Desenvolvimento de Programas por Composição de Implementações utilizando Lógica Clássica e *Lógica Default*

Marcelo Antonio Thomaz de Aragão

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovado por :

Sheila Regina Murgel Veloso, D.Sc.
(Presidente)

Paulo Augusto Silva Veloso, Ph.D.

Mário Roberto Folhadela Benevides, Ph.D.

Roberto Ierusalimschy, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
Setembro de 1993

ARAGÃO, MARCELO ANTONIO THOMAZ DE

Desenvolvimento de Programas por Composição de Implementações utilizando Lógica Clássica e *Lógica Default* [Rio de Janeiro] 1993.

XI, 209 p., 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1993

Tese – Universidade Federal do Rio de Janeiro, COPPE.

1 – Desenvolvimento Formal de Programas
2 – Tipos Abstratos de Dados
3 – Lógica Clássica
4 – Lógica de *Defaults*
I. COPPE/UFRJ
II. Título (série).

A Luiz Carlos e Celeste,
Adriana e Cristiana.

**** ****

Agradecimentos

À Sheila, caso o trabalho estivesse para ser iniciado hoje, a minha opção por Orientação seria exatamente a mesma.

Ao Paulo Veloso, pela receptividade a nossas idéias, pela colaboração sempre interessada e pelo suporte fundamental para a consecução deste trabalho.

Ao Mário e ao Gerson, pelo incentivo constante.

Ao Evande, ao Odinaldo e ao Helano, amigos e companheiros de sempre.

A todo o pessoal que passou pela "embaixada" cearense no Rio, e a todos os novos amigos.

Ao Wamberto, Guy, Edson e Clécio pelo apoio na decisão de somar o curso de Mestrado à graduação.

A todos que confiaram e torceram para o nosso êxito, em especial a Giselda.

A minha família, a quem este trabalho é dedicado, pela presença distante e nunca ausente.

A Deus pela inspiração e perseverança.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

Desenvolvimento de Programas por Composição de Implementações utilizando Lógica Clássica e *Lógica Default*

Marcelo Antonio Thomaz de Aragão

Outubro, 1993

Orientadora : Sheila Regina Murgel Veloso
co-Orientador : Paulo Augusto Silva Veloso
Programa : Engenharia de Sistemas e Computação

Este trabalho aborda a composição de passos de implementação entre especificações lógicas para Tipos Abstratos de Dados (TAD) e investiga como um mecanismo de composição particular pode efetivamente ser aplicado a uma série de circunstâncias bem conhecidas no desenvolvimento e verificação de programas. A corretude destes mecanismos é formalmente confirmada pelo *Teorema da Modularização* e pelo *Teorema da Construtibilidade.*

A *Lógica de Primeira Ordem Poli-Sortida* é inicialmente usada como linguagem de especificação para TAD, visando raciocinar sobre o desenvolvimento de programas em uma base lógico-formal. Nesse contexto, o principal desafio é demonstrar algumas alternativas factíveis para afirmar a consistência dos resultados desejados, mesmo que a conservatividade não seja imposta. Adicionalmente, este texto propõe uma estratégia construtiva para realizar qualquer passo canônico, por decomposição sucessiva da especificação em outra menos elaborada, até ser produzida uma especificação suficientemente simples para ser trivialmente implementada. Então, compondo passos canônicos sempre por sobre o subjacente, pode ser conseguida uma implementação para a especificação original.

A mesma dicussão é transposta para um contexto não monotônico, i.e. usando a *Lógica Default* na especificação de TAD. As noções clássicas de Interpretação e Extensão Conservativa são reformuladas em *Lógica Default*, obtendo-se uma versão do *Teorema da Modularização* e do *Teorema da Construtibilidade*, em termos de teorias *default.* Examina-se ainda como o desenvolvimento de programas sucede na ausência da monotonicidade, e quais vantagens advem desta maior flexibilidade.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

Program Development by Composing Implementations using Classic Logic and Default Logic

Marcelo Antonio Thomaz de Aragão

October, 1993

Thesis Supervisor : Sheila Regina Murgel Veloso
Thesis co-Supervisor : Paulo Augusto Silva Veloso
Department : Computing and Systems Engineering

This work aims mostly to reason about composing implementation steps between logical specifications for Abstract Data Types, and how a particular composition mechanism may effectively be applied to a sort of well-known circumstances in program development and verification. The correctness of these mechanisms is formally assured by the *Modularization Theorem* and the *Constructibility Theorem.*

Many Sorted First Order Logic is initially used as a specification language for ADT, as framework for reasoning about program development in logic-formal basis. In this context, the main challenge is to demonstrate some feasible alternatives to claim the consistency of the disered outcomes, even though the conservativeness is not imposed *a priori.* Futhermore, this text proposes a constructive strategy to acheive any canonical step, by decomposing sucessively the specification to be implemented, into less elaborated ones, yielding some specification, simple enough to be trivially implemented. Then, by composing canonical steps, one step over the underlying antecedent one, a implementation to the original specification will be achieved.

The present dicussion is transposed into a non monotonic context, i.e. *Default Logic* will thereafter be used to specify ADT. The classical notions of Interpretation and Conservative Extension are reformulated in *Default Logic*, to allow both, the *Modularization Theorem* and the *Constructibility Theorem*, to be reestablished in terms of default theories. This examines how program development succeeds without monotonicity, and what advantages can be profited with such a flexibility.

Índice

Figuras

Capítulo I

Prólogo

Indubitavelmente, o desenvolvimento de progamas precisa ser conduzido dentro de uma disciplina formalmente definida, se há pretensão de fornecer resultados corretos com segurança e eficiência. Contudo, ainda argumenta-se contra a complexidade teórica que o rigor formal costuma exigir, questionando se compensa o esforço para empregar métodos formais em circunstâncias reais. Antes que alguma conclusão seja consensualamente aceita resta muito ainda a ser dito e feito.

O presente trabalho ambiciona evidênciar possibilidades reais de manter a disciplina lógico-formal no desenvolvimento de programas, não obstante limitações teóricas que lhes sejam inerentes. O espaço necessário para a intervenção criativa pode ser precisamente definido sem comprometer a natureza formal do desenvolvimento; portanto, a preocupação primordial deve ser encontrar, mesmo dentro do formalismo lógico, condições mais flexíveis, que não penalizem excessivamente o desenvolvimento.

O estudo será inteiramente fundamentado sobre o método de *Desenvolvimento do Programas com Tipos Abstratos de Dados*, segundo [Vel87], para o qual a composição de implementações irá desempenhar um papel relevante na definição de uma representação computacional definitiva, partindo de uma descrição abstrata do problema original.

A ênfase no texto resume-se sempre a aplicar mecanismos de composição de implementações em circunstâncias que, intuitivamente, tendem a requerer mais liberdade de expressão. Aproveitando melhor o formalismo lógico consegue-se alternativas para garantir a consistência dos resultados obtidos, sem impor a conservatividade. Por outro lado, escolhendo um formalismo que se adequa melhor, é contornada com relativa facilidade as limitações intrínsecas à monotonicidade.

No primeiro caso tratado aqui será adotada como linguagem de especificação a *Lógica de Primeira Ordem Poli-Sortida*, onde, então, o principal objetivo compreende definir com precisão formal resultados consistentes, ainda que não haja como pré-requesito a conservatividade. No segundo caso, as idéias postuladas antes são transpostas para um contexto não monotônico. Entre as opções encontadas na literatura, a *Lógica Default*, conforme proposta em [Rei80], foi preferida, e assim, considerando-a, os mecanismos de composição poderiam ser repensados na ausência de monotonicidade.

O capítulo II, apenas introdutório, objetiva situar aonde exatamente este trabalho pertende acrescentar-se. Para tanto, descreve brevemente o método de *Desenvolvimento do Programas com Tipos Abstratos de Dados*, e a sua repectiva verificação de corretude, no qual encontra-se a principal motiva,ão deste trabalho. Ainda será conceituado o *Passo Canônico* (definição formal para a noção intuitiva de implementação por passo de refinementos). Além disto, uma notação é padronizada para especificar Tipos Abstratos de Dados por todo o texto.

Os mecanismos de composição de implementações são apresentados e comparativamente analisados no capítulo III. Também, são enunciados e demonstrados o Teorema da Modularização e o Teorema da Construtibilidade, como forma de certificar que são válidos os resultados obtidos utilizando um ou outro mecanismo.

O capítulo IV dedica-se a exemplificar aplicações para o mecanismo de composição por sobre uma implementação subjacente, apresentado em III. Pricipalmente, é destacada a utilização de estratégias construtivas para definir passos canônicos elaborados, a partir de outros relativamente menos complexos.

Toda a dicussão anterior muda de contexto quando é esquecida a monotonicidade tranpondo, no capítulo V, para a *Lógica Default* os mecanismos de composição de implementação. Antes as noções lógicas de interpretação e extensão conservativa precisam ser apropriadamente reformuladas para teorias *default*, sem o que o passo canônico sequer poderia ser definido.

No capítulo VI, novamente retorna-se a observações mais pragmáticas, observando como a construção de uma implementação comporta-se livre das limitações inerentes à monotonicidade.

Finalmente, o Epílogo encerra este trabalho fazendo um retrospecto dos principais resultados e, logo após, indicando futuras direções para dar sequência ao estudo desenvolvido aqui.

Capítulo II

Desenvolvimento de Programas como uma Atividade Formal

Este capítulo situa a aplicabilidade das idéias a serem desenvolvidos neste trabalho dentro do desenvolvimento formal de programas, enquanto revisiona muito brevemente o que há feito, e aonde este trabalho se pretende adicionar. As noções básicas expostas aqui formam a intuição imprescindível para compreender o restante do texto.

Inicialmente, a primeira seção introduz o método de *Desenvolvimento de Programas com Tipos Abstratos de Dados,* apresentando objetivamente suas etapas. A fim de possibilitar a utilização prática deste método, é preciso formalizar uma notação capaz de representar sintaticamente e possibilitar a implementação de tipos abstratos de dados. Este é o objetivo das duas seções seguintes. No final, é estabelecido como proceder a verificação dos resultados conseguidos com a aplicação deste método.

II.1 Desenvolvimento de Programas com Tipos Abstratos de Dados

Um problema passível de solução automaticamente computável caracteriza-se por uma classe de modelos, tomada como argumento, que deve ser transformada, em conformidade com uma relação bem definida, em uma outra classe de modelos que constitue a solução esperada para o referido problema.

Admitindo-se que esta solução é calculável, em seu sentido algorítmico, solu-

cionar um problema é precisamente encontrar uma função computável, capaz de efetuar sempre a transformação de instâncias da classe de argumentos em instâncias adequadas contidas na classe de modelos solução.

Todavia, apelando-se para representações sintáticas, pode-se prescíndir da referência direta às classes de modelos. Logo, a especificação para a função de tranformação pretendida será definida simplesmente a partir das representações para a classe de argumentos e para a classe de soluções. A partir desta assim chamada *descrição da solução,* é concebido um algoritmo que representa a computação desta função.

A construção deste algoritmo é favorecida quando do emprego de métodos formais. Dentro deste espaço aparece o *Desenvolvimento de Programas com Tipos Abstratos de Dados,* segundo está proposto por Veloso, em [PV83] e apresentado em [Vel87]. Este método de programação foi estabelecido para "suportar a idéia de desenvolvimento descendente (*top down*) por refinamentos sucessivos, decompondo o problema com base no reconhecimento de abstrações".

A característica angular aqui é a fatoração da programação em duas partes independentes :

Programa Abstrato envolvendo operações abstratas sobre objetos abstratos; e

Módulo de Implementação estruturando tipos para representar estes objetos abstratos e codificando suas operações.

Para este método funcionar a contento, é preciso enfatizar a independência entre as duas partes. Assim, o programa abstrato limita-se a referenciar objetos especificamente escolhidos, e somente manipulá-los através do repertório de operações definido para cada objeto. Assim a representação de cada um fica encapsulada e protegida contra manipulações indesejáveis. O programa abstrato situa-se, portanto, mais próximo ao domínio real do problema. Ao escrevê-lo, são empregadas informações sobre o presumível comportamento dos objetos abstratos, conforme descrito por suas operações.

```
Program Sched;
uses    Dispatcher,Buffer,Process;
var     Dpt : Dispatcher;
        Buff : Buffer;
        RP,AP : Buffer;
```

```
begin
   Dpt := inicialize;
   while true do
      ¬ empty(Buff)   ⟶  RP := first(Buff);          //retira o pedido do Buffer
                         Buff := send(Buff);          //libera o pedido do Buffer
                         Dpt := schedule(Dpt,RP); []  // escalona o pedido
      findprocess(Dpt) ⟶  AP := next(Dpt);      // seleciona o pedido de impressão
                         Dpt := release(Dpt);    // libera o pedido do Dispatcher
                         print(AP);              // imprime o pedido solicitado
   end while
end
```

Acima, está codificado um exemplo de programa abstrato, constituido basicamente por um laço contínuo onde dois grupos de comandos são escolhidos não deterministicamente para executar, segundo a satisfação dos "guardas" [1] associados (representado pelos predicados findprocess(Dpt) e empty(Buff)). Este programa abstrato realiza duas operações distintas: ou retira um processo de um *Buffer* e o escalona propriamente em um estrutura *Dispatcher* que armazena os processos esperando para serem atendidos, ou seleciona um entre os processos aguardando o servidor, para ser atendido, liberando em seguida este processo do *Dispatcher*. Cada processo significa um pedido de impressão requesitado a um servidor de impressão. Este servidor é que executa continuamente o programa abstrato *Sched*.

Não há novidades quanto a esta linguagem de codificação[2], nem há nenhuma razão particular para esta escolha. É importante identificar facilmente alguns objetos abstratos (e.g. *Dispatcher*,*Buffer* e *Process*) e algumas operações abstratas (e.g. funções como *send* e *schedule* e predicados como *findprocess* e *empty*), que foram utilizadas na construção de *Sched*, mas ainda não estão convenientemente definidas.

O módulo de implementação contém a representação destes objetos em termos de objetos mais primitivos (no sentido que são suportados por uma linguagem algoritmicamente executável), e a descrição das operações abstratas em termos de procedimentos (nesta linguagem) que manipulam as mesmas representações escolhidas. A preocupação na codificação do módulo de implementação resume-se a como devem se comportar cada

[1] a mesma noção de guardas herdada de Djkistra

[2] A notação utilizada para o programa abstrato pretende uma certa semelhança com CSP (*Communicating Sequential Process*), definida por Hoare (ver [Hoa85])

conceito, abstraindo-se de seu uso efetivo no programa abstrato.

O módulo de implementação pode ser melhor compreendido como uma camada sobre o sistema/máquina subjacente, na qual pode ser programado diretamente o programa abstrato.

O ponto crucial deste método reside exatamente na comunicação entre ambas as partes, sem contudo, comprometer a independência de cada uma. Pensando no desenvolvimento, quando a concepção do programa abstrato precede a construção do módulo de implementação, a pergunta a ser feita é como construir um módulo de implementação apropriado para suportar a computação de um programa abstrato particular?

A resposta parece estar bem definida. O programa abstrato é construido presumindo que existem modelos satisfazendo algumas propriedades específicas, relativas aos objetos e operações abstratas. O módulo de implementação, enquanto isto, tem como função mostrar como encontrar tais modelos a partir de realizações possíveis para a linguagem mais primitiva. Esta classe de modelos significa o que foi admitido na concepção do programa abstrato, e deve ser garantido na construção do módulo de implementação. Logo, esta classe de modelos é que atua como elo de ligação entre um programa abstrato e o seu módulo de implementação.

Qualquer tentativa de desenvolver um módulo de implementação deve, portanto, identificar quais realizações suportam a computação executada pelo programa abstrato, e definir como induzí-las a partir da existência de realizações para a linguagem algoritmicamente executável escolhida.

Idealmente, isto deveria ser obtido abstraindo-se completamente das particularidades de codificação, adiando ao máximo qualquer detalhe irrelevante ao desenvolvimento. O produto final poderia ser uma especificação formal, que fosse satisfatível (porque suas realizações são responsáveis por induzir a classe de modelos para os objetos e operações abstratas). Assim, estaria descrita não apenas um único e possível módulo de implementação, mas toda uma classe de módulos, onde um seria efetivamente codificado, dependendo de conveniências em cada instalação particular. Esta especificação para o módulo de implementação chama-se de *implementação declarativa.*

A implementação declarativa deve fornecer todas as informações necessárias à codificação, porém mantendo a solução do problema ainda em um nível "abstrato"[3]. A

[3]O termo **abstrato** é de caráter sempre relativo, e indica um grau de independência do nível lingüístico escolhido para codificação

implementação declarativa orienta a codificação do módulo de implementação, da mesma forma que a descrição da solução serviu para projetar o programa abstrato.

A fatoração do desenvolvimento em duas etapas deu origem a correspondentes tanto no nível declarativo, descrição da solução e implementação declarativa, quanto no nível operacional, respectivamente programa abstrato e módulo de implementação.

Especificação	Codificação
Descrição da Solução	Programa Abstrato
Implementeção Declarativa	Módulo de Implementação

É desnecessariamente complicado lidar diretamente com classes de modelos; é preferível encontrar um meio para referenciá-las, sem importunar-se com enumerar ou decidir suas instâncias (na verdade, isto é impraticável). Entretanto, classes de modelos podem ser associadas a Tipos Abstratos de Dados, pois formalmente,

Definição II.1.1 (Tipos Abstratos de Dados (TAD)) *um tipo abstrato de dados é uma classe de estruturas, e.g.* $\mathcal{T}$*, sobre uma determinada linguagem* $\mathcal{L}$*, fechada pela existência de isomorfismos*[4]*, ou seja, considerando* $\Re$ *e* $\aleph$ *duas estruturas em* $\mathcal{L}$*, se* $\Re \in \mathcal{T}$ *e existe um isomorfismo* ζ *de* $\Re$ *sobre* $\aleph$*, então* $\aleph \in \mathcal{T}$*.*

Se um modelo pertence a um TAD, qualquer estrutura para a linguagem a ele isomorfa obrigatoriamente pertence também a classe de modelos denotada pelo TAD.

Logo, tanto a classe de modelos intermedia a ligação do programa abstrato com o módulo de implementação, quanto a classe de realizações admissíveis para a linguagem primitiva e quanto os modelos para a implementação declarativa podem ser associados a Tipos Abstratos de Dados.

No programa abstrato *Sched* destacam-se claramente três tipos abstratos de dados – $Dispatcher[Process]$,$Buffer[Process]$ e $Process$. É preferível caracterizá-los isoladamente para visualizar melhor as instâncias de cada um, e permitir uma modularização ainda maior na construção do módulo de implementação.

Finalizando, desenvolver programas com tipos abstratos de dados, resume-se sinteticamente à seguinte seqüência de etapas :

1. Encontrar uma descrição precisa para a solução do problema proposto;

[4]homomorfismos bijetivos (ver [End72])

2. A partir desta descrição codificar a solução do problema como um algoritmo (ou programa), criando para tanto um repertório de objetos e operações abstratas;

3. Identificar um tipo abstrato de dados, por exemplo $\mathcal{T}$, provendo a semântica para os objetos e operações abstratas concebidos na etapa anterior;

4. Definir um outro TAD, chamado talvez $\mathcal{T}'$, para representar as estruturas admissíveis na linguagem escolhida para o codificação definitiva;

5. Partindo de $\mathcal{T}$ e $\mathcal{T}'$ encontrar um TAD $\mathcal{D}$, que seja a classe de modelos satisfazendo uma especificação para a implementação declarativa Δ;

6. Codificar o módulo de implementação a partir da implementação declarativa Δ, encontrada na etapa anterior.

A próxima seção enfoca como representar Tipos Abstratos de Dados, como $\mathcal{T}, \mathcal{T}'$ e $\mathcal{D}$, e a seguinte elucida como construir a especificação Δ a partir das representações de $\mathcal{T}$ e $\mathcal{T}'$.

II.2 Especificação de Tipos Abstratos de Dados

A necessidade de representar sintaticamente classes de modelos motiva a formalização de uma notação capaz de descrever Tipos Abstratos de Dados, e argumentar sobre seu comportamento observável. Abstraindo-se de qualquer preocupação injustificavelmente antecipada com a codifição, é possível representar exatamente a semântica de um TAD descrevendo a sua linguagem e o conjunto de todas as propriedades satisfeitas por todas as suas instâncias.

Neste caso mais específico, defini-se como linguagem o conjunto de todas as sentenças formadas a partir do repertório de objetos e operações em questão. Assim, através de sentenças expressando como estes objetos podem ser manipulados legitimamente por suas operações, ficam estabelecidas o que, intuitivamente, seriam as propriedades válidas do conceito sendo representado.

Evitando lidar com o conjunto de todas as sentenças satisfeitas por uma classe de modelos (a teoria), pois um número infinito está sempre envolvido, este conjunto poderia ser equivalentemente substituido por um número mais razoável de sentenças que são fundamentais, os assim chamados axiomas, na suposição que exista um mecanismo

dedutível, que permite sempre inferir todas as demais propriedades válidas, não escritas explicitamente.

Os axiomas são finalmente a caracterização pretendida para a semântica de um TAD, do mesmo modo que algumas regras de formação evitam a necessidade de enumerar todas as sentenças da linguagem.

Desta forma, ficam bem claros dois aspectos da representação de uma TAD, a especificação da linguagem – Especificação Sintática; e a especificação do comportamento observável – Especificação Semântica.

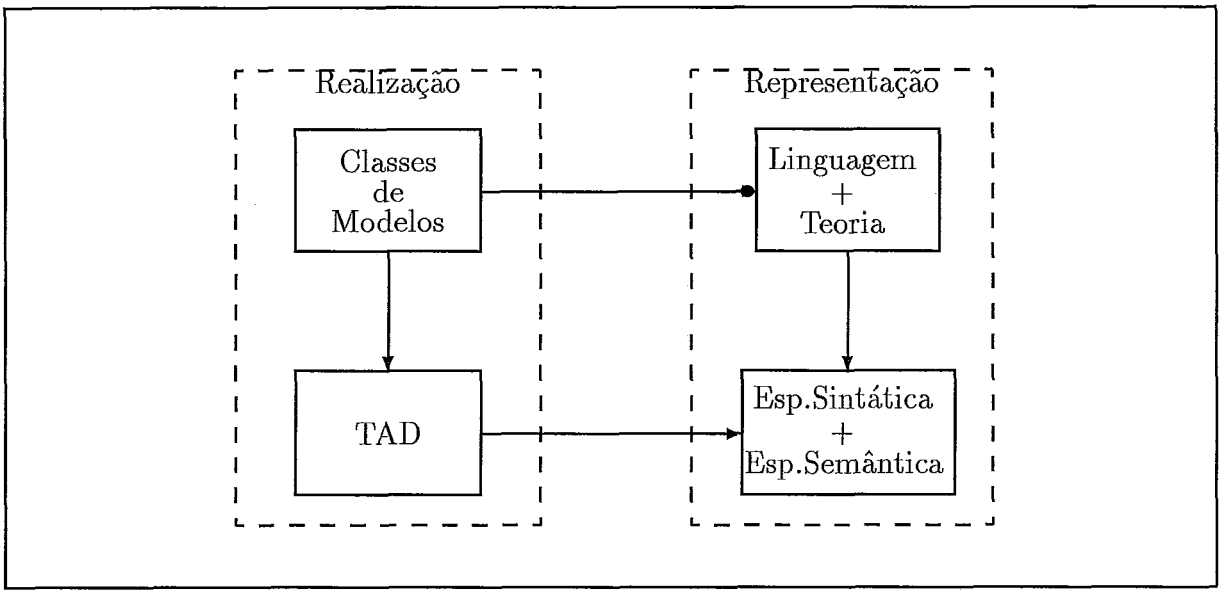


Figura II.2.1: Realização e Representação

Resta escolher um formalismo para representar adequadamente todas estas idéias. A opção naturalmente óbvia, dado o que já vem sendo esboçado aqui, seria utilizar a *Lógica,* neste caso de *Primeira Ordem,* e *Poli-sortida,* para enfatizar a distinção entre objetos abstratos diferentes. Neste contexto, objetos abstratos são associados a sortes, enquanto suas operações podem estar ligadas a predicados ou funções.

Para especificar a sintaxe de um TAD basta então, determinar o nome e as regras de formação para os símbolos extra-lógicos, pois a definição de termos e sentenças já acompanha a definição de fórmula bem formada.

A especificação semântica é constituida por um conjunto de axiomas (preferencialemente finito, quando possível), obedecendo a linguagem definida na especificação sintática, tais que a relação de conseqüência lógica usual (i.e. $\models$) permite deduzir qual-

quer outra fórmula satifatível pela classe de modelos associada ao TAD, e mais nenhuma outra.[5]

O exemplo a seguir ilustra a notação adotada no presente trabalho.

ADT *Indice*

Syntax

Function $maxpos : \longmapsto Indice;$

$pred : Indice \longmapsto Indice;$

$corresp : Indice \times Indice \times Indice \longmapsto Indice;$

Predicate $Indice < Indice;$

$inside(Indice);$

Axiomatization

$$(\forall i : Indice)\neg i < i \tag{1}$$

$$(\forall i, j, k : Indice)(i < j \ \wedge \ j < k) \Rightarrow i < k \tag{2}$$

$$(\forall i, j : Indice)(i < j \Rightarrow \neg j < i) \tag{3}$$

$$(\forall i, j : Indice)(i < j \ \vee \ j < i \ \vee \ i \stackrel{I}{=} j) \tag{4}$$

$$(\forall i : Indice)(\exists j : Indice)$$
$$i < j \ \wedge \ (\forall z : Indice)\{i < z \Rightarrow \neg z < j\} \tag{5}$$

$$(\forall i, j, k : Indice)inside(i, j, k) \Leftrightarrow \{\neg i < j \ \wedge \ \neg k < i\} \tag{6}$$

$$(\forall i : Indice)pred(i) < i \tag{7}$$

$$(\forall i, j, k : Indice)(\exists c : Indice)c \stackrel{I}{=} corresp(i, j, k) \Leftrightarrow$$
$$\{[pred(j) \stackrel{I}{=} i \ \wedge \ pred(k) \stackrel{I}{=} c] \ \vee \ [c \stackrel{I}{=} corresp(i, pred(j), pred(k))]\} \tag{8}$$

End

Este TAD especifica toda uma classe de instâncias finitas ou infinitas, onde é possível estabelecer uma relação de ordem ($<$) total e discreta. Portanto, *Indice* é uma classe de modelos possivelmente com cardinalidades distintas. As demais funções servirão adiante, para estruturar a especificação para novos TAD's partindo de *Indice*.

As declarações **Function** e **Predicate** formam a especificação sintática, enumerando os símbolos extra-lógicos, suas aridades, argumentos e resultados. Tudo enfim, o que é necessário para decidir se uma sentença pertence ou não a linguagem, ou seja, está corretamente expressa.

[5] veja A.1.12

A especificação semântica é formada por fórmulas de primeira ordem, na linguagem, e devidamente quantificadas segundo os sortes. Este conjunto de sentenças pode ser referido como a axiomatização do TAD, pois equivale exatamente a uma apresentação para a teoria dos modelos associados ao mesmo.

Com a especificação de *Indice*, outros tipos abstratos de dados mais elaborados podem ser hierarquicamente construidos.

ADT $Array[Indice, Elem]$

Syntax

Import $Indice, Elem$

Function $initialize : \longmapsto Array;$

$attrib : Array \times Indice \times Elem \longmapsto Array;$

$value : Array \times Indice \longmapsto Elem;$

Axiomatization

$$(\forall a : Array)\{\, a \stackrel{A}{=} initialize \vee$$

$$(\exists a' : Array)(\exists i : Indice)(\exists e : Elem) a \stackrel{A}{=} attrib(a', i, e)\} \quad (1)$$

$$(\forall a : Array)(\forall i : Indice)(\forall e : Elem)$$

$$value(atrib(a, i, e), i) \stackrel{E}{=} e \quad (2)$$

$$(\forall a : Array)(\forall i, i' : Indice)(\forall e : Elem)$$

$$\neg i \stackrel{I}{=} i' \Rightarrow value(atrib(a, i, e), i') \stackrel{E}{=} value(a, i') \quad (3)$$

End

Por questão de clareza e simplicidade, a especificação de um TAD fica subdividida por assim dizer em "unidades", onde cada unidade está relacionada exclusivamente a um único sorte (o sorte de interesse). Na verdade, cada unidade constiue por si um TAD, visto que há uma classe de modelos associada.

A declaração **Import** significa, então que outros sortes acompanhados de suas operações e axiomas estão implicitamente contidos no TAD em questão, embora, seus símbolos e sentenças não estejam explicitamente presentes. A declaração *Import* pode ser tecnicamente entendida como uma extensão conservativa (veja A.3.2). Assim, $Indice \leq Array[Indice, Elem]$.

Especificando um tipo abstrato de dados deste modo, além de economizar na notação, fica favorecida a disciplina, estruturando representações hierarquicamente. É possível e aconselhável ir desenvolvendo a especificação de cada TAD singularmente, e

ir compondo a especificação definitiva em unidades. Além de melhorar a legibilidade da especificação obtida, e ajudar a decompor a atividade de especificação, este proceder construtivo será útil futuramente neste texto.

Ainda uma declaração **HidenSymbols,** estabelece os símbolos auxiliares utilizados exclusivamente para melhorar a clareza e a legibilidade dos axiomas na especificação semântica.

Posto isto, as etapas 3 e 4 do desenvolvimento de programas com tipos abstratos de dados podem ser reformuladas, de maneira mais pragmática, como:

3. Identificar propriedades comuns aos objetos abstratos, e utilizá-las para construir uma especificação, talvez hieraquizada, para o TAD que fornece a semântica destes objetos;

4. Se ainda não está disponível, conceber uma especificação para a classe de realizações admissíveis para a linguagem escolhida para a codificação;

Uma vez evidente, por questão de simplicidade, não se fará explicitamente qualquer distinção entre estas noções, deste ponto em diante.

II.3 Implementação de Tipos Abstratos de Dados

Uma etapa do método não mereceu ainda suficientes esclarecimentos, pois parece ainda vago, como especificar a implementação declarativa Δ, relacionando as especificação para os TAD $\mathcal{T}$ e $\mathcal{T}'$. De algum modo, que agora deve ficar bem definido, as realizações para a especificação Δ, construida a partir de estruturas admissíveis na linguagem primitiva (definidas em $\mathcal{T}'$), precisam induzir modelos para $\mathcal{T}$, e consequentemente fornecer a semântica do programa abstrato.

Mesmo tomando o ponto de vista sintático, a especifcação Σ de $\mathcal{T}$, precisaria ser traduzida na linguagem primitiva, definida pela especifcação Σ' de $\mathcal{T}'$, mas contudo, preservando as propriedades originalmente expressas em Σ.

Visto que, a especifcação Σ' define um nível linguístico de expressividade simplificada, em relação a linguagem mais abstrata utilizada na especificação de Σ, é pouco provável que Σ' seja capaz de simular diretamente todas as propriedades inferíveis em Σ. Por conseqüência, não é uma mera tradução entre Σ e Σ' que soluciona o problema.

É preciso construir, empregando a linguagem de Σ', uma nova representação adequada para os objetos e operações abstratas referenciados em Σ, ou seja, enriquecer Σ' com novos símbolos e axiomas para refinar os sortes, funções e predicados em Σ. Porém este processo de refinamento embora fundamentado em Σ', deve manter sua descrição protegida contra interferências. O resultado deste enriquecimento deve ser justamente a especificação pocurada para a implementação declarativa.

O módulo de implementação pode então ser codificado de acordo com a especificação de Δ. Usualmente, o mecanismo linguístico descrito por Σ' talvez já seja suportado por alguma instalação, há de se codificar então, apenas as novas representações definidas sobre Σ'.

Agora que Δ está positivamente apta a simular o comportamento dos objetos e operações abstratas, falta associá-los devidamente aos seus refinamentos, isto significa traduzir a especificação Σ em termos de conseqüências apresentadas por Δ.

Deste modo, fica definida a relação entre Σ e Σ', que da origem a especificação Δ do módulo de impementação. Intuitivamente, acontece que Σ foi realmente implementada em Σ' resultando em Δ. Esta noção intuitiva de implementação permite uma formalização mais acurada, conforme a seguir.

II.3.1 Passo Canônico

A implementação entre duas especificações Σ e Σ', ou mais rigorosamente, entre as teorias que elas apresentam, envolve basicamente dois procedimentos : primeiro, enriquecer a teoria mais primitiva Σ', com representações para os símbolos na linguagem abstrata e segundo traduzir estes símbolos em termos de seus refinamentos construidos na etapa anterior.

Enriquecer uma teoria, no contexto da Lógica de Primeira Ordem, pressupõe sem dúvidas, a noção de extensão (A.3.1). Já que a teoria Σ' deve ser mantida a salvo de interferências durante o refinamento, a extensão deve ser conservativa (A.3.2). Por outro lado, a tradução preservando a capacidade de derivar conseqüências, relaciona-se com a noção de interpretação entre teorias (A.2.4). Logo, a implementação pode ser definida, através de noções lógicas como :

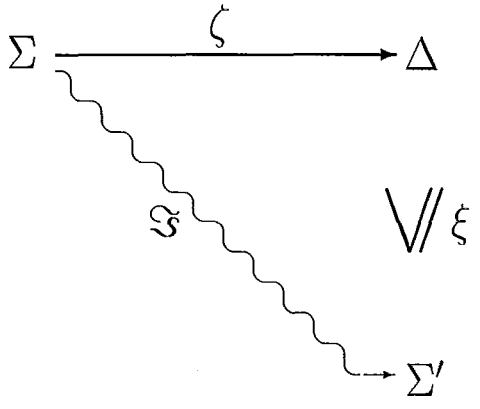
Definição II.3.1 (Implementação) *Seja Σ e Σ', duas teorias de primeira ordem consistentes. Uma implementação $\Im$, denotada por $\Sigma \rightsquigarrow \Sigma'$, é um par $\langle \zeta, \xi \rangle$ tal que :*

i) $\Sigma' \leq_{\xi} \Delta$

ii) $\Sigma \xrightarrow{\zeta} \Delta$, *onde* ζ *representa uma função de tradução entre linguagens;*

A teoria Δ *que media a implementação é chamada de implementação declarativa.*[6]

A implementação de Σ em termos de Σ' é vista diagramaticamente como,



Intuitivamente, este diagrama representa um passo no processo de implementação, ou um *passo canônico.*

Excetuando-se casos bastantes peculiares, de certa forma "artificiais", a extensão conservativa ξ é também expansiva. Isto significa que para os casos de "interesse" prático, qualquer realização satisfazendo a Σ' se expande a pelo menos uma realização para Δ. Pelo Teorema da Interpretação (A.2.2), os modelos para Δ induzem de volta modelos para Σ. De acordo com as expectativas iniciais, as realizações pertencentes a $\mathcal{T}'$ são levadas em realizações para $\mathcal{T}$, suportando assim os objetos e operações abstratas manipulados no programa abstrato.

A menos que seja estritamente necessário, será esquecida a distinção de significado entre a classe de modelos associada a um TAD, e sua especificação formal, ou seja, sempre que for mencionado o TAD ou sua especificação, implicitamente está envolvida uma referência a classe de modelos correlacionada.

Embora não seja trivial determinar a expansividade de uma extensão conservativa ξ, há algumas indicações úteis para garanti-la por construção (e.g. empregando

[6]Em [Vel87] esta denominação é reservada apenas para o conjunto de sentenças derivado em Δ na nova linguagem definida na extensão ($\mathcal{L}(\Delta - \Sigma')$).

extensões por definição de sorte, como as enunciadas em A.3.2).

Uma boa indicação no desenvolvimento de um passo canônico entre especificações Σ e Σ', seria, segundo [Vel87]:

i) Definir uma linguagem auxiliar $\mathcal{L}(\Sigma)^*$, que contenha símbolos correspondentes para todos os símbolos em $\mathcal{L}(\Sigma)$, além talvez de alguns símbolos auxiliares. Sem perda de generalidade, assume-se $\mathcal{L}(\Sigma)^*$ disjunta de $\mathcal{L}(\Sigma')$, isto é, sem símbolos em comum;

ii) Estender Σ' por definição de sorte, escolhendo para cada sorte em $\mathcal{L}(\Sigma)^*$ uma representação em termos dos sortes primitivos em $\mathcal{L}(\Sigma')$;

iii) Adicionar os símbolos de função e predicado em $\mathcal{L}(\Sigma)^*$ com axiomas para definir seus significados, porém, conservando os resultados obtidos em ii), de modo a obter uma especificação Δ, cuja a linguagem é $\mathcal{L}(\Sigma') \cup \mathcal{L}(\Sigma)^*$;

iv) Definir uma tradução natural ζ de $\mathcal{L}(\Sigma)$ em termos de $\mathcal{L}(\Delta)$ mas associando cada símbolo s em $\mathcal{L}(\Sigma)$ ao seu correspondente s^* em $\mathcal{L}(\Sigma)^*$. Se iii) foi corretamente conduzido o resultado deve ser uma interpretação de Σ em Δ, segundo a função de tradução ζ.

II.3.1.1 Preservação de Propriedades

Preservar propriedades sobre um conceito durante o processo de implementação denota, intuitivamente, que cada representação resultante apresenta, pelo menos, aquelas propriedades já descritas na especificação abstrata Σ e as conseqüências destas. É justamente esta capacidade de simular o comportamento de um conceito sendo descrito através de suas propriedades observáveis, que precisa ser mantido enquanto é construida a implementação.

Sintaticamente, significa manter a capacidade de derivar a tradução das conseqüências pertinentes a teoria, que antes especificava o conceito. Posto que propriedades são relativas à linguagem na qual estão descritas, deseja-se realmente preservá-las, mesmo havendo transição entre níveis linguísticos durante todo o processo de implementação.

A contra-partida semântica exige que a classe de realizações, satisfazendo a implementação declarativa resultante induza, em sentido inverso, uma classe de modelos que

correspondentemente satisfaz, pelo menos, todas as propriedades deriváveis na descrição abstrata.

Em síntese, uma vez obtida uma representação final na linguagem objetivo, existe e foi desenvolvida uma implementação correta para o TAD, conforme este foi abstraído e concebida inicialmente a sua especificação.

Em qualquer aplicação do passo canônico $\Im_2 : \Sigma_1 \rightsquigarrow \Sigma_1'$, propriedades são, por definição, preservadas. A mera interpretação, traduz em teoremas da implementação declarativa Δ_1, os axiomas de Σ_1. Uma interpretação $\zeta_1 : \Sigma_1 \longrightarrow \Delta_1$ é tal que: $\Sigma_1 \models \alpha$ então $\Delta_1 \models \alpha^{\zeta_1}$ para todo $\alpha \in Cn(\Sigma_1)$.

Por outro lado, a extensão conservativa $\Sigma_1' \leq \Delta_1$ tem o papel de enriquecer a linguagem concreta Σ_1', provendo recursos para representar adequadamente decisões de projeto, adição de detalhes e outros refinamentos que se fizerem necessários. Mas ainda assim, são conservadas as particularidades relativas à linguagem de Σ_1', isto é, $Cn(\Sigma_1') \subseteq Cn(\Delta_1)$

Ou seja, a implementação declarativa Δ_1 mantêm sempre a capacidade de derivar propriedades relativas à linguagem mais abstrata, descritas agora em termos da linguagem mais primitiva.

II.4 Verificação de Programas com Tipos Abstratos de Dados

Nenhum método de desenvolvimento está completo sem estabelecer como deve ser assegurada a corretude de seus resultados. A verificação permeia todas as etapas do desenvolvimento e é indissociável deste.

No caso específico do *Desenvolvimento com Tipos Abstratos de Dados,* o processo de verificação fica igualmente subdividido com a fatoração em programa abstrato e módulo de implementação. Assim, a verificação consiste em determinar a corretude de cada programa contra a sua especificação, tomando como base a especificação do TAD que o suporta, conforme [Vel87]. Portanto, o programa abstato é verificado contra a descrição da solução, tomando como base o TAD Σ que realiza os objetos e operações abstratas, e o módulo de implementação é comparado contra a implementação declarativa Δ, fundamentando-se no TAD Σ', que descreve a linguagem de codificação.

Na verificação de corretude do programa abstrato, os axiomas em Σ funcionam como lemas supostamente demonstrados. Deste modo, fica assegurado que qualquer modelo para a especificação Σ provê a semântica absolutamente correta para a computação executada pelo programa abstrato solucionando, assim, o problema proposto originalmente.

A corretude do programa como um todo só ficará garantida em definitivo, quando houver certeza que o módulo de implementação preserva estes lemas representados como axiomas de Σ. Uma vez que o módulo de implementação está correto em relação a implementação declarativa Δ, e com base em Σ', faltaria, na verdade, confirmar que o enriquecimento de Σ' em Δ foi apropriado e que Σ está interpretado em Δ.

Por conseguinte, não é suficiente apenas verificar cada parte em separado, é preciso constatar a válidade da relação entre o programa abstrato e o módulo de implementação. Isto implica em verificar a implementação de Σ em Σ', que resulta em Δ. Logo, é preciso mostrar que:

i) a extensão ξ de Σ' em Δ é realmente conservativa; e

ii) a tradução de Σ em Δ, segundo a função ζ é de fato uma intepretação.

Assim, finalmente, estará certificado todo o trabalho concluido durante o processo de desenvolvimento. Em síntese, o processo de verificação pode ser enumerado em três etapas:

1. verificar a corretude do programa abstrato contra a descrição inicial da solução;
2. verificar o passo canônico de Σ em Σ' dando origem a Δ; e finalmente,
3. verificar a corretude do módulo de implementação codificado contra a implementação declarativa Δ, tendo a especificação da linguagem de codificação Σ' como base.

II.5 Referências Bibliográficas

O interesse em formalizar o processo de desenvolvimento de programas motivou um bom número de trabalhos na literatura, sugerindo a relevância do assunto. Os mecanismos de abstração envolvidos na concepção de um programa estão examinados em [Mai86] e [Vel85a]. O *Desenvolvimento de Programas com Tipos Abstratos de Dados* aparece

em [PV83] e [MV81] e finalmente está descrito, com detalhes, em [Vel87]. Nas mesmas referências, o enfoque lógico (i.e. utilizando *Lógica de Primeira Ordem Poli-Sortida*) é adotado como alternativa para especificações algébricas, tradicionais na especificação de Tipos Abstratos de Dados.

As primeiras tentativas de formalizar a implementação de programas, por meio de noções lógicas (e.g. Interpretação e Extensão Conservativa), pretendendo caracterizar passos de refinamento sucessivos está em [MVS84], [MT84], [Vel85b] e [MVS85]. Em [MVS84] e [MT84] aparece a intuição que fundamentaria a definição de *Passo Canônico*, mas ainda antes de receber este nome. Esta noção de passo de implementação é melhor definida em [Vel87] e em [TM87]. Em [Vel88], esta mesma formalização é aplicada ao desenvolvimento de soluções para problemas. A verificação acompanhando cada passo de implementação, é também definida em [Vel87], conforme está resumida na seção II.4.

Capítulo III

Composição de Implementações

Uma vez posto formalmente o procedimento capaz de representar a noção intuitiva de implementação, cabe a este capítulo discutir como viabilizar sua aplicação efetiva no Desenvolvimento com Tipos Abstratos de Dados.

Estabelecer um passo canônico, entre TAD's, descritos em linguagens dissimilares por natureza, não é uma tarefa inconsequente cuja a complexidade seja negligenciável. Usualmente exige mais que uma abordagem *ad hoc* para produzir os resultados esperados.

Uma estratégia disciplinada para subdividir em passos menos elaborados a implementação e os respectivos meios para combinar corretamente decisões de refinamento e detalhes, dando origem a uma implementação declarativa satisfatória, permitiria construir o passo canônico inicialmente em questão, de uma forma mais eficiente e menos sujeita a erros.

Portanto, este capítulo pretende apresentar um mecanismo de composição de implementações já conhecido (veja [Vel87]), e complementando-o, propor e formalizar em termos lógicos um outro possível mecanismo de composição. Ambos, conjuntamente, atendem a construção de implementações quando o desenvolvimento quer ser formal.

Em síntese, a primeira seção anuncia duas formas de composição de implementações e os teoremas que asseguram a correspondente validade de cada uma, enquanto a segunda confirma que as propriedades especificadas são preservadas mesmo através da composição de passos canônicos.

III.1 Componibilidade de Implementações

A implementação de um tipo abstrato de dados em um outro, $\Im : \Sigma \rightsquigarrow \Sigma'$, seguindo o procedimento do passo canônico (ver figura III.1.1), pode ser, ainda, uma tarefa com uma razoável complexidade envolvida. A transição de um nível de detalhe para outro maior, nem sempre é tão trivial.

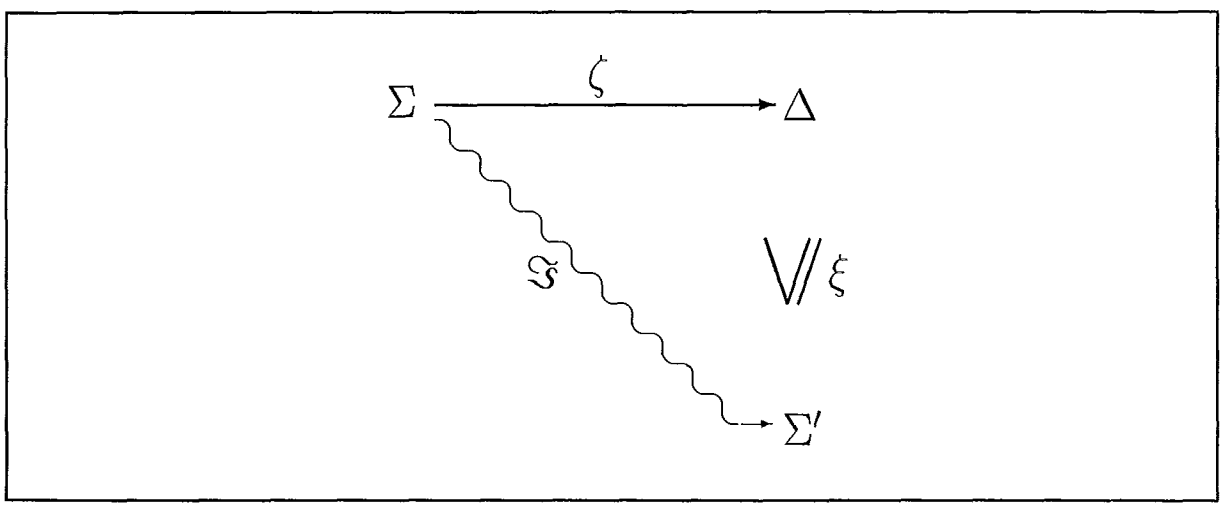


Figura III.1.1: O Passo Canônico

Muito da habilidade e da experiência pessoal do projetista podem ser requeridas para levar a efeito uma tradução correta. Além disto, boa parte de seu esforço costuma ser desperdiçado em partes não criativas da implementação, simplesmente por falta de uma disciplina que a sistematize e evite tentativas infrutíferas. Normalmente é possível que uma implementação precise ser reformulada ou refeita algumas vezes, até que um resultado satisfatório seja obtido.

Dependendo fundamentalmente da quantidade de sortes e operações em ambas as linguagens e de como estas operações relacionam-se entre si em cada especificação, a transição de conceitos mais abstratos, em Σ, para correspondentes mais concretos, em Σ', tende a exigir um esforço considerável, sendo portanto, propícia à ocorrência de erros. Isto desestimula o emprego de formalismos, como o passo canônico, nos casos de implementação mais complexos, onde justamente, uma abordagem mais formal mostra-se mais desejável.

A arte da implementação consiste então em escolher um par interpretação e extensão conservativa, $\langle \zeta, \xi \rangle$, que seja não só tecnicamente correto, mas também metodologicamente adequado ao projeto. Contudo, nisto está envolvido a satisfação de objetivos conflitantes. Por um lado, quanto mais forte for a especificação da implementação declar-

ativa, Δ, mais fácil será determinar uma interpretação, ζ; por outro, quanto mais liberal for sua especificação, mais facilmente consegue-se estabelecer a conservatividade da extensão, ξ. É importante ponderar estes objetivos, construindo Δ, nem mais nem menos do que se faz realmente necessário.

A sensível influência recíproca entre interpretações e extensões conservativas, evidenciada na aplicação do passo canônico, entretanto, tende a conduzir, ao nosso ver, no sentido de um caráter eminentemente construtivo, incorrendo para a decomposição da implementação em passos mais simples e concebendo especificações intemediárias para mediar a transição completa da descrição inicial para uma descrição objetivo.

Embora não seja a única estratégia aplicável ao procedimento de implementação, sua formalização admite liberdade para que as decisões de projeto sejam convenientemente tomadas e sequenciadas, de acordo com as circunstâncias locais em cada etapa do desenvolvimento, apesar de mantida a disciplina e a precisão formal.

A especificação de um TAD não é presumivelmente construída de uma só vez, simplesmente determinando-se um bloco monolítico de símbolos e sentenças. Ao contrário, usualmente, procede-se acrescentando os símbolos e os respectivos axiomas para defini-los, construtivamente, à proporção que as propriedades vão sendo abstraídas e compreendidas.[1] Pois, tornar-se-ia extremamente difícil na realidade apreender integralmente a totalidade de um conceito sendo modelado, captando todos os seus detalhes e decidindo quais são relevantes. Aceitando-se este fato, parece natural concordar que uma implementação, $\Im$, igualmente não deveria ser tentada em um só ensaio.

