ESPECIFICAÇÃO DO TIPO INTERVALAR PARAMETRIZADO EM CASL

Samara Pereira da Costa Melo

Dissertação submetida e aprovada no Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

NATAL, BRASIL
AGOSTO 2003

Divisão de Serviços Técnicos

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Melo, Samara Pereira da Costa.

Especificação do Tipo Intervalar Parametrizado em CASL. - Natal, RN, 2003.

113 f.

Orientador : Benjamin Rene Callejas Bedregal.

Dissertação (Mestrado) - Universidade Federal do Rio Grande do Norte. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas de Computação.

1. Linguagem de programação - Dissertação. 2. Intervalo parametrizado - Dissertação. 3. CASL - Dissertação. I. Bedregal, Benjamin René Callejas. II. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU 004.43(043.3)

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA

Os examinadores abaixo recomendam ao Departamento de Informática e Matemática Aplicada a aceitação da dissertação intitulada "**Especificação do Tipo Intervalar Parametrizado em CASL**" de autoria de **Samara Pereira da Costa Melo** como requisito para a obtenção do grau de **Mestre em Sistemas e Computação**.

Data: Agosto 2003

Orientador:

D.Sc. Benjamín René Callejas Bedregal (UFRN)

Examinadores:

D.Sc. Marcília Andrade Campos (UFPE)

D.Sc. Regivan Hugo Nunes Santiago (UFRN)

D.Sc. Anamaria Martins Moreira (UFRN)

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Data: **Agosto 2003**

Autor: **Samara Pereira da Costa Melo**

Titulo: **Especificação do Tipo Intervalar Parametrizado em CASL**

Depto: **Departamento de Informática e Matemática Aplicada**

Grau: **M.Sc.** Convocação: **Agosto** Ano: **2003**

Assinatura do Autor

A minha família e, especialmente, ao meu pai.

Sumário

Lista de Figuras

Lista de Símbolos

Símbolo	Significado Informal
$\mathbb{R}$	conjunto dos números reais
$\mathbb{I}(\mathbb{R})$	conjunto dos intervalos de números reais
$\bar{0}$	elemento neutro da soma de intervalos
$\bar{1}$	elemento neutro da multiplicação de intervalos
min	mínimo
max	máximo
pm	ponto médio
d	distância
abs_I	valor absoluto intervalar
amp	amplitude
$\cup$	união
$\Cup$	união convexa
$\cap$	interseção
$\vee$	ou
$\wedge$	e
$\subseteq$	contido ou igual
$\in$	pertence
$\notin$	não pertence
$\rightarrow$	se então

Símbolo	Significado Informal
$\leftrightarrow$	se e somente se
$\Rightarrow$	implicação na Álgebra de Heyting
$\Leftrightarrow$	biimplicação
$\leq$	menor igual
$\geq$	maior igual
I	matriz identidade intervalar
$amp(A_{ij})$	amplitude da matriz intervalar A
$pm(A_{ij})$	ponto médio da matriz intervalar A
$d(A_{ij}, B_{ij})$	distância entre as matrizes A e B
$\leq_k$	menor igual na ordem de Kulisch-Miranker
$\sqsubseteq$	menor igual para ordem de informação
$<_M$	menor para ordem de Moore
$\subset$	contido
$UB(A)$	(upper bound) conjunto de todos majorantes de A
$LB(A)$	(lower bound) conjunto de todos minorantes de A
$MUB(A)$	(minimum upper bound) conjunto de todos majorantes mínimos de A
$MLB(A)$	(maximum lower bound) conjunto de todos minorantes máximos de A
DS	conjunto dirigido
$x \sqcup y$	supremo de x e y
$x \sqcap y$	ínfimo de x e y
$\bigsqcup A$	supremo de A
$\bigsqcap A$	ínfimo de A
$l[a, b]$	(left) projeção esquerda
$r[a, b]$	(right) projeção direita
f^{-1}	função inversa de f
def	operação de definibilidade
$\overset{loc}{=}$	igualdade local

Símbolo	Significado Informal
$\otimes$	operador de multiplicação para o conjunto intervalar paramétrico
$\oplus$	operador de adição para o conjunto intervalar paramétrico

Agradecimentos

Gostaria de agradecer a muitas pessoas por ter proporcionado começar e terminar mais esta etapa da minha vida acadêmica. Posso dizer que não há uma pessoa em primeiro lugar, pois existe um intervalo delas que representam esta posição.

Agradeço ao meu orientador Prof. Benjamín Bedregal que acreditou em mim, apesar dos meus choros, e esteve sempre disponível, mesmo com os meus sumiços, e que hoje é uma pessoa que admiro mais ainda pois pude conhecer de perto a grande pessoa que ele é, de grandes simplicidade, compreessão e inteligência, qualidades as quais me espelho, e que um dia eu espero chegar lá; ao Prof. Regivan Santiago, que participou intensamente deste trabalho, diria mesmo que ele foi um co-orientador, um professor amigo que me socorreu em muitas ocasiões com seus grandes conhecimentos nesta área a qual entrei; a Profa. Anamaria Moreira, que deu o pontapé inicial para eu me interessar pelo mundo das especificações formais, quando eu ainda estava na graduação, especificamente, ao estudar para quarta prova;

Agradeço aos meus colegas e amigos que contribuiram, de alguma forma, para este trabalho e a Katiane Lopes que foi uma grande parceira;

Agradeço as pessoas da lista de discussão do HOL-CASL, em especial, Till Mossakowski, que foram atenciosos ao esclarecer nossas dúvidas tanto as encontradas na linguagem CASL quanto na ferramenta HOL-CASL;

Agradeço também ao meu pai, minha mãe e meus irmãos que sempre trabalharam muito para construir uma família sólida e unida, e que durante toda a vida sempre esteve do meu lado com muito amor e confiança; ao meu marido que conseguiu aturar meu mau humor bem como me apoiar em todos os momentos e decisões e ao meu filho que, inconscientemente,

abdicou da mãe por muito tempo; a minha santa sogra, meu sogro e minhas cunhadas que sempre se dispuseram a ajudar-me quando eu precisei.

Resumo

O uso do tipo intervalar em várias áreas favorece a idéia de se ter uma estrutura básica reutilizável, ou seja, um construtor intervalar que seja aplicado a um tipo de dados para se obter os intervalos desse tipo. Como um intervalo, intuitivamente é o conjunto de elementos que estão entre dois extremos, sua construção presupõe a noção de ordem, e portanto é razoável que este construtor trabalhe sobre tipos munidos de uma ordem parcial. Por outro lado, como o que se quer é operar com intervalos de objetos de um certo tipo como se opera com esses objetos, então também é razoável querer que as propriedades algébricas desses objetos sejam preservadas no seu tipo intervalar. Assim, o construtor intervalar fornece uma fundamentação teórica para o tipo intervalo parametrizado, ou seja, intervalos cujo parâmetro é generalizado podendo ser, por exemplo, números reais, complexos, etc.

A aplicação do intervalo em certas estruturas algébricas nem sempre garante a preservação de suas características, por exemplo, quando se trabalha com intervalos de números reais, seria conveniente que estes pudessem se comportar como se fossem os reais. Isto não acontece pois os reais satisfazem as propriedades algébricas de corpo, já os intervalos de reais não (por exemplo, não suporta a propriedade distributiva). Para superar esta dificuldade Santiago introduziu a teoria da igualdade local numa forma de enfraquecer a noção de igualdade "forte" fazendo com que propriedades satisfeitas localmente sejam relevantes, propriedades estas que antes poderiam ser descartadas.

A generalização da aritmética intervalar propõe a aplicação do construtor intervalar em estruturas algébricas ordenadas e enfraquecidas pela igualdade local, objetivando a manutenção de suas propriedades algébricas.

Como os intervalos são importantes em aplicações que trabalhem com dados contínuos, é interessante descrever esta teoria usando uma linguagem de especificação que permita o desenvolvimento de sistemas computacionais que usem tipos intervalares de modo disciplinado, confiável e seguro. Atualmente, as linguagens de especificações algébricas, que se baseiam em modelos matemáticos, têm sido muito usadas para este proposito. Dentre as

várias linguagens deste tipo existentes, foi escolhida CASL (*Common Algebraic Specification Language*) por conter diversas características relevantes para especificação do tipo intervalar parametrizado como, por exemplo, admitir parametrização e parcialidade.

Abstract

The interval datatype applications in several areas is important to construct a interval type reusable, i.e., a interval constructor can be applied to any datatype and get intervals this datatype. Since the interval is, of certain form, a set of elements limited for two bounds, left and right, with a order notions, then it's reasonable that interval constructor enclose datatypes with partial order. On the order hand, what we want is work with interval of any datatype like this we work with this datatype then. it's important to guarantee the properties of the datatype when maps to interval of this datatype. Thus, the interval constructor get a theory to parametrized interval type, i.e., a interval with generics parameters (for example rational, real, complex).

Sometimes, the interval application in some algebras doesn't guarantee the mainutenance of their properties, for example, when we use interval of real, that satisfies the field properties, it doesn't guarantee the distributivity propertie. A form to surpass this problem Santiago introduced the local equality theory that weakened the notion of "strong" equality, and thus, allowing some properties are local keeped, what can be discard before.

The interval arithmetic generalization aim to apply the interval constructor on ordered algebras weakened for local equality with the purpose of the keep their properties.

How the intervals are important in applications with continuous data, it's interesting specify that theory using a specification language that supply a system development using intervals of form disciplined, trustworth and safe. Currently, the algebraic specification language, based in math models, have been use to that intention often. We choose CASL (*Common Algebraic Specification Language*) among others languages because CASL has several characteristics excellent to parametrized interval type, such as, provide parcialiy and parametrization.

Capítulo 1

Introdução

A cada ano que passa a tecnologia avança com computadores superpotentes. Isso promove uma melhora em muitas áreas que precisam processar grandes quantidades de informação. Apesar de toda a capacidade existente, a máquina continua sendo limitada, fazendo com que números como π sejam representados com uma certa margem de erro, causado pelo truncamento ou arrendondamento do mesmo. Em processamento de informações físicas (fenômenos da natureza) como: distância, temperatura, tempo, intensidade, etc., isso fica mais grave pois a representação destes são limitadas inicialmente por dados obtidos através de algum instrumento que nunca consegue proporcionar o valor real, e consequentemente faz seu processamento gerar resultados com incertezas, devido à propagação do erro inicial.

Temos como exemplos de problemas de representação três grandes catástrofes [14] como a explosão do foguete Ariane5 em junho de 1996, o Missil Patriot em fevereiro de 1991, o acidente na plataforma Sleipner em agosto de 1991, resultantes de erros de limitações da máquina que não consegue tratar os números com toda sua extensibilidade, mesmo se tratando de máquinas supervelozes com capacidade de executar muitas tarefas. Isso acontece porque a representação do número real não pode ser feita de forma finita [12].

No final da década de 50, Moore [28] e Sunaga [51] propuseram a matemática intervalar. Os números representados como intervalos tendem a controlar a propagação do erro, tornando-a previsível e controlável. É possível computar um intervalo $[a, b]$ contendo uma

solução exata x para um problema, tomando-se o ponto médio do intervalo, $m = (a + b)/2$, como sendo a aproximação para a solução x e assim obtem-se $|x - m| \leq (b - a)/2$, ou seja, a distância (erro absoluto) entre x e m é no máximo a metade do comprimento entre a e b, com isso a computação de um intervalo contém uma solução exata, uma aproximação da solução e um limite de erro na solução aproximada [30], ou seja, os resultados intervalares sempre carregam consigo uma qualidade e um grau de incerteza, pois o diâmetro de um intervalo solução é um indicativo da influência do erro do dado de entrada no erro do resultado final obtido [12].

A teoria de Moore [28, 31, 29, 30] compreende além dos intervalos reais, os intervalos complexos, matrizes e vetores de intervalos reais ou complexos. Outras áreas do conhecimento também usam algum tipo intervalar, por exemplo, intervalos de valores entre 0 e 1 usados na lógica fuzzy [50, 52] e na probabilidade [10, 11], processamento digital de imagens [26]. Apesar de existir uma gama de aplicações de diversos tipos de intervalos, todas elas estão construídas sobre uma estrutura comum do tipo intervalo, o que torna interessante a construção de um tipo intervalar paramétrico. Este tipo poderia ter como parâmetros diferentes tipos de dados se comportando como um TAD (Tipo Abstrato de Dados). Os TAD são independentes de suas representações, no sentido que detalhes de como eles são implementados são geralmente escondidos e, ao usuário do tipo, basta informar o que ele faz e como ele funciona.

O tipo intervalar paramétrico é uma generalização da teoria intervalar sobre os números reais para outros tipos intervalares possíveis. Uma vez que os intervalos reais estão definidos através de uma ordem parcial sobre o conjunto dos reais, podemos definir os intervalos sobre qualquer conjunto parcialmente ordenado. Porém, as propriedades algébricas do tipo original nem sempre são as mesmas do tipo intervalar. Por exemplo, os reais formam um corpo enquanto que os intervalos de reais formam uma estrutura algébrica mais fraca (um anelóide [23]). Para podermos preservar a estrutura algébrica teremos que usar a teoria da igualdade local para enfraquecer as estruturas algébricas em estruturas algébricas locais. A fundamentação teórica para construir o tipo intervalar paramétrico é fornecida pelo chamado construtor intervalar, que será usado para construir a estrutura algébrica local intervalar de tal forma que, ao aplicar o construtor intervalar, por exemplo, no corpo dos números reais

obteremos um corpo local ordenado de informações reais. Com isso há uma generalização da aritmética intervalar real de Moore.

A álgebra é um importante método para especificação de tipos abstratos de dados, uma vez que ela trabalha sobre operações e predicados que satisfazem um determinado *sort*; enfatizando-se a funcionalidade e comportamento, as especificações algébricas podem ser entendidas como tipos abstratos de dados, pois têm como teoria matemática a lógica equacional. Os modelos matemáticos ou interpretações que satisfazem essa teoria são conhecidos como álgebras da teoria [17, 53].

Hoje em dia, linguagens de programação como C, Pascal e Java tem sido estendidas para usar o tipo de dado intervalo como primitivo, estas linguagens são chamadas de **XSC** (e**X**tension for **S**cientific **C**omputation) [2, 16, 21]. Porém a idéia deste trabalho é especificar o tipo intervalar paramétrico usando uma linguagem de especificação algébrica que dê suporte à fundamentação teórica desse tipo, como é o caso de CASL (*Common Algebraic Specification Language*) que suporta, por exemplo, parcialidade que é uma condição necessária para a construção de conjuntos locais. Um outro exemplo desse tipo de linguagem é OBJ3, para qual já foi desenvolvida a teoria intervalar [10].

Para um entendimento da especificação do tipo intervalar paramétrico, inicialmente, no capítulo 2, mostraremos as características da matemática intervalar que deve ser generalizada para dar suporte ao construtor intervalar. A teoria para essa generalização é encontrada no capítulo 3. Para especificação desse construtor bem como sua análise foram usados CASL e HOL-CASL que serão apresentadas no capítulo 4. Finalmente, no capítulo seguinte temos a especificações para o tipo intervalar parametrizado e por fim teremos as conclusões e trabalhos futuros.

Capítulo 2

Matemática Intervalar

A matemática intervalar foi introduzida no final da década de 50 se dinfundiu, principalmente, devido aos efeitos causados em máquinas computacionais pelos erros de truncamento e/ou arredondamento e propagação de erro. Tais tipos de erros eram causados diretamente pela incerteza dos dados iniciais ou incerteza dos parâmetros nos modelos matemáticos [30]. Várias áreas científicas como física, química, estatística, movimento de satélites, sistemas Fuzzy [50], etc., que trabalham dependendo de resultados de cálculos mais precisos, foram e são grandes impulsionadoras da teoria intervalar.

"A Matemática Intervalar é uma teoria matemática que se apresenta como uma solução não somente para o controle rigoroso e automático dos erros de resultados em computações numéricas, mas também para o tratamento e modelagem da incerteza em computação"[12].

Em virtude da sua crescente relevância, a matemática intervalar tem sido acrescentada a ferramentas matemáticas como MatLab e Maple [12], onde tem sido desenvolvidos projetos de bibliotecas que contém funções sobre intervalos. Linguagens de programação de alto nível como Pascal e Java [16], também são alvo de projetos que estendem um sistema de tipos.

Neste capítulo serão apresentados alguns conceitos da teoria intervalar.

2.1 Conceitos Básicos

Definição 2.1.1 (Intervalo) *Um* **intervalo**[1] *é representado por um par de elementos, onde o primeiro elemento do par, limite inferior, é menor ou igual ao segundo elemento do par, limite superior. Assim, um intervalo é definido da seguinte forma:*

$$[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\} \tag{2.1}$$

onde $a, b \in \mathbb{R}$.

Com esta definição tem-se que um intervalo tem uma natureza dual: ora é visto como um conjunto de números reais, ora como um par de números reais.

A igualdade entre intervalos se dá quando os limites inferiores e superiores dos intervalos assim o forem:

$$[a, b] = [c, d] \Leftrightarrow a = c \wedge b = d \tag{2.2}$$

onde a, b, c e $d \in \mathbb{R}$.

Quando os limites inferior e superior de um intervalo são iguais, o intervalo é dito **degenerado**, $[a, a] = a, a \in \mathbb{R}$. $[a, a]$ é a representação do número real a em $\mathbb{I}(\mathbb{R})$, que é o conjunto dos intervalos de reais. Logo nós temos uma imersão de $\mathbb{R}$ em $\mathbb{I}(\mathbb{R})$.

2.2 Aritmética Intervalar

Na aritmética intervalar introduzida por Moore [28] e Sunaga [51] o conjunto $\mathbb{I}(\mathbb{R})$ provê as operações binárias de adição, subtração, multiplicação e divisão, bem como operações unárias. Tais operações estão definidas a seguir:

Definição 2.2.1 (Adição) *Sejam* $X = [a, b]$ *e* $Y = [c, d]$*, então a* **soma** *de* X *e* Y *é dado por* $X + Y = [a, b] + [c, d] = \{x + y : x \in [a, b] \wedge y \in [c, d]\}$.

[1] *Um intervalo fechado de números reais é chamado, nesse texto, simplesmente, intervalo.*

Definição 2.2.2 (Pseudo Inverso Aditivo) *Seja* $X = [a, b]$*, o* **pseudo inverso aditivo** *de* X *é dado por* $-X = -([a, b]) = \{-x : x \in X\} = [-b, -a]$.

O pseudo inverso aditivo é assim chamado pelo fato de nem sempre a igualdade $X - X = 0$ ser verdadeira. Por exemplo, seja um intervalo $X = [0, 2]$, consequentemente $-X = [-2, 0]$, assim $X - X = [-2, 2] \neq [0, 0]$, porém $[0, 0] \subseteq X - X$.

Definição 2.2.3 (Subtração) *Sejam* $X = [a, b]$ *e* $Y = [c, d]$*, então a* **diferença** *entre* X *e* Y *é dado por* $X - Y = \{x - y : x \in [a, b] \wedge y \in [c, d]\}$.

Sunaga [51] provou que $[a, b]+[c, d] = [a+c, b+d]$, $-[a, b] = [-b, -a]$ e $x-y = [a-d, b-c]$.

Definição 2.2.4 (Multiplicação) *Sejam* $X = [a, b]$ *e* $Y = [c, d]$*, então a* **multiplicação** *de* X *e* Y *é dado por* $X * Y = \{x * y : x \in X, y \in Y\}$.

Como o resultado de $X * Y$ é também um intervalo, conforme provado por Sunaga [51], temos que seus limites inferior e superior são dados por $X * Y = [min(a * c, a * d, b * c, b * d), max(a * c, a * d, b * c, b * d)]$.

De acordo com os sinais de a, b, c e d é possível reduzir a fórmula acima em nove casos especiais, que são:

1. Se $a \geq 0$ e $c \geq 0$ então $X * Y = [a * c, b * d]$
2. Se $a < 0 < b$ e $c \geq 0$ então $X * Y = [a * d, b * d]$
3. Se $b \leq 0$ e $c \geq 0$ então $X * Y = [a * d, b * c]$
4. Se $a \geq 0$ e $c < 0 < d$ então $X * Y = [b * c, b * d]$
5. Se $b \leq 0$ e $d \leq 0$ então $X * Y = [a * d, a * c]$
6. Se $a \leq 0$ e $d \leq 0$ então $X * Y = [b * c, a * d]$
7. Se $a < 0 < b$ e $d \leq 0$ então $X * Y = [b * c, a * c]$

8. Se $b \leq 0$ e $d \leq 0$ então $X * Y = [b * d, a * c]$

9. Se $a < 0 < b$ e $c < 0 < d$ então $X * Y = [min(a * d, c * b), max(a * c, b * d)]$

Definição 2.2.5 (Pseudo inverso multiplicativo) *Seja um intervalo* $X = [a, b]$*, o* **pseudo inverso** *de* X *é dado por* $1/X$*, desde que o número zero não pertença a* X*, ou seja,* $1/X = 1/[a, b] = [1/b, 1/a]$ *se* $b < 0$ *ou* $a > 0$*.*

Definição 2.2.6 (Divisão) *Sejam* $X = [a, b]$ *e* $Y = [c, d]$*, a* **divisão** X/Y *é dada por* $X/Y = X * (1/Y) = [a, b] * [1/d, 1/c]$ *se o número* 0 *não pertence a* Y*, ou seja, ou* $c > 0$ *ou* $d < 0$*.*

2.3 Relações de Ordem sobre Intervalos

Os intervalos reais têm várias semânticas como, por exemplo, representação de números reais, quanto mais próximos os extremos do intervalo estiverem do valor "correto", melhor será a representação desse valor. A definição de ordem entre intervalos depende da abordagem ou semântica utilizada. A ordem de informação [3, 48] é um tipo de relação entre intervalos que satisfaz a abordagem de representação. A seguir serão definidas a ordem de informação, bem como as ordens de Moore [30], de inclusão e de Kulisch-Miranker [23].

