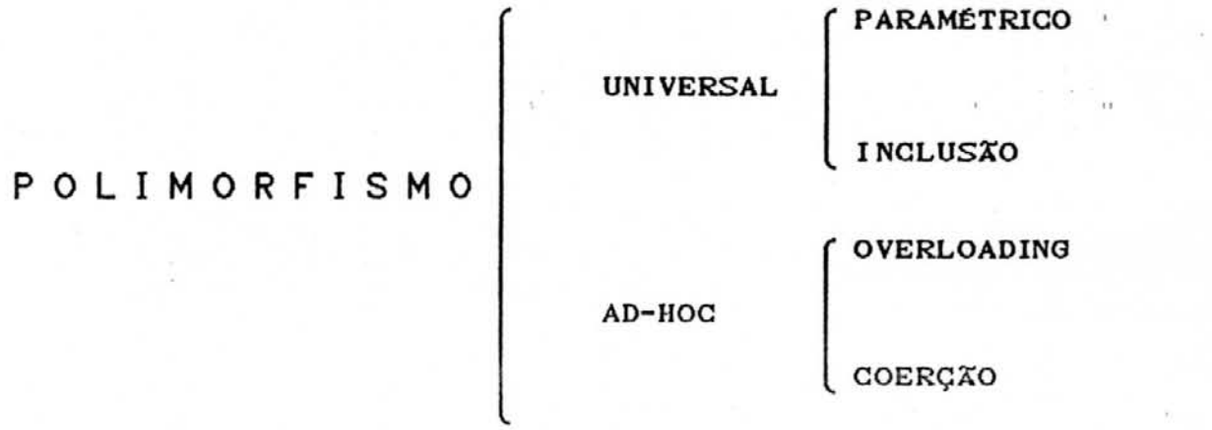


Fig. 3.1 : Classes de Polimorfismo

Considere o exemplo anterior da função identidade, mas agora com polimorfismo implícito:

let Id = fun (a : α) a; onde α é uma variável de tipo.

A aplicação de Id para um inteiro fica: Id(3) e para um booleano : Id(true).

Existe uma relação, teórico-prática, entre polimorfismo explícito e implícito. O polimorfismo implícito pode ser considerado como uma forma abreviada do explícito, onde os parâmetros de tipos são omitidos e devem ser redescobertos pelo verificador de tipos. Omitir os parâmetros de tipos leva a ter identificadores não ligados a um tipo, e estes são precisamente as variáveis de tipo. Para determinar o tipo das expressões é necessário então, a inferência de tipo. De fato, no polimorfismo implícito pode-se omitir totalmente a informação de tipo se o programa é pensado também como um procedimento que tem variáveis de tipo associadas aos parâmetros e identificadores. Assim, os programas podem ser totalmente livres de tipo: let Id= fun(a) a.

Polimorfismo explícito é mais expressivo que o implícito porque pode tipar programas que não podem ser tipados no implícito, mas é mais verboso. Por outro lado, o polimorfismo implícito é ambíguo, já que o mesmo programa polimórfico implicitamente pode corresponder a diferentes programas explicitamente polimórficos Na prática, é desejável omitir alguma informação de tipo, mesmo com polimorfismo explícito, criando assim uma zona cinza entre totalmente explícito e totalmente implícito. Nesta região cinza, as técnicas de inferência de tipos usadas para polimorfismo implícito podem ser muito úteis.

Quanto ao polimorfismo ad-hoc, este admite duas divisões : *overloading* e coerção [CAR 85]. A distinção entre ambas formas muitas vezes é confusa.

Na primeira classe do polimorfismo ad-hoc, a *overloading*, o mesmo símbolo é utilizado para denotar diferentes funções e através do contexto determina-se qual é a função que está sendo denotada. Logo, *overloading*, é uma noção puramente sintática pois utiliza-se o mesmo nome para denotar objetos semânticos diferentes.

Um exemplo típico de *overloading* nas linguagens de programação é o operador " + ", o qual pode denotar: "soma de inteiros", "soma de reais", "concatenação de strings", etc.

A *overloading* é resolvida quando um único significado pode ser associado para cada ocorrência do símbolo. Isto é feito em tempo de compilação.

Por outro lado, a coerção é uma operação semântica que consiste em converter um argumento ao tipo esperado pela função.

Tal conversão pode ser implícita ou explícita. É implícita quando ela é feita automaticamente pelo compilador e explícita quando o programador deve escrever o código necessário para causar a conversão [AHO 86]. Neste último caso, a coerção pode ser feita estática ou dinamicamente.

3.6.2 Herança

Existem duas maneiras de estruturar dados nas linguagens de programação. A primeira, e mais comum, é derivada da matemática. Nela um dado é estruturado como produtos cartesianos (tipos registros), somas disjuntas (tipos variantes) e espaços de funções (funções e procedimentos).

A segunda maneira é derivada da lógica de classes . Nesta os dados são organizados numa hierarquia de classes, onde um dado em qualquer nível herda todos os atributos do superior na hierarquia [CAR 84].

A noção de tipo suporta esta maneira de estruturação, que denomina-se *herança de atributos*, devido a que, como já foi mencionado no início deste capítulo, os tipos 'classificam" o universo de discurso em classes de objetos não necessariamente disjuntas.

Na seção anterior, quando se falou em polimorfismo, descreveu-se uma classe de polimorfismo especial, chamado de polimorfismo de inclusão, e que modela o conceito de herança. Nesta seção se tratará deste polimorfismo com mais detalhe.

A noção de herança nas linguagens tipadas é obtida pela capacidade de expressar relações entre tipos dentro da linguagem, e especificamente a relação de ordem. Essa relação é definida para cada construtor de tipo da linguagem, e a maneira de definí-la dependerá essencialmente da noção de tipos considerada na linguagem.

Assim, por exemplo, pensando os tipos como conjuntos (tanto de valores como de operações) é lógico pensar que a relação de ordem estará dada pela inclusão de conjuntos: um tipo será subtipo de outro quando seja subconjunto daquele.

Por outro lado, na noção de tipos como fórmulas,onde um predicado da lógica denota um tipo, um tipo será subtipo de outro se todo valor que o "satisfaz" também "satisfaz" ao supertipo.

A herança pode ser classificada em *simples* ou *múltipla*. No caso da herança simples a hierarquia de classes tem a forma de uma árvore, i. é, cada classe tem uma única superclasse.

Inversamente, na herança múltipla, um objeto pode pertencer a mais de uma superclasse diferente, a relação de subclasse é então um grafo qualquer.

A importância de determinar a relação de ordem entre os tipos de uma linguagem reside no fato de que se φ é um subtipo de φ' então existe uma *conversão implícita* desde os valores do tipo φ para os valores do tipo φ'. Assim uma expressão do tipo φ pode ser usada em qualquer contexto que se aceite expressões do tipo φ'.

Esta relação de subtipo pode ser expressa como:

$\varphi \leq \varphi'$; a qual indica que φ é *subtipo* de φ'.

A idéia de que qualquer expressão do tipo φ pode ser usada como uma expressão do tipo φ' é formalizada pela seguinte regra de inferencia:

$$\frac{\Pi \vdash e : \varphi \quad \wedge \quad \varphi \leq \varphi'}{\Pi \vdash e : \varphi'} \ ;$$

onde Π denota o contexto válido num determinado momento e "$e : \varphi$" significa que o objeto e tem um tipo φ.

A relação $\leq$ é uma relação de ordem parcial, i. é satisfaz:

- *reflexiva :* $\varphi \leq \varphi$.
- *transitiva:* Se $\varphi \leq \xi$ e $\xi \leq \mu$ então $\varphi \leq \mu$.
- *antisimétrica:* Se $\varphi \leq \xi$ e $\xi \leq \varphi$ então $\varphi \equiv \xi$.

Semanticamente as leis de reflexividade e transitividade implicam na existência de *conversões implícitas* que devem ocorrer corretamente se o significado da linguagem não é ambíguo.

Para cada tipo, a reflexividade assegura que é possível ter uma conversão implícita que seja aplicada a qualquer valor do tipo sem mudar seu tipo , i. é, a conversão deve ser a *função identidade*.

Além disso, quando acontece que $\varphi \leq \xi$ e $\xi \leq \mu$ é possível converter φ a μ diretamente ou através da conversão prévia para ξ. Logo, a conversão deve ser a *composição funcional das conversões* de φ a ξ e de ξ a μ.

Para cada construtor da linguagem existe uma conversão implícita, ou seja, existe uma relação de subtipagem definida para cada um deles.

Suponha uma linguagem que tem os seguintes construtores: espaço de funções (→), construtor de tipos registros ou produto cartesiano (x), construtor de tipos variantes ou soma disjunta (+).

Um objeto de tipo registro (registro) é um conjunto de campos (tuplas) nomeados, não ordenados, onde cada campo pertence a um tipo possivelmente diferente.

A notação que se usará para denotar um registro é a seguinte:

$\{ a_1 : \tau_1, a_2 : \tau_2, \ldots a_n : \tau_n \}$; onde a_i são os nomes dos campos do registro e os τ_i são os tipos desses campos.

Os tipos registro são utilizados para modelar classes e subclasses nas linguagens orientadas a objetos e por isso a relação de subtipagem sobre eles corresponde à noção de herança [CAR 85].

Neles a relação de herança está implícita nos nomes e tipos dos componentes do registro. Ela só depende da estrutura dos objetos e portanto não necessita ser declarada explicitamente.

Como os tipos registros modelam classes, os registros modelam as instâncias dessas classes e são construídos explicitamente especificando os valores dos componentes.

Quanto aos tipos variantes, os objetos desse tipo são (como os registros) conjuntos não ordenados de pares nome-tipo com a seguinte notação:

$[a_1 : \tau_1, a_2 : \tau_2, \ldots a_n : \tau_n]$; onde os a_i são os nomes dos componentes e os τ_i são seus tipos.

Seja C uma conversão implícita de ξ_2 a ξ_1 e outra C' de χ_1 a χ_2. É natural converter a função f de tipo $\xi_1 \rightarrow \chi_1$ para a função $\lambda x : C' (f (C x)) : \xi_2 \rightarrow \chi_2$.

Se esta conversão de funções é considerada como implícita de $\xi_1 \to \chi_1$ a $\xi_2 \to \chi_2$ então o operador ($\to$) satisfaz a seguinte regra de inferência (regra de tipagem) que formaliza a relação de subtipagem para tipos funções:

Se $\xi_2 \leq \xi_1 \wedge \chi_1 \leq \chi_2$ então $\xi_1 \to \chi_1 \leq \xi_2 \to \chi_2$.

Assim o operador ($\to$) será monotônico no seu segundo operando e antimonotônico no primeiro, i. é, o domínio se contrai e o codomínio se expande(ver figura 3.2).

No caso dos registros, um tipo registro A é subtipo do tipo B se e somente se A tem todos os campos de B, e possivelmente mais, e além disso os campos comuns de A e B estão na relação de $\leq$.

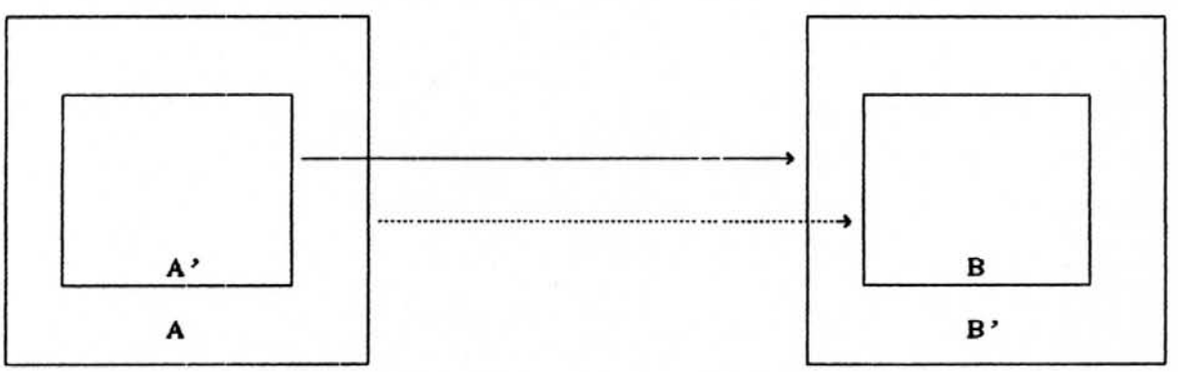
Assim a regra de tipagem é:

- $\eta \leq \eta$ onde η é um tipo básico.
- $\{a_1 : \tau_1, a_2 : \tau_2, \ldots a_n : \tau_n, \ldots a_m : \tau_m\} \leq$ $\{a_1 : \sigma_1, a_2 : \sigma_2, \ldots a_n : \sigma_n\}$ sss $n \leq m \wedge \tau_i \leq \sigma_i$; para todo i= 1, ... n com n,m $\in$ N.

Logo, o tipo registro é monotônico em todos seus operandos.

Das duas regras anteriores pode-se derivar a seguinte regra:

- Se a : τ e $\tau \leq \sigma$ então a : σ; a qual é o enunciado do Teorema da Subtipagem Sintática de [CARD 86].



$A \to B \leq A' \to B' \iff A' \leq A \wedge B \leq B'$

Fig. 3.2 : Relação de Subtipagem para o Tipo Função

Por último, no caso de tipos variantes a relação de subtipagem fica definida através da seguinte regra de tipagem:

- $[a_1 : \tau_1, a_2 : \tau_2, \ldots a_n : \tau_n] \leq$

  $[a_1 : \sigma_1, a_2 : \sigma_2, \ldots a_n : \sigma_n \ldots a_m : \sigma_m]$ sss

  $n \leq m$ e $\tau_i \leq \sigma_i$; para todo $i = 1, \ldots n$ com $n,m \in N$.

Do ponto de vista da metodologia do software, os subtipos são essenciais para a extensão ordenada de grandes sistemas de software.

Eles fornecem um mecanismo flexível que permite aos usuários ampliar as estruturas de dados e operações providas pelo sistema.

Assim se o sistema fornece um tipo de dados, os usuários podem criar um subtipo dele com operações especializadas.

A vantagem deste mecanismo está em que o velho sistema reconhecerá os novos subtipos como instâncias dos tipos anteriores e será possível operar sobre eles.

Os subtipos desta forma incrementam a confiabilidade dos sistemas porque fornecem uma maneira de incrementar a funcionalidade sem ter que mudar o sistema original.

Uma questão que deve ser destacada é que a relação de subtipagens só vale para os tipos e não existe nenhuma relação equivalente para os objetos

3.6.3 Abstração e Tipos Abstratos de Dados

A *abstração* é o processo de identificar as qualidades ou propriedades importantes do fenômeno que está sendo modelado [GHE 82].

Determinar quais propriedades são importantes depende da finalidade para a qual a abstração está sendo realizada.

Por exemplo para uma pessoa que está aprendendo a dirigir, um carro pode ser visto como: um volante de direção, um pedal de embreagem, um pedal de freio e um pedal de acelerador; mas o

---

(1) Variável de tipos denota tipos, seu valor depende do contexto. Por este motivo se diz que as variáveis de tipos denotam tipos genéricos ou universais.

engenheiro que projeta o carro, considera além de todos aqueles elementos, que são importantes para o motorista, o motor e a relação que existe entre ele e os pedais. Note que estas últimas propriedades são irrelevantes para o motorista.

Assim, a abstração surge do reconhecimento de similitudes entre certos objetos, situações ou processos no mundo real. Logo, ela é a decisão de concentrar-se sobre tais similitudes e ignorar as diferênças.

Segundo [HOA 72] o processo de abstração comprende quatro etapas. A primeira consiste em concentrar-se sobre as propriedades ou caraterísticas compartilhadas por muitos objetos ou situações no mundo real, esquecendo suas diferênças. Esta etapa é chamada de abstração.

A segunda, chamada representação, é a escolha de um conjunto de símbolos para suportar a abstração.

A terceira é a de manipulação e trata-se da definição das regras para a transformação das representações simbólicas. Tal transformação deve produzir os mesmos efeitos que os da manipulação dos objetos no mundo real.

Por último, está a etapa de axiomatização, a qual consiste em definir as propriedades que foram abstraídas e que são compartilhadas pelas manipulações do mundo real e dos símbolos.

3.6.3.1. Abstração e computação.

O processo de abstração está intimamente relacionado com o universo matemático e o computacional.

No projeto de um programa o programador faz abstração porque ele se concentra nas caraterísticas mais importantes do problema a ser resolvido deixando de lado as propriedades irrelevantes. Após isso, ele deve decidir como essa informação será representada no computador.

Logo um programa é uma abstração da realidade (modelo abstrato) [GHE 82].

Assim, as linguagens de programação têm uma dupla relação com a abstração, por um lado são ferramentas para implementar modelos abstratos (programas), e por outro lado, são modelos abstratos em si mesmas

Continuando com os passos que realiza um programador no projeto de um programa, após abstrair e escolher a representação, que são as primeiras duas etapas definidas por [HOA 72], deve definir a axiomatização e a manipulação.

Com respeito à axiomatização, ela, em geral, não é realizada explícitamente, e corresponderia à especificação do programa.

A axiomatização é utilizada na verificação da correção do programa já que ela especifica as propriedades que o programa deve ter, i. é, determina o comportamento do programa.

As vantagens da axiomatização é que clarifica o conceito abstraído, explicando suas caraterísticas essenciais.

Além disso, quando o programa baseia-se nela, pode reduzir-se a quantidade de erros e, obviamente, isto provê um aumento na confiabilidade do software.

Por outro lado, a axiomatização também ajuda à etapa de documentação e explicação dos programas.

Quanto à etapa de manipulação, ela é programação propriamente dita. O programador, uma vez escolhida uma representação para a abstração, programa as operações sobre essa representação.

Tais operações correspondem com as manipulações sobre os objetos no mundo real

O êxito de um programa, segundo [GHE 82], depende de três condições:

- A axiomatização é uma descrição correta dos aspetos do mundo real.
- A axiomatização é uma descrição correta do comportamento do programa.

---

(1) Termo-λ é um termo que pertence à linguagem do cálculo-λ de primeira ordem.

• As escolhas da representação e o método de manipulação são tais que o custo de execução do programa é aceitável.

O uso da abstração ajuda a postergar algumas decisões sobre a representação dos dados até ter mais conhecimento sobre o comportamento do programa e as caraterísticas dos dados, evitando certos erros. Logo a abstração incrementa a confiabilidade do software.

Outra vantagem do uso da abstração tanto no desenvolvimento de grandes sistemas como na programação é que ela é uma maneira de estruturar dados e programas. Com isto se consegue suportar a idéia de desenvolvimento descendente (top down) por refinamentos sucessivos, decompondo os problemas com base no reconhecimento de abstrações.

3.6.3.2 Tipos Abstratos de Dados (TAD)

Os tipos definidos pelo usuário que satisfazem :

• a associação de uma representação às operações concretas, numa unidade apropriada da linguagem (módulo) que implementa os tipos novos,

• ocultamento da representação do tipo novo das unidades que o usam,

são chamados Tipos Abstratos de Dados (TAD) [GHE 82].

Um TAD encapsula informação que proporciona uma representação de um objeto complexo e fornece operações que implementam as manipulações possíveis sobre o objeto. A representação interna do objeto definido pelo TAD é completamente oculta para os usuários, e só as operações que implementam manipulações desta representação são visíveis para eles [DAN 88].

---

(1) Uma variável é livre do ponto de vista semântico, quando seu significado (valor denotado) depende do contexto.

Assim os TAD satisfazem o Princípio da invisibilidade da informação de Parnas o qual diz que um módulo deve esconder informação no sentido seguinte:

- ele esconde um segredo que é invisível fora dele,
- ele só fornece a informação que deseja, através de um conjunto de funções de acesso, protegendo assim a representação interna.

Esta idéia envolve alguns conceitos das linguagens de programação:

- encapsulamento: representação dos objetos e implementação das operações abstratas reunidas numa única região do programa (módulo).
- proteção: os objetos só podem ser manipulados por meio das operações de um conjunto pré-estabelecido, conseguindo que a representação fique invisível e protegida contra manipulações incorretas.

Os TAD são uma maneira familiar e efetiva de estruturar programas já que suportam a noção de refinamentos sucessivos, decompondo os problemas através de abstrações.

Uma *especificação* do TAD é uma descrição precisa, independente da representação, de suas propriedades gerais, i. é, seu comportamento.

Contrariamente, uma *implementação* de um TAD é uma maneira particular de definir o processamento para simular tal comportamento.

Em geral, a descrição do comportamento (especificação) do TAD é declarativa. Enquanto que a implementação é operativa envolvendo detalhes que codificam decisões de projetos, deixados em aberto na especificação. Por isto, a especificação é feita antes que a implementação [VEL 87].

A classe de abstração oferecida pelos TAD's pode ser suportada por qualquer linguagem com um mecanismo de chamada de procedimentos, dando protocolos apropriados para ser observados pelos programadores.

A verificação de tipo nestas linguagens não foi baseada em nenhuma teoria dando significado para os TAD's. Este é o grande desafio para pesquisadores em semântica de linguagens de programação, que buscam colocar a programação sobre base firme com teorias semânticas úteis. Para estes pesquisadores, um programa e os objetos representados nele deveriam ser valores não ambigüos em algum domínio semântico [DAN 88].

Os tipos fornecem uma maneira de representar tais significados. Se se caracteriza os objetos de computação como elementos de algum domínio semântico (ou Tipo), os TAD's devem ter alguma interpretação nesse sentido. Existem várias possibilidades.

a) Método Algébrico (Primeira Ordem)

Nas décadas passadas, varias idéias algébricas foram úteis na área de TAD [GOG 77] [GOG 77a][GOG 78][GUT 78].

Uma álgebra é uma entidade matemática formal essencialmente composta de conjuntos de valores (*carriers*) e operações sobre estes valores (este tema será desenvolvido amplamente no capítulo 4). Como é possível deduzir, esta definição de álgebra corresponde com a noção de TAD.

Uma especificação algébrica ou axiomática consiste numa especificação sintática, que determina os nomes das operações e seus domínios e contra-domínios, e numa especificação semântica, conjunto de relações que define o significado das operações [GUT 77].

Em resumo uma especificação de um TAD deve conter:

- denominação do conjunto ou sorte do TAD;
- o conjunto das operações definidas com seus tipos;
- o conjunto de predicados, se existirem;
- o conjunto de axiomas que especificam as propriedades das operações do TAD.

Um exemplo de especificação de um TAD é:

PILHA-DE-(ELEMENTOS: TAD)

(# ELEMENTOS é um TAD que é parâmetro de PILHA #)

sortes: Pilha, Elemento.

operações:

empilha (Pilha, Elemento) → Pilha ;

desemplilha (Pilha) → Pilha ;

topo (Pilha) → Elemento ;

cria ( ) → Pilha .

predicados:

vazia? (Pilha).

axiomas:

1) (∀p:Pilha, ∀e:Elemento) desempilha(empilha(p,e))=p.

2) (∀p:Pilha, ∀e:Elemento) topo(empilha(p, e))=p.

3) desempilha(cria)= cria.

4) (∀p:Pilha) (vazia?(p) ⟺ p = cria).

5) (∀p:Pilha) (p = cria ⟺ ((∀q:Pilha, ∀e:Elemento) ¬ p = empilha (q, e))).

A principal restrição do método algébrico é que ele é de primeira ordem, i. é, funções e também TAD's, não podem ser passados como argumentos ou retornados de funções.

b) Métodos de Alta Ordem e Tipos de TAD

Os métodos de alta ordem são embasados na noção de *tipos-como-fórmulas*. Esta noção surge no trabalho de Curry e Feys [CUR 58], no qual foi observada uma correspondência entre os axiomas da lógica proposicional e combinadores básicos. A noção foi usada por Girard em [GIR 71] e posteriormente aperfeiçoada em [GIR 89]. Nesse trabalho, Girard introduziu uma forma de cálculo lambda de segunda ordem como uma ferramenta de prova teorética (proof-theoretic).

Por outro lado, Reynolds [REY 78] também utilizou a noção de tipos-como-fórmulas, ao definir uma linguagem (independentemente de Girard) que foi conhecida posteriormente como cálculo-λ de segunda ordem.

Mitchell e Plotkin [MIT 84] usaram uma versão estendida do

cálculo-λ de segunda ordem criando uma linguage chamada SOL, para representar TAD's. O objetivo principal do trabalho era representar o tipo de um TAD. Note a diferença com o enfoque puramente algébrico, nele os tipos (conjuntos de domínios de valores) permaneciam num plano semântico diferente dos TAD's (álgebras). Em outras palavras, um TAD não possuia um tipo e portanto não era considerado um objetivo de primeira classe. Ele só tinha uma assinatura que era a representação sintática da álgebra.

Mitchell e Plokin começaram aceitando o modelo algébrico dos TAD's e consideraram sua representação concreta (implementação) como sendo a álgebra de dados. Logo, mostraram como as álgebras de dados podem ser consideradas valores tipados dentro de SOL. O resultado é que as álgebras de dados podem ser consideradas valores de primeira classe.

A ideia central de SOL que suporta esta capacidade é a de *tipo existencial*. Os tipos existenciais permitem conhecer quais são as operação possíveis numa álgebra de dados e como tais operações devem ser utilizadas, sem descrever a implementação das operações ou o tipo usado como *carrier* da álgebra.

Na figura 3.3 apresenta-se um exemplo de definição de uma álgebra de números complexos em SOL.

```
abstype complexo with
        criar : real → real → complexo,
        soma : complexo x complexo → complexo,
        re: complexo → real,
        ima: complexo → real,
is
pack real ∧ real
        λx : real.λy : real.<x, y>
        λz : real ∧ real.λw :real ∧ real.
                <fst(z)+fst(w), snd(z)+snd(w)>
        λz :real ∧ real. fst(z)
        λz :real ∧ real. snd(z)
to ∃ t.[(real → real → t) ∧ (t → t) ∧ (t → real)
                ∧ (t→ real)].
```

Fig. 3.3 : Exemplo de definição de uma álgebra em SOL

Na figura acima os identificadores complexo, criar, soma, re e ima formam a interface da álgebra de dado, cujos elementos são representados como pares de reais (expressão do pack). A expressão começando com pack é a definição da álgebra de dado, i. é, a implementação do TAD.

Enquanto que a expressão que segue a abstype é a especificação do tipo abstrato, e nela são dados os tipos das operações que manipulam o TAD sem ter em conta nenhuma caraterística de implementação.

Por outro lado, na implementação não são utilizados os nomes das operações do tipo abstrato, só são definidas as operações em função do tipo concreto.

A correspondência entre as operações externas do TAD e suas implementações é posicional.

Da figura anterior pode-se ver que o tipo de uma álgebra de dados tem a seguinte forma:

$$\exists\ \tau.\ \sigma_1(\tau) \wedge \sigma_2(\tau) \wedge \ldots \wedge \sigma_n(\tau)$$

onde τ é o nome do TAD e $\sigma_i(\tau)$ são os tipos das operações.

Uma descrição mais detalhada das características da linguagem SOL pode ser encontrada em [PAS 90].

Cardelli e Wegner [CAR 85], também desenvolveram uma linguagem similar a SOL, na qual tipos existenciais são usados para suportar TAD's. Esta linguagem, chamada FUN, apresenta sobre à anterior a vantagem de ter uma sintaxe mais flexível. Como em SOL, a base da linguagem FUN, é o cálculo-λ e a noção de TAD aparece como uma consequência natural de permiteir quantificação de tipo existencial.

A especificação de tipo existencial tem a seguinte forma:

$\exists\ \tau.$ texp(τ); onde texp é uma expressão de tipo que pode conter ocorrências livres da variável de tipo τ. Se alguma variável a tem um tipo existencial desta forma, interpreta-se que para algum tipo τ o tipo de texp(τ) é o tipo de a.

Como FUN inclui registros, nenhuma sintaxe especial é necessária para representar as assinaturas dos TAD's. Assim texp(τ) pode simplesmente ser o tipo de um registro cujos componentes serão as operações permitidas para o TAD.

Uma caraterística interesante de FUN é que todo objeto poderia ser pensado como tendo um tipo existencial, o que levaria a ter um estilo de programação com tipos abstratos.

Assim, por exemplo, dado o par (3,4) que tem o tipo produto (Int x Int), poderia ser representado pela seguinte expressão : ∃ *t*. *t* x *t* ou ∃ *t*. *t*. No primeiro caso, *t* = Int, no segundo *t* = Int x Int, i. é, o mesmo objeto pode ter diferentes tipos existenciais. Para simplificar a verificação de tipos, FUN precisa de uma sintaxe especial para criar objetos cujos tipos são existenciais, de forma que o tipo existencial desejado pelo usuário seja explicitamente indicado.

A sintaxe fornecida por FUN para criar objetos de tipo existencial é baseada na idéia de empacotamento (packaging) da estrutura interna do tipo.

Por exemplo, para representar o par (5, suc) como um objeto abstrato, o tipo existencial seria : ∃ t. (obj: t, op: t → Int), onde a texp é do tipo registro.

Para denotar um objeto existencial utiliza-se a sintaxe :

pack[a=τ, e:texp(τ)]:∃ τ. texp(τ);

Continuando com o exemplo, a definição do par (5,suc) como um objeto existencial seria:

value x = **pack** [t=Int **in** (obj : t, op: T → Int)] (5, suc)

Para utilizar o objeto existencial é só referir a suas componentes pelos nomes, i. é , a expressão x.op(x.obj) da ó (não esquecer que a assinatura do objeto existencial é de tipo registro).

Existe uma restrição no uso do objeto existencial, que é a seguinte, antes de usar o objeto, este deve ser aberto com o operador **open**. No exemplo fica : **open** x **as** valor1[m] **in** valor1.op(valor1.obj), que avalia ó como já foi visto. Esta

sintaxe permite dar um nome local (valor1) ao objeto existencial (x) dentro da expressão que está sendo avaliada, e outro nome local (no exemplo, m) para o tipo da representação oculta.

Pelo apresentado é possível deduzir que os tipos existenciais são tipos ordinários em FUN e os pacotes (objetos existenciais) são valores ordinários que podem ser manipulados em todas as formas usuais.

Um fato interesante, é que por exemplo, o objeto (list(1 2 3), lenght) pode ser representado pelo mesmo tipo existencial do exemplo anterior. O que demostra a flexibilidade de FUN onde os tipos existenciais conseguem captar as similitudes entre objetos diferentes.

4 MODELO ALGÉBRICO

Serão apresentadas neste capítulo, primeiro as estruturas matemáticas que dão suporte ao modelo matemático e em segundo lugar o método de especificação que será utilizado para desenvolver a semântica de RECON-II. Para este fim, utilizar-se-ão três bibliografias básicas : Grätzer [GRÄ 68] e Lipson [LIP 81] para a primeira parte e [GOG 78] para a segunda. Esses livros compreendem a maior parte do presente capítulo e que, devido ao seu caracter matemático, não conterá referências aos mencionados livros. Ficará, portanto, subentendido que toda definição e/ou teorema é obtido diretamente de tal bibliografia.

Cabe destacar que se pressupõe um conhecimento prévio sobre teoría dos conjuntos e conceitos básicos sobre ordens parciais e reticulados. Para o leitor leigo no assunto recomenda-se a leitura prévia da parte I do livro de John D. Lipson [LIP 81].

4.1 Sistemas Algébricos e Abstração

Em termos simples um *sistema algébrico* é um conjunto de elementos com operações definidas sobre eles, por exemplo o conjunto dos naturais com as operações de soma e multiplicação.

Formalmente seria:

Definição 4.1 : Um *sistema algébrico* $\mathcal{S}$ é uma par $\langle \mathcal{A}; \mathcal{F} \rangle$ onde

- $\mathcal{A}$ é um conjunto não vazio, chamado de "carrier".
- $\mathcal{F}$ é uma coleção de operações *finitárias*(*) definidas sobre $\mathcal{A}$, cada operação $f \in \mathcal{F}$ é uma função $\mathcal{A}^n \to \mathcal{A}$, tal que $a_1, a_2, \ldots, a_n \in \mathcal{A}$ e $f(a_1, a_2, \ldots, a_n) \in \mathcal{A}$, onde a "aridade" n de f é um inteiro não negativo que depende de f, i. é $n = n(f)$. $\mathcal{F}$ não é necessáriamente finito e pode ser vazio.

(*) Operação finitária é aquela que possui aridade finita, i. é, a quantidade de argumentos da operação é um número natural.

Um sistema algébrico será denotado por $\langle \mathcal{A}; \mathcal{F} \rangle$ ou simplesmente por $\mathcal{A}$ se o conjunto $\mathcal{F}$ de operações é fixo e conhecido, deixando o contexto decidir quando se fala do conjunto ou quando da álgebra. Também pode-se escrever $\langle \mathcal{A}; f, g, h, \ldots \rangle$. Por exemplo, $\langle \mathbb{N}; +, . \rangle$ denota o conjunto dos naturais com as operações de soma e multiplicação.

A cardinalidade de $\mathcal{A}$ será notada por $|\mathcal{A}|$ e é denominada, no caso de um sistema algébrico, de *ordem* do sistema.