Haveria, deste modo, sentido prático para a capacidade de subdividir ou decompor o processo de implementação em passos mais simples e compreensíveis. Por conseguinte, seria bem mais fácil determinar a extensão conservativa e a interpretação, concentrando-se sobre cada um destes passos isoladamente, pois a atenção estaria focalizada sobre um domínio bem mais restrito com uma visão mais ordenada e hierarquizada a respeito dos objetivos do projeto, atingindo obviamente melhores resultados.

Pensando em decompor o processo a fim de simplificá-lo, uma primeira tentativa seria reduzir as dificuldades inerentes a uma transição direta entre níveis lingüísticos conceitualmente distantes, interpondo descrições intermediárias e gradativamente sequeciando passos de implementação entre elas, de forma a obter uma implementação correspondente, partindo da especificação inicial, Σ, até a especificação objetivo, Σ'.

[1] A notação para especificação adotada neste trabalho, visa promover estas idéias

Inegavelmente, construir um passo de implementação, agora entre TAD's descritos por especificações mais assemelhadas, é normalmente mais espontâneo, sendo em conseqüência, menos susceptível a incompreensões ou erros.

De uma forma mais sutil, talvez seja compensador voltar também a atenção para a complexidade envolvida na definição de apenas um passo de implementação particular, objetivando descomplicar a construção da extensão conservativa ξ, conseqüentemente a definição da interpretação ζ. Certificar a corretude da implementação declarativa Δ também poderia valer-se desta simplificação. A idéia aqui é ligeiramente diferente, consiste em implementar Σ em termos de Σ', utilizando, para tanto, passos de implementação já estabelecidos entre pares de especificações comparativamente menos elaboradas.

A existência de mecanismos de recombinação, aplicáveis recursivamente, possibilitaria, nestas circunstâncias, proceder a construção de um passo canônico relativamente complexo, por composição de passos arbitrariamente simples, de acordo com as conveniências e necessidades, em cada caso particular. E então, ater-se exclusivamente a um passo, a cada momento, deixando ao mecanismo, a responsabilidade de compor uma implementação completa equivalente.

Estas duas abordagens são pragmaticamente aplicáveis se e somente se for possível assegurar que os resultados que seriam obtidos através de um mecanismo de composição, são sempre coerentes, ou seja representam um passo real de implementação, consoante com o que era inicialmente desejado.

Ambas as abordagens guardam em comum a motivação de fazer face a complexidade associada, simplesmente viabilizando a decomposição em passos melhor manipuláveis. De maneira que os resultados parciais, produzidos dentro de domínios restritos, façam-se componíveis, preservando-se a semântica das especificações iniciais. Entretanto, na primeira situação, o processo de implementação é visto globalmente, enquanto no segundo caso, a decomposição dedica-se a cada passo canônico em isolado.

Portanto, precisaria haver à disposição meios comprovadamente corretos e eficazes para combinar passos de implementação intermediários obtendo implementações entre tipos abstratos de dados ou mais elaborados ou conceitualmente díspares. Somente assim, a utilização prática destas estratégias de decomposição do problema de implementação e posterior recomposição seria realista.

Em síntese, há vantagens complementares em explorar a capacidade de compor implementações nas duas possibilidades, isto é:

- na composição de implementações realizadas em passos subsequentes, de forma a obter uma tradução direta do nível abstrato inicial ao nível concreto objetivo;
- na construção de uma nova implementação por sobre uma outra subjacente já estabelecida, de forma a estender esta última, incorporando seus resultados.

As circunstâncias concretas, para as quais cada uma revela-se singularmente apropriada, serão objetivo das próximas seções.

III.1.1 Composição de Implementações Subsequentes

A primeira situação destaca-se especialmente por transições consecutivas entre níveis lingüísticos totalmente distintos. E o mecanismo de composição correspondente desempenha seu papel quando o passo canônico pode ser aplicado iteradas vezes, e a composição destes passos subsequentes é capaz de transpor a distância conceitual entre o nível lingüístico no qual se acha expresso a especificação original, e um nível destino determinado.

A implementação direta de Σ_1 em Σ_3, poderia ser equivalentemente substituída por dois passos canônicos consecutivos, por exemplo, $\Im_1$ e $\Im_2$, interpondo-se uma especificação Σ_2 intermediária. Em princípio, esta transição em etapas é bem mais natural, pois as representações definidas encontram mais facilmente seus refinamentos em um nível lingüístico semelhante. Observe esta situação na figura III.1.2.

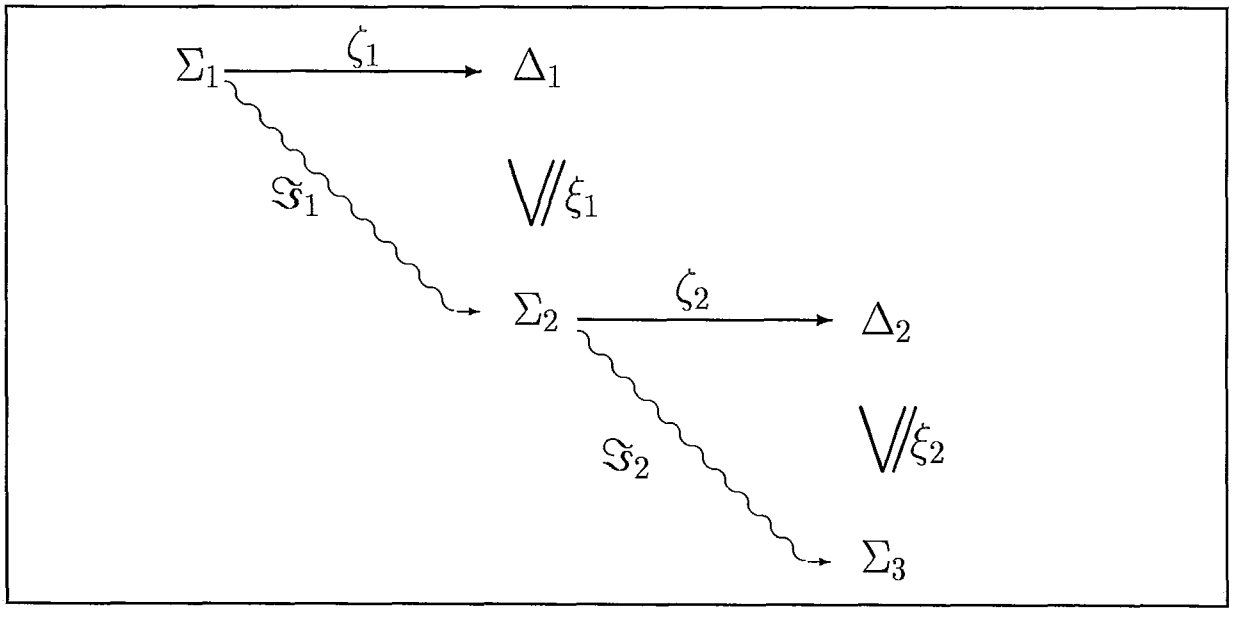
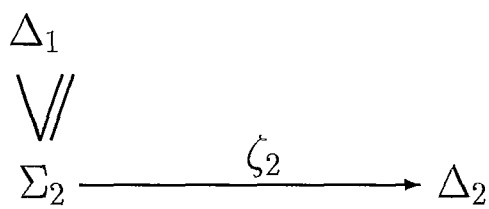


Figura III.1.2: $\Im_1$ e $\Im_2$: Passos Canônicos Subsequentes

O mesmo raciocínio poderia ser aplicado novamente, na construção de $\Im_1$ e $\Im_2$. E assim indutivamente, até a distância conceitual entre a descrição original e a representação objetivo ser vencida.

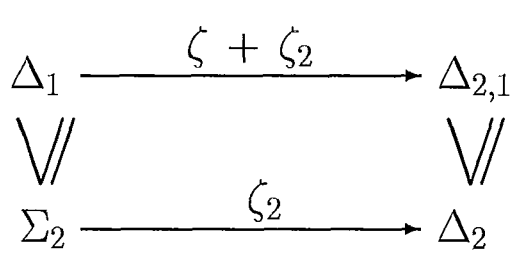
Quando a atividade de implementação está apropriadamente modularizada, é razoável esperar que o trabalho criativo, realizado durante a implementação $\Im_1 : \Sigma_1 \rightsquigarrow \Sigma_2$ não precise ser reconstituído; e a composição $\Im_1 . \Im_2$, de algum modo, leve a diante as propriedades especificadas em Σ_1, e os refinamentos introduzidos em Δ_1, ao nível lingüístico especificado por Σ_3, fornecendo então uma implementação direta satisfatória.

Observando isoladamente as especificações Δ_1, Σ_2 e Δ_2, retiradas da figura acima, forma-se o seguinte diagrama :



As especificações Δ_1 e Δ_2 representam todas as propriedades decididas em cada passo individual de implementação. Logo, qualquer tentativa de composição entre $\Im_1$ e $\Im_2$ deveria tomá-las como ponto de partida para definir uma especificação candidata a mediar a implementação direta de Σ_1 a Σ_3. Mas, de que maneira ?

Seria necessário construir uma especificação, por exemplo chamada $\Delta_{2,1}$, capaz de completar o diagrama anterior conservando as propriedades em Δ_2 e transpondo todas informações descritas em Δ_1, isto é, atendendo ao diagrama,



para só então, recolocando o diagrama na figura III.1.3, estabelecer um passo canônico resultante da composição de $\Im_1$ e $\Im_2$, através da nova implementação declarativa $\Delta_{2,1}$,

que satisfez ao diagrama anterior.

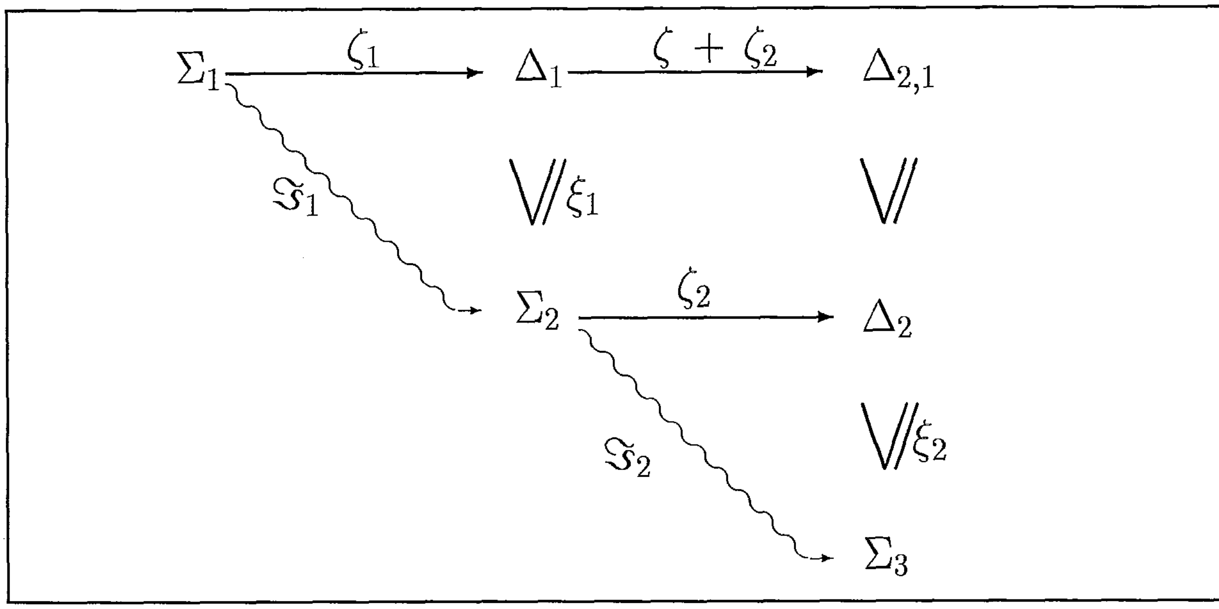


Figura III.1.3: Composição de $\Im_1$ com $\Im_2$

Este procedimento de composição seria válido somente quando matematicamente fundamentado, certificando que há sempre uma tal especificação $\Delta_{2,1}$, definida por uma extensão conservativa de Δ_2 e a interpretação ζ_2 extendida a $\zeta + \zeta_2$[2].

III.1.2 Teorema da Modularização

Seja o enunciado seguinte, denominado Teorema da Modularização. A partir de seus resultados, consegue-se automaticamente, estabelecer uma extensão conservativa e uma interpretação, determinando uma especificação para intermediar a implementação direta $\Sigma_1 \rightsquigarrow \Sigma_3$ de acordo com procedimento descrito brevemente acima.

Teorema III.1.1 (Teorema da Modularização) *Considere teorias consistentes* Δ_1, Σ_2 *e* Δ_2 *tais que*

$$\begin{array}{lll} \Delta_1 & & \\ \vee\!\!/ & & \\ \Sigma_2 & \xrightarrow{\zeta_2} & \Delta_2 \end{array}$$

[2]a interpretação $\zeta + \zeta_2$ é definida por uma função de tradução que concorda com ζ_2 em todas as fórmulas na linguagem de Σ_2, e estende para o restante de seu domìnio de acordo com a tradução ζ

Então, pode-se construir pelo menos uma teoria $\Delta_{2,1}$, *tal que*

$$\begin{array}{ccc} \Delta_1 & \xrightarrow{\zeta+\zeta_2} & \Delta_{2,1} \\ \Vdash & & \Vdash \\ \Sigma_2 & \xrightarrow{\zeta_2} & \Delta_2 \end{array}$$

Prova : A demonstração para este teorema pode ser encontrada em [Vel92c], ou outra versão mais simples em [Vel93]. Todavia, aqui são delineados alguns aspectos capazes de esclarecer as idéias envolvidas na demonstração original.

Inicialmente, considere o diagrama

$$\begin{array}{ccc} \Delta_1 & & \\ \Vdash & & \\ \Sigma_2 & \xrightarrow{\zeta_2} & \zeta_2[\Sigma_2] \end{array}$$

Então, tomando uma função de tradução $\zeta + \zeta_2$, sobre a linguagem de Δ_1, que estende naturalmente ζ_2 (de forma que os símbolos em $\mathcal{L}(\Delta_1)$ mas que não pertencem a $\mathcal{L}(\Sigma_2)$ são mapeado por $\zeta + \zeta_2$ fora de $\zeta_2[\Delta_2]$), é possível estabelecer :

$$\begin{array}{ccc} \Delta_1 & \xrightarrow{\zeta+\zeta_2} & (\zeta + \zeta_2)[\Delta_1] \\ \Vdash & & \Vdash \\ \Sigma_2 & \xrightarrow{\zeta_2} & \zeta_2[\Sigma_2] \end{array}$$

mas, por transitividade (A.2.7),

$$\Sigma_2 \xrightarrow{\zeta_2} \Delta_2 \text{ se e somente se } \Sigma_2 \xrightarrow{\zeta_2} \zeta_2[\Sigma_2] \xrightarrow{id} \Delta_2$$

contudo, $\zeta_2[\Sigma_2] \xrightarrow{id} \Delta_2$ significa que $\zeta_2[\Sigma_2] \subseteq \Delta_2$. Logo, o diagrama inicial pode ser reestabelecido como:

$$\begin{array}{ccccc} \Delta_1 & \xrightarrow{\zeta+\zeta_2} & \zeta + \zeta_2[\Delta_1] & & \\ \Vdash & & \Vdash & & \\ \Sigma_2 & \xrightarrow{\zeta_2} & \zeta_2[\Sigma_2] & \subseteq & \Delta_2 \end{array}$$

Por construção : $\mathcal{L}(\Delta_2) \cap \mathcal{L}(\zeta + \zeta_2[\Delta_1]) = \mathcal{L}(\zeta_2[\Sigma_2])$, portanto, através do Teorema da Consistência de Robinson Estendido (A.3.1) existe pelo menos uma teoria $\Delta_{2,1}$, tal que

$$\begin{array}{ccccc} \Delta_1 & \xrightarrow{\zeta+\zeta_2} & \zeta+\zeta_2[\Delta_1] & \subseteq & \Delta_{2,1} \\ \Vdash & & \Vdash & & \Vdash \\ \Sigma_2 & \xrightarrow{\zeta_2} & \zeta_2[\Sigma_2] & \subseteq & \Delta_2 \end{array}$$

Novamente, pelo mesmo argumento da transitividade, visto que $\Delta_{2,1}$ é consistente por construção, então $\Delta_1 \xrightarrow{\zeta+\zeta_2} \Delta_{2,1}$. Também, pelo diagrama, $\Delta_2 \leq \Delta_{2,1}$, conforme é preciso para satisfazer totalmente ao enunciado. □

O Teorema da Modularização e suas aplicações têm merecido apuradas investigações na literatura, tratamento mais detalhado pode ser encontrado em [TM87].

Este texto procura, por sua vez, concentrar-se mais particularmente na segunda forma de composição, tentando explorar mais detidamente a sua aplicabilidade ao desenvolvimento de programas.

III.1.3 Composição com Implementações Subjacentes

A necessidade de compor uma nova implementação por sobre outra pré-existente advém, em circunstâncias reais, sobretudo quando passos de implementação entre especificações relativamente menos complexas podem servir como base para direcionar a aplicação do passo canônico envolvendo outras bem mais elaboradas. A iteração destas composições faz sempre sentido em uma construção incremental de implementações. Além do que, frequentemente verifica-se a necessidade de reconsiderar trabalho já realizado, em face a novos requisitos ou a uma melhor compreensão a respeito de algum conceito implementado.

Em uma primeira análise, esta composição revela-se naturalmente adequada em situações caracterizadas pelo enriquecimento de níveis lingüísticos, entre os quais já se encontraria determinada uma relação de implementação. Distintamente do caso mencionado na seção anterior, a nova implementação está sempre circunscrita a estes mesmos níveis, ou a suas extensões. Ou seja, os resultados pretendidos referem-se a linguagens mais expressivas, mas ainda contendo as conseqüências lógicas presentes em um passo de implementação anterior.

O objetivo de cada enriquecimento varia, de acordo com as particularidades da situação na qual se deseja realizar um novo passo de implementação. De um modo bem geral, uma linguagem é enriquecida simplesmente adicionando novos símbolos, juntamente com alguns axiomas, aumentando sua expressividade. O que invariavelmente remete ao

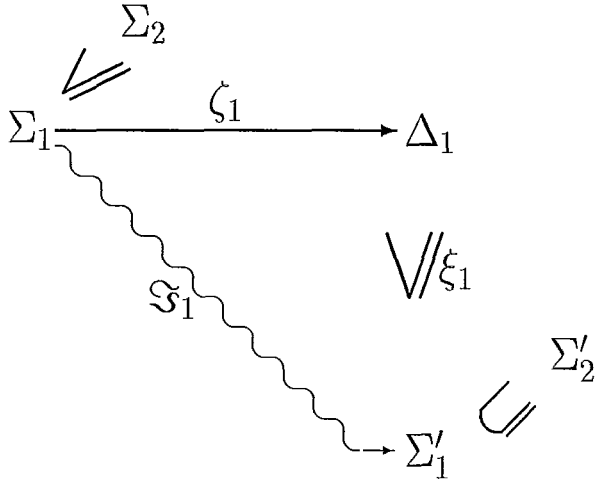
mecanismo lógico de extensão, para superpor um próximo passo.

Há ainda uma outra situação mais sutil, onde cabe também aplicar esta forma de composição : quando são mantidas as linguagens e contudo pretende-se acrescentar à especificação mais concreta axiomas definindo novas propriedades relativas aos símbolos já em seu repertório, restringindo assim possíveis realizações. Novamente, lida-se com extensões, porém agora não conservativas.

Esta última situação apresenta-se em circunstâncias inegavelmente distintas, porque se há de compor uma nova implementação, mas incorporando algo que ainda não pertencia ao domínio de uma outra subjacente. Em geral, isto tende a ser bem mais complexo que simplesmente compor passos de implementação, sem sequer acrescentar novas propriedades na linguagem da especificação subjacente.

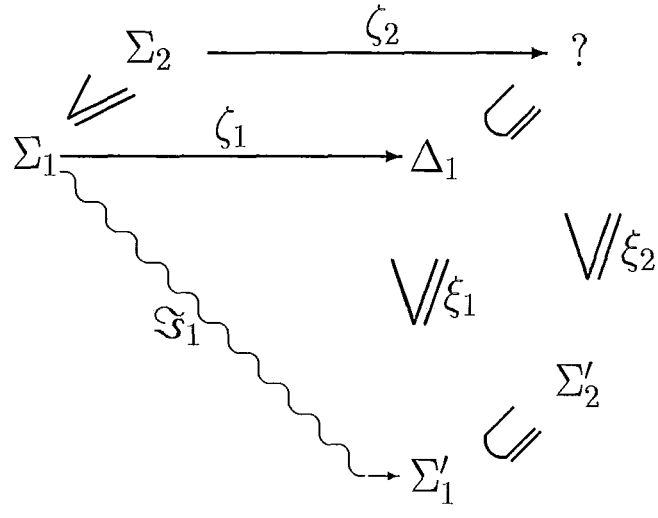
Situações intrinsecamente não conservativas, não se encaixam às condições para aplicar o Teorema da Modularização, porque não há meios para garantir a consistência dos resultados, e fica, pois, afastada sua aplicação, segundo [TM87].

Contudo, todas as situações, felizmente, compartilham características comuns, permitindo formalizá-las em um único caso genérico, considerando extensões tanto da especificação abstrata e quanto da concreta, esta última não conservativa, consoante o diagrama que procura sintetizá-las.



Similarmente, esperar-se-ia recursos para aproveitar integralmente o trabalho representado na construção da implementação $\Im_1 : \Sigma_1 \rightsquigarrow \Sigma'_1$, ou ao menos sua parte criativa, e sobre este definir a nova implementação $\Im_2 : \Sigma_2 \rightsquigarrow \Sigma'_2$, incorporando estes resultados consolidados em $\Im_1$.

O passo canônico a ser construído deveria então reter a extensão conservativa ξ_1, e a interpretação ζ_1, participantes do passo $\Im_1$, subjacente, como forma de adquirir o conjunto das conseqüências deriváveis em Δ_1. Para tanto, desejar-se-ia estender o par $\langle \zeta_1, \xi_1 \rangle$, a fim de obter uma interpretação ζ_2 e uma extensão conservativa ξ_2, implementando Σ_2 em Σ'_2, segundo o diagrama,



A implementação $\Im_2$ deveria resultar em uma especificação que possa corretamente substituir ?, completando o diagrama. Preferivelmente construindo uma implementação declarativa Δ_2, que intermediasse a transição de Σ_2 em Σ'_2. Os resultdos desta composição estão expressos na figura que se segue.

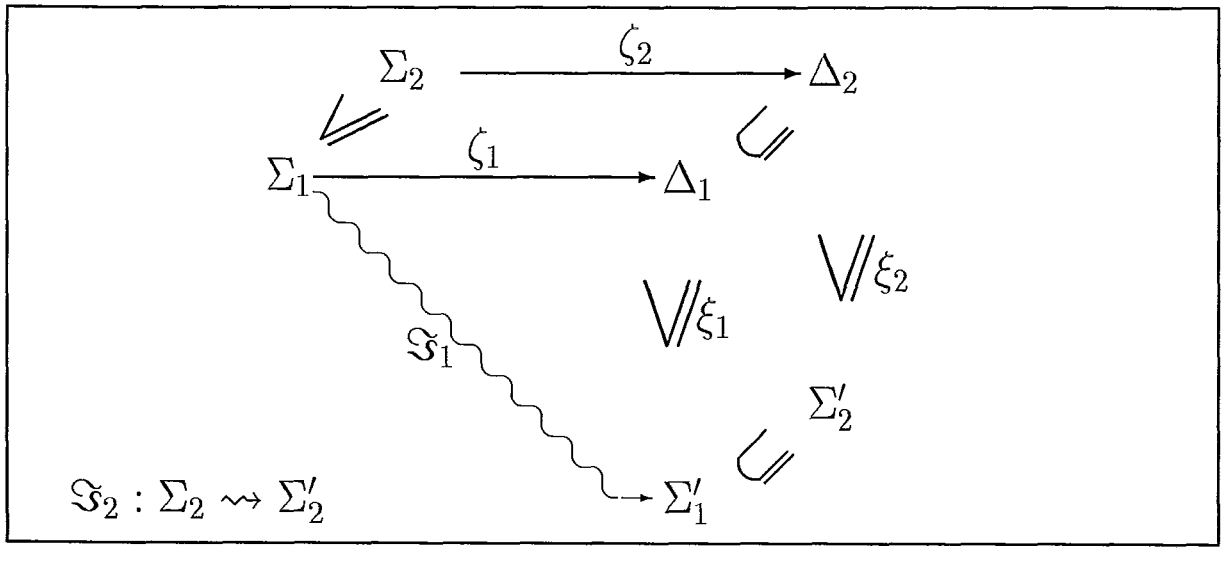


Figura III.1.4: Composição de $\Im_2$ por sobre $\Im_1$

Pensando em construir a implementação $\Im_2$, fundamentada em $\Im_1$, a especi-

ficação Δ_1 (que media $\Im_1$), funciona "mais ou menos" como se fosse um "protótipo", tendo o papel de estabelecer as propriedades centrais à implementação declarativa Δ_2.

Carece uma formalização precisa para apoiar a composição de novas implementações superpondo outras anteriores, e determinar os limites de sua corretude, desempenhando um papel análogo ao Teorema da Modularização em relação a composição de passos canônicos subsequentes. Somente assim, justifica-se a sua aplicação ao processo de desenvolvimento de programas.

III.1.4 Teorema da Construtibilidade

A fim de fornecer suporte à demonstração de algumas questões relaciondas com o desenvolvimento de programas, mais exatamente, no tocante à composição de implementações, de acordo com os requisitos delineados na seção anterior (seção III.1.3), apresenta-se aqui o resultado capaz de justificar a plena utilização prática de estratégias de implementação com caráter eminentemente construtivo.

Tal resultado está expresso na forma de um teorema, denominado "Teorema da Construtibilidade" (a motivação para o nome parece bem óbvia), e enunciado a seguir.

Teorema III.1.2 (Teorema da Construtibilidade) *Considere existir teorias* Σ_1, Σ_2, Δ_1 *e* Σ'_1, Σ'_2, *esta última consistente. Se*

$$\begin{array}{ccccc} \Sigma_2 & & & & \\ \mathbb{V} & & & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 & & \\ & & \mathbb{V} & & \\ & & \Sigma'_1 & \subseteq & \Sigma'_2 \end{array}$$

Então, pode-se construir pelo menos uma teoria Δ_2 *consistente, tal que* $\Delta_1 \subseteq \Delta_2$ *e*

$$\begin{array}{ccccc} \Sigma_2 & \xrightarrow{\zeta_2} & \Delta_2 & & \\ \mathbb{V} & & & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 & & \mathbb{V} \\ & & \mathbb{V} & & \\ & & \Sigma'_1 & \subseteq & \Sigma'_2 \end{array}$$

Prova : Antes, a consistência da teoria Σ'_2 é suficiente para estabelecer a consistência

de Σ_1'. E portanto, serão igualmente consistentes Δ_1, pois estende Σ_1' conservativamente, e Σ_1, já que esta está interpretada em uma teoria consistente Δ_1.

Inicialmente, se

$$\begin{array}{ccc} \Delta_1 & & \\ \mathbb{V} & & \\ \Sigma_1' & \subseteq & \Sigma_2' \end{array}$$

então, fazendo a linguagem de Σ_2' disjunta da linguagem de Δ_1, a menos dos símbolos presentes em uma interseção comum em Σ_1',

$$\mathcal{L}(\Delta_1) \cap \mathcal{L}(\Sigma_2') = \mathcal{L}(\Sigma_1')$$

e já que tanto Σ_2' quanto Δ_1 são, por hipótese, consistentes, o Teorema da Consistência de Robinson Estendido, (Teorema A.3.1) garante a existência de uma teoria δ e a sua consistência, de forma que,

$$\begin{array}{ccc} \Delta_1 & \subseteq & \delta \\ \mathbb{V} & & \mathbb{V} \\ \Sigma_1' & \subseteq & \Sigma_2' \end{array}$$

A teoria δ é construída, como sendo exatamente a união das teorias Σ_2' e Δ_1.

Assim, seja δ uma extensão consistente de Δ_1, e como Σ_1 está interpretado em Δ_1, também é possível estabelecer uma interpretação de Σ_1 em δ, utilizando a mesma função de tradução ζ_1 (i.e. $\Sigma_1 \xrightarrow{\zeta_1.id} \delta$). Assim,

$$\begin{array}{ccccc} \Sigma_2 & & & & \\ \mathbb{V} & & & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 & \xrightarrow{id} & \delta \\ & & \mathbb{V} & & \mathbb{V} \\ & & \Sigma_1' & \subseteq & \Sigma_2' \end{array}$$

Retirando-se do diagrama acima,

$$\begin{array}{ccc} \Sigma_2 & & \\ \mathbb{V} & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \delta \end{array}$$

e então, o Teorema da Modularização, (Teorema III.1.1), fornece uma teoria consistente Δ_2, e uma interpretação $\Sigma_2 \xrightarrow{\zeta_2} \Delta_2$,

tal que,

$$\begin{array}{ccc} \Sigma_2 & \xrightarrow{\zeta_2} & \Delta_2 \\ \vee & & \vee \\ \Sigma_1 & \xrightarrow{\zeta_1} & \delta \end{array}$$

A teoria Δ_2, é construída por Δ_1 e Σ_2', e por uma tradução onde os axiomas em Σ_2 são teoremas (Maiores detalhes sobre a construção de Δ_2 podem ser entendidos examinando-se a prova para o Teorema da Modularização, III.1.1).

Finalmente, o diagrama original será completado, conforme,

$$\begin{array}{ccccc} \Sigma_2 & & \xrightarrow{\zeta_2} & & \Delta_2 \\ \vee & & & & \vee \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 & \subseteq & \delta \\ & & \vee & & \vee \\ & & \Sigma_1' & \subseteq & \Sigma_2' \end{array}$$

Tomando Δ_2 assim construída, estão satisfeitas as condições no enunciado:

i) $(\delta \leq \Delta_2)$ e δ é consistente, então Δ_2 é consistente.

ii) $(\Sigma_2 \xrightarrow{\zeta_2} \Delta_2)$ pelo Teorema da Modularização.

iii) $(\Sigma_2' \leq \delta) \wedge (\delta \leq \Delta_2) \Rightarrow (\Sigma_2' \leq \Delta_2)$ por transitividade de extensões.

iv) $(\Delta_1 \subseteq \delta) \wedge (\delta \leq \Delta_2) \Rightarrow (\Delta_1 \subseteq \Delta_2)$ por transitividade de extensões.

□

A demonstração do Teorema da Construtibilidade resume-se esquematicamente na figura III.1.5, permitindo visualizar a construção de uma teoria consistente Δ_2 a partir de Σ_2, Σ_2' e Δ_1, contanto que estejam satisfeitas as restrições.

III.1.4.1 Aplicação do Teorema da Construtibilidade

Agora que está definido e demonstrado o Teorema da Construtibilidade como então aplicá-lo a composição com implementações subjacentes? Como conclusão imediata do teorema

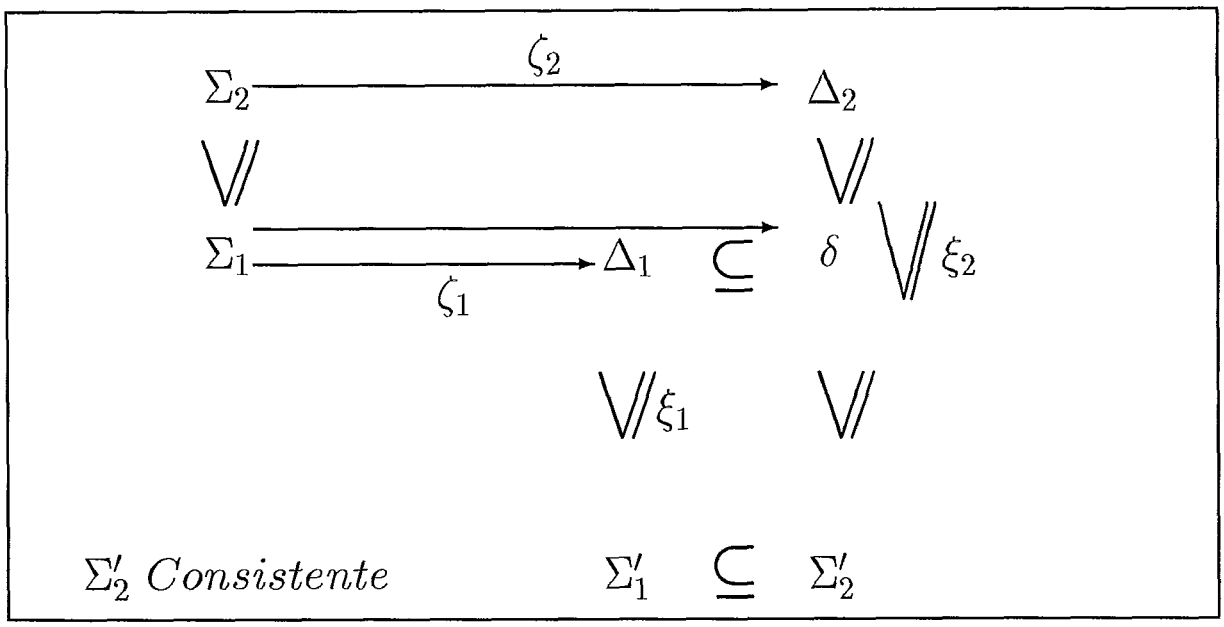


Figura III.1.5: Esquema da Demonstração do Teorema da Construtibilidade

pode-se afirmar que:

> "Dado um passo de implementação $\Im_1 : \Sigma_1 \rightsquigarrow \Sigma_1'$, é sempre possível compor um novo passo $\Im_2 : \Sigma_2 \rightsquigarrow \Sigma_2'$, implementando Σ_2 em Σ_2', desde que Σ_2 estenda conservativamente Σ_1, e Σ_2' seja uma extensão consistente de Σ_1'."

Consequentemente, o Teorema da Construtibilidade pode ser reformulado através de um corolário, diretamente aplicável à composição de implementações, pois estão subentendidas as fases intermediárias empregadas durante a demonstração.

Corolário III.1.1 *Considere que existam especificações* Σ_1,Σ_2 *e* Σ_1',Σ_2' *tais que:*

i) $\Sigma_1 \leq \Sigma_2$

ii) $\Sigma_1' \subseteq \Sigma_2'$, *onde* Σ_2' *é consistente*

Então, $\Sigma_1 \rightsquigarrow \Sigma_1'$ *implica em* $\Sigma_2 \rightsquigarrow \Sigma_2'$

Prova : Conseqüência direta do Teorema da Construtibilidade (teorema III.1.2), simplesmente tomando Δ_2, construída de acordo com o teorema, como a implementação declarativa que intermedia a nova implementação de Σ_2 em Σ_2'.

□

A utilização do corolário pode ser sinteticamente representada em III.1.6.

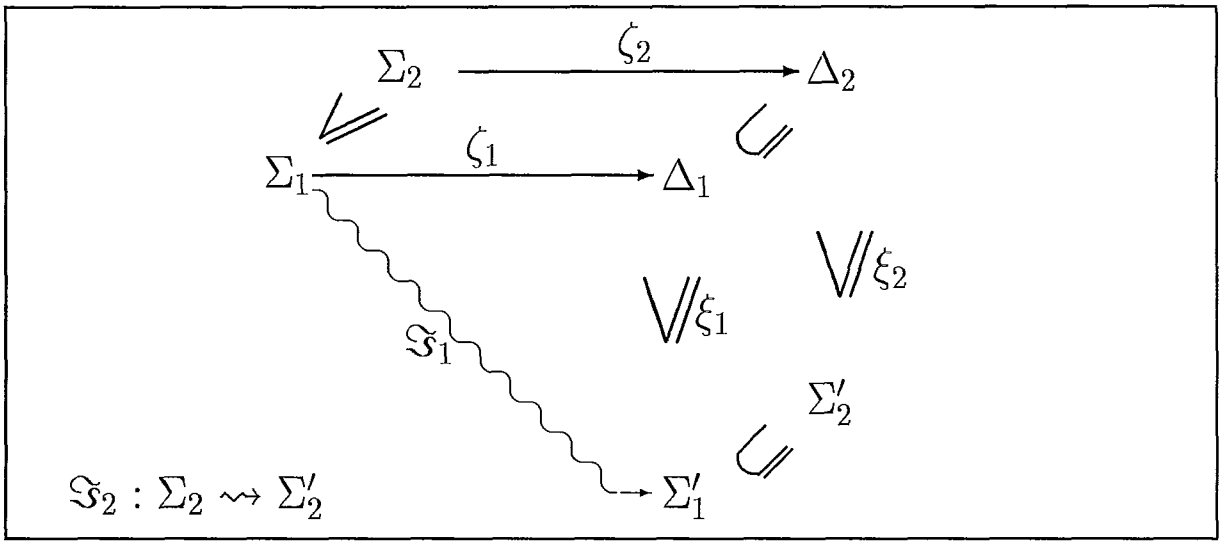


Figura III.1.6: Aplicação do Teorema da Construtibilidade

III.1.4.2 Comentários Adicionais

As condições impostas à aplicabilidade do teorema não são severamente restritivas. Mas ao contrário, pragmaticamente, o minímo que se esperaria da extensão, que amplia a teoria sobre a qual será realizada uma implementação, é que, não obstante os elementos adicionais, a especificação Σ_2 permaneça consistente. Ou seja, não há interesse objetivo caso Σ_2 seja trivializada, e não existam realizações que a satisfaçam. Portanto, a consistência da teoria Σ'_2 é, mais que uma contingência teórica, um requisito prático.

A própria demonstração do Teorema da Construtibilidade, estabelece um procedimento capaz de expandir um certo par extensão e interpretação, $\langle \zeta_1, \xi_1 \rangle$, para obter uma aplicação do passo canônico, $\langle \zeta_2, \xi_2 \rangle$, que represente a composição superposta. E conseqüentemente, também como construir uma teoria, Δ_2, indicando quais informações necessariamente estarão presentes ao seu conjunto de conseqüências.

Assim, considerando individualmente as etapas da demonstração, têm-se base para inferir a pertinência de alguns teoremas às conseqüências de Δ_2. A saber:

- Em concordância com o Teorema da Consistência de Robinson Estendido, que δ estende a implementação previamente realizada, ou seja, a teoria descrita em Δ_1 está contida em Δ_2, incorporando pois os resultados previamente obtidos pela implementação declarativa subjacente.
- Também pelo mesmo resultado, afirma-se que δ inclui, conservativamente, o mecanismo lingüístico especificado em Σ'_2 para suportar a implementação, o que con-

sequentemente assim o faz Δ_2.

- Segundo o Teorema da Modularização, que as propriedades a serem implementadas, expressas em axiomas de Σ_2, estão corretamente traduzidos em conseqüências de Δ_2. Ou seja, Δ_2 realmente media a implementação de Σ_2 em Σ_2'.

Em suma, a teoria Δ_2 deriva certamente a tradução ζ_2 das propriedades relativas a Σ_2, o mecanismo lingüístico em Σ_2' e a implementação declarativa Δ_1. Ou seja,

$$\Delta_2 \equiv (\Delta_1 \cup \Sigma_2' \cup \zeta_2[\Sigma_2])$$

A construção da implementação declarativa Δ_2, como descrita na demonstração, objetiva prover, pelo menos, informações necessárias à implementação $\Im_2 : \Sigma_2 \rightsquigarrow \Sigma_2'$, e por simplicidade, somente estas. Este critério de suficiência é porém, plenamente satisfatório consoante com os resultados mínimos que se esperaria obter em uma implementação de Σ_2 em Σ_2', qualquer que fosse a maneira utilizada para definir $\Im_2$.

Além do que, Δ_2 é na verdade a menor teoria, mediando o novo passo de implementação $\Im_2$, mas que ainda contém, no seu conjunto de conseqüências, as propriedades descritas pela implementação declarativa subjacente, Δ_1. Isto significa que qualquer teoria capaz de intermediar o passo canônico sendo superposto, e cujos os teoremas podem ser interpretados em Δ_2 é sempre igual a Δ_2, a menos de diferenças relativas à linguagem.

Proposição III.1.1 (Minimalidade) *Seja a composição de implementações :*

$$\begin{array}{ccccc} \Sigma_2 & & \xrightarrow{\zeta_2} & & \Delta_2 \\ \mathbin{/\!\!/} & & & & \\ \Sigma_1 & \longrightarrow & \Delta_1 & & \mathbin{/\!\!/} \\ & & \mathbin{/\!\!/} & & \\ & & \Sigma_1' & \subseteq & \Sigma_2' \end{array}$$

com Δ_2 construída de acordo com o Teorema da Construtibilidade. Seja uma teoria consistente $\mathcal{D}$, tal que:

i) $\Sigma_2 \xrightarrow{\kappa} \mathcal{D}$

ii) $\Sigma_2' \subseteq \mathcal{D}$

iii) $\Delta_1 \subseteq \mathcal{D}$

Então existe uma função de tradução tal que $\Delta_2 \xrightarrow{\mu} \mathcal{D}$.

Prova : Antes de tudo, pelo Teorema da Construtibilidade,

$$(\Delta_1 \cup \Sigma_2' \cup \zeta_2[\Sigma_2]) \equiv \Delta_2$$

Assim, é definida a seguinte função de tradução μ, sobre a linguagem de Δ_2 :

$$\mu(\phi) = \begin{cases} \phi & \text{se e somente se } \phi \in \mathcal{L}(\Delta_1) \\ \phi & \text{se e somente se } \phi \in \mathcal{L}(\Sigma_2') \\ \alpha^{\kappa} & \text{se e somente se } \phi \in \mathcal{L}(\zeta[\Sigma_2]) \\ & \quad \text{para algum } \alpha \in \mathcal{L}(\Sigma_2) \\ & \quad \text{talque } \phi = \alpha^{\zeta_2} \end{cases}$$

mas, $\Delta_2 \models \phi$ se e somente se $(\Delta_1 \cup \Sigma_2' \cup \zeta_2[\Sigma_2]) \models \phi$, logo,

$$(\Delta_1 \cup \Sigma_2' \cup \kappa[\Sigma_2]) \models \phi^{\mu}$$

por construção de μ. Diretamente da definição,

i) Se $\Sigma_2 \xrightarrow{\kappa} \mathcal{D}$ então $\kappa[\Sigma_2] \subseteq \mathcal{D}$

ii) Por hipótese, $\Sigma_2' \subseteq \mathcal{D}$ e $\Delta_1 \subseteq \mathcal{D}$

de modo que, $(\Delta_1 \cup \Sigma_2' \cup \kappa[\Sigma_2]) \subseteq \mathcal{D}$, e assim, $\mathcal{D} \models \phi^{\mu}$, consequentemente,

$$\Delta_2 \xrightarrow{\mu} \mathcal{D}$$

□

Em consonância com a proposição acima (III.1.1), fica certificado que qualquer teoria que fosse capaz de mediar o passo canônico superposto, haveria de ser, no mínimo, equivalente (a menos de isomorfismos) a Δ_2. A classe de modelos que a satisfaz seria, no máximo, isomórfica as possíveis realizações para Δ_2 (i.e. Δ_2 especifica as condições necessárias para uma implementação declarativa superposta).

Além de estabelecer, construtivamente, uma teoria Δ_2, satisfazendo as condições do enunciado, o teorema define ainda uma possível interpretação $\Sigma_2 \xrightarrow{\zeta_2} \Delta_2$. De

acordo com a demonstração, é sempre possível tomar como ζ_2, a extensão da tradução de Σ_1 em δ para a linguagem de Σ_2, fornecida pelo Teorema da Modularização.

Porque a interseção entre as linguagens de Δ_1 e Σ_2' precisa ser exatamente a linguagem de Σ_1', para possibilitar a aplicação do Teorema da Consistência de Robison Extentendido, traz implicações importantes para o processo de implementação, isto é, a extensão Σ_1' a Σ_2', limita-se somente a informações diretamente pertinentes ao nível lingüístico mais concreto. Nada deve ser expresso a respeito, explicitamente, de Δ_1, que não seja derivável em Σ_1'. O enriquecimento não conservativo de Δ_1, portanto, faz-se de forma implícita, somente através da composição originando Δ_2. O que parece ser razoável e presumivelmente aceitável.

A proposição III.1.2 confirma o relacionamento entre Δ_2, e qualquer outra teoria que por ventura venha a ser preferida para mediar a implementação superposta.

Proposição III.1.2 (Condições Suficientes) *Seja a composição de implementações:*

$$\begin{array}{ccccc} \Sigma_2 & & \xrightarrow{\zeta_2} & & \Delta_2 \\ \mathbin{\rotatebox{90}{$\vdash$}} & & & & \\ \Sigma_1 & \longrightarrow & \Delta_1 & & \mathbin{\rotatebox{90}{$\vdash$}} \\ & & \mathbin{\rotatebox{90}{$\vdash$}} & & \\ & & \Sigma_1' & \subseteq & \Sigma_2' \end{array}$$

com Δ_2 construída de acordo com o Teorema da Construtibilidade. Seja uma teoria consistente $\mathcal{D}$, tal que:

i) $\Delta_2 \xrightarrow{\kappa} \mathcal{D}$

ii) $\Sigma_2' \leq \mathcal{D}$

então, $\mathcal{D}$ pode ser uma teoria mediando a implementação $\Sigma_2 \rightsquigarrow \Sigma_2'$

Prova : Trivialmente,

i) $\Sigma_2 \xrightarrow{\zeta_2} \Delta_2$ e $\Delta_2 \xrightarrow{\kappa} \mathcal{D}$ portanto $\Sigma_2 \xrightarrow{\zeta_2.\kappa} \mathcal{D}$, por transitividade (A.2.7)

ii) $\Sigma_2' \leq \mathcal{D}$, por hipótese

tomando $\zeta_2.\kappa$ como a interpretação, e extensão conservativa assumida como hipótese, temos um par definindo uma aplicação do passo canônico. □

Não obstante, a simplicidade da proposição, esta serve para denotar um critério, capaz de checar quais teorias realmente satisfazem as condições para mediar uma implementação. Somente a existência de uma tradução partindo da teoria construída pelo Teorema da Construtibilidade, que mantenha a capacidade de derivar a tradução de seus axiomas, e a conservação das propriedades do mecanismo lingüístico primitivo, faz-se exigida. Δ_2 representa informações suficientes para uma implementação superposta Σ_2 em Σ_2', mas ainda contendo Δ_1.

Estas condições são, na verdade, bem liberais, pois não obrigam a conservação da implementação declarativa subjacente Δ_1, assim, uma maior variedade de linguagens e teorias acham-se então disponíveis, para serem utilizadas, em consoância com questões metodologicas, de clareza, ou de eficiência, como convém a cada situação particular.

Finalizando, a validade do Teorema da Construtibilidade, e portanto sua aplicabilidade, está condicionada às linguagens lógicas que apresentam a propriedade de interpolação estabelecida pelo Lema da Interpolação de Craig (ver apêndice, lema A.1.1). Igualmente, em [Vel92c], já havia sido estabelecido a mesma relação entre o Lema da Interpolação de Craig e o Teorema da Modularização.

Sintetizando as duas formas de composição apresentadas aqui a tabela abaixo:

Composição de Implementações	Fundamentação Lógica	Aplicabilidade
Composição de dois passos canônicos subsequentes	Teorema da Modularização	Decompor a implementação interpondo especificações para que a descrição definitiva para a linguagem de codificação seja refinada sucessivamente, através do sequenciamento de passos canônicos entre especificações conceitualmente assemelhadas.
Composição de um novo passo canônico com um outro subjacente	Teorema da Construtibilidade	Decompor a implementação a ser realizada em um único passo canônico, simplificando a especificação mais abstrata, de modo que a implementação, uma vez estabelecida em um nível bem trivial, seja recomposta construtivamente até que seja obtido uma implementação para a especificação original.

Esta tabela enfatiza a complementaridade entre as duas formas de composição,

cada uma atua em circunstâncias distintas, enfrentando o mesmo problema sob óticas diferentes.

III.2 Preservação das Propriedades

O desenvolvimento de um programa constitue-se em um número arbitrário de passos de implementação, sequenciados até ser atingida uma versão final satisfatória, geralmente em linguagem mecanicamente executável. Portanto, o requisito fundamental para que este processo transcorra a contento, é ser possível conduzir a capacidade de inferir todas as conseqüências sobre aplicações do passo canônico anteriores, para os passos subsequentes. Mais precisamente, à proporção que passos de implementações são iterados, também são acumuladas propriedades reificadas e decisões de refinamento.

Desta forma, a preservação de propriedades deve ser similarmente componível, no mesmo sentido que o são as implementações. Além disto, a ordem em que estas são compostas precisa ser irrelevante para a preservação.

A composição de interpretações é uma interpretação, então está garantido que cada realização, satisfazendo a implementação declarativa produzida pelo Teorema da Modularização, deriva a tradução dos teoremas de especificação abstrata original, agora descritos na linguagem mais primitiva.

Na composição com implementações subjacentes, a construção de Δ_2 toma como base a implementação declarativa da iteração anterior Δ_1, de modo que $\Delta_1 \subseteq \Delta_2$, portanto, carrega as decisões e refinamentos definidos durante a implementação $\Im_1$, representados em Δ_1. Não há motivos para que este processo, repetido um número indeterminado de vezes, não mantenha a mesma capacidade de inferir conseqüências sobre implementações antecedendo o nível presente.