Definição 2.3.1 (Ordem da informação) *Sejam dois intervalos* $[a, b]$ *e* $[c, d]$*, a* **ordem da informação** *define que* $[a, b] \sqsubseteq [c, d] \Leftrightarrow [a, b] \subseteq [c, d] \Leftrightarrow a \leq c$ *e* $d \leq b$*.*

Definição 2.3.2 (Ordem de Moore) *Sejam dois intervalos* $[a, b]$ *e* $[c, d]$*, a* **ordem de Moore** *define que* $[a, b] <_M [c, d] \Leftrightarrow \forall x \in [a, b]$ *e* $\forall y \in [c, d], x < y \Leftrightarrow b < c$*.*

Definição 2.3.3 (Ordem de inclusão) *Sejam dois intervalos* $[a, b]$ *e* $[c, d]$*, a* **ordem de inclusão** *define que* $[a, b] \leq [c, d] \Leftrightarrow c \leq a$ *e* $b \leq d$*.*

Definição 2.3.4 (Ordem de Kulisch-Miranker) *Sejam dois intervalos* $[a, b]$ *e* $[c, d]$*, a* **ordem de Kulisch-Miranker** *define que* $[a, b] \leq_K [c, d] \Leftrightarrow \forall x \in [a, b].\exists y \in [c, d], x \leq y \wedge \forall y \in [c, d].\exists x \in [a, b], x \leq y \Leftrightarrow a \leq c \ e \ b \leq d.$

Proposição 2.3.1 (Inclusão Monotônica) *Sejam* X, Y, Z, W *intervalos, tais que* $X \subseteq Z$ *e* $Y \subseteq W$*. Então valem as relações:*

- $X + Y \subseteq Z + W$
- $-X \subseteq -Z$
- $X - Y \subseteq Z - W$
- $X * Y \subseteq Z * W$
- $1/X \subseteq 1/Z$*, sempre que* $0 \notin Z$
- $X/Y \subseteq Z/W$*, sempre que* $0 \notin W$

Prova: Direta das definições dos operadores aritméticos.

2.4 Propriedades Algébricas

As propriedades algébricas da aritmética intervalar são consequências imediatas do conjunto de definições das operações da aritmética intervalar.

Proposição 2.4.1 *Para todo* $X, Y, Z \in \mathbb{I}(\mathbb{R})$ *vale:*

- **Associatividade na adição***:* $(X + Y) + Z = X + (Y + Z)$
- **Associatividade na multiplicação***:* $(X * Y) * Z = X * (Y * Z)$

- **Comutatividade na adição**: $X + Y = Y + X$
- **Comutatividade na multiplicação**: $X * Y = Y * X$
- **Identidade na adição**: *Existe um único elemento* $\bar{0} \in \mathbb{I}(\mathbb{R})$ *tal que* $X + \bar{0} = \bar{0} + X = X$
- **Identidade na multiplicação**: *Existe um único elemento* $\bar{1} \in \mathbb{I}(\mathbb{R})$ *tal que* $X * \bar{1} = \bar{1} * X = X$
- **Subdistributividade**: $X * (Y + Z) \subseteq X * Y + X * Z$

Prova: Ver [31].

Em relação às identidades aditiva e multiplicativa tem-se que $\bar{0} = [0, 0]$ e $\bar{1} = [1, 1]$, respectivamente.

A distributividade em intervalos $X * (Y + Z) = X * Y + X * Z$ só acontecerá se X for um intervalo degenerado, ou se Y e/ou Z forem o intervalo $\bar{0}$.

Definição 2.4.1 (Intervalo Simétrico) *Seja* $X = [a, b] \in \mathbb{I}(\mathbb{R})$ *um intervalo. Dizemos que* X *é* **simétrico**, *se* $-X = X$.

2.4.1 Funções de Intervalos em Reais

Para os intervalos estão definidas algumas funções que podem determinar características importantes na solução de um problema, como a amplitude de um intervalo que determina a qualidade do mesmo, enquanto representação de um número real. Algumas funções são definidas a seguir:

Definição 2.4.2 (Projeções Esquerda - l e Direita - r) *A* **projeção esquerda**, l, *de um intervalo* $X = [a, b]$ *é o limite inferior, ou seja,* $l(X) = a$. *Já* **projeção direita**, r, *se caracteriza pelo limite superior, ou seja,* $r(X) = b$.

Definição 2.4.3 (Ponto médio - pm) *O* **ponto médio** *de um intervalo* $X = [a, b]$ *é dado por* $pm([a, b]) = (a + b)/2$.

Definição 2.4.4 (Distância - d) *A* **distância** *entre dois intervalos* $X = [a, b]$ *e* $Y = [c, d]$ *é dada por* $d([a, b], [c, d]) = max(abs(a - c), abs(b - d))$ *onde,* $abs(x)$ *é o valor absoluto de* x *para* $x \in \mathbb{R}$.

Definição 2.4.5 (Valor Absoluto - abs$_I$) *O* **valor absoluto** *de um intervalo* $X = [a, b]$ *é o valor máximo obtido entre os valores absolutos dos seus limites inferior e superior, ou seja,* $abs_I([a, b]) = max(abs(a), abs(b))$.

Definição 2.4.6 (Amplitude - amp) *A* **amplitude** *de um intervalo* $X = [a, b]$ *é dada pela diferença dos seus limites inferior e superior, isto é,* $amp([a, b]) = b - a$.

2.4.2 Operações entre Conjuntos

Definição 2.4.7 (União) *Sejam* $X, Y \in \mathbb{I}(\mathbb{R})$*, onde* $X = [a, b]$ *e* $Y = [c, d]$*. A* **união** *entre* X *e* Y *é definida por* $[a, b] \cup [c, d] = [min(a, c), max(b, d)]$ *a se* $(c \geq a \wedge c \leq b) \vee (a \geq c \wedge a \leq d)$.

Definição 2.4.8 (União Convexa) *A* **união convexa** $\Cup$ *entre intervalos é definida pelo menor intervalo contendo ambos, ou seja, sendo* $X = [a, b]$ *e* $Y = [c, d]$ *temos que* $[a, b] \Cup [c, d] = [min(a, c), max(b, d)]$.

Definição 2.4.9 (Interseção) *Sejam os intervalos* $X, Y \in \mathbb{I}(\mathbb{R})$*, onde* $X = [a, b]$ *e* $Y = [c, d]$*. A* **interseção** *entre* X *e* Y *é dada por* $[a, b] \cap [c, d] = [max(a, c), min(b, d)]$ *se* $(c \geq a \wedge c \leq b) \vee (a \geq c \wedge a \leq d)$.

2.5 Matriz de Intervalos

Definição 2.5.1 (Matriz Intervalar) *Uma* **matriz intervalar** *é definida por uma matriz $A = (A_{ij})_{m\times n}$ com ordem m (linhas) por n (colunas), onde cada elemento (A_{ij}) é um intervalo.*

$A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ é um exemplo de matriz intervalar com duas linhas e duas colunas.

Supondo $A = (A_{ij})$ uma matriz intervalar de ordem $m \times n$, se $m = 1$, então a matriz A é chamada **matriz linha** ou **vetor linha** e, se $n = 1$, então A é chamada **matriz coluna** ou **vetor coluna**. Uma matriz coluna, por convenção, é considerada um **vetor intervalar**.

$A = \begin{pmatrix} [a,b] & [c,d] & [e,f] \end{pmatrix}$ é um exemplo de vetor linha, uma matriz de uma linha e três colunas, e $B = \begin{pmatrix} [a,b] \\ [c,d] \\ [e,f] \end{pmatrix}$ é um exemplo de vetor coluna, uma matriz de três linhas e uma coluna.

Definição 2.5.2 (Igualdade entre Matrizes Intervalares) *Sejam as matrizes intervalares: $A = (A_{ij})_{m\times n}$ e $B = (B_{ij})_{r\times s}$. Diz-se que a* **igualdade** *$A = B$ ocorre se, e somente se, $m = r$, $n = s$ e $A_{ij} = B_{ij}$, para todos os índices $1 \leq i \leq n$ e $1 \leq j \leq m$.*

Exemplo: as matrizes $A = \begin{pmatrix} [1,3] & [-1,1] \\ [1,e] & [3,3] \end{pmatrix}$ e $B = \begin{pmatrix} [\sqrt{1},\sqrt{9}] & [-1,1] \\ exp[0,1] & [3,3] \end{pmatrix}$ são matrizes intervalares iguais.

Definição 2.5.3 (Matriz Intervalar Nula) *Uma matriz intervalar $A = (A_{ij})_{m\times n}$ é considerada* **nula** *se todos os seus elementos são nulos, ou seja, se $A_{ij} = [0, 0]$, $\forall i, j$. A matriz nula é representada por 0.*

Exemplo: $A = \begin{pmatrix} [0,0] & [0,0] \\ [0,0] & [0,0] \end{pmatrix}$ é uma matriz nula de ordem 2×2.

Definição 2.5.4 (Matriz Intervalar Identidade) *Uma matriz intervalar $I = (I_{ij})_{m\times m}$ é chamada* **identidade**, *se todos os seus elementos da diagonal principal são intervalos identidades e os demais elementos são intervalos nulos, ou seja, se $I_{ij} = [1, 1]$ para $i = j$ e I_{ij} = [0,0] para $i \neq j$.*

Exemplo: $I = \begin{pmatrix} [1,1] & [0,0] \\ [0,0] & [1,1] \end{pmatrix}$ é uma matriz intervalar identidade com duas linhas e duas colunas.

2.5.1 Operações Aritméticas entre Matrizes Intervalares

A seguir serão apresentadas as principais operações com matrizes intervalares como: soma, subtração, produto por um intervalo, multiplicação.

Definição 2.5.5 (Soma de Matrizes Intervalares) *Sejam $A = (A_{ij})$ e $B = (B_{ij})$ duas matrizes intervalares. A matriz* **soma** *das matrizes A e B, é definida como sendo a matriz $C = A + B$, onde cada elemento é definido por $C_{ij} = A_{ij} + B_{ij}, \forall i, j$.*

Exemplo: dadas as matrizes intervalares $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e $B = \begin{pmatrix} [m,n] & [o,p] \\ [q,r] & [s,t] \end{pmatrix}$ então, $C = A + B = \begin{pmatrix} [a,b]+[m,n] & [c,d]+[o,p] \\ [e,f]+[q,r] & [g,h]+[s,t] \end{pmatrix}$.

Definição 2.5.6 (Produto de um Intervalo por uma Matriz) *Seja $A = (A_{ij})$ uma matriz intervalar de ordem $m \times n$ e X um intervalo. O* **produto do intervalo** *X pela matriz A é a matriz intervalar $C = (C_{ij})$, onde cada elemento é dado por $C_{ij} = X * A_{ij}$, $\forall_{ij}$.*

Exemplo: dada a matriz $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e o intervalo $X = [p, q]$ então, $C = XA = \begin{pmatrix} [p,q]*[a,b] & [p,q]*[c,d] \\ [p,q]*[e,f] & [p,q]*[g,h] \end{pmatrix}$.

Definição 2.5.7 (Subtração de Matrizes Intervalares) *Sejam $A = (A_{ij})$ e $B = (B_{ij})$ duas matrizes intervalares de mesma ordem. A* **diferença** *entre A e B é definida por $C = A - B$, onde cada elemento é formado por $C_{ij} = A_{ij} - B_{ij}, \forall i, j$. A diferença pode ser entendida também como $C = A - B = A + ([-1, -1] * B)$.*

Exemplo: dadas as matrizes intervalares $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e $B = \begin{pmatrix} [m,n] & [o,p] \\ [q,r] & [s,t] \end{pmatrix}$ então, $C = A - B = \begin{pmatrix} [a,b]-[m,n] & [c,d]-[o,p] \\ [e,f]-[q,r] & [g,h]-[s,t] \end{pmatrix}$ e como $-B = [-1, -1] * B = \begin{pmatrix} [-n,-m] & [-p,-o] \\ [-r,-q] & [-t,-s] \end{pmatrix}$, tem-se: $C = A + (-B) = \begin{pmatrix} [a,b]+[-n,-m] & [c,d]+[-p,-o] \\ [e,f]+[-r,-q] & [g,h]+[-t,-s] \end{pmatrix}$.

Definição 2.5.8 (Multiplicação entre Matrizes Intervalares) *Sejam* $A = (A_{ij})_{m\times p}$ *e* $B = (B_{ij})_{p\times n}$. *A* **multiplicação** *de A por B é uma matriz intervalar* $C = (C_{ij})_{m\times n} = A*B$, *cujos elementos são dados por* $(C_{ij}) = \Sigma_{k=1}^{p} A_{ik} \times B_{kj}$.

Exemplo: dadas as matrizes intervalares $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e $B = \begin{pmatrix} [m,n] & [o,p] \\ [q,r] & [s,t] \end{pmatrix}$ então, $C = A * B = \begin{pmatrix} [a,b]*[m,n]+[c,d]*[q,r] & [a,b]*[o,p]+[c,d]*[s,t] \\ [e,f]*[m,n]+[g,h]*[q,r] & [e,f]*[o,p]+[g,h]*[s,t] \end{pmatrix}$.

2.5.2 Operações entre Conjuntos com Matrizes Intervalares

Definição 2.5.9 (Interseção de Matrizes Intervalares) *Sejam* $A = (A_{ij})$ *e* $B = (B_{ij})$ *duas matrizes intervalares de mesma ordem. A* **interseção** *de A com B é a matriz intervalar* $C = A \cap B$, *onde cada* $C_{ij} = A_{ij} \cap B_{ij}$, $\forall i, j$.

Exemplo: dadas as matrizes $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e $B = \begin{pmatrix} [m,n] & [o,p] \\ [q,r] & [s,t] \end{pmatrix}$ então, $C = A \cap B = \begin{pmatrix} [a,b]\cap[m,n] & [c,d]\cap[o,p] \\ [e,f]\cap[q,r] & [g,h]\cap[s,t] \end{pmatrix}$.

Se não existir interseção para algum par de elementos correpondentes, ou seja, $A_{ij} \cap B_{ij} = \emptyset$ para algum i, j, então a interseção entre A e B não existe.

Definição 2.5.10 (União de Matrizes Intervalares) *Sejam* $A = (A_{ij})$ *e* $B = (B_{ij})$ *duas matrizes intervalares de mesma ordem. A* **união** *de A com B é a matriz intervalar* $C = A \cup B$, *onde cada* $C_{ij} = A_{ij} \cup B_{ij}$, $\forall i, j$.

Exemplo: dadas as matrizes $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e $B = \begin{pmatrix} [m,n] & [o,p] \\ [q,r] & [s,t] \end{pmatrix}$ então, $C = A \cup B = \begin{pmatrix} [a,b]\cup[m,n] & [c,d]\cup[o,p] \\ [e,f]\cup[q,r] & [g,h]\cup[s,t] \end{pmatrix}$.

Se não existir união para algum par de elementos correpondentes, ou seja, $A_{ij} \cup B_{ij} = \emptyset$ para algum i, j, então, a união entre A e B não existe.

Definição 2.5.11 (Inclusão de Matrizes Intervalares) *Sejam* $A = (A_{ij})$ *e* $B = (B_{ij})$ *duas matrizes intervalares de mesma ordem. A matriz A é uma* **inclusão***para B, se todo*

elemento da matriz A está contido no seu correspondente elemento da matriz B, ou seja, $A \subseteq B \Leftrightarrow A_{ij} \subseteq B_{ij}$, $\forall i, j$.

Exemplo: dadas as matrizes $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e $B = \begin{pmatrix} [m,n] & [o,p] \\ [q,r] & [s,t] \end{pmatrix}$ então, $A \subseteq B \Leftrightarrow [a,b] \subseteq [m,n], [c,d] \subseteq [o,p] \quad e [e,f] \subseteq [q,r], [g,h] \subseteq [s,t]$.

2.5.3 Propriedades das Matrizes Intervalares

Proposição 2.5.1 *Para quaisquer matrizes intervalares A, B e C as seguintes propriedades são válidas:*

- **Associatividade***: $A + (B + C) = (A + B) + C$*
- **Comutatividade***: $A + B = B + A$*
- **Elemento Neutro da Adição***: Existe uma única matriz intervalar 0 tal que $A + 0 = 0 + A = A$.*
- **Elemento Neutro da Multiplicação***: Existe uma única matriz intervalar I tal que $A * I = I * A = A$*
- **Subdistributividade***: $(A + B) * C \subseteq A * C + B * C$*

Em virtude da lei da subdistributividade a propriedade associativa da multiplicação não é válida para as matrizes intervalares, ou seja, $A * (B * C) \neq (A * B) * C$. É possível observar que a ordem de adição e multiplicação dos pares de elementos é diferente.

Exemplo: dadas as matrizes intervalares $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ e $B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$ e $C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$, temos:

1. $(A * B) * C = \begin{pmatrix} (a_{11}*b_{11}+a_{12}*b_{21}) & (a_{11}*b_{12}+a_{12}*b_{22}) \\ (a_{21}*b_{11}+a_{22}*b_{21}) & (a_{21}*b_{12}+a_{22}*b_{22}) \end{pmatrix} * \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} =$

$$\begin{pmatrix} ((a_{11}*b_{11}+a_{12}*b_{21})*c_{11}+(a_{11}*b_{12}+a_{12}*b_{22})*c_{21}) & ((a_{11}*b_{11}+a_{12}*b_{21})*c_{12}+(a_{11}*b_{12}+a_{12}*b_{22})*c_{22}) \\ ((a_{21}*b_{11}+a_{22}*b_{21})*c_{11}+(a_{21}*b_{12}+a_{22}*b_{22})*c_{21}) & ((a_{21}*b_{11}+a_{22}*b_{21})*c_{12}+(a_{21}*b_{12}+a_{22}*b_{22})*c_{22}) \end{pmatrix}$$

2. $A * (B * C) = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} (b_{11}*c_{11}+b_{12}*c_{21}) & (b_{11}*c_{12}+b_{12}*c_{22}) \\ (b_{21}*c_{11}+b_{22}*c_{21}) & (b_{21}*c_{12}+b_{22}*c_{22}) \end{pmatrix} =$
$\begin{pmatrix} (a_{11}*(b_{11}*c_{11}+b_{12}*c_{21})+a_{12}*(b_{21}*c_{11}+b_{22}*c_{21})) & ((a_{11}*(b_{11}*c_{12}+b_{12}*c_{22})+a_{12}*(b_{21}*c_{12}+b_{22}*c_{22})) \\ (a_{21}*(b_{11}*c_{11}+b_{12}*c_{21})+a_{22}*(b_{21}*c_{11}+b_{22}*c_{21})) & (a_{21}*(b_{11}*c_{12}+b_{12}*c_{22})+a_{22}*(b_{21}*c_{12}+b_{22}*c_{22})) \end{pmatrix}$

Observando cada elemento das matrizes resultantes de (1) e (2) verifica-se que eles são diferentes.

A comutatividade também não é válida para a multiplicação de matrizes, isto é, sendo A_{ij} e B_{ij} duas matrizes intervalares então $A * B \neq B * A$, e isto é um fato da teoria das matrizes.

2.5.4 Outras Definições sobre Matrizes Intervalares

Definição 2.5.12 (Matriz Amplitude) *Seja $A = (A_{ij})$ uma matriz intervalar. A* **matriz amplitude** *de A é a matriz onde cada elemento corresponde a amplitude do respectivo intervalo da matriz intervalar, ou seja, $amp(A) = (amp(A_{ij}))$.*

Exemplo: seja $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ então, $amp(A) = \begin{pmatrix} amp([a,b]) & amp([c,d]) \\ amp([e,f]) & amp([g,h]) \end{pmatrix}$.

Definição 2.5.13 (Matriz Ponto Médio) *Seja $A = (A_{ij})$ uma matriz intervalar. A* **matriz ponto médio** *de A é a matriz onde cada elemento corresponde ao ponto médio do respectivo intervalo da matriz intervalar, ou seja, $pm(A) = (pm(A_{ij}))$.*

Exemplo: seja $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ então, $pm(A) = \begin{pmatrix} pm([a,b]) & pm([c,d]) \\ pm([e,f]) & pm([g,h]) \end{pmatrix}$.

Definição 2.5.14 (Distância entre Matrizes Intervalares) *Sejam A e B duas matrizes intervalares da mesma ordem. A* **distância** *D entre A e B é a matriz real cujos elementos correspondem a distância entre os respectivos intervalos das matrizes A e B, ou seja, $D(A,B) = (d(A_{ij}, B_{ij}))$, $\forall i, j$.*

Exemplo: dadas as matrizes $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ e $B = \begin{pmatrix} [m,n] & [o,p] \\ [q,r] & [s,t] \end{pmatrix}$ então, $D(A,B) = \begin{pmatrix} d([a,b],[m,n]) & d([c,d],[o,p]) \\ d([e,f],[q,r]) & d([g,h],[s,t]) \end{pmatrix}$.

Definição 2.5.15 (Matriz Módulo) *Seja $A = (A_{ij})$ uma matriz intervalar. A* **matriz módulo** *de A é a matriz onde cada elemento corresponde ao módulo do respectivo intervalo da matriz intervalar, ou seja, $|A| = (|A_{ij}|)$.*

Exemplo: seja $A = \begin{pmatrix} [a,b] & [c,d] \\ [e,f] & [g,h] \end{pmatrix}$ então, $|(A)| = \begin{pmatrix} abs_I([a,b]) & abs_I([c,d]) \\ abs_I([e,f]) & abs_I([g,h]) \end{pmatrix}$.

2.6 Considerações Finais do Capítulo

Como foi visto, o conjunto $\mathbb{I}(\mathbb{R})$ comtempla várias propriedades, algebricas e topológicas, do conjunto dos reais, mas não todas. Isso ocorre, por exemplo, com a operação de subtração, ou seja, tomando um $x \in \mathbb{R}$ é possível concluir que $x - x = 0$, já se tomarmos o intervalo não degenerado $X \in \mathbb{I}(\mathbb{R})$ a mesma equação, $X - X = 0$, nem sempre será válida (por exemplo, $X = [-2, 0], [-2, 0] - [-2, 0] = [-2, 0] + (-[0, 2]) = [-2, 2]$). Porém o resultado da operação contém o resultado desejado (seguindo o exemplo, $[0, 0] \subset [-2, 2]$). Isso acontece também com outras propriedades sobre $\mathbb{I}(\mathbb{R})$.

Uma vez que se deseja representar $\mathbb{R}$ através de $\mathbb{I}(\mathbb{R})$ devemos garantir que as propriedades do primeiro conjunto sejam respeitadas no segundo. Em [44] foi apresentada uma teoria, teoria da igualdade local, para garantir esta representação. Este fato somado à diversidade de aplicações dos intervalos fortaleceu a possibilidade de estender essa capacidade de representação dos intervalos sobre outras estruturas algébricas [9], o que será apresentado no próximo capítulo.