Enquanto a idéia de um sistema algébrico captura a substância da álgebra, a *abstração* captura seu espírito. Três são os níveis de abstração que evoluíram ao longo do desenvolvimento das álgebras. Eles são:

a) Nível Concreto

Este é o nível do mundo empírico das aplicações matemáticas, onde trabalha-se com conjuntos específicos de elementos e operações sobre estes conjuntos que são definidas por regras. Diz-se que um sistema algébrico $\langle \mathcal{A}; \mathcal{F} \rangle$ é conhecido a nível concreto quando conhece-se os elementos do conjunto $\mathcal{A}$ e como avaliar as operações de $\mathcal{F}$ sobre $\mathcal{A}$.

Um exemplo deste nível é o sistema $\langle \mathbb{Z}; +,0 \rangle$, onde $\mathbb{Z}$ é o conjunto dos inteiros, "+" é a operação de soma de inteiros e "0" é a constante zero, que pode ser pensada como uma operação nulária, i. é, de aridade zero. Note-se que o conjunto $\mathcal{A}$ pode ser tanto finito como infinito.

b) Nível Postulacional ou Axiomático

Neste nível um sistema algébrico é definido por *postulados* (*axiomas*, *leis*) as quais especificam propriedades gerais de operações. Na literatura encontra-se sistemas algébricos definidos desta maneira sob o nome de *álgebra moderna*.

Dentro deste nivel de abstração encontra-se os monóides, aneis, etc. Por exemplo, um monóide é um sistema algébrico $\langle \mathcal{M}; .,1 \rangle$ onde "." é uma operação binária e "1" é uma operação nulária que satisfazem duas leis : associatividade e identidade.

Este nível é mais abstrato que o anterior porque tanto o conjunto $\mathcal{M}$ como suas operações são indefinidos, i. é, "desconhecidos". Para esclarecer esta visão, convém mencionar que

o exemplo dado no nível concreto também é um monóide do ponto de vista axiomático.

c) Nível Universal

O nível universal de abstração incrementa generalidade pensando os sistemas algébricos como compostos por um conjunto de valores ($\mathcal{A}$), e um conjunto de operações mas sem nenhum conjunto de axiomas específico que o sistema deva satisfazer como no caso anterior.

4.2 Subálgebras

Antes de entrar no assunto principal do capítulo é necessário discutir brevemente certas características das álgebras e definir outros conceitos que venham a ser de utilidade para o desenvolvimento do modelo.

Definição 4.2:

Seja $\mathbb{S}=(\mathcal{A}, \mathcal{F})$ e $\mathbb{U}=(\mathcal{B}, \mathcal{F})$ álgebras. $\mathbb{U}$ é dita uma *subálgebra* de $\mathbb{S}$ se e somente se $\mathcal{B}$ é um subconjunto não vazio de $\mathcal{A}$ e é fechado para qualquer operação $f \in \mathcal{F}$, i. é:

$$b_1, \ldots, b_n \in \mathcal{B} \Rightarrow f(b_1, \ldots, b_n) \in \mathcal{B}.$$

Escreve-se $\mathcal{B} \leq \mathcal{A}$ para denotar que $\mathcal{B}$ é uma subálgebra de $\mathcal{A}$.(*), e $\mathcal{B} < \mathcal{A}$ para denotar que $\mathcal{B}$ é uma subálgebra *própria* de $\mathcal{A}$: $\mathcal{B} \leq \mathcal{A}$ e $\mathcal{B} \neq \mathcal{A}$.

Proposição 4.1 :

Sejam $\langle \mathcal{B}_i; \mathcal{F} \rangle$, $i \in I$ subálgebras de $\mathbb{S}$, $\mathcal{B} = \bigcap (\mathcal{B}_i \mid i \in I)$.
Se $\mathcal{B} \neq \emptyset$, então $\langle \mathcal{B}; \mathcal{F} \rangle$ é *subálgebra* de $\mathbb{S}$.

(*) Note-se que $\mathcal{A}$ é chamada álgebra, embora seja só o *carrier* da álgebra. Quando não for ambigüo se utilizará indistintamente $\mathcal{A}$ ou $\mathbb{S}$.

A proposição 4.1 implica que se $\mathcal{S}$ é uma álgebra, $H \subseteq \mathcal{A}$, $H \neq \emptyset$, então existe um menor subconjunto $\mathcal{B}$ contendo H tal que $\langle \mathcal{B}; \mathcal{F} \rangle$ é uma subálgebra. Denotar-se-á o subconjunto $\mathcal{B}$ por [H] e a álgebra formada por $\langle [H]; \mathcal{F} \rangle$ chamar-se-á de *álgebra gerada por* H, e H será chamado *conjunto gerador*.

Estendendo a notação [H] para o conjunto vazio:

$[\emptyset] = \emptyset$ se não existem operações nulárias ;

$[\emptyset]$ é subálgebra gerada pelos valores das operações nulárias, se existem as operações nulárias.

Se $\mathcal{A} = [H]$ para algum subconjunto finito H, então se diz que $\mathcal{A}$ é *finitamente gerado* por H.

Proposição 4.2 :

Seja $\mathcal{A}$ uma álgebra e H um subconjunto de $\mathcal{A}$. Seja $\mathcal{K}$ definida como *a família de todas as subálgebras de* $\mathcal{A}$ que inclui H: $\mathcal{K} = \{ \mathcal{B} \leq \mathcal{A} \mid H \subseteq \mathcal{B} \}$. Então

$$[H] = \bigcap \mathcal{K}.$$

Proposição 4.3 :

Seja $\mathcal{A}$ uma álgebra e $H \subseteq \mathcal{A}$. $\mathcal{B} \subseteq \mathcal{A}$ será definido recursivamente por:

a) qualquer $h \in H$ está em $\mathcal{B}$

b) $\forall f \in \mathcal{F}, b_1, \ldots, b_n \in \mathcal{B} \Rightarrow f(b_1, \ldots, b_n) \in \mathcal{B}$;

c) nada mais pertence a $\mathcal{B}$.

Logo, $\mathcal{B} = [H]$.

4.3 Morfismos

A idéia de *morfismo* é a de um mapeamento que preserve estruturas de um sistema algébrico em outro.

Antes de definir morfismo é preciso determinar a noção de *similaridade* entre álgebras. Duas álgebras $\langle \mathcal{A}; \mathcal{F} \rangle$ e $\langle \mathcal{A}'; \mathcal{F}' \rangle$ são *similares* quando existe uma bijeção de $\mathcal{F}$ para $\mathcal{F}'$com a propriedade que operações correspondentes $f \in \mathcal{F}$ e $f' \in \mathcal{F}'$tenham a mesma aridade. Em resumo, duas álgebras são similares quando tem correspondendo operações nulárias, binárias, ternárias, etc. Assim, por exemplo, dois grupos são similares.

Definição 4.3:

Seja $\langle \mathcal{A}; \mathcal{F} \rangle$ e $\langle \mathcal{A}'; \mathcal{F}' \rangle$ álgebras similares.

Uma função ϕ: $\mathcal{A} \longrightarrow \mathcal{A}'$ é um *morfismo* (também chamado *homomorfismo*) de $\langle \mathcal{A}; \mathcal{F} \rangle$ para $\langle \mathcal{A}'; \mathcal{F}' \rangle$ se $\forall f \in \mathcal{F}$ e $\forall a_1, \ldots, a_n \in \mathcal{A}$,

$$\phi\,(f(a_1, a_2, \ldots, a_n)) = f'\,(\phi(a_1), \ldots, \phi(a_n)).$$

Segundo a classe de função que um morfismo pode ser, em particular (injetora, bijetora, sobrejetora), tem-se uma classificação para a noção de morfismo.

Assim, um morfismo se diz *isomórfismo* quando a função ϕ é bijetiva. Desse modo, $\mathcal{A}$ e $\mathcal{A}'$ são *isomórficas*, denotado $\mathcal{A} \cong \mathcal{A}'$, quando ϕ é um isomorfismo. Em outras palavras, $\mathcal{A}$ e $\mathcal{A}'$ são isomórficas quando são idênticas abstratamente, i. é, quando $\mathcal{A}$ difere de $\mathcal{A}'$ apenas nos nomes dos elementos. Logo, qualquer axioma satisfeito por $\mathcal{A}$ deve ser satisfeito por $\mathcal{A}'$. Por exemplo, suponha-se que existe um isomorfismo de $\langle \mathcal{A}; . \rangle$ para $\langle \mathcal{A}', * \rangle$ e "." é associativa, então "*" também deve sê-lo. Isto é satisfeito por definição de isomorfismo.

Continuando com a classificação dos morfismos, o caso seguinte surge quando a função ϕ é sobrejetiva. Nesse caso o morfismo denomina-se *epimorfismo*. Se $\phi : \mathcal{A} \longrightarrow \mathcal{A}'$ é um epimorfismo, $\mathcal{A}'$ é chamada de *imagem epimórfica de* $\mathcal{A}$. $\mathcal{A}'$ é chamado também de *abstração* ou *modelo* de $\mathcal{A}$.

Por último, quando ϕ é injetiva, o morfismo denomina-se *monomorfismo*.

Os morfismos além de preservarem operações, também preservam subálgebras (teorema 4.1) e geradores de álgebras (teorema 4.2).

Teorema 4.1: Morfismos preservam subálgebras.

Seja $\mathcal{A}$ e $\mathcal{A}'$ álgebras e ϕ: $\mathcal{A} \longrightarrow \mathcal{A}'$ um morfismo. Então

a) $\mathcal{B} \leq \mathcal{A} \Rightarrow \phi(\mathcal{B}) \leq \mathcal{A}'$;

b) $\mathcal{C} \leq \mathcal{A}' \Rightarrow \phi^{-1}(\mathcal{C}) \leq \mathcal{A}$.

Teorema 4.2: Morfismos preservam geradores.

Seja ϕ: $\mathcal{A} \longrightarrow \mathcal{A}'$ um morfismo e H um subconjunto de $\mathcal{A}$. Então

$$\phi([H]) = [\phi(H)].$$

Corolário : Seja ϕ: $\mathcal{A} \longrightarrow \mathcal{A}'$ um epimorfismo e H um conjunto de geradores para $\mathcal{A}$. Logo ϕ(H) é um conjunto de geradores para $\mathcal{A}'$.

Teorema 4.3 : Unicidade do Morfismo.

Sejam ϕ ; ψ : $\mathcal{A} \longrightarrow \mathcal{A}'$ morfismos, e H um conjunto de geradores para $\mathcal{A}$. Se $\phi(x) = \psi(x)$ $\forall x \in H$ então $\phi = \psi$.

4.4 Algebras Quocientes

Um problema fundamental em álgebras é determinar as imagens homomórficas de uma álgebra dada. A Algebra Universal oferece uma solução interessante para este problema, baseando-se na noção de *álgebra quociente*.

Definição 4.4 :

Uma *relação de congruência* E é uma relação de equivalência sobre $\mathcal{A}$ que satisfaz a seguinte *propriedade de substituição*: $\forall f \in \mathcal{F}$

$$a_i \; E \; b_i \text{ para } i=1, \ldots, n \Rightarrow f(a_1, \ldots, a_n) \; E \; f(b_1, \ldots, b_n)$$

A definição anterior diz que se operandos correspondentes de uma operação são equivalentes, então também são seus valores.

Uma partição π induzida por uma relação de congruência é chamada uma *partição com a propriedade de substituição*.

Por exemplo, no caso de uma operação binária como a multiplicação, a propriedade de substituição é : $a \; E \; c \wedge b \; E \; d \Rightarrow ab \; E \; cd$.

Teorema 4.4 : Álgebras Quocientes

Seja $\mathcal{A}$ uma álgebra e E a relação de congruência sobre $\mathcal{A}$. Então,

a) O conjunto quociente $\mathcal{A}/E$ é uma álgebra se se define $f \in \mathcal{F}$ sobre $\mathcal{A}/E$ por $f([a_1], \ldots, [a_n]) = [f(a_1, \ldots, a_n)]$;

b) $\mathcal{A}/E$ é uma imagem homomórfica de $\mathcal{A}$ sob o mapeamento natural $\upsilon : a \longmapsto [a]/E$.

Uma álgebra quociente será denotada como :$\mathbb{U}/E = \langle \mathcal{A}/E; \mathcal{F} \rangle$.

Proposição 4.5 :

Seja $\phi:\mathcal{A} \longrightarrow \mathcal{A}'$ um morfismo de álgebras. Então a relação E_ϕ,

$$a \; E_\phi \; b \leftrightarrow \phi(a) = \phi(b)$$

é uma relação de congruência sobre $\mathcal{A}$.

Proposição 4.6 :

Suponha-se que ϕ: $\mathcal{A} \longrightarrow \mathcal{B}$ é um homorfismo de $\mathbb{U}$ para $\mathbb{V}$. Logo, $\langle \mathcal{A}/E; \mathcal{F} \rangle$ é uma subálgebra de $\mathbb{V}$.

Proposição 4.7 :

Sejam $\mathcal{A}$ e $\mathcal{C}$ álgebras e $\mathcal{B}$ uma subálgebra de $\mathcal{A}$. Seja E uma relação de congruência sobre $\mathcal{A}$ e ϕ um homomorfismo de $\mathcal{A}$ para $\mathcal{C}$. Então, E_B, i. é, a restrição de E para $\mathcal{B}$ é uma relação de congruência sobre $\mathcal{B}$ e ϕ, i. é, a restrição de ϕ a $\mathcal{B}$, é um homomorfismo de $\mathcal{B}$ para $\mathcal{C}$.

Proposição 4.8 :

Sejam $\mathcal{A}$, $\mathcal{B}$, e $\mathcal{C}$ álgebras, ϕ um homorfismo de $\mathcal{A}$ para $\mathcal{B}$. e ψ um homomorfismo de $\mathcal{B}$ para $\mathcal{C}$. Então a composta, $\phi\psi$ é um homomorfismo de $\mathcal{A}$ para $\mathcal{C}$.

4.5 Álgebras Parciais

Definição 4.5 :

Uma *álgebra parcial* $\mathbb{U}$ é um par $\langle \mathcal{A}; \mathcal{F} \rangle$ onde $\mathcal{A}$ é um conjunto não vazio e $\mathcal{F}$ é uma coleção de operações parciais sobre $\mathcal{A}$.

O tipo de uma álgebra parcial é definido da mesma maneira que o tipo de álgebras.

Duas álgebras $\mathbb{U}=(\mathcal{A}, \mathcal{F})$ e $\mathbb{S}=(\mathcal{B}, \mathcal{F}')$ do mesmo tipo são *isomórficas* se existe um mapeamento ϕ bijetivo de $\mathcal{A}$ para $\mathcal{B}$ tal que $f_k(a_0, \ldots, a_{k-1})$ existe se e somente se $f_k(a_0\phi, \ldots, a_{k-1}\phi)$ existe e :

$$f_k(a_0, \ldots, a_{k-1})\phi = f_k(a_0\phi, \ldots, a_{k-1}\phi)$$

As álgebras parciais surgem porque a exigência de que todas as operações de $\mathcal{F}$ sejam definida para todos os elementos de $\mathcal{A}$ é

muito restritiva. Desse modo, dada um $f_k \in \mathcal{F}$ e $a_0, \ldots, a_{k-1} \in \mathcal{A}$, se $f_k(a_0, \ldots, a_{k-1}) \in \mathcal{A}$, então a operação é definida e não existe nenhum problema. Mas se $f_k(a_0, \ldots, a_{k-1}) \notin \mathcal{A}$, então se diz que $f_k(a_0, \ldots, a_{k-1})$ é *indefinida*, e a álgebra que surge é parcial.

Teorema 4.5

Seja $\mathbb{U} = \langle \mathcal{A}; \mathcal{F} \rangle$ uma álgebra parcial. Então existe uma álgebra $\mathbb{U}' = \langle \mathcal{A}'; \mathcal{F} \rangle$, e $\mathcal{A}' \subseteq \mathcal{A}$ tal que $\mathbb{U} \cong \mathbb{U}'$.

Em álgebras existe uma única maneira de definir os conceitos de subálgebra, morfismo e relação de congruência. Para as álgebras parciais existem três tipos de subálgebras, três tipos de homomorfismos, e dois tipos de relação de congruência.

Definição 4.6.1 :

Seja $\mathbb{U}$ uma álgebra parcial e seja $\emptyset \neq \mathcal{B} \subseteq \mathcal{A}$. Diz-se que $\mathcal{B}$ é uma *subálgebra* de $\mathcal{A}$ se é fechado para todas as operações $\mathcal{F}$, i. é, se $b_0, \ldots, b_{k-1} \in \mathcal{B}$ e $f_k(b_0, \ldots, b_{k-1})$ é definida em $\mathbb{U}$ então $f_k(b_0, \ldots, b_{k-1}) \in \mathcal{B}$.

Esta definição coincide com a antiga definição de subálgebra para as álgebras.

Definição 4.6.2:

Seja $\mathbb{U}$ uma álgebra parcial e seja $\emptyset \neq \mathcal{B} \subseteq \mathcal{A}$. Define-se f_i sobre $\mathcal{B}$ como segue: $f_i(b_0, \ldots, b_{k-1})$ é definido para $b_0, \ldots, b_{k-1}$ e igual a b sss $f_i(b_0, \ldots, b_{k-1})$ é definida em $\mathbb{U}$ e $f_k(b_0, \ldots, b_{k-1}) = b \in \mathcal{B}$ em $\mathbb{U}$.

Neste caso tem-se que $\mathbb{B}$ é uma *subálgebra relativa* de $\mathbb{U}$ e $\mathbb{U}$ é uma *extensão* de $\mathbb{B}$.

Definição 4.6.3:

Seja $\mathbb{U}$ uma álgebra parcial e $\emptyset \neq \mathcal{B} \subseteq \mathcal{A}$. Suponha que se tem operações parciais f_k' definidas sobre $\mathcal{B}$ tal que se $f_k'(b_0, \ldots b_{k-1}) = b$, então $f_k(b_0, \ldots, b_{k-1}) = b$.

Seja $\mathcal{F}' = \langle f'_c, f'_1, \ldots, f'_k, \ldots \rangle$, então a álgebra $\langle \mathcal{B}; \mathcal{F}' \rangle$ denomina-se *subálgebra fraca*.

A seguir são apresentadas as três noções de homomorfismo:

Definição 4.7.1:

Sejam $\mathcal{A}$ e $\mathcal{B}$ duas álgebras parciais ϕ: $\mathcal{A} \longrightarrow \mathcal{B}$ é chamado um *homomorfismo* de $\mathcal{A}$ em $\mathcal{B}$ se sempre que $f_k(a_0, \ldots, a_{k-1})$ é definida então também é definida $f'_k(a_0\phi, \ldots, a_{k-1}\phi)$ e

$$f'_k(a_0\phi, \ldots, a_{k-1}\phi) = f_k(a_0, \ldots, a_{k-1})\phi$$

Por esta definição, se f_k pode ser operado sobre alguns elementos de $\mathcal{A}$, então f'_k pode ser operado sobre suas imagens. Um homomorfismo é chamado *total* se $f_k(a_0\phi, \ldots, a_{k-1}\phi) = a\phi$, com $a_0, \ldots, a_{k-1} \in \mathcal{A}$, implica que existe $b_0, \ldots, b_{k-1} \in \mathcal{A}$, com $b_0\phi = a_0\phi, \ldots b_{k-1}\phi = a_{k-1}\phi$ e $f_k(b_0, \ldots, b_{k-1}) = b$.

Definição 4.7.2:

Um *homomorfismo forte* ϕ é um homomorfismo tal que $f_k(a_0, \ldots, a_{k-1})$ é definida em $\mathcal{A}$ se e somente se $f'_k(a_0\phi, \ldots, a_{k-1}\phi)$ é definida em $\mathcal{B}$.

Todo homomorfismo forte é um homomorfismo total, mas a inversa é falsa. Todo homomorfismo total é um homomorfismo, mas a inversa é falsa.

Definição 4.7.3:

Seja ϕ um homomorfismo de $\mathcal{A}$ para $\mathcal{B}$, $\mathcal{C} = \mathcal{A}\phi$ e $\mathcal{C}$ a subálgebra relativa correspondente de $\mathcal{B}$. Se ϕ é um isomorfismo de $\mathcal{A}$ para $\mathcal{C}$ então ϕ é chamado de um homomorfismo *embedding* de $\mathcal{A}$ para $\mathcal{B}$.

No caso de álgebras, os três conceitos, aqui apresentados, são equivalentes ao conceito de homomorfismo de álgebra.

Apresenta-se a seguir as definições correspondentes às relações de congruência.

Definição 4.8.1:

Dada uma álgebra parcial $\mathbb{U}$. Uma relação de equivalência E diz-se uma *relação de congruência* se se tem:

$f_k(a_0, \ldots, a_{k-1}) \equiv f_k(b_0, \ldots, b_{k-1})(E)$, sempre que $a_i \equiv b_i(E)$.

Definição 4.8.2:

Uma relação de congruência E sobre $\mathbb{U}$ é chamada *forte* se sempre que $a_i \equiv b_i(E)$, $0 \leq i \leq k$ e $f_k(a_0, \ldots, a_{k-1})$ existe:

$$f_k(b_0, \ldots, b_{k-1}).$$

Proposição 4.9 :

Seja $\mathbb{U}$ e $\mathbb{S}$ álgebras parciais e ϕ um homomorfismo de $\mathbb{U}$ em $\mathbb{S}$. Seja E_ϕ a relação de equivalência induzida por ϕ. Então E_ϕ é uma relação de congruência.

Proposição 4.10 :

Seja $\mathbb{U}$ e $\mathbb{S}$ álgebras parciais e ϕ um homomorfismo forte. Então E_ϕ é uma relação de congruência forte.

4.6 Assinatura de uma Álgebra

Definição 4.9: *Álgebras Heterogêneas*

Uma *Álgebra heterogênea* $\mathbb{H}$ é um par $\langle S; \mathcal{F} \rangle$; onde S é uma familia de conjuntos ou "sorts"(*) e $\mathcal{F}$ é um conjunto de operações tal como foi especificado na definição 4.1.

A partir da definição anterior é possível classificar as álgebras em *homogêneas* e *heterogêneas*, segundo possuam um ou mais "sorts". Se uma álgebra é heterogênea, i. é, possui mais de um "sort", existe o chamado "*sort*" *principal* que é o conjunto de objetos que justifica a existência da álgebra. Por isto, geralmente as álgebras são chamadas de álgebras S-sortidas, onde S é o *sort* principal da álgebra.

(*) Devido uma álgebra heterogênea se caracterizar por ter muitos "sorts", ela é denominada "polisortida" (do inglês *many-sorted*), embora esse termo não exista no português.

Como exemplo, considere a álgebra das pilhas sobre naturais:

S = ⟨ Pilhas, Nat, Bool ⟩

$\mathcal{F}$ = ⟨ criar-pilha; insere-elem, elimina-elem,..., existe-pilha?⟩

É obvio que o conjunto das pilhas (Pilhas) é o conjunto principal nesta álgebra, pois é quem justifica a existência da mesma e além disso todas as operações $f \in \mathcal{F}$ são fechadas nele, i. é: criar-pilha : ⟶ Pilha

insere-elem : Pilha x Nat ⟶ Pilha

elimina-elem : Pilha ⟶ Nat

existe-pilha? : Pilha ⟶ Bool, etc.

Cabe mencionar que todos os conceitos expostos para álgebras homomórficas nas seções anteriores podem ser generalizados para álgebras heterogêneas [GRA 68][LIP 81].

Definição 4.10: *Assinatura de uma Álgebra*

Uma *assinatura* Σ é constituída de dois elementos (S, F), onde S é uma familia não vazía de *sorts* e F é um conjunto não vazío de símbolos de função aos quais está associado uma*funcionalidade* $s_1 X\ s_2 X\ \ldots\ X\ s_n \longrightarrow s_{n+1}$ com $s_1, s_2, \ldots\ s_n, s_{n+1} \in S$ (n ≥ 0).

Utilizar-se-á a notação $\Sigma \subseteq \Sigma'$ para $S \subseteq S'$ e $F \subseteq F'$.

Em geral, toda álgebra é caracterizada por uma *assinatura* Σ.

A figura 4.1 apresenta um exemplo de assinatura que denota à álgebra dos números naturais.

```
sort Nat;
funções :
 zero : ⟶ Nat;
 succ :Nat ⟶ Nat;
```

Fig. 4.1 : [illegible] eros Naturais

[illegible] PASSERINO

Figura 2.2 - pg. 23

O relacion [illegible] álgebra pode ser ilustrado atravé [illegible] a 4.2 é mostrado o ralcionamento e [illegible] na figura 4.1 e a álgebra dos núm [illegible] ento é indicado pela *função de denot*

[illegible]

p. 21. Rede de autômato
25. Figura 2.3
26. " 2.4
30. "

A parte esquerda do diagrama consiste de nomes dentro da assinatura, enquanto que a parte direita contém uma álgebra, cujos elementos (objetos e funções) formam o mundo abstrato e são representados entre aspas. As setas representam funções .

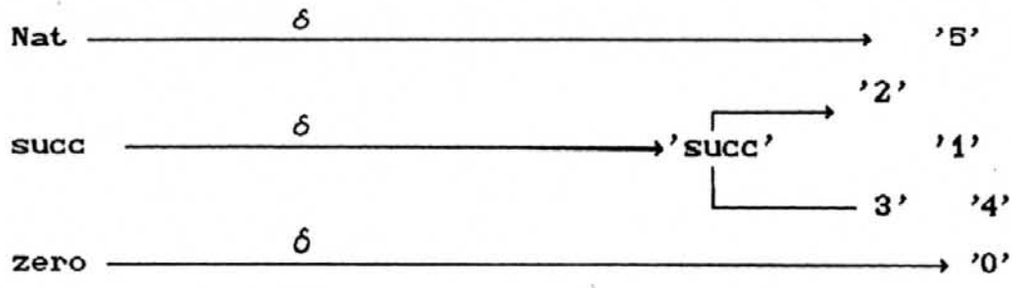


Fig. 4.2: Relação entre uma assinatura e sua álgebra

Note que, na figura 4.2, foi dada só uma das muitas possíveis definições da função de denotação para mapear a assinatura dada na álgebra dos naturais. Mais ainda, não existe correspondência um a um entre assinaturas e álgebras. Muitas assinaturas podem denotar uma mesma álgebra e uma assinatura pode denotar várias álgebras, éstas últimas recebem o nome de *álgebras similares*, e determinam um conjunto que se denomina *variedade* [HOR 89].

Para exemplificar isto, compare-se a seguinte assinatura, que denota álgebra dos números naturais com a assinatura que foi definida na figura 4.1 e o relacionamento da mesma com a mencionada álgebra que aparece na figura 4.4.

```
sort N;
funções :
 start: ⟶ N;
 next :N ⟶ N;
```

Fig. 4.3 : Outro exemplo de Assinatura dos Naturais

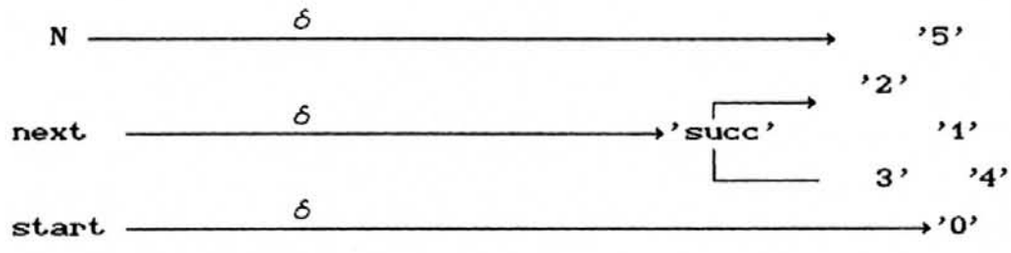


Fig. 4.4 : Relação entre uma assinatura e a álgebra dos naturais

A assinatura da figura 4.3 pode também denotar a algebra formada por um conjunto unitário e a função identidade, como se pode observar na figura 4.5

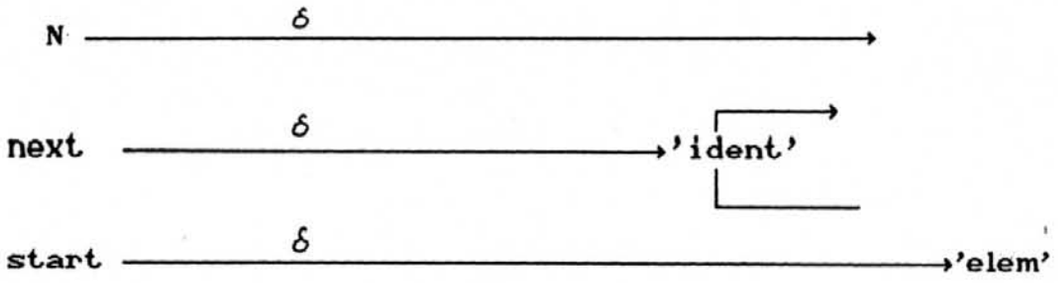


Fig. 4.5: Outra relação entre assinatura e álgebra

Como cada assinatura pode denotar muitas álgebras diferentes, de muitas maneiras diferentes, é necessário desenvolver um mecanismo matemático que permita associar uma única álgebra a cada assinatura.

4.7 Álgebra Livre

Definição 4.11: *Álgebra-Σ*

Dada $\Sigma=\langle S,F\rangle$, uma álgebra-Σ $\mathbb{U}=\langle \mathcal{A}_s ; \mathcal{F}\rangle$ consiste de uma família não vazía de conjuntos $\mathcal{A}_{si}$ para cada $s_i \in S$ ($\mathcal{A}_{si}$ é chamado de *carrier* de $\mathcal{A}$ do *sort* s) e de um conjunto não vazío $\mathcal{F}$ com f_A: $\mathcal{A}_{s1} \times \mathcal{A}_{s2} \times \ldots \times \mathcal{A}_{sn} \longrightarrow \mathcal{A}_s$ para cada $f \in F$, i. é, as funcionalidades de $f \in F$ coincidem com as de $f \in \mathcal{F}$.

Definição 4.12: *Homomorfismo-Σ*

Se $\mathbb{U}$ e $\mathbb{V}$ são ambas álgebras-Σ, um *homomorfismo-Σ* $\phi : \mathbb{U} \longrightarrow \mathbb{V}$ é uma família de funções $\langle \phi_s : \mathcal{A}_s \longrightarrow \mathcal{B}_s \rangle_{s \in S}$ que preserva as operações , i. é, satisfaz:

- Se $f \in F$ então $\phi_s(f_A) = f_B$;
- Se $f \in F$ e $\langle a_1, \ldots, a_n \rangle \in \mathcal{A}_{s1} \times \mathcal{A}_{s2} \times \ldots \times \mathcal{A}_{sn}$ então $\phi_s[f_A(a_1, \ldots, a_n)] = f_B[\phi_{s1}(a_1), \ldots, \phi_{sn}(a_n)]$.

Definição 4.13: *Categoria das álgebras-Σ.*

O conjunto das álgebras denotadas por uma assinatura Σ junto com um homomorfismo definido sobre todas as álgebras (incluindo o homomorfismo identidade) determina o que se chama *categoría de álgebras* **C**, sobre Σ.

Definição 4.14: *Álgebra Inicial*

Uma álgebra $\mathbb{I}$ é denominada *inicial* na categoria **C**, sobre Σ, se e somente se $\mathbb{I} \in$ **C** e para toda álgebra $\mathbb{B}$pertencente à **C** existe um único homomorfismo de $\mathbb{I}$ para cada álgebra $\mathbb{B}$ em **C**.

É possivel demostrar que dada uma categoria **C**, sobre Σ, sempre existe uma álgebra inicial e esta é única, i. é, se existe $\mathbb{I}$ e $\mathbb{I}'$ em **C** deve existir um ϕ: $\mathbb{I} \longrightarrow \mathbb{I}'$ bijetivo (ou seja ϕ é um isomorfismo). Isto é enunciado na proposição seguinte:

Proposição 4.11:

Se álgebras $\mathcal{A}$ e $\mathcal{A}'$ são ambas iniciais na categoria **C** de algebras-Σ, então $\mathcal{A}$ e $\mathcal{A}'$ são isomórficas. Se uma álgebra $\mathcal{A}''$ em **C** é isomórfica com uma álgebra $\mathcal{A}$ que é inicial em **C**, então $\mathcal{A}''$ é também inicial.

Definição 4.15: *Álgebra Terminal*

Uma álgebra-Σ, $\mathbb{Z} \in$ **C** é chamada *terminal* em **C** se para todo $\mathbb{A} \in$ **C** existe pelo menos um homomorfismo-Σ ϕ: $\mathbb{A} \longrightarrow \mathbb{Z}$.

Broy define a noção de *álgebra inicial fraca* e *álgebra terminal fraca* a partir do conceito de *homomorfismo fraco* [BRO 82]. Uma familia ϕ $(\phi_k : s_k \longrightarrow s_k')$ de operações, possivelmente parciais, é chamada de homomorfismo *fraco* de $\langle S; F\rangle$ para $\langle S', F'\rangle$ se para todo $f: s_1 X\ s_2\ X\ \ldots\ X\ s_n \longrightarrow s_{n+1} \in F$ com $s_1, s_2, \ldots s_n, s_{n+1} \in S$ $(n \geq 0)$ e para todo $a_1 \in s_1 \ldots a_n \in s_n$

$$f'(\phi_{s1}(a_1), \ldots \phi_{sn}(a_n)) \text{ definido} \rightarrow f(a_1 \ldots a_n) \text{ definida}$$

Entre duas álgebras-Σ existe pelo menos um homomorfismo total ou um homomorfismo fraco. Se existe tanto um homomorfismo fraco quanto um total, ambos são idênticos.