Há, contudo, uma diferença, já que a extensão entre Δ_1 e Δ_2 não é necessariamente conservativa : fica impossível assegurar que modelos para Δ_1 expandem-se para modelos satisfazendo Δ_2. À porporção que composições com implementações subjacentes são iteradas, algumas possíveis realizações são eliminadas, isto significa aproximadamente um processo indireto de seleção de modelos. Entretanto, Δ_2 é, por construção, consistente, o que implica na existência de modelos, induzidos por ζ_2 em Σ_1.

III.2.1 Conservação das Propriedades

Caso não seja suficiente apenas preservar a capacidade de deduzir conseqüências, e faz-se desejável além, conservar totalmente a integridade da especificação mais concreta bastaria garantir que seja conservativa a extensão de Δ_1 a Δ_2. Pois assim, fica bloqueado a ampliação da conseqüências relativas aos símbolos já implementados e resguardado o trabalho realizado em Δ_1 contra interferências desavisadas, em futuras iterações do processo de construção de uma implementação. Deste modo, Δ_2 apenas derivaria novas propriedades, contanto que estas fossem relativas aos novos símbolos sendo especificados. Mais ainda qualquer realização, satisfazendo a implementação declarativa subjacente, seria expandida a modelos para a nova especificação resultante.

Em determinadas circuntâncias, isto pode vir a ser extremamente desejável, visando evitar a ocorrência de efeitos provocados pela introdução descuidada de novas propriedades em Σ_2'. Muito embora não estivesse realmente comprometida a existência de uma nova composição.

O Teorema da Construtibilidade (teorema III.1.2) atenderia ao que se deseja, contanto que fosse exigido da extensão entre as especificações na linguagem mais primitiva, a sua conservatividade.

Teorema III.2.1 (Teorema da Construtibilidade (versão forte)) *Analogamente, considere as teorias* Σ_1, Σ_2, Σ_1', Δ_1 *e* Σ_2', *esta última consistente. Se*

$$\begin{array}{ccc} \Sigma_2 & & \\ \mathbin{\rotatebox{90}{$\vdash$}} & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 \\ & & \mathbin{\rotatebox{90}{$\vdash$}} \\ & & \Sigma_1' \quad \leq \quad \Sigma_2' \end{array}$$

Então, existe uma teoria Δ_2, *tal que* $\Delta_1 \leq \Delta_2$ *e*

$$\begin{array}{ccccc} \Sigma_2 & & \xrightarrow{\zeta_2} & & \Delta_2 \\ \mathbin{\rotatebox{90}{$\vdash$}} & & & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 & & \mathbin{\rotatebox{90}{$\vdash$}} \\ & & \mathbin{\rotatebox{90}{$\vdash$}} & & \\ & & \Sigma_1' \quad \leq & & \Sigma_2' \end{array}$$

Prova : Antes, considere: Se Σ_1' for consistente, então Σ_2' também o será, trivialmente pela conservatividade da extensão. Porém, através do Teorema da Construtibilidade (III.1.2), já foi determinado que se

$$\begin{array}{ccc} \Sigma_2 & & \\ \mathbb{V} & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 \\ & & \mathbb{V} \\ & & \Sigma_1' \subseteq \Sigma_2' \end{array}$$

então, existe pelo menos uma teoria Δ_2 consistente, tal que

$$\begin{array}{ccccc} \Sigma_2 & & \xrightarrow{\zeta_2} & & \Delta_2 \\ \mathbb{V} & & & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 & & \mathbb{V} \\ & & \mathbb{V} & & \\ & & \Sigma_1' & \subseteq & \Sigma_2' \end{array}$$

onde $\Delta_1 \subseteq \Delta_2$. Logo, resta mostrar que se $\Sigma_1' \leq \Sigma_2'$ então $\Delta_1 \leq \Delta_2$. Entretanto,

$$\begin{array}{ccc} \Sigma_2' & \leq & \Delta_2 \\ \mathbb{V} & & \cup\!\!| \\ \Sigma_1' & \leq & \Delta_1 \end{array}$$

Por construção, $\Delta_2 = \Delta_1 \cup \Sigma_2' \cup \zeta_2[\Sigma_2]$, logo, para qualquer $\alpha \in \mathcal{L}(\Delta_1)$,

$$\text{Se } \Delta_2 \models \alpha \text{ então } \Delta_1 \cup \Sigma_2' \cup \zeta_2[\Sigma_2] \models \alpha$$

mas, foi assumido que $\mathcal{L}(\Delta_1 \cup \Sigma_2') \cap \mathcal{L}(\zeta_2[\Sigma_2]) \subseteq \mathcal{L}(\zeta_1[\Sigma_1])$, portanto, pelo Lema da Interpolação de Craig existe um conjunto de sentenças $I \subseteq \zeta_1[\Sigma_1]$, tal que:

$$\Delta_1 \cup \Sigma_2' \cup I \models \alpha$$

mas $\Sigma_1 \xrightarrow{\zeta_1} \Delta_1$, logo por definição $I \subseteq \Delta_1$, assim,

$$\Delta_1 \cup \Sigma_2' \models \alpha$$

também foi assumido que $\mathcal{L}(\Delta_1) \cap \mathcal{L}(\Sigma_2') \subseteq \mathcal{L}(\Sigma_1')$, e novamente pelo Lema da Interpolação de Craig, existe um conjunto de sentenças $I' \subseteq \Sigma_1'$, tal que:

$$\Delta_1 \cup I' \models \alpha$$

e $\Sigma_1' \leq \Delta_1$, por isto, $I' \subseteq \Delta_1$, e finalmente,

$$\text{Se } \Delta_2 \models \alpha \text{ então } \Delta_1 \models \alpha$$

portanto, $\Delta_2 \leq \Delta_1$, confirmando a válidade do argumento.

Associando este resultado ao Teorema da Construtibilidade, demonstra-se diretamente a sua versão fortalecida.

□

A rigor, as duas versões do Teorema da Construtibilidade, apresentados aqui, mostram-se diferentes exclusivamente porque Δ_1 conserva Δ_2. Em decorrência, Δ_2 não amplia o conjunto das conseqüências, relacionadas à linguagem de Δ_1, resguardando, desta maneira, propriedades já presentes e as decisões de refinamento introduzidas na construção de Δ_1, isto é, justamente o que era inicialmente pretendido. A parte criativa representada na implementação subjacente será, deste modo, conservada na composição recém definida.

Quando necessário, o Teorema da Construtibilidade (versão forte) pode ser empregado à composição de uma implementação por sobre outra, conforme se vê na figura III.2.7.

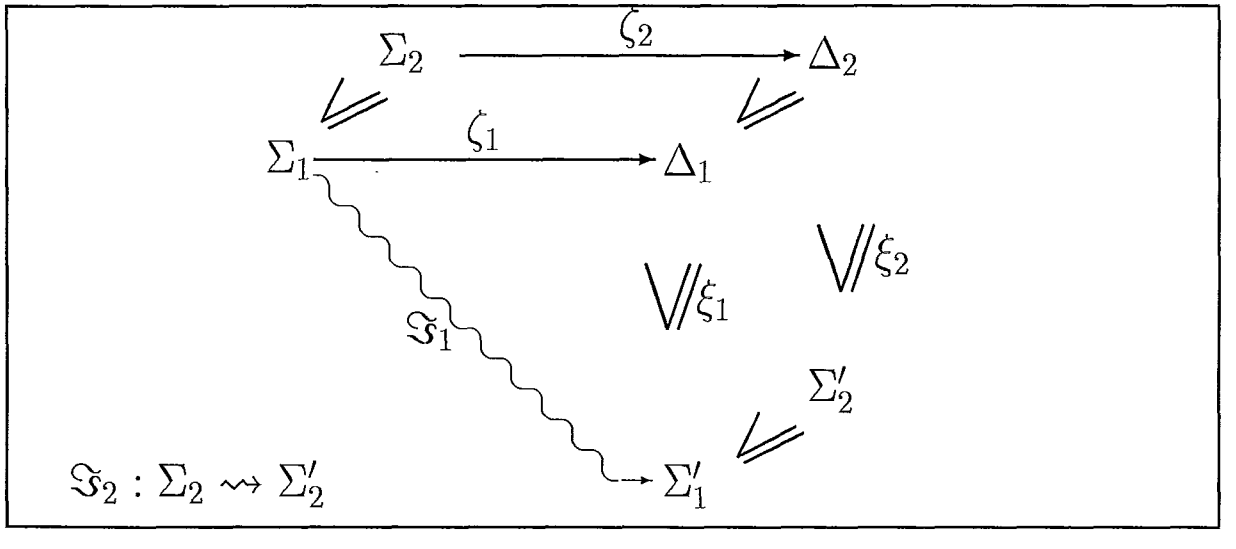


Figura III.2.7: Aplicação do Teorema da Construtibilidade (versão forte)

As duas versões do Teorema da Construtibilidade podem inclusive ser iteradas, alternando-se de acordo com as necessidades surgidas a cada passo da implementação. Esta opção proporciona o controle para ora permitir a especialização do trabalho já real-

izado, ora conservá-lo, bloqueando influências posteriores.

Passos de implementação são, em um certo sentido, componíveis refletindo a composição de extensões entre implementações declarativas construídas em cada um. Então, nada impede que uma próxima superposição, envolvendo extensões não conservativas, interfira em conseqüências antes conservadas, pois o mecanismo permitiria ampliar os teoremas relacionados com qualquer subconjunto da linguagem, na implementação declarativa subjacente, desde que seja mantida a consistência. Portanto, uma atenção especial é sempre requerida, para decidir quando uma representação deve ser necessariamente conservada, ou meramente preservada.

III.3 Referências Bibliográficas

A idéia de desenvolver programas por composição de passos de implementação veio de [Vel87], e motivou este trabalho. O Teorema da Modularização (III.1.1) aplicado à composição de passos canônicos subsequentes é argumentada em [Mai86] e em [TM87]. Finalmente, em [Vel92c] há uma demonstração definitiva para a válidade do argumento, e novamente em [Vel93] uma nova prova, mais simples, confirma o Teorema da Modularização.

A relação entre o Teorema da Modularização e o Teorema da Consistência de Robinson Extendido (A.3.1) foi apresentada em [TM87]. A importância da propriedade de interpolação de Craig (A.1.1), para o desenvolvimento formal, está discutida em [MS85], e ainda mais evidenciada, com a demonstração para o Teorema da Modularização em [Vel92c]. A preservação da capacidade de simular a inferência de propriedades durante a tradução entre níveis lingüísticos, foi explorada em [TM87].

Capítulo IV

Composição de Implementações Subjacentes

Este capítulo revela-se mais pragmático, porque objetiva trazer a exemplos mais concretos os mecanismos referidos no capítulo anterior, aproximando as considerações lógico-teóricas da prática de implementação. Prioritariamente, este capítulo sugere algumas soluções realistas para dificuldades bem conhecidas e aponta novas possibilidades para formalizar o *Desenvolvimento de Programas com Tipos Abstratos de Dados.*

Contudo, o enfoque fica voltado quase exclusivamente sobre a composição com implementação subjacente, visto que a composição entre implementações subsequentes já veio merecendo atenção nos trabalhos que nos precederam. Algumas entre as referências citadas exploram convenientemente a aplicabilidade para o Teorema da Modularização (III.1.1).

Convém a este capítulo preocupar-se com aplicações reais para o Teorema da Construtibilidade na construção de implementações, e discutir como empregar conjuntamente ambas as formas de composição, favorecendo-se das caracaterísticas particulares de cada uma.

A seção inicial propõe adotar uma estratégia eminentemente construtiva para definir passos canônicos mais complexos, fundando no Teorema da Construtibilidade a recombinação dos resultados. Enquanto a segunda seção a completa, aludindo os benefícios de proceder também a verificação construtivamente.

A seção IV.3 abre espaço para a evolução dos objetivos e requesitos de uma

implementação, ainda que após a "conclusão"[1] desta, contornando inclusive a exigência da conservatividade. Finalmente a seção IV.4 discute como ambos os mecanismos prestam-se para viabilizar a reusabilidade de especificações e implementações. E a quinta seção fundamenta, através da composição por sobre um passo canônico subjacente, a implementação entre Tipos Abstratos de Dados parametrizados.

IV.1 Implementação Construtiva

A cada passo do processo de implementação, símbolos, presentes na especificação mais abstrata, são decompostos em símbolos mais concretos, expressando o significado de suas propriedades agora em termos de sortes e operações mais primitivos, pertencentes a um repertório especialmente escolhido.

As decisões envolvidas no refinamento envolvem, contudo, conhecimento razoável de ambos os níveis linguísticos, pois a representação escolhida para cada símbolo abstrato precisa ser particularmente apropriada àquela linguagem destino. Em geral, a construção de uma implementação declarativa já exige que alguns aspectos sejam antevistos, como a interpretação que associa símbolos na linguagem abstrata aos seus refinamentos. Por tudo isto, a implementação requer um esforço criativo não desprezível.

Além das limitações naturais, inerentes à lide com quantidades razoáveis de conceitos complexos, existe ainda uma dificuldade adicional : o forte interelacionamento entre como objetos concretos são combinados e estruturados para representar objetos mais abstratos, e como deverão ser descritas as operações para manipular estes objetos.

A seleção de um repertório de operações para manipular sortes abstratos não pode ser dissociada da escolha de representações para estes sortes, sob pena de comprometer a própria semântica da especificação.

Esta situação torna-se, na verdade, bem difícil de caracterizar, e abordá-la usualmente envolve o uso indisciplinado de criatividade para escolher a "melhor" combinação de opções. O que realmente vem a ser "melhor", depende da influência de inúmeros fatores, boa parte não perfeitamente quantificáveis, como :

- conhecimento sobre o sentido global de todo o projeto;

[1] A rigor, uma implementação nunca está definitivamente concluida, daí as áspas, pois há sempre perspectiva de evoluir, em conformidade com a dinâmica da situação real representada

- algum sentimento sobre o que é matematicamente correto, e uma noção sobre o que é computacionalmente tratável;
- considerações sobre eficiência;
- experiência e intuição, ou requisitos ainda mais subjetivos.

Um importante aspecto a ser igualmente considerado é a exigência de que $\mathcal{T}'$, destinado a suportar a implementação de $\mathcal{T}$, esteja *a priori* completamente especificado, para a aplicação do passo canônico. Isto, provavelmente ocasionará algumas dificuldades na definição da classe de modelos $\mathcal{T}'$, principalmente em estágios intermédiarios, ou se a distância conceitual entre as especificações de $\mathcal{T}$ e $\mathcal{T}'$ for significativa.

Na prática, duas situações exatamente opostas tendem a ocorrer. Primeiro, $\mathcal{T}'$ pode haver sido especificado como um repertório de operações demasiadamente simplificado, comprometendo uma implementação eficiente, ou ao menos, dificultando a construção de uma extensão conservativa. Talvez, porque parte significativa das propriedades que deveriam estar presentes em sua especificação, hajam sido compreendidas tardiamente, apenas durante a implementação.

Em geral, costuma ser difícil vislumbrar, antes de tentar uma implementação, o que será necessário especificar constituindo o TAD mais primitivo. Por este motivo, o projetista pode ser levado a retroceder e redefinir a especificação para $\mathcal{T}'$, se não pelo menos, a dedicar atenção adicional ao estender $\mathcal{T}'$, tentando enriquecer e superar o reduzido poder de expressividade de sua linguagem.

Outras vezes, ocorre justamente o inverso, $\mathcal{T}'$ foi super especificado, ou porque seu repertório dispõe de um número maior de operações e sortes do que realmente seria requerido, ou porque suas operações são desnecessariamente mais elaboradas. Em todo caso, pouca liberdade é deixada quando este, por sua vez, vier a ser implementado em termos de um outro TAD ainda mais primitivo. À medida em que o processo é iterado, ganha-se complexidade cada vez maior, introduzindo detalhes não relevantes ao problema original, perdendo gradativamente a precisa noção dos objetivos globais do projeto.

Ambas as situações mostram-se claramente indesejáveis. Entretanto, isto não obscurece os méritos da aplicação do passo canônico ao desenvolvimento de programas, pois, os problemas comentados acima não lhe são inerentes. São, na realidade, decorretes da limitação humana em manipular razoável quantidade de conceitos complexos. Ao contrário, o passo canônico não condiciona *per si* qualquer estratégia para definir a

interpretação e a extensão conservativa que implique em complicações extrínsecas.

Manipular quase sempre com informações incompletas e/ou parcialmente compreendidas é sem dúvidas o ponto crítico na implementação. Pouco provável estar disponível a princípio, o conhecimento integral a respeito de todas as peculiaridades do conceito a ser representado, e é a própria implementação que costuma, nestas situações, ser elucidativa. Isto costuma provocar uma abordagem em estilo *tentativa e erro*, desgastante e ineficiente, ou mais drasticamente transtornar todo o projeto.

Não há realmente meios automáticos, para orientar a compreensão a respeito de um conceito qualquer, nem sequer para apreender detalhes e decidir sua relevância, de forma a produzir informações em ordem, exatamente conforme decisões de refinamento deveriam ser sequenciadas, a fim de obter resultados satisfatórios. Por conseguinte, resta atacar a outra face do problema.

Acaso fosse possível decompor a atividade de implementação e proceder os refinamentos, concentrando-se apenas sobre domínios bem mais reduzidos, e havendo meios para recombinar os resultados obtidos, compondo cada nova implementação a partir de alguns passos relativamente simples, o processo como um todo refletiria os efeitos desta disciplina. Seria então possível, manter sob controle a complexidade inerente e reduzir significativamente o esforço para, em um só passo, realizar a implementação.

Para tanto, há que simplificar a especificação, sucessivamente até que a decomposição resulte em uma nova, contida na descrição anterior, e simples o suficiente para permitir uma aplicação quase direta do passo canônico. Então, retrocede-se um nível na simplificação e tenta-se estender os resultados obtidos no passo anterior, compondo com as implementações mais simples. Continuando assim, até finalmente ser possível oferecer uma implementação completa e satisfatória para a especificação inicial.

Em resumo, argumenta-se a viabilidade de proceder a aplicação do passo canônico, por assim dizer, em "micropassos", passo a passo. Esta estratégia procura, simplesmente, imitar e formalizar o que faz, por intuição, um bom projetista. Assim, o desenvolvimento da implementação capta o mesmo caráter construtivo que é apresentado no desenvolvimento da especificação.

Esta proposta concorda com o paradigma de desenvolvimento de programas por *Refinamentos Sucessivos,* levando ainda mais além as mesmas idéias, que inspiraram a formalização do passo canônico.

IV.1.1 Estratégia para Implementação Construtiva

A utilização prática destes resultados motiva a criação de uma estratégia bem fundada para sistematizar o desenvolvimento incremental de implementações, enquanto a influência criativa é disciplinada. Considerando construir uma implementação como um problema a ser resolvido, argumenta-se aqui a aplicabilidade de uma estratégia estilo *Divisão e Conquista*,[2] para desenvolver uma solução correta e eficiente.

Se há dificuldades em implementar um TAD especificado por um determinado repertório de símbolos, por que não tentar antes estabelecer um passo de implementação, envolvendo apenas um pequeno subconjunto de sortes e operações, e gradualmente prosseguir ampliando os resultados conseguidos, até abranger todos os símbolos da linguagem inicial ?

Esta seção, procura basicamente esboçar uma resposta realizável para esta pergunta, delineando uma orientação, formalmente comprovada, para proceder passos de implementação relativamente mais complexos.

Seja Σ a especificação de um TAD a ser implementado. Ao selecionar um reduzido subconjunto de sua linguagem fica induzida a formação de uma subestrutura[3] σ_1, ou seja, tomando como repertório os símbolos neste subconjunto da linguagem de Σ, e como semântica os axiomas em Σ, restritos a este repertório.

A teoria descrita por σ_1 pode ser feita tão reduzida quanto se desejar, portanto, faz sentido estabelecê-la simples o suficiente para ser compreendida inteiramente. Aconselha-se direcionar as escolhas, no sentido de caracterizar como σ_1, o que parece mais relevante ou mais "central" à especificação Σ. Em princípio, estas decisões podem talvez apoiar-se em aspectos mais ou menos subjetivos, porém seria ideal conseguir formalizar critérios para orientá-las.

Entretanto, ao invés de tentar definir de uma vez σ_1, seria melhor ir progressivamente excluindo da especificação Σ, componentes e propriedades que pareçam ser contigênciais, ou dependentes de outros elementos, mais centrais na especificação, até ser escolhido como σ_1, uma subestrutura restante. O critério para decidir quando σ_1 já está suficientemente simples, talvez não seja bem determinado, por envolver fatores singulares a cada situação específica.

---

[2] veja [Vel80] e [VV81]

[3] apresentação para uma teoria contida na teoria especificada por Σ

Cada subestrutura σ_i, nesta simplificação, é por transitividade, uma subestrutura de Σ, e em decorrência, fica simultaneamente definida uma orientação natural, para a etapa de recombinação.

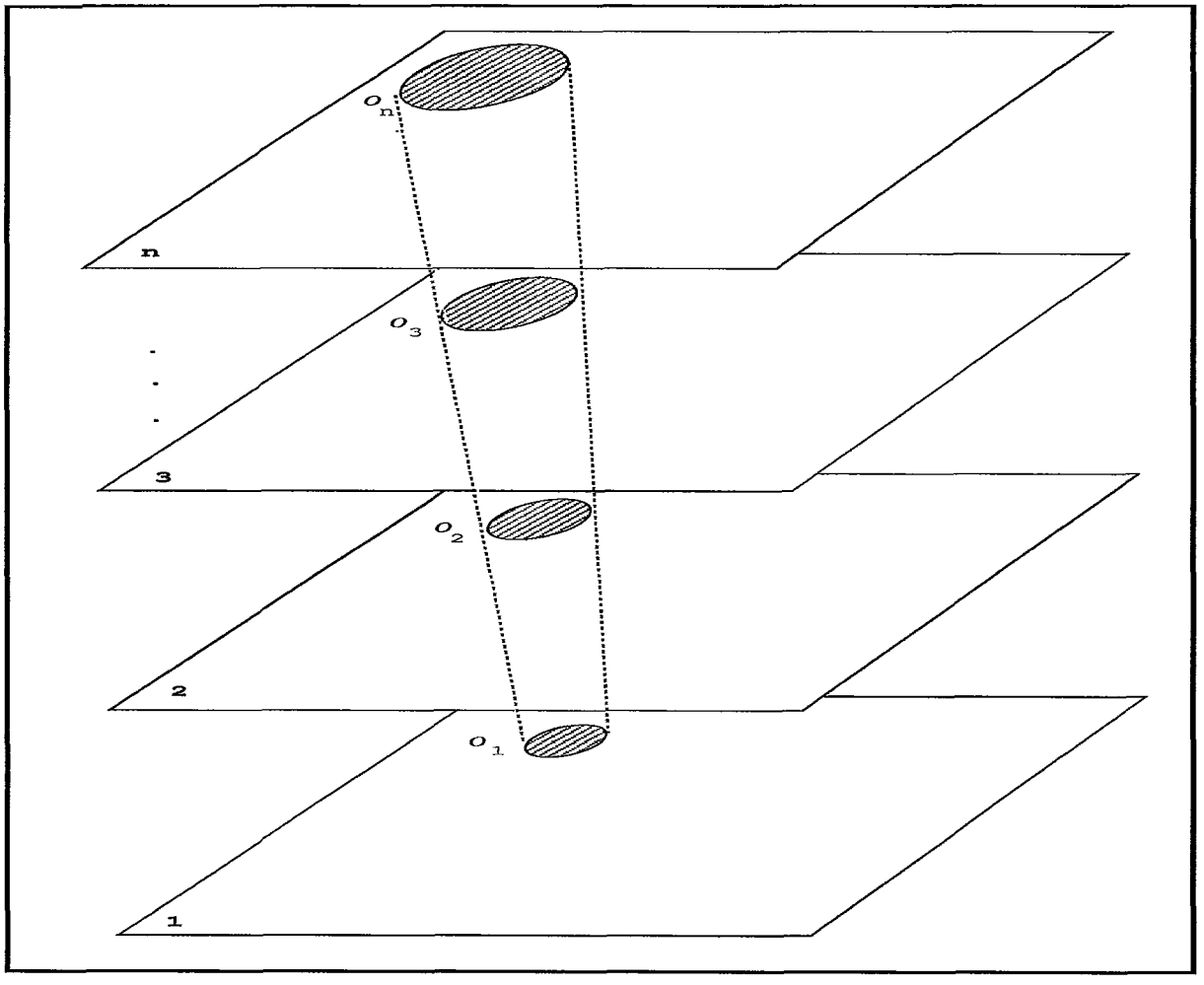


Figura IV.1.1: Simplificando a especificação Σ

Cada nível de simplificação i está associado a uma subestrutura, σ_i, que será implementada a partir de resultados fornecidos pelo passo subjacente, implementando σ_{i-1}. Para tanto, cada subestrutura σ_i, resultante da eliminação de alguns símbolos e axiomas, deve ser conservativamente extendível à anterior, ou seja, σ_{i+1}, conforme a figura ??. A importância desta condição ficará mais clara adiante.

Outra vantagem nesta estratégia, é contornar a exigência de uma especificação, Σ', *a priori* definitiva, para prover o suporte adequado à implementação de Σ. Porque não ir especificando Σ', recombinando subestruturas projetadas para receber a implementação de cada subestrutura correspondente, gerada na simplificação Σ?

Havendo escolhido σ_1, a fim de implementá-lo, é concebida, então uma especificação σ_1'. Neste ponto, somente precisaria suportar a implementação de σ_1, que é comparativamente menos complexa que a especificação original, Σ, pois dispõe apenas, de um pequeno subconjunto de sortes e operações, entre os que existiam inicialmente. Em geral, neste nível, lida-se somente com o sorte de interesse, e um pequeno grupo de operações primárias. Logo, σ_1' precisa, por sua vez, fornecer um conjunto de operações e sortes igualmente simplificado.

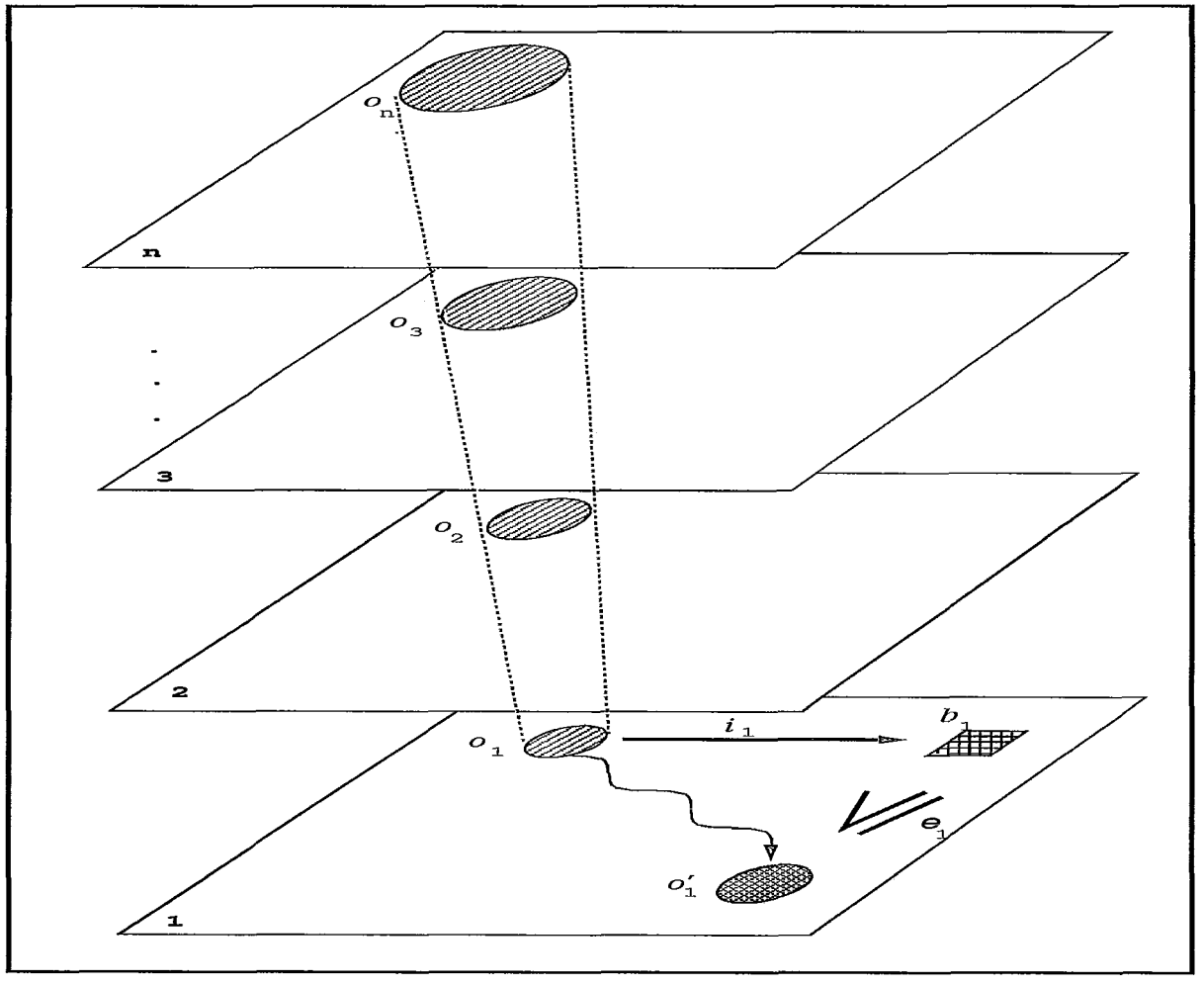


Figura IV.1.2: Implementando a subsestrutura resultante, σ_1

Então, concentrando-se unicamente em σ_1, realiza-se um primeiro ensaio, estabelecendo a aplicação inicial do passo canônico, para implementar σ_1 sobre σ_1', construindo uma extensão conservativa ϵ_1 de σ_1', acrescentando os símbolos juntamente com alguns axiomas, de modo a fornecer a especificação semântica para o refinamento dos símbolos em σ_1, e a representação das decisões tomadas. Feito isto, completa-se a especificação da implementação declarativa δ_1. Finalmente, é definida uma interpretação ι_1, associando

cada símbolo em σ_1 um correspondente em δ_1.

Quando estiver concluido este primeiro protótipo, parte-se para uma subestrutura maior, σ_2, retrocedendo um nível na simplificação, para então, implementá-la, utilizando os resultados já representados por $\langle \epsilon_1, \iota_1 \rangle$. A implementação dos símbolos que agora aparecem em σ_2, e haviam sido retirados para obter σ_1, é concluida sobre uma extensão consistente de σ'_1, denominada σ'_2, definindo o par $\langle \epsilon_2, \iota_2 \rangle$, e construindo como resultado δ_2. Assim sucessivamente, até finalmente atingir a especificação completa Σ, quando $\sigma_n \equiv \Sigma$.

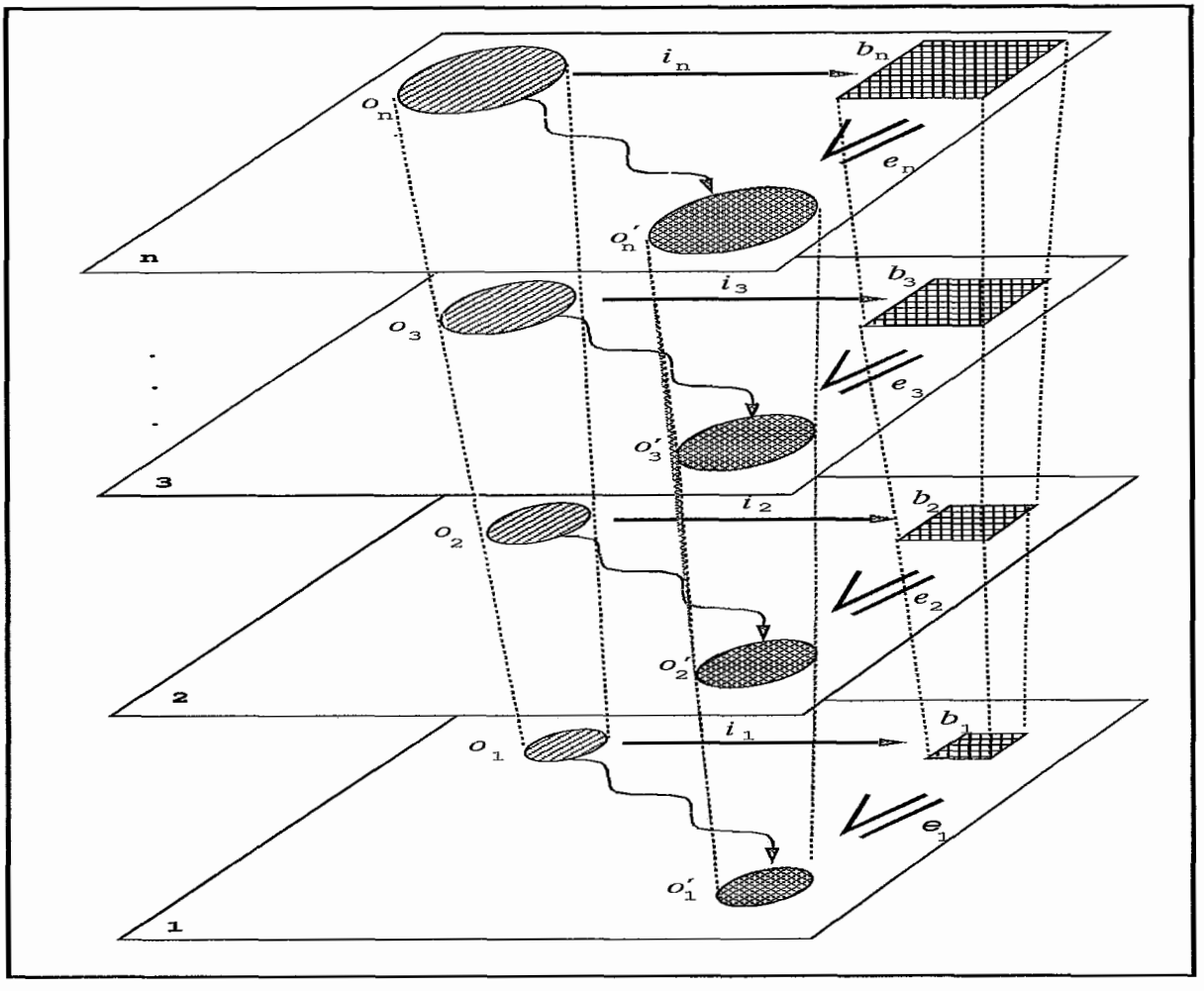


Figura IV.1.3: Recombinando uma implementação para Σ

Após concluida a recombinação, é obtido $\langle \epsilon_n, \iota_n \rangle$, que é exatamente a implementação $\Sigma \rightsquigarrow \Sigma'$ pretendida. Da mesma forma, a implementação declarativa desejada é justamente dada por δ_n. Paralelamente, σ'_n, construida durante a recombinação, especifica o mecanismo lingüístico, Σ', onde se desenvolve a implementação.

A implementação de Σ foi desenvolvida construtivamente passo a passo, limitando a cada momento a quantidade de informações a serem consideradas e retringindo a atenção ao conhecimento local, que deve ser empregado na representação de cada pequeno grupo de sortes e operações, conforme os objetivos principais desta proposta.

No procedimento, consoante descrito acima, está caracterizado a utilização de uma estratégia estilo *Divisão e Conquista,* destacando três etapas bem definidas, em seqüência:

- Simplificação
- Trivialização
- Recombinação

Embora, ocasionalmente exista um passo de implementação mais eficiente, é sempre justificável uma estratégia sistemática, correta e conceitualmente simples para proceder a implementação.

IV.1.1.1 Simplificação

A idéia mais natural, é fundamentar a simplificação justamente no " histórico" do desenvolvimento da especificação. Mais precisamente, deseja-se aproximar um possível "histórico", entre os que podem ter supostamente conduzido àquela especificação, e que seja consoante com aquilo que realmente ocorreu na sua construção.

Não é factível concluir exatamente a seqüência em que as propriedades foram abstraídas e então reificadas nos axiomas, nem a ordem exata em que foram tomadas as decisões de refinamento, posto que, estas informações não se encontram explicitamente descritas na especificação. Contudo, é possível identificar, baseando-se na especificação semântica, algumas informações aplicáveis a etapa de simplificação. Observe a especificação para o TAD *Dispatcher*[*Process*]:

ADT *Dispatcher*
Syntax
Import *Process*

Function $inicialize : \longmapsto Dispatcher;$

$schedule : Dispatcher \times Process \longmapsto Dispatcher;$

$release : Dispatcher \longmapsto Dispatcher;$

$next : Dispatcher \longmapsto Process$

Predicate $findprocess(Dispatcher);$

Axiomatization

$$(\forall D : Dispatcher)D \stackrel{D}{=} inicialize \vee$$

$$(\exists D' : Dispatcher)(\exists P : Process)D \stackrel{D}{=} schedule(D', P) \quad (1)$$

$$\neg FindProcess(inicialize) \quad (2)$$

$$(\forall D : Dispatcher)\neg findprocess(D) \Rightarrow \neg findprocess(release(D)) \quad (3)$$

$$(\forall D : Dispatcher)findprocess(D) \Rightarrow$$

$$(\exists D' : Dispatcher)release(D) \stackrel{P}{=} D' \wedge \neg D \stackrel{D}{=} D' \quad (4)$$

$$(\forall D : Dispatcher)findprocess(D) \Leftrightarrow$$

$$(\exists D' : Dispatcher)(\exists P : Process)D \stackrel{D}{=} schedule(D', P) \quad (5)$$

$$(\forall D : Dispatcher)(\forall P : Process)$$

$$D \stackrel{D}{=} schedule(Inicialize, P) \Rightarrow P \stackrel{P}{=} next(D) \quad (6)$$

$$(\forall D : Dispatcher)findprocess(D) \Rightarrow$$

$$(\exists P : Process)next(D) \stackrel{P}{=} P \quad (7)$$

End

Por instância, é bastante razoável que um símbolo de sorte (ou operação), que haja sido utilizado para definir outro qualquer, seja implementado antes, já que uma representação para o outro poderia ser escolhida, tomando como base a sua própria implementação. A hieràrquia em unidades já fornece uma primeira evidência da existência de um tal sequenciamento. Também, em $Dispatcher[Process]$, as funções *next* e *release* foram expressas através do relacionamento com a função *schedule*, logo, o refinamento de qualquer uma pode ser cuidado, tomando como partida a representação estabelecida para *schedule*. É exatamente esta noção intuitiva que caracteriza, de certa forma, uma relação de precedência temporal embutida na axiomatização.

Obviamente, qualquer relacionamento deste tipo está proximamente ligado ao desenvolvimento da especificação. Novos símbolos são acrescidos à linguagem, juntamente com suas propriedades, mas procurando não interferir com as conseqüências já presentes. Isto sugere tentar encontrar um suposto histórico da construção, na forma de algum se-

quenciamento de extensões conservativas, que presumivelmente resultaria na especificação completa. Posto um sequenciamento possível, há indicações suficientes para deduzir, pelo menos aproximadamente, quais sortes e operações foram primariamente definidos e quais foram só posteriormente, através de algumas dependências entre as definições de símbolos.

Se um símbolo e seus axiomas podem ser adicionados conservativamente, em algum subconjunto da especificação, que não o contém, certamente este símbolo é contingencial, em relação aos demais presentes no subconjunto. Assim, é possível estabelecer uma relação de precedência, talvez parcial, que serviria como orientação para simplificar a especificação.

A cada momento, é escolhido algum símbolo (ou mesmo símbolos), em relação ao qual não há qualquer outro contingencial ainda presente, para ser retirado. Ou seja, a definição de nenhum símbolo, ainda não retirado, dependeria da sua. No exemplo, o primeiro símbolo a ser esquecido poderia ser *next*. Então, retringe-se a linguagem do passo anterior eliminando-o, e consequentemente, tudo o que a ele se refere na especificação. Assim, retira-se da especificação semântica todos os axiomas que apresentam símbolos eliminados, e.g. (6) e (7). Analogamente, os símbolos de sorte podem ser retirados da linguagem quando não existir mais nenhuma operação que os referencie.

ADT *Dispatcher*

Syntax

Import *Process*

Function $inicialize : \longmapsto Dispatcher;$

$schedule : Dispatcher \times Process \longmapsto Dispatcher;$

$release : Dispatcher \longmapsto Dispatcher;$

Predicate $findprocess(Dispatcher);$

Axiomatization

$$(\forall D : Dispatcher)D \stackrel{D}{=} inicialize \vee$$
$$(\exists D' : Dispatcher)(\exists P : Process)D \stackrel{D}{=} schedule(D', P) \quad (1)$$

$$\neg FindProcess(inicialize) \quad (2)$$

$$(\forall D : Dispatcher)\neg findprocess(D) \Rightarrow \neg findprocess(release(D)) \quad (3)$$

$$(\forall D : Dispatcher)findprocess(D) \Rightarrow$$
$$(\exists D' : Dispatcher)release(D) \stackrel{P}{=} D' \wedge \neg D \stackrel{D}{=} D' \quad (4)$$

$$(\forall D : Dispatcher)findprocess(D) \Leftrightarrow$$

$$(\exists D' : Dispatcher)(\exists P : Process)D \stackrel{D}{=} schedule(D', P) \quad (5)$$

End

Resultando, deste nível a formação de σ_i, subestrutura de σ_{i+1}, produzida no passo anterior, formado pela retirada de *next* e dos axiomas (6) e (7). Então, diz-se que σ_i determina um novo nível de simplificação.

Continua progressivamente a simplificação, pretendendo reduzir ao máximo, a quantidade de novos elementos, pertencentes a cada nível, pois estes serão conjuntamente implementados. Repetindo este procedimento um número suficiente de vezes, eliminando em seqüência *release*,

ADT *Dispatcher*

Syntax

Import *Process*

Function $inicialize : \longmapsto Dispatcher;$

$schedule : Dispatcher \times Process \longmapsto Dispatcher;$

Predicate $findprocess(Dispatcher);$

Axiomatization

$$(\forall D : Dispatcher)D \stackrel{D}{=} inicialize \vee$$

$$(\exists D' : Dispatcher)(\exists P : Process)D \stackrel{D}{=} schedule(D', P) \quad (1)$$

$$\neg FindProcess(inicialize) \quad (2)$$

$$(\forall D : Dispatcher)findprocess(D) \Leftrightarrow$$

$$(\exists D' : Dispatcher)(\exists P : Process)D \stackrel{D}{=} schedule(D', P) \quad (3)$$

End

e logo após, *findprocess*, *schedule* e finalmente *inicialize*, até que seja resultante uma subestrutura σ_1, que representa as propriedades fundamentais, sobre as quais se apoiaria o restante da especificação Σ. Quanto mais simples for feito σ_1, maiores certamente serão os ganhos a conseguir com a decomposição da implementação.

Proposição IV.1.1 *Seja Σ uma teoria consistente, e $\mathcal{L}(\Sigma) \neq \emptyset$ a sua linguagem. Então existe sempre uma seqüência de teorias $\sigma_1, \sigma_2, \ldots, \sigma_n$, para $n > 1$, tal que:*

$$\sigma_1 \leq \sigma_2 \leq \ldots \leq \sigma_n$$

e $\sigma_n \equiv \Sigma$.

Prova : Seja a teoria consistente σ_i, e seja λ_i um subconjunto próprio de $\mathcal{L}(\sigma_i)$, ou seja $\lambda_i \subset \mathcal{L}(\sigma_i)$, então:

$$\mathrm{Cn}(\sigma_i|_{\lambda_i}) \leq \sigma_i$$

Aplicando indutivamente este argumento, e como é sempre possível escolher uma seqüência de subconjuntos da linguagem, cada um fazendo o papel de um λ_i, de tal forma que cada λ_i seja subconjunto próprio do posterior, pois a linguagem de Σ não é vazia, a restrição da linguagem a cada subconjunto λ_i, correspondente induz uma seqüência de teorias, $\sigma_1,\sigma_2,\ldots,\sigma_n$, onde cada uma é extensão conservativa da posterior. Já que $\mathcal{L}(\Sigma) \neq \emptyset$, então n pode ser pelos menos maior que 1. □

Pela proposição, sempre é possível conduzir a simplificação de uma teoria, produzindo um certo número de subestruturas. Ainda que não haja garantia explícita, que cada teoria gerada seja finitamente axiomatizável. Mesmo que seja irreal lidar com teorias infinitas, diretamente, sem dispor de especificação finita, isto continua indiferente à aplicação do Teorema da Construtibilidade, pois este estabelece uma relação entre teorias, não importando a existência ou não de apresentações.

Por outro lado, a simplificação é quanto mais vantajosa, quanto mais níveis forem gerados, pois o objetivo claro é reduzir ao máximo, o número de informações a serem implementadas em cada um. Uma sugestão razoável seria selecionar sempre um pequeno subconjunto de sortes e operações, na linguagem corrente, de modo que sua eliminação da teoria não retira das conseqüências, propriedades relativas a outros símbolos não pertencentes ao subconjunto escolhido. Isto indica, que não há mais símbolos na linguagem ainda definidos em termos dos sortes e operações retirados, e consequentemente, tende a facilitar a futura implementação dos mesmos, espelhando as dependências expressas pela especificação. [4]

IV.1.1.2 Trivialização

A primeira questão a ser respondida nesta seção deve ser : Até quando vai o processo de simplificação, ou seja, quando há condições para afirmar que a subestrutura gerada é simples o suficiente, para permitir uma aplicação quase direta do passo canônico ? Não existe realmente uma resposta única, que seja consensualmente satisfatória a todos os

[4] Até aqui, foi evitado decidir qualquer critério mais rígido para conduzir a simplificação, porque isto seria impertinente ao escopo deste trabalho.

casos. Talvez, informações particulares ao TAD sendo implementado, e à disponibilidade de elementos no mecanismo lingüístico concreto, sejam importantes para determinar esta noção de suficiência.

Todavia, uma alternativa apropriada seria : isolar o sorte de interesse, definindo como σ_1 a teoria inicial mas restrita a linguagem contendo somente o símbolo que o representa, no exemplo antorior seria deixado em σ_1 somente o próprio sorte *Dispatcher*. Esta teoria não é usualmente vazia, pois há sempre uma símbolo de igualdade associado a cada sorte, e portanto, a teoria ainda expressa em sentenças, propriedades, por exemplo relativas à cardinalidade, ainda que não existam, na linguagem, operações que manipulam instâncias do sorte. Estas propriedades, independentes das operações, costumam oferecer informações suficientes para lhe construir uma representação adequada.

Esta sugestão está naturalmente apoiada na experiência empírica; intuitivamente, o que poderia ser mais primordial na especificação para uma TAD, se não o referido sorte de interesse. Sua representação seria a escolha óbvia a ser cuidada em primeiro lugar, pois, justamente sobre ela fundamenta-se a implementação de todo o repertório de operações referenciando o sorte de interesse.

Definido σ_1, resta então iniciar a especificação para o mecanismo lingüístico concreto. Mas a princípio, somente estabelecendo alguns sortes básicos, juntamente, talvez, com predicados de relativização ou relações de equivalência que formam correspondentemente a parte principal da linguagem concreta. A especificação para o sorte *Dispatcher* seria implementada então em termos de um novo sorte, *PriorityQueue*, escolhido exclusivamente para este passo canônico.

Então, apenas empregando mecanismos de extensão por definição de sortes (estudados em [MV92]), pode ser construido uma primeiro protótipo, δ_1, para a implementação declarativa, representando *Dispatcher*:

ADT *Dispatcher**

Syntax

Import PriorityQueue[Process],Process

Axiomatization

$$(\forall d : Dispatcher^*)(\exists q : PriorityQueue) d \stackrel{Q}{=} q \qquad (1)$$

End

Resta estabelecer uma interpretação, ι_1, traduzindo o sorte *Dispatcher*, em um

sorte definido na extensão ϵ_1, no caso *Dispatcher**, porém respeitando suas propriedades inferidas na teoria σ_1. Indiretamente, *Dispatcher** herdou propriedades particulares de *PriorityQueue*[*Process*], como a sua cardinalidade. A aplicação do passo canônico neste nível está assim terminada.

Concluido isto, a implementação já tem um passo básico, que pode ser propagado, retrocedendo os níveis de simplificação, implementando as operações restantes e os demais sortes, durante a etapa de recombinação.

IV.1.1.3 Recombinação

O objetivo principal da etapa de recombinação é obter ao final uma especificação para o módulo de implementação. Nesta etapa, realiza-se efetivamente a implementação de uma especificação mais abstrata, sobre uma outra mais concreta, talvez construida especialmente para este passo.

A construção de forma incremental da implementação declarativa espelha o próprio histórico do desenvolvimento da especificação Σ, pois segue, em ordem inversa, os níveis gerados durante a etapa de simplificação, enquanto, o passo canônico é aplicado seguidamente às subestruturas $\sigma_1,\sigma_2,\ldots,\sigma_n$ de Σ.