Capítulo 3

Generalização da Aritmética de Moore

Moore [30] desenvolveu uma aritmética intervalar a fim de resolver o controle de erro em computações numéricas. No entanto, a aritmética desenvolvida por Moore que estende a aritmética dos números reais, não é completamente fiel a ela, no sentido que a estrutura algébrica de intervalos não é um corpo, como o são os números reais. Isto pode ter consequências, por exemplo, na resolução de equações de segundo grau, que tem diversas aplicações e várias em áreas científicas e tecnologicas. Vários trabalhos tem sido desenvolvidos para superar esta dificuldade, entre eles está o de Markov [27]. Santiago em [44] observou que o problema não vem da estrutura algébrica dos intervalos, mas da noção primitiva de igualdade para os intervalos; com isso, propôs uma nova noção de igualdade para intervalos, chamada *igualdade local*, e determinou algumas das consequências de tal abordagem. Uma delas é a simulação do comportamento de corpo sobre os intervalos. Ele apresentou uma abordagem à luz da teoria dos domínios de Scott para intervalos e uma "igualdade simples de Scott", restringindo esta última à igualdade local.

A teoria de R. Moore não compreende somente os intervalos reais, mas também intervalos complexos, matrizes e vetores de intervalos reais ou complexos. Tal teoria vem sendo aplicada em várias linguagens de programação e ferramentas matemáticas. Hoje, em algumas linguagens de programação seus sistemas de tipos têm sido estendidos para usar estes tipos de dados intervalares como primitivos. No futuro, é relevante pensar num tipo único que

absorva tipos de dados intervalares usados em qualquer ciência, ou seja, que suportem o tipo de dado intervalo paramétrico.

Visto que intervalos reais são definidos através da ordem parcial sobre o conjunto dos reais, é possível definir, de forma análoga, intervalos sobre qualquer conjunto parcialmente ordenado. Bedregal em [6, 7, 8] tratou o conjunto dos intervalos como um construtor categórico onde foram mostradas que algumas categorias de domínio, como a categoria de espaços quasemétricos, são fechadas sob esse construtor.

Utilizando-se a teoria da igualdade local para enfraquecer a noção de estruturas algébricas para estruturas algébricas locais e usando-se o construtor intervalar para desenvolver a estrutura algebrica local, é possível aplicar o construtor intervalar em corpos para obter um corpo de ordem local que generaliza a aritmética de Moore. As definições que seguem podem ser encontradas em [5].

3.1 Conjuntos Parcialmente Ordenados

Definição 3.1.1 (Ordem Parcial) *Seja D um conjunto. Uma relação binária $\leq$ sobre D é uma* **ordem parcial** *sobre D se para cada $x, y, z \in D$ as seguintes condições são satisfeitas:*

Reflexividade: $x \leq x$

Antisimetria: se $x \leq y$ *e* $y \leq x$ *então* $x = y$

Transitividade: se $x \leq y$ *e* $y \leq z$ *então* $x \leq z$

Definição 3.1.2 (Poset) *conjunto D com uma ordem parcial $\leq$ sobre D é um* **conjunto parcialmente ordenado** *ou apenas um* **poset**. *Uma ordem parcial é dita* **total** *se para cada $x, y \in D$, $x \leq y$ ou $y \leq x$. Um* **conjunto totalmente ordenado** *é um conjunto com uma ordem total.*

Definição 3.1.3 (Conjuntos para Cima e para Baixo) *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um poset. Um subconjunto* A *de* D *é um* **um conjunto para cima** *se* $x \in A$ *implica* $y \in A$ *para todo* $x \leq y$. *Denotaremos por* $\uparrow A$ *o conjunto de todos os elementos acima de algum elemento de* A, *ou seja,* $\uparrow A = \{x \in y : y \leq xparaalgumy \in A\}$. *Se não houver perigo de confusão, abreviaremos* $\uparrow \{x\}$ *como* $\uparrow x$. *O dual deste é denominado* **conjunto para baixo** *e corresponde ao conjunto de todos elementos abaixo de algum elemento de* A, *este é denotado por* $\downarrow A$.

Definição 3.1.4 (Majorante e Minorante) *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um poset. Seja* $A \subseteq D$. *Um elemento* $x \in D$ *é chamado um* **majorante** *de* A *se* x *está acima de qualquer elemento de* A. *Usaremos a notação* $A \leq x$, *para este caso e denotaremos* $UB(A)$ *o conjunto de todos os majorantes de* A. *Dualmente, chama-se um* **minorante** *de* A *se* x *está abaixo de qualquer elemento de* A *e* $x \leq A$. *O conjunto de todos os minorantes de* A *é denotado por* $LB(A)$.

Definição 3.1.5 (Maior e Menor Elementos) *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um poset. Se todos os elementos de* D *estão abaixo de um único elemento* $x \in D$, *dizemos que* x *é* **o maior elemento** *ou* **topo**, *denotado por* $\top$. *Dualmente, se todos elementos de* D *estão acima de um único elemento* $x \in D$, *dizemos que* x *é* **o menor elemento** *de um poset, também chamado de* **bottom**, *comumente denotado por* $\bot$.

Definição 3.1.6 (Supremo e Ínfimo) *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um poset. Se num subconjunto* $A \subseteq D$, $UB(A)$ *tem um menor elemento, ele é chamado de* **supremo** *de* A *e denotado por* $\bigsqcup A$. *Em outra direção, se* $LB(A)$ *tem um maior elemento, ele é chamado de* **ínfimo** *de* A *e denotado por* $\bigsqcap A$. *Se* A *é um conjunto finito* $A = \{x_1, \ldots, x_n\}$ *então pode-se escrever* $x_1 \sqcup x_2 \sqcup \ldots \sqcup x_n$ *e* $x_1 \sqcap x_2 \sqcap \ldots \sqcap x_n$, *respectivamente.*

Definição 3.1.7 (Reticulado) *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um poset.* $\mathbf{D}$ *é um* **reticulado** *se para todo subconjunto finito* $A \subseteq D$ *tem-se que* $\bigsqcap A \in D$ *e* $\bigsqcup A \in D$

Definição 3.1.8 (Reticulado Completo) *Um poset* $\mathbf{D} = \langle D, \leq \rangle$ *com maior e menor elementos,* $\top$ *e* $\bot$*, respectivamente, onde para todo subconjunto* $A \subseteq D$, $\bigsqcap A \in D$ *e* $\bigsqcup A \in D$ *chama-se* **reticulado completo**.

Definição 3.1.9 (Função Monotônica) *Sejam* **D** *e* **E** *posets. Uma função* $f : E \longrightarrow D$ *é chamada* **monotônica** *se para cada* $x, y \in D$ *tal que* $x \leq y$*, tem-se que* $f(x) \leq f(y)$.

Definição 3.1.10 (Isomorfismo) *Dois posets,* **D** *e* **E***, estão* **isomorficamente ordenados** *se existe uma função bijetiva e monotônica* $f : D \longrightarrow E$*, tal que* $f^{-1} : E \longrightarrow D$ *(sua inversa) também é monotônica. Neste caso* f *chama-se* **isomorfismo** *entre* D *e* E.

3.2 O Construtor Intervalar

Em 1991, Acióly [3] usou a ordem de informação como base para desenvolver uma fundamentação computacional para a teoria intervalar de Moore.

Assim como a definição de intervalos reais, a ordem de informação sobre o conjunto de intervalos reais depende da ordem usual da reta. Com isso, pode-se generalizar esta construção considerando qualquer conjunto de ordem parcial, ao invés do conjunto dos números reais. Isso significa que pode-se pensar em um conjunto de intervalos como sendo a imagem de um construtor na categoria **POSET**.

Definição 3.2.1 *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um poset. O poset* $\mathbb{I}(\mathbf{D})\top = \langle \mathbb{I}(D)^{\top}, \sqsubseteq \rangle$ [1]*, onde:*

1. $\mathbb{I}(D) = \{[a, b] : a, b \in D \text{ e } a \leq b\}$
2. $\mathbb{I}(D)^{\top} = \mathbb{I}(D) \cup \{\top\}$
3. $[a, b] \sqsubseteq [c, d] \Leftrightarrow a \leq c \leq d \leq b$

[1] *Originalmente o construtor intervalar não considera " $\top$", o topo, o qual pode ser pensado como o "intervalo vazio".*

4. $X \sqsubseteq \top$ *para todo* $X \in \mathbb{I}(D)^{\top}$

é chamado o **poset de intervalos de D**, *onde* $[a, b] = \{x \in D : a \leq x \leq b\}$ *poderia ser pensado como um par ordenado ou como o conjunto de elementos de* D *entre* a *e* b.

Há duas funções naturais de $\mathbb{I}(D)$ para D, que são as projeções esquerda e direita: $l : \mathbb{I}(D) \longrightarrow D$ e $r : \mathbb{I}(D) \longrightarrow D$, respectivamente, definidas por:

$$l([a, b]) = a \tag{3.1}$$

$$r([a, b]) = b \tag{3.2}$$

A função l é monotônica e, portanto, é um morfismo do poset $\mathbb{I}(\mathbf{D})$[2] para o poset $\mathbf{D}$. A função r não é monotônica, mas isto pode ser superado revertendo a ordem sobre $\mathbf{D}$.

Definição 3.2.2 *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um poset. O* **poset oposto** *de* $\mathbf{D}$, *denotado por* $\mathbf{D}^{op}$, *é o par* $\mathbf{D}^{op} = \langle D^{op}, \leq_{op} \rangle$, *onde* $D^{op} = D$ *e* $x \leq_{op} y$ *se* $y \leq x$.

Assim, todo poset $\mathbf{D}$ tem um poset oposto e a função $r : \mathbb{I}(D) \longrightarrow D^{op}$ é monotônica.

Proposição 3.2.1 *Sejam* $\mathbf{D} = \langle D, \leq_D \rangle$ *e* $\mathbf{E} = \langle E, \leq_E \rangle$ *posets. Então* $\mathbb{I}(\mathbf{D} \times \mathbf{E})$ *e* $\mathbb{I}(\mathbf{D}) \times \mathbb{I}(\mathbf{E})$ *são isomórfas.*

Proposição 3.2.2 *Sejam* $\mathbf{D} = \langle D, \leq_D \rangle$ *e* $\mathbf{E} = \langle E, \leq_E \rangle$ *posets. Sejam* $f, g : D \longrightarrow E$ *funções monotônicas tal que* $f \sqsubseteq g$ *(isto é,* $f(x) \leq_E g(x)$ *para cada* $x \in D$*). Então a função* $F : \mathbb{I}(D) \longrightarrow \mathbb{I}(E)$ *definida por*

$$F([a, b]) = [f(a), g(b)] \tag{3.3}$$

é a única função monotônica que comuta o diagrama da figura 3.1.

As provas para as proposições 3.2.1 e 3.2.2 podem ser encontradas em [6].

Teorema 3.2.1 *Seja* $\mathbf{D} = \langle D, \leq \rangle$ *um reticulado. Então* $\mathbb{I}(\mathbf{D})^{\top}$ *é um reticulado.*

[2] $\mathbb{I}(\mathbf{D}) = \langle \mathbb{I}(D), \sqsubseteq \rangle$ onde $\sqsubseteq$ é igual a $\sqsubseteq$ em $\mathbb{I}(\mathbf{D})^{\top}$ restrita a $\mathbb{I}(D)$.

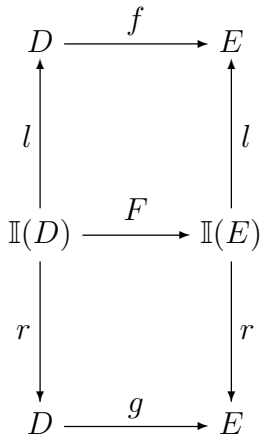

Figura 3.1: Diagrama comutativo para F([a,b])

Prova: Se $[a,b],[c,d] \in \mathbb{I}(D)^{\top}$ então provaremos que:

$$[a,b] \sqcup [c,d] = \begin{cases} [a \sqcup c, b \sqcap d] & \text{se } a \sqcup c \leq b \sqcap d, \\ \top & \text{senão.} \end{cases}$$

e

$$[a,b] \sqcap [c,d] = [a \sqcap c, b \sqcup d].$$

Suponha que $[u,v] \in \mathbb{I}(D)^{\top}$ é um majorante de $[a,b]$ e $[c,d]$, isto é, $[a,b] \sqsubseteq [u,v]$ e $[c,d] \sqsubseteq [u,v]$. Assim $a \leq u$, $c \leq u$, $v \leq b$ e $v \leq d$ e, portanto, $a \sqcup c \leq u$ e $v \leq b \sqcap d$. Como $u \leq v$ então, $a \sqcup c \leq b \sqcap d$. Logo, $[a \sqcup c, b \sqcap d] \sqsubseteq [u,v]$.

Suponha agora que $[u,v] \in \mathbb{I}(D)^{\top}$ é um minorante de $[a,b]$ e $[c,d]$, isto é, $[u,v] \sqsubseteq [a,b]$ e $[u,v] \sqsubseteq [c,d]$. Assim, $u \leq a$, $u \leq c$, $b \leq v$ e $d \leq v$. Portanto $u \leq a \sqcap c$ e $b \sqcup d \leq v$.

3.3 Igualdade Local

Scott [49] propôs uma alternativa de axiomatização para a igualdade visando englobar os objetos definidos parcialmente. Esta axiomatização substitui o axioma da reflexividade, $x = x$, por

$$\textbf{(refl)} x = x \leftrightarrow def(x), \tag{3.4}$$

onde $def(x)$ significa que x está definido.

Os modelos para a teoria da igualdade de Scott são os Ω-conjuntos.

Definição 3.3.1 (Álgebra de Heyting Completa) *Uma* **álgebra de Heyting completa**, *cHa, é um reticulado completo* Ω *que satisfaz a lei da* $\bigvee, \wedge -$ *distributividade; a saber:*

$$p \wedge \bigvee_{j \in J} \{q_j\} = \bigvee_{j \in J} \{p \wedge q_j\} \tag{3.5}$$

para todo $p \in \Omega$ *e* $\{q_j : j \in J\} \subseteq \Omega$.

Definição 3.3.2 (Ω-conjuntos) *Dada uma algébra de Heyting completa* Ω, *um* **Ω-conjunto** *é uma estrutura da forma:* $\langle A, [\![. = .]\!] : A \times A \to \Omega, [\![def]\!] : A \to \Omega \rangle$, *onde:*

(i) $[\![x = x]\!] = [\![def(x)]\!](refl)$

(ii) $[\![x = y]\!] = [\![y = x]\!](simetria)$

(iii) $[\![x = y]\!] \wedge [\![y = z]\!] \leq [\![x = z]\!](transitividade)$

Sobre qualquer Ω-conjunto é possível definir a relação de equivalência $[\![. \equiv .]\!] : A \times A \to \Omega$ da seguinte forma [3]:

$$[\![x \equiv y]\!] = (([\![def(x)]\!] \vee [\![def(y)]\!]) \Rightarrow [\![x = y]\!]). \tag{3.6}$$

Santiago estendeu a igualdade simples de Scott de tal forma que a consistência se torna uma relação de equivalência mais fraca [44].

A **teoria da igualdade local**, $\overset{loc}{=}$, estende a teoria da igualdade de Scott, acrescentando os seguintes axiomas:

(i) $x \overset{loc}{=} x \leftrightarrow [\![def(x)]\!]$ (refl)

[3] $\Rightarrow$ é definida canonicamente na álgebra de Heyting completa. Ver [42, 41]

(ii) $x \overset{loc}{=} y \to y \overset{loc}{=} x$ (simetria)

(iii) $[\![def(x \sqcup z)]\!] \to (x \overset{loc}{=} y \wedge y \overset{loc}{=} z \to x \overset{loc}{=} z)$ (transitividade local)

(iv) $x \sqcup x \equiv x$

(v) $x \sqcup y \equiv y \sqcup x$

(vi) $x \sqcup (y \sqcup z) \equiv (x \sqcup y) \sqcup z$

Onde $\sqcup$ é um símbolo funcional binário.

Os modelos para a teoria da igualdade local são os conjuntos locais definidos a seguir.

Definição 3.3.3 (Conjunto Local) *Os* **conjuntos locais** *são Ω-conjuntos acrescidos das operações $[\![. \overset{loc}{=} .]\!] : A \times A \to \Omega$ e $\sqcup : A \times A \to A$, onde $[\![. \overset{loc}{=} .]\!]$ e $\sqcup$ têm as seguintes propriedades:*

(i) $[\![x \overset{loc}{=} x]\!] \Leftrightarrow def(x)$

(ii) $[\![x \overset{loc}{=} y]\!] \Rightarrow [\![x \overset{loc}{=} y]\!]$

(iii) $[\![def(x \sqcup z)]\!] \Rightarrow ([\![x \overset{loc}{=} y]\!] \wedge [\![y \overset{loc}{=} z]\!] \Rightarrow [\![x \overset{loc}{=} z]\!])$

(iv) $[\![a \sqcup a \equiv a]\!] = \top$

(v) $[\![a \sqcup b \equiv b \sqcup a]\!] = \top$

(vi) $[\![a \sqcup (b \sqcup c) \equiv (a \sqcup b) \sqcup c]\!] = \top$

onde $\top$ é o topo de Ω.

A relação $[\![. \overset{loc}{=} .]\!]$ pode ser definida canonicamente a partir de um Ω-conjunto com uma operação $\sqcup : A \times A \to A$ que satisfaz $(iv) - (vi)$ da seguinte forma:

$$[\![x \overset{loc}{=} y]\!] = [\![def(x \sqcup y)]\!] \tag{3.7}$$

Note que esta não é a única maneira de se obter canonicamente esta relação e esta operação a partir de um Ω-conjunto, o importante é que ela satisfaça os seis axiomas da teoria da igualdade local que venha a ser um modelo da teoria da igualdade local.

Proposição 3.3.1 *Seja $\mathcal{A}$ um Ω-conjunto e $\sqcup : A \times A \to A$ satisfazendo (iv-vi) da definição 3.3.3. Então o conjunto local canonicamente obtido como acima satisfaz os seis axiomas da teoria da igualdade local.*

Prova: Trivialmente os axiomas (iv) - (vi) são satisfeitos.

(i) $[\![x \overset{loc}{=} x]\!] = [\![def(x \sqcup x)]\!] = [\![def(x)]\!]$

(ii) $[\![x \overset{loc}{=} y]\!] = [\![def(x \sqcup y)]\!] = [\![def(y \sqcup x)]\!] = [\![y \overset{loc}{=} x]\!]$

(iii) Como $[\![def(x \sqcup z)]\!] = [\![x \overset{loc}{=} z]\!]$, então o axioma (iii) da teoria da igualdade local é uma tautologia da forma $\alpha \Rightarrow (\beta \Rightarrow \alpha)$. [4]

Santiago em [47] construiu um modelo intuicionista para a igualdade local. Entretanto a lógica subjacente de CASL, apesar de parcial, é clássica, e isso faz com que seja necessário definir um modelo booleano para que se possa utilizar a igualdade local em CASL, sendo assim propõe-se a seguinte definição.

Definição 3.3.4 *Sejam* **D** *um reticulado e $\Omega = \{0, 1\}$ a álgebra de Heyting completa booleana. Definem-se as operações $[\![def]\!] : \mathbb{I}(D)^{\top} \to \Omega$ e $[\![=]\!] : \mathbb{I}(D)^{\top} \times \mathbb{I}(D)^{\top} \to \Omega$ da seguinte forma:*

(i) $[\![def(X)]\!] = 1$ se e somente se $X \in \mathbb{I}(D)$,

[4] Observe que a igualdade nas provas acima, $[\![x \overset{loc}{=} y]\!] = [\![def(x \sqcup x)]\!] = [\![def(x)]\!]$ interpreta a equivalência dos axiomas na álgebra de Heyting completa Ω. Analogamente o $\leq$ interpreta a implicação.

(ii) $[\![X = Y]\!] = 1$ *se e somente se* $[\![def(X)]\!] = 1 \wedge [\![def(Y)]\!] = 1 \wedge X \sqsubseteq Y \wedge Y \sqsubseteq X$

Proposição 3.3.2 $[\![. = .]\!]$ *é uma igualdade simples para* $\mathbb{I}(D)$. $\square$

Prova: É suficiente mostrar que a igualdade intervalar satisfaz o axioma (refl) já que é sabido que ele é simétrico e transitivo. Assim,

$$[\![X = X]\!] = 1 \text{ se e somente se } [\![def(X)]\!] = 1 \wedge X \sqsubseteq X \text{ se e somente se } [\![def(X)]\!] = 1 \quad (3.8)$$

e portanto os intervalos munidos desta igualdade são um Ω-conjunto.

Definição 3.3.5 (Igualdade Local Intervalar) *Sejam* $\Omega = \{0, 1\}$ *a álgebra de Heyting completa booleana e* **D** *um reticulado. A* **igualdade local intervalar** *é a função* $[\![. \overset{loc}{=} .]\!] : \mathbb{I}(D)^{\top} \times \mathbb{I}(D)^{\top} \to \Omega$ *definida por*

$$[\![X \overset{loc}{=} Y]\!] = 1 \text{ se e somente se } [\![def(X \cap Y)]\!] = 1 \quad (3.9)$$

onde, $\cap : \mathbb{I}(D)^{\top} \times \mathbb{I}(D)^{\top} \to \mathbb{I}(D)^{\top}$ *é definida por:*

(i) $\top \cap X = \top$

(ii) $X \cap \top = \top$

(iii) $[a, b] \cap [c, d] = [a \sqcup c, b \sqcap d]$ *se* $a \sqcup c \leq b \sqcap d$

(iv) $[a, b] \cap [c, d] = \top$ *se* $(a \sqcup c) > (b \sqcap d)$

Proposição 3.3.3 *Seja* **D** *um reticulado. Então* $\langle \mathbb{I}(D), [\![. = .]\!], def, [\![. \overset{loc}{=} .]\!], \cap \rangle$ *é um conjunto local.*

3.3.1 Álgebras Localmente Ordenadas

Usando a teoria da igualdade local é possível enfraquecer as condições de estruturas algébricas, substituindo a igualdade clássica pela igualdade local, ou seja, pode-se enfraquecer as noções clássicas de várias estruturas algébricas, tais como grupos, anéis e corpos considerando a igualdade local ao invés da igualdade usual.

Por simplicidade notacional escreveremos $t \overset{loc}{=} s$ em vez de $[\![t \overset{loc}{=} s]\!] = 1$, como também, que $\langle D, [\![. = .]\!], [\![def]\!], [\![. \overset{loc}{=} .]\!], \cap \rangle$ é um conjunto local e diremos, simplesmente, que $\overset{loc}{=}$ é uma igualdade local sobre D.