Definição 4.16: *Redução-Σ, Extensão-Σ.*

Seja $\mathbb{U}$ uma álgebra-Σ, $\mathbb{U}$ é chamada de *redução*-Σ de uma álgebra-Σ, $\mathbb{U}'$, se $\Sigma \subseteq \Sigma'$ e $s_i = s_j'$ e $f_n = f_m'$ $\forall s \in S;\ s' \in S';\ f \in F;\ f' \in F'$.

$\mathbb{U}'$ denomina-se a *extensão-Σ de* $\mathbb{U}$.

Note que a diferença com a noção de subálgebra é que a redução é formada por seleção de uma subfamília de conjuntos e de funções, enquanto que a subálgebra é constituída a partir dos subconjuntos de *todos* os conjuntos da álgebra e das restrições de *todas* suas funções.

Definição 4.17: *Termo Bem Formado-Σ (tbf-Σ), Termo Básico.*

Seja $\Sigma = \langle S, F \rangle$ uma assinatura e $X = \langle X_s \rangle$ a família de conjuntos de identificadores dos conjuntos $s \in S$.
O conjunto dos *termos bem formados*-Σ é definido como o menor conjunto que satisfaz:

i) $\forall x \in X_s$, x é um tbf-Σ;

ii) $\forall f \in F$ tal que $f: \longrightarrow s$, f é um tbf-Σ;

iii) $\forall f: s_1 X s_2 X \ldots X s_n \longrightarrow s_{n+1} \in F$ $(n \geq 0)$ e para todos os tbf-Σ $t_1, \ldots, t_n$ dos conjuntos $s_1, s_2, \ldots, s_n$ $f(t_1, \ldots, t_n)$ é tbf-Σ.

Um *termo básico* é um tbf-Σ sem variáveis livres.

Qualquer tbf-Σ pode ser interpretado numa álgebra-Σ. Tal interpretação associa a cada termo ou um valor ou indefinido.

Definição 4.18: *Interpretação de um termo-Σ*

Seja $\mathbb{U}$ uma álgebra-Σ e $V = \langle V_s : X_s \longrightarrow S \rangle$ uma função de avaliação. Uma *interpretação* $\Im$ de t (tbf-Σ) é dada por:

i) $\Im(t) = V_s(x)$, $\forall x \in X_s$;

ii) Se $t = f$ com $f : \longrightarrow S$, então $\Im(t) = [\![f]\!]$;

iii) Se $t = f(t_1, \ldots, t_n)$, então
$\Im(t) = [\![f]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$.

Definição 4.19: *Álgebra Livre W(Σ, X)*

Álgebra Livre $W(\Sigma, X)$ é a álgebra construida a partir de Σ e de X, tal que seus conjuntos consistem exatamente de todos os tbf dos conjuntos s respetivos e para todas as funções $f^{W(\Sigma, X)}$ verifica-se :

$$f^{W(\Sigma, X)}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n).$$

A álgebra $W(\Sigma) = W(\Sigma, \emptyset)$ de termos básicos denomina-se *Álgebra de termo.*

A álgebra inicial da categoria sobre a assinatura sempre existe e ela é a álgebra livre da assinatura. Isto foi enunciado por [GOG 78] no seguinte teorema:

Teorema 4.6 : ***Existência da Álgebra Inicial***

W(Σ, *X*) é uma álgebra inicial na categoria $\mathbf{Alg}_{\Sigma}$ de todas as álgebras-Σ.

4.8 Tipos Abstratos de Dados

Definição 4.20: ***Tipos Abstratos de Dados***

Um *Tipo Abstrato (de dado)* T é a classe isomórfica da álgebra inicial na categoria $\mathbf{Alg}_{\Sigma}$ das álgebras-Σ.

Mais ainda, um *Tipo Abstrato de Dados (TAD)* T é um par (Σ, E), onde Σ é a assinatura de uma álgebra-Σ e E é um conjunto de fórmulas universalmente quantificadas.

Analisando a definição anterior é possível ver que se uma assinatura com axiomas (alguns autores chamam isto de *apresentação*) é um TAD, então uma álgebra é um *objeto abstrato* ou uma *instância* do tipo, já que, como foi mencionado antes, uma assinatura *denota* muitas álgebras. Da mesma maneira um tipo *denota* muitos objetos, ou melhor, muitos objetos *pertencem* (no sentido de conjunto) a um certo tipo.

Note que uma *apresentação* é uma assinatura extendida com axiomas [HOR 89]. Uma álgebra é *denotada por uma apresentação* se é denotada pela assinatura da presentação. Uma álgebra denotada pela apresentação *satisfaz* os axiomas de sua apresentação.

A *categoria das álgebras* sobre uma dada apresentação, chamada $\mathbf{Alg}_{\Sigma,E}$, é um conjunto de álgebras junto com um número de homomorfismos entre elas incluído o homomorfismo identidade.

Uma álgebra 𝔸 é *inicial* numa categoria $\mathbf{Alg}_{\Sigma,E}$, sobre uma apresentação, se e somente se 𝔸 pertence a $\mathbf{Alg}_{\Sigma,E}$ e para cada álgebra 𝔹 em $\mathbf{Alg}_{\Sigma,E}$, existe um único homomorfismo de 𝔸 para 𝔹. A

álgebra inicial existe e é única a menos de isomorfismo [GOG 78] [EHR 80].

Definição 4.21: *Congruência-Σ*

Uma *congruência-Σ*, ≡, sobre uma álgebra-Σ $\mathcal{A}$, é uma família $\langle \equiv_s \rangle_{s \in S}$ de relações de equivalência, $\equiv_s$ sobre $\mathcal{A}_s$ para $s \in S$, tal que se $f \in F$ se $a_i, a_i' \in \mathcal{A}_{si}$, com $a_i \equiv a_i'$ para i=1, ..., n, então $f_A(a_1, ..., a_n) \equiv_s f_A(a_1', ..., a_n')$.

Se $\mathcal{A}$ é uma álgebra-Σ e ≡ é uma congruência-Σ sobre $\mathcal{A}$, seja $(\mathcal{A}/\equiv)_s = \mathcal{A}_s/\equiv_s$ o conjunto de classes de equivalência-$\equiv_s$ de $\mathcal{A}_s$. Para $s \in \mathcal{A}_s$ seja [a] a classe de equivalência-$\equiv_s$ contendo a.

Proposição 4.12

Se $\mathcal{A}$ é uma álgebra-Σ e ≡ é uma congruência-Σ sobre $\mathcal{A}$, então $\mathcal{A}/\equiv$ é uma álgebra-Σ, chamada *quociente* de $\mathcal{A}$ por ≡.

Proposição 4.13

Seja $\mathcal{A}$ uma álgebra-Σ e R uma relação sobre $\mathcal{A}$. Então existe a menor relação de congruência sobre $\mathcal{A}$ contendo R, chamada *relação de congruência gerada por* R sobre $\mathcal{A}$.

Teorema 4.7

Seja $\mathcal{E}$ uma presentação-Σ e $\equiv_{\mathcal{E}}$ a relação de congruência-Σ sobre $W(\Sigma)$ gerada por $\mathcal{E}(W(\Sigma))$. Então $W(\Sigma)/\equiv_{\mathcal{E}}$, quociente de $W(\Sigma)$ por $\equiv_{\mathcal{E}}$, denotada por $W(\Sigma,E)$, ou $W(\mathcal{E})$, é a álgebra inicial na categoria $\mathbf{Alg}_{\Sigma,\mathcal{E}}$ de todas as álgebras que satisfazam $\mathcal{E}$.

Definição 4.22: *Especificação de um TAD*

Uma *especificação* de um TAD T é uma tripla ⟨S, Σ, E⟩onde Σ é uma assinatura S-sortida, S é *sort* principal e E é um conjunto de equações-Σ.

Definição 4.23:

Seja ⟨S, Σ, E⟩ e ⟨S∪S', Σ∪Σ', E'⟩ especificações com S∩S'=∅ e Σ∩Σ'=∅. Seja S#=S∪S', Σ#=Σ∪Σ' e E#=E∪E'. Chamar-se-á ⟨S#, Σ#, E#⟩*extensão* de ⟨S, Σ, E⟩ se $W(\Sigma, E) \cong W(\Sigma\#, E\#)$ como álgebras-Σ S-sortidas (esquecendo os conjuntos em S'e as operações em Σ').

Se $S'=\emptyset$ então $\langle S, \Sigma\#, E\# \rangle$ chamar-se-á um *enriquecimento* de $\langle S, \Sigma, E \rangle$.

4.9 Estruturas de Primeira Ordem

Definição 4.24: *Estruturas de Primeira Ordem*

Uma *Estrutura de Primeira Ordem* $\mathfrak{U}$ é uma tripla $\langle \mathcal{A}; \mathcal{F}, \mathcal{R} \rangle$, onde $\mathcal{A}$ é um conjunto não vazío.

$\mathcal{F}$ é uma coleção de operações *finitárias* definidas sobre $\mathcal{A}$, cada operação $f \in \mathcal{F}$ é uma função $\mathcal{A}^n \rightarrow \mathcal{A}$, i. é, $a_1, a_2, \ldots, a_n \in \mathcal{A}$ implica $f(a_1, a_2, \ldots, a_n) \in \mathcal{A}$, onde a "aridade" n de f é um inteiro não negativo que depende de f, i. é $n = n(f)$. $\mathcal{F}$ não é necessáriamente finito e pode ser vazío.

E $\mathcal{R}$ é uma família de relações finitárias sobre $\mathcal{A}$.

Uma álgebra ou sistema algébrico como foi definido no início deste capítulo é um caso particular de uma estrutura de primeira ordem onde o conjunto $\mathcal{R}$ das relações é vazío.

Por outro lado, quando o conjunto $\mathcal{F}$ das operações é vazío a estrutura de primeira ordem consiste no que se chama de *sistema relacional*.

Definição 4.25: *Sistema Relacional*

Um *Sistema Relacional* $\mathbb{S}$ é um par $\langle \mathcal{A}; \mathcal{R} \rangle$ onde $\mathcal{A}$ é um conjunto não vazío e $\mathcal{R}$ é uma família de relações finitárias sobre $\mathcal{A}$. Se $\mathcal{R} = \langle r_0, \ldots, r_{n-1} \rangle$ se escreverá $\langle \mathcal{A}; r_0, \ldots, r_{n-1} \rangle$ para $\mathbb{S}$.

Cabe destacar que todos os conceitos mencionados antes para álgebras universais (subálgebras, morfismos, assignatura, etc.) podem ser generalizados para estruturas. A seguir apresenta-se um breve resumo de tais conceitos. Para mais detalhe ver capitulo seis em [GRA 68].

De modo análogo às álgebras, as estruturas de primeira ordem podem ser classificadas em *homogêneas* e *heterogêneas*. Uma

estrutura de primeira ordem é *heterogênea* ou *"polisortida"*, quando $\mathcal{A}$ é uma família de conjuntos, i. é, $\mathcal{A} = \bigcup_{i=0}^{n} \mathcal{A}_i$.

Quando nada for dito a respeito, considerar-se-á que $\mathcal{A}$ representa uma família de conjuntos.

Seja $\mathfrak{U} = \langle \mathcal{A}; \mathcal{F}, \mathcal{R} \rangle$ e $\mathfrak{B} = \langle \mathcal{B}; \mathcal{F}', \mathcal{R}' \rangle$ estruturas de primeira ordem, diz-se que $\mathfrak{B}$ é uma *subestrutura* de $\mathfrak{U}$ se $\langle \mathcal{B}; \mathcal{F}' \rangle$ é uma subálgebra de $\langle \mathcal{A}; \mathcal{F} \rangle$ e $r_i' \in \mathcal{R}'$ é a *restrição*(*) de $r_i \in \mathcal{R}$ para $\mathcal{B}$. Se $\mathfrak{B}$ é uma subestrutura de $\mathfrak{U}$ se diz, então, que $\mathfrak{U}$ é uma *extensão* de $\mathfrak{B}$. Se $\mathfrak{U}$ é uma estrutura, $\mathcal{B} \subseteq \mathcal{A}$, $\langle \mathcal{B}; \mathcal{F} \rangle$ uma subálgebra de $\langle \mathcal{A}; \mathcal{F} \rangle$, então $\mathfrak{B}$ denotará a subestrutura correspondente. Em particular se $\mathfrak{B}$ é um sistema relacional, então para cada $\emptyset \neq \mathcal{B} \subseteq \mathcal{A}$, se tem uma subestrutura $\mathfrak{B}$.

Uma estrutura $\mathfrak{B}$ é chamada *subestrutura fraca* de uma estrutura $\mathfrak{U}$ se $\mathcal{A} \subseteq \mathcal{B}$, $f' \subseteq f|_B$ (**) e $r' \subseteq r|_B$.

Uma estrutura $\mathfrak{U} = \langle \mathcal{A}; \mathcal{F}, \mathcal{R} \rangle$ é chamada *estrutura parcial* quando o sistema algébrico correspondente é parcial, i. é, existe pelo menos um $f \in \mathcal{F}$ tal que f é parcial.

Se ϕ é um mapeamento de $\mathcal{A}$ para $\mathcal{B}$, então ϕ é um *homomorfismo* de $\mathfrak{U}$ em $\mathfrak{B}$ se é um homomorfismo de $\langle \mathcal{A}; \mathcal{F} \rangle$ em $\langle \mathcal{B}; \mathcal{F}' \rangle$ e $r_i(a_1, a_2, \ldots, a_n)$, com $a_1, a_2, \ldots, a_n \in \mathcal{A}$, implica que $r_i'(\phi(a_1), \phi(a_2), \ldots, \phi(a_n))$. Se ϕ é surjetivo e $r'_i(b_1, b_2, \ldots, b_n)$, com $b_1, b_2, \ldots, b_n \in \mathcal{B}$, implica a existência de $a_1, a_2, \ldots, a_n \in \mathcal{A}$ tal que $r_i(a_1, a_2, \ldots, a_n)$ e $\phi(a_1) = b_1, \ldots, \phi(a_n) = b_n$, então $\mathfrak{B}$ é a *imagem homomórfica* de $\mathfrak{U}$.

A estrutura $\mathfrak{U} = \langle \mathcal{A}; \mathcal{F}, \mathcal{R} \rangle$ é uma *expansão* de $\mathfrak{U}' = \langle \mathcal{A}; \mathcal{F}; \mathcal{R}' \rangle$ se $r_i \supseteq r_i'$ $\forall i = 1, n$, onde $r_i \in \mathcal{R}$ e $r_i' \in \mathcal{R}$. Em outras palavras, se os conjuntos base e as operações são idênticas, mas as relações são expandidas. $\mathfrak{B}$ é uma *imagem homomórfica expandida* de $\mathfrak{U}$, se $\mathfrak{B}$ é uma expansão da imagem homomórfica de $\mathfrak{U}$, ou seja, existe um epimorfismo de $\mathfrak{U}$ para $\mathfrak{B}$.

(*) Se r é uma relação n-ária sobre $\mathcal{A}$ e $\mathcal{B} \subseteq \mathcal{A}$, então $r_{\mathcal{B}} = r \cap \mathcal{B}^n$ é uma relação n-ária sobre $\mathcal{B}$, chamada *restrição* de r para $\mathcal{B}$.

(**) $f|_B$ significa f aplicado aos elementos do conjunto $\mathcal{B}$, deve-se lembrar que $f \in \mathcal{F}$ definido sobre $\mathcal{A}$.

ϕ é um *isomorfismo* de $\mathfrak{U}$ e $\mathfrak{B}$ se $\mathfrak{b}$ é uma imagem homomórfica de $\mathfrak{U}$ com respeito a ϕ e ϕ é bijetiva.

Uma *assinatura* Ω para uma estrutura de primeira ordem $\mathfrak{U} = \langle \mathcal{A}; \mathcal{F}, \mathcal{R} \rangle$ consiste de um par $\Omega = (\Sigma, R)$, onde Σ é a assinatura (S, F) da álgebra correspondente $\langle \mathcal{A}; \mathcal{F} \rangle$ e R é um conjunto não vazío de símbolos de relações aos quais está associado uma cardinalidade $r \subseteq s_1 \times s_2 \times \ldots \times s_n, \forall r \in R$.

Como no caso da assinatura de uma álgebra utilizar-se-á a notação $\Omega \subseteq \Omega'$ para indicar $\Sigma \subseteq \Sigma'$ e $R \subseteq R'$.

Note também, que de maneira equivalente às álgebras, uma estrutura pode ser denotada por várias assinaturas diferentes e que uma assinatura particular denota muitas estruturas. Neste caso as estruturas denominm-se *estruturas similares* e o conjunto que elas determinam é chamado de *variedade* (ver seção 4.6).

Define-se uma *categoria de estrutura* $\mathfrak{K}$ como um par (V, H), onde V é a variedade de uma assinatura Ω e H é uma família de homomorfismos (incluindo o homomorfismo identidade) definido sobre todas as estruturas de variedade.

A partir do conceito anterior é possível definir *estrutura inicial* e *estrutura terminal* da mesma forma em que foi feito na seção 4.7

Na seção anterior apresenta-se uma série de conceitos muito importantes relacionados com álgebra-Σ, da mesma maneira, como foi realizado antes, é fatível estender tais noções para as estruturas. Assim, dada $\Omega = (\Sigma, R)$, uma estrutura heterogênea $\mathfrak{U} = \langle \mathcal{A}; \mathcal{F}, \mathcal{R} \rangle$ é uma *estrutura-Ω* se $\langle \mathcal{A}; \mathcal{F} \rangle$ é álgebra-Σ e as aridades de $r \in R$ coincidem com as aridades de $\mathfrak{r} \in \mathcal{R}$.

Seja uma estrutura-Ω $\mathfrak{U}$, $\mathfrak{U}$ é uma *redução-Ω* de $\mathfrak{B}$ se $\Omega \subseteq \Omega'$ e (S, F) é redução-Σ de (S', F') e $r_n = r'_m \quad \forall r \in R \; ; r' \in R'$.

$\mathfrak{B}$ é uma *extensão* de $\mathfrak{U}$.

Seja $\Omega = (\Sigma, R)$ com $\Sigma = (S, F)$ e $X = \{X_s\}$ uma família de conjuntos de identificadores dos conjuntos $s \in S$. O conjunto dos *termos bem formados-Ω* é definido como o menor conjunto que satisfaz:

i) $\forall x \in X_s$, x é um tbf-Ω;

ii) $\forall f \in F$ tal que $f: \longrightarrow s$, f é um tbf-Ω;

iii) $\forall f: s_1 X s_2 X \ldots X s_n \longrightarrow s_{n+1} \in F$ $(n \geq 0)$ e para todos os tbf-Ω $t_1, \ldots, t_n$ dos conjuntos $s_1, s_2, \ldots, s_n$ $f(t_1, \ldots, t_n)$ é tbf-Ω.

iv) $\forall r \subseteq s_1 X s_2 X \ldots X s_n \in R$ $(n \geq 0)$ e para todos os tbf-Ω $t_1, \ldots, t_n$ dos conjuntos $s_1, s_2, \ldots, s_n$ $r(t_1, \ldots, t_n)$ é tbf-Ω.

Um *termo básico* é um tbf-Ω sem variáveis livres.

Seja $\mathfrak{A}$ uma estrutura-Ω e $V = (V_A: X_A \longrightarrow \mathcal{A})$ uma função de avaliação. Logo, a *interpretação* $\mathfrak{I}$ de t é :

i) $\mathfrak{I}(t) = V_A(x)$, $\forall x \in X_A$;

ii) Se $t = f$ com $f: \longrightarrow \mathcal{A}$, então $\mathfrak{I}(t) = [\![f]\!]$;

iii) Se $t = f(t_1, \ldots, t_n)$, então
$\mathfrak{I}(t) = [\![f]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$;

iv) Se $t = r(t_1, \ldots, t_n)$, então
$\mathfrak{I}(t) = [\![r]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$;

Uma *Estrutura Livre* $E(\Omega, X)$ é a estrutura construida a partir de Ω e de X, tal que seus conjuntos consistem exatamente de todos os tbf dos conjuntos s respectivos e para todas as funções $f^{L(\Omega,X)}$ verifica-se :

$$f^{L(\Omega, X)}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n).$$

Quando $X=\emptyset$ então estrutura denomina-se *Estrutura de termo.*

Um *Tipo Abstrato de Primeira Ordem (TPO)* T é um par (Ω, E), onde Ω é a assinatura de uma estrutura de primeira ordem-Ω e E é um conjunto de fórmulas universalmente quantificadas.

5 SEMÂNTICA DE RECON-II

5.1 Introdução

Definir um Sistema de Tipos para uma linguagem requer conhecer as expressões significativas que podem ser geradas a partir dos construtores da linguagem, e especialmente determinar quando duas expressões são equivalentes, i. é, possuem o mesmo significado. Em outras palavras, é necessário determinar a semântica das expressões da linguagem previamente para poder definir o Sistema de Tipos.

Especificar uma semântica para uma linguagem significa dar regras para traduzir a linguagem em outra (metalinguagem) para a qual é conhecida uma semântica fixa.

Contudo, pode ser preferível definir uma classe de modelos semânticos numa forma algébrica "abstrata" por meio de equações condicionais de primeira ordem.

São bem conhecidas as vantagens que decorrem do uso da abstração em qualquer processo de modelagem: incremento de flexibilidade, concentração nas características essenciais, verificabilidade, modularização, etc.

No entanto quando estas técnicas são aplicadas na especificação de uma linguagem, adicionam-se outros benefícios:

- Como a semântica é dada por meio de relações de congruência sobre a linguagem, se tem diretamente a noção de "programas equivalentes".
- Não existe uma metalinguagem adicional que tenha de ser estudada previamente (a metalinguagem é a matemática e supõe-se que o leitor está familiarizado com a mesma).
- A definição semântica emprega as mesmas técnicas que a especificação de TAD. Isto fornece um esquema algébrico coerente tanto para as estruturas de dados como para as de controle.

É claro que a teoria dos tipos abstratos de dados deve ser estendida para permitir expressar, por exemplo, recursividade mútua. Em outras palavras, é necessário permitir a caracterização dos modelos semânticos que correspondem ao menor ponto fixo na

classe de todos os modelos do tipo abstrato sob consideração [WIR 80].

Como foi descrito no capítulo anterior, o modelo (inicial) de um tipo abstrato de dados é sua álgebra inicial, mas existe também uma classe de modelos não isomórficos que vão da álgebra inicial à algebra terminal (modelo terminal). Neste trabalho a semântica de RECON-II será fornecida apenas pelo modelo inicial, deixando para um trabalho futuro definir a classe dos modelos de RECON-II.

O método utilizado neste trabalho é o apresentado por Broy em [BRO 87], que especifica uma linguagem algebricamente por um tipo T e define sua semântica como a classe de modelos do tipo T. Isto pode ser feito tanto para linguagens procedurais [BRO 79] como para linguagens declarativas [WIR 80], [BRO 82].

Os métodos algébricos associam a uma linguagem um único modelo semântico (a menos de isomorfismo) ou uma classe de modelos não isomórficos.

Se os modelos semânticos são "equivalentes extensionalmente" definirão o mesmo comportamento extensional ("observável") para o programa embora a equivalência de programas possa ser bastante diferente. [BRO 87] [BRO 82] [MIL 78]

O método algébrico fornece, desta maneira, um meio para comparar diferentes modelos semânticos e analisar seus relacionamentos. Cabe destacar que, embora a definição da equivalência extensional seja muito interessante do ponto de vista dos modelos semânticos, ela não será incluida no presente trabalho por escapar ao seu escopo.

Antes de começar com a especificação da linguagem RECON-II é necessário descrever primeiro o método que será utilizado, assim como, também, a linguagem de especificação (metalinguagem) adotada.

Em geral, uma especificação algébrica forma um grafo dirigido. As assinaturas representam os nodos, e os relacionamentos de dependências entre elas representam os arcos do grafo. Um caso especial é a *especificação hierárquica* onde o grafo dirigido é acíclico. Se uma assinatura A é direta ou indiretamente utilizada na construção de outra assinatura B, a assinatura B não pode ser utilizada direta ou indiretamente na construção da assinatura A. Em outras palavras, duas assinaturas não podem ser mutuamente recursivas. Usar especificações hierárquicas é um método útil de construção porque a complexidade que a mente humana tem que considerar num dado nível é consideravelmente menor que o da especificação total. É mais simples focalizar, primeiro as assinaturas primitivas, para depois considerar aquelas que usam diretamente as primitivas e assim sucessivamente. Logo, não é necessário entender a especificação como um todo de uma única vez, senão que é possível compreendê-la através de um processo de construção por nível, também chamado *bottom-up*.

Entretanto, construir uma especificação como um grafo acíclico nem sempre é possível por causa da recursividade mútua entre assinaturas. Para poder continuar gozando das vantagens da especificação hierárquica, sem ter que restringir seu uso para casos simples, onde não exista recursão mútua, adotou-se a solução proposta por Horebeek em [HOR 89]. Horebeek transforma o grafo cíclico numa hierarquia, agrupando as assinaturas mutuamente recursivas numa superassinatura, chamada *cluster*. Um *cluster* é simplesmente um pacote de assinaturas individuais que pertencem a um laço num grafo dirigido e que devem ser consideradas simultaneamente. Em outros termos, a ordem parcial da hierarquia é substituída por uma pré-ordem.

No início deste capítulo foi dito que uma das vantagens do uso do método algébrico para dar a semântica de uma linguagem é que não existe metalinguagem adicional. Isto é verdade só em parte, porque para especificar objetos simples, como por exemplo os números naturais, uma notação matemática é suficientemente clara. Mas quando se quer especificar objetos mais complexos, a

simples notação matemática diminui a legibilidade da especificação. Por isto é conveniente complementá-la com algumas construções sintáticas que contribuam à compreensão.

De tal maneira foram incorporadas certas construções , algumas das quais existem na literatura corrente [HOR 89], [MIL 78], [BRO 87] e outras surgiram como necessidades naturais do processo de especificação. Cabe destacar que não é objetivo deste capítulo fornecer a semântica do método empregado, mas sim dar intuitivamente o significado de cada construção para melhor comprensão do texto.

O motivo pelo qual não foi adotada uma das tantas linguagens de especificação algébrica existentes na literatura reside no fato de não se querer influenciar este trabalho com as limitações de uma sintaxe particular. Além disso, na opinião particular da autora, essas linguagens de especificação parecem-se mais com linguagens de programação (declarativas) que com estruturas matemáticas. Logo, o uso de alguma linguagem de especificação particular exigiria, de parte do leitor, conhecimentos sobre conceitos de programação, além de um estudo prévio da sintaxe da linguagem e dispersaria à atenção do mesmo em detalhes não relevantes desde o ponto de vista matemático e que estão mais relacionados à implementação das especificações. Portanto, fiel à crença de que um método algébrico não requer estudo prévio de nenhuma metalinguagem, a autora optou por construir sua própria sintaxe, conservando-se o mais perto possível da notação algébrica tradicional.

Contudo as construções sintáticas utilizadas serão introduzidas paulatinamente na medida do necessário, evitando dessa maneira sobrecarregar o leitor de muita informação acessória ao objetivo do trabalho.

Este capítulo é formado por duas seções importantes, a seção 5.2, onde são apresentadas as características de uma linguagem do ponto de vista de tipos hierárquicos e a seção 5.3, onde apresenta-se a especificação algébrica da RECON-II.

5.2 Linguagens como tipos hierárquicos

Uma linguagem é caracterizada por três elementos: a sintaxe livre de contexto, condições de contexto que restringem o conjunto de programas significativos e sua especificação semântica. Estes itens podem ser descritos por tipos abstratos da sequinte maneira:

- A sintaxe livre de contexto corresponde à assinatura do tipo. Dada uma BNF pode-se construir uma assinatura introduzindo um *sort* S para cada símbolo não terminal $\langle s \rangle$ e uma operação p para cada regra de produção P. Se P tem a forma seguinte (com não terminais $\langle s_i \rangle$ e terminais t_j)

$$\langle s_0 \rangle ::= t_0 \langle s_1 \rangle t_1 \ldots \langle s_n \rangle t_n,$$

então p tem a funcionalidade

$$p : s_1 \times s_2 \times \ldots \times s_n \longrightarrow s_0$$

Note que os símbolos terminais t_j que foram "esquecidos", aqui, podem ser reconstruídos pela unicidade do nome p. Devido a esta independência dos símbolos terminais, a assinatura resultante chama-se *sintaxe abstrata*.

Obviamente os *sorts* devem ser restritos àqueles símbolos não terminais relevantes e ignorar símbolos auxiliares.

A álgebra de termos W_Σ da sintaxe abstrata Σ é isomórfica à álgebra das árvores *parser* [BRO 87].

- As condições de contexto para um construtor f da linguagem (i. é, para o símbolo de operação f da sintaxe abstrata) são expressas por termos lógicos $c \in W_\Sigma[x_1, \ldots, x_n]$. Seja a funcionalidade de f :

$$f : s_1 \times s_2 \times \ldots \times s_n \longrightarrow s_0,$$

então se tem o axioma para erros de contexto:

$c(t_1, \ldots t_n) = \text{false} \longrightarrow \text{undef}(f(t_1, \ldots t_n))$, ou

$f(t_1, \ldots t_n)) = x \longrightarrow c(t_1, \ldots t_n) = \text{true}.$

- A semântica da linguagem é especificada por um conjunto de equações condicionais (os axiomas do tipo).

O método algébrico indica claramente porque as condições de contexto são também chamadas de *semântica estática*: exatamente como a semântica "real", elas são especificadas por equações condicionais, mas desde que são restritas a premissas *boolena-valued*, a completeza suficiente dos tipos é decidível (tecnicamente, esta decidibilidade significa que podem ser avaliadas em tempo de compilação).

A especificação completa de uma linguagem pode ser vista, portanto, como uma sequência de restrições. Começando desde a álgebra de termo W_{Σ} derivada da sintaxe concreta. As condições de contexto definem uma redução para os termos "admissíveis" da linguagem, especificando uma subálgebra de "programas significativos". Finalmente uma relação de congruência induzida pelos axiomas de um tipo levam para álgebras quocientes dessa subálgebra, que podem ser consideradas como a "semântica" da linguagem.

Uma especificação semântica de uma linguagem deve ter em conta o princípio da *composicionalidade*, o qual requer que o significado de uma expressão seja explicado em função dos significados de suas componentes (subexpressões). Este princípio coincide naturalmente com a noção de homomorfismo presente na teoria algébrica.

5.3 Especificação Algébrica de RECON-II

5.3.1 Considerações Gerais

Na especificação da linguagem RECON-II são considerados três tipos abstratos como primitivos:

- O tipo BOOL com os valores *true* e *false*, com suas operações características.
- O tipo CHAR contendo o conjunto **ch** de caracters e as operações **eq-ch** para testar igualdade entre caracteres, e **letra?** que devolve *true* quando o caracter é uma letra e *false* em qualquer outro caso.

• O tipo IDENT contendo o conjunto (contável) **id** de identificadores e as operações **eq-id** para testar a igualdade entre identificadores, e **id-str** para converter um identificador num string.

Convém destacar que, de agora em diante, usar-se-á indistintamente álgebra e tipo (semântico) e assinatura e tipo (sintático), já que do método utilizado resulta que uma especificação é um tipo abstrato de dados.

5.3.2 Construção da especificação algébrica

5.3.2.1 Especificação dos Objetos Básicos

Analisando a sintaxe concreta de RECON-II identificam-se imediatamente quatro elementos principais da mesma: conceitos, relações, funções e redes. Mas para a especificação desses objetos estruturados é necessário primeiro especificar o que se chamará de *objetos atômicos* ou *básicos* (Os primitivos foram mencionados na seção anterior e são três: booleanos, caracteres e identificadores). Um objeto atômico é aquele que não possui estrutura. Logo, os objetos primitivos são também atômicos,mas para fins da especificação de RECON-II, são considerados conhecidos e portanto não serão especificados. Os objetos atômicos não primitivos são especificados a partir dos primitivos num processo de construção *bottom up*.

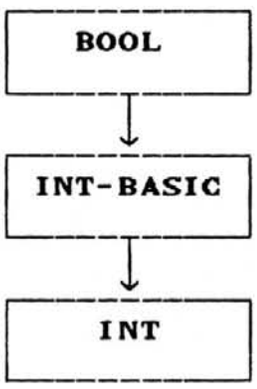


Fig. 5.1 : Hierarquia de Construção dos objetos básicos

A partir da álgebra BOOL constroi-se a álgebra dos inteiros, como é ilustrado na figura 5.1. As álgebras são representadas por quadrados e o processo de construção pelas setas. Esse processo define uma relação de dependência ou inclusão entre álgebras (tipos semânticos).