A cada nível $i+1$ de simplificação, precisa ser construida uma extensão da especificação σ'_i, para suportar a implementação de σ_{i+1} deste nível. Esta extensão, chamada de σ'_{i+1}, é formada introduzindo-se, se necessário, novos sortes e operações, estruturados, pelo menos, a partir de símbolos em σ'_i, aumentando o repertório já disponível. Igualmente, acrecentando à especificação semântica os axiomas que definem o significado das novas funções e predicados primitivos. Por exemplo, a fim de representar *next*, até o momento esquecido do processo de recombinação, é preciso que o TAD *PriorityQueue*[*Process*] disponha de mais uma operação, *high*, completando a especificação a seguir:

ADT *PriorityQueue*[*Process*]

Syntax

Import *Process*

Function $create : \longmapsto PriorityQueue;$

$put : PriorityQueue \times Process \longmapsto PriorityQueue;$

$get : PriorityQueue \times Process \longmapsto PriorityQueue;$

$high : PriorityQueue \longmapsto Process$

Predicate $empty(PriorityQueue);$

Axiomatization

$$(\forall q : PriorityQueue)$$
$$q \stackrel{Q}{=} create \vee (\exists q' : PriorityQueue)(\exists p : Process) q \stackrel{Q}{=} put(q', p) \quad (1)$$
$$empty(create) \quad (2)$$
$$(\forall q : PriorityQueue)(\exists p : Process) \neg empty(put(q, P)) \quad (3)$$
$$(\forall q : PriorityQueue)(\exists p : Process) get(put(q, p), id(p)) \stackrel{Q}{=} q \quad (4)$$
$$(\forall q : PriorityQueue)(\exists p : Process)$$
$$\neg p \stackrel{P}{=} p' \Rightarrow get(put(q, p), p') \stackrel{Q}{=} put(get(q, p'), p) \quad (5)$$
$$(\forall q : PriorityQueue)(\forall p : Process)$$
$$high(put(q, p)) \stackrel{P}{=} p \Leftrightarrow get(put(D, P)) \stackrel{D}{=} D \quad (6)$$
$$(\forall D : PriorityQueue)(\forall p : Process)\{\neg high(put(q, p)) \stackrel{P}{=} P \Leftrightarrow$$
$$get(put(q, p)) \stackrel{D}{=} put(get(q), p)\} \quad (7)$$

End

Caso a etapa de simplificação haja isolado as informações não relevantes, em cada momento, a implementação em um dado nível depende somente do passo de implementação, executado no nível de simplificação imediatamente inferior. Pois em cada nível, a interpretação e a extensão conservativa são constuídas, estendendo o par imediatamente subjacente, para aqueles símbolos presentes pela primeira vez neste nível. Em mais uma instância, é sensilvelmente limitado o foco de atenção a apenas poucos símbolos. Por fim, a recombinação compõe a seguinte implementação declarativa:

ADT $Dispatcher^*$

Syntax

Import $PriorityQueue[Process], Process$

Function $inicialize^* : \longmapsto Dispatcher^*;$
$schedule^* : Dispatcher^* \times Process \longmapsto Dispatcher^*;$
$release^* : Dispatcher^* \longmapsto Dispatcher^*;$
$next^* : Dispatcher^* \longmapsto Process$

Predicate $findprocess^*(Dispatcher^*);$

Axiomatization

$$(\forall d : Dispatcher^*)(\exists q : PriorityQueue) d \stackrel{H}{=} q \quad (1)$$
$$inicialize^* \stackrel{H}{=} create \quad (2)$$

$$(\forall d : Dispatcher^*) findprocess^*(d) \Leftrightarrow \neg empty(d) \quad (3)$$

$$(\forall d : Dispatcher^*)(\forall P : Process) schedule^*(d,p) \stackrel{H}{=} put(d,p) \quad (4)$$

$$(\forall d : Dispatcher^*) \neg empty(d) \Rightarrow release^*(d) \stackrel{H}{=} get(d, high(d)) \quad (5)$$

$$(\forall d : Dispatcher^*)(\forall P : Process) next^*(d) \stackrel{P}{=} p \Leftrightarrow p \stackrel{P}{=} high(s) \quad (6)$$

End

Quando for concluída a constução de δ_{i+1}, acima, resta somente expandir a interpretação ι_i, definida no nível anterior, para complementar a tradução com os novos símbolos, associando naturalmente cada símbolo em σ_{i+1}, ao seu correspondente em δ_{i+1}. Feito isto, esta conluída a implementação neste nível, como na figura IV.1.4.

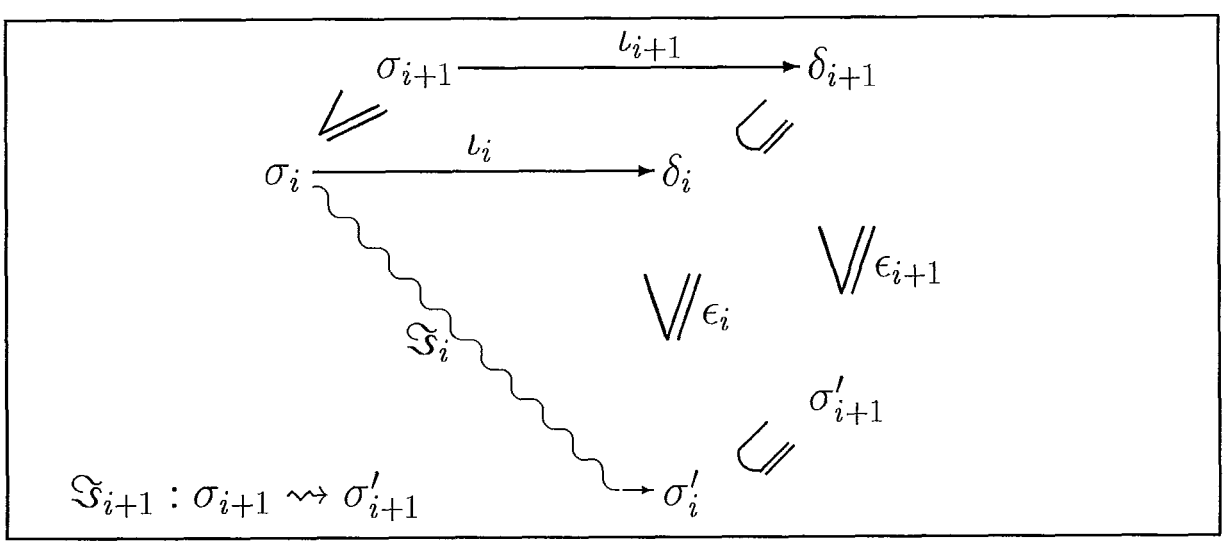


Figura IV.1.4: Etapa de Recombinação

Isto condiz com o Teorema da Construtibilidade, pois : "Se um TAD, especificado em σ_i, é implementado sobre um outro TAD, especificado em σ'_i, resultando em um módulo de implementação especificado por δ_i, então é sempre possível implementar uma extensão conservativa σ_{i+1} de σ_i, em uma extensão consistente σ'_{i+1} de σ'_i, obtendo uma implementação declarativa δ_{i+1}, também consistente".

Proposição IV.1.2 *Se existem seqüências* $\sigma_1,\sigma_2,\ldots,\sigma_n$ *e* $\sigma'_1,\sigma'_2,\ldots,\sigma'_n$ *de teorias consistentes tais que:*

i) $\sigma_1 \leq \sigma_2 \leq \ldots \leq \sigma_n$

ii) $\sigma'_1 \subseteq \sigma'_2 \subseteq \ldots \subseteq \sigma'_n$

e se $\sigma_1 \rightsquigarrow \sigma'_1$ *então* $\sigma_n \rightsquigarrow \sigma'_n$.

Prova : A demonstração é claramente indutiva. Primeiro, tomando como base da indução a hipótese : $\sigma_1 \rightsquigarrow \sigma'_1$, resta provar o passo da indução:

$$\sigma_i \rightsquigarrow \sigma'_i \;\Rightarrow\; \sigma_{i+1} \rightsquigarrow \sigma'_{i+1}$$

Ora isto é exatamente o Corolário III.1.1 do Teorema da Construtibilidade, e portanto já provado. Logo, se $\sigma_1 \rightsquigarrow \sigma'_1$ então, indutivamente, $\sigma_n \rightsquigarrow \sigma'_n$. Conforme a figura IV.1.5. □

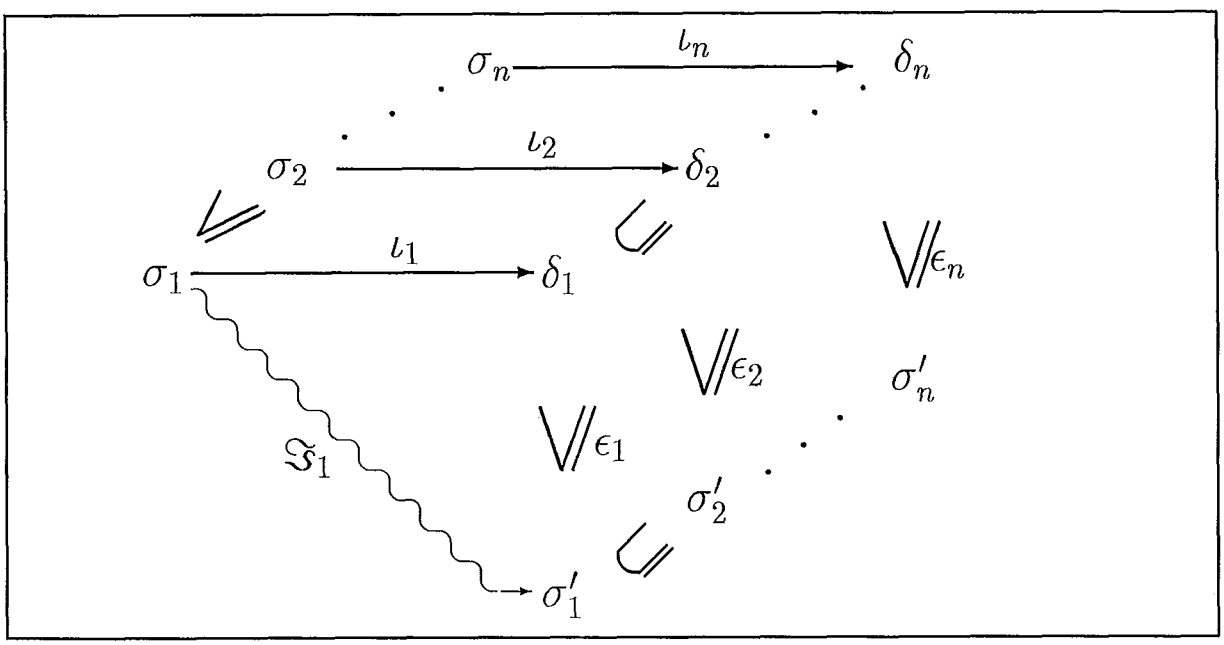


Figura IV.1.5: Iteração dos Passos de Implementação

O Teorema da Construtibilidade não se limitaria somente a comprovar a existência de uma teoria, especificada por δ_{i+1}, para intermediar o próximo passo canônico. Aponta um procedimento capaz de compô-la com a implementação declarativa subjacente, δ_i, a partir das especificações para σ_{i+1} e σ'_{i+1}, levando em consideração o caráter construtivo apresentado por sua demonstração. Determina igualmente, uma interpretação de σ_{i+1} em σ'_{i+1}. Embora, esta interpretação não permite associar novos símbolos, introduzidos em σ'_{i+1}, aos já presentes na implementação declarativa subjacente, δ_i, nem aos adicionados à especificação concreta, σ_{i+1} (veja a seção III.1.4).

Sendo pragmático, esta limitação raramente é desejável, pois em geral, ao acrescentar novos símbolos em σ'_{i+1}, já se supõe refinar símbolos novos, pertencentes a linguagem abstrata.

Presumivelmente isto era esperado, o teorema não deveria, de qualquer forma, certificar interferências criativas, onde nenhum critério disciplinante foi imposto. Porém,

está evidente, que a proposta na demonstração do teorema não precisa ser a única implementação legítima (proposição III.1.2). Talvez seja preciso recorrer à ação intelectual, associando mais convenientemente novos símbolos abstratos, com suas representações em termos de símbolos concretos.[5] Devido cuidado precisa, entretanto, ser tomado de forma que não seja perdida a consistência de δ_{i+1}, porque esta é a finalidade da construção, estabelecida na demonstração do teorema.

Apesar de aparentar ser severa, esta restrição pode ser contornada sem maiores complicações. O problema todo está em associar a tradução de símbolos da linguagem mais abstrata a símbolos pertencentes a linguagem primitiva, todavia isto não é absolutamente imperativo. A escolha de uma linguagem auxiliar, estendendo a linguagem primitiva, para definir o refinamento de cada símbolo abstrato, conforme a orientação sugerida na seção II.3, é suficiente para solucionar o problema. A vantagem desta linguagem auxiliar é separar devidamente cada nível lingüístico, transferindo a responsabilidade da construção de refinamentos da extensão $\sigma'_i \subseteq \sigma'_{i+1}$ para $\sigma'_{i+1} \leq \delta_{i+1}$, como há sido feito até aqui (e.g. no passo canônico $Dispatcher \rightsquigarrow PriorityQueue[Process]$).

Resumindo, o teorema sugere sempre uma forma para compor a nova implementação declarativa. Caso não seja a mais adequada, cabe ao projetista, com a melhor compreensão, encontrar uma outra mais apropriada. Porém, em qualquer caso a construção dada pelo teorema provê os requisitos mínimos necessários a uma implementação que preserve as propriedades representadas no passo canônico subjacente, consoante com III.1.2.

Caso fosse conveniente realizar a implementação sobre um TAD existente, já especificado, por exemplo um TAD público ou suportado pela linguagem de programação, nenhuma dificuldade adicional seria encontrada. Ao contrário, é conseguido de todo modo, um maior controle da complexidade e, uma maior segurança, podendo passo a passo verificar a adequação de decisões determinadas (veja a próxima seção).

Dando continuidade a implementação do TAD *Dispatcher*, agora é o TAD $PriorityQueue[Process]$ quem deve ser implementada em termos ainda mais primitivos. Para representá-la, pode ser empregado o TAD $Array[Indice, Process]$, preferencialmente porque este TAD é suportado pela maioria das linguagens de programação. Por con-

[5]Em [Sav90], está apresentada um proposta interessante de um procedimento, para associar a operações abstratas, implementação em termo de operações da linguagem concreta. Desde que sejam escolhidas, *a priori*, representações adequadas para os sortes. O procedimento basea-se na unificação, utilizando o fechamento por *tableaux*.

seguinte, $Array[Indice, Process]$ está *a priori* inteiramente especificado.

Estendendo $Array[Indice, Process]$, é definido uma representação para uma fila de prioridade na forma de um *Heap,* conforme,

ADT $Heap[Process]$

Syntax

Import $Array[Indice, Process], Process$

Function $create^* : \longmapsto Heap;$

$put^* : Heap \times Process \longmapsto Heap;$

$get^* : Heap \longmapsto Heap;$

$high^* : Heap \longmapsto Process$

Predicate $empty^*(Heap);$

HidenSymbols $List;$

$root : List \longmapsto Indice;$

$last : List \longmapsto Indice;$

$array : List \longmapsto Array;$

$heap : List \longmapsto Heap;$

Axiomatization

$$(\forall r : List)(\exists a : Array)(\exists i, i' : Indice)$$
$$\{root(r) \stackrel{I}{=} i \ \wedge \ last(r) \stackrel{I}{=} i' \ \wedge \ array(r) \stackrel{A}{=} a\} \tag{1}$$
$$(\forall r, r' : List)\neg r \stackrel{L}{=} r' \Rightarrow$$
$$\neg\{root(r) \stackrel{I}{=} root(r') \ \wedge \ last(r) \stackrel{I}{=} last(r') \ \wedge \ list(r) = list(r')\} \tag{2}$$
$$(\forall r, r' : List)heap(r) \stackrel{L}{=} heap(r') \Leftrightarrow (\forall i : Indice)inside(i, root(r), last(r))$$
$$\Rightarrow value(array(r), i) \stackrel{P}{=} value(array(r'), corresp(i, last(r), last(r'))) \tag{3}$$
$$(\forall h : Heap)(\exists r : List)heap(r) \stackrel{H}{=} h \tag{4}$$
$$(\forall a : Array)(\forall p : Process)(\forall i, j : Indice)$$
$$\{2i \stackrel{I}{=} j \ \wedge \ x \rhd value(a, 2i)\}$$
$$\Rightarrow insert(a, p, i, j) \stackrel{A}{=} attrib(insert(a, p, 2i, j), i, value(a, 2i)) \tag{5}$$
$$(\forall a : Array)(\forall i : Indice)$$
$$value(a, 2i) \rhd value(a, 2i + 1) \Leftrightarrow highleftchild(a, i) \tag{6}$$
$$(\forall a : Array)(\forall p : Process)(\forall i, j : Indice)$$
$$\{2i < j \ \wedge \ \neg highleftchid(a, i) \ \wedge \ x \rhd value(a, 2i)\}$$
$$\Rightarrow insert(a, p, i, j) \stackrel{A}{=} attrib(Insert(a, p, 2i, j), i, value(a, 2i)) \tag{7}$$

$$(\forall a : Array)(\forall p : Process)(\forall i, j : Indice)$$
$$\{2i < j \ \wedge \ highleftchid(a, i) \ \wedge \ x \rhd value(a, 2i)\}$$
$$\Rightarrow insert(a, p, i, j) \stackrel{A}{=} attrib(insert(a, p, 2i + 1, j), i, value(a, 2i + 1)) \quad (8)$$

$$(\forall a : Array)(\forall p : Process)(\forall i, j : Indice)$$
$$\{2i < j \ \wedge \ \neg x \rhd value(a, 2i) \ \wedge \ \neg x \rhd value(a, 2i + 1)\}$$
$$\Rightarrow insert(a, p, i, j) \stackrel{A}{=} attrib(a, i, x) \quad (9)$$

$$(\forall a : Array)(\forall p : Process)(\forall i, j : Indice)$$
$$\neg j < 2i \ \wedge \ \neg j \stackrel{I}{=} 2i \Rightarrow insert(a, p, i, j) \stackrel{A}{=} attrib(a, i, x) \quad (10)$$

$$(\forall h : Heap) create^* \stackrel{H}{=} h \Leftrightarrow (\exists r : List)$$
$$heap(r) \stackrel{H}{=} h \ \wedge \ pred(root(r)) \stackrel{I}{=} last(r) \ \wedge \ array(r) \stackrel{A}{=} inicialize \quad (11)$$

$$(\forall h, h' : Heap)(\forall p : Process) put^*(h, p) \stackrel{H}{=} h' \Rightarrow$$
$$(\exists r, r' : List)\{heap(r) \stackrel{H}{=} h \ \wedge \ heap(r') \stackrel{H}{=} h'\}$$
$$root(r') \stackrel{I}{=} pred(root(r)) \ \wedge \ last(r') \stackrel{I}{=} last(r)$$
$$\wedge \ array(r') \stackrel{A}{=} insert(array(r), p, pred(root(r)), last(r)) \quad (12)$$

$$(\forall h, h' : Heap)(\forall p : Process) get^*(h, p) \stackrel{H}{=} h' \Rightarrow \neg empty^*(h)$$
$$(\exists r, r' : List)\{heap(r) \stackrel{H}{=} h \ \wedge \ heap(r') \stackrel{H}{=} h'\} \Rightarrow \{array(r') \stackrel{A}{=}$$
$$insert(array(r), value(array(r), last(r)), root(r), pred(last(r)))\} \quad (13)$$

$$(\forall h : Heap)(\forall p : Process) high^*(h) \stackrel{H}{=} p \Rightarrow$$
$$(\exists r : List) heap(r) \stackrel{H}{=} h \ \wedge \ p \stackrel{P}{=} value(array(r), root(r)) \quad (14)$$

$$(\forall h : Heap)$$
$$empty^*(h) \Leftrightarrow (\exists r : List) heap(r) \stackrel{H}{=} h \ \wedge \ last(r) \stackrel{P}{=} pred(root(r)) \quad (15)$$

End

Na realidade, tudo o que é necessário fazer é, em cada nível de simplificação, escolher um subconjunto de operações e sortes em Σ', para suportar a implementação deste nível. Estas decisões inclusive, podem basear-se em critérios semelhantes aos adotados para particionar a especificação Σ, durante a simplificação.

Proposição IV.1.3 *Sejam as teorias consistentes Σ e Σ'. Se existe uma seqüência de teorias, $\sigma_1, \sigma_2, \ldots, \sigma_n$, consistentes tais que: $\sigma_1 \leq \sigma_2 \leq \ldots \leq \sigma_n$ com $\sigma_n \equiv \Sigma$ e se $\sigma_1 \rightsquigarrow \Sigma'$ então $\Sigma \rightsquigarrow \Sigma'$.*

Prova : Pela Proposição IV.1.2, meramente tomando $\sigma'_1 \equiv \Sigma', \sigma'_2 \equiv \Sigma', \ldots, \sigma'_n \equiv \Sigma'$ conclue-se diretamente que $\sigma_n \rightsquigarrow \sigma'_n$, logo $\Sigma \rightsquigarrow \Sigma'$, conforme IV.1.6. □

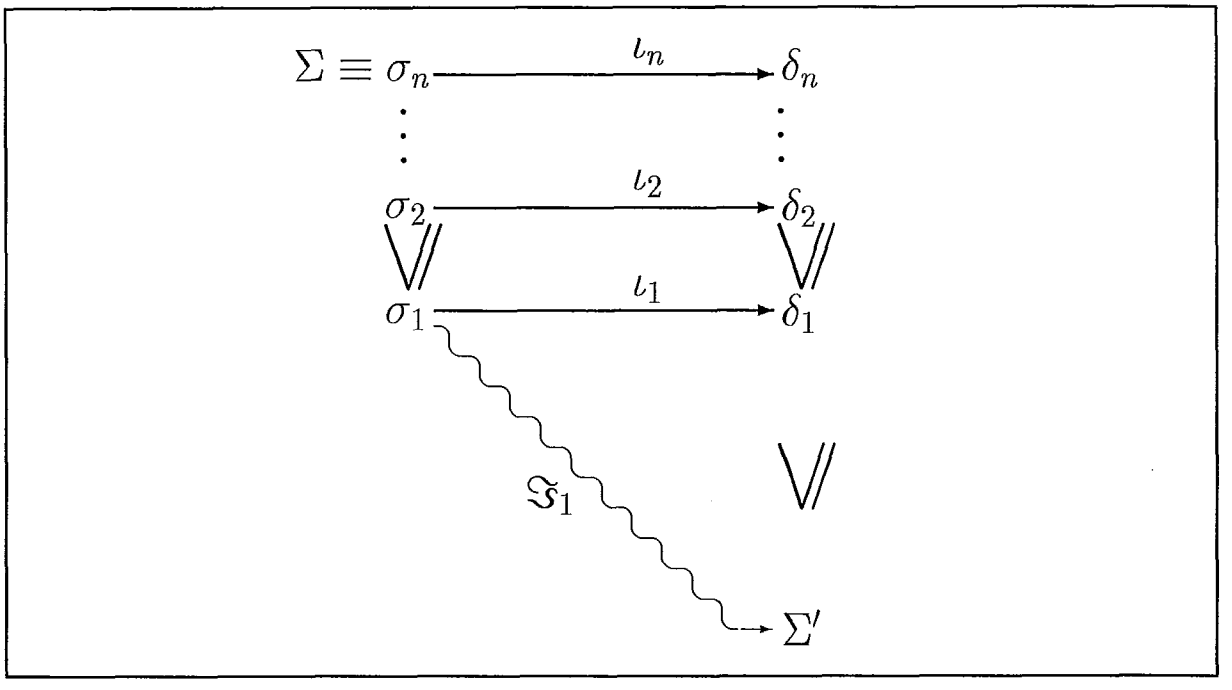


Figura IV.1.6: Desenvolvimento Construtivo sobre um TAD já existente

IV.1.2 Decisões de Refinamento

O processo de refinamento é de algum modo simétrico ao processo de abstração. Enquanto abstrair, na construção de uma especificação, significa ignorar detalhes não necessários, buscando generalizar uma representação para um conceito, refinar, ao contrário, quer dizer exatamente recolocar propriedades, expressando-as como sentenças sobre símbolos de uma linguagem destino, contanto que esteja preservado o significado semântico inicialmente abstraído.

Em um dado passo canônico, $\Im_{i+1}$, a finalidade de uma interpretação, ι_{i+1}, é procurar uma tradução, associando cada símbolo na linguagem abstrata a um correlato, na mesma categoria sintática. Ao passo que, na formação da extensão conservativa ϵ_{i+1}, é escolhido e estruturado um repertório de símbolos primitivos representando o refinamento de sortes e operações abstratas.

Esta seção trata sobre a composição ou combinação de símbolos primitivos a fim de formar uma imagem para interpretação ι_{i+1}, na continuidade do processo de recombinação. Principalmente no que se relaciona a sistematização e controle de complexidade,

promovidos por meio de uma estratégia de caráter construtivo.

A especificação mais concreta σ'_{i+1} deve ser estendida em dois sentidos:

i) Definindo a representação para os dados, indicando como os sortes concretos são combinados e estruturados para representar os sortes mais abstratos;

ii) Implementando as funções e predicados da linguagem abstrata em termos das operações tornadas disponíveis na especificação concreta, presumivelmente fazendo uso de como foram definidas as representações para os dados.

Nas duas seções subsequentes, analisa-se com mais detalhes ambos os sentidos.

IV.1.2.1 Representação de Sortes

De acordo com a etapa de simplificação, um símbolo de sorte pode ser eliminado, quando forem retiradas todas as operações que o referenciam, quer como argumento, quer como resultado. Por conseguinte, somente é preciso decidir sobre sua representação, ao implementar uma subestrutura em que, pela primeira vez, estejam operações que o manipulam.

Desta forma, concomitantemente a especificação de novas operações, cada σ'_{i+1} deve prover sortes suficientes para representar aqueles sortes abstratos a serem implementados neste nível. Por isso, enquanto a linguagem de σ'_{i+1} é projetada, estendendo a especificação de σ'_i, alguns novos sortes são provavelmente escolhidos ampliando, pois, o repertório concreto.

Finalmente quando é construida a extensão conservativa ϵ_{i+1}, símbolos de sorte em σ'_{i+1}, são combinados, estruturando sortes para representar aqueles em σ_{i+1}, compatibilizando, assim, as linguagens de σ_{i+1} e σ'_{i+1}, e permitindo que as operações que os manipulam sejam mais convenientemente implementadas.

Como definir a representação para um novo sorte abstrato, a partir do repertório de σ'_{i+1} ? Simplesmente por intermédio de extensões por definição de sortes, apresentadas no apêndice B. A interpretação segue traduzindo propriamente cada sorte abstrato, sobre sua representação correspondente, assim construida.

Ainda em relação ao programa abstrato *Sched*, persiste indefinida a especificação e a implementação para o TAD *Buffer*. Porém, agora é oportuno cuidar disto, para ilustrar a definição de refinamentos:

ADT *Buffer*[*Process*]

Syntax

Import *Process*

Function $create : \longmapsto Buffer;$

$recieve : Buffer \times Process \longmapsto Buffer;$

$send : Buffer \longmapsto Buffer;$

$first : Buffer \longmapsto Process$

Predicate $empty(Buffer);$

$full(Buffer);$

Axiomatization

$(\forall b : Buffer)$

$b \stackrel{B}{=} create \ \vee \ (\exists b' : Buffer)(\exists p : Process) b \stackrel{B}{=} recieve(b', p)$ (1)

$empty(create)$ (2)

$(\forall b : Buffer) empty(b) \Rightarrow \neg full(b)$ (3)

$(\forall b : Buffer)(\forall p : Process) full(b) \Leftrightarrow recieve(b, p) \stackrel{B}{=} b$ (4)

$(\forall b : Buffer)(\forall P : Process) \neg empty(recieve(b, p))$ (5)

$(\forall b : Buffer)(\exists b' : Buffer) send(b) \stackrel{B}{=} b' \Leftrightarrow$

$(\exists p : Process) b \stackrel{B}{=} recieve(b', p) \ \wedge \ \neg b \stackrel{B}{=} b'$ (6)

$(\forall b : Buffer)(\forall p : Process)$

$\neg(empty(b) \ \vee \ full(b)) \Rightarrow send(recieve(b, p)) \stackrel{B}{=} receive(send(b, e))$ (7)

$(\forall b : Buffer)(\forall p : Process) empty(b) \Rightarrow first(receive(b, p)) \stackrel{P}{=} p$ (8)

$(\forall b : Buffer)(\forall p : Process)$

$\neg empty(b) \Rightarrow first(recieve(b, p)) \stackrel{P}{=} first(b)$ (9)

End

Simplificando sua especifcação, considere somente a responsabilidade de representar o sorte *Buffer*. Para tanto, será utilizado o TAD instaciado $Array[CIndice, Processo]$ (a substituição dos paramêtros *Indice* e *Elem* em $Array[Indice, Elem]$, apresentado na seção II.2, por *Process*). O novo TAD *CIndice* é obtido pela extensão de *Indice*, resultando em,

ADT *CIndice*

Syntax

Import *Indice*

Function $inc \quad : \quad CIndice \longmapsto CIndice;$

$offset \quad : \quad CIndice \times CIndice \times CIndice \longmapsto CIndice;$

Predicate $inbetween(CIndice, CIndice, CIndice);$

Axiomatization

$$(\forall i, j : CIndice)\neg i \stackrel{I}{=} maxpos \Rightarrow i < inc(i) \tag{1}$$

$$(\forall i : CIndice)\neg maxpos < inc(i) \tag{2}$$

$$(\forall i : CIndice)inc(maxpos) < inc(i) \tag{3}$$

$$(\forall i, j, k : CIndice)inbetween(i, j, k) \Leftrightarrow$$
$$\{inside(i, inc(maxpos), maxpos) \ \wedge \ [\neg k < j \Rightarrow (\neg i < j \ \wedge \ \neg k < i)] \ \wedge$$
$$[k < j \Rightarrow (\neg i < j \ \vee \ \neg k < i)]\} \tag{4}$$

$$(\forall i, j, k : CIndice)(\exists c : CIndice)c \stackrel{I}{=} offset(i, j, k) \Leftrightarrow$$
$$\{[inc(j) \stackrel{I}{=} i \ \wedge \ inc(k) = c] \ \vee \ [c \stackrel{I}{=} offset(i, inc(j), inc(k))]\} \tag{5}$$

End

A especificação de *Process* permanece inalterado tanto em *Buffer*, quanto em $Array[CIndice, Process]$. É iniciada a construção de $Buffer^*[Process]$, estendendo a especificação para $Array[CIndice, Process]$, definindo como representação o novo sorte $Buffer^*$ expressa através dos axiomas:

$$(\forall c : CList)(\exists a : Array)(\exists i, i' : CIndice)$$
$$head(c) \stackrel{I}{=} i \ \wedge \ rear(c) \stackrel{I}{=} i' \ \wedge \ array(c) \stackrel{A}{=} a$$
$$(\forall c, c' : CList)\neg c \stackrel{C}{=} c' \Rightarrow$$
$$\neg\{head(c) \stackrel{I}{=} head(c') \wedge rear(c) \stackrel{I}{=} rear(c') \wedge array(c) \stackrel{A}{=} array(c')\}$$
$$(\forall c, c' : CList)buffer(p) \stackrel{C}{=} buffer(p') \Leftrightarrow$$
$$(\forall i : CIndice)inbetween(i, head(c), rear(c)) \Rightarrow$$
$$value(array(c), i) \stackrel{P}{=} value(array(c'), offset(i, head(c), head(c')))$$
$$(\forall b : Buffer^*)(\exists c : CList)buffer(c) \stackrel{B}{=} b$$

Inicialmente um sorte *CList* é definido como o produto cartesiano (A.3.10) com as funções *head*, *rear* e *array* como projeções. Em seguida, uma relação de equivalência é refinada a partir da igualdade entre *CList* por intermédio da função *buffer*, definindo assim, o quociente (A.3.12). Com isto, as instâncias de *Buffer* são identificadas independentemente de sua representação como *CList*.

IV.1.2.2 Refinamento de Operações

Um símbolo de operação é por sua vez implementado, quando for aplicado o passo canônico ao nível de simplificação, no qual este símbolo foi eliminado da linguagem. Neste nível, só encontram-se disponíveis funções e predicados que foram julgadas convenientes à linguagem de σ'_{i+1}, visando especificamente esta etapa da implementação, além dos símbolos já especificados em níveis subjacentes. Portanto, as novas operações referidas em um determindo ponto, serão necessariamente definidas em termos deste conjunto mais restrito de símbolos concretos, ou de operações já implementadas.

Novamente, a extensão ϵ_{i+1} tem o papel de especificar funções e predicados, a partir da linguagem de σ'_{i+1}, a fim de representar a decomposição das operações em σ_{i+1}.

As operações enumeradas no repertório do TAD *Buffer* precisam ser gradativamente refinadas. O resultado completo da implementação declarativa $Buffer^*$, estabelecidos as representações adequadas para cada operação em *Buffer*:

ADT $Buffer^*[Process]$

Syntax

Import $Array[CIndice, Process], Process$

Function $create^* : \longmapsto Buffer^*;$

$recieve^* : Buffer^* \times Process \longmapsto Buffer^*;$

$send^* : Buffer^* \longmapsto Buffer^*;$

$first^* : Buffer^* \longmapsto Process$

Predicate $empty^*(Buffer^*);$

$full^*(Buffer^*);$

HidenSymbols $CList;$

$head : CList \longmapsto CIndice;$

$rear : CList \longmapsto CIndice;$

$array : CList \longmapsto Array;$

$buffer : CList \longmapsto Buffer^*;$

Axiomatization

$$(\forall c : CList)(\exists a : Array)(\exists i, i' : CIndice)$$
$$head(c) \stackrel{I}{=} i \ \wedge \ rear(c) \stackrel{I}{=} i' \ \wedge \ array(c) \stackrel{A}{=} a \qquad (1)$$
$$(\forall c, c' : CList) \neg c \stackrel{C}{=} c' \Rightarrow$$
$$\neg\{head(c) \stackrel{I}{=} head(c') \wedge rear(c) \stackrel{I}{=} rear(c') \wedge array(c) \stackrel{A}{=} array(c')\} \qquad (2)$$

$$(\forall c, c' : CList)buffer(p) \stackrel{C}{=} buffer(p') \Leftrightarrow$$
$$(\forall i : CIndice)inbetween(i, head(c), rear(c)) \Rightarrow$$
$$value(array(c), i) \stackrel{P}{=} value(array(c'), offset(i, head(c), head(c'))) \quad (3)$$
$$(\forall b : Buffer^*)(\exists c : CList)buffer(c) \stackrel{B}{=} b \quad (4)$$
$$(\forall b : Buffer^*)b \stackrel{B}{=} create^* \Leftrightarrow (\exists c : CList)$$
$$buffer(c) \stackrel{B}{=} b \ \wedge \ rear(c) \stackrel{I}{=} head(c) \ \wedge \ array(c) \stackrel{A}{=} inicialize \quad (5)$$
$$(\forall b : Buffer^*)$$
$$empty^*(b) \Leftrightarrow (\exists c : CList)buffer(c) \stackrel{B}{=} b \ \wedge \ rear(c) \stackrel{I}{=} head(c) \quad (6)$$
$$(\forall b : Buffer^*)$$
$$full^*(b) \Leftrightarrow (\exists c : CList)buffer(c) \stackrel{B}{=} b \ \wedge \ rear(c) \stackrel{S}{=} inc(head(c)) \quad (7)$$
$$(\forall b : Buffer^*)(\forall p : Process)b \stackrel{B}{=} recieve^*(b, p) \Leftrightarrow full(b) \quad (8)$$
$$(\forall b : Buffer^*)(\forall p : Process)(\exists b' : Buffer^*)b' \stackrel{B}{=} recieve^*(b, p) \Leftrightarrow$$
$$\neg full(b) \ \wedge \ (\exists c, c' : CList)\{buffer(c) \stackrel{B}{=} b \ \wedge \ buffer(c') \stackrel{B}{=} b'\} \Rightarrow$$
$$\{head(c') \stackrel{I}{=} head(c) \ \wedge \ inc(rear(c')) \stackrel{I}{=} rear(c)$$
$$\wedge \ array(c') \stackrel{A}{=} attrib(array(c), inc(rear(c)), p)\} \quad (9)$$
$$(\forall b : Buffer^*)(\exists b' : Buffer^*)b' \stackrel{B}{=} send^*(b) \Leftrightarrow \neg empty(b) \wedge$$
$$(\exists c, c' : CList)\{buffer(c) \stackrel{B}{=} b \ \wedge \ buffer(c') \stackrel{B}{=} b'\} \Rightarrow$$
$$inc(head(c')) \stackrel{I}{=} head(c) \wedge rear(c') \stackrel{I}{=} rear(c) \wedge array(c') \stackrel{A}{=} array(c) \quad (10)$$
$$(\forall s : Buffer^*)(\forall p : Process)p \stackrel{P}{=} first^*(s) \Leftrightarrow$$
$$(\exists c : CList)buffer(c) \stackrel{B}{=} b \Rightarrow value(array(c), head(c)) \stackrel{P}{=} p \quad (11)$$

End

Um princípio coerente costuma direcionar estas decisões de refinamento, no tocante a implementação de operações, tentar o desacoplamento da especificação produzida. Na realidade, isto significa expressar cada operação independentemente, e por conseqüência separar o foco de interesse, individualizando, a cada momento, o refinamento de uma única operação.

Isto nem sempre é um objetivo inteiramente factível, mas na medida das possibilidades é sempre pretendido. Daí surgem algumas vantagens como:

- limitar a abrangência dos efeitos causados por erros, e facilitar pois, a sua posterior correção;

- restringir as alterações demandadas por mudanças nos requisitos;
- além do que um maior grau de modularização só tende a favorecer a continuidade do desenvolvimento.

Ao proceder passo a passo a construção da implementação declarativa solicitada, exatamente quando for estabelecida uma extensão conservativa ϵ_{i+1} de σ'_{i+1}, é que se precisa cuidar do desacoplamento. Consegue-se isto, se os axiomas, que definem o significado de predicados e funções na linguagem de δ_{i+1}, contidos na imagem da interpretação ι_{i+1}, estejam descritos utilizando, exclusivamente, símbolos pertencentes a linguagem de σ'_{i+1}, como foi realizado no exemplo anterior.

Embora não seja mandatório desacoplar a implementação declarativa, caso isto seja requisitado, a própria seqüência de simplificações fornece boas indicações para orientar o desacoplamento, bastando ater-se à seqüência de simplificações realizadas, pois estas retringem, ao mínimo, o número de operações a serem implementadas em cada nível.

Símbolos auxiliares, como *CList*, *head*, *rear*, podem melhorar sensivelmente a clareza e legibilidade das sentenças na axiomatização, tanto definindo símbolos de funções para substituir termos mais elaborados, quanto introduzindo símbolos de predicado para representar subfórmulas. A adequação destas abreviações esta a critério do projetista, sua experiência dita o que e como abreviar.

IV.1.2.3 Decisões *Ex Post* de Refinamento

A extensão que enriquece o nível lingüístico mais primitivo, por não ser necessariamente conservativa, influencia o processo de refinamento, dando primeiro margem para rever a especificação concreta, e introduzindo, em forma de novos axiomas, decisões de refinamento, antes esquecidas.

Por exemplo, em determinado ponto da implementação, caso seja constatada a falta de alguma propriedade, que somente agora foi entendida como necessária, meramente adicioná-la à especificação concreta contorna o problema, desde que não seja violada a consistência.

Obviamente, os erros não podem ser simplesmente recuperados, nem decisões futuramente eliminadas da implementação, dado o caráter monotônico da linguagem de especificação, ou seja, a única razão desta impossibilidade teórica é o caráter monotônico

da Lógica de Primeira Ordem.[6] Mesmo assim, já se alivia sensivelmente, a necessidade de tentar continuamente reformular a implementação por causa de omissões.

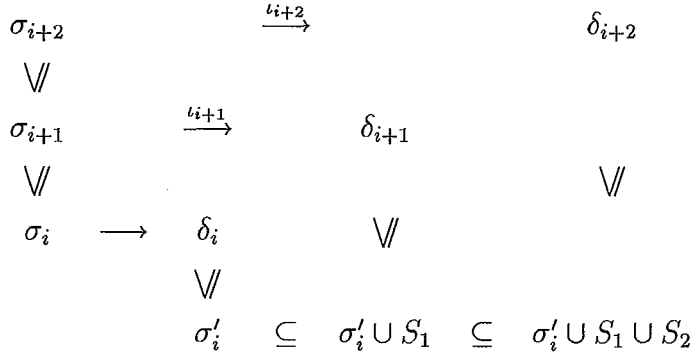
Na seção IV.1.1.3 o TAD *Dispatcher*[*Process*] havia sido implementado em termos do TAD *PriorityQueue*, entretanto a representação escolhida para a função *next* através de *high* não esclarece muito como realmente deve ser escolhido um processo para impressão. Mesmo após o passo canônico no nível de simplificação onde foi definida *next*, é factível refiná-la melhor, estendendo não conservativamente *PriorityQueue* pela adição dos axiomas:

$$(\forall q : PriorityQueue)(\exists P : Process)$$
$$high(put(q,p)) \stackrel{P}{=} p \Leftrightarrow priority(high(q)) < priority(p)$$
$$(\forall q : PriorityQueue)(\exists P : Process)$$
$$high(put(q,p)) \stackrel{P}{=} high(p) \Leftrightarrow \neg priority(high(q)) < priority(p)$$

Neste caso, a implementação tornar-se-ia mais precisa, pois a escolha do próximo pedido de impressão a ser atendido é função de $priority(p)$, logo, no programa abstrato *Sched* é sempre impresso o pedido associado ao processo com maior prioridade.

Uma outra conseqüência importante, relaciona-se com a comutatividade na ordem em que as decisões de refinamento precisariam ser sequenciadas, para obter uma implementação final consistente. Relaxando a exigência de conservatividade, para a extensão que enriquece o nível concreto, evita-se condicionar, em contra-partida, uma ordenação bem mais rígida, para a escolha e a definição dos refinamentos necessários.

Proposição IV.1.4 (Comutatividade) *Considere as teorias consistentes* S_1 *e* S_2. *Sejam as composições de implementações abaixo:*



[6]Porém, alguns resultados tem encorajado a exploração de linguagens não monotônicas aplicadas à especificação formal. Maiores detalhes estão adiados até VI.

e também

$$\begin{array}{ccccc}
\sigma_{i+2} & & \xrightarrow{\eta_{i+2}} & & \bar{\delta}_{i+2} \\
\Vdots & & & & \\
\sigma_{i+1} & \xrightarrow{\eta_{i+1}} & \bar{\delta}_{i+1} & & \\
\Vdots & & & & \Vdots \\
\sigma_i & \longrightarrow & \delta_i & \Vdots & \\
 & & \Vdots & & \\
 & & \sigma'_i & \subseteq \sigma'_i \cup S_2 & \subseteq \sigma'_i \cup S_2 \cup S_1
\end{array}$$

com δ_{i+1},δ_{i+2},$\bar{\delta}_{i+1}$ *e* $\bar{\delta}_{i+2}$ *resultantes do Teorema da Construtibilidade, então :*

i) $\delta_{i+2} \xrightarrow{\kappa} \bar{\delta}_{i+2}$

ii) $\bar{\delta}_{i+2} \xrightarrow{\mu} \delta_{i+2}$

Prova : Pelo Teorema da Construtibilidade, $(\delta_i \cup \sigma'_i \cup S_1 \cup \iota_{i+1}[\sigma_{i+1}]) \equiv \delta_{i+1}$, igualmente,

$$(\delta_i \cup \sigma'_i \cup S_1 \cup S_2 \cup \iota_{i+2}[\sigma_{i+2}]) \equiv \delta_{i+2}$$

Da mesma maneira, $(\delta_i \cup \sigma'_i \cup S_1 \cup S_2 \cup \eta_{i+2}[\sigma_i]) \equiv \bar{\delta}_{i+2}$

A fim de provar :

i) $\delta_{i+2} \xrightarrow{\kappa} \bar{\delta}_{i+2}$

ii) $\bar{\delta}_{i+2} \xrightarrow{\mu} \delta_{i+2}$

É definida a seguinte função de tradução κ, sobre a linguagem de δ_{i+2} :

$$\kappa(\phi) = \begin{cases} \phi & \text{se e somente se } \phi \in \mathcal{L}(\delta_i \cup \sigma'_i \cup S_1 \cup S_2) \\ \alpha^{\eta_{i+2}} & \text{se e somente se } \phi \in \mathcal{L}(\iota_{i+2}[\sigma_{i+2}]) \\ & \text{para algum } \alpha \in \mathcal{L}(\sigma_{i+2}) \\ & \text{talque } \phi = \alpha^{\iota_{i+2}} \end{cases}$$

logo por construção de κ,

$$\text{Se } \delta_{i+2} \models \phi \text{ então } \bar{\delta}_{i+2} \models \phi^{\kappa}$$

para qualquer fórmula ϕ na linguagem de δ_{i+2}. Consequentemente, $\delta_{i+2} \xrightarrow{\kappa} \bar{\delta}_{i+2}$. De modo Análogo, seja a construção para μ sobre a linguagem de $\bar{\delta}_{i+2}$:

$$\mu(\phi) = \begin{cases} \phi & \text{se e somente se} \quad \phi \in \mathcal{L}(\delta_i \cup \sigma_i' \cup S_1 \cup S_2) \\ \alpha^{\iota_{i+2}} & \text{se e somente se} \quad \phi \in \mathcal{L}(\eta_{i+2}[\sigma_{i+2}]) \\ & \text{para algum} \quad \alpha \in \mathcal{L}(\sigma_{i+2}) \\ & \text{talque} \quad \phi = \alpha^{\eta_{i+2}} \end{cases}$$

Da mesma forma, para qualquer fórmula ϕ na linguagem de $\bar{\delta}_{i+2}$:

$$\text{Se} \quad \bar{\delta}_{i+2} \models \phi \quad \text{então} \quad \delta_{i+2} \models \phi^{\mu}$$

Diretamente, $\bar{\delta}_{i+2} \xrightarrow{\mu} \delta_{i+2}$ Concluindo, então δ_{i+1} e $\bar{\delta}_{i+1}$ são equivalentes, a menos de diferenças relacionadas a linguagem, isto é :

i) $\delta_{i+1} \longrightarrow \bar{\delta}_{i+1}$

ii) $\bar{\delta}_{i+1} \longrightarrow \delta_{i+1}$

□

Já que não se exige, por isso, tomar inicialmente todas as decisões sobre a representação dos sortes e operações, consegue-se aliviar a necessidade de uma especificação definitiva *a priori* para o TAD que suportará a implementação. Consequentemente, são adiadas, ao máximo, decisões sobre o mecanismo lingüístico, especificando então, só e somente aquilo que realmente se faz necessário à implementação, dentro dos princípios de *Especificação Liberal.*

Mas aqui não se advoga a total liberalidade ou permissibilidade, onde fica permitido caminhar às cegas, tentando acertar no futuro o que foi laconicamente especificado. Muito ao contrário, acredita-se ser a compreensão completa do objeto da implementação, imprescindível para o seu bom termo. Todavia, reconhece-se as limitações às quais estamos sujeitos, e a não trivialidade da tarefa, e portanto, esta abordagem busca somente meios mais flexíveis que permitam aproveitar melhor o esforço intelectual empregado, e postergar a considerações futuras, sob a luz de melhores informações, decisões relativas a pontos ambiguos da implementação

IV.1.3 Adição não Conservativa de Detalhes

Durante a construção de um implementação, costuma ser frequente sentir que algum detalhe relevante ao desenvolvimento, passou despercebido à concepção inicial, e só posteriormente tornou-se evidente a sua necessidade. Ocorre usualmente isto, em fases iniciais do desenvolvimento, quando não há, em geral visão abrangente sobre todos os detalhes pertinentes ao conceito sendo modelado. A maturidade sobre a representação de um objeto e suas operações, pode advir durante o processo de desenvolvimento, à medida que o conceito é implementado em termos mais primitivos.

Quando, durante a implementação for sentida a falta de propriedades, que se poderiam revelar úteis aos propósitos de refinamento, talvez, seja necessário adicionar detalhes, e assim conseguir compatibilizar as descrições nos níveis abstrato e concreto, facilitando a construção de uma interpretação mais satisfatória.

Da mesma forma, algumas características desejáveis ao produto final da implementação, seriam conseguidas através da introdução acertada de detalhes. Por exemplo, que levassem em conta critérios de modularização, proteção das representações, encapsulamento, e outros requisitos indispensáveis para um bom projeto; inclusive, questões de eficiência, bem como outros detalhes de natureza algorítmica, poderiam também ser tratadas por propriedades complementares.

Variadas circunstâncias costumam requerer a adição de alguns novos elementos à especificação. Quer para compatibilizar linguagens em níveis de detalhe distintos, quer, mais localmente, buscando uma melhor implementação para um determinado conceito. Como proceder então, para obter uma implementação declarativa mais adequada em relação ao mecanismo lingüístico, sendo construido?

Em princípio, não é aceitável descartar o trabalho previamente realizado, e para tanto, deseja-se acrescentar estes detalhes, ao passo de implementação, sem contudo, perder resultados adquiridos, reutilizando, pois, como base a implementação declarativa pré-existente.

De forma mais sutil, que comumente passa despercebida, a tradução de representações entre níveis lingüísticos distintos introduz implicitamente algumas propriedades extrínsecas. Devido mesmo a uma maior disponibilidade de teoremas relacionados com a linguagem mais primitiva. A implementação reifica propriedades a respeito da descrição de um conceito, não apenas intencionalmente através do processo de refinamento, mas

porque a própria tradução escolhida está expressa em um nível lingüístico mais rico em detalhes, particularidades extras desta linguagem são adquiridas.

A tradução da teoria mais abstrata, está contida entre os teoremas da implementação declarativa, mas esta continência é usualmente própria, pois, por construção, a implementação declarativa é uma extensão da especificação concreta, e portanto, adquire as propriedades pertinentes a esta última.