Definição 3.3.6 *Seja $\overset{loc}{=}$ uma igualdade local sobre um conjunto D e duas operações binárias, $+$ e $*$ sobre D. Considerando as seguintes propriedades:*

1. $a + (b + c) \overset{loc}{=} (a + b) + c$ *para cada* $a, b, c \in D$

2. *Existe um único elemento* $0 \in D$ *tal que para cada* $a \in D$, $a + 0 \overset{loc}{=} a \overset{loc}{=} 0 + a$

3. *Para cada* $a \in D$, *existe um único elemento* $(-a) \in D$ *tal que se* $a + b \overset{loc}{=} 0 \overset{loc}{=} b + a$ *para algum* $b \in D$, *então* $b \overset{loc}{=} -a$

4. $a + b \overset{loc}{=} b + a$ *para cada* $a, b \in D$

5. $a * (b + c) \overset{loc}{=} (a * b) + (a * c)$ *para cada* $a, b, c \in D$

6. $a * (b * c) \overset{loc}{=} (a * b) * c$ *para cada* $a, b, c \in D$

7. *Existe um único elemento* $1 \in D$ *tal que para cada* $a \in D$, $a * 1 \overset{loc}{=} a \overset{loc}{=} 1 * a$

8. $a * b \overset{loc}{=} b * a$ *para cada* $a, b \in D$

9. *Para cada* $a \in D$ *não localmente igual a* 0, *existe um único elemento* $a^{-1} \in D$ *tal que* $a * a^{-1} \overset{loc}{=} 1 \overset{loc}{=} a^{-1} * a$ *e para algum* $b \in D$ *tem-se que* $a * b \overset{loc}{=} 1 \overset{loc}{=} b * a$, $b \overset{loc}{=} a^{-1}$

A estrutura $\langle D, \overset{loc}{=}, + \rangle$ *é um* **grupo local** *se ela satisfaz (1), (2) e (3). É um* **grupo abeliano local** *se ela é um grupo local e satisfaz (4). A estrutura* $\langle D, \overset{loc}{=}, +, * \rangle$ *é um* **anel local** *se* $\langle D, \overset{loc}{=}, + \rangle$ *é um grupo abeliano local e satisfaz (5) e (6). É um* **anel comutativo local** *se ela é um anel local e satisfaz (8). É um* **anel comutativo local com unidade** *se ela é um anel comutativo local e satisfaz (7). É um* **corpo local** *se satisfaz as propriedades de (1) até (9). Se* $\langle D, \overset{loc}{=}, + \rangle$ *é uma dessas álgebras mas não satisfaz a propriedade (6) ela é chamada anel local* **não-associativo**.

Definição 3.3.7 **D**= $\langle D, \leq, \overset{loc}{=}, + \rangle$ *é dito um* **grupo local ordenado** *(respectivamente um grupo abeliano local ordenado) se*

1. $\langle D, \leq \rangle$ *é um reticulado*
2. $\langle D, \overset{loc}{=}, + \rangle$ *é um grupo local (respectivamente um grupo abeliano local)*
3. $+$ *é monotônico em relação a* $\leq$*, isto é se* $a \leq c$ *e* $b \leq d$ *então* $a + b \leq c + d$

Definição 3.3.8 **D**= $\langle D, \leq, \overset{loc}{=}, + \rangle$ *é dito um* **anel local ordenado** *(respectivamente um anel comutativo, anel comutativo com unidade, corpo) se*

1. $\langle D, \leq \rangle$ *é um reticulado*
2. **D**= $\langle D, \overset{loc}{=}, +, * \rangle$ *é um anel local (respectivamente um anel comutativo, anel comutativo com unidade, corpo)*
3. $+$ *é monotônico em relação a* $\leq$*, isto é se* $a \leq c$ *e* $b \leq d$ *então* $a + b \leq c + d$
4. *Dado* $a \in D$*, a satisfaz ou:*

   (i) $\forall x, y \in D$*, tal que* $x \leq y$*,* $a' * x \leq a' * y$ *e* $x * a' \leq y * a'$ $\forall a' \in D$ *tal que* $a \leq a'$ *ou*

   (ii) $\forall x, y \in D$*, tal que* $x \leq y$*,* $a' * y \leq a' * x$ *e* $y * a' \leq x * a'$ $\forall a' \in D$ *tal que* $a' \leq a$

Lema 3.3.1 *Seja* $\mathbf{D} = \langle D, \leq, \overset{loc}{=}, +, * \rangle$ *um anel local ordenado.* $a \overset{loc}{=} b$ *se, e somente se, para cada* $c \in D$, $a * c \overset{loc}{=} b * c$.

Definição 3.3.9 *Sejam* $\mathbf{D}= \langle D, \leq_D, \overset{loc}{=}_D, +_D, *_D \rangle$ *e* $\mathbf{E} = \langle E, \leq_E, \overset{loc}{=}_E, +_E, *_E \rangle$ *anéis locais ordenados. Uma função* $f : D \longrightarrow E$ *é um* **homomorfismo local** *se* f *satisfaz as seguintes propriedades:*

1. *se* $a \overset{loc}{=}_D b$ *então* $f(a) \overset{loc}{=}_E f(b)$
2. f *é monotônica*
3. $f(a +_D b) \overset{loc}{=}_E f(a) +_E f(b)$
4. $f(a *_D b) \overset{loc}{=}_E f(a) *_E f(b)$

Definição 3.3.10 *Sejam* $\mathbf{D}= \langle D, \leq_D, \overset{loc}{=}_D, +_D, *_D \rangle$ *e* $\mathbf{E} = \langle E, \leq_E, \overset{loc}{=}_E, +_E, *_E \rangle$ *anéis locais ordenados. Um homomorfismo local* $f : D \longrightarrow E$ *é dito um* **isomorfismo local** *se* f *é bijetiva e* f^{-1} *também é um homomorfismo local.*

Lema 3.3.2 *Seja* $\mathbf{D}= \langle D, +, * \rangle$ *um anel e* $\leq$ *uma ordem parcial sobre* D *tal que* $\langle D, \leq \rangle$ *é um reticulado. Define-se sobre* $\mathbb{I}(\mathbf{D})$ *as seguintes operações:*

$$[a, b] \oplus [c, d] = [a + c, b + d] \tag{3.10}$$

$$[a, b] \otimes [c, d] = [\bigsqcap A, \bigsqcup A] \tag{3.11}$$

onde $A = \{a * c, a * d, b * c, b * d\}$. *Então* $\oplus$ *e* $\otimes$ *estão bem definidas.*

Teorema 3.3.1 *Seja* $\mathbf{D}= \langle D, \leq, =, +, * \rangle$ *um anel ordenado (respectivamente um anel comutativo, anel comutativo com unidade, corpo) tal que* $\langle D, \leq \rangle$ *é reticulado. Então* $\mathbb{I}(\mathbf{D})= \langle \mathbb{I}(D), \sqsubseteq, [\![. \overset{loc}{=} .]\!], \oplus, \otimes \rangle$*; onde*

$$[a,b] \overset{loc}{=} [c,d] \Leftrightarrow def([a,b] \cap [c,d]) \tag{3.12}$$

é um anel local ordenado (respectivamente um anel local comutativo, anel local comutativo com unidade, corpo).

Prova: Supondo que $\langle D, \leq, =, +, * \rangle$ é um anel ordenado onde $(D, \leq)$ é um reticulado. Devemos provar que $\langle \mathbb{I}(D), \sqsubseteq, \overset{loc}{=}, \oplus, \otimes \rangle$ é um anel local ordenado. É direto provar que $\langle \mathbb{I}(D), \sqsubseteq, \overset{loc}{=}, \oplus \rangle$ satisfaz as propriedades (1) e (4) da definição 3.3.6. Claramente $\mathbf{0} = [0,0]$ satisfaz $[a,b] \oplus \mathbf{0} \overset{loc}{=} [a,b] \overset{loc}{=} \mathbf{0} \oplus [a,b]$ para cada $[a,b] \in \mathbb{I}(D)$.

Supondo que $[x,y] \in \mathbb{I}(D)$ satisfaz $[a,b] \oplus [x,y] \overset{loc}{=} [a,b] \overset{loc}{=} [x,y] \oplus [a,b]$ para cada $[a,b] \in \mathbb{I}(D)$ então, em particular, quando $[a,b] = [0,0]$ tem-se que $[0,0] \oplus [x,y] \overset{loc}{=} [0,0]$. Assim, $[x,y] \overset{loc}{=} [0,0]$. Com isso, $\mathbf{0}$ satisfaz a propriedade (2).

A seguir a prova da propriedade (3) da definição 3.3.6 para $\mathbb{I}(D)$.

Dado $[a,b] \in \mathbb{I}(D)$ é claro que $[a,b] \oplus [-b,-a] \overset{loc}{=} [0,0]$ and $[-b,-a] \oplus [a,b] \overset{loc}{=} [0,0]$.

Isso prova-se que se $[x,y] \in \mathbb{I}(D)$ satisfaz $[a,b] \oplus [x,y] \overset{loc}{=} [0,0] \overset{loc}{=} [x,y] \oplus [a,b]$
então $[x,y] \overset{loc}{=} [-b,-a]$.

De fato,

$$\begin{aligned} [a,b] \oplus [x,y] &= [a+x, b+y] \\ &= [x+a, y+b] \\ &= [x,y] \oplus [a,b] \end{aligned} \tag{3.13}$$

Se $[x+a, y+b] \overset{loc}{=} [0,0]$ então, $x+a \leq 0$ e $0 \leq y+b$ ou, equivalentemente, $x \leq -a$ e $-b \leq y$.

Seja $z = \bigsqcup\{-b, x\}$. Uma vez que $x \leq y$ e $-b \leq y$ pode-se ter que $z \leq y$ e, claramente, $x \leq z$. Assim, $[x,y] \sqsubseteq [z,z]$. Por outro lado, já que $-b \leq -a$ e $x \leq -a$ tem-se que $z \leq -a$. Mas $-b \leq z$. Então, $[-b,-a] \sqsubseteq [z,z]$, logo $[x,y] \overset{loc}{=} [-b,-a]$.Assim, $\langle \mathbb{I}(D), \sqsubseteq, \overset{loc}{=}, \oplus \rangle$ é um grupo abeliano local.

A seguir a prova da propriedade (5) da definição 3.3.6 para $\mathbb{I}(D)$.

Seja $[a,a'],[b,b'],[c,c'] \in \mathbb{I}(D)$. Por definição tem-se que

$$[a,a'] \otimes ([b,b'] \oplus [c,c']) = [\bigsqcap A, \bigsqcup A] \tag{3.14}$$

onde $A = \{a*b + a*c, a*b' + a*c', a'*b + a'*c, a'*b' + a'*c'\}$. Por outro lado, tem-se que

$$([a,a'] \otimes [b,b']) \oplus ([a,a'] \otimes [c,c']) = [\bigsqcap C + \bigsqcap D, \bigsqcup C + \bigsqcup D] \tag{3.15}$$

onde $C = \{a*b, a*b', a'*b, a'*b'\}$ e $D = \{a*c, a*c', a'*c, a'*c'\}$.

Note que, pela monoticidade da soma em D, tem-se que

$$\bigsqcap C + \bigsqcap D \leq \bigsqcap A \leq \bigsqcup A \leq \bigsqcup C + \bigsqcup D$$

Assim, $[\bigsqcap C + \bigsqcap D, \bigsqcup C + \bigsqcup D] \sqsubseteq [\bigsqcap A, \bigsqcup A]$. Então,

$$[\bigsqcap C + \bigsqcap D, \bigsqcup C + \bigsqcup D] \stackrel{loc}{=} [\bigsqcap A, \bigsqcup A],$$

que prova a propriedade (5).

A seguir a prova da propriedade (6) para $\mathbb{I}(D)$.

Seja $[a,a'],[b,b'],[c,c'] \in \mathbb{I}(D)$. Por definição, tem-se que $[b,b'] \otimes [c,c'] = [x,y]$ onde $x = \bigsqcap\{b*c, b*c', b'*c, b'*c'\}$ e $y = \bigsqcup\{b*c, b*c', b'*c, b'*c'\}$. Além disso,

$$[a,a'] \otimes ([b,b'] \otimes [c,c']) = [\bigsqcap A, \bigsqcup A] \tag{3.16}$$

onde $A = \{a*x, a*y, a'*x, a'*y\}$. Analogamente, $[a,a'] \otimes [b,b'] = [z,w]$ onde $z = \bigsqcap\{a*b, a*b', a'*b, a'*b'\}$ e $w = \bigsqcup\{a*b, a*b', a'*b, a'*b'\}$

Além disso,

$$([a,a'] \otimes [b,b']) \otimes [c,c'] = [\bigsqcap B, \bigsqcup B] \tag{3.17}$$

onde $B = \{z*c, z*c', w*c, w*c'\}$

Seja $C = \{a*b*c, a*b*c', a*b'*c, a*b'*c', a'*b*c, a'*b*c', a'*b'*c, a'*b'*c'\}$. Há vários casos a serem analisados. Pode-se apenas provar a propriedade (6) em um caso especial, os outros casos são análogos. Então, supondo que a e c satisfazem a propriedade (4)(i) da definição 3.3.7. Neste caso, será provado que

$$\bigsqcap A \leq \bigsqcap C \leq \bigsqcup C \leq \bigsqcup A$$

De fato, já que $x = \bigsqcap\{b * c, b * c', b' * c, b' * c'\}$ tem-se que

$$a * x \leq \bigsqcap\{a * b * c, a * b * c', a * b' * c, a * b' * c'\}.$$

Por outro lado, já que $a \leq a'$ e a satisfazem a propriedade (4)(i), tem se que

$$a' * x \leq \bigsqcap\{a' * b * c, a' * b * c', a' * b' * c, a' * b' * c'\}.$$

Então,

$$\bigsqcap A \leq (a * x \sqcap a' * x) \leq \bigsqcap C$$

Analogoamente prova-se que $\bigsqcup C \leq \bigsqcup A$. Já que

$$\bigsqcap A \leq \bigsqcap C \leq \bigsqcup C \leq \bigsqcup A$$

tem-se que $[\bigsqcap A, \bigsqcup A] \sqsubseteq [\bigsqcap C, \bigsqcup C]$.

Analogamente, pode-se provar que $[\bigsqcap B, \bigsqcup B] \sqsubseteq [\bigsqcap C, \bigsqcup C]$. Assim,

$$[\bigsqcap A, \bigsqcup A] \stackrel{loc}{=} [\bigsqcap B, \bigsqcup B]$$

como se queira, provou-se (6).

Propriedade (7) da definição 3.3.6 segue as propriedades (2) com $\mathbf{1} = [1, 1]$.

Propriedade (8) desta definição é direta.

Contudo se $\langle D, \leq, +, * \rangle$ é um anel comutativo ordenado com unidade então $\langle \mathbb{I}(D), \sqsubseteq, \stackrel{loc}{=}, \oplus, \otimes \rangle$ é um anel comutativo ordenado local com unidade.

Supondo que $\langle D, \leq, +, * \rangle$ é um corpo ordenado. Deve-se provar que $\langle \mathbb{I}(D), \sqsubseteq, \stackrel{loc}{=}, \oplus, \otimes \rangle$ satisfaz a propriedade (9) da definição 3.3.6.

Seja $[a, b] \in \mathbb{I}(D)$ tal que $[a, b]$ não é localmente igual a $[0, 0]$. Seja $[c, d] = [a^{-1} \sqcap b^{-1}, a^{-1} \sqcup b^{-1}]$. Prova-se primeiro

$$[a, b] \otimes [c, d] \stackrel{loc}{=} [1, 1] \stackrel{loc}{=} [c, d] \otimes [a, b] \tag{3.18}$$

Supondo primeiro que a que satisfaz (4).(i) da definição 3.3.7. Por definição

$$[a,b]\otimes[c,d]=[\bigsqcap\{a*c,a*d,b*c,b*d\},\bigsqcup\{a*c,a*d,b*c,b*d\}] \tag{3.19}$$

Uma vez que $c=a^{-1}\sqcap b^{-1}$ e $d=a^{-1}\sqcup b^{-1}$ temos que $c\leq a^{-1}$, $c\leq b^{-1}$, $a^{-1}\leq d$ e $b^{-1}\leq d$. Então, já que a e b satisfazem a propriedade (4).(i) da definição 3.3.7 temos que

$a*c\leq a*a^{-1}=1$, $b*c\leq b*b^{-1}=1$, $1=a^{-1}*a\leq a*d$ and $1=b^{-1}*b\leq b*d$.

Então,

$$\begin{aligned}[a,b]\otimes[c,d] &= [a*c\sqcap b*c, a*d\sqcup b*d]\\ &\sqsubseteq [1,1]\end{aligned} \tag{3.20}$$

Assim $[a,b]\otimes[c,d]\overset{loc}{=}[1,1]$ se a satisfaz (4).(i).

Supondo que b satisfaz (4).(ii) da definição 3.3.7. Neste caso a também satisfaz (4).(ii).

Assim, conclui-se que

$$\begin{aligned}[a,b]\otimes[c,d] &= [a*d\sqcap b*d, a*c\sqcup b*c]\\ &\sqsubseteq [1,1]\end{aligned} \tag{3.21}$$

Então $[a,b]\otimes[c,d]\overset{loc}{=}[1,1]$ se b satisfaz (4).(ii).

Supondo que a satisfaz (4).(ii) e b satisfaz (4).(i). Neste caso conclui-se

$$\begin{aligned}[a,b]\otimes[c,d] &= [b*c\sqcap a*d, a*c\sqcup b*d]\\ &\sqsubseteq [1,1]\end{aligned} \tag{3.22}$$

Então $[a,b]\otimes[c,d]\overset{loc}{=}[1,1]$ neste caso.

Resta provar que se $[x,y]\in\mathbb{I}(D)$ é tal que $[a,b]\otimes[x,y]\overset{loc}{=}[1,1]\overset{loc}{=}[x,y]\otimes[a,b]$ então $[x,y]\overset{loc}{=}[c,d]$. Assim, $[a,b]\otimes[c,d]\overset{loc}{=}[1,1]$ e $[a,b]\otimes[x,y]\overset{loc}{=}[1,1]$. Então, por $[1,1]$ ser degenerado $[a,b]\otimes[c,d]\overset{loc}{=}[a,b]\otimes[x,y]$. Contudo, pelo lema 3.3.1, $[c,d]\overset{loc}{=}[x,y]$.□

Note que:

1. Se $[a,b]$ e $[c,d]$ são intervalos degenerados (isto é, $a=b$ e $c=d$), então $[a,b]\oplus[c,d]$ e $[a,b]\otimes[c,d]$ são intervalos degenerados $[a+c,a+c]$ e $[a*c,a*c]$, respectivamente.

2. Se $x \in [a,b]$ (isto é $a \leq x \leq b$) e $y \in [c,d]$ então $x+y \in [a,b] \oplus [c,d]$ e $x*y \in [a,b] \otimes [c,d]$.

3. A propriedade (4) da definição 3.3.7 foi usada para provar a associatividade do produto no teorema 3.3.1. Contudo, se $\mathbf{D} = \langle D, \leq, =, +, * \rangle$ é um anel ordenado (e também um anel comutativo ordenado, anel comutativo ordenado com unidade, corpo ordenado) que não satisfaz a propriedade (4) da definição 3.3.7 pode-se apenas garantir que $\mathbb{I}(\mathbf{D}) = \langle \mathbb{I}(D), \sqsubseteq, \stackrel{loc}{=}, \oplus, \otimes \rangle$ é um anel local ordenado não-associativo (e também um anel comutativo ordenado não-associativo, anel comutativo ordenado não-associativo, corpo ordenado não-associativo).

As definições e teoremas para grupos e grupos abelianos podem ser obtidas de maneira análoga a definição 3.3.8 e o teorema 3.3.1.

3.4 A Estrutura Algébrica Local com Matriz Intervalar Real

A aplicação seguinte mostra que um conjunto de matrizes intervalares real é um anel local ordenado não-associativo, recuperando contudo as propriedades bem conhecidas das matrizes intervalares real [30].

Proposição 3.4.1 *Seja $\Re = \langle \mathbb{R}, \leq \rangle$ o poset de números reais com a ordem usual. $\Re$ é um reticulado.*

Prova: Para todo $X \subseteq \mathbb{R}$ finito tem-se que $\bigsqcup X = max(X)$ e $\bigsqcap X = min(X)$.$\Box$

Um vez que $\langle \mathbb{R}, \leq, =, +, * \rangle$, onde $=$, $+$ e $*$ são a igualdade, adição e produto comuns, respectivamente é um corpo ordenado, então pelo teorema 3.2.1, $\langle \mathbb{I}(\mathbb{R}), \sqsubseteq, \stackrel{loc}{=}, \oplus, \otimes \rangle$ definido de acordo com o lema 3.3.2 e o teorema 3.3.1, é um corpo local ordenado. Note que $\langle \oplus e \otimes \rangle$ nesse caso coincidem com as definições 2.2.1 e 2.2.4, respectivamente.

Definição 3.4.1 *Sejam* $\mathsf{A} = (a_{ij})$ *e* $\mathsf{B} = (b_{ij})$ *matrizes de números reais de ordem* $m \times n$. *Pode-se dizer que* $\mathsf{A} \sqsubseteq \mathsf{B}$ *se* $b_{ij} \leq a_{ij}$ *para cada* $i, j = 1, \ldots, m$. *Analogamente,* $\mathsf{A} = \mathsf{B}$ *se* $a_{ij} = b_{ij}$ *para cada* $i, j = 1, \ldots, m$.

Proposição 3.4.2 *Seja* $\mathsf{M}_{m\times n}(\mathbb{R})$ *o conjunto de todas as matrizes de números reais de ordem* $m \times n$. *Então* $\langle \mathsf{M}_{m\times n}(\mathbb{R}), \sqsubseteq \rangle$ *é um reticulado.*

Prova: É direta das propriedades dos números reais. □

Definição 3.4.2 *Seja* $\mathsf{A} = (a_{ij})$ *e* $\mathsf{B} = (b_{ij})$ *em* $\mathsf{M}_{m\times n}(\mathbb{R})$. *Define a* **adição** *de* A *e* B, *denotada por* $\mathsf{A} + \mathsf{B}$, *como a matriz* $\mathsf{C} = (c_{ij})$ *cujos elementos são dados por* $c_{ij} = a_{ij} + b_{ij}$.

Definição 3.4.3 *Seja* $\mathsf{A} = (a_{ij})$ *e* $\mathsf{B} = (b_{ij})$ *em* $\mathsf{M}_{m\times n}(\mathbb{R})$. *Define o* **produto** *de* A *e* B, *denotado por* $\mathsf{A} * \mathsf{B}$, *como a matriz* $\mathsf{C} = (c_{ij})$ *cujos elementos são dados por* $c_{ij} = \Sigma_{k=1}^{m} c_{ik} b_{kj}$.