Algebricamente seria:

type INT-BASIC

based on BOOL

sort int

operations

zero: $\longrightarrow$ int

succ: int $\longrightarrow$ int

>0? : int $\longrightarrow$ bool

pred: int $\longrightarrow$ int

eq-int:int x int $\longrightarrow$ bool

norm: int $\longrightarrow$ int

is-0?: int $\longrightarrow$ bool

axioms

$\forall a, b, c \in$ int

1. >0?(a) = >'0?(norm(a))
2. >'0?(zero) = false
3. >'0?(succ(a)) = true
4. >'0?(pred(a)) = false
5. norm(pred(zero)) = pred(zero)
6. norm(succ(zero)) = succ(zero)
7. norm(zero) = zero
8. norm(succ(pred(a))) = norm(a)
9. norm(pred(succ(a))) = norm(a)
10. norm(succ(succ(a))) = norm'(succ(norm(succ(a))))
11. norm(pred(pred(a))) = norm'(pred(norm(pred(a))))
12. norm'(succ(succ(a))) = succ(succ(a))
13. norm'(pred(pred(a))) = pred(pred(a))
14. norm'(succ(pred(a))) = a
15. norm'(pred(succ(a))) = a
16. succ(pred(a)) = a
17. pred(succ(a)) = a
18. eq-int(a, b)=true $\Rightarrow$ eq-int(succ(a), succ(b))=true
19. eq-int(a, b)=true $\Rightarrow$ eq-int(pred(a), pred(b))=true

```
        20.eq-int(a, b)=false ⇒ eq-int(succ(a), succ(b))=false
        21.eq-int(a, b)=false ⇒ eq-int(pred(a), pred(b))=false
        22.eq-int(zero, a) = is-0?(a)
        23.eq-int(a, zero) = is-0?(a)
        24.is-0?(a) = is-0'?(norm(a))
        25.is-0'?(zero) = true
        26.is-0'?(succ(a)) = false
        27.is-0'?(pred(a)) = false
    endoftype
```

Escolheu-se demarcar sintaticamente o início e o fim da especificação com **type** e **endoftype**, seguindo a notação usada por Broy [BRO 87].

A cláusula **based on** representa o que graficamente descreveu-se como uma seta entre BOOL e INT-BASIC na figura 5.1. Essa notação quer significar que BOOL é uma subálgebra de INT-BASIC, i. é, que INT-BASIC herda todos os sorts e operações de BOOL. Essas duas álgebras formam o que se chama Tipo Hierárquico $T = (P, \Sigma_T, E_T)$, onde $P = (\Sigma_P, E_P)$ é o tipo primitivo.

As propriedades da equivalência (eq-int) são teoremas derivados dos axiomas anteriores :

```
        28.eq-int(a, a) = true
        29.eq-int(a, b) = true ⇒ eq_int(b, a) = true
        30.eq-int(a, b) = true .and. eq-int(b, c) = true →
            eq-int(a, c) = true
```

O tipo INT-BASIC pode ser enriquecido pelas operações usuais da aritmética dos inteiros:

```
    type INT
    based on INT-BASIC
    sort int
    operations
        _+_: int x int ⟶ int
        _-_: int x int ⟶ int
        _*_: int x int ⟶ int
        _/_: int x int ⟶ int
        abs : int ⟶ int
        sign: int x int ⟶ int
        ->- : int x int ⟶ bool
```

```
axioms
    ∀ a, b , c, d ∈ int
    1. a + zero = a
    2. a + succ(b) = succ(a + b)
    3. a + pred(b) = pred (a + b)
    4. a - zero = a
    5. a - succ(b) = pred(a - b)
    6. a - pred(b) = succ(a - b)
    7. a * zero = zero
    8. a * succ(b) = (a * b) + a
    9. a * pred(b) = (a * b) - a
    10.abs(a) = abs'(norm(a))
    11.abs'(zero) = zero
    12.abs'(succ(a))= succ(a)
    13.abs'(pred(a)) = succ(abs'(a))
    14.a > b = norm(a) >' norm(b)
    15.pred(a) >' pred(b) = a >'b
    16.zero >' pred(a) = true
    17.pred(a) >' zero = false
    18.succ(a) >' succ(b) = a >' b
    19.zero >' succ(a) = false
    20.succ(a) >' zero = true
    21.pred(a) >' succ(b) = false
    22.succ(a) >' pre(b) = true
    23. zero >' zero = false
    24.sign(a, b) = if ()-0?(a) .or. is-0?(a)) .and.
                          ()-0?(b) .or. is-0?(b))
                    then succ(zero)
                    else pred(zero)
    25.a/b = if abs(a) > abs(b) .or. eq-int(abs(a), abs(b))
          then ((abs(a) - abs(b))/abs(b) + succ(zero)) *
               sign(a, b))
               else zero
endoftype
```

O próximo tipo básico é o STRING, construído à maneira de INT-BASIC a partir do tipo primitivo CHAR.

```
type STRING
based on CHAR
sort str
operations
     "":          ⟶ str
     add-ch:str x char ⟶ str
     concat:      str x str ⟶ str
     eq-str:      str x str ⟶ bool
     prim-ch:     str ⟶ char
     null-str?: str ⟶ bool
     str-id: str ⟶ id
     resto-str: str ⟶ str
axioms
     ∀ a,b ∈ char    ∀ c, d, e ∈ str
     1. null-str?(" ") = true
     2. null-str?(add-ch(c, a) = false
     3. concat(c, " ") = c
     4. concat(c, add-ch(d, a)) = add-ch(concat(c, d), a)
     5. prim-ch(add-ch("",a) = a
     6. prim-ch( add-ch( add-ch(c,  a),  b)) =
                              prim-ch( add-ch(c, a)
     7. eq-str(c, " ") = null-str?(c)
     8. eq-str(" ", add-ch(c, a)) = false
     9. eq-str( add-ch(c, a), add-ch(d, b)) = eq-char(a, b)
                              .and. eq-str(c, d)
     10.str-id(c) = if letra?(prim-ch(c))
                    then c
                    else concat("a", c)
     11.resto-str(add-ch(c, a) = c
endoftype
```

Como no caso anterior, é possível deduzir as propriedades da relação de equivalência como teoremas:

```
     12.eq-str(c, d) = eq-str(d, c)
     13.eq-str(c, d) = true .and. eq-str(d, e) = true ⇒
        eq-str(c, e) = true
     14.eq-str(c, c) =true
```

5.3.2.2 Especificação dos Objetos Estruturados

Definidos todos os tipos básicos, resta agora definir a classe mais importante de RECON-II a álgebra VALOR. Este tipo, devido à sua complexidade, obriga a abandonar a construção hierárquica das especificações para introduzir uma nova notação que será explicada brevemente.

Antes de apresentar a especificação algébrica dos valores-RECON convém esclarecer o que se entende por um valor-RECON.

Um valor-RECON é qualquer valor (objeto) que possa ser referido por uma expressão-RECON válida. Foi visto no primeiro capítulo que RECON-II é uma linguagem que fala sobre conhecimento, i. é, conceitos ou objetos do mundo real ou imaginário. Agora bem, o que é exatamente uma expressão-RECON?. Uma expressão-RECON é qualquer construção admissível. Para determinar a admissibilidade de qualquer construção de alguma linguagem é necessário se remeter à sua sintaxe (geralmente uma BNF) e verificar se essa construção foi corretamente "construída". Essa noção de construção é a que o método algébrico tenta resgatar e valendo-se dela dá uma semântica para as construções da linguagem. Retomando o ponto em questão, segundo a BNF ou a sintaxe abstrata da RECON-II, dada no Anexo A e na seção 2.3.4, respectivamente,uma expressão-RECON é uma rede, um conceito, uma relação, uma função, ou o conteúdo de um atributo.

Logo, nota-se que a álgebra dos valores-RECON é o universo de discurso de RECON-II e que, portanto, um modelo para ela é um modelo para a linguagem RECON-II. Na figura 5.2 apresenta-se como é construída a especificação dos valores-RECON.

Como se pode observar na figura 5.2 o tipo VALOR é construído a partir dos tipos básicos. Por outro lado, ele possui subconjuntos de objetos estruturados que precisam do tipo VALOR para sua especificação e ao mesmo tempo formam parte dele. Estes subconjuntos são : listas, conjuntos, classes, pares ordenados, relações e funções.

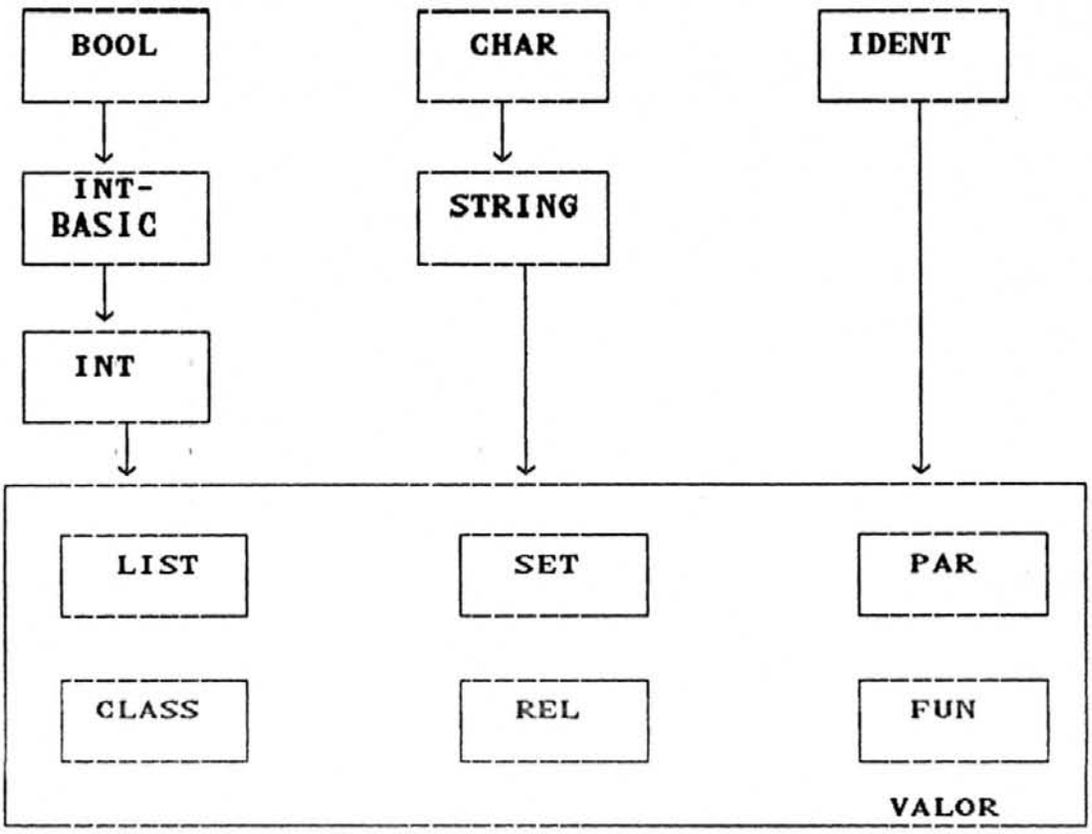


Fig. 5.2 : Composição do Tipo Valor

Isto é especificado da seguinte maneira:

```
CLUSTER
 type VALOR
 based on INT, STRING, BOOL
 sort val
 operations
        null: ⟶ val
        nulo?: val ⟶ bool
        num-val: int ⟶ val
        str-val: str ⟶ val
        bool-val:bool⟶ val
        numero?:val ⟶ bool
        string?:val ⟶ bool
        boolean?:val ⟶bool
        list-val:lst ⟶ val
        list?:val ⟶ bool
        set-val:set ⟶ val
        set?:val ⟶ bool
```

```
        par-val:par ⟶ val
        par?:val ⟶ bool
        class-val:class ⟶ val
        class?:val ⟶ bool
        rel-val:rel ⟶ val
        rel?:val ⟶ bool
        fun-val:fun ⟶ val
        fun?:val ⟶ bool
axioms
        1. ∀a ∈ int numero?(num-val(a)) = true
                numero?(null) = false
        2. ∀a ∈ str string?(str-val(a)) = true
                string?(null) = false
        3. ∀a ∈ bool boolean?(bool-val(a)) = true
                boolean?(null) = false
        4. ∀a ∈ lst list?(list-val(a)) = true
                list?(null) = false
        5. ∀a ∈ set set?(set-val(a)) = true
                set?(null) = false
        6. ∀a ∈ par par?(par-val(a)) = true
                par?(null) = false
        7. ∀a ∈ class class?(class-val(a)) = true
                class?(null) = false
        8. ∀a ∈ rel rel?(rel-val(a)) = true
                rel?(null) = false
        9. ∀a ∈ fun fun?(fun-val(a)) = true
                fun?(null) = false
        10.nulo?(null) = true
        11.∀a ∈ int nulo?(num-val(a)) = false
        12.∀a ∈ str nulo?(str-val(a)) = false
        13. ∀a ∈ bool nulo?(bool-val(a)) = false
        14. ∀a ∈ lst nulo?(list-val(a)) = false
        15. ∀a ∈ set nulo?(set-val(a)) = false
        16. ∀a ∈ par nulo?(par-val(a)) = false
        17. ∀a ∈ class nulo?(class-val(a)) = false
        18. ∀a ∈ rel nulo?(rel-val(a)) = false
        19. ∀a ∈ fun nulo?(fun-val(a)) = false
endoftype
```

Como pode ser observado na especificação da álgebra VALOR aparecem nomes de sorts que pertencem a álgebras (subálgebras de VALOR) que não foram definidas ainda. Isto se produz pela recursividade mútua entre VALOR e cada um dos seus subtipos. Exemplificando, uma lista é um valor mas ao mesmo tempo ela é pensada como uma sequência de valores.

Para conseguir especificar esta situação foi adotada uma notação diferente que permite especificar primeiro as álgebras para depois especificar suas subálgebras. Note-se que é exatamente o contrário do que vinha sendo feito com as álgebras básicas. A claúsula utilizada com este fim é: **expanded with**.

Uma semântica intuitiva desta claúsula será fornecida para ajudar à melhor comprensão da especificação.

Quando um objeto é representado por uma álgebra[1] é necessário fornecer duas coisas: uma família de "sorts" e um conjunto de operações. Da família de sorts tem-se que um sort é o chamado sort "principal"(definido pela claúsula **sort**) ou TOI (Type of Interest) na literatura de tipos abstratos de dados [GUT 78]. Esse sort principal é gerado, por convenção, a partir do conjunto vazio por aplicação das operações sobre os outros sorts.

Guttag [GUT 78] diferencia o conjunto de operações em dois subconjuntos disjuntos F = S + O, onde S é o conjunto das operações cujo codomínio é o TOI. Intuitivamente S contém as operações que podem ser usadas para gerar os valores do tipo sendo definido, enquanto que O são as operações que mapeiam valores do tipo em outros tipos. S nunca é vazío.

A claúsula **expanded with** visa a construção desse sort principal de uma maneira parcial ou por níveis. Dada a definição algébrica de um objeto fica determinado o sort principal (pela aplicação das operações de S ao conjunto gerador do TOI), mas pode ser necessário "expandir" esse sort com outras operações e/ou subconjuntos.

---

(1) O que é diferente de "ser interpretado como uma álgebra".

Observe que no método de especificação algébrica hierárquico esses subconjuntos deveriam ser especificados antes para logo depois definir o conjunto propriamente dito, mas no método não hierárquico o que se faz é definir primeiro o conjunto principal e depois acrescentar os subconjuntos que o compõem.

A seguir serão especificadas as subálgebras de VALOR. Note que na figura 5.2 não aparecem os relacionamentos entre as subálgebras de VALOR. A figura 5.3 e 5.4 , mostradas mais adiante, apresentam estes relacionamentos.

A álgebra das listas é especificada como:

type LIST
parameters Item **with:** equal:Item x Item ⟶ bool
VALOR **expanded with**
sort lst
operations
nil: ⟶ lst
_¦_: Item x lst ⟶ lst
cab: lst ⟶ Item
resto: lst ⟶ lst
∈-lst?: Item x lst ⟶ bool
eq-lst: lst x lst ⟶ lst
nil?: lst ⟶ bool
axioms
∀ *a*, *b* ∈ lst ∀ *c*, *d* ∈ Item
1. cab(*c*¦*a*) = *c*
2. resto(*c*¦*a*) = *a*
3. eq-lst(*a*, nil) = nil?(*a*)
4. eq-lst(nil, *c*¦*a*) = false
5. eq-lst(*c*¦*a*, nil) = false
6. eq-lst(*c*¦*a*, *d*¦*b*) = eq-lst(*a*, *b*) .and. equal(*c*, *d*)
7. ∈-lst?(*c*, *d*¦*a*) = if equal(*c*, *d*) then true
else ∈-lst?(*c*, *a*)
8. ∈-list?(*c*, nil) = false
9. nil?(nil) = true
10. nil?(*c*¦*a*) = false
endoftype

Nesta especificação foi incorporada uma nova construão sintática, **parameters**, para indicar que esta especificação é na realidade um *esquema de especificação*, onde dependendo da instanciação do parâmetro se obtém uma especificação algébrica particular.

Como se pode deduzir, um esquema de especificação não tem um modelo como as outras especificações.

O modelo de um esquema de especificação é justamente um *esquema de modelo*, onde para cada instanciação se tem uma álgebra inicial que é o modelo inicial da assinatura.

Continuando com a especificação das expressões-RECON, os próximos subtipos de VALOR são PAR-VALOR e SET, que, assim como LIST, são tipos parametrizados.

type PAR-VALOR
parameters Prim, Seg **with** eq1:Prim x Prim $\longrightarrow$ bool
eq2:Seg x Seg $\longrightarrow$ bool
VALOR **expanded with**
sort par
operations
par-elem: Prim x Seg $\longrightarrow$ par
prim-elem: par $\longrightarrow$ Prim
seg-elem: par $\longrightarrow$ Seg
eq-par:par x par $\longrightarrow$ bool
axioms
$\forall\ a, b \in$ par $\quad \forall\ c \in$ Prim $\quad \forall\ d \in$ Seg
1. par-elem(prim-elem(a), seg-elem(a)) = a
2. prim-elem(par-elem(c, d)) = c
3. seg-elem(par-elem(c, d)) = d
4. eq-par(par-elem(c_1, d_1), par-elem(c_2, d_2)) =
eq1(c_1, c_2) .and. eq2(d_1, d_2)
endoftype

type SET

parameters Elem **with** equal: Elem x Elem ⟶ bool

VALOR **expanded with**

sort set

operations

set(): ⟶ set

ins-elem: Elem x set ⟶ set

del-elem: Elem x set ⟶ set

-∪-: set x set ⟶ set

-∩-: set x set ⟶ set

-menos-: set x set ⟶ set

∅-set?: set ⟶ bool

⊆-set?: set x set ⟶ bool

∈-set?: set x Elem ⟶ bool

eq-set: set x set ⟶ bool

axioms

∀ a, b, c ∈ set ∀ e, e_1,e_2 ∈ Elem

1. ∅-set?(set()) = true
2. ∅-set?(ins-elem (e, a)) = false
3. ∈-set?(e_1, ins-elem(e_2, a)) = if equal(e_1, e_2)
   then true
   else ∈-set?(e_1, a)
4. ins-elem(e_1, ins-elem(e_2, a)) = ins-elem(e_2, ins-elem(e_1, a))
5. ∈-set?(e, set()) = false
6. a ∪ a = a
7. a ∪ b = b ∪ a
8. a ∪ set() = a
9. a ∪ ins-elem(e, b) = ins-elem(e, a ∪ b)
10. a ∩ a = a
11. a ∩ b = b ∩ a
12. a ∩ set() = set()
13. a ∩ ins-elem(e, b) = if ∈-set?(e, a)
    then ins-elem(e, a ∩ b)
    else a ∩ b
15. a menos a = set()
16. a menos set() = a

```
17.set() menos a = set()
18.ins-elem(e, b) menos a = if ∈-set?(e, a)
                            then del-elem(e, b) menos a
                            else ins-elem(e, b menos a)
19.set?(set()) = true
20.set?(ins-elem(e, a)) = set?(a)
21.⊆-set?(ins-elem(e, a), set()) = false
22.⊆-set?(ins-elem(e, a), b) = if ∈-set?(e, b)
                            then    ⊆-set?(del-elem(e,    a),
                                          del-elem(e, b))
                            else false
23.⊆-set?(set(), a) = true
24.del-elem(e, set()) = set()
25.del-elem(e1, ins(e2, a)) = if equal(e1, e2)
                            then del-elem(e1, a)
                          else (ins-elem(e2, del-elem(e1, a))
26.eq-set(a, b) = ⊆-set?(a, b) .and. ⊆-set?(b, a)
endoftype
```

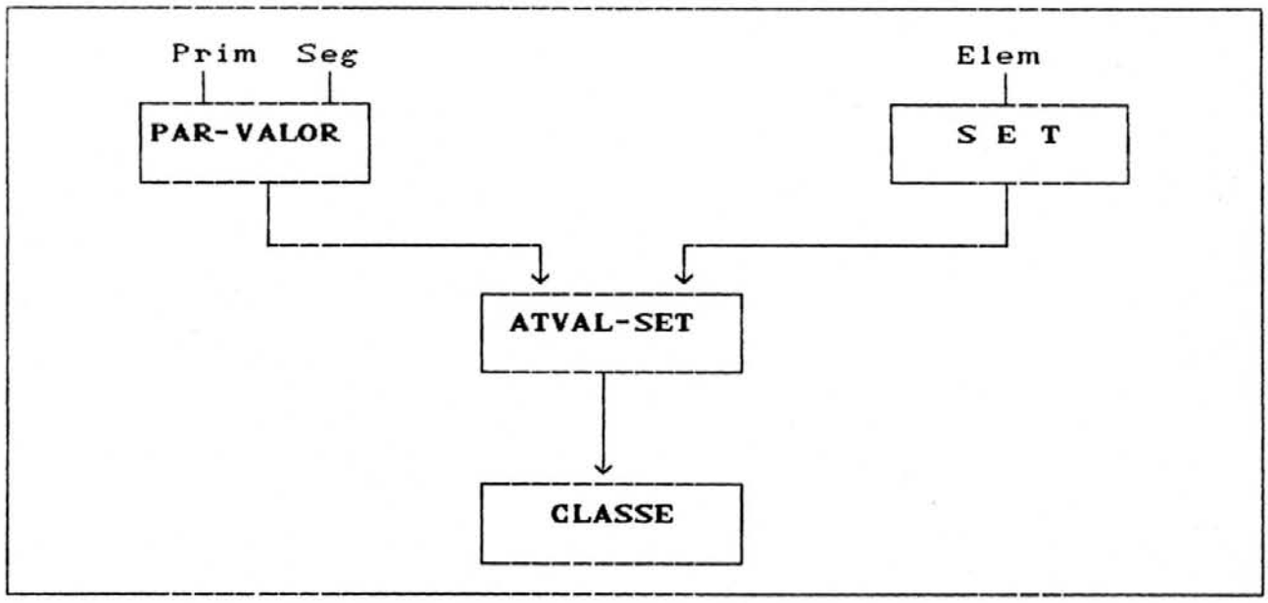


Fig.5.3 : Construção do Tipo Classe

Para especificar um conceito é preciso definir primeiro as àlgebras que são básicas a ele (ver figura 4.3.). Na bnf da RECON-II um conceito é composto de um nome seguido por atributos,

que podem ser pensados como pares de nomes e valores, i. é, um conceito é composto por um conjunto de pares de atributo-valor.

O tipo ATRIB-VALOR é uma instância particular do tipo PAR-VALOR. Isto é expresso pela claúsula **instance of with as**.

```
type ATRIB-VALOR
 instance of PAR-VALOR
      with
      Prim        as   id
      Seg         as   val
 rename par       as   av
 endoftype
```

Note que utiliza-se o predicado **rename as** com o único fim de nomear o conjunto de pares com um nome mais adequado para ser reconhecido nas especificações seguintes.

O tipo ATVAL-SET é uma instância particular do tipo SET com parâmetro ATRIB-VAL.

```
type AV-SET
 instance of SET
      with
      Elem        as   av
 rename set       as   av-set
 endoftype

 type ATVAL-SET
 AV-SET expanded with
 operations
      match: id x av-set ⟶ val
 axioms    ∀ a ∈ av-set   ∀ i, e ∈ id          ∀ υ ∈ val
      1. match(i, set()) = null
      2. match(i, ins-elem(par-elem(e, υ), a)) =
           if  eq-id(i, e) then υ else match(i, a)
 endoftype
```

Por último um conceito é especificado pela álgebra CLASSE da seguinte maneira:

type CLASSE

VALOR expanded with

sort class

operations

cria-cl: id x av-set ⟶ class

nome-cl: class ⟶ id

slots: class ⟶ av-set

valor-slot: class x id ⟶ val

add-slots:class x av ⟶ class

del-slots:class x av ⟶ class

union-cl: class x class ⟶ class

inter-cl: class x class ⟶ class

menos-cl: class x class ⟶ class

eq-cl: class x class ⟶ bool

≤-cl?: class x class ⟶ bool

axioms ∀ a ∈ class ∀ b, b_1, b_2...∈ av-set

∀ i, i_1,..., ∈ id ∀ c ∈ av

1. cria-cl(nome-cl(a), slots(a)) = a
2. nome-cl(cria-cl(i, b)) = i
3. slots(cria-cl(i, b)) = b
4. valor-slot(cria-cl(i, b), i_1) = match(i_1, b)
5. add-slots(cria-cl(i, b), c) = cria-cl(i, ins-elem(c, b))
6. del-slots(cria-cl(i, b), c) = cria-cl(i, del-elem(c, b))
7. union-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = cria-cl(str-id(concat(concat(id-str(i_1),"union"), id-str(i_2))), $b_1 \cup b_2$)
8. inter-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = cria-cl(str-id(concat(concat(id-str(i_1), "inter"), id-str(i_2))), $b_1 \cap b_2$)
9. menos-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = cria-cl(str-id(concat(concat(id-str(i_1), "menos"), id-str(i_2))), b_1 menos b_2)
10. eq-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) =

$$\text{eq-set}(b_1, b_2)$$

$$11.\leq\text{-cl}(\text{cria-cl}(i_1, b_1), \text{cria-cl}(i_2, b_2)) = \leq\text{-set}(b_1, b_2)$$

endoftype

Outro elemento importante de RECON-II são as relações. Elas são especificadas a partir dos conceitos (classes) como apresenta a figura 4.4

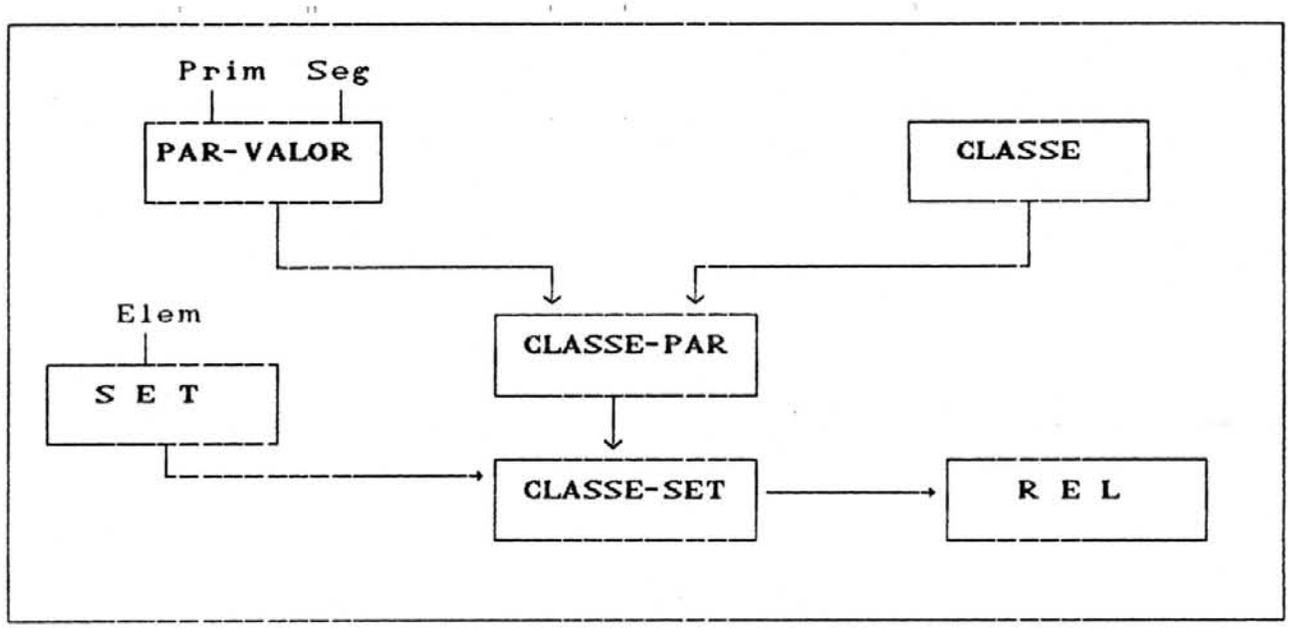


Fig. 5.4 : Construção do Tipo Rel

Algebricamente falando a figura anterior seria expressa como segue:

```
type CLASS-PAR
 instance of PAR-VALOR
      with
      Prim      as   class
      Seg       as   class
 rename par     as   par-cl
 endoftype
```

```
type CLASS-SET
 instance of SET
      with
      Elem        as   par-cl
 rename set       as   class-set
 endoftype
```

Uma característica interesante que possuem as relações é que podem ser definidas extensionalmente ou não.

Ás vezes é desejável definir uma relação através das propriedades que ela satisfaz, desta maneira não é necessário especificar os pares de conceitos que pertencem a ela de maneira completa, senão somente aqueles pares que não podem ser inferidos a partir das propriedades. RECON-II só possue um conjunto pré-definido de propriedades possíveis, as quais são especificadas algébricamente como um conjunto de identificadores:

```
type PRED
based on IDENT
sort pred
operations
      reflexiva: ⟶ pred
      simétrica: ⟶ pred
      transitiva: ⟶ pred
endoftype

type PRED-SET
 instance of SET
      with
      Elem        as   pred
 rename set       as   pred-set
 endoftype
```

Logo, uma relação é especificada da seguinte maneira:

type REL

VALOR expanded with

sort rel

operations

cria-rel: id x class-set x pred-set ⟶ rel

nome-rel: rel ⟶ id

pares-rel: rel ⟶ class-set

add-par: rel x par-cl ⟶ rel

del-par: relXpar-cl ⟶ rel

pred-rel: rel ⟶ pred-set

add-pred: rel x pred ⟶ rel

del-pred: rel x pred ⟶ rel

aplic-pred: rel ⟶ class-set

reflex: class-set ⟶ class-set

simet: class-set ⟶ class-set

transit: class-set ⟶ class-set

rel-cl: rel x class ⟶ class-set

union-rel: rel x rel ⟶ rel

inter-rel: rel x rel ⟶ rel

menos-rel: rel x rel ⟶ rel

eq-rel: rel x rel ⟶ bool

≤-rel?: rel x rel ⟶ bool

axioms

∀ a, b ∈ rel ∀ c ∈ class ∀ p ∈ pred ∀ i ∈ id

∀ d ∈ par-cl ∀ s ∈ class-set ∀ t ∈ pred-set

1. cria-rel(nome-rel(a), pares-rel(a), pred-rel(a)) = a
2. nome-rel(cria-rel(i, s, t)) = i
3. pares-rel(cria-rel(i, s, t)) = s
4. pred-rel(cria-rel(i, s, t)) = t
5. add-par(cria-rel(i, s, t), d) = cria-rel(i, ins-elem(d, s), t)
6. del-par(cria-rel(i, s, t), d) = cria-rel(i, del-elem(d, s), t)
7. add-pred(cria-rel(i, s, t), p) = cria-rel(i, s, ins-elem(p, t))
8. del-pred(cria-rel(i, s, t), p) = cria-rel(i, s, del-elem(p, t))

9.union-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2))
= cria-rel(concat(concat(i_1,"union"), i_2), $s_1 \cup s_2$,
$t_1 \cup t_2$)

10.inter-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2))
= cria-rel(concat(concat(i_1,"inter"), i_2), $s_1 \cap s_2$,
$t_1 \cap t_2$)

11.menos-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2))
= cria-rel(concat(concat(i_1,"menos"), i_2),
s_1 menos s_2, t_1 menos t_2)

12.eq-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2)) =
eq-set(aplic-pred(cria-rel(i_1, s_1, t_1),
aplic-pred(cria-rel(i_2, s_2, t_2))

13.aplic-pred(cria-rel(i, s, t)) =
case ∈-set?(reflexiva, t) .and.
∈-set?(simétrica, t) .and.
∈-set?(transitiva, t)
$s \cup$ (reflex(s) ∪ (simet(s) ∪ transit(s)))
case ∈-set?(reflexiva, t) .and.
∈-set?(simétrica, t)
$s \cup$ (reflex(s) ∪ simet(s))
case ∈-set?(reflexiva, t) .and.
∈-set?(transitiva, t)
$s \cup$ (reflex(s) ∪ transit(s))
case ∈-set?(simétrica, t) .and.
∈-set?(transitiva, t)
$s \cup$ (simet(s) ∪ transit(s))
case ∈-set?(reflexiva, t)
$s \cup$ reflex(s)
case ∈-set?(simétrica, t)
$s \cup$ simet(s)
case ∈-set?(transitiva, t)
$s \cup$ transit(s)
case ∅-set?(t)
s

14.reflex(set()) = set()

15.reflex(ins-elem(par-elem(c_1, c_2), s)) =
ins-elem(par-elem(c_1,c_1), ins-elem(par-elem(c_2,c_2),
reflex(s)

16.simet(set()) = set()

17.simet(ins-elem(par-elem(c_1, c_2), s) =
ins-elem(par-elem(c_2, c_1), simet(s)

18.transit(set()) = set()

19.transit(ins-elem(par-elem(c_1, c_2), ins-elem(
par-elem(c_3, c_4), s) = if eq-cl(c_1, c_4)
then ins-elem(par-elem(c_1, c_4),transit(ins-elem(
par-elem(c_1, c_2), ins-elem(par-elem(c_3, c_4),
ins-elem(par-elem(c_1, c_4), s)))))
else
if eq-cl(c_2, c_3)
then ins-elem(par-elem(c_3, c_2),transit(ins-elem(
par-elem(c_1, c_2), ins-elem(par-elem(c_3, c_4),
ins-elem(par-elem(c_3, c_2), s)))))
else transit(ins-elem(par-elem((c_1, c_2), s) ∪
transit(ins-elem(par-elem((c_3, c_4), s)

20.transit(ins-elem(par-elem((c_1, c_2), set()) =
if eq-cl(c_1, c_2) then ins-elem(par-elem(
(c_1, c_2),set())
else set()

21.≤-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2)) =
≤-set(aplic-pred(cria-rel(i_1, s_1, t_1),
aplic-pred(cria-rel(i_2, s_2, t_2))

endoftype

A próxima especificação é a do tipo Função, ela é definida através de um CLUSTER que agrupa a especificação de Expressão à de Função.