Isto faz parte do jogo, não há motivos para impedir o enriquecimento de uma representação. Ao contrário, é justamente este ganho de detalhes que possibilitará, em última análise, o desenvolvimento de uma representação final, computacionalmente tratável. Não haveria sentido em traduzir uma especificação, caso permanecesse sempre constante o conjunto de informações que podem ser obtidas a partir da mesma. Muito menos seria sensato iterar repetidos passos de implementação se nada de novo fosse realmente acrescentado.

A composição de $Dispatcher[Process] \rightsquigarrow PriorityQueue[Process]$ e $PriorityQueue[Process] \rightsquigarrow Array[Indice, Process]$, acaba por introduzir por exemplo, novas particularidades, ainda que indiretamente, a respeito de $Dispatcher[Process]$. Segundo a função *pred*, que é utilizada para calcular sempre o novo índice no *array* onde será armazenado qualquer novo processo recebido, a estrutura do *Heap* pode crescer indefinidamente. Por consequência, não há limites para o número de processos que podem ser recebidos. Esta decisão não estava entre as propriedades iniciais de $Dispatcher[Process]$, contudo, não havia nada que a contradissesse. Logo, a decisão sobre o "tamanho" de $Dispatcher[Process]$ foi omitida até um nível mais primitivo. Na implementação declarativa resultante da composição é possível inferir algo como:

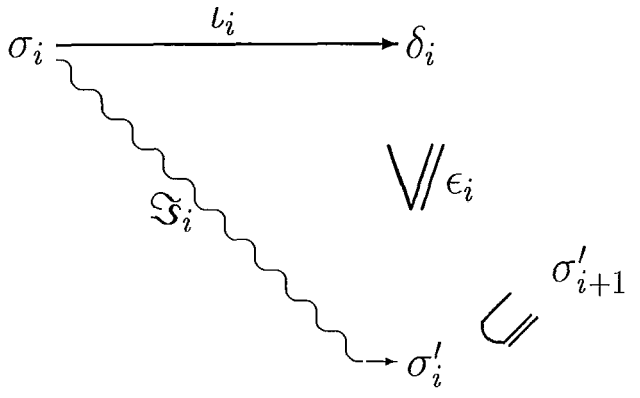
$$(\forall D : Dispatcher)(\forall p : Process)(\exists D' : Dispatcher)$$
$$D' \stackrel{D}{=} schedule(D, P) \ \wedge \ \neg D \stackrel{D}{=} D'$$

devidamente traduzida.

Portanto, favorecido com a capacidade de representar novas particularidades a respeito de conceitos já implementados, é possível utilizar o mecanismo de composição para acrescentar os detalhes necessários na forma de sentenças da linguagem mais primitiva, e implicitamente incorporá-los a nova implementação declarativa.

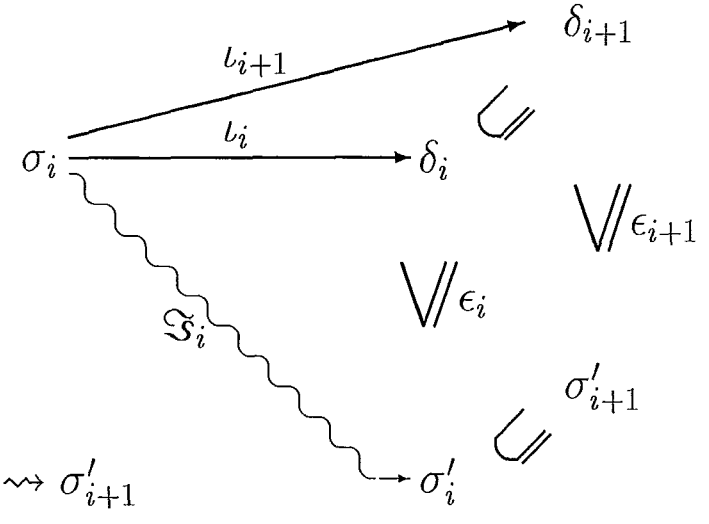
Supondo haver uma especificação σ_i, que está implementada em termos de uma outra σ_i' $(\Im i : \sigma_i \rightsquigarrow \sigma_i')$, a qual se pretende acrescer detalhes, seria então razoável

incorporar os detalhes, construindo uma nova teoria consistente, σ'_{i+1}, extensão de σ'_i. Examine o diagrama a seguir,



O Teorema da Construtibilidade permite então, compor uma nova implementação, onde estes detalhes são automaticamente incorporados a nova implementação declarativa δ_{i+1}, garantindo ao mesmo tempo, sua consistência.

Sempre existe uma interpretação ι_{i+1}, pois a adição de detalhes só tende a fortalecer a implementação declarativa. Assim, se antes era possível uma interpretação em δ_i, invariavelmente, será também sobre δ_{i+1}. Porque menos realizações são admitidas por δ_{i+1}, consequentemente a classe de modelos induzidos para σ_i, por ι_{i+1}, será no máximo igual a classe induzida por ι_i. Os novos axiomas introduzidos em σ'_{i+1}, somente fornecem informações extras a respeito da linguagem de σ'_i, e restringem suas possíveis realizações. Assim, $\langle \iota_{i+1}, \epsilon_{i+1} \rangle$ constituem-se em uma nova aplicação do passo canônico $(\Im_{i+1} : \sigma_{i+1} \rightsquigarrow \sigma'_{i+1})$, veja o diagrama,



Não haveria utilidade prática alguma, caso a adição de detalhes, ao nível primitivo, nada acrescentasse de novo, em relação à tradução de símbolos na linguagem abstrata. Ou seja, adicionar novas particularidades quer realmente dizer : expandir não conservativamente consequências relacionadas com a linguagem imagem da interpretação. Mas argumenta-se que isto pode ser efetivamente conseguido de modo implícito, por intermédio de uma simples extensão consistente do mecanismo lingüístico concreto. A próxima proposição (IV.1.5) formaliza a validade do argumento.

Proposição IV.1.5 *Seja a composição de implementações conforme abaixo:*

$$\begin{array}{ccccc} & & & & \delta_{i+1} \\ & \overset{\iota_{i+1}}{\nearrow} & & & \\ \sigma_i & \xrightarrow{\iota_i} & \delta_i & & \Vdash \\ & & \Vdash & & \\ & & \sigma'_i & \subseteq & \sigma'_{i+1} \end{array}$$

para toda sentença ϕ *tal que* $\phi \in \mathcal{L}(\iota_{i+1}[\sigma_i])$. *Se* $\delta_i \cup \sigma'_{i+1} \models \phi$ *e* $\delta_i \not\models \phi$ *então,*

$$\delta_{i+1} \models \phi \text{ e } \iota_{i+1}[\sigma_i] \not\models \phi$$

Prova :

1. Pelo Teorema da Construtibilidade, Se $\delta_i \cup \sigma'_{i+1} \models \phi$ então $\delta_{i+1} \models \phi$
2. então, por 1, se $\delta_i \cup \sigma'_{i+1} \models \phi$ e $\delta_i \not\models \phi$ então $\delta_{i+1} \models \phi$ e $\delta_i \not\models \phi$
3. por definição, $\iota_i[\sigma_i] \subseteq \delta_i$ e portanto, se $\delta_{i+1} \models \phi$ e $\delta_i \not\models \phi$ então $\delta_{i+1} \models \phi$ e $\iota_i[\sigma_i] \not\models \phi$
4. por construção, $\iota_i[\sigma_i] = \iota_{i+1}[\sigma_i]$

$$\delta_{i+1} \models \phi \text{ e } \iota_{i+1}[\sigma_i] \not\models \phi$$

□

Pela demonstração, fica comprovado que sentenças, na tradução da linguagem de σ_i, que antes não eram dedutíveis na implementação declarativa subjacente, podem estar entre os teoremas da implementação declarativa superposta pelo Teorema da Construtibilidade. Bastando para tanto, que certas propriedades sejam consistentemente

inseridas na especificação concreta, σ_{i+1}, mesmo que não estejam entre a tradução de sentenças válidas na teoria abstrata.

Decidir acrescentar alguns detalhes seguramente provoca um aumento no conjunto de consequências, deriváveis na implementação declarativa resultante e simultaneamente reduz o conjunto de possíveis realizações.

Representar novas particularidades concernentes ao TAD primitivo, corresponde, em contra-partida, escolher uma classe de modelos mais restrita. Isto significa, que nem toda realização para a implementação declarativa subjacente expande-se a um modelo, satisfazendo aos novos detalhes. À proporção que estes detalhes são adicionados, a implementação torna-se mais precisa e melhor definida, exatamente de acordo com o sentido natural do desenvolvimento. Somente realizações satisfazendo δ_{i+1}, portanto, apresentando os detalhes que se desejava adicionar, induzem agora realizações em σ_i, embora a mesma tradução dos axiomas em σ_i esteja presente na teoria de δ_{i+1}.

Ser capaz de representar novos detalhes referentes aos refinamentos já realizados, e de colocá-los na implementação, tem efeitos importantes nos princípios da implementação construtiva. Confirma-se a viabilidade de uma construção em paralelo da implementação e de uma especificação para suportá-la

A compreensão melhor dos resultados discutidos aqui, abre possibilidades realmente interessantes em relação ao desenvolvimento de programas. Por instância, a *Prototipagem Rápida* nada é senão meramente um desenvolvimento construtivo, onde continuamente faz-se necessário adicionar (não conservativamente) detalhes, percebidos enquanto a implementação elucida aspectos de seus requisitos. Apenas falta um mecanismo eficiente para retroceder etapas de implementação, quando determinados requisitos são refutados.

IV.1.4 Implementação e Criatividade

O caráter procedimental da descrição esboçada na seção IV.1.1, pode causar alguma confusão, entretanto, não há qualquer indicação de como devem ser construidas as extensões (tanto $\sigma_i \leq \sigma_{i+1}$, quanto $\sigma'_i \subseteq \sigma'_{i+1}$). Porque é precisamente nestes pontos onde se manifesta a interferência intelectual. Inegavelmente, realizar uma implementação é uma atividade intrinsecamente inteligente. Em essência, envolve escolhas e decisões criativas, e é justamente nessa subjetividade onde reside seu poder para enfrentar a complexidade.

A figura que se segue (IV.1.4), resume sinteticamente a relação entre cada passo

da construção de uma implementação subjacente e a apresentação do formalismo.

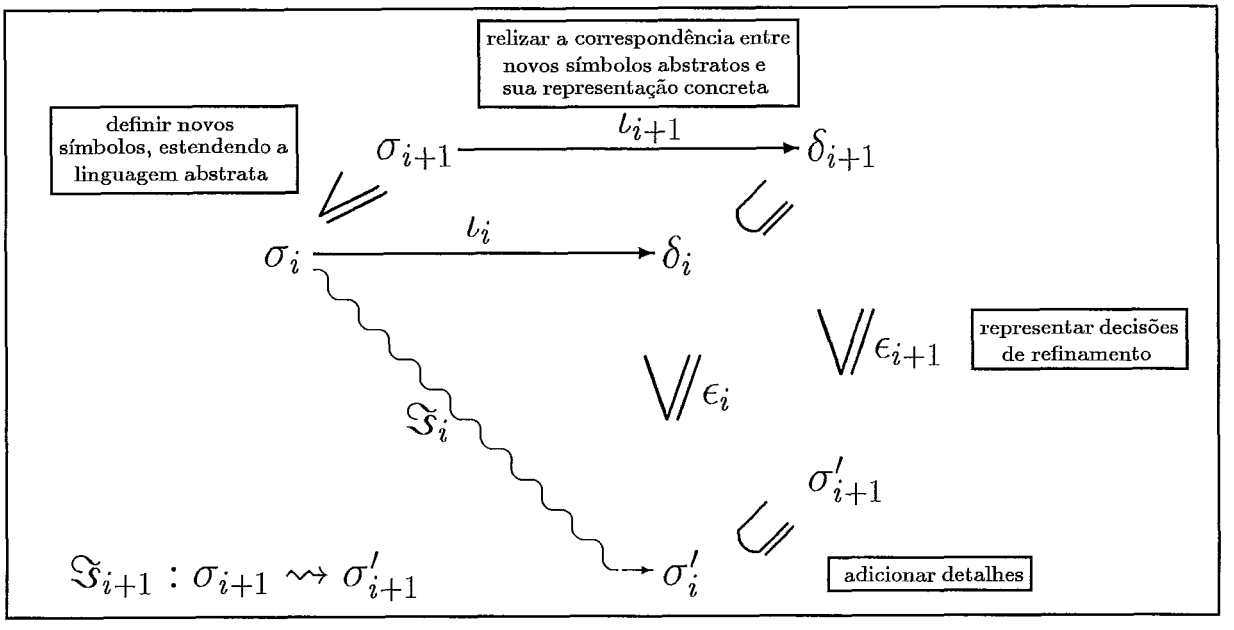


Figura IV.1.7: Implementação Construtiva

Conforme visto acima, em $\sigma_i \leq \sigma_{i+1}$ são definidos alguns novos elementos e suas propriedades na especificação abstrata, sob forma de símbolos e axiomas.

Por sua vez, em $\sigma'_i \subseteq \sigma'_{i+1}$ há espaço para reificar detalhes, e em $\sigma'_{i+1} \leq \delta_{i+1}$ são representadas as decisões de refinamento. Assim é construida uma especificação particularmente adequada para suportar a implementação da representação abstrata correspondente.

Qualquer tradução da linguagem abstrata em termos da linguagem primitiva pode ser empregada, desde que permita inferir como teorema alguma tradução dos axiomas especificando δ_{i+1}, do modo que foi definida pelo Teorema da Construtibilidade. E além disto, seja conservada a teoria de σ'_{i+1}.

Com esta proposta não se pretende eliminar, nem mesmo substituir, a criatividade necessária ao processo de implementação. Os resultados do teorema, unicamente confirmam a estratégia para compor mecanicamente o trabalho criativo realizado, ampliando o seu domínio de abrangência. Esta estratégia proposta aqui visa, somente, limitar a etapas particulares do processo de implementação toda a influência criativa necessária, aproveitando melhor seus recursos. Ao contrário, a disciplina e a sistematização introduzidas objetivam tão somente, manter sob controle a complexidade, e por outro lado, fornecer uma maior flexibilidade para manipular conceitos complicados.

IV.2 Verificação Construtiva de Programas

Ao invés de encarar a verificação como um etapa necessariamente posterior, é mais sensato trazê-la para cada subpasso resultante do processo de simplificação, antecipando a credibilidade nas decisões tomadas e nos refinamentos definidos. Conduzindo-a *pari-passo* com a construção de um passo canônico, as prováveis omissões e enganos tendem a transparecer mais cedo. Portanto, os erros podem ser corrigidos logo que detectados, impedindo sua propagação para os próximos passos superpostos, quando o esforço para reajustar a implementação será invariavelmente de maior custo.

Da mesma maneira, não pode ser ignorado que as informações produzidas durante a demonstração de corretude, em geral, são úteis para orientar o prosseguimento da implementação e elucidar pontos inicialmente obscuros.

O processo de verificação é realizado também construtivamente em cada passo, e a verificação concluida serve como um lema, inteiramente demonstrado, para certificar a implementação no passo canônico a ser superposto. A cada momento fica sobremodo limitado o escopo da demonstração, concentrando a atenção em um reduzido subconjunto de símbolos e axiomas.

Deste modo, uma demonstração de corretude mais elaborada seria vantajosamente substituida por uma correpondente hierárquia de lemas, relativamente menos complexos, e progressivamente provados. A verificação pode então ser construtiva em consonância com a implementação.

Seguindo a etapa de recombinação (ver seção IV.1.1.3), cada passo deveria ser verificado antes de proceder a implementação no próximo passo superposto, a começar pelo passo estabelecido como mais básico. Demonstrar a corretude para o passo canônico na etapa de trivialização é sempre mais simples, devido, mesmo, a pouca quantidade de sortes, operações e propriedades envolvidas.

Havendo demonstrado para o mesmo, que a respectiva implementação declarativa estende conservativamente a subestrutura mais concreta, e que a tradução da subestrutura mais abstrata constitue-se em uma interpretação, a verificação continua, mas concentrando-se apenas sobre os novos elementos presentes e suas propriedades, pois a verificação subjacente atua como um lema já provado.

Isto fica bem claro, observando as proposições seguintes,

Proposição IV.2.1 *Sejam as teorias consistentes* σ_i,σ_{i+1},σ'_i, σ'_{i+1}, δ_i *e* δ_{i+1}, *de acordo com o diagrama abaixo,*

$$\begin{array}{ccccc} \sigma_{i+1} & & \xrightarrow{\zeta_{i+1}} & & \delta_{i+1} \\ \Vdash & & & & \\ \sigma_i & \xrightarrow{\zeta_i} & \delta_i & & \cup| \\ & & \Vdash & & \\ & & \sigma'_i & \subseteq & \sigma'_{i+1} \end{array}$$

Onde a teoria δ_{i+1} *é construida conforme o teorema III.1.2. A extensão* $\sigma'_{i+1} \subseteq \delta_{i+1}$ *é conservativa se* $\sigma'_i \subseteq \sigma'_{i+1}$ *e* σ'_{i+1} *é consistente.*

Prova : Para $\sigma'_{i+1} \leq \delta_{i+1}$, é preciso que para todas as sentenças $\alpha \in \mathcal{L}(\sigma'_{i+1})$,

Se $\delta_{i+1} \models \alpha$ então $\sigma'_{i+1} \models \alpha$

Suponha inicialmente que $\delta_{i+1} \models \alpha$, para alguma sentença $\alpha \in \mathcal{L}(\sigma'_{i+1})$:

i) $\delta_i \cup \sigma'_{i+1} \models \alpha$

Pois, pelo Teorema da Construtibilidade,

$$\delta_{i+1} \equiv \zeta_{i+1}[\sigma_{i+1}] \cup \delta_i \cup \sigma'_{i+1}$$

e por construção, $\mathcal{L}(\delta_i \cup \sigma'_{i+1}) \cap \mathcal{L}(\zeta_{i+1}[\sigma_{i+1}]) \subseteq \mathcal{L}(\zeta_i[\sigma_i])$ portanto, pelo Lema da Interpolação de Craig (A.1.1), existe um conjunto de sentenças $I \subseteq \zeta_i[\sigma_i]$ tal que :

$$I \cup \delta_i \cup \sigma'_{i+1} \models \alpha$$

mas por construção $\sigma_i \xrightarrow{\zeta_i} \delta_i$, logo $I \subseteq \delta_i$. Consequentemente, $\delta_i \cup \sigma'_{i+1} \models \alpha$

ii) $\sigma'_{i+1} \models \alpha$

Porque também foi admitido que $\mathcal{L}(\delta_i) \cap \mathcal{L}(\sigma'_{i+1}) \subseteq \mathcal{L}(\sigma'_i)$, mas por **i)** $\delta_i \cup \sigma'_{i+1} \models \alpha$. Visto que σ'_{i+1} é consistente, novamente pelo Lema de Interpolação de Craig, existe $I' \subseteq \sigma'_i$, tal que,

$$I' \cup \sigma'_{i+1} \models \alpha$$

mas $I' \subseteq \sigma'_i$, e por hipótese $\sigma'_i \subseteq \sigma'_{i+1}$, consequentemente $\sigma'_{i+1} \models \alpha$

Concluindo, se $\delta_{i+1} \models \alpha$ então $\sigma'_{i+1} \models \alpha$ para qualquer sentença $\alpha \in \mathcal{L}(\sigma'_{i+1})$, portanto, $\sigma'_{i+1} \leq \delta_{i+1}$

□

Neste caso, a conservatividade da extensão de $\sigma'_{i+1} \leq \delta_{i+1}$ pode ser alternativamente garantida como conseqüência direta da extensão conservativa $\sigma'_i \leq \delta_i$, relativa ao passo canônico subjacente, e constatando que a extensão $\sigma'_i \subseteq \sigma'_{i+1}$ manteve a consistência. Isto talvez seja comparativamente menos complexo para ser determinado.

Proposição IV.2.2 *Sejam as teorias consistentes* σ_i,σ_{i+1},σ'_i, σ'_{i+1}, δ_i *e* δ_{i+1}, *de acordo com o diagrama abaixo,*

$$\begin{array}{ccccc} \sigma_{i+1} & & & & \delta_{i+1} \\ \mathbb{V} & & & & \\ \sigma_i & \xrightarrow{\zeta_i} & \delta_i & & \mathbb{V} \\ & & \mathbb{V} & & \\ & & \sigma'_i & \subseteq & \sigma'_{i+1} \end{array}$$

Onde a teoria δ_{i+1} *é construida conforme o teorema III.1.2. Seja uma função de tradução* ζ_{i+1} *talque* $\zeta_{i+1}|_{\mathcal{L}(\sigma_i)} = \zeta_i$. *Logo,*

$$\textit{Se } \zeta_{i+1}[\sigma_{i+1} - \sigma_i] \models \alpha \textit{ implica que } \delta_{i+1} \models \alpha^{\zeta_{i+1}} \textit{ então } \sigma_{i+1} \xrightarrow{\zeta_{i+1}} \delta_{i+1}$$

Prova : A fim de demonstrar que $\sigma_{i+1} \xrightarrow{\zeta_{i+1}} \delta_{i+1}$ é preciso verificar o Teorema da Interpretação, ou seja,

$$\text{Se } \sigma_{i+1} \models \alpha \text{ então } \delta_{i+1} \models \alpha^{\zeta_{i+1}}$$

para qualquer $\alpha \in \mathcal{L}(\sigma_{i+1})$. Já que $\sigma_i \leq \sigma_{i+1}$, há dois casos a considerar:

i) Supondo que também $\sigma_i \models \alpha$, consequentemente, $\delta_i \models \alpha^{\zeta_i}$ Por construção, $\delta_i \subseteq \delta_{i+1}$, então $\delta_{i+1} \models \alpha^{\zeta_i}$, como $\zeta_{i+1}[\sigma_i] = \zeta_i[\sigma_i]$, logo $\delta_{i+1} \models \alpha^{\zeta_{i+1}}$

ii) Se $\sigma_{i+1} \models \alpha$, mas $\sigma_i \not\models \alpha$, então $\alpha \notin \mathcal{L}(\sigma_i)$, porque $\sigma_i \leq \sigma_{i+1}$. Logo, é $[\sigma_{i+1} - \sigma_i]$ quem efetivamente deriva α. Por hipótese, se $[\sigma_{i+1} - \sigma_i] \models \alpha$ então $\delta_{i+1} \models \alpha^{\zeta_{i+1}}$

Concluindo, qualquer que seja α se $\sigma_{i+1} \models \alpha$ então $\delta_{i+1} \models \alpha^{\zeta_{i+1}}$. Portanto, assumindo δ_{i+1}, determinado pelo Teorema da Construtibilidade, a existência de uma interpretação $\sigma_{i+1} \xrightarrow{\zeta_{i+1}} \delta_{i+1}$ fica condicionada a verificar

$$\text{Se } \zeta_{i+1}[\sigma_{i+1} - \sigma_i] \models \alpha \text{ então } \delta_{i+1} \models \alpha^{\zeta_{i+1}}$$

Conforme assumido no enuciado, validando assim a proposição.

□

Pela segunda proposição, garantir que uma interpretação satisfaz o teorema da interpretação (A.2.2) pode ser conseguido, verificando apenas as sentenças presentes na teoria abstrata neste nível, contanto que a função de tradução participante seja uma simples extensão da anterior adaptada a nova linguagem, e que seja conservativa a extensão $\sigma_i \subseteq \sigma_{i+1}$. A verificação neste nível resume-se a demonstrar que a teoria $\sigma_{i+1} - \sigma_i$ pode ser interpretada sobre δ_{i+1}, tomando como hipótese a interpretação $\sigma_i \longrightarrow \delta_i$, que presumivelmente já está verificada.

Vendo a implementação como um todo, se for mantida a consistência, enquanto é construido o TAD que suporta a implementação, e já que a conservatividade das extensões entre as subestruturas abstratas decorre simplesmente de uma etapa de simplificação corretamente conduzida, restaria apenas certificar que a cada passo a interpretação é definida ampliando a função de tradução empregado no passo subjacente para traduzir corretamente os novos componentes de cada subestrutura abstrata.

Seja a especificação para o TAD $Queue[Elem]$:

ADT $Queue[Elem]$

Syntax

Import $Elem$

Function $create : \longmapsto Queue;$

$enqueue : Queue \times Elem \longmapsto Queue;$

$dequeue : Queue \longmapsto Queue;$

$front : Queue \longmapsto Elem$

Predicate $empty(Queue);$

Axiomatization

$(\forall q : Queue)$

$$q \overset{Q}{=} create \vee (\exists q' : Queue)(\exists e : Elem) q \overset{Q}{=} enqueue(q', e) \qquad (1)$$

$$empty(create) \tag{2}$$

$$(\forall q : Queue)(\forall e : Elem)\neg empty(enqueue(q, e)) \tag{3}$$

$$(\forall q : Queue)empty(q) \Rightarrow empty(dequeue(q)) \tag{4}$$

$$(\forall q : Queue)(\forall e : Elem)$$
$$\neg empty(q) \Rightarrow dequeue(enqueue(q, e)) \stackrel{Q}{=} enqueue(dequeue(q), e) \tag{5}$$

End

Por suposição, $Queue[Elem]$ está sendo implementada sobre $Sequence[Elem]$, cuja a especificação está sendo construida *pari-passo* com a implementação de $Queue[Elem]$. No nível presente, seja a especificação para $Sequence[Elem]$:

ADT $Sequence[Elem]$

Syntax

Import $Elem$

Function
$$\begin{array}{lll} create & : & \longmapsto Sequence; \\ insere & : & Sequence \times Elem \longmapsto Sequence; \\ concat & : & Sequence \times Sequence \longmapsto Sequence; \\ tail & : & Sequence \longmapsto Sequence; \\ head & : & Sequence \longmapsto Elem \end{array}$$

Predicate $isnull(Sequence)$;

Axiomatization

$$(\forall s : Sequence)$$
$$s \stackrel{S}{=} null \ \vee \ (\exists s' : Sequence)(\exists e : Elem)s \stackrel{S}{=} insere(s', e) \tag{1}$$

$$isnull(create) \tag{2}$$

$$(\forall s : Sequence)(\forall e : Elem)\neg isnull(insere(s, e)) \tag{3}$$

$$(\forall s, s' : Sequence)concat(s', insere(s, e)) \stackrel{S}{=} insere(concat(s', s), e) \tag{4}$$

$$(\forall s, s' : Sequence)isnull(s) \Rightarrow concat(s', s) \stackrel{S}{=} s' \ \wedge \ concat(s', s) \stackrel{S}{=} s' \tag{5}$$

$$(\forall s' : Sequence)(\exists s : Sequence)(\exists e : Elem)$$
$$s' \stackrel{S}{=} insere(s, e) \Rightarrow head(s') \stackrel{E}{=} e \tag{6}$$

$$(\forall s, s' : Sequence)\neg isnull(s) \Rightarrow head(concat(s', s)) \stackrel{E}{=} head(s) \tag{7}$$

$$(\forall s' : Sequence)(\exists s : Sequence)(\exists e : Elem)$$
$$s' \stackrel{S}{=} insere(s, e) \Rightarrow tail(s') \stackrel{S}{=} s \tag{8}$$

$$(\forall s : Sequence)isnull(s) \Rightarrow isnull(tail(s)) \tag{9}$$

$$(\forall s : Sequence)\neg isnull(s) \Rightarrow tail(concat(s', s)) \stackrel{E}{=} concat(s', tail(s)) \quad (10)$$

End

No próximo nível, precisa ser construido um passo canônico superposto, de modo que o símbolo $front$ de $Queue[Elem]$, definido pelos axiomas:

$$(\forall q : Queue)(\forall e : Elem)$$
$$empty(q) \Rightarrow front(enqueue(q, e)) \stackrel{E}{=} e$$
$$(\forall q : Queue)(\forall e : Elem)\neg empty(q) \Rightarrow front(enqueue(q, e)) \stackrel{E}{=} front(q)$$

Neste nível, $Sequence[Elem]$ já tem símbolos suficientes para representar $front$. Concluida a representção de $front$ é obtida a especificação:

ADT $Queue^*[Elem]$

Syntax

Import $Sequence[Elem], Elem$

Function $create^* : \longmapsto Queue^*;$
$enqueue^* : Queue^* \times Elem \longmapsto Queue^*;$
$dequeue^* : Queue^* \longmapsto Queue^*;$
$front^* : Queue^* \longmapsto Elem$

Predicate $empty^*(Queue^*);$

Axiomatization

$$(\forall q : Queue^*)(\exists s : Sequence)q \stackrel{S}{=} s \quad (1)$$
$$create^* \stackrel{S}{=} null \quad (2)$$
$$(\forall q : Queue^*)empty^*(q) \Leftrightarrow isnull(q) \quad (3)$$
$$(\forall q : Queue^*)(\forall e : Elem)$$
$$enqueue^*(q, e) \stackrel{S}{=} concat(insere(null, e), q) \quad (4)$$
$$(\forall q : Queue^*)dequeue^*(q) \stackrel{S}{=} tail(q) \quad (5)$$
$$(\forall q : Queue^*)front^*(q) \stackrel{E}{=} head(q) \quad (6)$$

End

Tomando-se como Lema a implementação executada no passo subjacente ao refinamento de $front$, é preciso verificar que entre as conseqüências de $Queue^*[Elem]$ a tradução dos axiomas que definem $front$. Somente isto é necessário para certificar que

o último passo canônico composto está realmente constituido corretamente, segundo as proposições IV.2.1 e IV.2.2. Portanto, partindo dos axiomas de $Queue^*[Elem]$ é que serão derivados os axiomas para $front$, conforme abaixo:

$$(\forall s : Sequence)(\exists s' : Sequence)(\exists e : Elem) s \stackrel{S}{=} insere(s', e) \Rightarrow head(s) \stackrel{E}{=} e$$

se o axioma vale para qualquer s, em particular vale para $insere(null, e)$, pois,

$$(\exists s : Sequence)(\exists e : Elem) s \stackrel{S}{=} insere(null, e)$$

logo,

$$\begin{aligned}
&(\exists s' : Sequence)(\exists e : Elem) \\
&\qquad insere(null, e) \stackrel{S}{=} insere(s', e) \Rightarrow head(insere(null, e)) \stackrel{E}{=} e \\
&(\exists s' : Sequence)(\exists e : Elem) s' \stackrel{S}{=} null \Rightarrow head(insere(null, e)) \stackrel{E}{=} e \\
&(\exists s' : Sequence)(\exists e : Elem) isnull(s') \Rightarrow head(insere(null, e)) \stackrel{E}{=} e
\end{aligned}$$

pelas propriedades de $Sequence$, como s' também é nulo, pode ser concatenado com a sequência $null$, sem alterar o resultado, logo,

$$\begin{aligned}
&(\exists s' : Sequence)(\exists e : Elem) isnull(s') \Rightarrow head(insere(concat(s', null), e)) \stackrel{E}{=} e \\
&(\exists s' : Sequence)(\exists e : Elem) isnull(s') \Rightarrow head(concat(s', insere(null, e)) \stackrel{E}{=} e \\
&(\exists s' : Sequence)(\exists e : Elem) isnull(s') \Rightarrow head(concat(insere(null, e), s') \stackrel{E}{=} e
\end{aligned}$$

Pela função de tradução diretamente,

$$(\exists s' : Queue^*)(\exists e : Elem) empty^*(s') \Rightarrow front^*(enqueue^*(s', e)) \stackrel{E}{=} e$$

O segundo axioma na definição de $front$ segue da tradução direta do axioma (7) de $Sequence$, tomando como s' a constante $insere(null, e)$), ou seja,

$$(\forall s : Sequence) \neg isnull(s) \Rightarrow head(concat(insere(null, e), s)) \stackrel{E}{=} head(s)$$

já que e aparece livre pode ser universalmente quantificado,

$$(\forall s : Sequence)(\forall e : Elem) \neg isnull(s) \Rightarrow head(concat(insere(null, e), s)) \stackrel{E}{=} head(s)$$

Diretamente pela função de tradução,

$$(\forall s : Queue^*)(\forall e : Elem) \neg isnull(s) \Rightarrow front^*(enqueue^*(s, e)) \stackrel{E}{=} front^*(s)$$

IV.3 Evolução

Há um consenso em concordar que todo e qualquer programa, destinado a aplicações computacionais, deva ser desenvolvido já levando em conta a possibilidade de evoluir, a fim de absorver crescimentos ou mudanças nos requisitos e/ou objetivos que motivaram a sua concepção.

Além do que, o respectivo domínio sob consideração de um programa, embora seja abstrato (filosoficamente sua existência está circunscrita à percepção pessoal do projetista), por pretender espelhar entidades, atributos e fenomenos reais, é em essência, dinâmico e mutável.

Ao construir uma especificação, há pois, a preocupação de modelar alguns conceitos, abstraído neste domínio, e a eles associar representações formais, mas que também sejam capazes de evoluir, em resposta, refletindo qualquer transformção real correspondente.

Foge inteiramente ao escopo deste texto discutir sobre causas e motivos, interessando-se em saber apenas que provavelmente surge, a necessidade de expandir ou redefinir uma especificação. Consequentemente, todo o processo de implementação seguinte será de algum modo afetado.

Quando e quais novos requisitos por ventura venham a se fazer necessários, não costuma ser facilmente previsível. Se o fossem, não haveria porque não já aceitá-los de ante mão. Então, o processo de implementação deve estar sempre preparado para fazer face às novas contingências.

Em circunstâncias reais, por exemplo, são comuns solicitações para acrescentar alguns novos elementos, ou alterar os já existentes, mesmo após concluida a implementação. Tecnicamente, isto implica em estender as conseqüências que podem ser inferidas, à partir de sua implementação declarativa. Ou no segundo caso, modificar este conjunto de conseqüências.

IV.3.1 Extensão em uma Implementação

Frequentes são as situações em que se faz exigida a introdução de novos elementos a uma implementação antes estabelecida. Quer em uma perspectiva de ampliação dos resultados implementados, quer ao se constatar a não suficiência das informações presentes na

implementação, requerendo restrições adicionais, que a tornem menos permissiva e mais específica.

Caracteriza-se de uma forma ou de outra, a necessidade de enriquecer a implementação declarativa. O que invariavelmente remete ao mecanismo lógico de extensão e por conseqüência direta, entram em cena novamente, os mecanismos de composição de implementação, aludidos na seção III.1.

Para tanto é aconselhável distinguir a natureza dos elementos a serem acrescentados à implementação:

- Um componente completo, que não pertencia à concepção inicial agora se faz necessário;
- Algumas propriedades agora importantes, não constam inicialmente na especificação.

A primeira situação relaciona-se com a necessidade de ampliar em a implementação, enquanto a outra significa obviamente especializá-la.

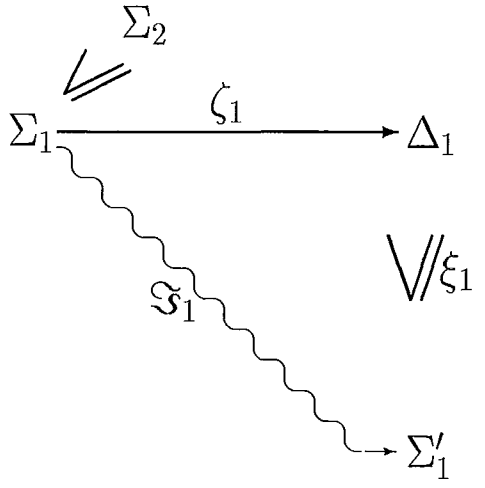
À princípio, suponha haver uma especificação Σ_1, que está implementada em termos de uma outra Σ_1' ($\Im_1 : \Sigma_1 \rightsquigarrow \Sigma_1'$). É justamente fundamentando-se sobre esta relação de implementação $\Im_1$, onde se deseja ter introduzido novos elementos, evoluindo $\Im_1$ para fazer face a contingências mais recentes.

IV.3.1.1 Ampliação em uma Implementação

Quanto a primeira situação, não parece ser razoável, e em geral não é, ampliar as conseqüências relacionadas com uma determinada implementação declarativa Δ_1, resultante do passo $\Im_1$, simplesmente adicionando-lhe diretamente os novos elementos solicitados. Dependendo do nível de detalhes da linguagem na qual foi especificado Δ_1, tende a ser desnecessariamente complicado, definir representações corretas e adequadas para os elementos a acrescentar, se isto for feito diretamente em Δ_1. Além de uma dificuldade a mais, para convencer que os resultados obtidos constituem, de fato, uma passo de implementação.

Alternativamente, estes novos componentes pode ser descritos em uma especificação de mais "alto" nível, conceitualmente mais próxima da percepção inicial da realidade modelada, onde se supõe ser mais natural sua representação. Então, resta deixar o mecanismo de composição encarregar-se de incorporá-los

A princípio, um componente completo deveria ser adicionado, o que supostamente significaria acrescentar ao menos um símbolo inteiramente novo, não previsto na linguagem de Σ_1. Este caso, portanto, talvez não seja tão problemático, e meramente adicioná-lo conservativamente a Σ_1, deve ser satisfatório. Os símbolos, juntamente com os axiomas que expressam suas propriedades, são colocados em Σ_1, através de uma extensão conservativa, formando uma nova especificação, Σ_2. Observe então o diagrama a seguir,



Por instância, seja Σ_1 o TAD $SStruct[Key, Value]$, denotando uma estrutura de armazenamento de informações (e.g. $Value$), que devem ser recuperadas eficientemente, de acordo com uma chave de sorte Key.

ADT $SStruct[Key, Value]$

Syntax

Import $Key, Value$

Function $inicialize : \longmapsto SStruct;$

$insere : SStruct \times Key \times Value \longmapsto SStruct;$

$remove : SStruct \times Key \longmapsto SStruct;$

$search : SStruct \times Key \longmapsto Value$

Predicate $isempty(SStruct);$

Axiomatization

$$(\forall s : SStruct)\{ s \stackrel{S}{=} inicialize \vee$$

$$(\exists s' : SStruct)(\exists k : Key)(\exists v : Value) s \stackrel{S}{=} insere(s', k, v)\} \quad (1)$$

$$isempty(inicialize) \quad (2)$$

$$(\forall s : SStruct)(\forall k : Key)(\forall v : Value) \neg isempty(insere(s, k, v)) \quad (3)$$

$$(\forall s : SStruct)(\forall k : Key)isempty(s) \Rightarrow isempty(remove(s,k)) \quad (4)$$

$$(\forall s : SStruct)(\forall k : Key)(\forall v : Value)$$
$$remove(insere(s,k,v),k) \stackrel{S}{=} s \quad (5)$$

$$(\forall s : SStruct)(\forall k' : Key)(\forall v : Value)\{\neg(isempty(s) \vee k \stackrel{K}{=} k')$$
$$\Rightarrow remove(insere(s,k,v),k') \stackrel{S}{=} insere(remove(s,k'),k,v)\} \quad (6)$$

$$(\forall s : SStruct)(\forall k : Key)(\forall v : Value)$$
$$search(insere(s,k,v),k) \stackrel{V}{=} v \quad (7)$$

$$(\forall s : SStruct)(\forall k' : Key)(\forall v : Value)$$
$$\neg(isempty(s) \vee k \stackrel{K}{=} k') \Rightarrow search(insere(s,k,v),k') \stackrel{V}{=} search(s,k') \quad (8)$$

End

Supondo uma relação de ordem ($\lhd$) definida no sorte *Label*, uma árvore binária ordenada, será utilizada para implementar $SStruct[Key, Value]$, especificando antes, para cada nó da árvore um sorte auxiliar *Item*, como o produto cartesiano de *Label* e *Node*, dois sortes que futuramente representarão *Key* e *Value*:

ADT *Item*

Syntax

Import *Label*, *Node*

Function $label : Item \longmapsto Label;$
$node : Item \longmapsto Node;$

Axiomatization

$$(\forall a : Label)(\forall n : Node)(\exists i : Item)label(i) \stackrel{L}{=} a \wedge node(i) \stackrel{N}{=} n \quad (1)$$

$$(\forall i, i' : Item)label(i) \stackrel{L}{=} label(i') \wedge node(i) \stackrel{N}{=} node(i') \Rightarrow i \stackrel{I}{=} i' \quad (2)$$

End

Assim seja $BinTree[Item]$:

ADT $BinTree[Item]$

Syntax

Import *Item*

Function $null : \longmapsto BinTree$;
$left : BinTree \longmapsto BinTree$;
$right : BinTree \longmapsto BinTree$;
$root : BinTree \longmapsto Item$;
$btree : BinTree \times Item \times BinTree \longmapsto BinTree$;
$rightmost : BinTree \longmapsto Item$;
Predicate $isnull(BinTree)$;

Axiomatization

$$(\forall l, r : BinTree)(\forall i : Item)(\exists b : BinTree)$$
$$\{left(b) \stackrel{T}{=} l \ \wedge \ root(b) \stackrel{I}{=} i \ \wedge \ right(b) \stackrel{T}{=} r\} \quad (1)$$
$$(\forall b, b' : BinTree) \neg b \stackrel{T}{=} b' \Rightarrow$$
$$\neg\{left(b) \stackrel{T}{=} left(b') \ \wedge \ root(b) \stackrel{I}{=} root(b') \ \wedge \ right(b) \stackrel{T}{=} right(b')\} \quad (2)$$
$$(\forall l, r : BinTree)(\forall i : Item)(\exists b : BinTree) b \stackrel{T}{=} btree(l, i, r) \quad (3)$$
$$isnull(null) \quad (4)$$
$$(\forall l, r : BinTree)(\forall i : Item) \neg isnull(btree(l, i, r)) \quad (5)$$
$$(\forall b : BinTree) \neg isnull(left(b)) \Rightarrow label(root(b)) \lhd label(root(left(b))) \quad (6)$$
$$(\forall b : BinTree) \neg isnull(right(b)) \Rightarrow label(root(b)) \lhd label(root(right(b))) \quad (7)$$
$$(\forall l, r : BinTree)(\forall i : Item)$$
$$isnull(r) \Rightarrow rightmost(btree(l, i, r)) \stackrel{I}{=} i \quad (8)$$
$$(\forall l, r : BinTree)(\forall i : Item)$$
$$\neg isnull(r) \Rightarrow rightmost(btree(l, i, r)) \stackrel{I}{=} rightmost(r) \quad (9)$$

End

Para mediar este passo de implementação foi estabelecida a especificação:

ADT $SStruct^*[Key^*, Value^*]$

Syntax

Import $BinTree[Item], Key^*, Value^*$

Function $inicialize^* : \longmapsto SStruct^*$;
$insere^* : SStruct^* \times Key^* \times Value^* \longmapsto SStruct^*$;
$remove^* : SStruct^* \times Key^* \longmapsto SStruct^*$;
$search^* : SStruct^* \times Key^* \longmapsto Value^*$

Predicate $isempty^*(SStruct^*)$;

Axiomatization

$$(\forall k, k' : Key)k \rhd k' \Leftrightarrow \neg(k \lhd k' \ \vee \ k \stackrel{K}{=} k') \tag{1}$$

$$(\forall s : SStruct^*)(\exists b : BinTree)s \stackrel{S}{=} b \tag{2}$$

$$(\forall k : Key^*)(\exists a : Label)k \stackrel{L}{=} a \tag{3}$$

$$(\forall v : Value^*)(\exists n : Node)v \stackrel{N}{=} n \tag{4}$$

$$inicialize^* \stackrel{T}{=} null \tag{5}$$

$$(\forall s : SStruct^*)isempty^*(s) \Leftrightarrow isnull(s) \tag{6}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)(\forall v : Value^*)isempty^*(s) \Rightarrow (\exists i : Item)$$
$$\{label(i) \stackrel{L}{=} k \ \wedge \ node(i) \stackrel{N}{=} v \ \wedge$$
$$insere^*(s,k,v) \stackrel{T}{=} btree(isnull, i, isnull)\} \tag{7}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)(\forall v : Value^*)$$
$$k \lhd label(root(s)) \Rightarrow insere^*(s,k,v) \stackrel{T}{=} insere^*(left(s),k,v) \tag{8}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)(\forall v : Value^*)$$
$$k \rhd label(root(s)) \Rightarrow insere^*(s,k,v) \stackrel{T}{=} insere^*(right(s),k,v) \tag{9}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)(\forall v : Value^*)$$
$$k \stackrel{L}{=} label(root(s)) \Rightarrow insere^*(s,k,v) \stackrel{T}{=} s \tag{10}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)$$
$$k \lhd label(root(s)) \Rightarrow remove^*(s,k) \stackrel{T}{=} remove^*(left(s),k) \tag{11}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)$$
$$k \rhd label(root(s)) \Rightarrow remove^*(s,k) \stackrel{T}{=} remove^*(right(s),k) \tag{12}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)\{k \stackrel{L}{=} label(root(s)) \ \wedge \ isnull(left(s))\} \Rightarrow$$
$$remove^*(s,k) \stackrel{T}{=} btree(left(right(s)), root(right(s)), right(right(s))) \tag{13}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)\{k \stackrel{L}{=} label(root(s)) \ \wedge \ \neg isnull(left(s))\} \Rightarrow \tag{14}$$
$$(\exists b : BinTree)b \stackrel{T}{=} remove^*(left(s), rightmost(left(s)))$$
$$remove^*(s,k) \stackrel{T}{=} btree(b, rightmost(left(s)), right(s)) \tag{15}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)$$
$$k \lhd label(root(s)) \Rightarrow search^*(s,k) \stackrel{N}{=} search^*(left(s),k) \tag{16}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)$$
$$k \rhd label(root(s)) \Rightarrow search^*(s,k) \stackrel{N}{=} search^*(right(s),k) \tag{17}$$

$$(\forall s : SStruct^*)(\forall k : Key^*)$$
$$k \stackrel{L}{=} label(root(s)) \Rightarrow search^*(s,k) \stackrel{N}{=} node(root(s)) \quad (18)$$

End

Ocasionalmente, talvez seja notada a necessidade de uma operação que retorne o valor correspondente a "maior" chave armazenada (maior em relação a $\lhd$). Descrevendo esta operação, nomeada de max (i.e. $max : SStruct \longmapsto Value$) sejam o axioma:

$$(\forall s : SStruct)(\forall v : Value)v \stackrel{V}{=} max(s) \Leftrightarrow$$
$$(\exists k : Key)v \stackrel{V}{=} search(s,k) \ \wedge \ (\forall k' : Key)(\forall v' : Value)$$
$$v' \stackrel{V}{=} search(s,k') \Rightarrow \neg k \lhd k'$$

Faz sentido proteger as propriedades, já descritas através da especificação, bloqueando perturbações não intencionais em suas conseqüências. A imposição da conservatividade funciona exatamente como esta proteção. Logo, o axioma acima seria adicionado conservatimente em $SStruct$, formando uma nova especificação (Σ_2).

Então, o mecanismo de composição têm agora a reponsabilidade de incorporar à teoria descrita por Δ_1, todas as informações relativas aos novos ou símbolos acrescentados em Σ_2. As conseqüências de suas respectivas propriedades devem estar presentes, na teoria que intermedia o novo passo de implementação, como se cada um destes símbolos constasse antes da especificação abstrata original.

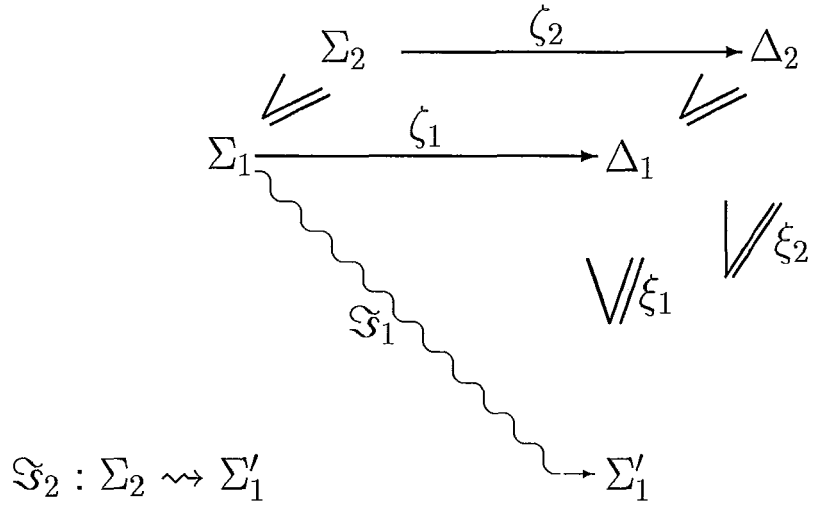
Sendo este o caso, eventualmente poder-se-ia lançar mão do Teorema da Modularização.[7] Mas ainda assim, poder-se-ia recorrer ao Teorema da Construtibilidade, pois, sua utilização coaduna com a exata noção intuitiva, pensando em ampliar como superpor um próximo passo de implementação, estendendo o já realizado. Dado que a ampliação se propõe a ser conservativa, a escolha óbvia é sua versão mais forte (teorema III.2.1), pois esta conserva inteiramente a teoria especificada por Δ_1 e então resguardando detalhes e decisões de refinamento.

Evidentemente, este texto se propõe a explorar preferencialmente a segunda possibilidade, pretendendo homogeneizar esta abordagem com a *especialização de implementações,* conforme será visto adiante.

A implementação de Σ_2 em Σ_1' ($\Im_2 : \Sigma_2 \rightsquigarrow \Sigma_1'$) é na realidade quem efetua a adição dos componentes, representados em Σ_2, na implementação declarativa subjacente,

[7]Em [TM87], encontra-se melhor descrito como fazê-lo.