Proposição 3.4.3 $\langle \mathsf{M}_{m\times n}(\mathbb{R}), \sqsubseteq, =, +, * \rangle$ *é um anel ordenado com unidade que não satisfaz a propriedade (4) da definição 3.3.8.*

Prova: Ver [25]. □

Corolário 3.4.1 $\langle \mathbb{I}(M_{m\times n}(\mathbb{R})), \sqsubseteq, \stackrel{loc}{=}, \oplus, \otimes \rangle$ *é um anel local ordenado não-associativo com unidade.*

Prova: Segue da prova do teorema 3.3.1. □

Usando o construtor intervalar definido no lema 3.3.2 obtem-se o conjunto de intervalos de matrizes de números reais. Por exemplo, se A e B são matrizes tal que $\mathsf{A} \sqsubseteq \mathsf{B}$ então $[\mathsf{A}, \mathsf{B}]$ é o intervalo destas matrizes. Para Moore, uma matriz intervalar é uma matriz cujos elementos são intervalos. Todavia, claramente, cada intervalo de matrizes $[\mathsf{A}, \mathsf{B}]$ pode ser entendido como a matriz intervalar $([A_{ij}, B_{ij}])$. Em outras palavras tem-se um isomorfismo local entre $\mathbb{I}(M_{m\times n}(\mathbb{R}))$ e $M_{m\times n}(\mathbb{I}(\mathbb{R}))$.

3.5 Considerações Finais do Capítulo

Neste capítulo vimos que a união de posets bi-fortemente consistentemente completos com a teoria da igualdade local permite enfraquecer propriedades algébricas de certas álgebras. Com isso, possibilitar a aplicação do construtor intervalar, ou seja, que o tipo intervalo paramétrico possa adquirir as propriedades algébricas dos seus tipos passados como parâmetros.

Esse construtor será especificado na linguagem de especificação algébrica chamada CASL, a qual vai ser introduzida no próximo capítulo.

Capítulo 4

CASL e HOL-CASL

As linguagens de especificações algébricas desenvolvidas nas décadas de 70 e 80 estão baseadas em diversos fundamentos algébricos básicos. Apesar de existirem muitas linguagens similares não havia um framework comum o que impediu a disseminação e aplicação de resultados de pesquisa em especificações algébricas. Em 1995 surgiu uma iniciativa internacional para projetar um framework comum para trabalhar com especificações algébricas, o CoFI (*International Common Framework Initiative*) que objetiva obter um acordo comum entre os membros da comunidade de especificações algébricas a respeito dos conceitos básicos, das linguagens em vários níveis, metodologia de desenvolvimento e suporte de ferramentas [32] e, consequentemente, poder disseminar e aplicar os resultados de pesquisa em especificações algébricas.

O grupo CoFI desenvolveu a linguagem de especificação algébrica CASL (*Commom Algebraic Specification Language*). Esta linguagem reune características já encontradas em várias linguagens existentes como subsorts, funções parciais, lógica de primeira ordem e especificações estruturadas e arquiteturais.

CASL consiste de várias partes importantes que podem ser entendidas e usadas separadamente. Estas partes estão associadas a como se constrói uma especificação bem como ela pode ser reutilizada. Esta divisão de CASL é vista como **especificações básicas**, **especificações estruturadas**, **especificações arquiteturais** e **especificações de biblioteca**.

4.1 Especificações Básicas

Uma especificação básica de CASL denota uma classe de modelos CASL, que são estruturas de primeira ordem parcial poli-sortida: álgebras onde são permitidas funções parcias ou totais e predicados. Estas álgebras são classificadas por assinaturas que listam nomes dos sorts, de funções e predicados juntamente com os seus perfís.

A especificação básica de CASL inclui declarações, para introduzir componentes de assinaturas, e axiomas para dar propriedades a essas estruturas que são consideradas os modelos da especificação. Os axiomas são escritos em lógica de primeira ordem e construídos sobre fórmulas atômicas que incluem igualdades forte e existencial, definibilidade de fórmulas e aplicações de predicados.

Em CASL um mesmo símbolo pode ser declarado com vários perfís numa especificação, a chamada sobrecarga de símbolo. Quando ocorre a sobrecarga de símbolo, o contexto é quem determina o perfil do mesmo. Isso pode ser visto, por exemplo, quando declaramos o símbolo "+" como uma operação sobre naturais, inteiros, reais e strings. Ver exemplo na figura 4.1, quando define-se a adição nos inteiros utilizando o símbolo "+".

```
...
forall m,n,r,s: Int; a,b,c,d: Nat
.(a - b) + (c - d) = (a + c) - (b + d)
...
```

Figura 4.1: Sobrecarga de símbolo

O subsort é adequado para declarar um sort como um subsort de outro quando os valores do subsort são considerados no outro sort. Isso acontece, por exemplo na figura 4.2, quando declara-se os inteiros ("Int") considerando valores pertencentes ao sort dos naturais ("Nat").

Especificações de tipos de dados com construtores e seletores são frequentemente necessários. Existem construtores especiais para declaração de tipos de dados que podem ser *loose*, onde todos os modelos são permitidos; *generated*, onde são tomados apenas os modelos gerados pelos construtores, porém os mesmos dados podem ser construídos de diferentes formas e *free*, onde são permitidos os modelos onde os sorts são declarados livremente pelos

```
...
sort  Nat < Int
   forall a: Nat
   .  a = a - 0
...
```

Figura 4.2: Subsort

construtores.

Os construtores de CASL para especificações básicas são: declarações e definições (de sorts, operações, predicados e tipos de dados), restrições na geração de sorts e axiomas (envolvendo declaraçõ de variáveis, quantificadores, conectivos, fórmulas atômicas e termos).

- **Sort**

  A declaração de sort pode ser feita junto com outros sorts bem como sendo um subsort (usando "$<$"). Os valores de um sort podem ser definidos através de uma fórmula.

```
sorts Elem, List
sorts Nat, Neg < Int
sort Pos = {n : Nat . n > 0}
```

Figura 4.3: Declarações de sort

- **Operações**

  As operações podem ser declaradas como total(usando "$\rightarrow$") ou parcial (usando "$\rightarrow ?$") e, a elas, podem ser atribuídas as propriedades de associatividade (**assoc**), comutatividade (**comm**), unidade (**unit**). Ver exemplos na figura 4.4, onde estão sendo declarados os operadores de sucessor, subtração e soma para os naturais. Observe que a soma é total, associativa, comutativa e tem como elemento neutro o número 0.

- **predicados**

```
ops 0  : Nat;
    suc: Nat -> Pos;
    __-__: Nat * Nat ->? Nat;
    __+__: Nat * Nat ->  Nat, assoc, comm, unit 0
```

Figura 4.4: Declaração de operações

A declaração de predicado é semelhante a declaração de operação, porém não existe sort resultante. Os predicados podem ser definidos ao mesmo tempo que são declarados. Ver exemplos na figura 4.5.

```
preds odd  : Nat;
      __<__: Nat * Nat

preds even(n : Nat) <=> not odd(n);
      __<__ (m,n : Nat) <=> m < n \/ m = n
```

Figura 4.5: Declaração de predicados

- **Tipo**

  Uma declaração de tipo de dados é feita usando "::=" onde, o símbolo à esquerda declara o sort e cada alternativa à direita declara um construtor (possivelmente com algum seletor). Os tipos de dados declarados como *generated* restringem os sorts através de seus construtores. Ver exemplo na figura 4.6.

4.2 Especificação Estruturada

As especificações estruturadas possuem as mesmas noções de assinatura e classe de modelos das especificações básicas. De fato, a parte estruturada de CASL é essencialmente independente dos detalhes da especificação básica: a mesma estrutura pode ser usada tanto para

```
...
spec Nat = free type Nat ::= 0 | suc(pre:? Nat)
...
spec Int =
     Nat
then %def
     generated type Int ::= __ - __(Nat;Nat)
     forall a,b,c,d: Nat
     . a - b = c - d <=> a + d = c + b    %(Int_equality)%
...
```

Figura 4.6: Especificação generated

restringir as especificações básica (por exemplo, eliminando funções parciais, subsorts, predicados ou quantificadores explícitos) como para estendê-las (por exemplo, funções de mais alta ordem).

CASL utiliza o princípio "mesmo nome, mesma coisa", ou seja, quando são existem duas ocorrências do mesmo sort na mesma especificação básica, pode-se garantir que eles são interpretados a partir do mesmo conjunto. Para as operações e predicados, a situação é um pouco diferente pois o "nome" é acompanhado do perfil.

O princípio "mesmo nome, mesma coisa" é aplicado principalmente em uniões e extensões - mas não entre especificações nomeadas em bibliotecas pois, o mesmo sort pode ser usado em diferentes especificações nomeadas na mesma biblioteca, com interpretações completamente diferentes. Similarmente isso ocorre para operações ou predicados com mesmos perfís.

Quando especificações nomeadas estão "combinadas" na mesma especificação estruturada (com referência aos seus nomes - talvez indiretamente via outras especificações nomeadas), as coincidências podem ser eliminadas traduzindo os símbolos usados nas especificações para novos símbolos.

Um outro ponto em CASL, é a facilidade de "esconder" (*hidden*) símbolos auxiliares, isto é, símbolos que não são inerentes ao que vai ser especificado.

Uma especificação "nomeia" uma especificação, permitindo reuso por referência ao nome.

Por exemplo, "*Int*" pode referir-se à especificação de inteiros. Em CASL, uma especificação nomeada pode ter parâmetros; o corpo da especificação é uma extensão do que está especificado nos parâmetros. Cada referência à especificação requer instanciação de todos seus parâmetros. Por exemplo, "*List*" pode referir-se a uma especificação que estende uma especificação parametrizada nomeada de "*Elem*" então, qualquer referência a "*List*" tem que fornecer uma especificação com argumento que seja compatível com "*Elem*".

Na linguagem existem várias maneiras de construir especificações estruturadas que são: tradução, redução, união, extensão, etc.

- **Tradução**: A tradução de símbolos declarados para novos símbolos é especificada dando uma lista de mapeamentos da forma $old \rightarrow new$. Mapeamentos identificados por $old \rightarrow old$ podem ser abreviados por *old*, ou simplesmente ser omitidos. Ver exemplos na figura 4.7.

```
NAT with NAT |->  Natural, suc |-> succ_
NAT with op __+__ |-> plus, pred __<__ |-> lt
```

Figura 4.7: Tradução

- **Redução**: A redução significa remover símbolos da assinatura de uma especificação, e remover os correpondentes itens dos modelos. Quando um sort é removido, remove-se todas as operações e predicados cujos perfís incluem aquele sort. CASL fornece duas formas de especificar uma redução: listando os símbolos a serem escondidos (*hidden*), ou listando aqueles que devem ser revelados (*reveal*). No segundo caso, os símbolos mostrados podem também ser traduzidos para novos símbolos. Ver exemplos na figura 4.8.

- **União**: A união (*and*) de duas ou mais especificações é a união de suas assinaturas. A união pode ser vista de duas formas extremas: quando as especificaçôes tem assinaturas disjuntas, sua união fornece "almagamação" dos seus modelos; quando eles têm a mesma assinatura, resulta na "interseção" dos modelos que satisfazem ambas as especificações de uma só vez. Ver exemplos na figura 4.9.

```
NAT hidden Pos, suc
NAT reveal NAT, 0, __+__, __<__ |-> lt
```

Figura 4.8: Redução

```
NAT and String
MONOID and Commutative
```

Figura 4.9: União

- **Extensão**: A extensão (*then*) de uma especificação pode especificar novos símbolos (conhecidos como "enriquecidos") ou apenas requerer propriedades novas a partir das velhas. Ver exemplos na figura 4.10.

```
NAT then
     sort NAT < INT;
     ops __+__:Int * Int -> Int;
...
```

Figura 4.10: Extensão

  A extensões podem ser classificadas pela forma em que elas afetam a classe de modelos especificada. Por exemplo, uma extensão pode ser chamada de conservativa quando nenhum modelo é perdido. CASL fornece anotações como *%implies*, *%def* e *%cons* para denotar, respectivamente, que a classe de modelo não é mudada, que cada modelo da especificação pode ser unicamente estendido para um modelo de uma especificação estendida ou que a extensão é conservativa.

- **Especificações Livres**: Nas especificações livres (*free*) o caso mais simples de especificação ocorre quando a especificação restrita para ser interpretada livremente é fechada. A assinatura da especificação não é mudada, mas os modelos estão restritos ao seu modelo inicial. Ver exemplo na figura 4.11.

```
free
   {sort Nat; ops 0 : Nat; suc : Nat -> Nat}
```

Figura 4.11: Especificação livre

Geralmente, uma especificação livre (*free*) pode ser uma extensão livre. Ver exemplo na figura 4.12.

```
sort Elem then
free
   {type Set ::= {} | {__}(Elem) | __U__(Set;Set)
   op __U__: Set*Set -> Set, assoc, comm, idem, unit{}}
```

Figura 4.12: Especificação livre como extensão

- **Especificações nomeadas, genéricas e parâmetros**: Uma especificação pode ser estruturada com a presença de parametrização, onde o parâmetro é considerada uma subespecificação fechada. Ver exemplos na figura 4.13.

```
spec ELEM =  sort Elem
spec LIST[ELEM] = free type List ::= nil | cons(Elem;List)
```

Figura 4.13: Especificação genérica e parametrização

Uma especificação nomeada é essencialmente uma extensão de todos seus parâmetros. Uma referência a uma especificação nomeada com parâmetros é chamada de "instanciação", e tem que determinar um argumento de especificação para cada parâmetro, indicando como ele "ajusta" (*fits*), dando uma descrição do parâmetro da assinatura para o argumento da assinatura. Ver exemplo na figura 4.14.

```
LIST[NAT fit Elem |-> Nat]
```

Figura 4.14: Instanciação

4.3 Especificação Arquitetural

Uma especificação arquitetural em CASL consiste de uma lista de declarações de **unidades** (*units*), indicando os módulos de componentes requeridos com especificações para cada um deles, junto com uma **expressão de unidade** que descreve a forma como estes modulos devem ser combinados para dar um resultado. As expressões de unidades podem ser feitas com amalgamação, redução, tradução, etc. Ver exemplo na figura 4.15.

```
arch spec IMP_NAT_LIST =
   units N : NAT;
         F : NAT -> LIST[NAT]
   result F[N]
```

Figura 4.15: Especificação arquitetural

No exemplo da lista de naturais, figura 4.15, tem-se uma implementação N de NAT, uma função F que estende algum N para uma implementação de $List[NAT]$, uma forma de obter o resultado desejado: aplicando F a N.

Os modelos da especificação arquitetural em CASL são requesitos para fornecer as unidades de N e F, bem como suas composições. Se a implementação N de NAT for mudada, F pode ser reusada, sem ser re-implementada, (F também pode ser mudada sem que N mude).

Para explicitar assunções (*assumptions*) que são partes de uma implementação construída sobre outras partes são usadas as chamadas interfaces. Em CASL, interfaces são expressadas como especificações (estruturadas) ordinárias, onde os símbolos declarados pela especificação não somente têm que ser implementados, mas também têm que satisfazer todas as propriedades declaradas.

Um aspecto crucial de especificações arquiteturais é que elas não só fornecem decomposições de tarefas desenvolvidas dentro de pequenas sub-tarefas como também indicam como compor (ou ligar) os resultados de sub-tarefas. Uma especificação de unidade expressa tudo aquilo que quem está implementando precisa conhecer.

É importante fazer a distinção entre estrutura de especificações e especificações de estrutura. Ao especificar, por exemplo, "INT" como uma extensão de NAT não é preciso separar as implementações das duas especificações. O que pode não ser tão óbvio é que a distinção é realmente essencial, no mínimo se uma especificação está usando o construtor de especificação estruturada fornecido pelo CASL, então considera-se a união de duas especificações com declaração de símbolos comuns mas com diferentes axiomas. Se cada especificação é implementada separadamente, sem levar em conta as propriedades requeridas pela outra especificação, pode acontecer que os símbolos comuns tenham diferenças e incompatibilidade de implemetação que não podem ser combinadas.

Como já foi visto anteriormente, um modelo de uma especificação estrutural consiste de:

- um coleção de construtores de unidades, junto com
- a unidade (construtor) resultante da composição particular destas unidades

Os construtores são constantes ou funções de unidades para unidades. No caso de serem funções, são sempre persistentes, estendendo seus argumentos. Quando funções têm mais que um argumento, estes devem ser compatíveis, implementando qualquer símbolo comum da mesma forma.

Uma definição de uma **especificação arquitetural** especifica unidades e como elas são compostas.

Uma **declaração de unidade** nomeia uma unidade que está sendo desenvolvida, e dá seu tipo. Quando o tipo é uma especificação ordinária, a unidade será um modelo ordinário dela; quando o tipo é uma função, a unidade é uma função do modelo ordinário (compatível) que estende os argumentos. Ver exemplo na figura 4.16.

A declaração de unidade usando **given** fornece uma declaração implícita de uma função de unidade que é aplicada apenas uma vez. Se a declaração de F fosse substituída por L no

```
N : NAT
L : LIST[NAT] given N
F : NAT -> LIST[NAT]
```

Figura 4.16: Declaração de unidade

exemplo de especificação arquitetural, então uma implementação da especificação arquitetural poderia envolver ainda uma função que poderia dar uma implementação de LIST[NAT] estendendo qualquer implementação N de NAT.

Uma **definição de unidade** nomeada como unidade pode ser obtida de unidades anteriores (na mesma especificação arquitetural), possivelmente envolvendo *fitting*, *hidding*, tradução, etc. Ver exemplo na figura 4.17.

```
L = F[N]
L = F[N fit ... ] hidden ...
```

Figura 4.17: Definição de unidade

Uma **especificação de unidade** nomeia um tipo de unidade, permitindo o seu reuso. As próprias especificações arquiteturais podem também ser reusadas como especificações de unidades. Ver exemplo na figura 4.18.

```
unit spec GEN_LIST = NAT -> LIST[NAT]
```

Figura 4.18: Especificação de unidade

As formas de **expressões de unidades** mais usadas são:

- *aplicação* (para argumentos compatíveis): $F[N]$, $F[N$ **fit** $\ldots]$
- *abstração*: $\lambda\ N : NAT \bullet \ldots N \ldots$
- *tradução, redução*: U **with** $\ldots$, U **hide** $\ldots$, $\ldots$

- *amalgamação* (de unidades compatíveis): N **and** C

4.4 Especificações de Bibliotecas

Bibliotecas em CASL são formadas pela agregação de definições de especificações estruturadas, visões, especificações arquiteturais e especificações de unidades. Além disso, bibliotecas podem importar especificações definidas em outras bibliotecas.

Quando especificações estão agregadas dentro de bibliotecas, surge a questão de visibilidade de símbolos entre especificações. Em CASL, os símbolos disponíveis na especificação são aqueles declarados dentro dela, junto com aqueles declarados como *hidden* nas especificações nomeadas que ela referencia. Assim, quando uma especificação em uma biblioteca é mudada, outras especificações podem ser afetadas.

Um outro aspecto consiste na visibilidade de especificações nomeadas. A visibilidade em CASL é linear. Uma especificação pode apenas se referir aos nomes de especificações (e visões) que a antecedem na biblioteca.

Uma biblioteca tem um único nome, que é usado para que a biblioteca seja referenciada a partir de outras bibliotecas. As bibliotecas CASL possuem números de versão que são importantes mas não obrigatórios.

4.5 HOL-CASL

HOL-CASL é um sistema que fornece uma interface entre a linguagem de especificação CASL e o sistema de provador de teoremas Isabelle/HOL [39]. Isabelle está baseado em táticas e é implementado em ML. A lógica pura é usada para representar uma variedade de lógicas dentro de Isabelle, uma delas é a HOL (*higher-order logic*). A lógica de CASL é diferente da lógica de Isabelle, que, por exemplo, não aceita a parcialidade de CASL que é tratada através do HOL-CASL uma vez que o provador de teorema não atua diretamente na lógica. de CASL. O HOL-CASL transforma os teoremas a serem provados em submetas que serão processadas pelo provador.

A ferramenta HOL-CASL está disponível para máquinas linux e unix. Ela possui toda a biblioteca básica de CASL, que inclui as especificações dos números naturais, inteiros, racionais, booleanos, relação de ordem, álgebra, entre outros, bem como alguns exemplos prova de especificações.

Para se analisar especificações na ferramenta HOL-CASL são necessárias algumas instruções que vão desde a escolha da especificação, do contexto, até comandos para introduzir axiomas já provados para facilitar a resolução das metas.

4.5.1 Exemplo

Com o objetivo de ver como funciona, basicamente, a ferramenta HOL-CASL vamos utilizar a especificação "Demo_Nat.casl" [33]. Ver figura 4.19.

```
library Demo_Nat
%prec { __+__ } < { __*__ }
  spec Nat =
    free type Nat ::= 0 | suc(Nat)
    ops   __ + __, __ * __: Nat * Nat -> Nat;
          1,2: Nat;
    forall m,n: Nat
     . 0 + m = m                          %(Nat_add_0)%
     . suc(n) + m = suc(n + m)            %(Nat_add_suc)%
     . 0 * m = 0                          %(Nat_mult_0)%
     . suc(n) * m = n * m + m             %(Nat_mult_suc)%
     . 1 = suc (0)                        %(Nat_1_def)%
     . 2 = suc (1)                        %(Nat_2_def)%
  end
```

Figura 4.19: Especificação Demo_Nat.casl

Inicialmente deve-se carregar a especificação, ou seja, no caso o nome do arquivo que contém a especificação que se quer analisar. Tal análise é feita não só nas especificações do arquivo mas também naquelas que estão sendo usadas (importadas) pelas especificações do arquivo. Para isso usa-se o comando:

```
use_casl ‘‘nome_do_arquivo’’;
```

Após esse comando, se não houver erros ao carregar a especificação, obtem-se representações dos axiomas como saída, ou seja, são gerados valores.

```
HOL-CASL> use_casl"Demo_Nat";
Reading Demo_Nat.casl
Analyzing spec Nat...
Number of encoded axioms: 9
val Nat_1_def = "1 = (suc(0))" : thm
val Nat_2_def = "2 = (suc(1))" : thm
val Nat_add_0 = "(0+ ?m) = ?m" : thm
val Nat_add_suc = "((suc(?n))+ ?m) = (suc((?n+ ?m)))" : thm
val Nat_mult_0 = "(0* ?m) = 0" : thm
val Nat_mult_suc = "((suc(?n))* ?m) = ((?n* ?m)+ ?m)" : thm
val ga_disjoint_0_suc = "not 0 = (suc(?Y0.0))" : thm
val ga_generated_Nat =
 "[| ?P0.0 0; !!x1. ?P0.0 x1 =>?P0.0 (suc(x1))  |] ==> ?P0.0 ?x0.0" : thm
val ga_injective_suc =
 "((suc(?X0.0)) = (suc(?Y0.0))) = (?X0.0 = ?Y0.0)" : thm
structure Nat :
  sig
    val axioms : (string * thm) list
    val casl : casltheory
  end
Trying Demo_Nat.ML
val it = () : unit
```

Figura 4.20: Análise de Demo_Nat.casl

Neste exemplo podemos ver que foram gerados 9 axiomas: 6 são vistos na especificação, os nomes destes estão relacionados com os comentários que existem ao lado de cada axioma na especificação da figura 4.19 e existem 3 axiomas gerados, “ga_”. Neste exemplo temos os axiomas gerados que definem que:

1) 0 não é sucessor de nenhum outro elemento;

2) a geração de um número natural através da indução, onde a hipótese está limitada por

“[|”e “|]” ;

3) se existe o sucessor de um elemento que é igual ao sucessor de outro elemento então, estes elementos são iguais.