CLUSTER

type FUN

VALOR **expanded with**

sort fun

operations

lambda: id x id x expr ⟶ fun

aplic: fun x val ⟶ val

eq-fun: fun x fun ⟶ bool

compos: fun x fun ⟶ fun

axioms

$\forall f, g \in$ fun $\forall a, b \in$ val $\forall e \in$ expr

$\forall i, x, \in$ id

1. aplic(lambda(i, x, e), a) = eval(subst(cte(a),e, x))
2. aplic(f, null) = fun-val(f)
3. fun?(lambda(i, x, e)) = true
4. eq-fun(lambda(i_1, x_1, e_1), lambda(i_2, x_2, e_2)) =
   eq-expr(e_1, subst(x_1, e, x_2)
5. compos(lambda(i_1, x_1, e_1), lambda(i_2, x_2, e_2)) =
   lambda(str-id(concat(concat(id-str(i_1), '•'),
   id-str(i_2))), x_1, subst(e_1, e_2,x_2))

endoftype

Antes de apresentar a especificação da álgebra das expressões RECON-II convém destacar que as operações desta álgebra são justamente todos os operadores da álgebra dos valores, dado que uma expressão denota um valor. Essa função de denotação é indicada pela operação de avaliação *eval*. Outra operação importante desta álgebra é a operação de sustituição *subst*. Nesta especificação não serão apresentadas todas suas operações nem seus axiomas, já que como se verá eles não passam de transcrições de operações e axiomas anteriores.

type EXPRESSÃO
based on VALOR
sort expr
operations

eval: expr ⟶ val
var: id ⟶ expr
cte: val ⟶ expr
subst: expr x expr x id ⟶ expr
$\underline{+}$: expr x expr ⟶ expr
$\underline{-}$: expr x expr ⟶ expr
$\underline{*}$: expr x expr ⟶ expr
$\underline{/}$: expr x expr ⟶ expr
$\underline{\cup}$: expr x expr ⟶ expr
$\underline{\cap}$: expr x expr ⟶ expr
menos : expr x expr ⟶ expr

```
        concat : expr x expr ⟶ expr
         ! : expr x expr ⟶ expr
        ...
        union-cl : expr x expr ⟶ expr
        inter-cl : expr x expr ⟶ expr
        menos-cl : expr x expr ⟶ expr
        union-rel : expr x expr ⟶ expr
        inter-rel : expr x expr ⟶ expr
        menos-rel : expr x expr ⟶ expr
        compos : expr x expr ⟶ expr
        aplic : expr x expr ⟶ expr
        if-then-else : expr x expr x expr ⟶ expr
    axioms
        ∀ a, b ∈ expr   ∀ υ ∈ val        ∀ x, y ∈ id
        1. eval(var(x)) = erro
        2. eval(cte(υ)) = υ
        3. eval(a + b) = eval(a) + eval(b)
        4. eval(a ∪ b) = eval(a) ∪ eval(b)
        5. eval(union-cl (a, b)) = union-cl(eval(a), eval(b))
        6. eval(union-rel(a, b)) = union-rel(eval(a), eval(b))
        7. eval(compos(a, b)) = compos(eval(a), eval(b))
        8. eval(if-then-else(a, b1, b2)) = if eval(a)
                                              then eval(b1)
                                              else eval(b2)
        9. subst(var(y), a, x) = if eq-ident(x, y)
                                    then a
                                    else var(y)
        10. subst(cte(υ), a, x) = cte(υ)
        11.subst(a1 + a2, a, x) = subst(a1, a, x) + subst(a2, a, x)
        12.subst(union-cl (a1, a2), a, x) = union-cl(
                                 subst(a1, a, x), subst(a2, a, x))
        . . .
    endoftype
ENDCLUSTER FUN
```

Por último, a álgebra das Redes é especificada como :

```
type SET-CL
instance of SET
     with
     Elem       as   class
rename set      as   set-cl
endoftype

type SET-REL
instance of SET
     with
     Elem       as   rel
rename set      as   set-rel
endoftype

type SET-FUN
instance of SET
     with
     Elem       as   fun
rename set      as   set-fun
endoftype

type REDE
based on VALOR, EXPR
sort rede
operations
     cria-rede: id x set-cl x set-rel x set-fun ⟶ rede
     nome-r: rede ⟶ id
     class-r: rede ⟶ set-cl
     rel-r: rede ⟶ set-rel
     fun-r: rede ⟶ set-fun
     eq-rede: rede x rede ⟶ bool
     ≤-rede?: rede x rede ⟶ bool
     union-r : rede x rede ⟶ rede
     inter-r : rede x rede ⟶ rede
     menos-r : rede x rede ⟶ rede
     add-cl: rede x class ⟶ rede
     del-cl: rede x class ⟶ rede
```

add-rel: rede x rel ⟶ rede

del-rel: rede x rel ⟶ rede

add-fun: rede x fun ⟶ rede

del-fun: rede x fun ⟶ rede

axioms $\forall a \in$ set-cl $\forall b, b_1, b_2 \ldots \in$ set-rel

$\forall i, i_1, \ldots, \in$ id $\forall c, c_1, c_2 \in$ set-fun

$\forall r, r_1, \ldots \in$ rede $\forall d \in$ class $\forall e \in$ rel

$\forall f \in$ fun

1. cria-rede(nome-r(r), class-r(r),
   rel-r(r), fun-r(r)) = r
2. nome-r(cria-rede(i, a, b, c)) = i
3. class-r(cria-rede(i, a, b, c)) = b
4. add-cl(cria-rede(i, a, b, c), d) =
   cria-rede(i, ins-elem(d, a), b, c)
5. del-cl(cria-rede(i, a, b, c), d) =
   cria-rede(i, del-elem(d,a), b, c)
6. add-rel(cria-rede(i, a, b, c), e) =
   cria-rede(i, a),ins-elem(b, e), c)
7. del-rel(cria-rede(i, a, b, c), e) =
   cria-rede(i, a, del-elem(b, e), c)
8. add-fun(cria-rede(i, a, b, c), f) =
   cria-rede(i, a, b, ins-elem(c, f))
9. del-fun(cria-rede(i, a, b, c), f) =
   cria-rede(i, a, b, del-elem(c, f))
10. union-r(cria-rede(i_1, a_1, b_1, c_1),
    cria-rede(i_2, a_2, b_2, c_2)) =
    cria-rede(str-id(concat(concat(id-str(i_1), "union"),
    id-str(i_2))), $a_1 \cup a_2$,
    $b_1 \cup b_2$, $c_1 \cup c_2$)
11. inter-r(cria-rede(i_1, a_1, b_1, c_1),
    cria-rede(i_2, a_2, b_2, c_2)) =
    cria-rede(str-id(concat(concat(id-str(i_1), "inter"),
    id-str(i_2))), $a_1 \cap a_2$, $b_1 \cap b_2$, $c_1 \cap c_2$)
12. menos-r(cria-rede(i_1, a_1, b_1, c_1),
    cria-rede(i_2, a_2, b_2, c_2)) =
    cria-rede(str-id(concat(concat(id-str(i_1),
    "menos"), id-str(i_2))), a_1 menos a_2, b_1 menos b_2,
    c_1 menos c_2)

13.eq-rede(cria-rede(i_1, a_1, b_1, c_1),
cria-rede(i_2, a_2, b_2, c_2)) = eq-set(a_1, a_2)
.and. eq-set(b_1, b_2) .and. eq-set(c_1, c_2)

14.≤-cl(cria-rede(i_1, a_1, b_1, c_1),
cria-rede(i_2, a_2, b_2, c_2)) = ≤-set(a_1, a_2)
.and. ≤-set(b_1, b_2) .and. ≤-set(c_1, c_2)

endoftype

ENDCLUSTER VALOR

6 SISTEMA DE TIPOS PARA RECON-II

6.1 Introdução

No presente capítulo se definirá um Sistema de Tipos para RECON-II.

Segundo a definição dada no capítulo 3, um Sistema de Tipos é composto essencialmente por uma linguagem de tipos e um sistema de inferência.

Quando se fala em definir uma linguagem é necessário determinar por um lado sua semântica e por outro sua sintaxe.

Com relação à semântica da linguagem de tipos, no capítulo 5 apresentou-se a semântica de RECON-II e ao mesmo tempo classificou-se o Universo de Discurso em classes de objetos, i. é, determinou-se subconjuntos, não disjuntos, do Universo de Discurso que constituem os chamados Tipo Semânticos.

Resta, portanto, definir uma sintaxe adequada para a linguagem de tipos de RECON-II, que chamar-se-á, de agora em diante, LTR-II.

Por último, uma vez estabelecida a sintaxe da linguagem de tipos, fica pendente especificar o relacionamento entre LTR-II e RECON-II, i. é, definir como atribuem-se as expressões de LTR-II às expressões de RECON-II. Isto é obtido através da definição da Atribuição de Tipos (ver seção 3.5), que geralmente fica determinada com a definição de um conjunto de regras de inferência.

Neste capítulo serão apresentadas, por ser mais intuitivo e melhorar a compreensão, primeiro as regras de inferência e depois a sintaxe de LTR-II.

Como existem regras de inferência para cada tipo básico e para cada construtor de tipo, o capítulo foi dividido em duas seções 6.2, e 6.3 para tratar dos tipos básicos e dos estruturados, respectivamente. Na seção 6.4 aparece a sintaxe completa de LTR-II.

6.2 Tipos Básicos

Antes de começar a descrição dos tipos básicos e suas regras de inferência é necessário esclarecer que assim como numa linguagem de programação existem objetos básicos ou atômicos e objetos estruturados, numa linguagem de tipo existem os tipos básicos e os tipos estruturados. Mais ainda, a relação entre objetos e tipos é mais íntima: a cada objeto básico corresponde um tipo básico e a cada objeto estruturado, um construtor de tipo estruturado. Esta relação é definida através do Sistema de Tipos e, mais precisamente, através do Sistema de Inferência de Tipos.

Os tipos básicos em LTR-II são : Bool, Char, String, Int, Ident e Null.

6.2.1. NULL

O tipo Null é o mais primitivo de todos. Ele representa intuitivamente o conjunto vazio e, portanto, possui uma única regra de inferência, aquela que indica que Null é um tipo, chamada de *regra de formação*. As regras de formação especificam os construtores de tipos e seu modo de aplicação [COS 90]. Nos tipos básicos, obviamente, a regra de formação consiste unicamente em indicar que o tipo básico *b* é um tipo, como pode ser observado na regra de formação de Null:

$$\frac{}{\texttt{Null} :: \textbf{TYPE}}$$

6.2.2. BOOL

O tipo Bool possui dois elementos, i. é, existem dois objetos que tem o tipo Bool. Portanto se definem duas regras chamadas de *introdução* para construir esses objetos: as regras de introdução especificam os construtores de objetos e seu modo de aplicação [COS 90].

Sempre que se define uma regra de introdução é necessário definir também sua contrapartida: a *regra de eliminação*. As

regras de eliminação especificam modos de aplicar objetos ativos e de desmontar objetos passivos.

Por último, existe uma quarta espécie de regra de inferência a *regra de computação*, que especifica modos de simplificar objetos compostos. Esta classificação das regras de inferência é definida por Martin-Löf e apresentada em [COS 90]. Cabe mencionar que existe uma forte vinculação entre estas quatro espécies de regras, o que é evidenciado pelo fato de as regras de formação e introdução determinarem completamente as outras [BAC 89].

A regra de formação do tipo Bool é:

$$\frac{}{\text{Bool} :: \textbf{TYPE}}$$

As regras de introdução são :

$$\frac{}{\pi \vdash \text{true} : \text{Bool}} \qquad \frac{}{\pi \vdash \text{false} : \text{Bool}}$$

Nestas regras aparece um novo elemento, o ambiente π. A interpretação da regra é a seguinte: o objeto *true* deriva-se do ambiente π e tem o tipo Bool. O ambiente π é um conjunto de suposições.

A regra de eliminação é:

$$\frac{\pi \vdash a : \text{Bool} \quad \pi \vdash c : T \quad \pi \vdash d : T}{\pi \vdash \text{if } a \text{ then } c \text{ else } d : T}$$

onde T denota um tipo válido.

A interpretação desta regra é a seguinte: a expressão if a then c else d de RECON-II tem o tipo T se se verificam as condições de a ser do tipo Bool e, c e d do tipo T.

Finalmente as regras de computação :

$$\text{if } \textit{true} \text{ then } c \text{ else } d \longrightarrow c$$
$$\text{if } \textit{false} \text{ then } c \text{ else } d \longrightarrow d$$

As operações sobre os objetos de tipo Bool, como .and., .or., e .not. podem também ser definidas através de regras de inferência. Por exemplo,

$$\frac{\pi \vdash a : \text{Bool}}{\pi \vdash .\text{not}.a : \text{Bool}}$$

e .not. *true* ⟶ *false*

.not. *false* ⟶ *true*

Dado que estas regras não são essenciais para a definição do Sistema de Tipos, não serão apresentadas aqui.

6.2.3. CHAR

O tipo Char, possui regra de formação e de introdução, mas não de eliminação nem de computação. A razão para isto é simples, ele comporta-se como um conjunto de objetos atômicos finito, comnenhuma operação definida sobre ele.

Por motivo de simplicidade na leitura e compreensão das regras de inferência, de agora em diante, o ambiente π será omitido na definição das mesmas.

Regra de Formação :

$$\frac{}{\text{Char} :: \textbf{TYPE}}$$

Regra de Introdução : as regras de introdução são 128, uma para cada caracter ASCII, dada a simplicidade destas regras será definido um *esquema de regra:*

$$\frac{}{c : \text{Char}} \; ; \; c \text{ é caracter}$$

6.2.4. STRING

Regra de Formação :

$$\frac{}{\texttt{Str} :: \textbf{TYPE}}$$

Regras de Introdução :

$$\frac{}{\text{" "} : \texttt{Str}}$$

$$\frac{s : \texttt{Str} \quad a : \texttt{Char}}{\texttt{add-str}(s, a) : \texttt{Str}}$$

Regras de Eliminação :

$$\frac{s : \texttt{Str}}{\texttt{prim-ch}(s) : \texttt{Char}}$$

$$\frac{s : \texttt{Str}}{\texttt{resto-str}(s) : \texttt{Str}}$$

Regras de Computação :

$\texttt{prim-ch}(\text{""}) \longrightarrow \perp$

$\texttt{prim-ch}(\texttt{add-str}(s, a)) \longrightarrow a$

$\texttt{resto-str}(\text{" "}) \longrightarrow \text{" "}$

$\texttt{resto-str}(\texttt{add}(s, a)) \longrightarrow s$

$\texttt{add-str}(\texttt{resto-str}(s), \texttt{prim-ch}(s)) \longrightarrow s$

Como no caso anterior a operação concatenação (^) pode ser expressa através de uma regra de inferência.

6.2.5 IDENT

O tipo Id é um subtipo do tipo Str com a restrição que o primeiro caracter seja uma letra. É um tipo atômico com uma regra de formação e uma de introdução e sem regras de eliminação.

Regra de Formação :

$$\frac{}{\text{Id :: TYPE}}$$

Regra de Introdução :

$$\frac{a : \text{Str}}{a : \text{Id}} \text{ ; prim-ch}(a) \text{ é letra}$$

6.2.6. INT

Regra de Formação :

$$\frac{}{\text{Int :: TYPE}}$$

Regras de Introdução :

$$\frac{}{0 : \text{Int}}$$

$$\frac{a : \text{Int}}{\text{succ}(a) : \text{Int}}$$

Regra de Eliminação :

$$\frac{a : \text{Int}}{\text{pred}(a) : \text{Int}}$$

Regras de Computação

$0 \longrightarrow 0$

$pred(0) \longrightarrow \perp$

$succ(0) \longrightarrow succ(0)$

$succ(pred(a)) \longrightarrow a \quad se\ a \neq 0$

$pred(succ(a)) \longrightarrow a$

Neste caso também o tipo Int tem um conjunto de operações (soma, diferênça, multiplicação, divisão) que podem ser expressas através de regras de inferência, mas que não serão definidas neste capítulo.

Nos tipos que contém regras de introdução recursivas (como os tipos STR e INt) pode-se usar as regras de eliminação para caracterizar o Princípio de Indução que se aplica ao tipo. No caso do tipo Int, as regras seriam relativas à computação de funções definidas por recursão primitiva:

$$\frac{n : \mathrm{Int} \quad f : \mathrm{Int} \longrightarrow \mathrm{Int} \quad c : \mathrm{Int}}{recprim(n, f, c) : \mathrm{Int}}$$

Com regras de Computação :

$recprim(0, f, c) \longrightarrow c$

$recprim(succ(n), f, c) \longrightarrow f(recprim(n, f, c))$.

6.3. Tipos Estruturados

Antes de começar a definição das regras de inferência para os tipos estruturados é necessário dar certas regras gerais de inferência. Tais regras são as que especificam a relação de generalidade entre tipos,também chamada de subtipagem, e que será denotada pelo símbolo $\leq$.

A relação de subtipagem é uma relação de ordem parcial, i. é, reflexiva, antisimétrica e transitiva, e modela muitas características do que é conhecido comumente como herança (de atributos, de métodos, etc.) em linguagens de representação de conhecimento e orientadas a objetos [CAR 89]. Não obstante, cabe

destacar que herança de atributos tem um significado difuso, não possui uma definição inequivocamente estabelecida, enquanto que subtipagem tem um significado técnico bem definido.

Basicamente, a relação de subtipagem consiste no seguinte: se x tem o tipo A, i. é, $x : A$, e A é um subtipo de B, i. é., $A \leq B$, então, x tem o tipo B. Notacionalmente $x : A \wedge A \leq B \rightarrow x : B$. Esta propriedade básica consitui o chamado Teorema da Subtipagem Sintática [CAR 84].

Logo, intuitivamente comprende-se que subtipagem é uma inclusão de conjunto.

6.3.1 A relação de generalidade entre tipos.

Antes de especificar as regras de inferência que definem a relação de generalidade entre tipos, convém destacar: V_n são variáveis de tipo da linguagem de tipos, T_n são expressões de tipos arbitrários (possivelmente com variáveis livres) e π define um conjunto de suposições sobre atribuições de tipos e/ou generalidade entre tipos.

As regras são :

Suposição: $\pi \cup \{T_1 \leq T_2\} \vdash T_1 \leq T_2$;

Reflexividade: $\pi \vdash T \leq T$;

Reflexividade significa que de qualquer conjunto de suposições e o tipo T pode se deduzir que $T \leq T$, i. é, todo tipo tem um menor tipo mais geral que é ele próprio.

Transitividade: $$\frac{\pi_1 \vdash T_1 \leq T_2 \quad \pi'_2 \vdash T_2 \leq T_3}{\pi_1 \cup \pi_2 \vdash T_1 \leq T_3}$$

Esta regra pode ser lida como: "se se pode provar que de π_1 deriva-se $T_1 \leq T_2$ e que de π_2 deriva-se $T_2 \leq T_3$, então pode-se provar que de $\pi_1 \cup \pi_2$ deriva-se $T_1 \leq T_3$.

Instanciação: $\pi \vdash T_1[T_2/V] \leq \forall V.T_1$

Expressões da forma $T_1[T_2/V]$ significam "T_1 com ocorrências livres de V substituídas por T_2. Um objeto que opera com todos os

tipos de uma classe de tipos é mais geral que uma instância dele que só opera com uma subclasse daqueles tipos.

Generalização: $\dfrac{\pi \vdash T_1 \leq T_2}{\pi \vdash \forall V.T_1 \leq T_2}$; V não livre em π ou T_2

Se um tipo T_2 é mais geral que um tipo parametrizado sobre V, então T_2 é também mais geral que a versão generalizada do tipo.

Regra de Subtipagem $\dfrac{\pi \vdash T_1 \leq T_2 \quad \pi \vdash a : T_1}{\pi \vdash a : T_2}$

6.3.2 PRODUTO CARTESIANO (x)

Regra de Formação :

$$\frac{S :: \mathbf{TYPE} \qquad T :: \mathbf{TYPE}}{S \times T :: \mathbf{TYPE}}$$

Regra de Introdução :

$$\frac{a : S \qquad b : T}{(a, b) : S \times T}$$

Regras de Eliminação :

$$\frac{p : S \times T}{\mathbf{prim\text{-}elem}(p) : S}$$

$$\frac{p : S \times T}{\mathbf{seg\text{-}elem}(p) : T}$$

Regra de Computação :

$\mathbf{prim\text{-}elem}((a, b)) \longrightarrow a$

$\mathbf{seg\text{-}elem}((a, b)) \longrightarrow b$

$(\mathbf{prim\text{-}elem}(p), \mathbf{seg\text{-}elem}(p)) \longrightarrow p$

Regra de Subtipagem:

$$\frac{T_1 \leq T_2 \quad S_1 \leq S_2}{T_1 \times S_1 \leq T_2 \times S_2}$$

6.3.3 LIST

Regra de Formação :

$$\frac{T :: \mathbf{TYPE}}{List[T] :: \mathbf{TYPE}}$$

Regras de Introdução :

$$\frac{}{nil : List\ [T]}$$

$$\frac{a : List[T] \quad b : T}{a \mid b : List[T]}$$

Regras de Eliminação:

$$\frac{a : List[T]}{head(a) : T}$$

$$\frac{a : List[T]}{tail(a) : List[T]}$$

Regras de Computação :

$$head(a \mid b) \longrightarrow a$$

$$tail(a \mid b) \longrightarrow b$$

$$head(a) \mid tail(a) \longrightarrow a$$

Regra de Subtipagem

$$\frac{T_1 \leq T_2}{List[T_1] \leq List[T_2]}$$

6.3.4 SET

Regra de Formação :

$$\frac{T :: \mathbf{TYPE}}{Set[T] :: \mathbf{TYPE}}$$

Regras de Introdução:

$$\frac{}{\{\ \} : Set[T]}$$

$$\frac{\{a_1, a_2, \ldots, a_n\} : Set[T] \quad b : T}{\{a_1, a_2, \ldots, a_n, b\} : Set[T]}$$

Regras de Eliminação :

$$\frac{s : Set[T] \quad a_i : T}{del\text{-}elem(s, a_i) : Set[T]}$$

$$\frac{s : Set[T]}{choose\text{-}elem(s) : T}$$

Regras de Computação :

$\{a_1, a_2, \ldots, a_n\} \longrightarrow \{a_2, \ldots, a_n, a_1\}$

$del\text{-}elem(\{a_1, a_2, \ldots, a_n\}, a_1) \longrightarrow \{a_2, \ldots, a_n\}$

$choose\text{-}elem\ (\{a_1, a_2, \ldots, a_n\}) \longrightarrow a_i$; para algum i=1. . . n.

Regra de Subtipagem:

$$\frac{T_1 \leq T_2}{Set[T_1] \leq Set[T_2]}$$

6.3.5 FUN

Regra de Formação :

$$\frac{T_1 :: \mathbf{TYPE} \quad T_2 :: \mathbf{TYPE}}{T_1 \longrightarrow T_2 :: \mathbf{TYPE}}$$

Regra de Introdução :

$$\frac{x : T_1 \quad e : T_2}{\mathrm{fun}(x : T_1).\ e : T_1 \longrightarrow T_2}$$

Note que o argumento x da função pode ser um objeto de qualquer tipo, em particular pode ser uma lista. Portanto é facil generalizar esta regra para outra que trate funções de n argumentos.

Regra de Eliminação :

$$\frac{f : T_1 \longrightarrow T_2 \quad e : T_1}{f\ (e) : T_2}$$

Regra de Computação :

$$(\mathrm{fun}(x : T_1).e_1)(e_2) \longrightarrow e_1[e_2/x]$$

Em matemática uma função de T_1 para T_2 pode também ser considerada como uma função de S_1 para S_2 se $S_1 \subseteq T_1$ e $T_2 \subseteq S_2$. Isto é chamado *contravariância* (no primeiro argumento) do operador espaço de função.

Em programação esta noção é usada, especialmente em linguagens com sistema de tipo coerente, pois assegura que uma função que trabalha com objetos de um tipo dado, trabalhará também com objetos de qualquer subtipo do tipo, e que a função que retorna um objeto do tipo dado pode ser vista como retornando um objeto de algúm supertipo do tipo. A regra de inferência para este caso é:

Regra de Subtipagem:

$$\frac{S_1 \leq T_1 \qquad T_2 \leq S_2}{T_1 \longrightarrow T_2 \leq S_1 \longrightarrow S_2}$$

Nas regras anteriores omitiu-se, por questões de simplicidade, o identificador associado a uma função. Este identificador é opcional pois na construção de conceitos ele é desnecessário. Em RECON-II existem duas maneiras de definir funções: dentro dos conceitos, como atributos dos mesmos ou isoladamente. Neste último caso é necessário dar um nome à função para poder ser utilizada em outro contexto. A sintaxe é:

Lambda f $(x : T_1)$ ::= e

A alteração nas regras de introdução e eliminação é mínima e consiste especificamente em exigir a premisa f : Id na regra de introdução e enunciar mais uma regra de eliminação :

$$\frac{g : T_1 \longrightarrow T_2}{\text{nome-fun}(g) : \text{Id}}$$

6.3.6. SIGMA (Σ)

O tipo Sigma é o tipo dos conceitos RECON-II. No capítulo anterior, a especificação algébrica dos conceitos foi denominada Class, mas aqui decidiu-se mudar essa denominação porque não expressava fielmente a noção dos conceitos e podia gerar confusão a respeito do significado do tipo de um conceito.

Um conceito é um objeto-Recon. Ele pode ser um conceito genérico (vulgarmente chamado de classe nas áreas de IA e de Orientação a Objetos) ou conceito simples (conhecido como individuaos ou instância de uma classe).

Neste trabalho tenta-se dar um tipo ao construtor de conceitos, independente dele ser genérico ou não. Geralmente nos sistemas de tipos escritos para linguagens orientadas a objetos se diz que o tipo de uma instância é sua classe, mas o que

pretende-se aqui é dar um tipo para a própria classe e implementar as noções de generalização e classificação através da tipagem, i. é, de modo a tais noções poderem ser realizadas via o sistema de tipos.

Ao fornecer uma semântica para as expressões-Recon no capítulo 4 definiu-se uma assinatura (CLASSE) para os conceitos. Tal assinatura representa a álgebra dos conceitos. Mas, como já foi mencionado, existe mais de uma álgebra que satisfaz uma determinada assinatura. No caso dos conceitos é interessante descrever uma interpretação para a assinatura CLASSE que permite definir uma noção de tipo para estes objetos de uma maneira natural e simples. Tal interpretação consiste em pensar os conceitos como álgebras onde seu *carrier* principal é o conjunto dos objetos que ele define e suas operações são os atributos do conceito. Para esclarecer considere o seguinte exemplo:

```
conceito Livro ::= ( autor          : List[Str]
                     título         : Str
                     estilo = [conto, poesia, crónica ...]
                   )
```

Neste exemplo o *carrier* principal é o conjunto de todos os livros (LIVRO). O atributo autor pode ser pensado como uma operação que para um determinado livro fornece todos seus autores, i. é, autor : LIVRO ⟶ List[Str]. O mesmo raciocínio é aplicado para todos os atributos do conceito Livro.

As relações de herança e de instanciação podem ser inferidas pelos tipos ou somente ser verificadas segundo a tipagem seja implícita ou explícita, respectivamente. No caso de RECON será exigido para os conceitos explicitar ambas relações. Isto visa somente simplificar o trabalho do usuário, já que se se optasse pela tipagem implícita o usuário deveria fornecer todos os atributos num subconceito (inclusive os herdados) para que o sistema de tipos seja capaz de inferir a relação de herança (ver herança no capítulo 3 seção 3.6.2). O mesmo acontece com a instanciação de conceitos.

Na figura 6.1 é possível ver um exemplo que inclui herança e instanciação.

SINTAXE	INTERPRETAÇÃO
conceito Livro::=(	Livro = Σ L.(
autor : List[Str]	autor:L⟶List[Str]
título : Str	título:L ⟶ Str
estilo = [conto, poesia, crónica...]	estilo:L⟶[conto...]
)	)
conceito Livro-Romance::=(	Livro-Romance = Σ LR.(
subconceito: [Livro]	subconceito:LR⟶[Livro]
estilo = [romance]	estilo(t)=LR⟶[romance]
tipo=[policial,fição ...]	tipo:LR⟶[policial, fição, ...])
)	
conceito Livro-A ::=(	LA ∈ Livro-Romance com
instância : [Livro-Romance]	autor(LA)=[U. Ecco]
autor: [Umberto Ecco]	título(LA)=O nome..
tipo=[policial,histórico]	tipo(LA)=[policial...]
título:O nome da Rosa	
)	

Fig. 6.1 : Exemplo de Herança em RECON-II

Cabe mencionar que na linguagem RECON usada atualmente as noções de subconceito e de instância não são diferenciadas e ambas são definidas atraves da relação pré-definida *especialização*.

Outro ponto interessante a destacar é que a interpretação de uma instância de um conceito é como termo da álgebra que denota o conceito. Portanto a relação de instaciação é uma relação de pertinência. E a relação de subconceito é a relação de subálgebra

Na figura 6.2 mostra-se a sequência de abstrações feitas para obter a interpretação algébrica dos conceitos. Nela observa-se que uma álgebra tem um tipo associado que é sua assinatura. Portanto um conceito-Recon possui um tipo que é a assinatura da álgebra e que na literatura é conhecido como *tipo*

quantificado existencialmente ou *tipo existencial* [BRO 79] [MIT 88] [DAN 88] [COS 90] [GIR 89].

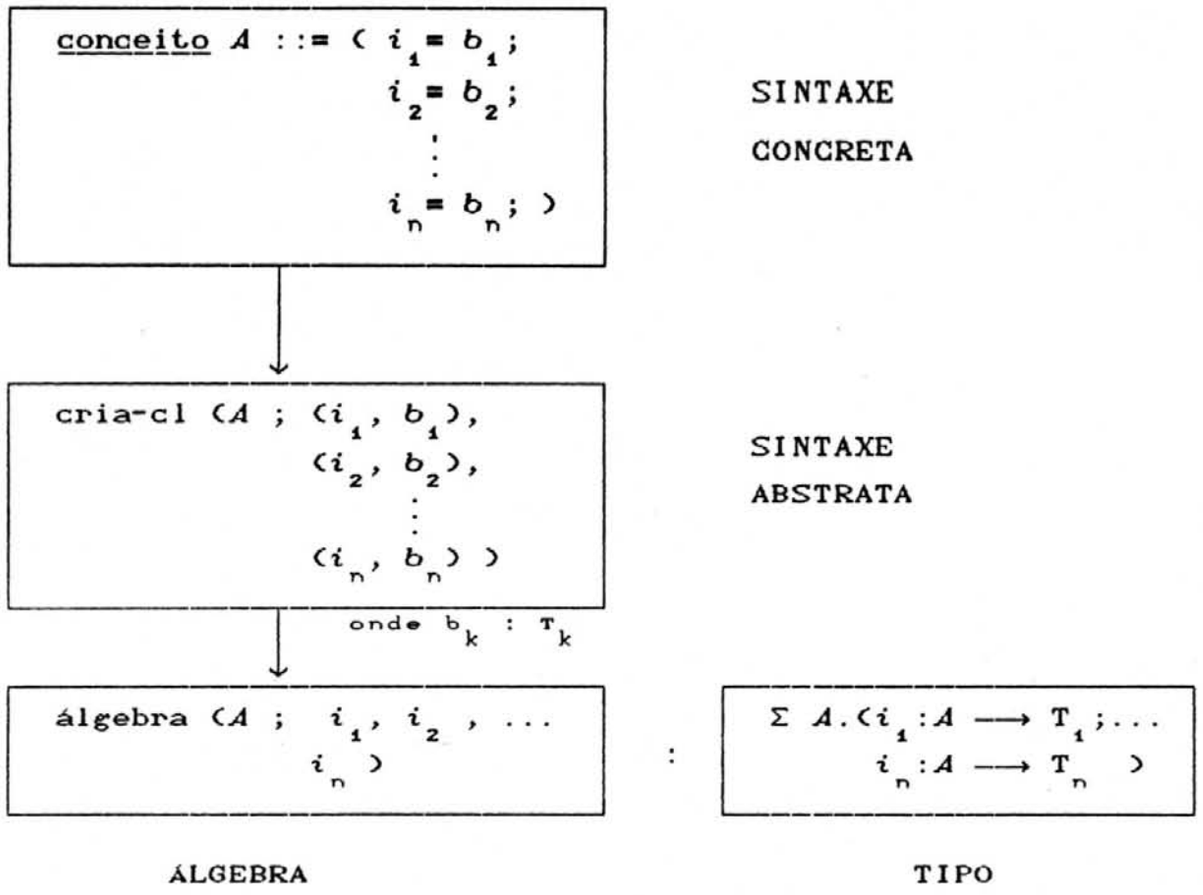


Fig. 6.2 : Interpretação algébrica dos conceitos

As regras para o tipo Σ são as seguintes:

Regra de Formação :

$$\frac{T_1, \ldots, T_n :: \mathbf{TYPE} \quad T :: \mathbf{TYPE} \Rightarrow i_1(T) : T_1 \ \ldots \ T :: \mathbf{TYPE} \Rightarrow i_n(T) : T_n}{(\Sigma x \in \mathrm{Id}.\ i_1 : x \longrightarrow T_1 , \ldots , i_n : x \longrightarrow T_n) :: \mathbf{TYPE}}$$

onde ⇒ significa implicação lógica.