Δ_1. Pelo Teorema da Construtibilidade (versão forte), é sempre possível estabelecer uma implementação superposta, construindo uma nova teoria especificada em Δ_2, por sobre Δ_1, estendendo esta última conservativamente. Existe também, pelo menos uma interpretação possível de Σ_2 em Δ_2 ($\Sigma_2 \longrightarrow \Delta_2$), que traduz em teoremas de Δ_2, os novos axiomas especificados em Σ_2, como foi desejado.



Por conseguinte, em uma extensão conservativa de $SStruct^*[Key^*, Value^*]$, (Δ_1) é possível refinar *max* incorporando o axioma:

$$(\forall s : SStruct^*) max^*(s) \stackrel{N}{=} node(rightmost(s))$$

Se o resultado de implementação é ampliado conservativamente, então é natural esperar que o conjunto de realizações possíveis na implementação declarativa Δ_1 não seja reduzido. Melhor dizendo, cada modelo induzido por ζ_1 em Σ_1, seja expandido a pelo menos um modelo correspondente para Σ_2, induzido por ζ_2, mas que satisfaça exatamente as mesmas sentenças na linguagem de Σ_1.

Proposição IV.3.1 *Seja a composição de implementações conforme abaixo:*

$$\begin{array}{ccc} \Sigma_2 & \xrightarrow{\zeta_2} & \Delta_2 \\ \Vdash & & \Vdash \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 \\ & & \Vdash \\ & & \Sigma'_1 \end{array}$$

Se $\Delta_1 \trianglelefteq \Delta_2$ *então,* $[\forall \Re \in \boldsymbol{Mod}(\Delta_1)] \quad (\Re^{\zeta_2^{-1}})|_{\mathcal{L}(\Sigma_1)} \in \boldsymbol{Mod}(\Sigma_1)$

Prova :

1. Inicialmente, uma vez que $\Sigma_2 \xrightarrow{\zeta_2} \Delta_2$, pelo teorema A.2.2, $\Re^{\zeta_2^{-1}} \in \mathbf{Mod}\,(\Sigma_2)$.
2. como $\Delta_1 \trianglelefteq \Delta_2$, por definição, $\Re|_{\mathcal{L}(\Delta_1)} \in \mathbf{Mod}\,(\Delta_1)$
3. novamente por A.2.2, $(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}} \in \mathbf{Mod}\,(\Sigma_1)$

Resta mostrar que o reduto (A.1.8) de qualquer estrutura, e.g. $\Re^{\zeta_2^{-1}}$, induzida por ζ_2 na $\mathcal{L}(\Sigma_1)$, é modelo para Σ_1 sempre que $\Re^{\zeta_2^{-1}}$ está entre os modelos para Σ_2. Para tanto, é preferível mostrar que qualquer realização $\Re|_{\mathcal{L}(\Delta_1)}$ para Δ_1, induzida por ζ_1 para Σ_1, é extamente o reduto $(\Re^{\zeta_2^{-1}})|_{\mathcal{L}(\Sigma_1)}$ esperado. Ou seja, $(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}}$ concorda com $(\Re^{\zeta_2^{-1}})|_{\mathcal{L}(\Sigma_1)}$ na realização de todos os símbolos na linguagem de Σ_1.

Deste modo, seja S um símbolo de sorte em $\mathcal{L}(\Sigma_1)$, por definição de modelo induzido, A.2.3,

$$(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}}|S| = \Re|_{\mathcal{L}(\Delta_1)}\,|\zeta_1(S)|$$

mas por construção de ζ_2, $\zeta_1[\Sigma_1] = \zeta_2[\Sigma_1]$, logo,

$$(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}}|S| = \Re|_{\mathcal{L}(\Delta_1)}\,|\zeta_2(S)|$$

mas para S^{ζ_2} pertencente a linguagem de Δ_1, $\Re$ e $\Re|_{\mathcal{L}(\Delta_1)}$ concordam quanto a S, portanto,

$$(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}}|S| = \Re|\zeta_2(S)|$$

Novamente, por definição de modelo induzido,

$$(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}}|S| = \Re^{\zeta_2^{-1}}|(S)|$$

O mesmo argumento pode ser repetido para todos os símbolos de função ou predicado em $\mathcal{L}(\Sigma_1)$, consequentemente, $(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}}$ e $\Re^{\zeta_2^{-1}}$ estão de acordo com a realização para qualquer símbolo no repertório de Σ_1, consequentemente, $(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}}$ concorda na verdade com o reduto de $\Re^{\zeta_2^{-1}}$ na linguagem $\mathcal{L}(\Sigma_1)$, então,

$$(\Re|_{\mathcal{L}(\Delta_1)})^{\zeta_1^{-1}} = (\Re^{\zeta_2^{-1}})|_{\mathcal{L}(\Sigma_1)}$$

Concluindo assim a demonstração. □

A proposição IV.3.1 agora demonstrada, afirma que sendo expansiva a extensão entre as implementações declarativas Δ_1 e Δ_2, qualquer modelo induzido por ζ_1 irá expandir-se a pelo menos um modelo induzido por ζ_2, na linguagem abstrata. Isto significa que a ampliação realizada não perde as realizações admitidas no passo canônico subjacente.

Ocasionalmente, o passo canônico $\Im_1$, onde a ampliação precisa ser realizada, participa de uma consecução de passos $\ldots, \Im_0, \Im_1, \ldots$, e então, é provável que a construção de $\Im_2$ desencadeie a necessidade de evoluir também em outros passos. Na verdade, $\Im_2$ ainda acontece em termos de Σ_1', logo não há qualquer efeito sobre os passos que o sucedem. Isto pode ser observado em IV.3.8.

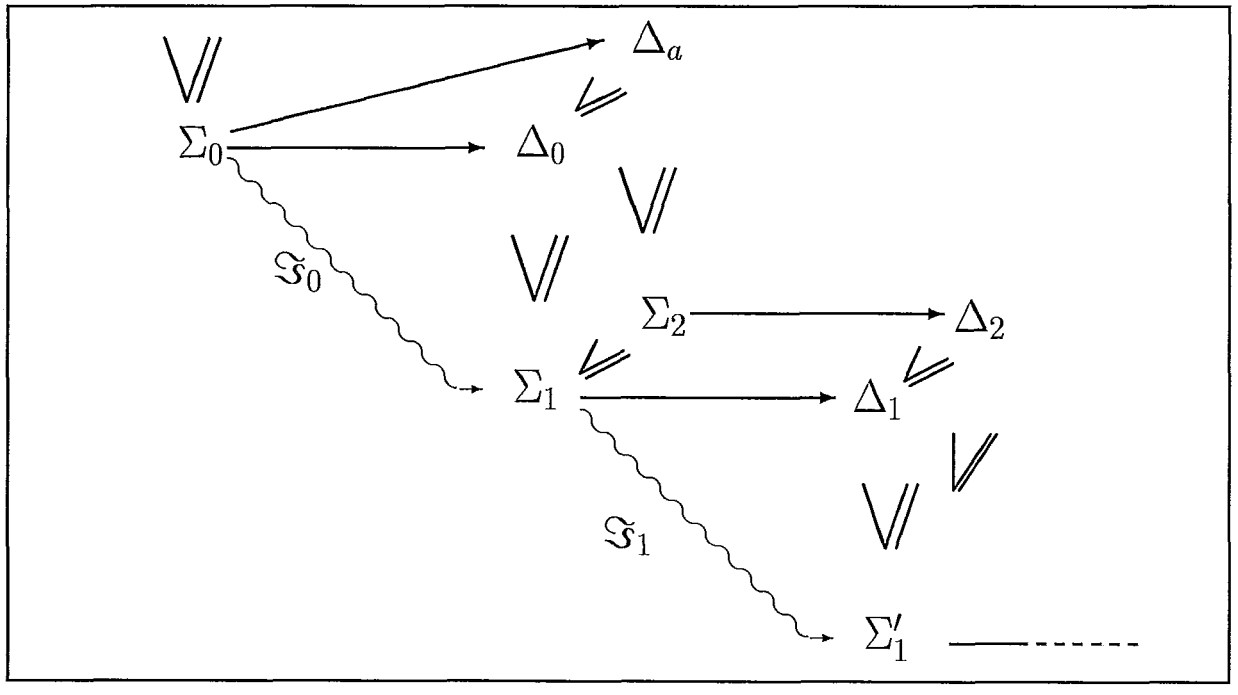


Figura IV.3.8: Propagação da ampliação em Passos Subsequntes

Por outro lado, a especificação sendo implementada, foi extendida de Σ_1 para Σ_2, e portanto, foi perdida a conexão com os passos que precediam $\Im_1$. Esta amplição causa ao passo antecedente $\Im_0$, a necessidade de evoluir, exigindo uma nova implementação declarativa capaz de absorvê-la, e reestabelecer o sequenciamento de passos.

Um vez mais, deixa-se à composição de implementações subjacentes, a incumbência de propagar a ampliação realizada em $\Im_2$, para $\Im_0$. A implementação declarativa Δ_a substitue Δ_0, da mesma forma que Δ_2 fica no lugar de Δ_1. As especificações Δ_0, Σ_1 e Δ_1 são desconsideradas na implementação.

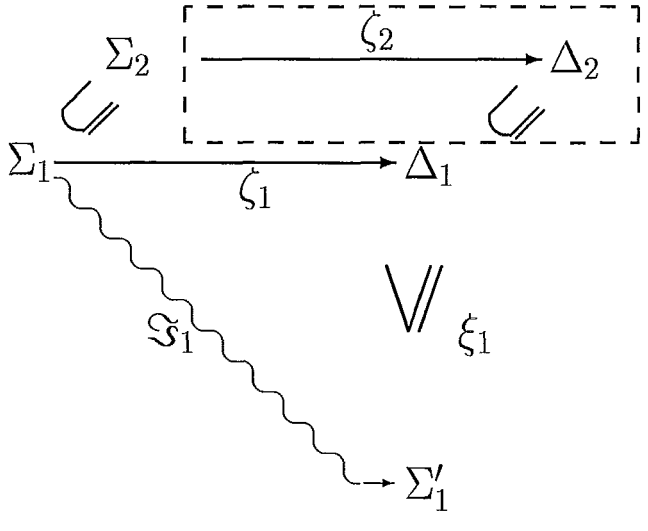
Recompor novamente toda a seqüência de passos resulta em uma implementação declarativa, que efetua a transição direta da representação mais abstrata para a mais primitiva, e adquire automaticamente o novo componente e as conseqüências a ele relativas.

IV.3.1.2 Especialização em uma Implementação

Eventualmente a implementação falha em atender aos objetivos que provocaram sua construção. Ou por que evoluiram estes, ou por algo importante haver sido deixado de fora na concepção inicial. De qualquer modo, algumas propriedades, convenientemente incorporadas, talvez sejam suficientes torná-la mais uma vez satisfatória, ao invés de remontar novamente todo o trabalho. Portanto, pode ser necessária, ou no mínimo vantajosa, esta capacidade de especializar em implementações.

A especialização em uma implementação revela-se em um contexto bem mais interessante. Se novas propriedades podem agora ser derivadas, por outro lado, a representação obtida torna-se mais restritiva. Pois, algumas entre as suas possíveis realizações são descartadas, porque não mais satisfazem a nova teoria formada. Somente são modelos para a nova teoria aquelas realizações que atendem, intuitivamente, as novas propriedades. A implementação declarativa evoluiria, ganhando em precisão e especificidade.

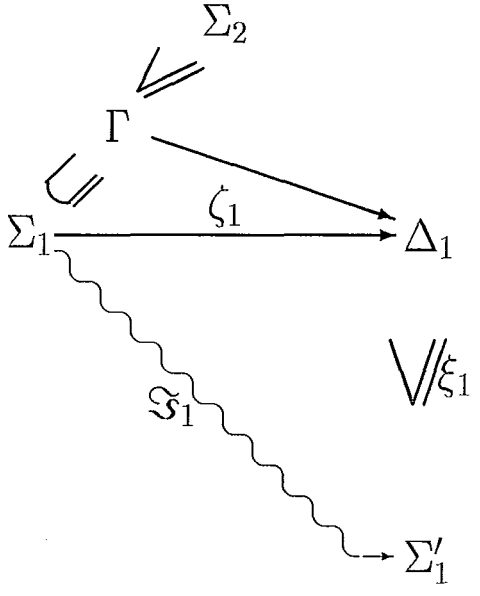
Superficialmente, pareceria ser razoável também adicionar a Σ_1, diretamente na linguagem deste, axiomas descrevendo as propriedades que se pretende introduzir (Solução semelhante à adotada antes para ampliar uma implementação), e então, estendendo-se a implementação subjacente, representada pelo par $\langle \zeta_1, \xi_1 \rangle$, para a especificação resultante Σ_2, de modo a interpretar em Δ_2, as informações recém adicionadas.



Aparentemente, esta situação é similar às condições iniciais para o Teorema da Modularização. Era esperado consequentemente, completar o diagrama da mesma maneira, conforme está apresentado acima. Entretanto, esta adição amplia, necessariamente, conseqüências relativas a símbolos pertencentes a Δ_1, portanto, não se trata de uma extensão conservativa.

Um acréscimo posterior de propriedades revela-se problemático, quando está envolvida uma extensão não conservativa,[8] porque não é necessariamente possível, determinar a consistência de Δ_2, mesmo exigindo a consistência de Δ_1, e logo, é incerta a existência de modelos para Δ_2. Ainda que supostamente consistente, esta não mediaria, por definição o passo $\Sigma_2 \rightsquigarrow \Sigma_1'$, já que não haveria como garantir que $\Sigma_1' \subseteq \Delta_2$ é conservativa.

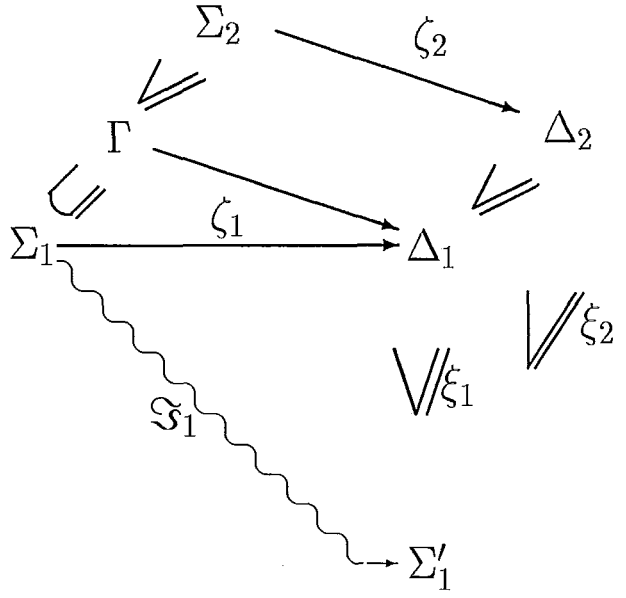
Uma tentativa inicial, para contornar estas dificuldades, seria encontrar uma teoria Γ, que fosse ao mesmo tempo uma extensão consistente de Σ_1 e extendível a Σ_2 conservativamente. Assim, Γ, mais ou menos, media a extensão de $\Sigma_1 \subseteq \Sigma_2$ (isto é $\Sigma_1 \subseteq \Gamma \leq \Sigma_2$).



Resta procurar uma interpretação de Γ em Δ_2, e deste modo, tomando como uma aplicação subjacente do passo canônico, a implementação $\Gamma \rightsquigarrow \Sigma_1'$ assim definida, pelo Teorema da Construtibilidade haveria também uma implementação $\Sigma_2 \rightsquigarrow \Sigma_1'$. Mais ainda, a implementação declarativa produzida Δ_2 seria uma extensão conservativa de Δ_1,

[8] conforme discutido em [TM87]

e apresentaria entre os seus teoremas, a tradução dos axiomas especificando Σ_2.



Este procedimento pode ser formalmente fundamentado através da proprosição seguinte:

Proposição IV.3.2 *Dado teorias consistentes* Σ_1,Σ_1', Σ_2 *e* Σ_2' *tais que:*

i) $\Sigma_1 \rightsquigarrow \Sigma_1'$

ii) $\Sigma_1 \subseteq \Sigma_2$

iii) $\Sigma_1' \subseteq \Sigma_2'$

então existe uma teoria Γ, *tal que* $\Sigma_1 \subseteq \Gamma \leq \Sigma_2$ *e* $\Sigma_2 \rightsquigarrow \Sigma_2'$ *se* $\Gamma \rightsquigarrow \Sigma_2'$.

Prova : Primeiro, caso $\Sigma_1 \subseteq \Sigma_2$ então existe sempre uma teoria Γ, de tal forma que $\Sigma_1 \subseteq \Gamma \leq \Sigma_2$, simplesmente tomando $\Gamma = \Sigma_2|_{\mathcal{L}(\Sigma_1)}$.

Além disto, $\Gamma = \Sigma_2|_{\mathcal{L}(\Sigma_1)}$ é a menor teoria que satisfaz a $\Sigma_1 \subseteq \Gamma \leq \Sigma_2$. Para tanto, suponha $\Gamma' \neq \Sigma_2|_{\mathcal{L}(\Sigma_1)}$ e ainda que $\Sigma_1 \subseteq \Gamma' \leq \Sigma_2$. Há duas alternativas:

i) $\Gamma' \subset \Sigma_2|_{\mathcal{L}(\Sigma_1)}$ Neste caso, $\Sigma_1 \subseteq \Gamma' \subset \Sigma_2|_{\mathcal{L}(\Sigma_1)} \leq \Sigma_2$, logo existe, por definição, pelo menos uma sentença ϕ na linguagem $\mathcal{L}(\Sigma_1)$, que pertence as conseqüências de Σ_2 e não está em Γ',

$$\Sigma_2 \models \phi \quad e \quad \Gamma' \not\models \phi$$

e portanto $\Gamma' \not\leq \Sigma_2$, e deste modo é impossível que Γ' esteja contida em Γ.

ii) $\Gamma' \not\subset \Sigma_2 |_{\mathcal{L}(\Sigma_1)}$ Ou seja, existe pelo menos uma sentença ϕ na linguagem de Σ_2, tal que,

$$\Gamma' \models \phi \quad e \quad \Sigma_2 |_{\mathcal{L}(\Sigma_1)} \not\models \phi$$

Todavia, como $\Gamma' \leq \Sigma_2$, ϕ nunca poderia pertencer a $\mathcal{L}(\Sigma_1)$, isto é

$$\text{Se } \Sigma_2 |_{\mathcal{L}(\Sigma_1)} \models \phi \text{ então } \Gamma' \models \phi$$

consequentemente, $\Sigma_2 |_{\mathcal{L}(\Sigma_1)} \subseteq \Gamma'$.

O que comprova que $\Gamma \equiv \Sigma_2 |_{\mathcal{L}(\Sigma_1)}$ é a menor teoria que satisfaz:

$$\Sigma_1 \subseteq \Gamma \leq \Sigma_2$$

Finalmente, se $\Gamma \rightsquigarrow \Sigma_2'$ e $\Gamma \leq \Sigma_2$ então, por conseqüência direta do Teorema da Construtibilidade III.1.2, tem-se $\Sigma_2 \rightsquigarrow \Sigma_2'$.

□

Sempre é possível atender a primeira condição na proposição acima, então apenas seria necessário demonstrar que uma interpretação, ζ_1, para o passo canônico subjacente, também será uma interpretação, considerando $\Gamma \xrightarrow{\zeta_1} \Delta_1$.

Sendo realista, não há certeza que a tradução seja também neste caso uma interpretação, o que não é realmente incomum na prática. Basta que $\Sigma_2 |_{\mathcal{L}(\Sigma_1)}$ seja mais "forte" que a teoria Δ_1, no sentido em que admite menos modelos, para inviabilizar a interpretação. Em geral, quando se trata de uma especialização, este costuma ser o caso. Além do que a teoria $\Sigma_2 |_{\mathcal{L}(\Sigma_1)}$ pode nem ser finitamente axiomatizável, dificultando a qualquer utilização efetiva no desenvolvimento de programas.

Todavia, não há realmente obrigatoriedade, em especializar a implementação, explicitamente através de axiomas descritos na linguagem relativa a especificação Σ_1. Nem é de modo algum absurdo, pensar em introduzir restrições, implicitamente na transição de nível lingüístico mais abstrato para o mais concreto. Assim seria então, uma alternativa bastante razoável acrescentar estas propriedades adicionais à implementação, indiretamente, representando-as na linguagem de Σ_1', que é um recurso semelhante ao empregado para adicionar detalhes não conservativamente (veja a seção IV.1.3).

A fim de evidenciar a capacidade de especializar uma especificação, seja Σ_1 o TAD $Stack[Elem]$ descrito a seguir:

ADT $Stack[Elem]$

Syntax

Import $Elem$

Function $create : \longmapsto Stack;$

$push : Stack \times Elem \longmapsto Stack;$

$pop : Stack \longmapsto Stack;$

$top : Stack \longmapsto Elem$

Predicate $empty(Stack);$

Axiomatization

$$(\forall s : Stack)$$

$$s \stackrel{S}{=} create \vee (\exists s' : Stack)(\exists e : Elem) s \stackrel{S}{=} push(s', e) \quad (1)$$

$$empty(create) \quad (2)$$

$$(\forall s : Stack)(\forall e : Elem)\neg empty(push(s, e)) \quad (3)$$

$$(\forall s : Stack)(\exists s' : Stack)$$

$$\neg empty(s) \Rightarrow \{pop(s) \stackrel{S}{=} s' \Rightarrow (\exists e : Elem) push(s', e) \stackrel{S}{=} s \quad (4)$$

$$(\forall s : Stack)\neg empty(s) \Rightarrow (\exists e : Elem) top(s) \stackrel{E}{=} e \quad (5)$$

End

Pressupondo que seja vaga a noção a respeito de pilha, sua axiomatização foi feita notadamente bem liberal, demonstrando algumas incertezas quando da concepção inicial. Certos aspectos, por exemplo se a pilha tem ou não um limite físico para o seu crescimento, não foram determinados. Novamente, a implementação de $Stack$ será tentada sobre $Array[Indice, Elem]$, produzindo:

ADT $Stack^*[Elem]$

Syntax

Import $Array[Indice, Elem], Elem$

Function $create^* : \longmapsto Stack^*;$

$push^* : Stack^* \times Elem \longmapsto Stack^*;$

$pop^* : Stack^* \longmapsto Stack^*;$

$top^* : Stack^* \longmapsto Elem$

Predicate $empty^*(Stack^*);$

HidenSymbols $PointerArray$;

$down : PointerArray \longmapsto Indice$;

$up : PointerArray \longmapsto Indice$;

$array : PointerArray \longmapsto Array$;

$stack : PointerArray \longmapsto Stack^*$;

Axiomatization

$(\forall i, j, k : Indice)(\exists c : indice) c \stackrel{I}{=} position(i, j, k) \Leftrightarrow$

$\{[addone(j) \stackrel{I}{=} i \wedge addone(k) = c] \vee [c \stackrel{I}{=} position(i, addone(j), addone(k))]\}$ (1)

$(\forall p : PointerArray)(\exists a : Array)(\exists i, i' : Indice)$

$up(p) \stackrel{I}{=} i \ \wedge \ down(p) \stackrel{I}{=} i' \ \wedge \ array(p) \stackrel{A}{=} a$ (2)

$(\forall p, p' : PointerArray) \neg p \stackrel{PA}{=} p' \Rightarrow$

$\neg\{down(p) \stackrel{I}{=} down(p') \wedge up(p) \stackrel{I}{=} up(p') \wedge array(p) \stackrel{A}{=} array(p')\}$ (3)

$(\forall p, p' : PointerArray) stack(p) \stackrel{S}{=} stack(p') \Leftrightarrow$

$(\forall i : Indice)\{inside(i, down(p), up(p)) \Rightarrow$

$value(array(p), i) \stackrel{P}{=} value(array(p'), position(i, up(p), up(p')))\}$ (4)

$(\forall s : Stack^*)(\exists p : PointerArray) stack(p) \stackrel{S}{=} s$ (5)

$(\forall s : Stack^*) s \stackrel{S}{=} create^* \Leftrightarrow (\exists p : PointerArray)$

$\{stack(p) \stackrel{S}{=} s \ \wedge \ addone(up(p)) \stackrel{I}{=} down(p) \ \wedge \ array(p) \stackrel{A}{=} inicialize\}$ (6)

$(\forall s : Stack^*) empty^*(s) \Leftrightarrow$

$(\exists p : PointerArray) stack(p) \stackrel{S}{=} s \ \wedge \ addone(up(p)) \stackrel{I}{=} down(p)$ (7)

$(\forall s : Stack^*)(\forall e : Elem)(\exists s' : Stack^*) s' \stackrel{S}{=} push^*(s, e) \Leftrightarrow$

$(\exists p, p' : PointerArray)\{stack(p) \stackrel{S}{=} s \ \wedge \ stack(p') \stackrel{S}{=} s'\} \Rightarrow$

$\{up(p') \stackrel{I}{=} addone(up(p)) \ \wedge \ down(p') \stackrel{I}{=} down(p) \ \wedge$

$array(p') \stackrel{A}{=} attrib(array(p), addone(up(p)), e)\}$ (8)

$(\forall s : Stack^*)(\exists e : Elem) e \stackrel{E}{=} top^*(s) \Leftrightarrow$

$(\exists p : PointerArray) stack(p) \stackrel{S}{=} s \Rightarrow value(array(p), up(p)) \stackrel{E}{=} e$ (9)

$(\forall s : Stack^*)(\exists s' : Stack^*) s' \stackrel{S}{=} pop^*(s) \Leftrightarrow$

$(\exists p, p' : PointerArray)\{stack(p) \stackrel{S}{=} s \ \wedge \ stack(p') \stackrel{S}{=} s'\} \Rightarrow$

$\{addone(up(p')) \stackrel{I}{=} up(p) \ \wedge \ down(p') \stackrel{I}{=} down(p)$

$\wedge \ array(p') \stackrel{A}{=} array(p)\}$ (10)

End

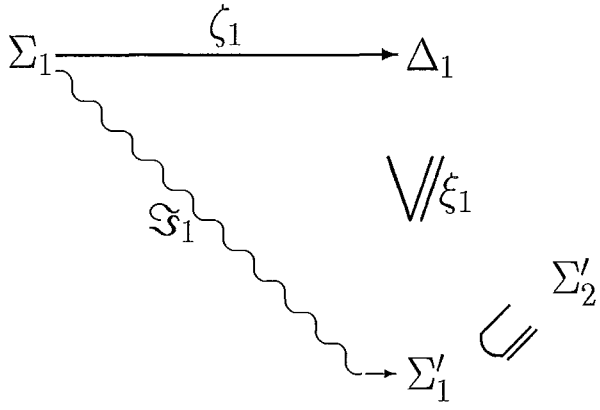
Eventualmente, foi decidido que em um certo ponto, associado a constante *maxpos* a pilha não mais se comporta normalmente, porque está impedida de aumentar. Isto implica que qualquer tentativa de empilhar um novo elemento será inócua.

Para conseguir esta propriedade não é preciso modificar a especificação do TAD *Stack*, o mesmo efeito é obtido através da escolha de uma representação adequada para a função *addone* [9] até então indefinida. Refinando-a, seja os axiomas :

$$(\forall i, j : Indice) \neg i \stackrel{I}{=} maxpos \Rightarrow i < addone(i)$$
$$addone(maxpos) \stackrel{I}{=} maxpos$$

introduzidos não conservativamente na especificação semântica de *Indice*.

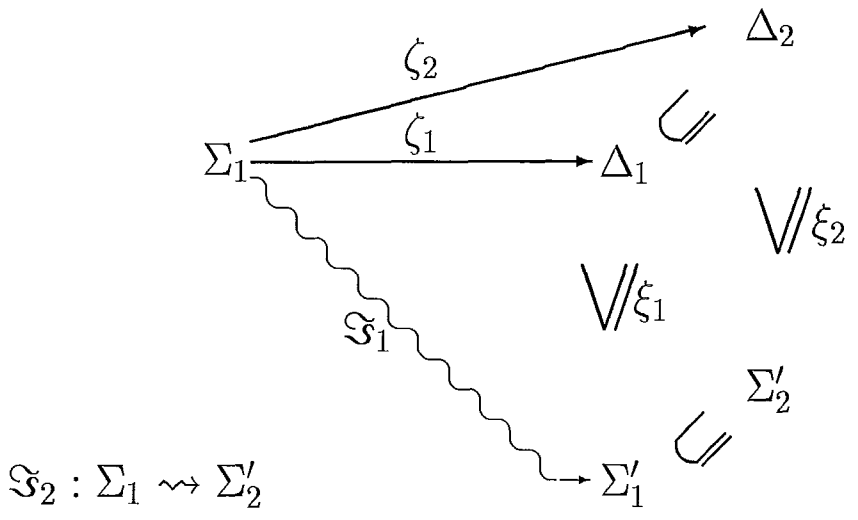
Genericamente, esta situação poderia ser representada melhor através do seguinte diagrama :



Se assim for feito, vale a aplicação do Teorema da Construtibilidade também nesta situação, como uma alternativa viável e especialmente adequada à limitação do Teorema da Modularização.

Portanto, espontaneamente aplicar-se-ia o mecanismo para compor um novo passo de implementação por sobre $\Im_1$, visando incorporar automaticamente as restrições descritas em Σ'_2, construindo a partir de Δ_1 uma nova implementação declarativa Δ_2, estendendo consistentemente a anterior, a fim de completar um novo passo canônico, conforme o diagrama a seguir.

[9] presumindo que a função *addone* já pertencesse a linguagem de *Indice*, definido na seção II.2



O novo passo canônico superposto $\Im_2$ permitiria deduzir em Δ_2, extensão de $Stack^*[Elem]$ algo como a propriedade:

$$(\exists s : Stack^*)(\forall e : Elem)s = push^*(s, e)$$

que indica que a pilha tem um limite de crescimento, embora nada afirme isto na especificação para $Stack[Elem]$. A adição dos axiomas definindo o significado de *addone* foi responsável pela especialização neste novo passo canônico superposto.

A implementação declarativa Δ_2 é certamente mais específica e mais restrita que Δ_1. Neste sentido, a implementação $\Im_1$ foi especializada, dando origem a $\Im_2$, pois, qualquer realização para a teoria especificada por Δ_1, que não satisfaça as novas restrições representadas através de axiomas em Σ'_2, não será realização para a implementação declarativa Δ_2. Ou seja, o conjunto de realizações, modelos para Δ_2, é subconjunto, geralmente próprio, do conjunto de realizações modelos para Δ_1. Isto comprova que apenas o domínio de pilhas com um limite de crescimento definido são capazes de induzir realizações para $Stack[Elem]$. Isto é, as realizações ilimitadas que não satisfazem as novas propriedades serão sumariamente descartadas com a especialização ocorrida em *Indice*.

Proposição IV.3.3 *Seja a composição de implementações conforme abaixo:*

$$\begin{array}{ccccc} & & & & \Delta_2 \\ & \overset{\zeta_2}{\nearrow} & & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 & & \Vdash \\ & & \Vdash & & \\ & & \Sigma'_1 & \subseteq & \Sigma'_2 \end{array}$$

Assumindo que $\mathcal{L}(\Sigma'_1) = \mathcal{L}(\Sigma'_2)$,

Se $\boldsymbol{Mod}\,(\Sigma_2') \subset \boldsymbol{Mod}\,(\Sigma_1')$ *então* $\boldsymbol{Mod}\,(\Sigma_1)^{\zeta_2^{-1}} \subset \boldsymbol{Mod}\,(\Sigma_1)^{\zeta_1^{-1}}$

Onde $\boldsymbol{Mod}\,(\Sigma_1)^{\zeta_i^{-1}}$ *denota a classe de modelos de* Σ_1*, induzidos por* ζ_i*.*

Prova :

1. por hipótese $\mathcal{L}(\Sigma_1') = \mathcal{L}(\Sigma_2')$, se $\mathbf{Mod}\,(\Sigma_2') \subset \mathbf{Mod}\,(\Sigma_1')$ então $\mathbf{Cn}(\Sigma_1') \subset \mathbf{Cn}(\Sigma_2')$, por definição, logo existe $\phi \in \mathcal{L}(\Sigma_1')$, tal que: $\Sigma_2' \models \phi$ e $\Sigma_1' \not\models \phi$
2. já que $\Sigma_1' \leq \Delta_1$, se $\Sigma_1' \not\models \phi$ então $\Delta_1 \not\models \phi$
3. por outro lado, $\Sigma_2' \leq \Delta_2$, se $\Sigma_2' \models \phi$ então $\Delta_2 \models \phi$
4. consequentemente, por 2 e 3, obtem-se que $\Delta_2 \models \phi$ e $\Delta_1 \not\models \phi$
5. então, por 4, $[\exists \aleph \in \mathbf{Mod}\,(\Delta_1)][\forall \Re \in \mathbf{Mod}\,(\Delta_2)] \setminus \Re|_{\mathcal{L}(\Delta_1)} \not\equiv \aleph$
6. visto que $\Sigma 1 \xrightarrow{\zeta_1} \Delta_1$ e $\Sigma 1 \xrightarrow{\zeta_2} \Delta_2$, então, tomando o modelo $\aleph^{\zeta_1^{-1}} \in \mathbf{Mod}\,(\Sigma_1)$, por definição, será tal que :

   i) $\aleph^{\zeta_1^{-1}} \in \mathbf{Mod}\,(\Sigma_1)^{\zeta_1}$, mas

   ii) $\aleph^{\zeta_1^{-1}} \notin \mathbf{Mod}\,(\Sigma_1)^{\zeta_2}$

   pelo Teorema da Tradução.
7. Finalemente, por conseqüência direta de 6, tem-se que $\mathbf{Mod}\,(\Sigma_1)^{\zeta_1^{-1}} \subset \mathbf{Mod}\,(\Sigma_1)^{\zeta_2^{-1}}$

□

A proposição IV.3.3 confirma que a redução no conjunto de realizações para Σ_1', consegue realmente restringir modelos induzidos em Σ_1, simplesmente descrevendo novas particularidades, em termos da linguagem primitiva, na verdade, o que condiz com o significado entendido convencionalmente, denotando a especialização de uma implementação. Desta forma, apesar que o conjunto de conseqüências da implementação declarativa haver sido estendido, em contra-partida o conjunto de suas possíveis realizações fica mais restrito. Mas Δ_2, construida segundo o Teorema da Construtibilidade, representa uma teoria consistente, portanto, admite sempre realizações, atendendo ao que se pretendia.

Especializar em o passo canônico $\Im_1$, quando Σ_1' já está implementado, através de uma série de passos subsequentes, desvia o sentido da implementação completamente, porque, depois da especialização, é Σ_2' quem deve ser implementado, e Σ_1' consequentemente descartado.

Como Σ'_1 ainda está contido em Σ'_2, era desejável que o trabalho já realizado não fosse simplesmente perdido. Porém, existem limitações teóricas, porque não é conservativa a extensão $\Sigma'_1 \subseteq \Sigma'_2$, não há meios viáveis para determinar se as propriedades incorporadas na especialização podem ser absorvidas, sem causar inconsistências nos passos subsequentes. Todavia, se a implementação declarativa no passo seguinte, Δ_3, ainda é capaz de suportar a interpretação de Σ'_2, seria suficiente apenas redefinir uma nova tradução, agora de Σ'_2 em Δ_3 (talvez a mesma, caso nenhum símbolo novo haja sido adicionado), de acordo com a figura IV.3.9.

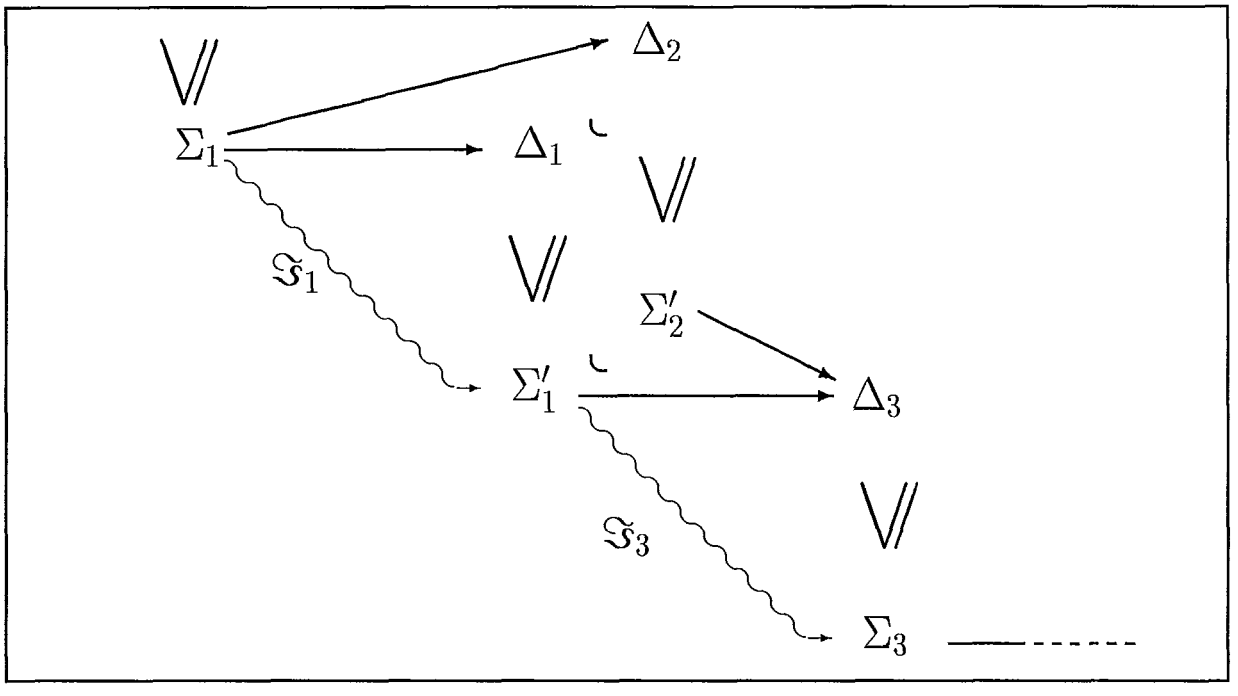


Figura IV.3.9: Propagação da Especialização para os Passos Subsequentes (I)

Claramente, os próximos passos de implementação permaneceriam inalterados, pois a evolução em ϑ_2 não tocou em Σ_3.

Falhando esta tentativa, uma outra implementação declarativa, Δ_4, necessitaria ser reconstruida, reestabelecendo a implementação em termos de Σ_3. Mas não há como compor algoritmicamente Δ_4, a partir de Σ'_2, Δ_3 e Σ_3, preservando ainda a consistência, porque o Teorema da Construtibilidade não se aplica a esta situação. Exclusivamente, é possível afirmar que a substituição de Δ_3 por Δ_4, definindo um outro passo de implementação de Σ'_2 em Σ_3, sempre soluciona o problema. E assim, conseqüências da evolução em ϑ_1 não interfeririam com os passos canônicos sucedendo a Σ_3. Conforme a figura seguinte.

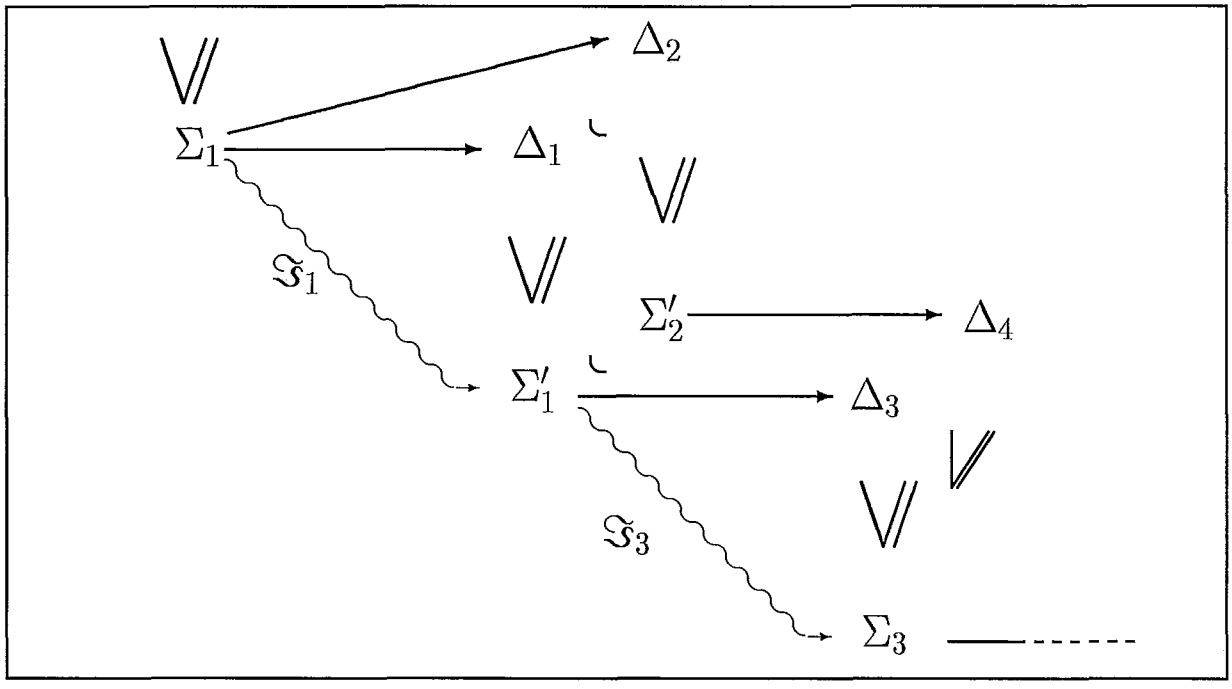


Figura IV.3.10: Propagação da Especialização para os Passos Subsequentes (II)

Já que, este passo de implementação ainda se dá sobre Σ_3, não há razões para interferir com o prosseguimento da implementação após este ponto. No exemplo mencionado na presente seção, tudo o que precisaria ser adicionalmente feito é tentar na construção de Δ_4 uma representação para a função *addone*. Ou seja, "Nem toda a cadeia precisaria ser refeita, porém apenas um elo substituido."

Deste modo, os danos conseqüências de uma especialização, extendem-se no máximo até o passo imediatamente seguinte. Isto está coerente, porque se as modificações em Σ'_2 estivessem relaciondas com Σ_3, deveriam estar expressas diretamente em sua linguagem. E assim minimiza-se os prejuizos causados por um evolução *a posteriori* na implementação.

Para concluir, tanto a ampliação quanto a especialização de implementações são ferramentas reconhecidamente úteis à prática de desenvolvimento de programas.

IV.3.2 Alterações

Nem sempre a evolução dá-se por extensão, requerendo meramente incorporar novos elementos. Talvez seja menos frequente, quando a etapa de apreensão da realidade foi meticulosa e o projeto conduzido formalmente, mas a evolução manifesta-se também pela necessidade de retirar algo, ou mesmo substituir. A substituição é precedida também pela

retirada de elementos pré-existentes. Logo, ambas foram generalizadas por "alteração", porque implicam em modificar o conjunto de conseqüências a partir de algum ponto determinado da implementação. Ou as circunstâncias iniciais já não são as mesmas, pois a realidade é por natureza dinâmica, e o exercício de previsão não foi bem sucedido, ou ainda erros foram inadvertidamente cometidos, importa que situações ideais usualmente não acontecem, e a formalização da implementação não deveria omitir-se neste processo de alteração.

Contudo, há limitações teóricas ainda mais crítcas que a conservatividade. Uma vez incluido um axioma na teoria, todas as suas conseqüências estão permanentemente incorporadas, e não existem mecanismos de primeira ordem para evitar sua dedução. Isto decorre da monotonicidade, própria da *Lógica da Primeira Ordem*, mas tem efeitos importantes sobre a implementação. Assim que uma propriedade é acrescentada a uma especificação, fruto de alguma decisão de refinamento ou de um mero detalhe julgado pertinente, incondicionalmente esta propriedade será conduzida, apenas por traduções, até a representação objetivo.

Quando surge a necessidade de alteração, não há altenativas para continuar, senão retroceder buscando a raiz do problema, isto é, descobrir onde as propriedades indesejáveis foram inicialmente colocadas, e antes deste ponto desviar a implementação para uma nova tentativa, revendo os passo subsequentes, conforme as alterações requeridas.

Obviamente, este retorno e revisão implica em perder parte do trabalho. Quanto menos contingências forem as alterações necessárias, mais consideráveis os efeitos e mais custoso repará-los. Uma situação extrema exigiria que toda a implementação fosse refeita.

Neste contexto, resta ao alcance somente procurar minimizar o impacto, determinando os passos canônicos subsequentes, penalizados com uma necessária reformulação, para absorver as alterações realizadas. Caso a implementação haja sido conduzida sistematicamente, tanto mais fácil tende a ser a revisão. É aí onde entra a estratégia de implementação construtiva.

Sendo mais preciso, a seqüência de extensões conservativas produzida pela etapa de simplificação (IV.1.1.1), poderia ser uma orientação adequada, pois cada subestrutura aí independe de todas as propriedades superpostas, logo qualquer alteração limitadas a estas propriedades não a influenciaria, sendo possível mantê-la da mesma forma. Assim, consegue-se delimitar quem precisa ser revisado, e quem poderia ser reaproveitado. Conservando inclusive o processo de verificação para as subestruturas subjacentes não

afetadas.

Novamente, faz-se útil decompor a especificação gerando subestruturas bastante simples e facilmente compreensíveis, restringindo a quantidade de novos elementos e propriedades em cada uma, porque em caso de alteração, a reutilização de partes da implementação acaba favorecida.

Este critério, além de relativamente simples, tem a vantagem de ser subproduto da própria implementação, logo não necessitaria ser redeterminado sempre que surgisse perspectivas de alteração. A reimplementação segue orientando-se por passos canônicos já construidos na implementação original.

Infelizmente, isto aparenta ser o máximo que pode ser obtido sem relegar o formalismo. Embora limitada no contexto presente, esta é a única abordagem viável para os problemas que envolvem alteração. A habilidade para lidar com situações parcialmente compreendidas também fica comprometida, bem como o tratamento de erros cometidos. Porém, existe uma maior flexibilidade para atacar este problema, como será visto adiante, em um outro contexto.

IV.4 Reusabilidade

Os recursos para composição de implementação trazem a possibilidade de ampliar a reutilização para níveis bem mais abstratos, permitindo além do reaproveito da representação final, e dispor de especificações e implementações já devidamente concluidas e certificadas.

Indubitavelmente, todo o processo de desenvolvimento é favorecido, pois:

- o tempo de desenvolvimento consegue ser reduzido;
- a confiabilidade dos resultados aumenta efetivamente;
- a probabilidade de erros de especificação ou implementação diminui sensivelmente;
- padrões organizacionais podem ser embutidos.

A reusabilidade deveria ser então, um objetivo a ter-se em mente, considerando um possível uso posterior para o presente objeto de implementação, e empregar o trabalho criativo anterior em outra implementação semelhante.

De um modo geral, é tanto menos provável encontrar especificações ou passos de implemetação adequadamente prontos para serem utilizados nos estágios iniciais do desenvolvimento, quanto tende a ser mais vantajosa a possibilidade desta reutilização em um nível mais conceitual. Justamente por isto, que a reusabilidade seria favorecida pela aplicação disciplinada dos recursos para compor implementações.

A composição de implementações atuaria basicamente de duas formas: promovendo a assimilação de implementeções previamente executadas e facilitando a reutilização de especificações, mesmo que já implementadas. A primeira forma esta diretamente relacionada à composição de passos canônico subsequentes, enquanto a última envolve a composição por sobre passos canônicos superpostos.

IV.4.1 Reusabilidade de Implementações

Sempre que um tipo abstrato de dados, especificado por Σ_1, encontrar implementação em termos de Σ_2, para o qual já foi determinada anteriormente uma representação final na linguagem objetivo, todo o processo de implementação desencadeado subsequente a Σ_2, poderia ser reutilizado, e por composição, seria definida algoritmicamente a representação pretendida para Σ_1. Para tanto, é suficiente que o passo canônico de Σ_1 em Σ_2 seja estabelecido e demonstrado, utilizando os mesmos lemas certificados pela própria implementação de Σ_2. Por conseguinte, também o processo correspondente de verificação é reaproveitado.

Exemplificando, se $PriorityQueue[Process]$ já estivesse inteiramente implementada em termos do TAD $Array[Indice, Process]$, antes mesmo que a implementação do TAD $Dispatcher[Process]$ sequer fosse pensada, todo o desenvolvimento subsequente a $PriorityQueue[Process]$ será automaticamente aproveitado, inclusive o processo de verificação realizado, quando o passo $Dispatcher[Process] \rightsquigarrow PriorityQueue[Process]$ for estabelecido.

Assim, a perspectiva de reusabilidade inclina-se para níveis conceitualmente mais abstratos, isto é, primariamente relacionados com a descrição original. E quanto mais próximo do domínio do problema sucede a reutilização, melhores os resultados obtidos e maior a viabilidade econômica desta opção. Tudo porque a menor presença de detalhes desnecessários facilita decidir o que se poderia valer de trabalhos realizados no passado, e diminui a tendência de particularidades irrelevantes em Σ_2 virem a comprometer sua reutilização.

IV.4.2 Reusabilidade de Especificações

A composição com implementações subjacentes presta-se à reutilização simplesmente permitindo que a uma especificação, por exemplo Σ_1, sejam incorporados novos componentes que faltam ainda para uma representação aceitável de Σ_2, ou seja, a especificação de Σ_1 é conservada e serviria como base para projetar uma especificação, Σ_2, mais elaborada. Supondo que Σ_1 estivesse implementada em Σ_1', restaria apenas tentar encontrar alguma extensão de Σ_1' (se é que Σ_1' já não fosse capaz de suportar a implementação de Σ_2). Em qualquer caso, o Teorema da Construtibilidade apresentaria pelo menos um implementação candidata para Δ_2.