Em seguida deve se definir o contexto(especificação) no qual se quer provar os teoremas, da seguinte forma:

`CASL_context ‘‘especificação.casl’’ ;`

```
HOL-CASL> CASL_context Nat.casl;
val it = () : unit
```

Figura 4.21: Definição do contexto

O contexto deve ser definido uma vez que, em um mesmo arquivo, podem existir várias especificações dele ou “importadas” de outros arquivos. Para o nosso exemplo não existe dúvidas quanto ao contexto a ser trabalhado, pois no arquivo só exite uma especificação. A definição do contexto pode ser vista na figura 4.21.

Uma vez feita a análise é importante que se adicionem os axiomas da especificação dentro do simplificador Isabelle/HOL, isso pode ser feito usando o comando que segue:

`AddsimpALL();`

Para realização das provas ou metas para a especificação usa-se a palavra *Goal* e em seguida descreve-se o que se quer provar.

`Goal‘‘...’’;`

Dependendo da meta construída podem ser geradas várias submetas a serem resolvidas, para conseguir provar a metas. Por exemplo, podemos tentar provar a meta da figura 4.22.

Observa-se no exemplo acima que no nível 0, primeiro passo, foi gerada uma submeta a ser provada.

Para resolver as submetas o simplificador automático de Isabelle contém várias táticas que podem ser usadas. Algumas dessas táticas são “induct_tac”(induz variáveis), “resolve_tac”e “assume_tac”(tenta resolver submetas com auxílio de teoremas definidos em Isabelle ou na

```
HOL-CASL> Goal"1+1=2";
Level 0 (1 subgoal)
(1+ 1) = 2
 1. (1+ 1) = 2
val it = [] : thm list
HOL-CASL>
```

Figura 4.22: Declaração de meta

própria especificação carregada, simplificando as provas envolvendo eliminação ou destruição de regras).

```
HOL-CASL> by (Simp_tac 1);
Level 1
(1+ 1) = 2
No subgoals!
val it = () : unit
HOL-CASL>
```

Figura 4.23: Resolvendo a meta

Na figura 4.23 observamos que não foi gerada submetas, "No subgoals" , mostrando que a meta foi provada através do simplificador no nível 1.

Depois de provadas as metas, é possível salvar o resultado nomeando-o com o comando "qed", para que as provas possam ser utilizadas posteriormente. Ver figura 4.24.

```
HOL-CASL> qed "one_plus_one";
val one_plus_one = "(1+ 1) = 2" : thm
val it = () : unit
HOL-CASL>
```

Figura 4.24: Salvando a meta

Outras táticas podem ser encontradas em [33, 37, 36, 35, 39].

4.6 Considerações Finais do Capítulo

Neste capítulo vimos várias partes da linguagem CASL onde podemos observar que esta suporta parcialidade bem como a possibilidade de tratar sort a partir de outros sorts, ou seja, permite parametrização. Estes fatos são de bastante relevância para a construção do tipo intervalar paramétrico. E para provar a especificação usaremos o HOL-CASL.

Capítulo 5

Tipo Intervalar Parametrizado

Desde o final da década de 50, quando o tipo intervalar foi proposto por Moore e Sunaga, algumas das áreas da ciência que utilizam dados contínuos e se deparam com problemas de erro de máquina vêm adotando, de várias formas, o tipo intervalar. Uma delas é estendendo linguagens de programação de alto nível, como Pascal e Java, somando aos seus sistemas de tipos o tipo intervalo. Outro exemplo é a ferramenta Matlab, para o qual foi construída uma biblioteca com operações e funções para trabalhar com esse tipo de dado [12].

Em linguagens de alto nível orientada a objetos, como Java, uma importante característica é o conceito de reutilização, o que faz com que muito do que já existe seja usado como parte de novos projetos. Isso foi um dos principais pontos com que fizeram com que tais linguagens se tornassem bastante utilizadas. Os tipos abstratos de dados (TAD) se propõem a modelar alguma estrutura e podem ser aplicadas em várias situações e se comportar obedecendo suas regras de construção associadas à aplicação.

Os trabalhos já realizados utilizando o tipo intervalo adotam apenas um tipo de dado, ou seja, definiu-se um intervalo de números naturais, inteiros, racionais, ou outro tipo básico. A relevância de se ter um tipo intervalo que possa absorver qualquer tipo de dado, faz com que ele se torne um TAD, ou seja, uma estrutura que pode ser aplicada usando diferentes tipos de dados de diferentes áreas, de acordo com suas respectivas necessidades. A construção de um TAD está diretamente ligada à idéia do tipo parametrizado (ou paramétrico).

A proposta deste trabalho é especificar a teoria que dá suporte à construção de uma estrutura intervalar que se comporte como um TAD, ou seja, o tipo intervalar parametrizado. Dessa forma poderíamos ter uma estrutura generalizada que possa absorver como parâmetro "qualquer" outro tipo de dado.

A idéia se resume em, tendo como base a especificação de relações de ordem parciais, podermos estender a biblioteca básica de CASL acrescentando os conceitos de reticulado para intervalos bem como a noção de igualdade local em intervalos para, assim, definir o construtor intervalar, gerando a especificação do tipo intervalo paramétrico [1].

Uma vez construídas as especificações na linguagem CASL utilizamos o provador de teoremas HOL-CASL para desenvolver provas relacionadas a essas especificações a fim de validá-las como funcionais.

As provas foram feitas utilizando-se como parâmetro o tipo (a especificação) dos racionais onde foram acrescentadas algumas características a fim de podermos utilizar propriedades desenvolvidas na especificação dos números racionais em nosso construtor intervalar.

A seguir vamos falar de cada biblioteca de especificações construída bem como o tipo intervalar parametrizado foi provado.

5.1 Especificação

Para a especificação do tipo intervalar paramétrico foram construídas três bibliotecas de especificações, que são:

- Especificações da álgebra ordenada: Teve como base a especificação *Algebra_I.casl*, pertencente à biblioteca básica de CASL. Onde foram adaptados alguns conceitos para satisfazer a nossa especificação.

- Especificações de reticulado: Esta é uma extensão da biblioteca *RelationsAndOrders.csl* onde é acrescentada a noção de reticulado oposto.

[1] No texto as palavras paramétrico e parametrizado são equivalentes

- Especificações para desenvolver e aplicar o construtor intervalar: nesta são usadas algumas especificações da biblioteca básica de CASL e as mencionadas anteriormente, ou seja, esta reúne toda a fundamentação teórica para construir a especificação do tipo intervalar paramétrico.

5.1.1 Especificação da Álgebra Ordenada

Na especificação chamada *Order_ Algebra.casl* foi acrescentada a noção de ordem parcial aos conceitos de grupo, anel e corpo, para assim obtermos as álgebras ordenadas. Com isso, podemos trabalhá-las como as álgebras localmente ordenadas definidas no capítulo 3.

Algumas especificações presentes em *Algebra_ I.casl*, que complementam as especificações das álgebras clássicas, foco do nosso trabalho, não serão mostradas pois não alterarão nossos resultados. A idéia foi criar um novo arquivo com o objetivo de não alterar *Algebra_ I*. Porém, como o que pretendemos é estender, os resultados não seriam diferentes se tivessemos acrescentado a ela o que precisavamos. *Algebra_ I* pode ser vista no apêndice A.

```
library Order_Algebra
version 0.1
%% Extension  Basic/Algebra_I of the authors: M.Roggenbach,
   T.Mossakowski, L.Schröder
%prec {__*__} < {__^ __}
%prec {__ + __, __ - __, -__} < {__ / __, __ * __}
from Basic/RelationsAndOrders version 0.7 get PartialOrder
from Basic/Numbers version 0.7 get Nat, Int, Rat
%[introdução da noção de ordem]%
spec BinAlg = PartialOrder
then
sort Elem
```

```
op __ * __: Elem * Elem -> Elem
end
spec SemiGroup =
     BinAlg
then
op __ * __: Elem * Elem -> Elem
forall x,y,z  : Elem . ((x * y) * z) = (x * (y * z))
end
spec CommutativeSemiGroup =
     SemiGroup
then
op __*__: Elem * Elem -> Elem
forall x,y : Elem . (x * y) = (y * x)
end
spec Monoid =
     SemiGroup
then
ops e: Elem;
    __ * __:Elem * Elem -> Elem, unit e
end
spec CommutativeMonoid =
     Monoid
and
CommutativeSemiGroup
```

```
end
%[a seguir foram incluídas as declaração e definição da operação do inverso
aditivo a fim de obter a comutatividade da adição para a especificação de
grupo]%
spec Group =
     Monoid
then
op invAdd__ : Elem -> Elem
forall x, y : Elem
. invAdd(x) * x = e
. x * invAdd(x) = e
. x * y = e => y = invAdd(x)
end
spec AbelianGroup =
     Group
and
CommutativeMonoid
end
spec Ring =
     AbelianGroup with sort Elem,
ops __ * __ |-> __ + __,
    e       |-> 0
and
SemiGroup
```

```
then
forall x,y,z:Elem
. ((x + y) * z) = ((x * z) + (y * z))       %(distributividade)%
. (z * (x + y)) = ((z * x) + (z * y))
end
spec CommutativeRing =
     Ring
and
CommutativeSemiGroup
end
%[CommutativeRing With Unity]%
spec IntegralDomain =
     {CommutativeRing and CommutativeMonoid with op e |-> 1}
then
forall x,y: Elem
. ((x * y) = 0) => ((x = 0) \/ (y = 0))     %(não_divisão_por_zero)%
. not (1 = 0)                          %(zero_diferente_de_um)%
end
spec Field =
     IntegralDomain
then
%[a seguir foram incluídas as declaração e definição da operação do inverso
multiplicativo a fim de obter a comutatividade da multiplicação para a
especificação de corpo]%
```

```
op invMult__ : Elem -> Elem
forall x, y : Elem
. invMult(x) * x = 1
. x * invMult(x) = 1
. x * y = 1 => y = invMult(x)
end
%[nas especificações seguintes são criadas as álgebras ordenadas a partir
das álgebras definidas anteriormente unidas a algumas propriedades como a
monoticidade]%
spec OrderedGroup =
     Group
then
op __+__ : Elem * Elem -> Elem
forall a,b,c,d : Elem
. (a <= c /\ b <= d) => (a + b <= c + d)
end
spec OrderedAbelianGroup =
     {OrderedGroup and AbelianGroup}
end
spec OrderedRing =
    {OrderedAbelianGroup and Ring}
then
op  __*__ : Elem * Elem -> Elem
%[definição da noção de números positivos e negativos]%
```

```
forall a : Elem
. (forall a' : Elem . a <= a' =>
   (forall x,y : Elem . (x <= y) =>
    (a' * x <= a'* y /\ x * a' <= y * a'))) \/
  (forall a' : Elem . a'<= a  =>
   (forall x,y : Elem . (x <= y) =>
    (a' * y <= a'* x /\ y * a' <= x * a')))
end
spec OrderedCommutativeRing =
     {OrderedRing and CommutativeRing}
end
spec OrderedCommutativeRingWithUnity =
     {OrderedCommutativeRing and IntegralDomain }
end
spec OrderedField =
    {OrderedCommutativeRingWithUnity and Field }
end
```

5.1.2 Especificação do Construtor Intervalar

Nesta biblioteca inicialmente foi definida a especificação de um par de elementos que tem como parâmetro qualquer estrutura que obedeça as condições de reticulado, o que vai nos garantir que as estruturas serão limitadas inferiormente e superiormente.

Os elementos, esquerdo e direito, em *Pair* não obedece nenhuma ordem. Em *interval* a ordem entre esses elementos é estabelecida e é acrescentada a ordem de informação, que faz parte da definição do construtor intervalar e a operação de interseção, que é fundamental

para posterior definição de igualdade local entre intervalos.

Para completar aplicamos às álgebras ordenadas toda essa fundamentação o que nos possibilitará chegar a provar a proposa deste trabalho, que é aplicar nessa estrutura intervalar um tipo de dado qualquer [2]. Como exemplo utilizamos os racionais, que é a especificação de Rat que encontramos em *Numbers.casl*

```
library ConstrutorIntervalar
version 0.1
from Numbers version 0.7 get Rat
from Order_Algebra version 0.1 get OrderedGroup, OrderedAbelianGroup,
     OrderedRing, OrderedCommutativeRing,
     OrderedCommutativeRingWithUnity, OrderedField

%[especificação de um tipo que é formado por um par de elementos que pertencem
a um conjunto que obedece a estrutura de reticulado]%
spec Pair[lattice] =
%def
generated type Pair ::= [__.__](left:Elem;right:Elem)
ops
 left__  : Pair -> Elem;
 right__ : Pair -> Elem;
 forall a,b : Elem
 . left([a..b])  = a                                    %(projeção_esquerda)%
 . right([a..b]) = b                                    %(projeção_direita)%
 . [left([a..b])..right([a..b])] = [a..b]
```

[2]Este tipo já deve estar especificado e bem formado

```
then
%[definição do tipo intervalo paramétrico, aqui o par de elementos
  obdecem a uma ordem]%
sort  Interval = {a : Pair . left(a) <= right(a)}
then
 pred __InfOrder__:Interval*Interval;              %(ordem_de_informação)%
 forall x,y : Interval
 . x InfOrder y <=>
   ((left(y) <= left(x)) /\ (left(x) <= right(x)) /\ (right(x) <= right(y)))
then
%[a interseção é definida através do ínfimo e supremo de dois intervalos]%
 op  __cap__ : Interval * Interval ->? Interval        %(interseção)%
 forall x,y : Interval
 . exists z : Interval . x cap y = z if
    (def([sup(left(x),left(y))..inf(right(x),right(y))]) /\
     z = [sup(left(x),left(y))..inf(right(x),right(y))])
end
%[a igualdade local é definida através da interseção. Isso nos proporcionará
a garantia dos axiomas dessa teoria]%
spec IntervalLocalSet[lattice] =
     Pair[lattice]
then
 pred __loc__ : Interval * Interval         %(igualdade_local)%
 forall x,y : Interval
```

```
 . x loc y <=> def(x cap y)
end
%[a seguir teremos aplicação de intervalo local nas álgebras de grupo,
  anel e corpo.]%
spec IntervalLocalGroup[OrderedGroup] =
     IntervalLocalSet[lattice]
then
  __oplus__ : Interval * Interval -> Interval;      %(soma_intervalar)%
   invAdd__ : Interval -> Interval
 forall x,y,z,w: Interval
 . x oplus y = [(left(x) + left(y))..(right(x) + right(y))]
 . ((x InfOrder y) /\ (z InfOrder w)) => ((x oplus z) InfOrder (y oplus w))
 . ((invAdd(x) oplus x) loc y) /\ y = [e..e]
 . ((x oplus invAdd(x)) loc y) /\ y = [e..e]
 . ((((x oplus y) loc z)  /\ ((y oplus x) loc z) /\ z = [e..e]) => y = invAdd(x))
 . ((x oplus y) oplus z) loc (x oplus (y oplus z))
end
spec IntervalAbelianLocalGroup[OrderedAbelianGroup] =
     IntervalLocalGroup[OrderedGroup]
then
 forall x,y : Interval
 . (x oplus y) loc (y oplus x)                          %(comut_soma_intervalar)%
end
spec IntervalLocalRing[OrderedRing] =
```

```
     IntervalAbelianLocalGroup[OrderedAbelianGroup]
then
 op   __otimes__ : Interval * Interval -> Interval     %(mult_intervalar)%
 forall x,y,z : Interval
 . x otimes y =
     [(inf(inf(left(x)*left(y),left(x)*right(y)),
       inf(right(x)*left(y),right(x)*right(y)))..
      (sup(sup(left(x)*left(y),left(x)*right(y)),
       sup(right(x)*left(y),right(x)*right(y))))]
 . (x oplus (y otimes z)) loc ((x otimes  y) oplus (x otimes z))
then
 forall a, a' : Interval
  .((a InfOrder a') =>
    (forall x,y : Interval . (x InfOrder y) =>
     (((a' otimes x) InfOrder (a' otimes y)) /\
      ((x otimes a') InfOrder (y otimes a'))))) \/
    ((a' InfOrder a) =>
     (forall x,y : Interval . (x InfOrder y) =>
       (((a' otimes y) InfOrder (a' otimes x)) /\
       ((y otimes a') InfOrder (x otimes a')))))
end
spec IntervalCommutativeLocalRing[OrderedCommutativeRing] =
     IntervalLocalRing[OrderedRing]
then
```

```
 forall x,y : Interval
 . (x otimes y) loc (y otimes x)                    %(comut_mult_intervalar)%
end
spec IntervalCommutativeLocalRingWithUnity[OrderedCommutativeRingWithUnity] =
     IntervalCommutativeLocalRing[OrderedCommutativeRing]
then
 forall x,y,z : Interval
 . (((x otimes y) loc z) => ((x loc z) \/ (y loc z))) <=> z = [0..0]
 . not (x loc y) <=> x = [1..1] /\ y = [0..0]
end
spec IntervalLocalField[OrderedField] =
     IntervalCommutativeLocalRingWithUnity[OrderedCommutativeRingWithUnity]
then
 ops invMult__:Interval -> Interval
 forall x,y,z : Interval
 . ((invMult(x) otimes x) loc z) <=> z = [1..1]
 . (((x otimes y) loc z)  => (y loc invMult(x))) <=> z = [1..1]
end

%[Deste ponto em diante estaremos aplicando o intervalo paramétrico aos números
racionais]%
spec RatOrderedField =
     OrderedField with sort Elem |-> Rat
end
```

```
spec IntervalLocalRatField[RatOrderedField] =
     IntervalLocalField[OrderedField] with
 ops __oplus__  |-> __+__,
     __otimes__ |-> __*__
then
 op
 degen__ : Rat -> Interval
 forall r : Rat;
 . degen(r) = [r..r]                                  %(intervalo_degenerado)%
end
```

Trabalhando na aplicação do construtor intervalar sobre os racionais, construímos as especificaçôes RatOrderedField e IntervalLocalRatField. Criamos a operação que constrói o intervalo degenerado. Esta última foi onde aplicamos nosso exemplo de prova pois, em HOL-CASL temos que definir o contexto para o que queremos provar e no contexto dessa especificação temos o construtor intervalar aplicado a uma estrutura álgebrica local que é Rat.

5.2 Prova

Como já foi dito no capítulo 3 para se provar metas no HOL-CASL devemos definir o contexto, descrever o que queremos provar e a partir disso aplicar táticas que não gerem novas submetas, o que garantirá que o que se quer provar foi realmente provado e que a especificação funciona como desejado.

No nosso caso queremos que propriedades construídas em RatOrderedField (racionais com características de corpo) em uma estrutura intervalar local (com características de corpo) possam ser mapeadas para a estrutura intervalar.

O exemplo construído foi de provar que qualquer intervalo somado a um intervalo degenerado é uma operação localmente comutativa. O que desejamos aqui é mostrar que as operações dos racionais são mapeadas para as operações construídas para o construtor intervalar, no caso "+ |-> oplus", bem como utilizarmos a igualdade local.

Primeiramente foi carregada a especificação onde obtemos na saída representações dos seus axiomas que poderam ser usados em provas. Como a saída é muito extensa optou-se por mostrar apenas parte como uma forma de visualizar como tais representações são construídas.

Uma vez carregada a biblioteca foi definido o contexto onde queremos fazer a(s) prova(s).

```
HOL-CASL> CASL_context IntervalLocalRatField.casl;
val it = () : unit
HOL-CASL>
```

Como definimos que o contexto é a especificação IntervalLocalRatField então nós usaremos operações (ou predicados) desta para provar que se temos um intervalo x e um número racional r então *(x + degen(r)) loc (degen(r) + x)*.

```
HOL-CASL> Goal"forall x: Interval; r : Rat .
              ((x + degen(r)) loc (degen(r) + x))";
Level 0 (1 subgoal)
x+ (degen r)loc (degen r)+ x
 1. x+ (degen r)loc (degen r)+ x
val it = [] : thm list
HOL-CASL> by (resolve_tac [Ax020] 1);
Level 1
```

```
x+ (degen r)loc (degen r)+ x

No subgoals!

val it = () : unit

HOL-CASL>
```

Ao utilizarmos como tática o axioma *Ax020 = "(?xoplus ?y)oplus ?zloc ?xoplus (?yoplus ?z)": thm* conseguimos não gerar submetas, garantindo o desejado no nosso exemplo e verificar que há um mapeamento da operação soma do racionais para operação soma do construtor intervalar ao usarmos a igualdade local.

Com isso, podemos observar uma transferência de uma operação de uma especificação parâmetro para uma do construtor intervalar.

5.3 Considerações Finais do Capítulo

As especificações vistas neste capítulo incluiu teorias e definições vistas nos capítulos anteriores. Especificações foram alteradas e criadas a fim de comportá-las. Embora não tenhamos utilizado e/ou conseguido abranger o todo das mesmas, pudemos obter um resultado positivo agregando informações a linguagem e codificando a teoria.

Capítulo 6

Conclusão e Trabalhos Futuros

A escolha da linguagem utilizada, CASL, foi fundamental pois além de trabalhar com parcialidade em operações e predicados ela permite que especificações sejam passadas como parâmetros influenciando as características da especificação parametrizada. Este fato nos auxiliou muito pois isso foi usado com muita frequência na construção das especificações.

Apesar de nos favorecer em vários aspectos CASL é uma linguagem nova e, como tudo que é novo, tivemos algumas dificuldades no seu aprendizado. Este fato nos levou a entrar em contato com seus desenvolvedores a fim de esclarecimentos, uma vez que nos documentos existentes não conseguiamos todas respostas.