Regra de Introdução :

$$\frac{a : \mathrm{Id} \quad i_1(a) = v_1 : T_1 \ \ldots \ i_n(a) = v_n : T_n}{\underline{\text{conceito}}\ a ::= (i_1 = v_1 ; \ldots ; i_n = v_n) : (\Sigma x . i_1 : x \longrightarrow T_1 , \ldots , i_n : x \longrightarrow T_n)}$$

Notar que em :

$$\underline{\text{conceito}}\ a ::= (\text{atributo}_1 = \text{Tipo}_1\ ;\ \ldots ; \text{atributo}_n = \text{Tipo}_n)$$

atributo_i é uma função, $\text{atributo}_i : \text{Id} \longrightarrow \text{Tipo}_i$,

tal que: $\text{atributo}_i(a) = \text{Valor}_i : \text{Tipo}_i$, de modo que a quantificação existencial Σx é sobre identificadores:

$$(\Sigma x \in \text{Id}.\text{atributo}_1 : x \longrightarrow T_1, \ldots, \text{atributo}_n : x \longrightarrow T_n)$$

Regras de Eliminação :

$$\frac{a : (\Sigma x.i_1 : x \longrightarrow T_1, \ldots, i_n : x \longrightarrow T_n)}{\text{nome-conceito}(a) : \text{Id}}$$

$$\frac{a : (\Sigma x.i_1 : x \longrightarrow T_1, \ldots, i_n : x \longrightarrow T_n)}{a.i_r : T_r}\ 0<r<n+1$$

$$\frac{a : (\Sigma x.i_1 : x \longrightarrow T_1, \ldots, i_n : x \longrightarrow T_n)}{\text{atributos}(a) : \text{Set}(\text{Par}[\text{Id}, T])}$$

Note que é possivel definir as regras de eliminação através de functores. Por exemplo, atributos é um functor que dado qualquer conceito obtém o conjunto dos atributos do mesmo. A vantagem disto é que permite definir outros operadores sobre conceitos, já que estes são valores de primeira classe.

O functor atributos seria definido como:

$$\text{atributos}:E_{\text{Id}} \longrightarrow \text{Set}[\text{Id} \times T]:(\underline{\text{conceito}}\ a ::= (i_1 = v_1 \ldots i_n = v_n) \longrightarrow \{(i_1, v_1), \ldots, (i_n, v_n)\})$$

Nesta definição existe uma constante de subespécie de tipo que será utilizada exclusivamente para tipos existenciais de classes que é a subespécie $E_{\text{Id}} \leq \textbf{TYPE}$. Frequentemente, porém, E_{Id} será escrito apenas como E, com possíveis subscritos indexadores.

Regra de Instanciação :

$$\frac{b:\mathrm{Id}\quad a:(\Sigma x.\, i_1:x\to T_1 \ldots i_n:x\to T_n)\quad i_p(b)=v_p:T'_p \ldots i_{p+k}(b)=v_{p+k}:T'_{p+k}}{\underline{\text{conceito}}\ \mathrm{b} ::= \underline{\text{instância}}\ (a,\ i_p = v_p \ldots i_{p+k} = v_{p+k}\ ):(\Sigma x \in \mathrm{Id}.\ i_1:x\to T_1 \ldots i_p:x\to T'_p \ldots i_{p+k}:x\to T'_{p+k} \ldots i_n:x\to T_n\ )}$$

Onde $1 \le p \le p+k \le n$; $T'_i \le T_i$ para $p \le i \le p+k$;

A regra de instanciação pode ser generalizada para suportar uma instanciação múltipla. Por questões de simplicidade sintática não será escrita essa regra, mas, ela consiste basicamente em colocar como premissas, em lugar de um único conceito a, um grupo de conceitos a_i.

Regra de Subtipagem :

$$\frac{b:\mathrm{Id}\quad a:(\Sigma x.i_1:x\to T_1 \ldots i_n:x\to T_n)\quad i_{n+1}(b)=v_{n+1}:T_{n+1} \ldots i_{n+k}(b)=v_{n+k}:T_{n+k}}{\underline{\text{conceito}}\ \mathrm{b} ::= \underline{\text{subconceito}}(a,\ i_{n+1} = v_{n+1} \ldots i_{n+k} = v_{n+k}\ ) : (\Sigma x \in \mathrm{Id}.\ i_1:x\to T_1 \ldots i_{n+k}:x\to T_{n+k})}$$

Como no caso anterior a generalização desta regra é simples e consiste em definir como premissas um grupo de conceitos a_i em lugar de um único conceito a.

Regras de Computação :

$$\text{nome-conceito}(\underline{\text{conceito}}\ a ::= (i_1 = v_1;\ \ldots;\ i_n = v_n)) \longrightarrow a$$

$$\text{atributos}(\underline{\text{conceito}}\ a ::= (i_1 = v_1;\ \ldots;\ i_n = v_n)) \longrightarrow ((i_1, v_1)\ ;\ \ldots;\ (i_n, v_n))$$

$$(\underline{\text{conceito}}\ a ::= (i_1 = v_1;\ \ldots;\ i_n = v_n)).i_k \longrightarrow v_k$$

6.3.7 REL

O tipo REL é basicamente um conjunto de pares de tipos Sigma. Uma relação, matematicamente falando, é um subconjunto do produto cartesiano entre dois conjuntos, i. é, $r \subseteq A \times B$.

Em RECON-II os conjuntos de interesse são os conceitos. A interpretação de uma relação entre conceitos é o mesmo que em matemática: um conjunto de pares.

Antes de definir as regras de inferência deste tipo é necessário definir um tipo especial que embora atômico deixou-se para ser definido aqui por manter uma ligação íntima com o tipo REL. Quando no capítulo anterior descreveu-se a semântica de RECON-II, foi definida uma álgebra chamada PRED, essa álgebra é a álgebra dos predicados lógicos que podem aparecer nas relações. Estes predicados são limitados, no estado atual da linguagem, a três: reflexiva, simétrica e transitiva. Dado que este tipo só é utilizado na construção de REL e é limitado a constantes sua definição foi adiada para esta seção do presente capítulo.

6.3.7.1 PRED

Regra de Formação :

$$\frac{}{\text{Pred} :: \textbf{TYPE}}$$

Regras de Introdução :

$$\frac{}{\text{reflexiva} : \text{Pred}} \qquad \frac{}{\text{simetrica} : \text{Pred}}$$

$$\frac{}{\text{transitiva} : \text{Pred}}$$

6.3.7.2 Regras para o tipo REL

Regra de Formação :

$$\frac{\text{Set[Pred]::TYPE} \quad E_{11} \times E_{12} :: \text{TYPE} \quad \ldots \quad E_{n1} \times E_{n2} :: \text{TYPE}}{\text{Rel(Set[Pred]; } (E_{11} \times E_{12}, \ldots, E_{n1} \times E_{n2})) :: \text{TYPE}}$$

Regra de Introdução :

$$\frac{r{:}\text{Id} \quad p{:}\text{Set[Pred]} \quad (a_1, b_1) : E_{11} \times E_{12} \ \ldots \ (a_{n1}, b_{n2}) : E_{n1} \times E_{n2}}{\begin{array}{l}\text{relação } r ::= \ \{\ \text{propriedades} = \langle p \rangle \\ \qquad\qquad \text{elementos} = \langle\ (a_1, b_1)\ ; \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad (a_n, b_n) \rangle \\ \qquad\qquad \}\ : \text{Rel(Set[Pred], } (E_{11} \times E_{12} \ \ldots \ E_{n1} \times E_{n2}))\end{array}}$$

Regras de Eliminação :

$$\frac{r : \text{Rel(Set[Pred], } (E_{11} \times E_{12} \ \ldots \ E_{n1} \times E_{n2}))}{\text{nome-rel(} r \text{)} : \text{Id}}$$

$$\frac{r : \text{Rel(Set[Pred], } (E_{11} \times E_{12} \ \ldots E_{n1} \times E_{n2}))}{\text{pares-rel(} r \text{)} : \text{Set[E x E]}}$$

$$\frac{r : \text{Rel(Set[Pred], } (E_{11} \times E_{12} \ \ldots \ E_{n1} \times E_{n2}))}{\text{pred-rel(} r \text{)} : \text{Set[Pred]}}$$

Dado que as relações são conjuntos, todas as operações conjunto-teoréticas são aplicáveis. Além destas operações sobre

relações se tem novas operações que usam o fato de que as relações não são meros conjuntos mas sim conjuntos de pares ordenados. Tais operações são a inversa e a composição de relações (ver definição no capítulo 4).

Quanto à relação de subtipagem para o tipo REL, a mesma será definida levando em conta a interpretação conjunto-teorética de REL, i. é, inclusão de conjuntos.

Inversão :

$\underline{\text{inversa}}$:Rel(Set[Pred],(E_{11} x E_{12} ... E_{n1} x E_{n2})) ⟶

Rel(Set[Pred],(E_{12} x E_{11}...E_{n2} x E_{n1})) :

$\underline{\text{relação}}$ r::=<$\underline{\text{propriedades}}$=<p>;$\underline{\text{elementos}}$=<($a_1$, b_1)...(a_n, b_n)>>

⟶ $\underline{\text{relação}}$ $\underline{\text{inversa}}$ (r)::=<$\underline{\text{propriedades}}$=<p>;

$\underline{\text{elementos}}$= <($b_1$, a_1) ... (b_n, a_n)>>

Composição :

$\underline{\text{composição}}$:Rel(E_{11} x E_{12}...E_{n1} x E_{n2})* x Rel(E_{12} x E'_{11}...E_{k2} x E'_{k1})*

⟶ Rel(E_{11} x E'_{11}...E_{k1} x E'_{k1}):

$\underline{\text{relação}}$ r::=<(a_1, b_1)...(a_n, b_n)>;$\underline{\text{relação}}$ r'::=<(b_1,a'_1)...(b_k,a'_k)>

⟶ $\underline{\text{relação}}$ $\underline{\text{composição}}$ (r•r') ::= <(a_1, a'_1)... (a_k, a'_k)>; k≤n

Note que r : Rel(E_{11} x E_{12}...E_{n1} x E_{n2})* denota o fecho de r, que é obtido a partir das propriedades da relação aplicadas ao conjunto de pares da mesma. Assim se, por exemplo, a relação tem a reflexividade como propriedade, deverá se calcular o fecho reflexivo sobre o conjunto de pares definidos na relação.

Regra de Subtipagem:

$$\frac{E_{11} \times E_{12} \leq E'_{11} \times E'_{12} \quad \ldots \quad E_{k1} \times E_{k2} \leq E'_{k1} \times E'_{k2}}{\text{Rel}(E_{11} \times E_{12} \ldots E_{n1} \times E_{n2})^* \leq \text{Rel}(E_{11} \times E_{12} \ldots E_{n1} \times E_{n2})^*} ; k \leq n$$

Regras de Computação :

nome-rel(relação r ::= { propriedades = ⟨p⟩

elementos = ⟨ (a_1, b_1) ;
⋮
$(a_n, b_n)\rangle\rangle \longrightarrow r$

pares-rel(relação r ::= { propriedades = ⟨p⟩

elementos = ⟨ (a_1, b_1) ;
⋮
$(a_n, b_n)\rangle\rangle \longrightarrow \{(a_1\ b_1)$
⋮
$(a_n, b_n)\}$

pred-rel(relação r ::= { propriedades = ⟨p⟩

elementos = ⟨ (a_1, b_1) ;
⋮
$(a_n, b_n)\rangle\rangle \longrightarrow$ p

6.3.8 TIPO REDE

Uma Rede pode ser interpretada como uma Estrutura de Primeira Ordem (ver seção 3.9) $\mathcal{U} = \langle \mathcal{S}, \mathcal{F}, \mathcal{R} \rangle$, onde a família de conjuntos $\mathcal{S}$ é composta pelos *carriers* dos conceitos da rede, i. é, um *carrier* de conceito é um *carrier* da estrutura $\mathcal{U}$.

O conjunto de operações $\mathcal{F}$ é formado pelas funções definidas na rede; enquanto que as relações constituem o conjunto $\mathcal{R}$.

Note que a interpretação feita dos conceitos na seção 6.3.6, onde eles eram vistos como álgebras permite esta interpretação de conjunto, já que dada uma álgebra é possível manipular seu *carrier* independentemente das operações, como simples conjunto.

Por outro lado, é possível generalizar a interpretação

algébrica dos conceitos para uma interpretação de estrutura de primeira ordem. Um conceito é uma estrutura de primeira ordem $\mathfrak{B} = \langle \mathcal{A}, \mathcal{G}, \mathcal{R} \rangle$ onde $\mathcal{R} = \emptyset$.

Esta generalização é muito útil porque permite estender a linguagem RECON_II, estendendo a noção de classe para englobar a noção de relação, para enriquecer, desta maneira, a noção de Rede, definindo funções sobre Redes, relações entre Redes e o mais importante, definindo uma ordem parcial entre Redes (sub-rede, super-rede).

Regra de Formação :

$E_1 :: \mathbf{TYPE} \ \ldots \ E_n :: \mathbf{TYPE}$

$i_1 : E_1 \times \ldots \times E_n \longrightarrow E_{j1} \qquad [1 \leq j1 \leq n]$

$\vdots$

$i_m : E_1 \times \ldots \times E_n \longrightarrow E_{jm} \qquad [1 \leq jm \leq n]$

$R_1 \leq E_1 \times \ldots \times E_n$

$\vdots$

$R_p \leq E_1 \times \ldots \times E_n$

$(\Omega x.\ i_1 : x \longrightarrow E_{j1}, \ \ldots, \ i_m : x \longrightarrow E_{jm}, \ R_1 \leq x, \ \ldots, \ R_p \leq x) :: \mathbf{TYPE}$

Notar que as operações se aplicam sobre toda a rede. A relação de subtipo para produto cartesiamo deve permitir que os R_i sejam projeções de $E_1 \times \ldots \times E_n$.

Regra de Introdução :

$$\frac{b : \mathrm{id} \quad a_1 : E_1 \ldots a_n : E_n \quad f_1 : S_1 \longrightarrow T_1 \ldots f_m : S_m \longrightarrow T_m \quad r_1 : R_1 \ldots r_p : R_p}{\underline{\text{rede}}\ i ::= \langle \langle a_1, \ldots, a_n \rangle ; \langle f_1, \ldots, f_m \rangle ; \langle r_1, \ldots, r_p \rangle \rangle \quad \underline{\text{fim-rede}} : (\Omega x.\ f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m, \ R_1 \leq x, \ldots, R_p \leq x)}$$

Onde $S_i = E_1 \times \ldots \times E_n$, $T_i = E_j$ $(1 \leq j \leq n)$ e R_i é um tipo REL como foi definido anteriormente.

Regras de Eliminação :

$$\frac{b : (\Omega x.\, f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \;,\; R_1 \leq x, \ldots, R_p \leq x)}{\underline{\text{nome-rede}}\ (\ b\ ) : \text{id}}$$

$$\frac{b : (\Omega x.\, f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \;,\; R_1 \leq x, \ldots, R_p \leq x)}{\underline{\text{conceitos-rede}}\ (\ b\ ) : \text{Set [E]}}$$

$$\frac{b : (\Omega x.\, f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \;,\; R_1 \leq x, \ldots, R_p \leq x)}{\underline{\text{funções-rede}}\ (\ b\ ) : \text{Set[S} \longrightarrow \text{T]}}$$

$$\frac{b : (\Omega x.\, f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \;,\; R_1 \leq x, \ldots, R_p \leq x)}{\underline{\text{relações-rede}}\ (\ b\ ) : \text{Set[R]}}$$

Regra de Subtipagem :

$$\frac{E_1 \leq E'_1 \ldots E_n \leq E'_n \quad S_1 \longrightarrow T_1 \leq S'_1 \longrightarrow T'_1 \ldots S_m \longrightarrow T_m \leq S'_m \longrightarrow T'_m \quad R_1 \leq R'_1 \ldots R_p \leq R'_p}{\begin{array}{l}(\Omega x.\, f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \;,\; R_1 \leq x, \ldots, R_p \leq x) \quad \leq \\ (\Omega y.\, f'_1 : y \longrightarrow T'_1 \ldots f'_m : y \longrightarrow T'_m \;,\; R'_1 \leq y, \ldots, R'_p \leq y)\end{array}}$$

Esta regra de subtipagem corresponde a subtipagem implícita. Ela é interessante quando se quer comparar duas redes. Mas, como no caso dos conceitos, é possível definir uma regra explícita que seria utilizada para estruturar oconhecimento. O verificador de tipos poderia ter as duas regras definidas e a decisão de quando ativar uma ou outra poderia ser inferida do contexto.

A regra de subtipagem explícita é uma extensão da definida para os conceitos (seção 6.3.6).

Regras de Computação :

nome-rede (rede i ::= { < a_1, ..., a_n > ;
< f_1, ..., f_k >; < r_1, ..., r_s >)fim-rede ⟶ i

conceitos-rede (rede i ::= { < a_1, ..., a_n > ;
< f_1, ..., f_k >; < r_1, ..., r_s >)fim-rede ⟶ < a_1, ..., a_n >

funções-rede (rede i ::= { < a_1, ..., a_n > ;
< f_1, ..., f_k >; < r_1, ..., r_s >)fim-rede ⟶ < f_1, ..., f_k >

relações-rede (rede i ::= { < a_1, ..., a_n > ;
< f_1, ..., f_k >; < r_1, ..., r_s >)fim-rede ⟶ < r_1, ..., r_s >

6.4 Linguagem de Tipos

A Linguagem de Tipos de RECON-II (LTR-II) está composta por tipos básicos pré-definidos e construtores de tipos. Os tipos pré-definidos formam o conjunto dos Tipos Básicos $\mathcal{B}$ e são **Bool**, **Int**, **String**, **Char** e **Null**, todos equipados com suas operações usuais.

O conjunto dos Construtores de Tipos $\mathcal{C}$ permite definir os tipos do seguintes valores: listas, conjuntos, produtos cartesianos, funções, conceitos, relações e redes.

A LTR-II permite também utilizar *variáveis de tipos*, fornecendo assim uma característica muito importante: o polimorfismo.

Uma expressão de tipo então é um tipo básico ou uma variável de tipos ou é formada pela aplicação de um construtor de tipo a outra expressão de tipo. Formalmente o conjunto $\mathcal{E}$ das expressões de tipo é definido como segue:

Seja $\mathcal{B}$ o conjunto dos tipos básicos da RECON-II, $\mathcal{C}$ o conjunto dos construtores de tipos de RECON-II e seja $\mathcal{X}$ um conjunto infinito enumerável de variáveis de tipo.

Define-se $\mathcal{E}$ como o conjunto das expressões de tipos de RECON-II se e somente se:

- $e \in \mathcal{B}$ então $e \in \mathcal{E}$,
- $x \in \mathcal{X}$ então $x \in \mathcal{E}$,
- $f_n \in \mathcal{C} \wedge x_1, x_2, \ldots, x_n \in \mathcal{E}$ então
  $f_n(x_1, x_2, \ldots, x_n) \in \mathcal{E}$,
- nada mais pertence a $\mathcal{E}$.

Uma outra definição de LTR-II é :

Dado o conjunto dos tipos básicos $\mathcal{B}$ e um conjunto infinito enumerável de variáveis de tipo $\mathcal{X}$, a sintaxe dos tipos define-se como segue:

- Se $\tau \in \mathcal{B}$ então τ é um tipo,
- Se $\tau \in \mathcal{X}$ então τ é um tipo,
- Se τ é um tipo então **List[**τ**]** é um tipo,
- Se τ é um tipo então **Set[**τ**]** é um tipo,
- Se τ e σ são tipos então $\tau \times \sigma$ é um tipo,
- Se τ e σ são tipos então $\tau \longrightarrow \sigma$ é um tipo,
- Se $\tau_1, \tau_2 \ldots \tau_n$ são tipos então $(\Sigma x \in Id.\ i_1 : x \longrightarrow \tau_1 \ldots i_n : x \longrightarrow \tau_n)$ é um tipo,
- Se $\tau_1, \tau_2 \ldots \tau_n$ são tipos e $\sigma_1, \sigma_2 \ldots \sigma_n$ são tipos então **Rel (Set[Pred]**, $\tau_1 \times \sigma_1, \ldots, \tau_n \times \sigma_n$) é tipo,
- Se $\tau_1, \tau_2 \ldots \tau_n$ são tipos e $\delta_1, \delta_2 \ldots \delta_p$ são tipos então $(\Omega x. f_1 : x \longrightarrow \tau_{1j}, \ldots, f_m : x \longrightarrow T_{mj}, \delta_1 \leq x, \ldots, \delta_p \leq x)$.
- nada mais é um tipo.

7 CONCLUSÃO

O Sistema de Tipos proposto no capítulo anterior representa,junto com o capítulo 5, a essência deste trabalho. Tanto um como outro ajudaram na depuração da linguagem RECON-II, que foi sendo modificada durante o desenvolvimento deste trabalho. Esta é a característica fundamental que diferencia este trabalho de outros onde o pesquisador escolhe uma linguagem já definida e especifica sua semântica. Aqui o processo foi interativo, a partir do núcleo da linguagem, este foi sendo estudado e refinado. Por isso, no processo de especificar uma semântica para RECON-II foram achados pontos contraditórios, mal definidos, ambígüos e até indefinidos que, graças a esse estudo foram detectados e corrigidos. Como exemplo pode-se citar a inexistência de funções de alta ordem, fato que restringia muito o poder de expressão da linguagem; também, a noção de rede não era um valor de primeira ordem, i. é, não era possível nem operar nem comparar duas redes. Quanto aos conceitos, estes não podiam aparecer como sendo valor de um atributo de outro conceito.

Assim, a semântica ajudou eliminando problemas na definição da RECON-II, determinando os domínios semânticos do universo de discurso da linguagem (tipos semânticos) e fornecendo todas as vantagens de uma especificação semântica: a definição precisa de conceitos, independente de qualquer implementação, e a utilização de técnicas de especificação não ambígüas, baseadas numa teoria formal, que podem ser usadas para gerar teoremas sobre a especificação. Uma última vantagem, não menos importante, é que a semântica ajuda na criação de uma boa documentação da linguagem.

A principal contribuição deste trabalho foi fornecer uma noção de tipo para o conceito de classe (conceito em RECON-II). Na literatura, geralmente, encontra-se análises simples que fornecem uma noção de tipo onde a classe é um tipo e suas instâncias tem a classe como tipo. Neste trabalho, um conceito tem um tipo, que é uma abstração existencial sobre o identificador do conceito, e suas instâncias são termos da

álgebra que o conceito denota, i. é, as instâncias de um conceito também possuem um tipo existencial Σ, subtipo do tipo do conceito.

Talvéz o mais importante, é que uma rede conceitual também possui um tipo que é uma extensão natural do tipo existencial Σ. Assim, as redes tornam-se objetos de primeira classe na linguagem, permitindo definir operações sobre elas, assim como compará-las, facilitando, desta maneira, o estudo dos processos de aquisição de conhecimentos. Em outras palavras, definiu-se um terceiro nível na hierarquia de tipos, criando um nivel meta de manipulação do universo de discurso.

Por último, muitos aspectos pesquisados tiveram que ser deixados de lado por questões de complexidade e tempo, permanecendo, só aqueles tópicos diretamente relacionados com o objetivo principal da dissertação. Dentre os aspectos não incluídos e que poderiam ser de interesse estudar mais profundamente, encontram-se: definição da classe de modelos não isomórficos para RECON-II, construção de um algoritmo de inferência para o sistema de tipos dado, definição de uma semântica denotacional para a RECON-II, construção de provas de teoremas usando a especificação algébrica, análise do comportamento dos tipos na definição de redes conceituais por parte dos usuários. Este último aspecto requer a implementação do algoritmo de inferência de tipos a fim de realizar testes adequados, especialmente com referência às facilidades de estruturação do conhecimento.

ANEXO A : SINTAXE DE RECON-II

```
<declaracao_de_rede> ::= { conceptual_net : <id> }
                                    <declarações>
                                {end}.

<id> ::= <letra> | <letra><alfanumericos> .

<declarações> ::= <declaração> | <declaração> <declarações>.
<declaração> ::= <conceito> | <relação> | <função>.

<conceito> ::=  { concept : <id> <descricao> }.

<relação> ::= ( <corpo> <descricao> ) .
<corpo> ::=       <nome_rel> <elementos> |
                  <nome_rel> <propriedades> |
                  <nome_rel> <propriedades> <elementos> .

<nome_rel> ::= relation : <id> .
<propriedades> ::= properties : [ <objetos> ]; .
<elementos> ::= elements : <declaracao_elementos>; .
<declaracao_elementos> ::=        <tupla> |
                                  <tupla> , <declaracao_elementos>.
<tupla> ::=  ( <id>, <id> ).

<descricao> ::=         <atributo> |
                        <atributo> ; <descricao> .
<atributo> ::= <id> : <valor>  |
               <id> : <descricao> .
<valor> ::= <id>
          | <inteiro>
          | <booleano>
          | <string>
          | <conjunto> .

<função> ::= { function : <id>
               domain : ( <dom> )
               body : lambda ( <objetos> ).
```

[<expressao>] ; } .

<dom> ::= <id> ⟶ <id> |
<id> , <dom> .

<objetos> ::= <objeto> |
<objeto> , <objetos> .
<objeto> :: = <id> .

<expressao> ::= <id> |
(<expressao>) <operador> (<expressao>) |
arco(<expressao>, <expressao>, <expressao>) |
<chama_função> |
if <expressao>
then <expressao>
else <expressao> .

<chama_funcao> ::= <id> (<argumentos>).
<argumentos> ::= <id> |
<id> , <argumentos> .

<operador> ::= union |
inter |
minus |
proj |
rel |
equal |
or |
and |
not |
< |
> |
<> |
≤ |
≥ .

ANEXO B : ESPECIFICAÇÃO ALGÉBRICA DE RECON-II

Este anexo é um resumo das álgebras definidas no capítulo 4. O objetivo deste anexo é oferecer ao leitor uma procura rápida das álgebras que modelam expressões-RECON.

1.

type INT-BASIC

based on BOOL

sort int

operations

zero: $\longrightarrow$ int

succ: int $\longrightarrow$ int

>0? : int $\longrightarrow$ bool

pred: int $\longrightarrow$ int

eq-int:int x int $\longrightarrow$ bool

norm: int $\longrightarrow$ int

is-0?: int $\longrightarrow$ bool

axioms

$\forall a, b, c \in$ int

1. >0?(*a*) = >'0?(norm(*a*))
1. >'0?(zero) = false
2. >'0?(succ(*a*) = true
3. >'0?(pred(*a*) = false
4. norm(pred(zero)) = pred(zero)
5. norm(succ(zero)) = succ(zero)
6. norm(zero) = zero
7. norm(succ(pred(*a*)) = norm(*a*)
8. norm(pred(succ(*a*)) = norm(*a*)
9. norm(succ(succ(*a*))) = norm'(succ(norm(succ(*a*))))
10. norm(pred(pred(*a*))) = norm'(pred(norm(pred(*a*))))
11. norm'(succ(succ(*a*))) = succ(succ(*a*))
12. norm'(pred(pred(*a*))) = pred(pred(*a*))
13. norm'(succ(pred(*a*))) = *a*
14. norm'(pred(succ(*a*))) = *a*
15. succ(pred(*a*)) = *a*
16. pred(succ(*a*)) = *a*
17. eq-int(*a*, *b*)=true $\Rightarrow$ eq-int(succ(*a*), succ(*b*))=true

18.eq-int(*a*, *b*)=true ⇒ eq-int(pred(*a*), pred(*b*))=true

17.eq-int(*a*, *b*)=false ⇒ eq-int(succ(*a*), succ(*b*))=false

18.eq-int(*a*, *b*)=false ⇒ eq-int(pred(*a*), pred(*b*))=false

19.eq-int(zero, *a*) = is-0?(*a*)

20.eq-int(*a*, zero) = is-0?(*a*)

21.is-0?(*a*) = is-0'?(norm(*a*))

22.is-0'?(zero) = true

23.is-0'?(succ(*a*)) = false

24.is-0'?(pred(*a*)) = false

endoftype

2.

type INT

based on INT-BASIC

sort int

operations

_+_: int x int ⟶ int

_-_: int x int ⟶ int

_*_: int x int ⟶ int

_/_: int x int ⟶ int

abs : int ⟶ int

sign: int x int ⟶ int

->- : int x int ⟶ bool

axioms

∀ *a*, *b* , *c*, *d* ∈ int

1. *a* + zero = *a*

2. *a* + succ(*b*) = succ(*a* + *b*)

3. *a* + pred(*b*) = pred (*a* + *b*)

4. *a* - zero = *a*

5. *a* - succ(*b*) = pred(*a* - *b*)

6. *a* - pred(*b*) = succ(*a* - *b*)

7. *a* * zero = zero

8. *a* * succ(*b*) = (*a* * *b*) + *a*

9. *a* * pred(*b*) = (*a* * *b*) - *a*

10.abs(*a*) = abs'(norm(*a*))

11.abs'(zero) = zero

12.abs'(succ(*a*))= succ(*a*)

13.abs'(pred(*a*)) = succ(abs'(*a*))

```
14.a > b = norm(a) >' norm(b)
15.pred(a) >' pred(b) = a >'b
16.zero >' pred(a) = true
17.pred(a) >' zero = false
18.succ(a) >' succ(b) = a >' b
19.zero >' succ(a) = false
20.succ(a) >' zero = true
21.pred(a) >' succ(b) = false
22.succ(a) >' pre(b) = true
23. zero >' zero = false
24.sign(a, b) = if (>-0?(a) .or. is-0?(a)) .and.
                        (>-0?(b) .or. is-0?(b))
                  then succ(zero)
                  else pred(zero)
25.a/b = if abs(a) > abs(b) .or. eq-int(abs(a), abs(b))
      then  ((abs(a)  -  abs(b))/abs(b)  +  succ(zero))  *
            sign(a, b))
            else zero
endoftype
```

3.

```
type STRING
based on CHAR
sort str
operations
      "":          ⟶ str
      add-ch:str x char ⟶ str
      concat:     str x str ⟶ str
      eq-str:     str x str ⟶ bool
      prim-ch:    str ⟶ char
      null-str?: str ⟶ bool
      str-id: str ⟶ id
      resto-str: str ⟶ str
axioms
      ∀ a,b ∈ char    ∀ c, d, e ∈ str
      1. null-str?(" ") = true
      2. null-str?(add-ch(c, a) = false
      3. concat(c, " ") = c
```

```
        4. concat(c, add-ch(d, a)) = add-ch(concat(c, d), a)
        5. prim-ch(add-ch("",a) = a
        6. prim-ch( add-ch( add-ch(c,  a),  b)) =
                                prim-ch( add-ch(c, a)
        7. eq-str(c, " ") = null-str?(c)
        8. eq-str(" ", add-ch(c, a)) = false
        9.  eq-str( add-ch(c,  a), add-ch(d,  b)) =  eq-char(a,  b)
                                .and. eq-str(c, d)
        10.str-id(c) = if letra?(prim-ch(c))
                        then c
                        else concat("a", c)
        11. resto-str(add-str(c, a)) = c
endoftype
```