Caso Σ_1' não seja exatamente adequado para a implementação de Σ_2, é sensato antes fazer opção por reutilizar a especificação de Σ_1, determinar se compensa o esforço para contornar a reduzida expressividade de Σ_1', no lugar de particularizar um outra especificação inteiramente nova, e implementar Σ_2 sobre este mecanismo lingüístico mais expresssivo.

Há ainda outra possibilidade de reutilizar uma especificação construida. Retornando ao exemplo de $SStruct[Key, Value]$ (na seção IV.3.1.1), suponha que já existisse a especificação para uma árvore binária, mas sem mencionar a propriedade de ordenação, ou seja, os axiomas,

$$(\forall b : BinTree)\neg isnull(left(b)) \Rightarrow label(root(b)) \lhd label(root(left(b)))$$
$$(\forall b : BinTree)\neg isnull(right(b)) \Rightarrow label(root(b)) \rhd label(root(right(b)))$$

que definem que o filho à esquerda é "menor" que a raiz, enquanto o filho à direita é sempre "maior". Esta especificação pode haver sido especializada por uma extensão não conservativa visando capacitá-la a suportar a implementação de $SStruct$.

A reutilização portanto, pode valer-se da tanto da especificação de Σ_1 quanto da Σ_1', com a diferença que a primeira só poderia ser adaptada por uma extensão conservativa, ao passo que, a mais primitiva aceita uma especialização não conservativa.

IV.5 Parametrização e Instanciação

Um mecanismo de reconhecida eficiência no *Desenvolvimento com Tipos Abstratos de Dados*, é a capacidade de definí-los como parametrizados, e utilizá-los oportunamente, pela simples especialização de seus paramêtros.

Tipos Abstratos de Dados parametrizados conjugam as possibilidades de reutilizar tanto uma especificação quanto a sua própria implementação. Proporcionando, assim, o conforto de ter disponíveis conceitos genéricos, *a priori* especificados, implementados e verificados, ou seja, inteiramente prontos para serem empregados, sempre que a instanciação de seus parâmetros os faça aplicáveis a uma situação específica.

A parte mais difícil é, na realidade, identificar propriedades comuns em uma determinada classe de conceitos com padrões semelhantes, e decidir generalizá-los sob forma de um TAD parametrizado, antevendo já um reuso futuro do TAD resultante.

O principal atrativo deste mecanismo de generalização–especialização reside mesmo em sua simplicidade conceitual. Representar classes de conceitos de modo parametrizado propicia uma compreensão mais clara, enfatizando as caraterísticas compartilhadas e destacando as particularidades que distinguem cada instância da classe. Além disto, este mecanismo estimula e reforça a disciplina na estruturação das propriedades representadas, facilitando tanto a especificação quanto a posterior implementação.

Antes, é preciso definir formalmente como proceder a instanciação de parâmetros.

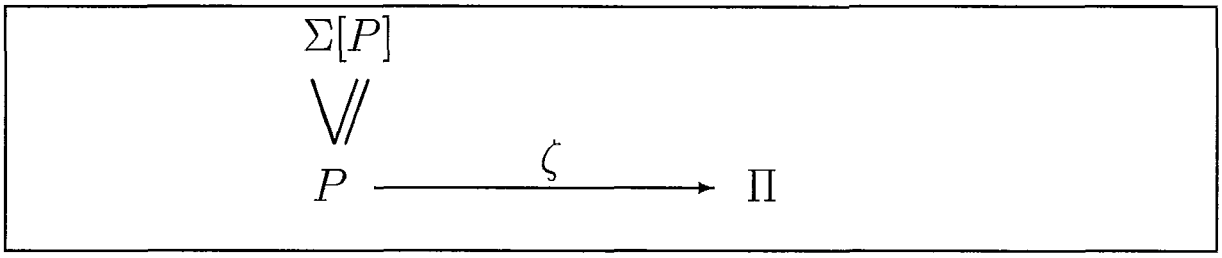
IV.5.1 Instanciação

Em linhas gerais, a especialização de uma instância dá-se por interpretação dos parâmetros formais em parâmetros reais, e a extensão deste resultados para a especificação do TAD parametrizado, conforme [Vel92c].

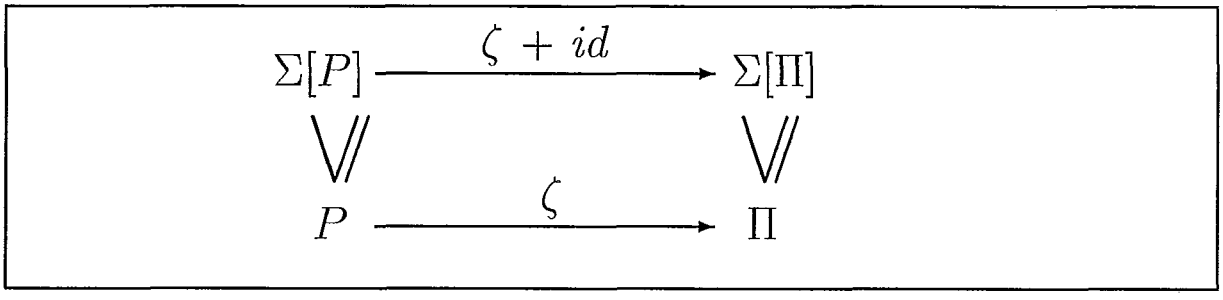
Um parâmetro formal P, pode ser constituido além de sortes, por operações sobre estes sortes, principalmente predicados para comparar instâncias. Logo, faz sentido também especificar P através de um TAD, descrevendo as propriedades necessárias a todas as instâncias da classe. A semântica deste TAD é que define quem poderia ser instanciado ou não. Por este motivo, as linguagens de programação usualmente obrigam a associação de um *tipo* a cada parâmetro.

Um parâmetro real Π é, por analogia, um TAD que satisfaz pelo menos as propriedades para completar uma instância válida, segundo alguma função de tradução. Inclusive, Π pode ainda ser um TAD parametrizado, neste caso representando não apenas uma instância única, mas provavelmente toda uma subclasse dos conceitos. Esta linha de racíocinio pode ser aplicada recursivamente, estruturando a especificação para uma instância singular.

A instanciação de um TAD parametrizado, $\Sigma[P]$, por Π pode ser melhor entendida através do diagrama,



Pelo Teorema da Modularização, conforme [Vel92c], é possível automaticamente completar,



Como poderia ter acontecido, por instância, com $BinTree[Item]$,

$$\begin{array}{ccc} BinTree[Elem] & \xrightarrow{\zeta+id} & BinTree[Item] \\ \Vdash & & \Vdash \\ Elem & \xrightarrow{\zeta} & Item \end{array}$$

antes de sua utlização para receber a implementação de $SStruct$.

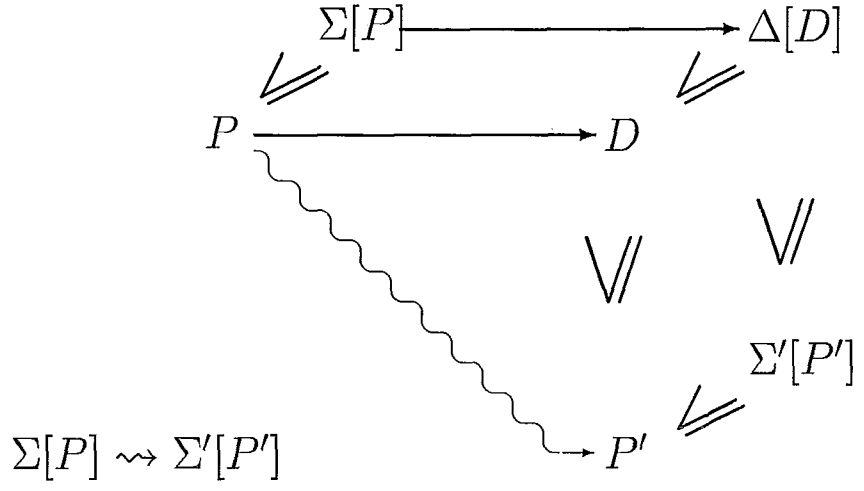
IV.5.2 Implementação de TAD Parametrizados

Uma vez adotado este procedimento para a instanciação, resta ver a implementação de Tipos Abastratos de Dados envolvendo parâmetros. Há três casos de real interesse a serem observados:

- a implementação de um TAD parametrizado mas não instanciado em um outro igualmente não instanciado (e.g. $Queue[Elem] \rightsquigarrow Sequence[Elem]$);
- a implementação de um TAD parametrizado e instanciado em um outro igualmente instanciado; (e.g. $Buffer[Process] \rightsquigarrow Array[Indice, Process]$);

- a implementação de um TAD não parametrizado em termos de um outro parametrizado e instanciado.

Nos dois primeiros casos fica mais evidente a composição de um passo de implementação entre os Tipos Abstratos de Dados parametrizados, por sobre um passo canônico definido entre os parâmetros, formais ou reais, respectivamente. De acordo com o diagrama,



Se P e P' tratam-se meramente de parâmetros formais, confirma-se a viabilidade de implementar completamente um TAD parametrizado, determinando sua representação final, antes de instanciá-lo ou mesmo utilizá-lo. Isto é que viabiliza a reutilização dos processos de implementação e verificação realizados especificamente para $\Sigma[P]$, e permite preparar TAD's genéricos, como $\Sigma[P]$, para uma eventual aplicação futura.

Neste caso, por questão de simplicidade não há problemas em assumir que P e P' são iguais (a mesma linguagem e axiomas), pois isto facilita a implementação de TAD parametrizados, permitindo a abstração do comportamente de possíveis instâncias. O passo canônico subjacente poderia ser considerado trivial, pois $P \rightsquigarrow P$, tendo P como implementação declarativa, tal como foram realizados todos os exemplos aqui que não envolveram a implementação de parâmetros.

Mesmo que P e P' sejam parâmetros reais, a implementação pode prosseguir normalmente. Portanto, a instanciação dos parâmetros não altera a continuidade do desenvolvimento.

O terceiro caso assimila comparavelmente o mesmo raciocínio, porque de forma mais sutil, envolve também um passo de implementação subjacente. Ou seja, quando um

TAD não parametrizado Σ houver de ser implementado em um TAD parametrizado $\Sigma'[\Pi]$, permaneceria escondida uma implementação entre uma parte de Σ e o parâmetro real Π.

Esta situação será exemplificada através da implementação para *String*, cuja a especificação encontra-se a seguir,

ADT *String*

Syntax

Import *Integer*

Function λ : $\longmapsto String$;

$concat$: $String \times String \longmapsto String$;

$lenght$: $String \longmapsto Integer$;

Predicate $substring(String)$;

Axiomatization

$$(\forall s : String)s \stackrel{S}{=} \lambda \vee (\exists s', s'' : String)s \stackrel{S}{=} concat(s', s'') \quad (1)$$

$$(\forall s : String)concat(s, \lambda) \stackrel{S}{=} s \quad (2)$$

$$(\forall s, s', s'' : String)$$

$$concat(s, concat(s', s'')) \stackrel{S}{=} concat(concat(s, s'), s'') \quad (3)$$

$$(\forall s : String)lenght(\lambda) \stackrel{N}{=} 0 \quad (4)$$

$$(\forall s, s' : String)$$

$$lenght(concat(s, s')) \stackrel{N}{=} lenght(s) + lenght(s') \quad (5)$$

$$(\forall s : String)substring(s, \lambda) \quad (6)$$

$$(\forall s : String)\neg substring(\lambda, s) \quad (7)$$

$$(\forall s, s' : String)$$

$$substring(s, s') \Leftrightarrow (\exists r, r' : String)s \stackrel{S}{=} concat(concat(r, s'), r') \quad (8)$$

End

Visando implementar *String*, o mesmo TAD $Sequence[Elem]$ será instanciado, substituido o parâmetro formal $Elem$ por um paramêntro real $Char$, onde $Char$ é formado por vinte e seis constantes, denotando as letras do alfabeto.

O passo canônico $String \rightsquigarrow Sequence[Char]$, resultaria possivelmente em uma implementação declarativa:

ADT $String^*[Elem]$

Syntax

Import $Sequence[Elem], Integer$

Function $\lambda^* : \longmapsto String^*;$

$concat^* : String^* \times String^* \longmapsto String^*;$

$lenght^* : String^* \longmapsto Integer;$

Predicate $substring^*(String^*);$

HidenSymbols $compare(String^*);$

Axiomatization

$$(\forall s : String^*)(\exists q : Sequence)s \stackrel{S}{=} q \tag{1}$$

$$\lambda^* \stackrel{S}{=} null \tag{2}$$

$$(\forall s : String^*)concat^*(null, s) =_S s \tag{3}$$

$$(\forall s, s' : String^*)concat^*(s, s') =_S insere(head(s), concat^*(tail(s), s')) \tag{4}$$

$$(\forall s : String^*)isnull(s) \Leftrightarrow lenght^*(s) \stackrel{N}{=} 0 \tag{5}$$

$$(\forall s : String^*)$$

$$\neg isnull(s) \Leftrightarrow lenght^*(s) \stackrel{N}{=} 1 + lenght^*(tail(p)) \tag{6}$$

$$(\forall s : Sequence)compare(s, null) \tag{7}$$

$$(\forall s, s' : Sequence)$$

$$compare(s, s') \Leftrightarrow head(s) \stackrel{C}{=} head(s') \ \wedge \ compare(tail(s), tail(s')) \tag{8}$$

$$(\forall s : String^*)substring^*(s, \lambda^*) \tag{9}$$

$$(\forall s : String^*)\neg substring^*(\lambda^*, s) \tag{10}$$

$$(\forall s, s' : String^*)$$

$$substring^*(s, s') \Leftrightarrow compare(s, s') \ \vee \ substring^*(tail(s), s') \tag{11}$$

End

O refinamento para a função *substring* é definida em termos de uma operação, a igualdade ($\stackrel{C}{=}$), definida sobre o TAD $Char$, também a representação escolhida para *concat* envolve uma operação entre seqüências e caracteres, a função *insere*. Mesmo a função *lenght* faz suposições sobre caracteres, que cada um formaria uma *String* de tamanho unitário. Isto significa que as funções do repertório de *String*, encontram seu refinamento em termos de propriedades de $Char$.

IV.6 Referências Bibliográficas

Estratégias estilo *Divisão e Conquista* foram alvo de interesse inicialmente em [Vel80] e depois em [VV81] e [Vel83], e daí foi retirada a idéia de construir um passo de implementação, simplificando e depois recombinado os resultados obtidos, conforme foi descrito na seção IV.1. A verificação da corretude de um passo canônico espelha-se em [Vel87], adaptada à implementação construtiva.

Os problemas que advêm da tentativa de liberar a conservatividade foram mencionados em [TM87], porém na seção IV.3.1.2, espera-se, há uma solução satisfatória para dar continuidade ao desenvolvimento, sem esquecer o formalismo, mesmo quando a necessidade de evoluir manifesta-se de forma não conservativa. A reusabilidade de especificações e/ou implementações é estimulada em [Som89], e contemplada na seção IV.4. Inicialmente em [Vel87], Tipos Abstratos de Dados parametrizados são especificados e utilizados no desenvolvimento de programas, e depois em [Vel92c], a instanciação de paramêtros é formalizada através do Teorema da Modularização.

Capítulo V

Composição de Implementações no Contexto não Monotônico

Toda a argumentação até o presente momento apoia-se no formalismo lógico de primeira ordem, e mesmo assim, foram conseguidos meios para contornar a restrição imposta pela conservatividade, sempre quando havia vantagens. Mas a implementação ainda ressentia-se das limitações decorrentes da monotonicidade clássica, e alguns problemas identificados antes, nunca seriam adequadamente tratados naquele contexo.

Embora a monotonicidade seja útil em inúmeras situações, há circunstâncias em que se torna um empecilho, sendo melhor simplesmente deixá-la de lado. Consequentemente, seria necessário mudar o contexto, para abordar determinadas circunstâncias formalmente.

Este capítulo tem a pretensão de ultrapassar mais um limite, liberando a monotonicidade, transpondo para a *Lógica Default* todos os mecanismos clássicos de implementação e composição vistos até aqui. Havendo-os redefinidos em um contexto essencialmente não monotônico, problemas relegados anteriormente estão prontos para uma abordagem disciplinada.

A primeira seção reapresenta os correspondentes em *Lógica Default* para as noções clássicas de extensão e interpretação, e a última reestabelece especificamente a definição do passo canônico, e os Teoremas da Modularização e da Construtibilidade para o contexto default.

V.1 Conceitos Básicos

A fim de transpor apropriadamente para o contexto não monotônico a argumentação sobre composição de implementações, será preciso antes escolher algum formalismo capaz de suportar os conceitos de interpretação e extensão conservativa entre teorias. Estes conceitos devem ser redefinidos, dentro do padrão de raciocínio não monotônico, mas ainda conservando uma certa coerência com suas contra-partidas na lógica clássica (conforme A.2.4, A.3.2).

No presente trabalho, entre os formalismos não monotônicos conhecidos na literatura, foi preferida a *Lógica Default* (apresentada inicialmente em [Rei80]). Uma sorte de fatores apoia esta opção:

- a viabilidade de estabelecer definições correspondentes aos conceitos clássicos de interpretação e extensão conservativa, de forma assemelhada e bem intuitiva;
- a simplicidade conceitual, que torna mais fácil a compreensão dos mecanismos de composição de implementações, a serem aqui definidos, permitindo correlacionar o significado intuitivo às representações;
- uma representação sintática bem adequada, favorecendo a construção de especificações para Tipos Abstratos de Dados;
- a perfeita distinção entre fatos irrefutáveis (axiomas) e informações sujeitas a futura confirmação ou revisão, possibilitando uma mais natural transição entre o contexto monotônico e o não monotônico. Portanto, a adaptação a este novo contexto tende a requerer um esforço menor;
- ampla divulgação na literatura, que faz a *Lógica Default* um entre os formalismo não monotônicos mais estudados e melhor explorados;
- a inspiração de um trabalho anterior, [AV93a],[1] que já antecipavam uma possível tradução para o passo canônico e o Teorema da Modularização nos termos da *Lógica Default,* confirmando a conveniência de utilizar formalismos não monotônicos para solucionar alguns problemas relativos ao desenvolvimento de programas.

[1] veja V.3

Foge completamente ao escopo deste trabalho qualquer revisionismo sobre a *Lógica Default.* Consegue-se melhores e mais detalhadas informações consultando as referências conhecidas ou aqui citadas, entretanto o apêndice B enumera sucintamente as definições e os teoremas relevantes ao presente estudo, observando os resultados estabelecidos em [Rei80] e [RC83]. Escolher a proposta original de Reiter constitue mais uma vantagem, posto que quaisquer entre os resultados a serem obtidos poderiam, em princípio, ser adaptados às diversas variações da proposição original para a *Lógica Default,* que se propagam na literatura.

Pretendendo uma melhor conveniência em relação à especificação de TAD, a *Lógica Default* tem sua sintaxe levemente modificada, considerando sentenças agora sobre uma linguagem poli-sortida também de primeira ordem, onde antes não havia nenhuma consideração explícita sobre sortes.

Obviamente esta adaptação é de caráter meramente sintático, e só favorece a construção de especificações, e de forma alguma compromete os resultados já estabelecidos para a *Lógica Default.*

Especificamente, isto requer introduzir na linguagem a noção de sortes, associando a variáveis quantificadas, predicados e funções, domínios bem definidos, portanto, os pré-requesitos, justificativas e consequentes de qualquer default, tratados deste ponto em diante, serão sentenças de primeira ordem poli-sortidas. As teorias default apresentam como axiomas sentenças poli-sortidas e um conjunto de defaults assim constituido. Por consqüências, as conclusões geradas por qualquer extensão default, destas teorias default, formam igualmente conjuntos de sentenças poli-sortidas de primeira ordem. Estas três noções elementares fundamentam a *Lógica Default,* por assim dizer "Poli-Sortida".

Isto posto, é tempo de transpor para a *Lógica Default* os mecanismos de interpretação e extensão conservativa, e só então será possível definir formalmente a noção de passo canônico e estabelecer os Teoremas da Modularização e da Construtibilidade, determinam a composição de implementações.

V.1.1 Interpretação

Formalizar o conceito de interpretação entre teorias *default* exige, de algum modo, uma coerência com a noção intuitiva de tradução (por exemplo, no contexto clássico isto está enunciado através do Teorema da Tradução A.2.1). Portanto, estabelecer uma definição

realmente satisfatória envolve também encontrar alguma proposição, capaz de fornecer o significado preciso para este conceito, definindo o critério que decide quando se verifica ou não uma interpretação.

Postulando uma semântica para a definição, uma primeira tentativa, seria afirmar que uma teoria *default*, $\Sigma = \langle D_1, W_1 \rangle$, seria interpretada em termos de uma outra, $\Delta = \langle D_2, W_2 \rangle$, dada alguma função de tradução ζ, entre suas linguagens, se e somente se cada modelo para Δ induzisse, em sentido oposto, um modelo para Σ, segundo ζ.

Entretanto, antes de divagar entre considerações mais aprofundadas sobre possíveis semânticas para a *Lógica Default,* determinadas observações de Reiter parecem fazer sentido nestas circunstâncias. A idéia seria ver os *defaults* na teoria *default* Δ como restringindo a classe de possíveis realizações para o conjunto de axiomas W_2, de tal modo que:

> Se $\mathcal{E}$ é uma extensão *default* para Δ, existe alguma classe de modelos para W_2, a qual é exatamente a classe de todos os modelos para $\mathcal{E}$.

Esta visão semântica de extensões *default* é bem mais simples e talvez servisse como base na formalização do significado de interpretação em termos de *Lógica Default,* dentro dos propósitos específicos de implementação.

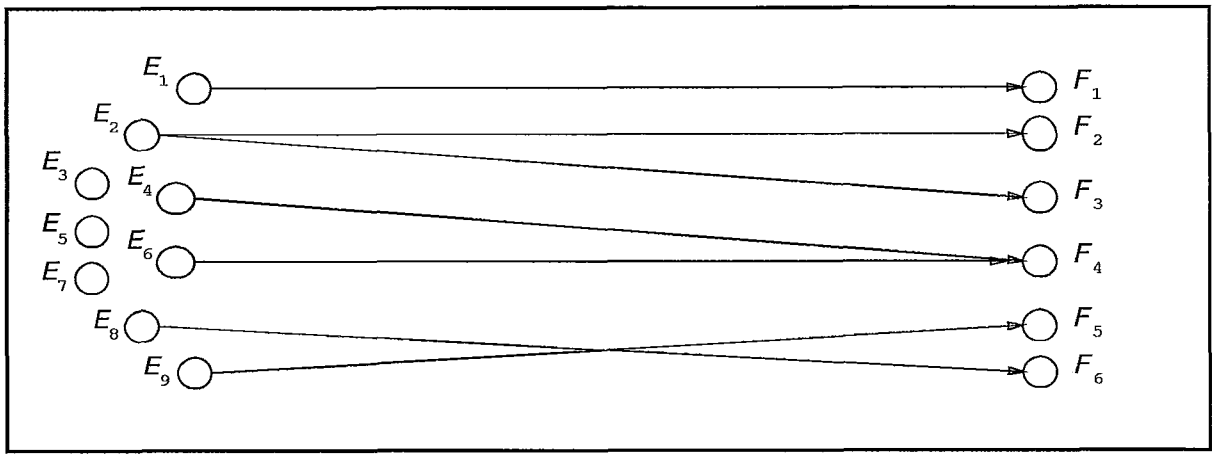
Surge naturalmente como alternativa, considerar o Teorema da Interpretação válido entre os conjuntos de axiomas W_1 e W_2, ou seja, cada possível realização para W_2 induziriam um modelo correspondente para W_1, segundo ζ, então haveria um critério necessário e suficiente para definir uma interpretação de Σ em Δ, retendo apenas à classe de modelos para W_2.

Todavia, o Teorema da Interpretação assim estabelecido ater-se-ia tão somente a parte monotônica (os axiomas) de ambas as teorias *default*, e consequentemente, as conclusões derivadas através da parte não monotônica (os *defaults*) seriam relevadas, em qualquer definição de interpretação assim fundamentada. Melhor dizendo, para decidir a interpretação entre teorias *defaults*, não importaria quais informações suas extensões *default* estariam supondo em relação às classes de modelos dos axiomas, quais conclusões representadas poderiam ser consideradas coerentes, porque exclusivamente a parte monotônica seria observada.

Logo, teorias *default* que diferem apenas por seu conjunto de *defaults*, apresentando os mesmos axiomas (a menos de isomorfismo), seriam sempre inter-permutáveis,

uma poderia substituir a outra meramente desconsiderando os respectivos *defaults*. Inclusive, teorias *default* que não apresentam extensões *default* poderiam ser normalmente consideradas. Então qual a justificativa para utilizar um formalismo não monotônico e continuar restrito aos mecanismos intrinsecamente monotônicos ?

Porém, sendo um pouco mais liberal, é razoável limitar a atenção àqueles modelos de W_2 que forem precisamente identificados em extensões *default* de Δ, de modo que somente estes modelos necessariamente induziriam modelos correspondentes para W_1, ou mais precisamente para as extensões *default* de Σ.



Na figura acima, por exemplo, seja $\{E_1, E_2, \ldots, E_9\}$ o conjunto de extensões *default* da teoria *default* Σ que está sendo interpretada em uma outra teoria *defautl* Δ, que gera o conjunto de extensões *default* $\{F_1, F_2, \ldots, F_9\}$. Neste exemplo, tanto F_2 quanto F_3 representam legítimas traduções de E_2, enquanto E_4 e E_6 podem ser interpretados na mesma extensão *default* F_4. Deste modo, a interpretação pode apontar mais de uma forma de traduzir as crenças descritas por uma única extensão *default*, como também poderia ocasionalmente resumir a interpretação de algumas extensões *default* distintas segundo um só conjunto de crenças.

Apenas é necessário que a seja total a relação[2] entre os respectivos conjuntos de extensões *default* a fim de associar cada modelo de W_2 determinado por uma extensão *default* F_i a um modelo induzido para uma extensão *default* F_j, e consequentemente a um modelo para W_1.

[2]**Relação Total :** uma relação r ($\mathbf{r} \subseteq A \times B$) é dita total se todo elemento de A está relacionado a um elemento de B. Ou seja, vale $(\forall x \in A)(\exists y \in b)[\langle x, y \rangle \in r]$.

Obviamente, esta proposição é mais permissiva que a versão inicial para o Teorema da Interpretação, uma vez que nem todos os modelos de W_2 induzem efetivamente modelos para W_1. Seria considerar apenas as classes de modelos sobre as quais Δ é capaz de inferir conlusões, através de seus *defaults*. Conciliar a noção tradicional de interpretação com a capacidade de raciocinar sobre hipóteses, que estão representadas nas extensões *default*, não é sem sentido, pois esta é motivação principal para justificar a utilização de lógicas não monotônicas.

Esta formulação pode ser reestabelecida sintaticamente, de um modo bem mais simples, considerando interpretações entre as extensões *default* de Σ e Δ, conforme abaixo:

Definição V.1.1 (Interpretação) *Considere as teorias default* Σ *e* Δ. *Uma função de tradução* ζ *da linguagem de* Σ *sobre a linguagem de* Δ ($\zeta : \mathcal{L}(\Sigma) \longmapsto \mathcal{L}(\Delta)$) *é uma interpretação de* Σ *em* Δ, *denotada por* $\Sigma \xrightarrow{\zeta} \Delta$, *se e somente se existe uma relação total* r *de* $ExtDef(\Delta)$ *em* $ExtDef(\Sigma)$, *(isto é,* $r \subseteq ExtDef(\Delta) \times ExtDef(\Sigma)$*)*[3], *de tal forma que:*

$$(\forall\, \mathcal{C} \in ExtDef(\Delta))(\exists\, \mathcal{B} \in ExtDef(\Sigma))\ \ \mathcal{C}\ r\ \mathcal{B}\ e\ \mathcal{B} \xrightarrow{\zeta} \mathcal{C}$$

A relação entre os conjuntos de extensões *default* foi condicionada como total, porque não há genericamente, necessidade de exigir para $\mathbf{r}$ mais restrições, e a totalidade aparenta ser a condição mínima, mais flexível, que pode ser aceita para estabelecer a interpretação.

Quando a relação $\mathbf{r}$ é total, não implica obrigatoriamente que todos os modelos para os axiomas de Δ induzam realizações para os axiomas de Σ, pois a definição agora colocada, apenas refere-se as classes de modelos que estão caracterizadas por extensões *default* de Δ, e nada informa sobre outros modelos possíveis para W_1. Entretanto, caso nem seja total a relação $\mathbf{r}$, há como afirmar que existem modelos para W_2 que nunca induziriam realizações para W_1. Neste caso a "definição" seria ainda mais liberal, contudo, seria intuitivamente permissiva em demasia, porque permite a perda de algumas informações a respeito da teoria Δ, não absorvidas na tradução de Σ.

Finalmente, considere uma versão para o Teorema da Interpretação, consoante a definição proposta acima.

[3]onde **ExtDef(A)** representa o conjunto de todas as extensões *default* geradas por uma teoria *default* A. Se A não tem extensões *default* **ExtDef(A)** é obviamente vazio.

Teorema V.1.1 (Teorema da Interpretação (*Lógica Default*)) *Considere duas teorias* default $\Sigma = \langle W_1, D_1 \rangle$ *e* $\Delta = \langle W_2, D_2 \rangle$. *A função de tradução* ζ *entre as linguagens* $\mathcal{L}(\Sigma)$ *e* $\mathcal{L}(\Delta)$ *constitue uma interpretação, denotada por* $\Sigma \xrightarrow{\zeta} \Delta$, *com a relação total subjacente* $\boldsymbol{r} \subseteq ExtDef(\Delta) \times ExtDef(\Sigma)$, *se e somente se para qualquer extensão* default $\mathcal{C} \in ExtDef(\Delta)$ *tal que* $\mathcal{C}\ \boldsymbol{r}\ \mathcal{B}$ *vale:*

i) *Se* $\mathcal{B} \models \alpha$ *então* $\mathcal{C} \models \alpha^{\zeta}$

ii) *Se* $\Re \in \boldsymbol{Mod}(\mathcal{C})$ *então* $\Re^{\zeta^{-1}} \in \boldsymbol{Mod}(\mathcal{C})$

Prova : Diretamente, pela versão clássica do Teorema da Interpretação (A.2.2), para as teorias $\mathcal{C}$ e $\mathcal{B}$ vale,

i) Se $\mathcal{B} \models \alpha$ então $\mathcal{C} \models \alpha^{\zeta}$

ii) Se $\Re \in \mathbf{Mod}(\mathcal{C})$ então $\Re^{\zeta^{-1}} \in \mathbf{Mod}(\mathcal{B})$

Se e somente se,

$$\mathcal{B} \xrightarrow{\zeta} \mathcal{C}$$

Portanto, se o argumento vale para qualquer extensão *default* $\mathcal{C}$ em $ExtDef(\Delta)$ então é válido para todas, atendendo a definição de interpretação V.1.1. □

Ao invés de estabelecer o Teorema da Interpretação em termos da parte monotônica de cada teoria *default*, preferivelmente este será válido entre os pares de extensões *default* relacionados segundo a definição de interpretação entre teorias *default*.

Em algumas situações, a relação $\mathbf{r}$ comporta-se de forma ainda mais restritiva, por exemplo, quando está envolvida uma teoria *default* normal, conforme a proposição a seguir.

Proposição V.1.1 *Seja* Σ *uma teoria* default *normal e* Δ *uma teoria* default *consistente, tais que* $\Sigma \xrightarrow{\zeta} \Delta$, *e considere* $\boldsymbol{r} : ExtDef(\Delta) \times ExtDef(\Sigma)$ *a relação total que estabelece a interpretação. Então vale para* $\boldsymbol{r}$ *a unicidade,*

$$(\forall \mathcal{E}'_1, \mathcal{E}'_2 \in ExtDef(\Delta))(\forall \mathcal{E}_1, \mathcal{E}_2 \in ExtDef(\Sigma))$$
$$\mathcal{E}'_1\ \boldsymbol{r}\ \mathcal{E}_1 \ \wedge\ \mathcal{E}'_2\ \boldsymbol{r}\ \mathcal{E}_2 \ \wedge\ \mathcal{E}_1 \neq \mathcal{E}_2 \ \Rightarrow\ \mathcal{E}'_1 \neq \mathcal{E}'_2$$

ou seja, $\boldsymbol{r}$ *é uma função.*

Prova : Sejam $\mathcal{E}_1$ e $\mathcal{E}_2$ duas extensões *default* distintas para Σ. Por contradição, suponha que $\mathcal{E}'_1 = \mathcal{E}'_2$. Por definição da relação **r** (V.1.1):

i) $\mathcal{E}_1 \xrightarrow{\zeta} \mathcal{E}'_1$

ii) $\mathcal{E}_2 \xrightarrow{\zeta} \mathcal{E}'_1$ (pois $\mathcal{E}'_2 = \mathcal{E}'_1$)

mas pela ortogonalidade (B.3.6) característica de teorias *default* normais, existe alguma sentença $\alpha \in \mathcal{L}(\Sigma)$, tal que:

$$\mathcal{E}_1 \models \alpha \text{ e } \mathcal{E}_2 \models \neg\alpha$$

consequentemente, por i) e ii),

$$\mathcal{E}'_1 \models \alpha \wedge \neg\alpha$$

logo, a extensão *default* $\mathcal{E}'_1$ (é inconsistente, e pelo teorema B.3.2, é a única extensão *default* gerada por Δ, consequentemente, a incosistência da teoria *default* Δ contradiz a suposição inicial. Portanto, vale necessariamente a unicidade para **r** neste caso. □

A ortogonalidade de Σ força deste modo, que a relação total estabelecendo a interpretação seja adicionalmente uma função, a menos que Δ seja uma teoria default inconsistente.

Caso permaneçam dúvidas quanto o sentido desta argumentada liberalidade, convém ressaltar que o objetivo central em utilizar a *Lógica Default*, é explorar suposições coerentes que podem ser presumidas sobre os modelos do conjunto de axiomas e assim caracterizando-os, identificar quais são os mais promissores para prosseguir o desenvolvimento. No sentido de representar escolhas ou decisões, tal formulação é perfeitamente razoável, conforme será visto no próximo capítulo, e até desejável.

V.1.2 Extensões

Uma extensão entre teorias é meramente uma interpretação, onde a função tradução entre linguagens é exatamente a identidade. Por conseguinte, qualquer formalização satisfatória para o conceito de extensão obrigatoriamente precisa refletir este fato. Logo, a definição a seguir será bem semelhante a V.1.1.

Definição V.1.2 (Extensão) *Considere as teorias* default Σ *e* Σ'. *A teoria* Σ' *estende* Σ, *denotando-se por* $\Sigma \subseteq \Sigma'$, *se e somente se existe uma relação total,* r , *entre os respectivos conjuntos de extensões* default, [4] *(i.e.* $r \subseteq ExtDef(\Sigma') \times ExtDef(\Sigma)$*), de tal forma que:*

$$(\forall\, \mathcal{C} \in ExtDef(\Sigma')\ )(\exists\, \mathcal{B} \in ExtDef(\Sigma)\ )\ \ \mathcal{C}\ r\ \mathcal{B}\ e\ \mathcal{B} \subseteq \mathcal{C}$$

Assumindo a definição acima, naturalmente encontra-se a definição para extensão conservativa entre teorias default:

Definição V.1.3 (Extensão Conservativa) *Considere teorias* default Σ *e* Σ'. *Diz-se que* Σ' *estende* Σ *conservativamente, denotada por* $\Sigma \leq \Sigma'$, *se e somente se existe uma função* f *de* $ExtDef(\Sigma')$ *em* $ExtDef(\Sigma)$ $(f : ExtDef(\Sigma') \longmapsto ExtDef(\Sigma))$, *de tal forma que:*

$$(\forall\, \mathcal{C} \in ExtDef(\Sigma')\ )(\exists\, \mathcal{B} \in ExtDef(\Sigma)\ )\ \ f(\mathcal{C}) = \mathcal{B}\ e\ \mathcal{B} \leq \mathcal{C}$$

Nesta definição aparece uma função f, para associar extensões *default*, em substituição a relação total anteriormente empregada para estabelecer uma simples extensão, por causa de uma propriedade particular de extensões *default*: a minimalidade (B.3.3). Explicando melhor, a minimalidade condiciona que extensões *default* distintas em Σ não poderiam ser extendidas conservativamente para uma mesma extensão *default* de Σ', como está demonstrado a seguir:

Proposição V.1.2 *Sejam* $\mathcal{E}$ *e* $\mathcal{F}$ *duas extensões* default *distintas de* Σ, *e seja* $\mathcal{C}$ *uma extensão* default *para* Σ', *tais que:*

$$\{\mathcal{E} \leq \mathcal{C} \Rightarrow \mathcal{F} \not\leq \mathcal{C}\} \quad \vee \quad \{\mathcal{F} \leq \mathcal{C} \Rightarrow \mathcal{E} \not\leq \mathcal{C}\}$$

Prova : Pela propriedade da minimalidade B.3.3, afirma-se que:

i) $\mathcal{E} \subseteq \mathcal{F} \Rightarrow \mathcal{E} = \mathcal{F}$

ii) $\mathcal{F} \subseteq \mathcal{E} \Rightarrow \mathcal{E} = \mathcal{F}$

[4] Para diferenciar as duas conotações para o termo **extensão**, o adjetivo *default* segue sempre após, quando a referência for ao conjunto de sentenças gerado por uma teoria *default*.

Como por hipótese $\mathcal{E} \neq \mathcal{F}$, então tem-se que nem $\mathcal{F} \not\subseteq \mathcal{E}$ e nem $\mathcal{E} \not\subseteq \mathcal{F}$. Portanto, para sentenças α e β, ambas pertencentes a linguagem $\mathcal{L}(\Sigma)$, tal que $\alpha \in \mathcal{E} - \mathcal{F}$ e $\beta \in \mathcal{F} - \mathcal{E}$

i) Se $\mathcal{E} \models \alpha$ então $\mathcal{F} \not\models \alpha$; e

ii) Se $\mathcal{F} \models \beta$ então $\mathcal{F} \not\models \beta$

Iniciamenlte suponha que $\mathcal{E} \leq \mathcal{C}$, e logo por definição,

$$\mathcal{C} \models \alpha \quad \text{porém} \quad \mathcal{F} \not\models \alpha$$

Consequentemente, $\mathcal{F} \not\leq \mathcal{C}$. Por outro lado, suponha que $\mathcal{F} \leq \mathcal{C}$, e igualmente por definição,

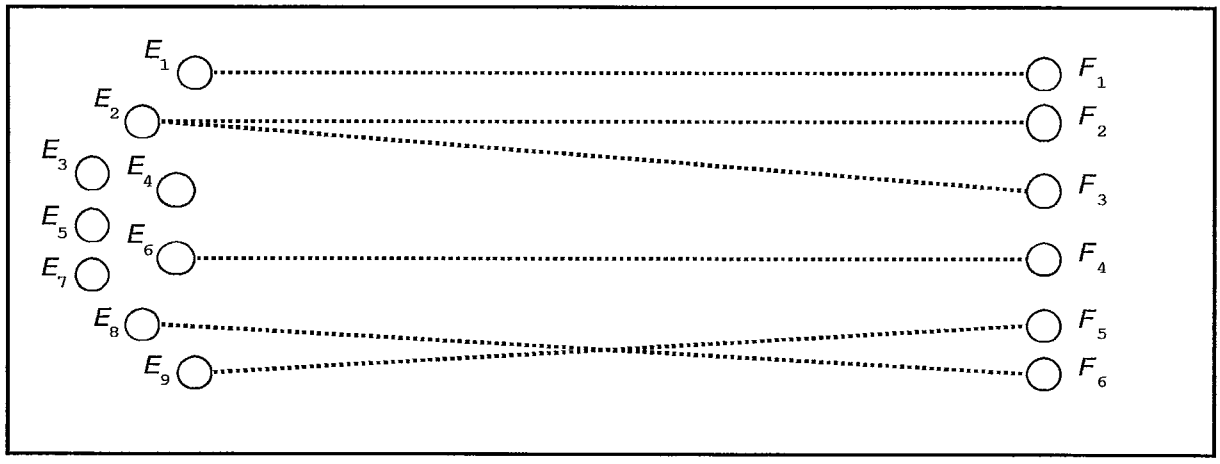
$$\mathcal{C} \models \beta \quad \text{porém} \quad \mathcal{E} \not\models \beta$$

Consequentemente, $\mathcal{E} \not\leq \mathcal{C}$. E portanto, a proposição enunciada é sempre verdade. □

O argumento na proposição acima pode ser aplicado repetidamente, e no final constatado que no máximo uma única extensão *default* de Σ, poderia extender de forma conservativa a uma outra extensão *default* de Σ'. Logo, sendo conservativa a extensão $\mathcal{B} \subseteq \mathcal{C}$, sobre a relação total f que a estabelece, é correto afirmar que vale a "unicidade",

$$(\forall\, \mathcal{C} \in \mathit{ExtDef}(\Sigma')\,)(\forall\, \mathcal{E}, \mathcal{F} \in \mathit{ExtDef}(\Sigma)\,)\;\; \mathcal{C}\,\mathbf{f}\,\mathcal{E} \;\wedge\; \mathcal{C}\,\mathbf{f}\,\mathcal{F} \;\Leftrightarrow\; \mathcal{E} = \mathcal{F}$$

e assim f é necessáriamente uma função.



Pela minimalidade fica impossibilitada a extensão conservativa de mais de um conjuntos de sentenças representando hipóteses distintas (mesmo que não inconsistentes

entre si) para uma única extensão *default*. Portanto, a relação total suficiente para definir uma extensão entre teorias *default* será agora uma função quando a extensão seja conservativa. Isto significa que um conjunto de extensões *default* não é diminuido através de extensões conservativas.

Levando em conta a definição anterior, caso seja possível adicionalmente, expandir todos os modelos para as extensões *default* de Σ para modelos das extensões *default* de Σ', adotando este critério mais liberal, uma extensão conservativa seria também expansiva, ou seja,

Definição V.1.4 (Extensão Expansiva) *Considere teorias default* Σ *e* Σ'. *Diz-se que* Σ' *estende* Σ *expansivamente, simbolizado por* $\Sigma \trianglelefteq \Sigma'$, *se e somente se existe uma função* f *sobrejetiva,* $f : ExtDef(\Sigma') \longmapsto ExtDef(\Sigma)$, *de tal forma que:*

$$(\forall\, \mathcal{C} \in ExtDef(\Sigma')\,)(\exists\, \mathcal{B} \in ExtDef(\Sigma)\,)\;\; f(\mathcal{C}) = \mathcal{B} \;\;\Leftrightarrow\;\; \mathcal{B} \trianglelefteq \mathcal{C}$$

Tomando uma função sobrejetiva, todas as classes de modelos determinadas por extensões *default* de Σ estão associadas à realizações possíveis para as extensões *default* de Σ'. Já que esta asssociação basea-se em uma extensão expansiva, então qualquer que seja a realização possível para uma extensão *default* de Σ, será sempre expandida a uma modelo para W_2.

Teorema V.1.2 *Sejam duas teorias* default Σ *e* Σ'.

$$\Sigma \trianglelefteq \Sigma' \;\Rightarrow\; \Sigma \leq \Sigma'$$

Prova : Assumindo por hipótese $\Sigma \trianglelefteq \Sigma'$, e sendo f a função que estabelece a extensão expansiva. Por definição, para toda extensão *default* $\mathcal{C} \in ExtDef(\Sigma')$, existe uma extensão *default* $f(\mathcal{C}) \in ExtDef(\Sigma)$ tal que:

$$f(\mathcal{C}) \trianglelefteq \mathcal{C}$$

Logo, pelo teorema A.3.2,

$$f(\mathcal{C}) \leq \mathcal{C}$$

Consequentemente, por definição $\Sigma \leq \Sigma'$. □

Todas as definições propostas aqui obedecem as propriedades esperadas, por exemplo, todas comportam-se transitivamente, pois a composição de relações totais, ou mesmo funções, é igualmente transitiva.

Outro fator importante refere-se ao comportamente destes mecanismos lógicos em relação a extensões *default* inconsistentes. Segundo B.3.2 e B.3.1, uma teoria *default* só apresentaria uma extensão default inconsistente quando o seu conjunto de axiomas também é trivializado, permitindo inferir qualquer sentença na linguagem, e esta seria a única extensão *default* gerada. Por conseqüência, qualquer teoria default com extensões *default* poderia ser interpretada em termos de uma teoria *default* inconsistente, mas apenas uma outra teoria *default* também inconsistente poderia estender-se a esta conservativamente. Um desempenho analógo às versões clássicas para as mesmas noções de interpretação e extensão conservativa.

Todas estas definições são completamente compatíveis com as definições correspondentes para a Lógica de Primeira Ordem. Ou seja, sempre que as teorias *default* envolvidas não possuem nenhum *default* aplicável, mas são coerentes (talvez sejam vazios os seus conjuntos de defaults), todos estes mecav nismos reduzem-se espontaneamente ao caso clássico, porque somente uma extensão *default* seria gerada, justamente o conjunto de conseqüências lógicas derivadas da parte monotônica, e então tudo funcionaria como se a interpretação ou a extensão conservativa trata-se com simples teorias de primeira ordem.

Estes fatos coadunam com as expectativas em relação a coerências destas definições e asseguram a confiança nos resultados produzidos.

V.1.2.1 Extensões por Definição

Estabelecer uma extensão e certificar *a posteriori* sua conservatividade não chega, nem de longe, a ser uma tarefa trivial, notadamente entre teorias *defaults*. Portanto, os procedimentos de implementação e composição tornam-se realistas no momento que se mostra possível definir símbolos adicionais, sem interferir com as propriedades já representadas na teoria, a exemplo de [VV90]. Na realidade, são as extensões conservativas por definição (ainda que liberais) quem fazem a construção de extensões conservativas pragmaticamente factível, na grande maioria das situações de interesse.

O que é possível ao contexto clássico sem dúvida poderia ser imitado através do conjunto de axiomas da teoria *default*. Mas, o atrativo é mesmo investigar a uti-

lização de defaults para construir extensões por definição, aproveitando inteiramente as características não monotônicas.

Primeiramente, considere a definição liberal de um símbolo de função f, proposta abaixo, como um correspondente *default* para a definição liberal por comportamento de E/S (A.3.8).

Definição V.1.5 (Definição Liberal de Função por *Default*) *Considere a linguagem $\mathcal{L}$ incluindo os símbolos de sortes $S_1,S_2,\ldots,S_n,S$, e seja f um símbolo não pertencente a esta linguagem. Considere ainda, ϕ uma fórmula em $\mathcal{L}$, sem variáveis livres. Considere ψ uma fórmula em $\mathcal{L}$ sem variáveis livres, além de v de sorte S. Seja d_f o seguinte* default,

$$\frac{\phi : (\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\ \psi[v/f(v_1, v_2, \ldots, v_n)]}{(\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\ \psi[v/f(v_1, v_2, \ldots, v_n)]}$$

Este default d_f *é chamado* Definição Liberal do Símbolo de Função f por Default.

Definição V.1.6 (*Extensão por Definição Liberal de Função por* Default) *Seja $\Sigma = \langle W, D\rangle$ uma teoria* default *normal fechada e $\mathcal{L}(\Sigma)$ a sua linguagem. Considere a nova teoria* default *normal fechada $\Sigma' = \langle W, D \cup d_f\rangle$, obtida a partir de Σ, introduzindo o símbolo f em sua linguagem $\mathcal{L}(\Sigma)$, como um símbolo de função sobre os sortes $S_1,S_2,\ldots,S_n$, em $\mathcal{L}(\Sigma)$, ou seja, $f(S_1,S_2,\ldots,S_n) \longmapsto S$. E também acrescentando ao conjunto de defaults D, d_f (conforme o* default *V.1.5) uma definição liberal da função f por* default. *Então, diz-se que Σ' é uma* Extensão de Σ por Definição Liberal do Símbolo de função f por Default.

A vantagem principal nesta definição é aproveitar ao máximo, a flexibilidade inerente ao constexto não monotônico. Um função f poderia, por exemplo, estar definida em um subconjunto das extensões *default* geradas por Σ', e nas restantes f permaneceria indefinido. Isto é, condicionar a definição de um novo símbolo a satisfação de uma certa propriedade.

Generalizar estas idéias permite que uma função apresente comportamentos diferentes em extensões *default* distintas, de acordo com pre-condições especificadas.