Uma vez construídas as especificações tivemos que analisá-las e então utilizamos CATS [1] que faz apenas análises sintáticas. Com isso partimos para utilizar HOL-CASL que é uma interface para o provador de teoremas genérico Isabelle. Nesta ferramenta tivemos um problema inicial, pois a versão até então construída não suportava algumas características de CASL que usamos na nossa especificação. A partir daí uma nova versão da ferramenta foi construída para adaptar-se a essas características e, enfim, pudemos realmente provar teoremas para as especificações.

Com a especificação da generalização da matemática intervalar [9] desenvolvida neste trabalho, usando uma linguagem de especificação algébrica, conseguimos construir uma estrutura intervalar paramétrica a partir das teorias do construtor intervalar bem como da

igualdade local.

Alguns trabalhos já realizados utilizando o tipo intervalo adotam apenas um tipo de dado, ou seja, intervalo de números naturais, inteiros, racionais, ou outro tipo básico. A relevância de se ter um tipo intervalo que possa absorver qualquer tipo de dado, faz com que ele se torne um TAD, ou seja, uma estrutura que pode ser aplicada usando diferentes tipos de dados de diferentes áreas, de acordo com suas respectivas necessidades. A construção de um TAD está diretamente ligada à idéia do tipo parametrizado.

Com base na especificação de relações de ordem parciais e de álgebras, propomos estender a especificação básica de CASL acrescentando os conceitos de reticulado oposto bem como a noção de igualdade local em álgebras para assim definir o construtor intervalar, gerando a especificação do tipo intervalo paramétrico. Esta especificação poderá evitar que tenhamos que desenvolver especificações intervalares para cada tipo de dados contínuos encontrados em aplicações do dia a dia, ou seja possibilitará a reutilização da especificação do tipo intervalor, acrescentando somente as características próprias de cada tipo. Por exemplo, o tipo de intervalos reais pode ser necessário para modelar valores reais em aplicações onde esses valores são obtidos via algum instrumento que mede com margens de erro; de intervalos complexos para aplicações como processamento de sinais (por exemplo) que lidem com valores complexos; o tipo intervalar de imagens digitais pode ser usado para representar uma imagem real obtidas via instrumentos (câmeras fotográficas, imagens de satelites, mamografias, etc) que discretizam a imagem real; etc. e com a especificação intervalar paramétrica estes tipos de intervalos poderão ser especificados instanciando esta especificação com as respectivas especificações dos tipos reais [43], complexos e imagens digitais (ainda não especificados) com suas operações e propriedades algébricas. É claro que essas instâncias do intervalo paramétrico poderão ser enriquecidas com propriedades e operações não contempladas nas álgebras (ou inclusive a própria especificação do tipo intervalo paramêtrico poderá ser estendida para considerar "álgebras" não especificadas aqui).

Atualmente as linguagens de programação que incluem tipos de dados intervalares, chamadas XSC (acrônimo de eXtension for Scientific Computing) consideram diversos tipos de intervalos como primitivos. O tipo de dado intervalar paramétrico não faz parte de nenhuma

linguagem de programação hoje em dia. No entanto com a especificação algébrica deste tipo de dados esperamos que viabilize, num futuro próximo, o desenvolvimento de linguagens de programação XSC com o tipo de dado intervalar paramétrico como primitivo, de tal forma que os operadores aritméticos intervalares possam ser implementados usando polimorfismo.

Neste trabalho fizemos alguns testes para nosso construtor intervalar encontrados na seção 5.2. Esses testes foram simples porém inportantes para garantir a viabilidade de se aplicar a especificações de outros tipos de dados bem como a matrizes, como no capítulo 3, e assim ser aplicados em sistemas que usem esses tipos de dados.

Apêndice A

Especificações da biblioteca básica de CASL

A.1 Algebra_I.casl

```
library Basic/Algebra\_I
version 0.7
%% authors: M.Roggenbach, T.Mossakowski, L.Schröder
%% date: 23.3.01
%prec {__*__} < {__^__}
%prec {__ +__,__ - __, -__} < {__ / __, __ *__}
from Basic/RelationsAndOrders version 0.7 get TotalOrder, ExtTotalOrder,
RichTotalOrder, PreOrder, EquivalenceRelation
from Basic/Numbers version 0.7 get Nat, Int, Rat
spec
     BinAlg =
```

```
     sort
          Elem
     op
          __ * __: Elem * Elem -> Elem
end
spec
     SemiGroup =
     BinAlg
then
     op
          __ * __: Elem * Elem -> Elem, assoc
end
spec
     SigPowerBinAlg [sort Exponent] =
     BinAlg
then
     op __ ^ __: Elem * Exponent -> Elem
end
spec
     PowerTheorems [sort Exponent;
    ops __ + __, __ * __: Exponent * Exponent -> Exponent] =
     SigPowerBinAlg [sort Exponent]
then
     forall x: Elem; n,m: Exponent
```

```
     . x ^ (n + m)=x ^ n * x ^ m %(Power_add)%
     . x ^ (n * m)=(x ^ n) ^ m              %(Power_mult)%
end
spec
     CommutativeSemiGroup =
     SemiGroup
then
     op __*__: Elem * Elem -> Elem, comm
end
spec
     PowerTheoremsComm [sort Exponent;
ops __ + __, __ * __: Exponent * Exponent -> Exponent] =
     SigPowerBinAlg [sort Exponent]
then
     forall x,y: Elem; n: Exponent
     . x^ n * y^ n=(x*y)^ n              %(Power_basemult)%
end
view
     CommutativeSemiGroup_in_ExtTotalOrder_max [TotalOrder] :
     CommutativeSemiGroup to ExtTotalOrder [TotalOrder] =
op
     __ * __ |-> max
end
view
```

```
     CommutativeSemiGroup_in_ExtTotalorder_min [TotalOrder] :
     CommutativeSemiGroup to ExtTotalOrder [TotalOrder] =
op
     __*__ |-> min
end
spec
     Monoid =
     SemiGroup
then
     ops
           e:      Elem;
           __ * __:Elem * Elem -> Elem, unit e
end
spec
     CommutativeMonoid =
     Monoid
and
     CommutativeSemiGroup
end
view
     CommutativeMonoid_in_Nat_Add:
     CommutativeMonoid to Nat
=
     sort
```

```
          Elem |-> Nat,
     ops
           e |-> 0,
           __ * __ |-> __ + __
end
view
     CommutativeMonoid_in_Nat_Mult:
     CommutativeMonoid to Nat
=
     sort
          Elem |-> Nat,
     ops
  e |-> 1,
  __ * __ |-> __ * __
end
view
     CommutativeMonoid_in_Int_Mult:
     CommutativeMonoid to Int
=
     sort
          Elem |-> Int,
     ops
  e |-> 1,
  __ * __ |-> __ * __
```

```
end
spec
     Group =
     Monoid
then
     forall x: Elem
     . exists x': Elem . x' * x = e
end
spec
     AbelianGroup =
Group
and
CommutativeMonoid
end
view
     AbelianGroup_in_Int_Add:
     AbelianGroup to Int
=
     sort
          Elem |-> Int,
     ops
  __ * __ |-> __ + __,
  e   |-> 0
end
```

```
spec
     MonoidAction [Monoid] =
     sort
          Space
     op
          __*__: Elem * Space -> Space
     forall x: Space; a,b: Elem
     . e * x = x                          %(MAction_unit)%
     . (a * b) * x = a * (b * x)          %(MAction_assoc)%
end
spec
     GroupAction [Group] =
     MonoidAction [Group]
end
spec
     Ring =
     AbelianGroup with sort Elem,
       ops __ * __ |-> __ + __,
   e    |-> 0
and
     Monoid with ops e, __*__
then
     forall x,y,z:Elem
     . (x + y) * z = (x * z) + (y * z)        %(Ring_distr1)%
```

```
     . z * ( x + y ) = (z * x) + (z * y)      %(Ring_distr2)%
end
view AbelianGroup_in_Ring_add:
     AbelianGroup to Ring =
     ops e |-> 0,
 __ * __ |-> __ + __
end
spec
     CommutativeRing =
     Ring
and
     CommutativeMonoid with ops e, __ * __
end
spec
     IntegralDomain =
     CommutativeRing
then
     forall x,y: Elem
     . ( x * y = 0 => ( x = 0 \/ y = 0 ) )  %(noZeroDiv)%
     . not (e = 0)                          %(zeroNeqOne)%
end
spec
     EuclidianRing =
     IntegralDomain and {Nat reveal pred  __<__ }
```

```
then
     op
          delta: Elem ->? Nat
     forall a,b: Elem
     . def delta(a) if not a = 0  %(ER_delta_dom)%
     . (exists q,r : Elem . a = q * b + r /\
                     (r = 0 \/ delta(r) < delta(b) ))
if not b = 0              %(ER_div)%
end
view
     EuclidianRing_in_Int :
     EuclidianRing to Int
=
     sorts
          Elem |-> Int,
     ops
          delta |-> abs,
  e |-> 1
end
spec
     ConstructField =
     CommutativeRing
then
     axiom
```

```
          not e = 0
     sort
          NonZeroElem = { x: Elem . not x = 0 }
then
     closed
     {
          Group with
           sort Elem |-> NonZeroElem,
           ops e, __*__
     }
end
%% an obvious view which helps to write the specification ExtField:
view
     AbelianGroup_in_ConstructField :
     AbelianGroup to ConstructField
=
     sort
          Elem |-> NonZeroElem
end
spec
     Field =
     ConstructField hide sort NonZeroElem
end
view
```

```
     Field_in_Rat: Field to Rat
=
     sort Elem |-> Rat,
     op e |-> 1
end
spec
     ExtSemiGroup[SemiGroup] given Nat =
%def
     SigPowerBinAlg [sort Pos]
then
     forall x: Elem; n: Pos
     . x ^ 1 = x                          %(SGroup_pow_1)%
     . x ^ suc(n) = x * (x ^ n)           %(SGroup_pow_suc)%
%[then
     %implies
     PowerTheorems [Nat fit sort Exponent |-> Pos]
]%
end
spec
     ExtCommutativeSemiGroup [CommutativeSemiGroup] given Nat =
%def
     ExtSemiGroup [SemiGroup]

end
```

```
spec
     ExtMonoid  [Monoid] given Nat =
%def
     ExtSemiGroup [SemiGroup]
then
     SigPowerBinAlg [sort Nat]
then
     forall x: Elem
     . x ^ 0 = e              %(Monoid_pow_0)%
then %implies
     forall n: Nat
     . e ^ n = e              %(Monoid_pow_unit)%
%%and
%%     PowerTheorems [Nat fit sort Exponent |-> Nat]
end
spec
     ExtCommutativeMonoid  [CommutativeMonoid]
     given Nat =
%def
     ExtMonoid [Monoid]
and
     ExtCommutativeSemiGroup [CommutativeSemiGroup]
%[then
     %implies
```

```
     PowerTheoremsComm [Nat fit sort Exponent |-> Nat]
]%
end
spec
     ExtGroup [Group] given Int =
%def
     ExtMonoid [Monoid]
then
     ops
           inv: Elem -> Elem;
           __ / __: Elem * Elem -> Elem;
     forall x,y: Elem
     . inv(x) * x = e                          %(Group_inverse_def)%
     . x / y = x * inv(y)                      %(Group_div_def)%
then
     SigPowerBinAlg [sort Int]
then
     forall x: Elem; p: Pos
     . x^ (- p)=inv(x ^ p)                     %(Group_pow_neg)%
then %implies
     forall x,y,z: Elem; n,m: Int
     . x * inv(x) = e                          %(Group_right_inv)%
     . x = y if z * x = z * y                  %(Group_left_canc)%
     . x = y if x * z = y * z                  %(Group_right_canc)%
```

```
     . inv(inv(x))=x                                  %(Group_inv_inv)%
     . inv(e)=e                                       %(Group_inv_e)%
     . inv(x*y)=inv(y)*inv(x)                         %(Group_inv_prod)%
%%and
%%     PowerTheorems [Int fit sort Exponent |-> Int]
end
spec ExtAbelianGroup  [AbelianGroup] given Int =
%def
     ExtGroup[AbelianGroup]
and
     ExtCommutativeMonoid[AbelianGroup]
end
spec
     ExtMonoidAction [MonoidAction [Monoid]] given Nat =
%def
     ExtMonoid [Monoid]
then
     pred connected: Space * Space
     forall x,y: Space
     . connected(x,y) <=> exists a: Elem . a * x = y
                                                  %(Action_connected_def)%
end
view PreOrder_in_ExtMonoidAction [MonoidAction [Monoid]] given Nat:
     PreOrder to ExtMonoidAction [MonoidAction [Monoid]] =
```

```
     sort Elem |-> Space,
     pred __ <= __ |-> connected
end
spec
     ExtGroupAction [GroupAction [Group]] given Int =
%def
     ExtMonoidAction [GroupAction [Group]]
and
     ExtGroup [Group]
then
     %implies
     forall a,b:Elem; x,y: Space
     . x = y if a * x = a * y                    %(GroupAct_inj)%
     . exists z: Space . a * z = x               %(GroupAct_surj)%

end
view EqRel_in_ExtGroupAction [GroupAction [Group]] given Int:
     EquivalenceRelation to ExtGroupAction [GroupAction [Group]] =
     sort Elem |-> Space,
     pred __ ~ __ |-> connected
end
spec
     ExtRing  [Ring ] given Int =
%def
```

```
     ExtAbelianGroup [view AbelianGroup_in_Ring_add]
     with ops   inv     |-> -__,
__ / __ |-> __ - __,
__ ^ __ |-> __ times __
and
     ExtMonoid[Monoid]
     with op  __ ^ __
and
     preds
          isIrred, isUnit: Elem
     sorts
           NonZero[Elem] = { x: Elem . not x = 0 };
           RUnit[Elem] = { x: Elem . isUnit(x) };
           Irred[Elem] = { x: Elem . isIrred(x) }
     forall x,y: Elem
     . isUnit(x) <=> exists y: Elem . x * y = e /\ y * x = e
                                                %(Ring_isUnit_def)%
     . isIrred(x) <=> (not isUnit(x) /\
forall y, z: Elem . (x = y * z => (isUnit(y) \/ isUnit(z))))
                                                %(Ring_isIrred_def)%
then
%def
     ops
e: RUnit[Elem];
```

```
        -__: RUnit[Elem] -> RUnit[Elem];
__ * __: RUnit[Elem] * RUnit[Elem] -> RUnit[Elem]
end
spec
     ExtCommutativeRing  [CommutativeRing] given Int =
%def
     ExtRing[Ring]
then
     preds
           hasNoZeroDivisors: ();
           __ divides __ :    Elem * Elem;
           associated:        Elem * Elem
     forall x,y: Elem
     . hasNoZeroDivisors <=> forall x,y: Elem. (x * y = 0 => x=0 \/ y=0)
                                                %(hasNoZeroDivisors_def)%
     . x divides y <=> exists z: Elem. x * z = y
                                                %(divides_def)%
     . associated(x,y) <=> exists u:RUnit[Elem]. x=u*y
                                                %(associated_def)%
then
     %implies
     forall x,y:Elem
     . associated(x,y) <=> (x divides y /\ y divides x)
end
```

```
view
     PreOrder_in_ExtCRing  [CommutativeRing] given Int:
     PreOrder to
     ExtCommutativeRing[CommutativeRing] =
     pred
          __ <= __ |-> __ divides __
end
view
     EqRel_in_ExtCRing  [CommutativeRing] given Int:
     EquivalenceRelation to
     ExtCommutativeRing[CommutativeRing]=
     pred
          __ ~ __ |-> associated
end
spec
     ExtIntegralDomain  [IntegralDomain] given Int =
%def
     ExtCommutativeRing [CommutativeRing]
then
     op __ * __:  NonZero[Elem] * NonZero[Elem] -> NonZero[Elem]
then
     %implies
     axiom hasNoZeroDivisors
end
```

```
spec
     ExtEuclidianRing  [EuclidianRing] given Int=
%def
     ExtIntegralDomain [IntegralDomain]
end
%[
spec
     ExtField  [Field] given Int=
%def
     ExtRing [Ring]
then
closed
     {
ExtAbelianGroup[view AbelianGroup_in_ConstructField]
with sort NonZeroElem|->NonZero[Elem],
     ops inv, __ / __, __ ^ __}
then
     op
           __ / __: Elem * Elem ->? Elem;
     forall x:Elem; n: NonZero[Elem]
     . 0/n=0                                    %(Field_div_def1)%
     . not def x/0                              %(Field_div_def2)%
then
     %implies
```

```
     forall x,y:Elem
     . def x/y <=> not y=0                          %(Field_div_dom)%
end
]%
spec RichSemiGroup =
     ExtSemiGroup [SemiGroup]
end
spec RichCommutativeSemiGroup =
     ExtCommutativeSemiGroup [CommutativeSemiGroup]
end
spec RichMonoid =
     ExtMonoid [Monoid]
end
spec RichCommutativeMonoid =
     ExtCommutativeMonoid [CommutativeMonoid]
end
spec RichGroup =
     ExtGroup [Group]
end
spec RichAbelianGroup =
     ExtAbelianGroup [AbelianGroup]
end
spec RichMonoidAction [Monoid] =
     ExtMonoidAction [MonoidAction [Monoid]]
```

```
end
view PreOrder_in_RichMonoidAction [Monoid]:
     PreOrder to RichMonoidAction [Monoid]=
     sort Elem |-> Space,
     pred __ <= __ |-> connected
end
spec RichGroupAction [Group] =
     ExtGroupAction [GroupAction [Group]]
end
view EqRel_in_RichGroupAction [Group]:
     EquivalenceRelation to RichGroupAction [Group]=
     sort Elem |-> Space,
     pred __ ~ __ |-> connected
end
spec RichRing =
     ExtRing [Ring]
end
spec RichCommutativeRing =
     ExtCommutativeRing [CommutativeRing]
end
view
     PreOrder_in_RichCRing:
     PreOrder to RichCommutativeRing =
     pred
```

```
          __ <= __ |-> __ divides __
end
view
     EqRel_in_RichCRing:
     EquivalenceRelation to RichCommutativeRing =
     pred
          __ ~ __ |-> associated
end
spec RichIntegralDomain =
     ExtIntegralDomain [IntegralDomain]
end
spec RichEuclidianRing =
     ExtEuclidianRing [EuclidianRing]
end
%[
spec RichField =
     ExtField [Field]
end
]%
view
     CommutativeSemiGroup_in_RichTotalOrder:
     CommutativeSemiGroup to RichTotalOrder =
op
     __ * __ |-> max
```

```
end
view
     CommutativeSemiGroup_in_RichTotalOrder :
     CommutativeSemiGroup to RichTotalOrder =
op
     __*__ |-> min
end
```

A.2 RelationsAndOrders.casl

```
library Basic/RelationsAndOrders
version 0.7
%% authors: M.Roggenbach, T.Mossakowski, L.Schröder
%% date: 23.3.01
%prec { __ cup __ } < { __ cap __}
from Basic/Numbers version 0.7 get Nat, Int, Rat
spec Relation =
     sort
          Elem
     pred
          __ ~ __: Elem * Elem
end
spec ReflexiveRelation =
     Relation
then
```

```
     forall
          x:Elem
     .
          x ~ x                              %(reflexive)%
end
spec SymmetricRelation =
     Relation
then
     forall
          x,y:Elem
     .
          x ~ y if y ~ x                     %(symmetric)%
end
spec TransitiveRelation =
     Relation
then
     forall
          x,y,z:Elem
     .
          x ~ z if x ~ y /\ y ~ z            %(transitive)%
end
spec SimilarityRelation =
     ReflexiveRelation and SymmetricRelation
end
```

```
spec PartialEquivalenceRelation =
     SymmetricRelation and TransitiveRelation
end
spec
     EquivalenceRelation =
     ReflexiveRelation and PartialEquivalenceRelation
end
spec
     SigOrder[sort Elem] =

     preds
          __ <= __, __ < __,
          __ >= __, __ > __ : Elem * Elem;
     forall
          x,y:Elem
     .
          x >= y <=> y <= x                %(SigOrder_geq_def)%
     .
          x < y <=> (x <= y /\ not (x=y)) %(SigOrder_less_def)%
     .
          x > y <=> y < x                  %(SigOrder_greater_def)%
end
spec
     PreOrder =
```

```
     {ReflexiveRelation and TransitiveRelation}
     with pred __ ~ __ |-> __ <= __
end
spec
     PartialOrder =
     PreOrder
then
     forall
          x,y:Elem
     .
          x = y if x <= y /\ y <= x       %(POrder_antisym)%
end
spec TotalOrder =
  PartialOrder
then
  forall x,y:Elem
  . x <= y \/ y <= x                      %(TOrder_comparability)%
end
view TotalOrder_in_Nat: TotalOrder to Nat =
     sort Elem |-> Nat
end
view TotalOrder_in_Int: TotalOrder to Int =
     sort Elem |-> Int
end
```

```
view TotalOrder_in_Rat : TotalOrder to Rat =
     sort Elem |-> Rat
end
spec BooleanAlgebra =
     sort  Elem
     ops
           0,1:       Elem;
           __ cap __: Elem * Elem -> Elem, assoc, comm, unit 1;
           __ cup __: Elem * Elem -> Elem, assoc, comm, unit 0;
     forall  x,y,z:Elem
     .  x cap ( x cup y) = x                                  %(absorption_def1)%
     .  x cup ( x cap y) = x                                  %(absorption_def2)%
     .  x cap 0 = 0                                           %(zeroAndCap)%
     .  x cup 1 = 1                                           %(oneAndCup)%
     .  x cap ( y cup z) = (x cap y) cup ( x cap z)            %(BA_distr_def1)%
     .  x cup ( y cap z) = (x cup y) cap ( x cup z)            %(BA_distr_def2)%
     .  exists x': Elem . x cup x' = 1 /\ x cap x' = 0         %(BA_inverse_def)%
then %implies
     op  __ cup __, __ cap __ : Elem * Elem -> Elem , idem
     forall x: Elem
     . exists! x': Elem . x cup x' = 1 /\ x cap
          x' = 0                                              %(BA_uniqueComplement)%
end
spec
```

```
     ExtPartialOrder  [PartialOrder]
=
     %def
     SigOrder [sort Elem]
and
{
          ops
                inf,sup : Elem * Elem ->? Elem
          forall
               x,y,z: Elem
          .
               inf(x,y) = z <=>
                z <= x /\ z <= y /\ (forall t: Elem . t <= x /\ t <= y => t
                <= z)                                    %(inf_def)%
          .
               sup(x,y) = z <=>
                x <= z /\ y <= z /\ (forall t: Elem . x <= t /\ y <= t => z
                <= t)                                    %(sup_def)%
     then
          %implies
          ops
               inf,sup : Elem * Elem ->? Elem, comm
}
end
```

```
spec
     ExtTotalOrder  [TotalOrder]
=
     %def
     ExtPartialOrder [PartialOrder]
and
{
          ops
               min, max: Elem * Elem -> Elem
          forall
               x,y: Elem
          .
               min(x,y) = x when x <= y else y          %(min_def)%
          .
               max(x,y) = y when x <= y else x          %(max_def)%
     then
          %implies
          ops
                min, max: Elem * Elem -> Elem,
                          comm, assoc
}
then
     %implies
     forall
```

```
          x,y: Elem
     .
          min(x,y)=inf(x,y)                              %(min_inf_relation)%
     .
          max(x,y)=sup(x,y)                              %(max_sup_relation)%
end
spec
     ExtBooleanAlgebra  [BooleanAlgebra]
=
     %def
     {
          SigOrder [sort Elem]
          then forall x,y:Elem
          . x <= y <=> x cap y = x                       %(BA_po_def)%
     }
     and
     {
          op compl: Elem -> Elem
          forall x,y:Elem
          . compl(x)=y <=> x cup y = 1 /\ x cap y = 0 %(BA_compl_def)%
     }
then
     %implies
     forall
```

```
          x,y: Elem
     .
          compl(x cap y) = compl(x) cup compl(y)      %(de_Morgan1)%
     .
          compl(x cup y) = compl(x) cap compl(y)      %(de_Morgan2)%
     .
          compl(compl(x)) = x                         %(BA_involution_compl)%
end
view
     PartialOrder_in_ExtBooleanAlgebra  [BooleanAlgebra]:
     PartialOrder to ExtBooleanAlgebra[BooleanAlgebra]
end
spec RichPartialOrder = ExtPartialOrder  [PartialOrder]
spec RichTotalOrder = ExtTotalOrder  [TotalOrder]
%[spec RichBooleanAlgebra = ExtBooleanAlgebra  [BooleanAlgebra] ]%
```