4.

```
type VALOR
based on INT, STRING, BOOL
sort val
operations
        null: ⟶ val
        nulo?: val ⟶ bool
        num-val: int ⟶ val
        str-val: str ⟶ val
        bool-val:bool⟶ val
        numero?:val ⟶ bool
        string?:val ⟶ bool
        boolean?:val ⟶bool
        list-val:lst ⟶ val
        list?:val ⟶ bool
        set-val:set ⟶ val
        set?:val ⟶ bool
        par-val:par ⟶ val
        par?:val ⟶ bool
        class-val:class ⟶ val
        class?:val ⟶ bool
        rel-val:rel ⟶ val
        rel?:val ⟶ bool
        fun-val:fun ⟶ val
```

fun?:val ⟶ bool

axioms

1. ∀a ∈ int numero?(num-val(a)) = true
   numero?(null) = false
2. ∀a ∈ str string?(str-val(a)) = true
   string?(null) = false
3. ∀a ∈ bool boolean?(bool-val(a)) = true
   boolean?(null) = false
4. ∀a ∈ lst list?(list-val(a)) = true
   list?(null) = false
5. ∀a ∈ set set?(set-val(a)) = true
   set?(null) = false
6. ∀a ∈ par par?(par-val(a)) = true
   par?(null) = false
7. ∀a ∈ class class?(class-val(a)) = true
   class?(null) = false
8. ∀a ∈ rel rel?(rel-val(a)) = true
   rel?(null) = false
9. ∀a ∈ fun fun?(fun-val(a)) = true
   fun?(null) = false
10. nulo?(null) = true
11. ∀a ∈ int nulo?(num-val(a)) = false
12. ∀a ∈ str nulo?(str-val(a)) = false
13. ∀a ∈ bool nulo?(bool-val(a)) = false
14. ∀a ∈ lst nulo?(list-val(a)) = false
15. ∀a ∈ set nulo?(set-val(a)) = false
16. ∀a ∈ par nulo?(par-val(a)) = false
17. ∀a ∈ class nulo?(class-val(a)) = false
18. ∀a ∈ rel nulo?(rel-val(a)) = false
19. ∀a ∈ fun nulo?(fun-val(a)) = false

endoftype

5.

type LIST

parameters Item **with:** equal:Item x Item ⟶ bool

VALOR **expanded with**

sort lst

operations

nil: $\longrightarrow$ lst

_|_: Item x lst $\longrightarrow$ lst

cab: lst $\longrightarrow$ Item

resto: lst $\longrightarrow$ lst

$\in$-lst?: Item x lst $\longrightarrow$ bool

eq-lst: lst x lst $\longrightarrow$ lst

nil?: lst $\longrightarrow$ bool

axioms

$\forall$ *a*, *b* $\in$ lst $\forall$ *c*, *d* $\in$ Item

1. cab(*c*|*a*) = *c*
2. resto(*c*|*a*) = *a*
3. eq-lst(*a*, nil) = nil?(*a*)
4. eq-lst(nil, *c*|*a*) = false
5. eq-lst(*c*|*a*, nil) = false
6. eq-lst(*c*|*a*, *d*|*b*) = eq-lst(*a*, *b*) .and. equal(*c*, *d*)
7. $\in$-lst?(*c*, *d*|*a*) = if equal(*c*, *d*) then true
   else $\in$-lst?(*c*, *a*)
8. $\in$-list?(*c*, nil) = false
9. nil?(nil) = true
10. nil?(*c*|*a*) = false

endoftype

6.

type PAR-VALOR

parameters Prim, Seg **with** eq1:Prim x Prim $\longrightarrow$ bool
eq2:Seg x Seg $\longrightarrow$ bool

VALOR **expanded with**

sort par

operations

par-elem: Prim x Seg $\longrightarrow$ par

prim-elem: par $\longrightarrow$ Prim

seg-elem: par $\longrightarrow$ Seg

eq-par:par x par $\longrightarrow$ bool

axioms

$\forall$ *a*, *b* $\in$ par $\forall$ *c* $\in$ Prim $\forall$ *d* $\in$ Seg

1. par-elem(prim-elem(*a*), seg-elem(*a*)) = *a*
2. prim-elem(par-elem(*c*, *d*)) = *c*

3. seg-elem(par-elem(c, d)) = d

4. eq-par(par-elem(c_1, d_1), par-elem(c_2, d_2)) =
   eq1(c_1, c_2) .and. eq2(d_1, d_2)

endoftype

7.

type SET

parameters Elem **with** equal: Elem x Elem ⟶ bool

VALOR **expanded with**

sort set

operations

set(): ⟶ set

ins-elem: Elem x set ⟶ set

del-elem: Elem x set ⟶ set

-∪-: set x set ⟶ set

-∩-: set x set ⟶ set

-menos-: set x set ⟶ set

∅-set?: set ⟶ bool

⊆-set?: set x set ⟶ bool

∈-set?: set x Elem ⟶ bool

eq-set: set x set ⟶ bool

axioms

∀ a, b, c ∈ set ∀ e, e_1,e_2 ∈ Elem

1. ∅-set?(set()) = true

2. ∅-set?(ins-elem (e, a)) = false

3. ∈-set?(e_1, ins-elem(e_2, a)) = if equal(e_1, e_2)
   then true
   else ∈-set?(e_1, a)

4. ins-elem(e_1, ins-elem(e_2, a)) = ins-elem(e_2,
   ins-elem(e_1, a))

5. ∈-set?(e, set()) = false

6. a ∪ a = a

7. a ∪ b = b ∪ a

8. a ∪ set() = a

9. a ∪ ins-elem(e, b) = ins-elem(e, a ∪ b)

10. a ∩ a = a

11. a ∩ b = b ∩ a

12. a ∩ set() = set()

```
13.a ∩ ins-elem(e, b) = if ∈-set?(e, a)
                        then ins-elem(e, a ∩ b)
                        else a ∩ b
15.a menos a = set()
16.a menos set() = a
17.set() menos a = set()
18.ins-elem(e, b) menos a = if ∈-set?(e, a)
                            then del-elem(e, b) menos a
                            else ins-elem(e, b menos a)
19.set?(set()) = true
20.set?(ins-elem(e, a)) = set?(a)
21.⊆-set?(ins-elem(e, a), set()) = false
22.⊆-set?(ins-elem(e, a), b) = if ∈-set?(e, b)
                              then ⊆-set?(del-elem(e, a),
                                          del-elem(e, b))
                              else false
23.⊆-set?(set(), a) = true
24.del-elem(e, set()) = set()
25.del-elem(e1, ins(e2, a)) = if equal(e1, e2)
                              then del-elem(e1, a)
                         else (ins-elem(e2, del-elem(e1, a))
26.eq-set(a, b) = ⊆-set?(a, b) .and. ⊆-set?(b, a)
endoftype
```

8.

```
type ATRIB-VALOR
instance of PAR-VALOR
     with
     Prim      as   id
     Seg       as   val
rename par     as   av
endoftype
```

9.

```
type AV-SET
instance of SET
     with
     Elem       as   av
rename set      as   av-set
endoftype
```

10.

```
type ATVAL-SET
AV-SET expanded with
operations
     match: id x av-set ⟶ val
axioms    ∀ a ∈ av-set   ∀ i, e ∈ id          ∀ υ ∈ val
     1. match(i, set()) = null
     2. match(i, ins-elem(par-elem(e, υ), a)) =
        if  eq-id(i, e) then υ else match(i, a)
endoftype
```

11.

```
type CLASSE
VALOR expanded with
sort class
operations
     cria-cl: id x av-set ⟶ class
     nome-cl: class ⟶ id
     slots: class ⟶ av-set
     valor-slot: class x id ⟶ val
     add-slots:class x av ⟶ class
     del-slots:class x av ⟶ class
     union-cl: class x class ⟶ class
     inter-cl: class x class ⟶ class
     menos-cl: class x class ⟶ class
     eq-cl: class x class ⟶ bool
     ≤-cl?: class x class ⟶ bool
```

axioms $\forall a \in$ class $\forall b, b_1, b_2 \ldots \in$ av-set
$\forall i, i_1, \ldots, \in$ id $\forall c \in$ av

1. cria-cl(nome-cl(a), slots(a)) = a
2. nome-cl(cria-cl(i, b)) = i
3. slots(cria-cl(i, b)) = b
4. valor-slot(cria-cl(i, b), i_1) = match(i_1, b)
5. add-slots(cria-cl(i, b), c) = cria-cl(i, ins-elem(c, b))
6. del-slots(cria-cl(i, b), c) = cria-cl(i, del-elem(c, b))
7. union-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = cria-cl(str-id(concat(concat(id-str(i_1),"union"), id-str(i_2))), $b_1 \cup b_2$)
8. inter-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = cria-cl(str-id(concat(concat(id-str(i_1), "inter"), id-str(i_2))), $b_1 \cap b_2$)
9. menos-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = cria-cl(str-id(concat(concat(id-str(i_1), "menos"), id-str(i_2))), b_1 menos b_2)
10. eq-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = eq-set(b_1, b_2)
11. $\subseteq$-cl(cria-cl(i_1, b_1), cria-cl(i_2, b_2)) = $\subseteq$-set(b_1, b_2)

endoftype

12.

```
type CLASS-PAR
instance of PAR-VALOR
     with      Prim      as   class
               Seg       as   class
rename par     as   par-cl
endoftype
```

13.

```
type CLASS-SET
 instance of SET
      with
      Elem        as   par-cl
 rename set       as   class-set
 endoftype
```

14.

```
type PRED
based on IDENT
sort pred
operations
      reflexiva: ⟶ pred
      simétrica: ⟶ pred
      transitiva: ⟶ pred
endoftype
```

15.

```
type PRED-SET
 instance of SET
      with
      Elem        as   pred
 rename set       as   pred-set
 endoftype
```

16.

```
type REL
VALOR expanded with
sort rel
operations
      cria-rel: id x class-set x pred-set ⟶ rel
      nome-rel: rel ⟶ id
      pares-rel: rel ⟶ class-set
      add-par: rel x par-cl ⟶ rel
      del-par: relXpar-cl ⟶ rel
      pred-rel: rel ⟶ pred-set
      add-pred: rel x pred ⟶ rel
```

del-pred: rel x pred ⟶ rel

aplic-pred: rel ⟶ class-set

reflex: class-set ⟶ class-set

simet: class-set ⟶ class-set

transit: class-set ⟶ class-set

rel-cl: rel x class ⟶ class-set

union-rel: rel x rel ⟶ rel

inter-rel: rel x rel ⟶ rel

menos-rel: rel x rel ⟶ rel

eq-rel: rel x rel ⟶ bool

≤-rel?: rel x rel ⟶ bool

axioms

∀ $a, b \in$ rel ∀ $c \in$ class ∀ $p \in$ pred ∀ $i \in$ id

∀ $d \in$ par-cl ∀ $s \in$ class-set ∀ $t \in$ pred-set

1. cria-rel(nome-rel(a), pares-rel(a), pred-rel(a)) = a
2. nome-rel(cria-rel(i, s, t)) = i
3. pares-rel(cria-rel(i, s, t)) = s
4. pred-rel(cria-rel(i, s, t)) = t
5. add-par(cria-rel(i, s, t), d) = cria-rel(i, ins-elem(d, s), t)
6. del-par(cria-rel(i, s, t), d) = cria-rel(i, del-elem(d, s), t)
7. add-pred(cria-rel(i, s, t), p) = cria-rel(i, s, ins-elem(p, t))
8. del-pred(cria-rel(i, s, t), p) = cria-rel(i, s, del-elem(p, t))
9. union-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2))
   = cria-rel(concat(concat(i_1,"union"), i_2), $s_1 \cup s_2$, $t_1 \cup t_2$)
10. inter-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2))
    = cria-rel(concat(concat(i_1,"inter"), i_2), $s_1 \cap s_2$, $t_1 \cap t_2$)
11. menos-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2))
    = cria-rel(concat(concat(i_1,"menos"), i_2), s_1 menos s_2, t_1 menos t_2)
12. eq-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2)) =
    eq-set(aplic-pred(cria-rel(i_1, s_1, t_1), aplic-pred(cria-rel(i_2, s_2, t_2))

```
13.aplic-pred(cria-rel(i, s, t)) =
      case ∈-set?(reflexiva, t) .and.
           ∈-set?(simétrica, t) .and.
           ∈-set?(transitiva, t)
           s ∪ (reflex(s) ∪ (simet(s) ∪ transit(s)))
      case ∈-set?(reflexiva, t) .and.
           ∈-set?(simétrica, t)
           s ∪ (reflex(s) ∪ simet(s))
      case ∈-set?(reflexiva, t) .and.
           ∈-set?(transitiva, t)
           s ∪ (reflex(s) ∪ transit(s))
      case ∈-set?(simétrica, t) .and.
           ∈-set?(transitiva, t)
           s ∪ (simet(s) ∪ transit(s))
      case ∈-set?(reflexiva, t)
           s ∪ reflex(s)
      case ∈-set?(simétrica, t)
           s ∪ simet(s)
      case ∈-set?(transitiva, t)
           s ∪ transit(s)
      case ∅-set?(t)
           s
14.reflex(set()) = set()
15.reflex(ins-elem(par-elem(c1, c2), s)) =
       ins-elem(par-elem(c1,c1), ins-elem(par-elem(c2,c2),
   reflex(s)
16.simet(set()) = set()
17.simet(ins-elem(par-elem(c1, c2), s) =
    ins-elem(par-elem(c2, c1), simet(s)
18.transit(set()) = set()
19.transit(ins-elem( par-elem(c1, c2), ins-elem(
           par-elem(c3, c4), s) = if eq-cl(c1, c4)
     then ins-elem(par-elem(c1, c4),transit(ins-elem(
         par-elem(c1, c2), ins-elem(par-elem(c3, c4),
     ins-elem(par-elem(c1, c4), s)))))
   else
     if eq-cl(c2, c3)
     then ins-elem(par-elem(c3, c2),transit(ins-elem(
```

par-elem(c_1, c_2), ins-elem(par-elem(c_3, c_4),
ins-elem(par-elem(c_3, c_2), s)))))
else transit(ins-elem(par-elem((c_1, c_2), s) ∪
transit(ins-elem(par-elem((c_3, c_4), s)
20.transit(ins-elem(par-elem((c_1, c_2), set()) =
if eq-cl(c_1, c_2) then ins-elem(par-elem(
(c_1, c_2),set())
else set()
21.≤-rel(cria-rel(i_1, s_1, t_1), cria-rel(i_2, s_2, t_2)) =
≤-set(aplic-pred(cria-rel(i_1, s_1, t_1),
aplic-pred(cria-rel(i_2, s_2, t_2))
endoftype

17. CLUSTER

17.1

type FUN
VALOR expanded with
sort fun
operations
lambda: id x id x expr ⟶ fun
aplic: fun x val ⟶ val
eq-fun: fun x fun ⟶ bool
compos: fun x fun ⟶ fun
axioms
∀ f, g ∈ fun ∀ a, b ∈ val ∀ e ∈ expr
∀ i, x, ∈ id
1. aplic(lambda(i, x, e), a) = eval(subst(cte(a),e, x))
2. aplic(f, null) = fun-val(f)
3. fun?(lambda(i, x, e)) = true
4. eq-fun(lambda(i_1, x_1, e_1), lambda(i_2, x_2, e_2)) =
eq-expr(e_1, subst(x_1, e, x_2)
5. compos(lambda(i_1, x_1, e_1), lambda(i_2, x_2, e_2)) =
lambda(str-id(concat(concat(id-str(i_1), '•'),
id-str(i_2))), x_1, subst(e_1, e_2,x_2))
endoftype

17.2

type EXPRESSÃO

based on VALOR

sort expr

operations

eval: expr ⟶ val

var: id ⟶ expr

cte: val ⟶ expr

subst: expr x expr x id ⟶ expr

$\underline{+}$: expr x expr ⟶ expr

$\underline{-}$: expr x expr ⟶ expr

$\underline{*}$: expr x expr ⟶ expr

$\underline{/}$: expr x expr ⟶ expr

$\underline{\cup}$: expr x expr ⟶ expr

$\underline{\cap}$: expr x expr ⟶ expr

menos : expr x expr ⟶ expr

concat : expr x expr ⟶ expr

$\underline{!}$: expr x expr ⟶ expr

...

union-cl : expr x expr ⟶ expr

inter-cl : expr x expr ⟶ expr

menos-cl : expr x expr ⟶ expr

union-rel : expr x expr ⟶ expr

inter-rel : expr x expr ⟶ expr

menos-rel : expr x expr ⟶ expr

compos : expr x expr ⟶ expr

aplic : expr x expr ⟶ expr

if-then-else : expr x expr x expr ⟶ expr

axioms

∀ a, b ∈ expr ∀ v ∈ val ∀ x, y ∈ id

1. eval(var(x)) = erro
2. eval(cte(v)) = v
3. eval(a $\underline{+}$ b) = eval(a) + eval(b)
4. eval(a $\underline{\cup}$ b) = eval(a) ∪ eval(b)
5. eval(union-cl (a, b)) = union-cl(eval(a), eval(b))
6. eval(union-rel(a, b)) = union-rel(eval(a), eval(b))
7. eval(compos(a, b)) = compos(eval(a), eval(b))

8. eval(if-then-else(a, b_1, b_2)) = if eval(a)
then eval(b_1)
else eval(b_2)

9. subst(var(y), a, x) = if eq-ident(x, y)
then a
else var(y)

10. subst(cte(v), a, x) = cte(v)

11. subst($a_1 + a_2$, a, x) = subst(a_1, a, x) + subst(a_2, a, x)

12. subst(union-cl (a_1, a_2), a, x) = union-cl(
subst(a_1, a, x), subst(a_2, a, x))

. . .

endoftype

ENDCLUSTER

18.

```
type SET-CL
instance of SET
     with
     Elem        as   class
rename set       as   set-cl
endoftype
```

19.

```
type SET-REL
instance of SET
     with
     Elem        as   rel
rename set       as   set-rel
endoftype
```

20.

```
type SET-FUN
instance of SET
     with
     Elem        as   fun
rename set       as   set-fun
endoftype
```

21.

type REDE

based on VALOR, EXPR

sort rede

operations

cria-rede: id x set-cl x set-rel x set-fun ⟶ rede

nome-r: rede ⟶ id

class-r: rede ⟶ set-cl

rel-r: rede ⟶ set-rel

fun-r: rede ⟶ set-fun

eq-rede: rede x rede ⟶ bool

≤-rede?: rede x rede ⟶ bool

union-r : rede x rede ⟶ rede

inter-r : rede x rede ⟶ rede

menos-r : rede x rede ⟶ rede

add-cl: rede x class ⟶ rede

del-cl: rede x class ⟶ rede

add-rel: rede x rel ⟶ rede

del-rel: rede x rel ⟶ rede

add-fun: rede x fun ⟶ rede

del-fun: rede x fun ⟶ rede

axioms ∀ $a \in$ set-cl ∀ $b, b_1, b_2 \ldots \in$ set-rel

∀ $i, i_1, \ldots, \in$ id ∀ $c, c_1, c_2 \in$ set-fun

∀ $r, r_1, \ldots \in$ rede ∀ $d \in$ class ∀ $e \in$ rel

∀ $f \in$ fun

1. cria-rede(nome-r(r), class-r(r),
   rel-r(r), fun-r(r)) = r
2. nome-r(cria-rede(i, a, b, c)) = i
3. class-r(cria-rede(i, a, b, c)) = b
4. add-cl(cria-rede(i, a, b, c), d) =
   cria-rede(i, ins-elem(d, a), b, c)
5. del-cl(cria-rede(i, a, b, c), d) =
   cria-rede(i, del-elem(d,a), b, c)
6. add-rel(cria-rede(i, a, b, c), e) =
   cria-rede(i, a),ins-elem(b, e), c)
7. del-rel(cria-rede(i, a, b, c), e) =
   cria-rede(i, a, del-elem(b, e), c)

```
8. add-fun(cria-rede(i, a, b, c), f) =
      cria-rede(i, a, b, ins-elem(c, f))
9. del-fun(cria-rede(i, a, b, c), f) =
      cria-rede(i, a, b, del-elem(c, f))
10.union-r(cria-rede(i1, a1, b1, c 1),
            cria-rede(i2, a2, b2, c2)) =
      cria-rede(str-id(concat(concat(id-str(i1),      "union"),
      id-str(i2))), a1 ∪ a2,
                  b1 ∪ b2, c1 ∪ c2)
11.inter-r(cria-rede(i1, a1, b1, c 1),
            cria-rede(i2, a2, b2, c2)) =
      cria-rede(str-id(concat(concat(id-str(i1),      "inter"),
      id-str(i2))), a1 ∩ a2,        b1 ∩ b2, c1 ∩ c2)
12.menos-r(cria-rede(i1, a1, b1, c 1),
            cria-rede(i2, a2, b2, c2)) =
      cria-rede(str-id(concat(concat(id-str(i1),
      "menos"), id-str(i2))), a1menos a2, b1menos b2,
      c1menos c2)
13.eq-rede(cria-rede(i1, a1, b1, c 1),
             cria-rede(i2, a2, b2, c2)) = eq-set(a1, a2)
           .and. eq-set(b1, b2) .and. eq-set(c1, c2)
14.≤-cl(cria-rede(i1, a1, b1, c 1),
             cria-rede(i2, a2, b2, c2)) = ≤-set(a1, a2)
           .and. ≤-set(b1, b2) .and. ≤-set(c1, c2)
endoftype
```

ANEXO C : SISTEMA DE INFERÊNCIA DE TIPOS PARA RECON-II

1. NULL

$$\frac{}{\texttt{Null} :: \textbf{TYPE}}$$

2. BOOL

$$\frac{}{\texttt{Bool} :: \textbf{TYPE}}$$

$$\frac{}{\texttt{true} : \texttt{Bool}} \qquad \frac{}{\texttt{false} : \texttt{Bool}}$$

$$\frac{a : \texttt{Bool} \quad c : \texttt{T} \quad d : \texttt{T}}{\texttt{if } a \texttt{ then } c \texttt{ else } d : \texttt{T}}$$

if *true* then *c* else *d* ⟶ *c*

if *false* then *c* else *d* ⟶ *d*

$$\frac{a : \texttt{Bool}}{\texttt{.not.}a : \texttt{Bool}}$$

.not. *true* ⟶ *false*

.not. *false* ⟶ *true*

$$\frac{a : \texttt{Bool} \quad b : \texttt{Bool}}{a \texttt{ .and.} b : \texttt{Bool}}$$

true.and. *true* ⟶ *true*

a .and. *false* ⟶ *false*

$$\frac{a : \texttt{Bool} \quad b : \texttt{Bool}}{a \texttt{ .or.} b : \texttt{Bool}}$$

true.or. *b* ⟶ *true*

false .or. *false* ⟶ *false*

3. CHAR

$$\frac{}{\text{Char} :: \textbf{TYPE}}$$

$$\frac{}{c : \text{Char}} \; ; \; c \text{ é Char}$$

4. STRING

$$\frac{}{\text{Str} :: \textbf{TYPE}}$$

$$\frac{}{\text{" "} : \text{Str}}$$

$$\frac{s : \text{Str} \quad a : \text{Char}}{\text{add-str}(s, a) : \text{Str}}$$

$$\frac{s : \text{Str}}{\text{prim-ch}(s) : \text{Char}}$$

$$\frac{s : \text{Str}}{\text{resto-str}(s) : \text{Str}}$$

$$\frac{s : \text{Str} \quad u : \text{Str}}{\text{concat}(s,u) : \text{Str}}$$

prim-ch("") $\longrightarrow \perp$

prim-ch(add-str(s, a)) $\longrightarrow a$

resto-str(" ") $\longrightarrow$ " "

resto-str(add(s, a)) $\longrightarrow s$

concat(s, "") $\longrightarrow s$

concat(s, add(u, a)) $\longrightarrow$ add(concat(s, u), a)

5.IDENT

$$\frac{}{\text{Id} :: \textbf{TYPE}}$$

$$\frac{a : \text{Str}}{a : \text{Id}} \; ; \; \text{prim-ch}(a) \text{ seja letra}$$

6. INT

$$\frac{}{\texttt{Int} :: \texttt{TYPE}}$$

$$\frac{}{\texttt{0} : \texttt{Int}}$$

$$\frac{a : \texttt{Int}}{\texttt{succ}(a) : \texttt{Int}} \qquad \frac{a : \texttt{Int}}{\texttt{pred}(a) : \texttt{Int}}$$

$\texttt{0} \longrightarrow \texttt{0}$

$\texttt{pred(0)} \longrightarrow \bot$

$\texttt{succ(0)} \longrightarrow \texttt{succ(0)}$

$\texttt{succ(pred}(a)\texttt{)} \longrightarrow a \quad \text{se } a \neq 0$

$\texttt{pred(succ}(a)\texttt{)} \longrightarrow a$

7.Regras Gerais

Suposição: $\pi \cup \{T_1 \leq T_2\} \vdash T_1 \leq T_2;$

Reflexividade: $\pi \vdash T \leq T;$

Transitividade: $$\frac{\pi_1 \vdash T_1 \leq T_2 \quad \pi'_2 \vdash T_2 \leq T_3}{\pi_1 \cup \pi_2 \vdash T_1 \leq T_3}$$

Instanciação: $\pi \vdash T_1[T_2/V] \leq \forall\, V.T_1$

Generalização: $$\frac{\pi \vdash T_1 \leq T_2}{\pi \vdash \forall\, V.T_1 \leq T_2}$$; V não livre em π ou T_2

Regra de Subtipagem $$\frac{\pi \vdash T_1 \leq T_2 \quad \pi \vdash a : T_1}{\pi \vdash a : T_2}$$

8. PRODUTO CARTESIANO

$$\frac{S :: \mathbf{TYPE} \quad T :: \mathbf{TYPE}}{S \times T :: \mathbf{TYPE}}$$

$$\frac{a : S \quad b : T}{(a, b) : S \times T}$$

$$\frac{p : S \times T}{\mathbf{prim\text{-}elem}(p) : S}$$

$$\frac{p : S \times T}{\mathbf{seg\text{-}elem}(p) : T}$$

$\mathbf{prim\text{-}elem}((a, b)) \longrightarrow a$

$\mathbf{seg\text{-}elem}((a, b)) \longrightarrow b$

$(\mathbf{prim\text{-}elem}(p), \mathbf{seg\text{-}elem}(p)) \longrightarrow p$

$$\frac{T_1 \leq T_2 \quad S_1 \leq S_2}{T_1 \times S_1 \leq T_2 \times S_2}$$

9. LIST

$$\frac{T :: \mathbf{TYPE}}{\mathbf{List}[T] :: \mathbf{TYPE}}$$

$$\frac{}{\mathbf{nil} : \mathbf{List}[T]}$$

$$\frac{a : \mathbf{List}[T] \quad b : T}{a \mid b : \mathbf{List}[T]}$$

$$\frac{a : \mathbf{List}[T]}{\mathbf{head}(a) : T}$$

$$\frac{a : \mathbf{List}[T]}{\mathbf{tail}(a) : \mathbf{List}[T]}$$

$\mathbf{head}(a \mid b) \longrightarrow a$

$\mathbf{tail}(a \mid b) \longrightarrow b$

$\mathbf{head}(a) \mid \mathbf{tail}(a) \longrightarrow a$

$$\frac{T_1 \leq T_2}{\mathbf{List}[T_1] \leq \mathbf{List}[T_2]}$$

10. SET

$$\frac{T :: \textbf{TYPE}}{Set[T] :: \textbf{TYPE}} \qquad \frac{}{\{\} : Set[T]}$$

$$\frac{\{a_1, a_2, \ldots, a_n\} : Set[T] \quad b : T}{\{a_1, a_2, \ldots, a_n, b\} : Set[T]}$$

$$\frac{\{a_1, a_2, \ldots, a_n\} : Set[T]}{\text{choose-elem}(\{a_1, a_2, \ldots, a_n\}) : T}$$

$$\frac{a_i : T \quad \{a_1, a_2, \ldots, a_n\} : Set[T]}{\text{del-elem}(\{a_1, a_2, \ldots, a_n\}, a_i) : Set[T]}$$

$$\frac{a : Set[T] \quad b : Set[T]}{a \cup b : Set[T]}$$

$$\frac{a : Set[T] \quad b : Set[T]}{a \cap b : Set[T]}$$

$$\frac{a : Set[T] \quad b : Set[T]}{a \text{ menos } b : Set[T]}$$

$$\frac{T_1 \leq T_2}{Set[T_1] \leq Set[T_2]}$$

$\text{del-elem}(\{a_1, a_2, \ldots, a_n\}, a_i) \longrightarrow \{a_2, \ldots, a_n\}$

$\text{choose-elem}(\{a_1, a_2, \ldots, a_n\}) \longrightarrow a_i$; para algum $i = 1 \ldots n$.

$\{a_1, a_2, \ldots, a_n\} \cup \{b_1, b_2, \ldots, b_k\} \longrightarrow \{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_k\}$

$\{a_1, a_2, \ldots, a_n\} \cap \{b_1, b_2, \ldots, b_k\} \longrightarrow \{a_1, a_2, \ldots, a_s\}$

sss $a_i = b_j$ $i = 1 \ldots n$ e $j = 1 \ldots k$, com $s <= \max(n, k)$.