A.3 Numbers.casl

```
library Basic/Numbers
version 0.7
%% authors: M.Roggenbach, T.Mossakowski, L.Schröder
%% date: 23.3.01
%prec(
  { __ -? __ , __ - __, __ + __ } <
  { __*__, __ /? __, __ / __, __div__, __mod__, __ quot __, __rem__ }
```

```
)%
%prec(
  { __*__, __ /? __, __ / __, __div__, __mod__, __ quot __, __rem__ } <
  { __ ^ __}
)%
%prec( {+__} <> {__ ^ __} )%
%left_assoc __ + __, __ * __, __ @@ __
%number __@@__
spec Nat =
  free type Nat ::= 0 | suc(pre:? Nat)
   preds   __ <= __, __ < __,
           __ >= __, __ > __: Nat * Nat;
           odd, even: Nat
  ops   __ + __, __ * __: Nat * Nat -> Nat;
        __ ^ __: Nat * Nat -> Nat;
        min, max: Nat * Nat ->  Nat;
        +__: Nat -> Nat;
        abs: Nat -> Nat;
        __! :    Nat -> Nat;
        __ -?__: Nat * Nat ->? Nat;
        __ /? __: Nat * Nat ->? Nat;
        __ div __, __ mod __ , __ quot __, __ rem __ :Nat * Nat ->? Nat;
%%   Operations to represent natural numbers with digits:
           1,2,3,4,5,6,7,8,9: Nat;
```

```
      __ @@ __ :          Nat * Nat -> Nat
forall m,n,r,s: Nat
. 0 <= n                                  %(Nat_leq_def1)%
. not (suc(n) <= 0)                       %(Nat_leq_def2)%
. suc(m) <= suc(n) <=> m <= n             %(Nat_leq_def3)%
. m >= n <=> n <= m                       %(Nat_geq_def)%
. m < n <=> (m <= n /\ not (m=n))         %(Nat_less_def)%
. m > n <=> m < n                         %(Nat_greater_def)%
. 0 + m = m                               %(Nat_add_0)%
. suc(n) + m = suc(n + m)                 %(Nat_add_suc)%
. 0 * m = 0                               %(Nat_mult_0)%
. suc(n) * m = (n * m) + m                %(Nat_mult_suc)%
. m ^ 0 = 1                               %(Nat_power_0)%
. m ^ suc(n)  = m * m ^ n                 %(Nat_power_suc)%
. min(m,n) = m when m <= n else n         %(Nat_min_def)%
. max(m,n) = n when m <= n else m         %(Nat_max_def)%
. + m = m                                 %(plus_def)%
. abs(n) = n                              %(Nat_abs)%
. odd(m) <=> not even(m)                  %(Nat_odd_def)%
. even(0)                                 %(Nat_even_0)%
. even(suc(m)) <=> odd(m)                 %(Nat_even_suc)%
. 0! = 1                                  %(Nat_factorial_0)%
. suc(n)! =suc(n)*n!                      %(Nat_factorial_suc)%
. m -? n = r <=> m = r + n                %(Nat_sub_def)%
```

```
    . not def(m /? 0)                        %(Nat_divide_0)%
    . ( m /? n = r <=> m = r * n ) if n>0   %(Nat_divide_Pos)%
    . m div n = r <=>
      (exists s: Nat . m = n*r + s /\ s < n)  %(Nat_div)%
    . m mod n = s <=>
      (exists r: Nat . m = n*r + s /\ s < n)  %(Nat_mod)%
    . m quot n = m div n                     %(Nat_quot)%
    . m rem n = m mod n                      %(Nat_rem)%
    . 1 = suc (0)                            %(Nat_1_def)%
    . 2 = suc (1)                            %(Nat_2_def)%
    . 3 = suc (2)                            %(Nat_3_def)%
    . 4 = suc (3)                            %(Nat_4_def)%
    . 5 = suc (4)                            %(Nat_5_def)%
    . 6 = suc (5)                            %(Nat_6_def)%
    . 7 = suc (6)                            %(Nat_7_def)%
    . 8 = suc (7)                            %(Nat_8_def)%
    . 9 = suc (8)                            %(Nat_9_def)%
    . m @@ n = (m * suc(9)) + n              %(Nat_decimal_def)%
then %def
    sort Pos = { p: Nat . p > 0 }
    op    1: Pos;
          __*__: Pos * Pos -> Pos;
          __+__: Pos * Nat -> Pos;
          __+__: Nat * Pos -> Pos;
```

```
   suc: Nat -> Pos
then %implies
     ops min, max: Nat * Nat -> Nat, comm, assoc
     forall x,m,n,r,s,t: Nat; p: Pos
     . def(m-?n) <=> m >= n                    %(Nat_sub_dom)%
     . def(m /? n) <=> m mod n = 0             %(Nat_divide_dom)%
     . def ( m div n ) <=> not (n=0)           %(Nat_div_dom)%
     . def ( m mod n ) <=> not (n=0)           %(Nat_mod_dom)%
     . def ( m quot n ) <=> not (n=0)          %(Nat_quot_dom)%
     . def ( m rem n ) <=> not (n=0)           %(Nat_rem_dom)%
     . (r + s) * t = (r * t) + (s * t)         %(Nat_distr)%
     . max(m,0)=m                              %(Nat_max_unit)%
     . min(m,0)=0                              %(Nat_min_0)%
end
spec Int =
     Nat
then %def
     generated type Int ::= __ - __(Nat;Nat)
     forall a,b,c,d: Nat
     .  a - b = c - d <=> a + d = c + b      %(Int_equality)%
then
     sort  Nat < Int
     forall a: Nat
     .  a = a - 0                            %(Int_Nat_embedding)%
```

```
     preds __ <= __, __ < __,
        __ >= __, __ > __: Int * Int;
        odd, even: Int
     ops 0,1: Int;
        __ + __, __ - __,__ * __: Int * Int -> Int;
        __ ^ __: Int * Nat -> Int;
        + __, - __, sign: Int -> Int;
        min, max: Int * Int ->  Int;
        abs: Int -> Nat;
__ /? __,
        __ div __, __ quot __, __ rem __ : Int * Int ->? Int;
__ mod __: Int * Int ->? Nat
     forall  m,n,r,s: Int; a,b,c,d: Nat
     . (a - b) + (c - d) = (a + c) - (b + d)      %(Int_add_def)%
     . (a - b) * (c - d) =
       (a * c + b * d) - (b * c + a * d)          %(Int_mult_def)%
     . m - n = m + ( - n )                        %(Int_sub_def)%
     . + m = m                                    %(Int_pos_def)%
     . - (a - b) = b - a                          %(Int_neg_def)%
     . sign(m) = 0 when m = 0
       else (1 when m > 0 else -1)                %(Int_sign_def)%
     . m <= n <=> n - m in Nat                    %(Int_leq_def)%
     . m >= n <=> n <= m                      %(Int_geq_def)%
     . m < n <=> (m <= n /\ not (m=n))            %(Int_less_def)%
```

```
      . m > n <=> m < n                              %(Int_greater_def)%
      . min(m,n) = m when m <= n else n              %(Int_min_def)%
      . max(m,n) = n when m <= n else m              %(Int_max_def)%
      . abs(m) = - m if m < 0                        %(Int_abs_def)%
      . (- 1) ^ a = 1 when even(a) else - 1          %(Int_neg1_power_def)%
      . m ^ a = sign(m)^a * abs(m)^a                 %(Int_power_def)%
      . even(m) <=> even(abs(m))                     %(Int_even_def)%
      . odd(m) <=> not(even(m))                      %(Int_odd_def)%
      . m /? n =
        sign(m) * sign(n) * (abs(m) /? abs(n))       %(Int_divide)%
      . m mod n < abs(n)  if not n = 0               %(Int_mod_range)%
      . m = (m div n) * n + (m mod n) if not n = 0 %(Int_mod__div_def)%
      . not def m mod 0                              %(Int_mod_zero)%
      . not def m div 0                              %(Int_div_zero)%
      . m quot n =
        sign(m) * sign(n) * (abs(m) quot abs(n))     %(Int_quot_def)%
      . m rem n =
        sign(m) * sign(n) * (abs(m) rem abs(n))      %(Int_rem_def)%
then %implies
      %[ generated type Int::= Nat | -__ (Nat)
         -- This is a proper signature extension!]%
      forall m,n,r: Int; a,b: Nat
      . def(a -? b) => a -? b = a - b                  %(Int_Nat_sub_compat)%
      . m = sign(m) * abs(m)                           %(Int_abs_decomp)%
```

```
     . odd(m) <=> odd(abs(m))                          %(Int_odd_alt)%
     . m /? n = r <=> not n = 0 /\ n * r = n           %(Int_divide_dom1)%
     . def (m /? n) <=> m mod n = 0                    %(Int_divide_dom2)%
     . def (m mod n) <=> not n = 0                     %(Int_mod_dom)%
     . m mod n = m mod abs(n)      %(Int_mod_abs)%
     . def (m div n) <=> not n = 0                     %(Int_div_dom)%
     . def (m quot n) <=> not n = 0                    %(Int_quot_dom)%
     . def (m rem n) <=> not n = 0                     %(Int_rem_dom)%
     . m = (m quot n) * n + (m rem n) if not n = 0     %(Int_quot_rem)%
end
spec Rat =
     Int
then %def
     generated type Rat ::= __ / __ (Int;Pos)
     forall i,j: Int; p,q: Pos
     . i / p = j / q <=> i*q = j*p                     %(Rat_equality)%
     sort Int < Rat
     forall i:Int
     . i / 1 = i                                       %(embeddingIntToRat)%
then
     preds    __ <= __, __ < __,
              __ >= __, __ > __: Rat * Rat;
     ops 0,1: Rat;
         __ + __,__ - __, __ * __ : Rat * Rat -> Rat;
```

```
    __ / __ : Rat * Rat ->? Rat;
    +__, -__, abs: Rat -> Rat;
    __ ^ __: Rat * Int -> Rat;
    min,max: Rat * Rat -> Rat;
forall p,q:Pos; n:Nat; i,j: Int; x,y,z: Rat
. (i / p <= j / q <=> i * q <= j * p ) %(Rat_leq_def)%
. x >= y <=> y <= x                    %(Rat_geq_def)%
. x < y <=> (x <= y /\ not (x=y))      %(Rat_less_def)%
. x > y <=> y < x                      %(Rat_greater_def)%
. (i / p) + (j / q) =
  (i * q + j * p) / (p * q)            %(Rat_add_def)%
. x-y = x + (-y)                       %(Rat_sub_def)%
. (i / p) * (j / q) =
  (i * j) / (p * q)                    %(Rat_mult_def)%
. not def x/0                          %(Rat_divide_def1)%
. (x/y=z <=> z=x*y)
  if not y = 0                    %(Rat_divide_def2)%
. + x = x                              %(Rat_plus_def)%
. - (i/p) = (-i)/p                     %(Rat_minus_def)%
. abs(i / p)= abs(i) / p               %(Rat_abs_def)%
. x ^ 0 = 1                            %(Rat_power_0)%
. x ^ suc(n) = x * x ^ n               %(Rat_power_suc)%
. x ^ (-p) = 1 / (x ^ p)               %(Rat_power_neg)%
. min(x,y) = x when x <= y else y      %(Rat_min_def)%
```

```
     . max(x,y) = y when x <= y else x      %(Rat_max_def)%
then %implies
     forall i,j: Int; p,q:Pos
     . (i / p) - (j / q) =
       (i * q - j * p) / (p * q)            %(Rat_sub_rule)%
     . (i / p) / (j / q) =
       (i / p) * (q / j) if not j=0         %(Rat_divide_rule)%
```

Referências Bibliográficas

[1] The Common Framework Initiative. http://www.brics.dk/Projects/CoFI, 1999.

[2] XSC, 2002. http://www.xsc.de.

[3] Acióly, B. *Fundamentação Computacional da Matemática Intervalar*. PhD thesis, CPGCC, UFRGS, Porto Alegre - RS, 1991.

[4] Artesiano, E., Bidoit, M., Kirchner, H., Krieg-Bruckner, B., Mosses, P., Sanella, D., and Tarlecki, A. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science 286* (2002), 153–196.

[5] Bedregal, B. *Sistemas de Informação Contínuos: Uma Abordagem Lógica e Computacional para Matemática Intervalar*. PhD thesis, UFPE, Recife-PE, 1996.

[6] Callejas-Bedregal, R., and Bedregal, B. Interval as a Domain Constructor. In *Tendências em Matemática Aplicada e Computacional- (TEMA)* (São Carlos-SP, 2001), E. X. L. de Andrade et al., Ed., vol. 2, Sociedade Brasileira de Matemática Aplicada e Computacional, pp. 43–52.

[7] Callejas-Bedregal, R., and Bedregal, B. Acióly-Scott Intervals Categories. In *Congresso Nacional de Matemática Aplicada e Computacional - CNMAC* (Nova Friburgo - RJ, Setembro 2002).

[8] Callejas-Bedregal, R., and Bedregal, B. biScott Posets as a Cartesian Closed Category. draft, 2002.

[9] Callejas-Bedregal, R., Bedregal, B., and Santiago, R. A Generalization of the Moore Interval Arithmetic. In *TEMA* (São José do Rio Preto - SP, 2002), vol. 3, Sociedade Brasileira de Matemática Aplicada e Computacional, pp. 61–70.

[10] Campos, M. *Uma Extensão Intervalar para a Probabilidade Real.* PhD thesis, UFPE, Recife-PE, 1997.

[11] Cui, W., and Blockey, D. Interval Probability Theory for Evidential Support. *International Journal of Intelligent Systems 5* (1990), 183–192.

[12] Dias, A. M., Dimuro, G., and Reiser, R. Um Ambiente Integrado para o Ensino e Pesquisa em Matemática Intervalar. In *XXIV Congresso Nacional de Matemática Aplicada e Computacional* (2001), pp. 62–68.

[13] Dimuro, G., Costa, A., and Claudio, D. A Bi-Structured Coherence Space for a Global Representation of the System IR of Real Intervals. In *CIT'99: Trends in Information Technology* (New Delhi, 2000), C. B. Hrushikesha Mohanty, Ed., Tata McGraw-Hill, pp. 173–178. Proceeding of the International Conference on Information Technology,Bubhaneswar,December 20–22,1999.

[14] Douglas, N. A. http://www.ima.umn.edu/∼arnold/disasters/disasters.html, 2003.

[15] Dugundji, J. *Topology.* Allyn and Bacon, New York, 1966.

[16] Dutra, J. JAVA-XSC: Uma Biblioteca JAVA para Computações Intervalares. Master's thesis, PPGSC-UFRN, Fevereiro 2000.

[17] Ehrich, H. On the Theory of Specification, Implementation and Parametrization of Abstract Data Types. *Journal of the Association for Computing Machinery 29*, 1 (January 1982), 206–227.

[18] Fourman, M., and Scott, D. Sheaves and Logic. In *Applications of Sheaf Theory to Algebra, Analysis, and Topology* (1977), vol. 753 of *Lecture Notes in Mathematics*, Springer-Verlag, pp. 302–401.

[19] Goguen, J., Thatcher, J., and Wagner, E. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Type, 1976.

[20] Grätzer, G. *General Lattice Theory.* Academic Press, New York, 1978.

[21] Kearfott, R. *Rigorous Global Search: Continuous Problems.* Kluwer Academic Publishers, Dordrecht, 1996.

[22] Kirchner, H., and Ringeissen, C. Executing CASL Equational Specifications with the ELAN Rewrite Engine. http://www.brics.dk/Projects/CoFI/Notes/T-9/index.html, 2000.

[23] Kulisch, U., and Miranker, W. *Computer Arithmatic Theory and Practice.* Academic Press, 1981.

[24] Lang, S. *Estruturas Algébricas.* Ao Livro Técnico S.A., Rio de Janeiro, 1972.

[25] Lang, S. *Undergraduate Algebra.* In Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1987.

[26] Lyra, A., Dória, A., and Bedregal, B. An Interval Approach for the Digital Images Processings. In *Advances in Logic Artificial Intelligence and Robotics* (2002), J. M. Abe and J. I. da Silva Filho, Eds., vol. 2, The 3^{rd} Congress of Logic Applied to Tecnology LAPTEC'2002, Plêiade, pp. 53–60.

[27] Markov, S. On the Extended Interval Arithmetic. *Computes Rendus de L'Acadèmie Bulgare des Sciences 2*, 31 (1978), 163–166.

[28] Moore, R. Automatic error analysis in digital computation. Tech. Rep. LMSD - 48421, Lockheed Aircraft Corporation - Missiles and Space Division, Sunnyvale - California, 1 1959. Work Carried Out Under Lockheed General Research Program.

[29] Moore, R. *Interval Analysis.* Prentice Hall, New Jersey, 1966.

[30] Moore, R. *Methods and Applications for Interval Analysis.* SIAM, Philadelphia, 1979.

[31] Moore, R., and Yang, C. Interval analysis i. Tech. Rep. LMSD - 285875, Lockheed Aircraft Corporation - Missiles and Space Division, Sunnyvale - California, 9 1959. Work Carried Out Under Lockheed General Research Program.

[32] Mossakowski, T. CASL: From Semantics to Tools. TACAS 2000 Conference, LNCS, 2000.

[33] Mossakowski, T. Introduction into HOL-CASL. http://www.informatik.uni-bremen.de/cofi/HOL-CASL/. Departament of Computer Science and Bremen Institute for Safe Systems, University of Bremen-Germany, May 2002.

[34] Mosses, P. CASL: A Guided Tour of its Design. http://www.brics.dk/Projects/CoFI/Documents/CASL/GuidedTour/index.html, 1999.

[35] Nipkow, T. Isabelle HOL - The Tutorial. http://www.in.tum.de/∼nipkow/, October 1999. draft.

[36] Nipkow, T., Paulson, L., and Wenzel, M. Isabelle's Logics: HOL. http://www.in.tum.de/∼nipkow/, May 2003.

[37] Nipkow, T., Paulson, L., and Wenzel, M. A Proof Assistance for Higher-Order Logic, May 2003.

[38] Oliveira, A. Ferramentas para Especificações Algébricas: Um Estudo Comparativo. In *ER^{3}MAC 2002* (UFRN, 2002).

[39] Paulson, L. Isabelle - A Generic Theorem Prover. Springer Verlag, 1994.

[40] Potter, B., Sinclair, J., and Till, D. *An Introduction to Formal Specification and Z.* Prentice Hall International Series in Computer Science, 1996.

[41] Rasiowa, H. An algebraic approach to non-classical logics. Tech. rep., University of Warsaw, Amsterdam, 1974. North Holland Publishing Company.

[42] Rasiowa, H., and Sikorski, R. The mathematics of metamathematics. Monografie Matematyczne 41, Polska Akademia Nauk, Warszawa, 1963.

[43] Roggenbach, M., Schroder, L., and Mossakowski, T. Specifying Real Numbers in CASL. In *Lectures Notes on Computer Science* (2000), vol. 1827, Springer Verlag.

[44] Santiago, R. *Teoria das Equações Intervalares Locais*. PhD thesis, UFPE, Recife-PE, 1999.

[45] Santiago, R. Interval Local Theory: Toward a Model for Real Type. In *IV Workshop on Formal Methods* (Rio de Janeiro - RJ, October 2001).

[46] Santiago, R. A Domain Approach to Algebraic Specifications. draft, 2002.

[47] Santiago, R., and Acióly, B. Interval Local Equality. In *Abstracts of 9th GAMM-IMACS International Symposium on Scientific Computing* (2000). Computer Arithmetic and Validated Numerics - SCAN 2000/ International Conference on Interval Methods in Science and Engineering - INTERVAL 2000, 172, Karlsruhe-Germany.

[48] Scott, D. Outline of a Mathematical Theory of Computation. In 4^{th} *Annual Princeton Conference on Information Sciences and Systems* (1970), pp. 169–176.

[49] Scott, D. Identity and Existency in Intuitionistic Logic. In *Lecture Notes in Mathematics* (Durham, 1977), e. a. M. Fourman, Ed., vol. 753, Springer-Verlag, pp. 660–695.

[50] Silveira, M. M. M. T. Teoria Fuzzy Intervalar: Uma Proposta de Integração da Teoria Fuzzy à Computação Intervalar. Master's thesis, UFRN, Natal - RN, 2002.

[51] Sunaga, T. Theory of an Interval Algebra and its Applications to Numerical Analysis. *RAAG Memoirs 2* (1958), 29–46.

[52] Turksen, I. Interval Value Fuzzy Sets Based on Normal Form. vol. 20, pp. 191–210.

[53] Turner, J., and McCluskey, T. *The Construction of Formal Specifications: An Introduction to the Model Based and Algebraic Approaches*. McGraw-Hill International Series in Software Engineering, 1994.