$\{a_1, a_2, \ldots, a_n\} \text{ menos } \{b_1, b_2, \ldots, b_k\} \longrightarrow \{a_1, a_2, \ldots, a_s\}$

sss $a_i \notin \{b_1, b_2, \ldots, b_k\}$ para $i = 1 \ldots n$, com $s <= n$

11. FUN

$$\frac{T_1 :: \mathbf{TYPE} \quad T_2 :: \mathbf{TYPE}}{T_1 \longrightarrow T_2 :: \mathbf{TYPE}}$$

$$\frac{x : T_1 \quad e : T_2}{\mathbf{fun}(x : T_1).\ e : T_1 \longrightarrow T_2}$$

$$\frac{f : T_1 \longrightarrow T_2 \quad e : T_1}{f\,(e) : T_2}$$

$$\frac{f : T_1 \longrightarrow S \quad g : S \longrightarrow T_2}{f \odot g : T_1 \longrightarrow T_2}$$

$$(\mathbf{fun}(x : T_1).e_1)(e_2) \longrightarrow e_1[e_2/x]$$

$$\frac{S_1 \leq T_1 \quad T_2 \leq S_2}{T_1 \longrightarrow T_2 \leq S_1 \longrightarrow S_2}$$

$$(\mathbf{fun}(x : T_1).e_1) \odot (\mathbf{fun}(y : S).e_2)(e_3) \longrightarrow e_2[e_1[e_3/x]/y]$$

12. SIGMA

$$\frac{T_1, \ldots, T_n :: \mathbf{TYPE} \quad T :: \mathbf{TYPE} \Rightarrow i_1(T) : T_1 \quad \ldots \quad T :: \mathbf{TYPE} \Rightarrow i_n(T) : T_n}{(\ \Sigma x \in \mathrm{Id}.\ i_1 : x \longrightarrow T_1\ ,\ \ldots\ ,\ i_n : x \longrightarrow T_n\ ) :: \mathbf{TYPE}}$$

onde $\Rightarrow$ significa implicação lógica.

$$\frac{a : \mathrm{Id} \quad i_1(a) = v_1 : T_1 \quad \ldots \quad i_n(a) = v_n : T_n}{\underline{\text{conceito}}\ a ::= (i_1 = v_1\ ;\ \ldots\ ;\ i_n = v_n) : (\Sigma x.\ i_1 : x \longrightarrow T_1, \ldots, i_n : x \longrightarrow T_n)}$$

$$\frac{a : (\Sigma x.\ i_1 : x \longrightarrow T_1, \ldots, i_n : x \longrightarrow T_n)}{\mathbf{nome\text{-}conceito}(\ a\ ) : \mathrm{Id}}$$

$$\frac{a : (\Sigma x.\, i_1 : x \longrightarrow T_1, \ldots, i_n : x \longrightarrow T_n)}{a.i_r : T_r} \; 0<r<n+1$$

$$\frac{a : (\Sigma x.\, i_1 : x \longrightarrow T_1, \ldots, i_n : x \longrightarrow T_n)}{\text{atributos}(a) : \text{Set}(\text{Par}[\text{Id}, T])}$$

$$\frac{b:\text{Id}\quad a:(\Sigma x.\, i_1 : x \to T_1 \ldots i_n : x \to T_n)\quad i_p(b) = v_p : T'_p \ldots i_{p+k}(b) = v_{p+k} : T'_{p+k}}{\underline{\text{conceito}}\ b ::= \underline{\text{instância}}\ (a,\ i_p = v_p \ldots i_{p+k} = v_{p+k}) : (\Sigma x \in \text{Id}.\ i_1 : x \to T_1 \ldots i_p : x \to T'_p \ldots i_{p+k} : x \to T'_{p+k} \ldots i_n : x \to T_n)}$$

Onde $1 \le p \le p+k \le n$; $T'_i \le T_i$ para $p \le i \le p+k$;

$$\frac{b:\text{Id}\quad a:(\Sigma x.\, i_1 : x \to T_1 \ldots i_n : x \to T_n)\quad i_{n+1}(b) = v_{n+1} : T_{n+1} \ldots i_{n+k}(b) = v_{n+k} : T_{n+k}}{\underline{\text{conceito}}\ b ::= \underline{\text{subconceito}}(a,\ i_{n+1} = v_{n+1} \ldots i_{n+k} = v_{n+k}) : (\Sigma x \in \text{Id}.\ i_1 : x \to T_1 \ldots i_{n+k} : x \to T_{n+k})}$$

$$\text{nome-conceito}(\underline{\text{conceito}}\ a ::= (i_1 = v_1;\ \ldots;\ i_n = v_n)) \longrightarrow a$$

$$\text{atributos}(\underline{\text{conceito}}\ a ::= (i_1 = v_1;\ \ldots; i_n = v_n)) \longrightarrow \{(i_1, v_1); \ldots; (i_n, v_n)\}$$

$$(\underline{\text{conceito}}\ a ::= (i_1 = v_1;\ \ldots;\ i_n = v_n)).i_k \longrightarrow v_k$$

13. PRED

Pred :: **TYPE**

reflexiva : Pred

simetrica : Pred

transitiva : Pred

14. REL

$$\frac{\text{Set[Pred]::TYPE} \quad E_{11} \times E_{12} :: \text{TYPE} \quad \ldots \quad E_{n1} \times E_{n2} :: \text{TYPE}}{\text{Rel(Set[Pred]; } (E_{11} \times E_{12}, \ldots, E_{n1} \times E_{n2})) :: \text{TYPE}}$$

$$\frac{r\text{:Id} \quad p\text{:Set[Pred]} \quad (a_1, b_1) : E_{11} \times E_{12} \ \ldots \ (a_{n1}, b_{n2}) : E_{n1} \times E_{n2}}{\begin{array}{l}\underline{\text{relação}}\ r ::= \ \{\ \underline{\text{propriedades}} = \langle p \rangle \\ \qquad\qquad \underline{\text{elementos}} = \langle\ (a_1, b_1)\ ; \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad (a_n, b_n) \rangle \\ \qquad\qquad \rangle\ :\ \text{Rel(Set[Pred], } (E_{11} \times E_{12} \ \ldots \ E_{n1} \times E_{n2}))\end{array}}$$

$$\frac{r\ :\text{Rel(Set[Pred], } (E_{11} \times E_{12} \ldots E_{n1} \times E_{n2}))}{\text{nome-rel(} r \text{) : Id}}$$

$$\frac{r\ :\text{Rel(Set[Pred], } (E_{11} \times E_{12} \ldots E_{n1} \times E_{n2}))}{\text{pares-rel(} r \text{) : Set[E x E]}}$$

$$\frac{r\ :\text{Rel(Set[Pred], } (E_{11} \times E_{12} \ldots E_{n1} \times E_{n2}))}{\text{pred-rel(} r \text{) : Set[Pred]}}$$

$$\underline{\text{inversa}}\ :\text{Rel(Set[Pred], } (E_{11} \times E_{12} \ \ldots \ E_{n1} \times E_{n2})) \longrightarrow \text{Rel(Set[Pred], } (E_{12} \times E_{11} \ldots E_{n2} \times E_{n1})) :$$

$\underline{\text{relação}}\ r ::= \{\underline{\text{propriedades}} = \langle p \rangle ; \underline{\text{elementos}} = \langle (a_1, b_1) \ldots (a_n, b_n) \rangle\}$
$\longrightarrow$ $\underline{\text{relação}}\ \underline{\text{inversa}}\ (r) ::= \{\underline{\text{propriedades}} = \langle p \rangle ; \underline{\text{elementos}} = \langle (b_1, a_1) \ \ldots \ (b_n, a_n) \rangle\}$

Composição :

composição :$Rel(E_{11} \times E_{12} \ldots E_{n1} \times E_{n2})^* \times Rel(E_{12} \times E'_{11} \ldots E_{k2} \times E'_{k1})^*$

$\longrightarrow Rel(E_{11} \times E'_{11} \ldots E_{k1} \times E'_{k1})$:

relação $r ::= \{(a_1, b_1) \ldots (a_n, b_n)\}$; relação $r' ::= \{(b_1, a'_1) \ldots (b_k, a'_k)\}$

$\longrightarrow$ relação composição $(r \bullet r') ::= \{(a_1, a'_1) \ldots (a_k, a'_k)\}$; $k \leq n$

Note que $r : Rel(E_{11} \times E_{12} \ldots E_{n1} \times E_{n2})^*$ denota o fecho de r, que é obtido a partir das propriedades da relação aplicadas ao conjunto de pares da mesma.

$$\frac{E_{11} \times E_{12} \leq E'_{11} \times E'_{12} \ldots E_{k1} \times E_{k2} \leq E'_{k1} \times E'_{k2}}{Rel(E_{11} \times E_{12} \ldots E_{n1} \times E_{n2})^* \leq Rel(E_{11} \times E_{12} \ldots E_{n1} \times E_{n2})^*} ; k \leq n$$

nome-rel(relação r ::= { propriedades = <p>

elementos = < (a_1, b_1) ;
⋮
(a_n, b_n)>>$\longrightarrow r$

pares-rel(relação r ::= { propriedades = <p>

elementos = < (a_1, b_1) ;
⋮
(a_n, b_n)>>$\longrightarrow$ {(a_1, b_1)
⋮
(a_n, b_n)}

pred-rel(relação r ::= { propriedades = <p>

elementos = < (a_1, b_1) ;
⋮
(a_n, b_n)>>$\longrightarrow$ p

15. REDE

$$\frac{\begin{array}{l} E_1 :: \mathbf{TYPE} \ \ldots \ E_n :: \mathbf{TYPE} \\ i_1 : E_1 \times \ldots \times E_n \longrightarrow E_{j1} \qquad [1 \le j1 \le n] \\ \vdots \\ i_m : E_1 \times \ldots \times E_n \longrightarrow E_{jm} \qquad [1 \le jm \le n] \\ R_1 \le E_1 \times \ldots \times E_n \\ \vdots \\ R_p \le E_1 \times \ldots \times E_n \end{array}}{(\Omega x.\ i_1 : x \longrightarrow E_{j1}, \ \ldots, \ i_m : x \longrightarrow E_{jm}, \ R_1 \le x, \ \ldots, \ R_p \le x) :: \mathbf{TYPE}}$$

$$\frac{b : \mathrm{id} \quad a_1 : E_1 \ldots a_n : E_n \quad f_1 : S_1 \longrightarrow T_1 \ldots f_m : S_m \longrightarrow T_m \quad r_1 : R_1 \ldots r_p : R_p}{\begin{array}{l} \underline{\text{rede}}\ i ::= \langle \langle a_1, \ldots, a_n \rangle ; \langle f_1, \ldots, f_m \rangle ; \langle r_1, \ldots, r_p \rangle \rangle \\ \underline{\text{fim-rede}} \ : (\Omega x. f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \ , \ R_1 \le x, \ldots, R_p \le x) \end{array}}$$

Onde $S_i = E_1 \times \ldots \times E_n$, $T_i = E_j$ $(1 \le j \le n)$ e R_i é um tipo REL como foi definido anteriormente.

$$\frac{b : (\Omega x. f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \ , \ R_1 \le x, \ldots, R_p \le x)}{\underline{\text{nome-rede}}\ (\ b\ ) : \mathrm{id}}$$

$$\frac{b : (\Omega x. f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \ , \ R_1 \le x, \ldots, R_p \le x)}{\underline{\text{conceitos-rede}}\ (\ b\ ) : \mathrm{Set}\ [E]}$$

$$\frac{b : (\Omega x. f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \ , \ R_1 \le x, \ldots, R_p \le x)}{\underline{\text{funções-rede}}\ (\ b\ ) : \mathrm{Set}[S \longrightarrow T]}$$

$$\frac{b \;:\; (\Omega x.\, f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \;,\; R_1 \leq x, \ldots, R_p \leq x)}{\underline{\text{relações-rede}}\ (\ b\ ) \;:\; \text{Set}[R]}$$

$$\frac{E_1 \leq E'_1 \ldots E_n \leq E'_n \quad S_1 \longrightarrow T_1 \leq S'_1 \longrightarrow T'_1 \ldots S_m \longrightarrow T_m \leq S'_m \longrightarrow T'_m \quad R_1 \leq R'_1 \ldots R_p \leq R'_p}{\begin{array}{l}(\Omega x.\, f_1 : x \longrightarrow T_1 \ldots f_m : x \longrightarrow T_m \;,\; R_1 \leq x, \ldots, R_p \leq x) \;\leq \\ (\Omega y.\, f'_1 : y \longrightarrow T'_1 \ldots f'_m : y \longrightarrow T'_m \;,\; R'_1 \leq y, \ldots, R'_p \leq y)\end{array}}$$

$\underline{\text{n ome-rede}}$ $(\underline{\text{rede}}\ i ::= \{ \langle a_1, \ldots, a_n \rangle ;$
$\langle f_1, \ldots, f_k \rangle ; \langle r_1, \ldots, r_s \rangle \}\underline{\text{fim-rede}} \longrightarrow i$

$\underline{\text{conceitos-rede}}$ $(\underline{\text{rede}}\ i ::= \{ \langle a_1, \ldots, a_n \rangle ;$
$\langle f_1, \ldots, f_k \rangle ; \langle r_1, \ldots, r_s \rangle \}\underline{\text{fim-rede}} \longrightarrow \langle a_1, \ldots, a_n \rangle$

$\underline{\text{f unções-rede}}$ $(\underline{\text{rede}}\ i ::= \{ \langle a_1, \ldots, a_n \rangle ;$
$\langle f_1, \ldots, f_k \rangle ; \langle r_1, \ldots, r_s \rangle \}\underline{\text{fim-rede}} \longrightarrow \langle f_1, \ldots, f_k \rangle$

$\underline{\text{r elações-rede}}$ $(\underline{\text{rede}}\ i ::= \{ \langle a_1, \ldots, a_n \rangle ;$
$\langle f_1, \ldots, f_k \rangle ; \langle r_1, \ldots, r_s \rangle \}\underline{\text{fim-rede}} \longrightarrow \langle r_1, \ldots, r_s \rangle$

BIBLIOGRAFIA

[AHO 86] AHO, A.V.; SETHI, R. ; ULLMAN, J.D. Compilers: Principles, Techniques and Tools. Reading: Addison-Wesley, 1986. Chap. 6. p 379-428.

[ALB 83] ALBANO, Antonio. Type Hierarchies and Semantic Data Models. ACM SIGPLAN Notices, California, v. 18, n. 6, p. 178-185, June 1983. Trabalho apresentado no SIGPLAN'83 Symposium on Programming Language issues in Software Systems.

[ALB 85] ALBANO, A.; CARDELLI, L. ; ORSINI R. GALILEO : A Strongly-Typed, Interactive Conceptual Language. ACM Transactions on Database Systems, New York, v. 10, n. 2, p. 230-260, June 1985.

[ALB 88] ALBANO, A. et al. The Type System of Galileo. In: Data Types and Persistence. Berlin : Springer-Verlag, 1988. 286p. p. 101-119.

[ATK 87] ATKINSON, Malcolm P.; BUNEMAN, O. Types and Persistence in Database Programming Languages. ACM Computing Surveys, v. 19, n. 1, June 1987.

[ATZ 88] ATZENI,P ; PARKER, S. Set Containment Inference and Sylogisms. Theoretical Computer Science, North-Holland, v. 62, p. 39-65, 1988.

[BAC 89] BACKHOUSE, R.; CHISHOLM, P. et al. Do-it-Yourself Type Theory. Formal Aspects Computing, v.11, n.11, p. 19-84, Jan-Mar 1989.

[BAI 87] BAILES, P. A. G : A Functional Language With Generic Abstract Data Types. Computer Language, Great Britain, v. 12, n. 2, p. 69-94, 1987.

[BAR 80] BARBUTI, R.; MARTELLI, A. Static Type Checking for Languages with Parametric Types and Polymorphic Procedures. In: COLLOQUE INTERNATIONAL SUR LA PROGRAMATION, 4., 22-24 April, 1980, Paris. Proceedings... Berlin : Springer-Verlag, 1980. p. 1-16.

[BAR 86] BARR. A. ; FEIGENBAUM, E. A. The Handbookof Artificial Intelligence I. Reading : Addison-Wesley, 1986. v. 1. Chap. 3, p. 141-222.

[BER 79] BERRY, D.M. ; SCHWARTZ, R.L. Type Equivalence in Strongly Typed Languages : One More Look. ACM SIGPLAN Notices, v. 14, n. 9, p. 35-40, Sept. 1979.

[BER 80] BERT, D. Types algébriques et sémantiques des languages de programmation. In : COLLOQUE INTERNATIONAL SUR LA PROGRAMATION, 4., 22-24 April, 1980, Paris. Proceedings... Berlin : Springer-Verlag, 1980. p. 30-43.

[BER 80a] BERTONI, A.; MAURI, G.; MIGLIOLI, P. Towards a Theory of Abstract Data Types: a Discussion on Problems and Tools. Colloque International sur la Programation, 4., 22-24 April, 1980, Paris. Proceedings... Berlin : Springer-Verlag, 1980. p. 44-58.

[BER 81] BERGSTRA, J., HEERING, J., KLINT, P. Algebraic Specification. New York : ACM Press, 1989. 397 p.

[BER 81a] BERGSTRA, J., BROY, M., TUCKER, J. ; WIRSING, M. On the power of algebraic specifications. In: SYMPOSIUM ON MATHEMATICAL FOUNDATIONS OF COMPUTER SCIENCE, 10, Aug. 31-Sept. 4, 1981, Strbskepleso. Czechoslovakia. Proceedings... Berlin : Springer-Verlag 1981. p. 193-204. (Lecture Notes in Computer Science, 118).

[BOB 77] BOBROW, D. G.; WINOGRAD, D. T. An Overview of KRL, a Knowledge Representation Language. Cognitive Science, v.1, n. 1, p. 3-46, Jan. 1977.

[BRA 83] BRACHMAN, R. J. What is-a is and isn't : An Analysis of Taxonomic Links in Semantic Networks. IEEE Computer, v. 16, n. 10, p. 30-36, Oct. 1983.

[BRA 91] BRAGA, J. L. ; CARVALHO, R. L. de. EPISTEME - Aquisição e Estruturação Semi-Automática de Conhecimento. In : SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 8., Jul. 18-21, 1991, Brasilia, DF, Brasil. Anais... Brasilia: SBC, 1991 304 p. p. 275-285.

[BRO 79] BROY, M. ; alii. Existencial Quantifiers in Abstract Data Types. In : COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING, 6., Jul. 16-20, 1979, GRAZ, AT. Proceedings... Berlin : Springer-Verlag, 1979. 684 p. p. 73-87. (Lecture Notes in Computer Science, 71).

[BRO 82] BROY, M.; WIRSING, M. Partial Abstract Types. Acta Informatica, v. 18, n. 1, p. 47-64, 1982.

[BRO 87] BROY, ; PEPPER, P. On the algebraic definition of Programming Languages. ACM Transactions on Programming Languages and Systems, New York, v. 9, n. 1, p. 54-99, Jan. 1987.

[BUR 84] BURSTALL, R.; LAMPSON, B. A Kernel Language for Abstract Types Modules. In: Semantics of Data Types. Berlin : Springer-Verlag, 1984. p. 1-50 (Lecture Notes in Computer Science, 173)

[CAR 81] CARBONELL J. G. Default Reasoning and Inheritance Mechanisms on Type Hierarchies. ACM Sigplan Notices, New York, v. 16, n. 1, p. 107-109, Jan. 1981.

[CAR 84] CARDELLI, Luca. A Semantics of Multiple Inheritance. In : Semantics of Data Types. Berlin :Springer-Verlag, 1984. p. 51-67. (Lecture Notes in Computer Science, 173)

[CAR 85] CARDELLI, Luca ; WEGNER, Peter. On Uderstanding Types, Data Abstraction, and Polimorphism. ACM Computing Surveys, New York, v. 17, n. 4, p. 471-522, Dec. 1985.

[CAR 87] CARDELLI, L. Basic Polymorphic Typechecking. Science of Computer Progamming, North-Holland, v. 8, n. 2, p. 157-172, April 1987.

[CAR 88] CARDELLI, Luca ; MACQUEEN, David. Persistence and Type Abstraction. In : Data Types and Persistence. Berlin: Springer-Verlag, 1988. 286p. Chap. 3, p. 31-41.

[CAR 89] CARDELLI, Luca. Typeful Programming. In: IFIP ADVANCE SEMINAR ON FORMAL DESCRIPTION OF PROGRAMMING Concepts, Feb. 14, 1989, Rio de Janeiro, Brasil. Anais... Rio de Janeiro : IFIP, 1989. 321p. p. 4-57.

[CAR 91] CARNOTA, R. La conexión entre lógica no monotona y lógica condicionais. In : SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 8., Jul. 18-21. 1991, Brasilia, DF. Anais... Brasilia : SBC, 1991. 304p. p. 237-243.

[CAS 90] CASA, M. Estudo das Noções de Herança e Reconhecimento em Sistemas de Representação de Conhecimento. Porto Alegre : CPGCC da UFRGS, Mar. 1990, 118p. (Trabalho Individual 173).

[CHA 78] CHARNIAK, E. On the Use of Framed Knowledge in Language Comprehension. Artificial Intelligence, v. 11, p. 225-265, 1978.

[COP 83] COPPO, M. On the Semantics of Polymorphism. Acta Informatica, Springer-Verlag, v. 20, n. 2, 1983.

[COW 86] COWLING, A. J. Type Checking in Polymorphic Languages. The Computer Journal, v. 29, n. 6, p. 538-544, 1986.

[COS 90] COSTA, Antonio C. da.Rocha. A Noção de Tipo e a Modelagem Semântica de Linguagens de Programação. Porto Alegre : CPGCC da UFRGS, 1990. Tópicos Especiais em Computação III : Teoria dos Tipos, CMPP84. Notas de Aula.

[DAN 88] DANFORTH Scott ; TOMLINSON Chris. Type Theories and Object-Oriented Programming. ACM Computing Surveys, New York, v. 20, n.1, p. 29-72. Mar 1988.

[DEL 86] DELGRANDE, J. P. ; MYLOPOULOS, J. Knowledge Representation : Features of Knowledge. In: Fundamentals of Artificial Intelligence : An Advanced Course. Berlin : Springer-Verlag, 1986. 313 p. (Lecture Notes in Computer Science, 232).

[EHR 80] EHRIG, H.; et al. Algebraic Implementetion of Abstract Data Types: concept, syntax, semantics and correctness. In: COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING, 7., Jul. 14-18, 1980, Noordwijkerhout, The Netherlands. Proceedings... Berlin : Springer-Verlag, 1980. p. 142-156.

[EHR 80a] EHRIG, H.; et al. Parameterized Data Types in Algebraic Specification Languages. In : COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING, 7., Jul. 14-18, 1980, Noordwijkerhout, The Netherlands. Proceedings... Berlin : Springer-Verlag, 1980. p. 157-168.

[EHR 81] EHRIG, H., Algebraic Theory of Parameterized Specifications with Requirements. In : CAAP '81 : Trees in Algebra and Programming 6th Colloquium, Genoa, Mar. 1981. Proceedings. Berlin : Springer -Verlag, 1981. (Lecture Notes in Computer Science, 112), p. 1-24.

[FAI 88] FAIRBAIRN, J. A New Type-Checker for a Functional Language. In : Data Types and Persistence. Berlin: Springer-Verlag, 1988. 286p. p. 69-87.

[FIK 85] FIKES, R. KEHLER, T. The Role of Framed-Based Representation in Reasoning. Communications of the ACM, New York, v. 28, n. 9, p. 904-920, Sept. 1985.

[FUH 90] FUH, Y.; MISHRA, P. Type Inference with Subtypes. Theoretical Computer Science, North-Holland, v. 73, p. 155-175, 1990.

[GEN 86] GENESERETH, M. R. ; NILSSON, N. J. Logical of Artificial Intelligence. Los Altos, Callifornia : Morgan Kaufman, 1986.

[GEN 87] GENESERETH, M. R. ; GINSBERG, M. L. Logic Programming. Communications of the ACM, New York, v. 28, n. 9, p. 933-941, Sept. 1987.

[GHE 82] GHEZZI, Carlo ; JAZAYERI, Mehdi. Conceitos de Linguagens de Progamação. Rio de Janeiro : Editora Campus, 1982. 306p.

[GIR 71] GIRARD, Y. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'elimination des coupures dans l'analyse et la théorie des types. In : SCANDINAVIAN LOGIC SYMPOSIUM, 2. , 1971 Proceedings... North-Holland : J. E. Fenstad, 1971. p. 63-92.

[GIR 89] GIRARD, Y. et al. Proofs and Types. Cambridge : Cambridge University Press, 1989. 176 p.

[GLU 91] GLUZ. J. C. Formalização de um Modelo de Representação de Conhecimentos Orientado a Objetos. Porto Alegre : CPGCC da UFRGS, 1991. 214 p. (Dissertação de Mestrado).

[GOG 77] GOGUEN, J. Abstract errors for abstract data types. In: UCLA SEMANTICS AND THEORY OF COMPUTATION REPORT 46.; IFIP WORKING CONFERENCE FORMAL DESCRIPTION OF PROGRAMMING LANGUAGES CONCEPTS, Aug. 01-05, 1977, St. Andrews, Canada. Proceedings... Amsterdam : North-Holland, 1978.

[GOG 77a] GOGUEN, J. A.; et al. Initial Semantics and Continuous Algebras. Journal of the Association for Computing Machinery, New York, v. 24, n. 1, p. 68-95, Jan 1977.

[GOG 78] GOGUEN, J. A.; THATCHER, J. W.; WAGNER, E.G. Data Structures. In : Currents Trends in Programming Methodology. New Jersey: Prentice-Hall, 1978. v. 4 : Data Structuring. 321 p. p. 80-149.

[GUT 78] GUTTAG, J. ; HORNING, J. The Algebraic Specification of Abstract Data Types. Acta Informatica, Berlin, v. 10., n. 1, p. 27-52, 1978.

[GUT 78a] GUTTAG, J. et al. Data Structures. In : Currents Trends in Programming Methodology. New Jersey : Prentice-Hall, 1978. v. 4 : Data Structuring. 321 p. p. 60- 79.

[GRA 68] GRÄTZER, G. Universal Algebra. New York : Springer-Verlag, 1968. 581 p.

[HAL 87] HALBERT, D. C. ; PATRICK, O. D. Using Types and Inheritance in O.O. Languages. IEEE Software, p. 71-79, Sept. 1987.

[HAY 85] HAYES-ROTH, F. Rule-Based Systems. Communications of the ACM, New York, v. 28, n. 9, p. 921-932. Sept. 1985.

[HOA 72] HOARE, C.A.R. Notes on Data Structuring. In : DAHL, O. J. ; DIJSKTRA, E. W.; HOARE, C. A. R. Structured Programming. London: Academic Press, 1972. p. 83-174.

[HOA 78] HOARE, C. A. R. Data Structures. In : Currents Trends in Programming Methodology. New Jersey: Prentice-Hall, 1978. v. 4 : Data Structuring. 321 p. p. 1-11.

[HOR 89] HOREBEEK, I.V.; LEWI, J. Algebraic Specifications in Software Engineering : An Introduction. Berlin : Springer-Verlag, 1989. 350 p.

[ISR 83] ISRAEL, D. J. The Role of Logicin Knowledge Representation. IEEE Computer, v. 16, n. 10, p. 37-41, Oct. 1983.

[KRA 87] KRAMER ,B. M. ; MYLOPOULOS, J. Representation Knowledege. In : Encyclopedia of Artificial Intelligence. New York: John Wiley, 1987. v. 2, p. 882-890.

[LIP 81] LIPSON, J. D. Elements of Algebra and Algebrais Computing. Menlo Park : Benjamin/Cummings, 1981. 342 p.

[MAI 87] MAIDA, A. S. Frame Theory. In : Encyclopedia of Artificial Intelligence. New York : John Wiley, 1987. v. 1. p. 302-312.

[MAJ 79] MAJSTER, M. E. Data Types, Abstract Data Types and their Specification Probiem. Theoretical Computer Science, v. 8, n. 1, p. 89-128, 1979.

[McC 79] McCARTHY. First-Order Theories of Individual Concepts and Propositions. In :MCHIE Expert Systems in the Micro-Electronic Age. Edinburgh : Edinburgh University Press, 1979. 287 p.

[McC 83] McCALLA, G. ; CERCONE, N. Guest Editors' Introduction : Approaches to Knowledge Representation. IEEE Computer, v. 16, n. 10, p. 12-18. Oct., 1983.

[MAT 89] MATTOS, N. M. An Approach to Knowledge Base Management. Kaiserslautern : Universitat Kaiserslautern, 1989. 255 p. (Ph.D. Thesis).

[MIN 68] MINSKY, M. Semantic Information Processing. Cambridge : MIT Press, 1968. 440 p.

[MIN 75] MINSKY, M. A Framework for Representing Knowledge. In : Psychology of Computer Vision. New York : McGraw-Hill, 1975. p. 211-277.

[MIL 78] MILNER, R. A Theory of Type Polymorphism in Programming. Journal of Computer Systems Science v. 17, n. 3, p. 348-375, Dec. 1978.

[MIT 88] MITCHELL,, J. C.; PLOTKIN G. D. Abstract Types have Existential Types. ACM Transactions on Programming Languages and Systems, New York, v. 10, n. 3, p. 470-502, Jul. 1988.

[MIT 84] MITCHELL, J. C. Type Inference and Type Containment. In: Semantics of Data Types. Berlin: Springer-Verlag, Jun. 1984. p. 231-252. (Lecture Notes in Computer Science, 173.).

[NEW 72] NEWELL A.; SIMON H. Human problems solving. Englewod Cliffs: Prentice-Hall, 1972.

[NIL 80] NILSSON, N. Principles of Artificial Intelligence. Berlin: Springer-Verlag, 1980. Chap. 9 : Structured Object Representations. p. 361-414. 476 p.

[OLI 90] OLIVEIRA, F.M. Controle da Aprendizagem no Nivel do Meta-conhecimento. Porto Alegre : CPGCC da UFRGS, 1990. 53p. (Relatório de Pesquisa, 124).

[OLI 91] OLIVEIRA, F.M. ; VICCARI, R. M. Representação de Modelos Cognitivos em RECON-II. In : SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 8., 18-21 Jul. 1991 Brasilia, DF. Anais... Brasilia :SBC, 1991. 304 p. p. 267-274.

[OLI 92] OLIVEIRA, F. M. Descrição de uma Linguagem de Redes Conceituais. Porto Alegre : CPGCC da UFRGS, 1992. 61 p. (Trabalho Individual, 246).

[PAL 89] PALAZZO, L. A. M. RHESUS : Um modelo experimental para representação de conhecimento. Porto Alegre : CPGCC da UFRGS, 1989. Cap. 1. p. 9-30. (Trabalho Individual, 1406).

[PAS 90] PASSERINO, L.M. Tipos em Linguagens de Programacão. Porto Alegre : CPGCC da UFRGS, 1990. (Trabalho Individual, 174). 139 p.

[PEQ 91] PEQUENO , T. To Think is to Resolve Conflicts : Logical Reasoning with Incompletness and Contradiction. In : SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 8., 18-21 Jul. 1991, Brasilia, DF, Anais... Brasília : SBC, 1991. 304 p. p. 245-250.

[PEQ 91a] PEQUENO, M. Three Principles for the Formalization of Nonmonotonic Reasoning. In : SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 8., 18-21 Jul. 1991, Brasilia, DF, Anais... Brasília : SBC, 1991. 304 p. p. 251-257.

[QUI 68] QUILLIAN, R. M. Semantic Memory. In : MINSKY, M. [ed] Semantic Information Processing. Cambridge: MIT Press, 1968. 440 p.

[REY 78] REYNOLDS, J. C. Towards a Theory of Type Structure. In: Programming Symposium, Paris, Apr. 1978. p. 408-425. (Lecture Notes in Computer Science, 19).

[REY 85] REYNOLDS, J. C. Three Approaches to Type Structure. In: Mathematical Foundations of Software, v. 1. In : INTERNATIONAL JOINT CONFERENCE THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, TAPSOFT, Berlin : Springer-Verlag, 1985. p. 97-138. (Lecture Notes in Computer Science, 185).

[ROB 65] ROBINSON, J. A. A Machine Oriented Logic Based on the Resolution Principle. Journal of ACM, New York, v. 12, Jan. 1965. p. 23-41.

[ROW 81] ROWE, L. A. Data Abstraction from a Programming Language Viewpoint. ACM Sigplan Notices, New York, v. 16. n. 1, p. 29-35, Jan. 1981.

[RUS 87] RUS, T. An algebraic model for programming languages. Computer Language, v. 12, n. 3/4, p. 173-195, 1987.

[SET 83] SETTE, A. J. S. Sobre a Noção Algébrica de Implementação de Tipos Abstratos de Dados. Recife : Universidade Federal Pernambuco, Depto. de Informática, Centro de Cs. Exatas e da Natureza, 1983. 71 p. (Relatório Interno).

[SET 88] SETTE, A. J. S. Tipos Abstratos de Dados : uma visão algébrica. In : CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 8., 17-22 Jul. 1988, Rio de Janeiro. Rio de Janeiro : UFRJ/NCE, 1988. 39 p.

[SHA 89] SHASTRI, L. Default Reasoning in Semantic Networks : A Formalizations of Recognition and Inheritance. Artificial Intelligence. v. 39, n. 3, p. 283-355, 1989.

[SOW 87] SOWA, J. Semantics Networks. In : Encyclopedia of Artificial Intelligence. New York : John Wiley. 1987. v. 2. p. 1011-1024.

[STO 61] STOLL, R. Set Theory and Logic. New York : W. H. Freeman, 1961. 474 p.

[TAK 90] TAKAHASHI, T. ; LIESENBERG, H. K. E. Programação Orientada a Objeto : Uma visão integrada do paradigma de objetos. São Paulo : IME, 1990. Cap. 1-2. 335 p. p. 1-141.

[TOU 87] TOURETZKY, D. S. Inheritance Hierarchy. In: Encyclopedia of Artificial Intelligence. New York : John Wiley, 1987. v. 1. p. 422-431.

[VEL 79] VELOSO, P. A. S.; PEQUENO, T. H. C. D'ont write more axioms then you have to : A Methodology for the Complete and Correct Specification of Abstract Data Types; with examples. Rio de Janeiro : PUCRJ, Depto de Informática, 1979. (Monografia em Ciência da Computação n. 10/79).

[VEL 86] VELOSO, P. TIPOS ABSTRATOS DE DADOS : Programação, Especificação, Implementação. In : Escola de Computação, 5., 10-18 Jul. 1986, Belo Horizonte. Belo Horizonte : UFMG, 1986. 328 p.

[WER 91] WERMELINGER, M. ; LOPES, J. G. Uma Ferramenta para Aquisição e Representação de Conhecimento baseada em Grafos Conceptuais. In: SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 8., 18-21 Jul. 1991, Brasilia, DF. Anais... Brasília : SBC. 1991. 304 p. p 287-294.

[WIN 84] WINSTON, P. H. Artificial Intelligence. Chap. 8 : Representing Commonsense Knowledge. 1984. p. 251-287.

[WIR 83] WIRSING, M. ; PEPPER,P., PARTSCH, H. ; DOSCH, W. On Hierarquies of Abstract data types. Acta Informatica Berlin, v. 20, n. 1, p. 1-33, 1983.

[WIR 80] WIRSING, M. ; BROY, M. Abstract data types as lattices of finitely generated models. In : SYMPOSIUM ON MATHEMATICAL FOUNDATIONS COMPUTER SCIENCE, 9., Sept. 01-05, 1980, Rydzyna, Poland. Proceedings... Berlin : Springer-Verlag, 1980. 723 p. p. 673-685. (Lecture Notes in Computer Science, 88).

[WOO 75] WOODS, W. A. What's in a Link: Foundations for Semantic Networks. In : Representation and Understanding. New York : Academic Press, 1975. p. 35-82.

[WOO 83] WOODS, W. A. What's important about Knowledge Representation ?. IEEE Computer, v. 16, n. 10, p. 22-27, Oct. 1983.

[WUL 80] WULF, W. A. Abstract Data Types : A Retrospective and Prospective View. In : SYMPOSIUM ON MATHEMATICAL FOUNDATIONS COMPUTER SCIENCE, 9., 1980, Rydzyna, Poland. Berlin : Springer-Verlag, 1980. 723 p. p. 94-112. (Lecture Notes in Computer Science, 88).

[ZIL 79] ZILLES, S. N. An Introduction to Data Algebras. In: Abstract Software Specifications, Copenhagen Winter School, 1979. Proceedings... Berlin: Springer-Verlag, 1979. p. 248-272. (Lecture Notes in Computer Science, 86).

[ZIL 81] ZILLES, S. N. Types, Algebras and Modelling. ACM Sigplan Notices, New York, v. 16. n. 1, p. 207-209, Jan. 1981.

Informática
UFRGS

"Um sistema de tipos para uma linguagem de representação estruturada de conhecimentos".

Dissertação apresentada aos Srs.:

Prof. Dr. Benedito Melo Acioly (UFPe)

Prof. Dr. Daltro José Nunes

Prof. Dr. Paulo Alberto de Azeredo

Profa. Dra. Rosa Maria Viccari

Vista e permitida a impressão.
Porto Alegre, 29/06/93.

Prof. Dr. Paulo Alberto de Azeredo,
Orientador.

Prof. Dr. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.