Mesmo perdendo em generalidade, ao limitar considerações apenas às teorias *default* normais, ganha-se muito em simplicidade. A condição necessária à conservatividade será relativamente mais fácil de certificar, visto que a semi-monotonicidade (B.3.7) encarrega-se de parte do trabalho. Logo, o esforço envolvido seria proporcionalmente

menor que verificar todas as conseqüências de ambos os conjuntos de extensões *default* em Σ e Σ'. A condição exigida aqui pelo enunciado, para validar a conservatividade, relembra inclusive a utilizada na definição clássica correspondente A.3.8. Seja a proposição

Proposição V.1.3 *Seja a teoria* default *normal fechada* $\Sigma' = \langle W, D \cup d_f \rangle$, *uma extensão de uma teoria* $\Sigma = \langle W, D \rangle$ *por definição liberal de símbolo de um função* f, *nos moldes da definição V.1.6, onde o* default d_f *que define liberalmente a função, é*

$$\frac{\phi : (\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\ \psi[v/f(v_1, v_2, \ldots, v_n)]}{(\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\ \psi[v/f(v_1, v_2, \ldots, v_n)]}$$

Considere a variável $u : S$ *não pertencente a* $\mathcal{L}(\Sigma)$, *se,*

$$W \models \phi \ \Rightarrow\ (\exists u : S)\psi[v/u]$$

então Σ' *é uma extensão conservativa de* Σ, *denotado por* $\Sigma \leq \Sigma'$, *conforme a definição V.1.3.*

Prova : A fim de demonstrar que $\Sigma \leq \Sigma'$ é necessário apresentar uma função h de $ExtDef(\Sigma)$ em $ExtDef(\Sigma')$, de tal modo que $h(\mathcal{E}') \leq \mathcal{E}'$, para qualquer extensão *default* $\mathcal{E}'$ de Σ'.

Lema V.1.3.1 *Toda extensão* default $\mathcal{E}'$ *de* Σ', *é tal que existe uma extensão* default $\mathcal{E}$ *de* Σ, *contida em* $\mathcal{E}'$, *ou seja,* $\mathcal{E} \subseteq \mathcal{E}'$.

Prova : Suponha $\mathcal{E}'$ uma extensão *default* arbitrária de Σ', por definição segundo B.3.2, em [Rei80], $\mathcal{E}'$ é sempre da seguinte forma:

$$\mathcal{E}' \equiv W \cup \textbf{Consequentes}\,(\textbf{GD}\,(\mathcal{E}', \Sigma'))$$

Considere um conjunto de sentenças $\mathcal{E}$ (fechado sob conseqüência lógica), obtido quando da retirada de d_f de $\textbf{GD}\,(\mathcal{E}', \Sigma')$, ou seja :

$$\mathcal{E} \equiv W \cup \textbf{Consequentes}\,(\textbf{GD}\,(\mathcal{E}', \Sigma') - \{d_f\})$$

Obviamente, $\mathcal{E}$ assim definido está sempre contido em $\mathcal{E}'$ ($\mathcal{E} \subseteq \mathcal{E}'$), por construção, assim seria suficiente mostrar que $\mathcal{E}$ constitue sempre uma extensão *default* para Σ.

Para tanto, o teorema B.3.4 afirma que $\mathcal{E}$ é uma extensão *default* para Σ se e somente se :

i) $W \subseteq \mathcal{E}$

ii) $\mathcal{E}$ é estável com relação a D

iii) $\mathcal{E}$ é acessível de W com relação a D

De início, a primeira condição é trivialmente satisfeita.

A fim de verificar *ii*), seja o mapeamento d (B.3.6) abaixo :

$$d(\mathcal{E}) = \begin{cases} \mathbf{Cn}(\mathcal{E} \cup \{\omega\}) & \text{se } \frac{\alpha:\omega}{\omega} \text{ é aplicável com relação a } \mathcal{E}. \\ \mathcal{E} & \text{caso contrário} \end{cases}$$

para satifazer a segunda condição, é preciso mostrar que $d(\mathcal{E}) = \mathcal{E}$ para qualquer *default* em D. Isto implica que $\mathcal{E}$ é maximal, i.e. não há em D nenhum *default* aplicável, cujo o conseqüente não seja derivável em $\mathcal{E}$.

Tomando $\frac{\alpha:\omega}{\omega}$ um default qualquer em D : Ou $\frac{\alpha:\omega}{\omega}$ não é aplicável com relação a $\mathcal{E}$, e neste caso vale diretamente :

$$d(\mathcal{E}) = \mathcal{E}$$

Ou $\frac{\alpha:\omega}{\omega}$ é aplicável, isto é :

$$\mathcal{E} \models \alpha \text{ e } \mathcal{E} \not\models \neg\omega$$

Mas se $\mathcal{E} \models \alpha$ então $\mathcal{E}' \models \alpha$,(I), pois $\mathcal{E} \subseteq \mathcal{E}'$. Por outro lado, suponha que $\mathcal{E}' \models \neg\omega$, ou seja,

$$W \cup \mathbf{Consequentes}\,(\mathbf{GD}\,(\mathcal{E}', \Sigma')) \models \neg\omega \tag{1}$$

Há dois casos a serem considerados :

Primeiro caso : $\mathcal{E}' \models \phi$, logo d_f é aplicável com relação a $\mathcal{E}'$ e portanto $d_f \in \mathbf{GD}\,(\mathcal{E}', \Sigma')$. Por isso, (1) pode ser reescrito como,

$$W \cup \mathbf{Consequentes}\,(\mathbf{GD}\,(\mathcal{E}', \Sigma') - \{d_f\}), \mathbf{Consequentes}\,(\{d_f\}) \models \neg\omega$$

e então,

$$\mathcal{E}, \mathbf{Consequentes}\,(\{d_f\}) \models \neg\omega \tag{2}$$

$$\mathcal{E}, (\exists u : S)\,\psi[v/u] \models \neg\omega \tag{3}$$

Todavia, (3) segue diretamente de (2) porque **Consequentes** $(\{d_f\})$ apresenta um termo, $f(v_1, v_2, \ldots, v_n)$, que não pertence a linguagem de Σ. Portanto, qualquer realização admissível para $(\exists u : S)\,\psi[v/u]$ expande-se a um modelo para **Consequentes** $(\{d_f\})$, e já que ω está mesmo na linguagem de Σ, são irrelevantes quaisquer possíveis realizações para $f(v_1, v_2, \ldots, v_n)$.

Entretanto, foi assumido que $\mathcal{E}' \models \phi$, onde $\phi \in \mathcal{L}(\Sigma)$. Assim, ϕ não pode ser conseqüência da aplicação de d_f, porque ϕ deve ser válido antes que d_f seja disparado na geração de $\mathcal{E}'$. Por conseguinte, ϕ deve ser conseqüência lógica de $W \cup$ **Consequentes** $(\mathbf{GD}\,(\mathcal{E}', \Sigma'))$ sem contudo envolver d_f, ou seja,

$$W \cup \textbf{Consequentes}\,(\mathbf{GD}\,(\mathcal{E}', \Sigma') - \{d_f\}) \models \phi$$

ou, $\mathcal{E} \models \phi$. Mas por hipótese, $W \models \phi \Rightarrow (\exists u : S)\psi[v/u]$ logo, $\mathcal{E} \models (\exists u : S)\psi[v/u]$. Finalmente,

$$\mathcal{E} \models \neg\omega$$

Concluindo,

$$\text{Se } \mathcal{E}' \models \neg\omega \text{ então } \mathcal{E} \models \neg\omega$$

ou por contra-posição

$$\text{Se } \mathcal{E} \not\models \neg\omega \text{ então } \mathcal{E}' \not\models \neg\omega$$

Juntando isto com (I), assegura-se que um *default* aplicável com relação a $\mathcal{E}$ igualmente será aplicável com relação a $\mathcal{E}'$. Intuitivamente, isso significa que d_f não prejudica a aplicabilidade dos *defaults* antes $\mathcal{E}$.

Mais formalmente, $\mathcal{E}'$ é uma extensão *default* de Σ', e por definição estável, assim qualquer default $\frac{\alpha:\omega}{\omega}$ aplicável com relação a $\mathcal{E}'$ estará necessariamente em $\mathbf{GD}\,(\mathcal{E}', \Sigma')$, isto é,

$$\left\{\frac{\alpha : \omega}{\omega}\right\} \subseteq \mathbf{GD}\,(\mathcal{E}', \Sigma')$$

porém este *default* é diferente de d_f, logo,

$$\left\{\frac{\alpha : \omega}{\omega}\right\} \subseteq \mathbf{GD}\,(\mathcal{E}', \Sigma') - \{d_f\}$$

então $\mathcal{E} \models \omega$ e consequentemente,

$$d(\mathcal{E}) = \mathcal{E} \cup \{\omega\} = \mathcal{E}$$

Segundo Caso : $\mathcal{E}' \not\models \phi$, logo d_f não é aplicável em $\mathcal{E}'$ e portanto $d_f \notin \mathbf{GD}(\mathcal{E}', \Sigma')$. Neste caso, (2) pode ser reformulado como :

$$W \cup \mathbf{Consequentes}(\mathbf{GD}(\mathcal{E}', \Sigma') - \{d_f\}) \models \neg\omega$$

e qualquer *default* aplicável com relação a $\mathcal{E}$ igualmente será aplicável com relação a $\mathcal{E}'$. Também, por causa da estabilidade de $\mathcal{E}'$, $\mathcal{E} \models \omega$ e novamente,

$$d(\mathcal{E}) = \mathcal{E} \cup \{\omega\} = \mathcal{E}$$

Conluindo o conjunto de crenças $\mathcal{E}$ é sempre estável com relação a D. Além do que, $\mathcal{E} = \mathcal{E}'$. Resta ainda demonstrar que $\mathcal{E}$ é acessível de W com relação a D. Analogamente existem dois casos a considerar :

Primeiro Caso : $d_f \in (\mathbf{GD}(\mathcal{E}', \Sigma')$
Por definição, $\mathcal{E}'$ é acessível de W com relação a $D \cup \{d_f\}$, ou seja para qualquer conjunto de sentenças $\mathcal{S}'$, tal que $W \subseteq \mathcal{S}' \subset \mathcal{E}'$ existirá em $D \cup \{d_f\}$ sempre um *default* capaz de satisfazer a :

$$\mathcal{S}' \subset d(\mathcal{S}') \subseteq \mathcal{E}'$$

em particular para qualquer subconjunto próprio de $\mathcal{E}$ (i.e. $\mathcal{S} \subset \mathcal{E}$). Deste modo, se $W \subseteq \mathcal{S} \subset \mathcal{E}$, então $W \subseteq \mathcal{S} \subset \mathcal{E} \subset \mathcal{E}'$, porque $\mathcal{E} \subset \mathcal{E}'$. Logo,

$$\mathcal{S} \subset d(\mathcal{S}) \subset \mathcal{E}'$$

Se o *default* disparado no mapeamento pertence a D, pela comprovação da estabilidade de $\mathcal{E}$ acima, o seu conseqüente está também em $\mathcal{E}$, e portanto $d(\mathcal{S}) \subseteq \mathcal{E}$, de onde segue que :

$$\mathcal{S} \subset d(\mathcal{S}) \subseteq \mathcal{E}$$

Mesmo que d_f seja aplicável em $\mathcal{S}$, existe outro default $\mathbf{GD}(\mathcal{E}', \Sigma') - \{d_f\}$ não aplicado ainda em $\mathcal{S}$, e que poderia ser alternativamente utilizado pois a sua aplicação nunca

depende de **Consequentes** ($\{d_f\}$), apenas de *defaults* em D. Finalmente, $\mathcal{E}$ será acessível de W com relação aos *defaults* apenas em D.

Segundo Caso : $d_f \notin$ (**GD** $(\mathcal{E}', \Sigma')$
Trivialmente $\mathcal{E} = \mathcal{E}'$. Visto que $\mathcal{E}'$ é acessível de W com relação a $D \cup \{d_f\}$, também $\mathcal{E}$ será acessível de W com relação a $D \cup \{d_f\}$, contudo nem $\mathcal{E}' \models$ **Consequentes** $(\{d_f\})$ e nem $\mathcal{E} \models$ **Consequentes** $(\{d_f\})$, portanto, o *default* aplicado no mapeamento $d(\mathcal{E})$ não pode ter sido d_f, necessariamente pertencia a D, por conseqüência, $\mathcal{E}$ será acessível de W com relação a D.

Em qualquer situação, $\mathcal{E}$ será acessível de W com relação a D o que finalmente satifaz a *iii*). Assim, $\mathcal{E}$ é uma extensão *default* de $\Sigma = \langle W, D \rangle$. □

Lema V.1.3.2 *Para uma extensão* default $\mathcal{E}$ *de* Σ *e* $\mathcal{E}'$ *de* Σ', *se* $\mathcal{E} \subseteq \mathcal{E}'$ *então* $\mathcal{E} \leq \mathcal{E}'$.

Prova : Novamente, há dois casos a considerar :
Primeiro : quando $\mathcal{E} \not\models \phi$, o *default* d_f nem chega a ser aplicável, e portanto $\mathcal{E}'$ continua equivalente a $\mathcal{E}$, ou seja, $\mathcal{E} = \mathcal{E}'$, e portanto, trivialmente, $\mathcal{E} \leq \mathcal{E}'$.

Segundo : se $\mathcal{E} \models \phi$, o *default* d_f pode ser disparado, logo, a extensão *default* $\mathcal{E}'$ que a contém é capaz derivar o consequente de d_f, ou seja,

$$\mathcal{E}' \models (\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\psi(f(v_1, v_2, \ldots, v_n))$$

mas por hipótese, $W \models \phi \Rightarrow (\exists u : S)\psi[v/u]$, consequentemente,

$$\mathcal{E} \models \phi \Rightarrow (\exists u : S)\psi[v/u]$$

então $\mathcal{E}'$ é uma extensão por definição liberal de f, por comportamento de E/S, e conforme o teorema A.3.5, afirma-se : $\mathcal{E} \leq \mathcal{E}'$

Portanto, conclue-se que o *default* d_f não acrescenta nenhuma capacidade dedutiva a teoria Σ, em relação aos elementos de sua linguagem. De qualquer forma fica mantida a conservatividade na extensão para Σ'. □

A partir do primeiro lema, afirma-se que para todas as extensões default de Σ', por instância $\mathcal{E}'$, existe alguma extensão default $\mathcal{E}$ em Σ, de modo que, $\mathcal{E} \subseteq \mathcal{E}'$.

Contudo, pelo segundo lema, é possível concluir que esta extensão é conservativa, portanto $\mathcal{E} \leq \mathcal{E}'$. Então facilmente é estabelecida a função h de $ExtDef(\Sigma')$ em $ExtDef(\Sigma)$ satisfazendo a $h(\mathcal{E}') = \mathcal{E}$, pois $\mathcal{E} \leq \mathcal{E}'$. Finalizando assim a demonstração.

□

Raciocínio similar ao empregado na demonstração acima também aplica-se para confirmar definições por *default* para predicados e funções, em correspondência com A.3.4 e A.3.6.

Definição V.1.7 (Definição de Predicados por *Default*) *Considere a linguagem $\mathcal{L}$ incluindo os símbolos de sortes $S_1,S_2,\ldots,S_n$, e seja p um símbolo não pertencente a esta linguagem. Considere ainda, π uma fórmula em $\mathcal{L}$ sem variáveis livres, com execeção de $v_1,v_2,\ldots,v_n$. Seja d_p o seguinte* default,

$$\frac{: (\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)[\pi \Leftrightarrow p(v_1, v_2, \ldots, v_n)]}{(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)[\pi \Leftrightarrow p(v_1, v_2, \ldots, v_n)]}$$

Este default *é chamado* Definição do Símbolo de Predicado p por Default.

Definição V.1.8 (Extensão por Definição de Predicados por *Default*) *Considere Σ uma teoria* default *normal fechada e $\mathcal{L}$ a sua linguagem. Considere a nova teoria* default *normal Σ', obtida a partir de Σ, introduzindo o símbolo p na linguagem $\mathcal{L}$, como um símbolo de predicado sobre os sortes $S_1,S_2,\ldots,S_n$, em $\mathcal{L}$, ou seja, $p(S_1,S_2,\ldots,S_n)$. E também acrescentando aos* defaults *pertencentes à especificação de Σ, uma definição do símbolo de predicado p (conforme o* default *V.1.7). Então, diz-se que Σ' é uma* Extensão de Σ por Definição de Predicado p por Default.

Proposição V.1.4 *Seja uma teoria Σ' uma extensão de uma teoria Σ por definição de um predicado p por* default. *Então $\Sigma \leq \Sigma'$*

E para símbolos de função,

Definição V.1.9 (Definição de Função por *Default*) *Considere a linguagem $\mathcal{L}$ incluindo os símbolos de sortes $S_1,S_2,\ldots,S_n,S$, e seja f um símbolo não pertencente a esta linguagem. Considere ainda, ϕ uma fórmula em $\mathcal{L}$ sem variáveis livres, além de $v_1,v_2,\ldots,v_n$ e v. Seja d_f o seguinte default:*

$$\frac{: (\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)(\forall v : S)\,[\phi \Leftrightarrow f(v_1, v_2, \ldots, v_n) = v]}{(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)(\forall v : S)\,[\phi \Leftrightarrow f(v_1, v_2, \ldots, v_n) = v]}$$

Este default d_f *é chamado* Definição do Símbolo de Função f por Default.

Definição V.1.10 (Extensão por Definição de Funções por *Default*) *Seja uma teoria* default *normal fechada* Σ *e* $\mathcal{L}$ *a sua linguagem. Seja a nova teoria* default *normal* Σ', *obtida a partir de* Σ, *introduzindo o símbolo* f *na linguagem* $\mathcal{L}$, *como um símbolo de função sobre os sortes* $S_1,S_2,\ldots,S_n,S$, *em* $\mathcal{L}$, *ou seja,* $f(S_1,S_2,\ldots,S_n) : S$. *E também acrescentando uma definição da função* f *por Default, (conforme o* default *V.1.9). Então, diz-se que* Σ' *é uma* Extensão de Σ por Definição do de Função f por Default.

Proposição V.1.5 *Seja uma teoria* Σ' *uma extensão de uma teoria* Σ *por definição de símbolo de um função* f, *e a fórmula* β_f *define a função* f, *de acordo com o* default *em V.1.9. Considere as fórmulas* ϵ_f *e* υ_f *abaixo, e as variáveis* u *e* u' *não em* $\mathcal{L}$,

$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)(\exists u : S)\, \phi[v/u]$$
$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)(\forall u, u' : S)\{\phi[v/u] \wedge \phi[v/u']\} \;\Rightarrow\; u = u'$$

As fórmulas ϵ_f *e* υ_f *são chamadas de condição de existência e de unicidade respectivamente. Se* ϵ_f *e* υ_f *pertencem as conseqüências de qualquer extensão* default *de* Σ, *então* Σ' *é uma extensão conservativa de* Σ.

$$(\forall \mathcal{E} \in ExtDef(\Sigma))[(\epsilon_f \wedge \upsilon_f) \in Cn(\mathcal{E})] \;\Rightarrow\; \Sigma \leq \Sigma'$$

Porém, evitando ser repetitivo este trabalho não pretende detalhar as demonstrações nestes casos. Tomando como hipótese as devidas condições necessárias em cada caso particular a conservatividade será equivalentemente determinada.

V.2 Implementação no Contexto não Monotônico

Concluidas as definições das noções lógicas necessárias para fundamentar os procedimentos de implementação, o passo seguinte é adaptar ao contexto não monotônico os princípios que regulam estes procedimentos, de modo a garantir sua corretude assim como, explorar a flexibilidade conseguida com o relaxamento da monotonicidade.

Isto significa redefinir o passo canônico, formalizando a implementação entre Tipos Abstratos de Dados especificados em *Lógica Default* e restabelecer os teoremas da Modularização e da Construtibilidade (III.1.1,III.1.2).

Indiscutivelmente, a capacidade de tirar conclusões que poderiam ser revistas e invalidadas futuramente, a luz de melhores informações, empresta aos formalismos não

monotônicos, em particular a *Lógica Default,* uma flexibilidade sem comparação. Esta flexibilidade pode ser extremamente vantajosa, caso seja adequadamente utilizada, expandindo os mecanismos já diponíveis para o desenvolvimento de programas fundamentado apenas na lógica clássica.

Todavia o ganho exige um preço a ser pago. Há uma considerável aumento em termos de dificuldade, uma maior atenção é requerida para lidar propriamente com os mecanismos não monotônicos, é sempre mais fácil perder a exata noção intuitiva dos resultados sendo produzidos, faz-se necessário um maior conhecimento lógico-teórico, para não mencionar problemas referentes a computabilidade. De qualquer forma, a consciência sobre todos estes fatores é imprescindível antes de decidir abandonar a monotonicidade. Portanto, é prudente refletir sobre o real aproveitamento deste potencial de expressibilidade, e quando seus recursos são efetivamente necessários.

Mas este é positivamente o caminho para contornar muitas entre as limitações do formalismo clássico, que afetam o desenvolvimento de programas. Principalmente, fornecer os meios necessários para tratar com situações parcialmente compreendidas, ou obter conclusões mesmo na ausência de informações suficientemente detalhadas, ou ainda, ter a habilidade de reconsiderar *a posteriori* algumas decisões. Nunca é preciso lembrar o quanto estas circunstâncias são costumeiras na prática de desenvolvimento de programas.

Esta maior liberdade só tende a aproximar o formalismo das situações reais. Embora seja recomendável não recorrer a lógicas não monotônicas sem absoluta necessidade, sua aplicação pode ser extremamente apropriada, e as vezes o único recurso.

V.2.1 Especificação de TAD em *Lógica Default*

Formalismos não monotônicos podem cumprir o mesmo papel, em substituição à lógica tradicional, na especificação de Tipos Abstratos de Dados, representando o significado de conceitos abstratos através de suas propriedades.

Contudo, um ponto fundamental precisa estar bem entendido, a diferença entre o caráter das conclusões geradas. Enquanto uma axiomatização em Lógica de Primeira Ordem permitia deduzir apenas os fatos válidos (i.e. teoremas), o que existe agora são conjuntos formados por crenças consistentes, ou seja, hipóteses supostamente coerentes com as informações que podem ser assumidas na realidade modelada. A noção absoluta de verdade cede lugar a uma noção mais liberal, talvez subjetiva, de racionalidade.

Dado um conjunto de axiomas de primeira ordem descrevendo parcialmente um determinado conceito, um conjunto de *defaults* pode ser acrescentado, visando uma caracterização plausível para algumas classes de modelos do conjunto de axiomas, isto é, os *defaults* apresentam um mecanismo adicional para complementar o conhecimento a respeito do conceito sendo especificado. Por meio de apresentações para teorias *default*, vários conjuntos de crenças consistentes sobre um mesmo conceito podem ser sucintamente especificadas.

As várias teorias de primeira ordem (i.e. extensões *default*) que são geradas por uma única teoria *default*, constituem as hipóteses que se pretende levantar, de forma coerente, sobre os modelos para o conjunto de axiomas.

Porém, nisto está o ponto forte e o fraco da especificação utilizando teorias *default*. As vantagens parecem óbvias, simplicidade na notação, clareza no entendimento, facilidade para manipular simultaneamente mais de um conjunto de informações distintas. Entretanto, é sempre aconselhável estar atento especificando apenas o que é relevante, porque em geral, a representação deve ser seletiva em relação as "possíveis crenças" enumeradas. A escolha de hipóteses também implica em decisões de refinamento ou projeto, e logo, deveriam fazer algum sentido para a continuidade da implementação.

A especificação para um TAD, de acordo com a seção II.2, fica constituida em duas partes. Uma reponsabiliza-se por definir regras de formação para os termos e fórmulas da linguagem, enquanto a outra, semântica, confere significado ao TAD.

No contexto da *Lógica Default*, nada há a acrescentar a parte sintática, porque ainda trabalha-se com linguagens de primeira ordem poli-sortidas, e a notação empregada antes continua satisfatória. A especificação semântica tem sua apresentação modificada pela introdução de um conjunto de *defaults*, contemplando a axiomatização existente com mecanismos de inferência não monotônicos. Deste modo, várias teorias de primeira ordem compartilhando um mesmo conjunto de axiomas seriam resumidas na apresentação para uma única teoria *default*.

ADT *Process*

Syntax

Import *Event*

Predicate $active(Process)$;
$running(Process)$;
$holding(Process)$;
$cpufree(Event)$;
$interupted(Event)$;

Axiomatization

$$(\forall p : Process)\{active(p) \Leftrightarrow (running(p) \ \vee \ holding(p))\} \quad (1)$$

$$(\forall p : Process)\{running(p) \Rightarrow \neg holding(p)\} \quad (2)$$

$$(\forall p : Process)$$
$$\{running(p) \Rightarrow (\forall p' : Process)(\neg p \stackrel{P}{=} p' \Rightarrow \neg running(p'))\} \quad (3)$$

$$cpufree(1) \quad (4)$$

$$(\forall p : Process)(\forall e : event)\{active(p) \ \wedge \ cpufree(e) \Rightarrow interruped(e+1)\} \quad (5)$$

$$(\forall e : event)\{interruped(e) \Rightarrow cpufree(e+1)\} \quad (6)$$

Hypothesis

$$\frac{(\exists e : event)cpufree(e) : (\exists p : Process)holding(p)}{(\exists p : Process)running(p)} \quad (7)$$

$$\frac{(\exists e : event)interrupted(e) : (\exists p : Process)running(p)}{(\exists p : Process)holding(p)} \quad (8)$$

End

A especificação acima difere da notação utilizada nos capítulos precedentes, pelo aparecimento da nova declaração **Hypothesis** que representa como um conjunto de defaults, todas as conjecturas importantes, levantadas sobre os axiomas.

Correlacionado esta notação com a notação usual para apresentação de teorias *default*, onde $\langle W, D\rangle$ aparece entenda-se que W representa a declaração **Axiomatization,** enquanto os *defaults* em D são enumerados em **Hypothesis.**

V.2.2 Passo Canônico

Em princípio, a intuição por trás da formulação do passo canônico deve ser respeitada, mesmo que seja diferente a linguagem e as noções lógicas empregadas, portanto, a definição para o passo canônico, no contexto não monotônico, não será modificada em essência.

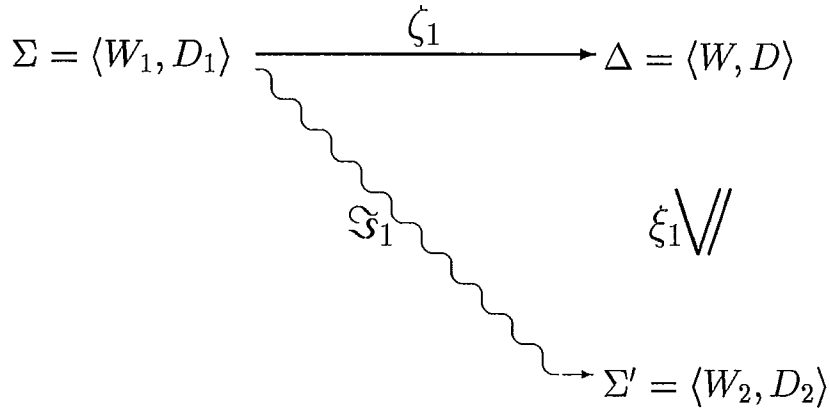
Definição V.2.1 (Implementação) *Seja* $\Sigma = \langle W_1, D_1 \rangle$ *e* $\Sigma' = \langle W_2, D_2 \rangle$, *duas teorias default. Uma implementação* $\Im$, *denotada por* $\Sigma \rightsquigarrow \Sigma'$, *é um par* $\langle \zeta, \xi \rangle$ *tal que :*

i) $\Sigma' \leq_\xi \Delta$

ii) $\Sigma \xrightarrow{\zeta} \Delta$, *onde* ζ *representa uma função de tradução entre linguagens;*

Onde a teoria default $\Delta = \langle W_3, D_3 \rangle$, *apresenta ao menos uma extensão* default *(ou seja,* $ExtDef(\Delta) \neq \emptyset$*).*

O mesmo diagrama também identifica um passo canônico entre teorias *default.*



Esta definição é praticamente idêntica a definição do passo canônico no contexto da lógica clássica (II.3.1).

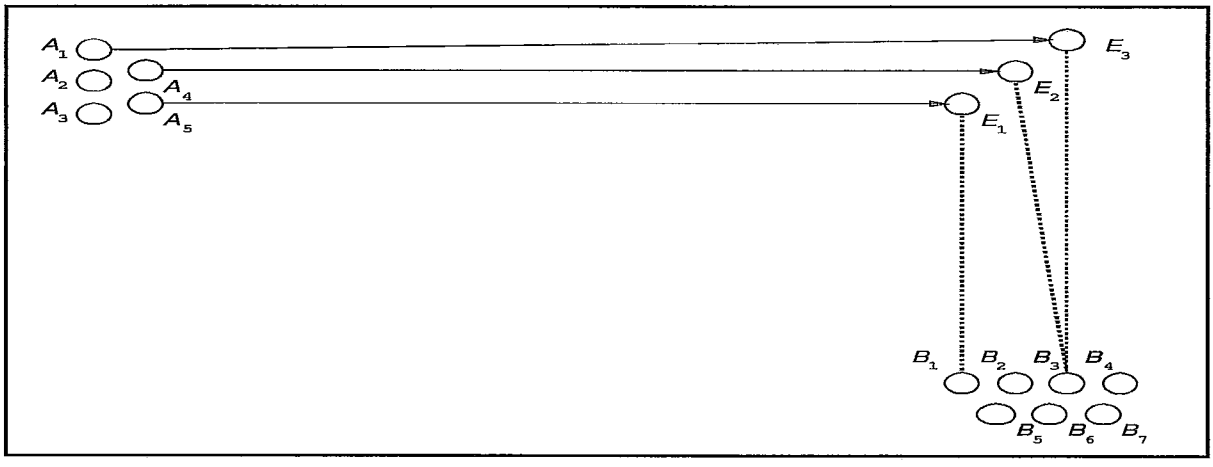


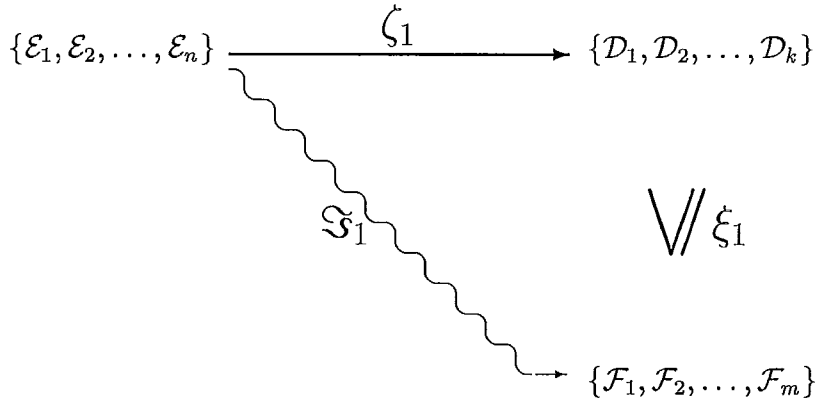
Figura V.2.1: O Passo Canônico visto pelos conjuntos de Extensões *Default*

Na figura V.2.1, nem todas as extensões *default* $\{A_1, A_2, \ldots, A_5\}$ de Σ foram escolhidas para serem implementadas, assim como somente as extensões *default* B_1, B_2 e B_3 de Σ' servem para suportar a implementação de Σ. Isto denota que decisões de projeto na interpretação de Σ selecionaram entre as admissíveis representações para o conceito abstrato quais deveriam ainda continuar este passo canônico, enquanto que na extensão conservativa de Σ' foi decidido o mecanismo lingüístico mais adequado para suportar esta implementação entre as representações diposníveis em $\{B_1, B_2, \ldots, B_7\}$.

A extensão *default* A_4 foi implementada no exemplo de duas maneiras diferentes, conforme foi escolhida B_2 ou B_3 para construir as extensões *default* E_2 ou E_3 da implementação declarativa Δ.

A imposição de uma teoria *default* Δ com pelo menos uma extensão default, para mediar o passo de implementação de Σ em Σ' encontra uma justificativa racional, porque visa impedir a escolha de uma implementação declarativa sem extensões default, que em termos práticos, seria completamente desprovida de significado, embora de acordo com as definições V.1.1 e V.1.3, na seção anterior, fosse legitíma esta decisão. Uma teoria *default* deste modo trivial, nunca acrescenta qualquer informação relevante ou representa refinamentos ou decisões de projeto, e principalmente tornaria viável a implementação de uma teoria *default* sobre qualquer outra indistintamente.

Todo o procedimento talvez seja melhor compreendido como se a implementação sucedesse realmente entre os conjuntos de extensões *default* de Σ, Σ' e Δ. Ou seja,



onde $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n\}$, $\{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m\}$ e $\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k\}$ são conjuntos de extensões *default* respectivamente para Σ, Σ' e Δ, e por definição, qualquer extensão *default* para Δ está relacionada a uma extensão *default* em $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n\}$, e a uma extensão *default* em $\{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m\}$.

Como $\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k\} \neq \emptyset$, existe pelo menos uma implementação de uma extensão *default* de Σ em uma extensão *default* de Σ'.

O passo canônico entre teorias *default* aparentemente poderia, esquecendo por um momento a questão da monotonicidade, ser uma generalização de sua definição clássica. De maneira que, dado um conjunto de teorias de primeira ordem, algumas entre elas seriam preferidas para continuar a implementação, sobre um outro conjunto de teorias escolhido, igualmente de primeira ordem.

V.2.3 Formalização da Composição de Implementações

Ambos os teoremas que garantem capacidade de compor passos de implementação precisam estar formalmente transpostos para a *Lógica Default,* antes que o desenvolvimento de programas disponha dos novos recursos oferecidos pela composição de implementações neste contexto. Por conseguinte, a intenção desta seção é restabelecer o Teorema da Modularização e o da Construtibilidade entre teorias *default.*

V.2.3.1 Teorema da Modularização

Inicialmente, seja o Teorema da Modularização reenunciado a seguir,

Teorema V.2.1 (Teorema da Modularização (*Lógica Default*))
Considere as teorias default *fechadas consistentes* $\Sigma_2 = \langle W_1, D_1 \rangle$, $\Delta_1 = \langle W_2, D_2 \rangle$ *e* $\Delta_2 = \langle W_3, D_3 \rangle$, *onde* W_2 *e* W_3 *são conjunto finitos de sentenças respectivamente nas linguagens* $\mathcal{L}(\Delta_1)$ *e* $\mathcal{L}(\Delta_2)$. *Bem como,* D_2 *e* D_3 *são conjunto finitos de* defaults *também sobre as linguagens* $\mathcal{L}(\Delta_1)$ *e* $\mathcal{L}(\Delta_2)$. *Caso*

$$\begin{array}{ccc} \Delta_1 & & \\ \mathbin{/\!/} & & \\ \Sigma_2 & \xrightarrow{\zeta_1} & \Delta_2 \end{array}$$

Então, existe pelo menos uma teoria default *fechada* $\Delta_{2,1} = \langle W_4, D_4 \rangle$, *tal que*

$$\begin{array}{ccc} \Delta_1 & \xrightarrow{\zeta_2} & \Delta_{2,1} \\ \mathbin{/\!/} & & \mathbin{/\!/} \\ \Sigma_2 & \xrightarrow{\zeta_1} & \Delta_2 \end{array}$$

Prova : Inicialmente, será construida uma teoria *default* $\Delta_{2,1}$, que seja uma interpretação para Δ_1, e então mostra-se que a teoria *default* $\Delta_{2,1}$ escolhida, também estende conservativamente Δ_2, ou seja,

i) $\Delta_1 \xrightarrow{\zeta_2} \Delta_{2,1}$

ii) $\Delta_2 \leq \Delta_{2,1}$

onde interpretação e extensão conservativa estão em acordo com as definições V.1.1, V.1.3 respectivamente.

Todavia, a partir de uma candidata a teoria *default* $\Delta_{2,1}$, é possivel reestabelecer *i*) e *ii*) alternativamente por intermédio de uma certa relação total $\mathbf{h}_r$ e uma função $\mathbf{h}_f$, ambas sobre o conjunto de extensões *default* de $\Delta_{2,1}$, definidas de modo a satisfazer:

i) $\mathbf{h}_r \subseteq ExtDef(\Delta_{2,1}) \times ExtDef(\Delta_1)$ onde se $\mathcal{G}\ \mathbf{h}_r\ \mathcal{C}$ então $\mathcal{C} \xrightarrow{\zeta_2} \mathcal{G}$

ii) $\mathbf{h}_f : ExtDef(\Delta_{2,1}) \longmapsto ExtDef(\Delta_2)$ onde $\mathbf{h}_f(\mathcal{G}) \leq \mathcal{G}$

Para tanto seja uma função de tradução ζ_2 sobre a linguagem de Δ_1, estendendo ζ_1 naturalmente, tal que $\zeta_2[\Sigma_2] = \zeta_1[\Sigma_2]$, e sempre é possível definir ζ_2 tal que $\mathcal{L}(\zeta_2[\Delta_1]) \cap \mathcal{L}(\Delta_2) \subseteq \mathcal{L}(\zeta_1[\Sigma_2])$. Mas por definição, conforme V.1.1,

$\Sigma_2 \xrightarrow{\zeta_1} \Delta_2$ existe pelo menos uma relação total, por exemplo r, do conjunto de extensões *default* de Δ_2 sobre as extensões *default* de Σ_2, isto é $\mathbf{r} \subseteq ExtDef(\Delta_2) \times ExtDef(\Sigma_2)$;

Igualmente, por V.1.3

$\Sigma_2 \leq \Delta_1$ existe pelo menos uma função, por instância f, do conjunto de extensões *default* de Δ_1 sobre as extensões *default* de Σ_2, isto é $f : ExtDef(\Delta_1) \longmapsto ExtDef(\Sigma_2)$

Seja uma relação auxiliar Ω definida sobre o produto cartesiano das extensões *default* de Δ_1 e Δ_2, mais formalmente

$$\Omega \subseteq ExtDef(\Delta_1) \times ExtDef(\Delta_2)$$

onde

$$\langle \mathcal{B}, \mathcal{C} \rangle \in \Omega \quad \Leftrightarrow \quad \mathcal{C}\ \mathbf{r}\ f(\mathcal{B})$$

Deste modo, pertencem a relação Ω, pares de extensões *default* para Δ_1 e Δ_2 que estejam associados, respectivamente por f e r, com a mesma *extensão default* em Σ_2.

Existem dois casos a serem considerados, dependendo da relação Ω, a saber:

Primeiro Caso : $\Omega = \emptyset$

Isto significa que o conjunto de extensões *default* de Δ_1 ou de Δ_2 é vazio, ou ainda que nenhuma *extensão default* para Δ_1 está associada por f com outra em Σ_2 que já esteja na imagem de $\mathbf{r}$, relacionada a alguma *extensão default* para Δ_2.

Em qualquer caso seria suficiente tomar a teoria *default* $\Delta_{2,1} = \langle W_4, D_4 \rangle$ de maneira que não hajam extensões *default*, ou seja por instância,

$$W_4 = \emptyset$$
$$D_4 = \left\{ \frac{: \mathbf{true}}{\mathbf{false}} \right\}$$

Neste caso verifica-se vacuamente o Teorema da Modularização, tomando como $\mathbf{h}_r$ uma relação vazia, e da mesma forma $\mathbf{h}_f$ uma função também vazia.

Embora sempre haja a possibilidade de recorrer a uma teoria sem extensões *default* capaz de satisfazer trivialmente ao teorema, não costuma ser satisfatório ou adequado utilizá-la. Quando possível é desejável exibir uma teoria com extensões *default* para completar o diagrama. Se Ω é não vazia isto pode ser possível, conforme será comprovado a seguir. Porém, o contrário verifica-se sempre, se uma teoria $\Delta_{2,1}$, que apresenta pelo menos uma *extensão default*, atende ao teorema, então a relação Ω é necessariamente não vazia. Este resultado está demonstrado na proposição V.2.1.

Segundo Caso : $\Omega \neq \emptyset$

Qualquer extensão *default*, por instância $\mathcal{B}$, para Δ_1 será finitamente axiomatizável, segundo B.3.1. Portanto, considere $\mathbf{Ax}(\mathcal{B})$ a conjunção de todas as sentenças em uma axiomatização[5] finita para $\mathcal{B}$.

Argumento análogo aplica-se em relação a extensão *default* para Δ_2, ou seja, também $\mathcal{C}$ será finitamente axiomatizável. E assim, seja $\mathbf{Ax}(\mathcal{C})$ a conjunção de todas as sentenças em uma axiomatização finita para $\mathcal{C}$.

Caso existam pares de extensões *default* satisfazendo a Ω, tem-se:

$$\Omega = \{ \ \langle \mathcal{B}_1, \mathcal{C}_1 \rangle, \langle \mathcal{B}_2, \mathcal{C}_2 \rangle, \ldots, \langle \mathcal{B}_n, \mathcal{C}_n \rangle \ \}$$

[5]veja A.1.12

Os índices em $\langle \mathcal{B}_i, \mathcal{C}_i \rangle$ não indicam qulquer enumeração no conjunto de extensões *default* de Δ_1 ou Δ_2, de modo que eventualmente $\mathcal{B}_i = \mathcal{B}_j$(o mesmo vale para $\mathcal{C}_i$), porém não há pares repetidos em Ω. A partir destes pares de extensões *default* procura-se construir um conjunto de teorias clássicas, que se constituirá nas extensões *default* para a teoria *default* $\Delta_{2,1}$ pretendida.

Seria razoável determinar estas teorias clássicas de modo a conter a ambas extensões *default* em cada par pertinente a Ω, contanto que o resultado desta união fosse coerente com as conjecturas assumidas na definição de cada uma. Para tanto seria necessário definir uma apresentação para $\Delta_{2,1}$ que fosse capaz de gerá-las como *extensões* default, e exclusivamente estas teorias.

A princípio, considere para cada par de extensões *default* em Ω, um *default* da seguinte forma:

$$d(\mathcal{B}_i, \mathcal{C}_i) = \left\{ \frac{:\bigwedge^{j \neq i} \neg(\ \zeta_2[\,\mathbf{Ax}(\mathcal{B}_j)\,]\ \wedge\ \mathbf{Ax}(\mathcal{C}_j)\ )\ \wedge\ \mathbf{Coerencia}(\mathcal{B}_i)}{\zeta_2[\mathbf{Ax}(\mathcal{B}_i)]\ \wedge\ \mathbf{Ax}(\mathcal{C}_i)} \right\}$$

onde,

$$\mathbf{Coerencia}(\mathcal{B}_i) = \zeta_2[\mathbf{Justificativas}\,(\mathbf{GD}\ (\mathcal{B}_i, \Delta_1))]$$

Então, por intermédio de Ω, constrõe-se a teoria *default* $\Delta_{2,1} = \langle W_4, D_4 \rangle$ da seguinte forma

$$\begin{aligned} W_4 &= \emptyset \\ D_4 &= \{d(\mathcal{B}_i, \mathcal{C}_i)\ \setminus\ \langle \mathcal{B}_i, \mathcal{C}_i \rangle \in \Omega\} \end{aligned}$$

Uma vez definida a nova teoria *default* $\Delta_{2,1}$, resta comprovar que existem uma relação total $\mathbf{h}_r$ e uma função $\mathbf{h}_f$, definida sobre $ExtDef(\Delta_{2,1})$, e que ainda satisfazem a i) e ii).

Analizando uma *extensão default* $\mathcal{G}$ particular de $\Delta_{2,1}$, é por definição,

$$\mathcal{G} = \mathbf{Cn}(W_4 \cup \mathbf{Consequentes}\,(\mathbf{GD}\ (\mathcal{G}, \Delta_{2,1})))$$

mas $W_4 = \emptyset$, e o primeiro termo na justificativa presente em cada *default* $d(\mathcal{B}_i, \mathcal{C}_i)$, garante que somente um seria aplicável na geração de cada *extensão default*. Talvez nem todos os *defaults* sejam capazes de gerar uma extensão *default*, porque o segundo termo assegura a coerência da *extensão default*. Um *default* só seria disparado caso seu consequente não

fosse capaz de inferir a negação da tradução de justificativas assumidas pelos geradores de $\mathcal{B}_i$.

Deste modo, se houver a *extensão default* gerada, onde foi aplicado o *default* $d(\mathcal{B}_i, \mathcal{C}_i)$, esta será exatamente a definida por:

$$\mathbf{Cn}(\ \zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i)\ )$$

Ora trivialmente obtem-se que, $\mathcal{B}_i \models \alpha$ se e somente se $\mathbf{Ax}(\mathcal{B}_i) \models \alpha$ também pelo Teorema da Tradução, $\mathcal{B}_i \models \alpha$ se e somente se $\zeta_2[\mathcal{B}_i] \models \alpha^{\zeta_2}$ portanto, $\mathcal{B}_i \models \alpha$ se e somente se $\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \models \alpha^{\zeta_2}$ e finalmente tem-se que,

$$\mathcal{B}_i \xrightarrow{\zeta_2} \mathbf{Cn}(\ \zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i)\ )$$

simplesmente ignorando conclusões deriváveis a partir de $\mathbf{Ax}(\mathcal{C}_i)$, pois $\zeta_2[\mathbf{Ax}(\mathcal{B}_i)]$ já é suficiente *per si*, para derivar a tradução de qualquer sentença em $\mathcal{E}_i$.

De maneira similar, tem-se que $\mathcal{C}_i \models \alpha$ se e somente se $\mathbf{Ax}(\mathcal{C}_i) \models \alpha$, consequentemente,

$$\mathcal{C}_i \subseteq \mathbf{Cn}(\ \zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i)\ )$$

sem levar em conta sentenças deriváveis a partir de $\zeta_2[\mathbf{Ax}(\mathcal{B}_i)]$.

Finalmente, resta apenas confirmar a conservatividade desta extensão, porém, por construção de Ω, está definido que $f(\mathcal{B}_i) \in ExtDef(\Sigma_2)$, e $\mathcal{C}_i \,\mathbf{r}\, f(\mathcal{B}_i)$, logo é possível estabelecer o diagrama a seguir, para qualquer *extensão default* de $\Delta_{2,1}$,

$$\begin{array}{ccc} \mathcal{B}_i & \xrightarrow{\zeta_2} & \mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i)) \\ \mathbin{\rotatebox{90}{$\Vdash$}} & & \mathbin{\rotatebox{90}{$\vdash$}} \\ f(\mathcal{B}_i) & \xrightarrow{\zeta_1} & \mathcal{C}_i \end{array}$$

porém fazendo que $\mathcal{L}(\zeta_2[\Delta_1]) \cap \mathcal{L}(\Delta_2) = \mathcal{L}(\zeta_1[\Sigma_2])$ e então, recorrendo a versão clássica para o Teorema da Modularização, pode ser determinada a conservatividade, ou seja,

$$\begin{array}{ccc} \mathcal{B}_i & \xrightarrow{\zeta_2} & \mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i)) \\ \mathbin{\rotatebox{90}{$\Vdash$}} & & \mathbin{\rotatebox{90}{$\Vdash$}} \\ f(\mathcal{B}_i) & \xrightarrow{\zeta_1} & \mathcal{C}_i \end{array}$$

Mas conforme visto, qualquer *extensão default* para $\Delta_{2,1}$, se acaso existir alguma, estará construida como a união de uma *extensão default*, $\mathcal{B}_i$, para Δ_1 e uma, $\mathcal{C}_i$, para Δ_2, relaciondas em consonância com Ω, portanto, será sempre da forma:

$$\mathcal{G} = \mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i))$$

Então, é diretamente possível encontrar uma relação total $\mathbf{h}_r$, de tal modo que:

$$(\forall \mathcal{G} \in ExtDef(\Delta_{2,1})) \setminus \mathcal{G} \ \mathbf{h}_r \ \mathcal{B}_i \quad \Leftrightarrow$$
$$\mathcal{G} = \mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i))$$

da mesma forma, definimos a função $\mathbf{h}_f$ também como:

$$(\forall \mathcal{G} \in ExtDef(\Delta_{2,1})) \setminus \mathbf{h}_f(\mathcal{G}) = \mathcal{C}_i \quad \Leftrightarrow$$
$$\mathcal{G} = \mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{C}_i))$$

Ainda que $\Delta_{2,1}$ não apresente nenhuma extensão *default* (embora Ω esteja sendo considerada não vazia), a relação total $\mathbf{h}_r$ e a função $\mathbf{h}_f$ estão apropriadamente definidas.

Assim portanto conclue-se diretamente que $\mathbf{h}_r$ e $\mathbf{h}_f$ atendem respectivamante a *i*) e *ii*), conforme desejado.

□

Convém algumas explicações sobre alguns aspectos da demonstração, enfatizando o "porquê" da construção escolhida para a teoria *default* $\Delta_{2,1}$.

De início, quanto as condições exigidas no enunciado, isto é, que as teorias *default* envolvidas sejam fechadas e com conjuntos finitos de axiomas e *defaults*, pretende garantir que todas as extensões *default* geradas sempre sejam finitamente axiomatizáveis. Substituir a referência explícita a todo conjunto de conseqüências de uma extensões *default*, facilita sobremodo definir a união (conforme foi construida a teoria *default* $\Delta_{2,1}$).

O Teorema da Modularização sempre será satisfeito, ainda que vacuamente, por uma teoria *default* $\Delta_{2,1}$ trivial. Há situações em que a opção pela vacuidade é inevitável, entretanto interessa na realidade aqueles casos onde $\Delta_{2,1}$ pode apresentar extensões *default* e ainda satisfazer o enunciado, porque estritamente neste casos é correta a composição de passos de implementação.

Enquanto que na lógica clássica, o Teorema da Modularização era suficiente para garantir que a composição de passos de canônicos subsequentes sempre se constituia

em um novo passo, representando a implementação direta da teoria inicial para a teoria destino, no contexto da *Lógica Default* isto não se verifica obrigatoriamente. Esta era uma caracteristica interessante, mas da qual houve necessidade de abrir mão, em troca de liberar as restrições impostas pela monotonicidade. Porém, a situação é amenizada porque é factível antever, sem maiores dificuldades, as circunstâncias que poderiam conduzir a problemas, e assim prevenir-se antecipadamente contra ocorrências indesejáveis.

Quando o próximo passo canônico não é perfeitamente componível com o presente, o desenvolvimento está seguindo por um caminho infrutífero.

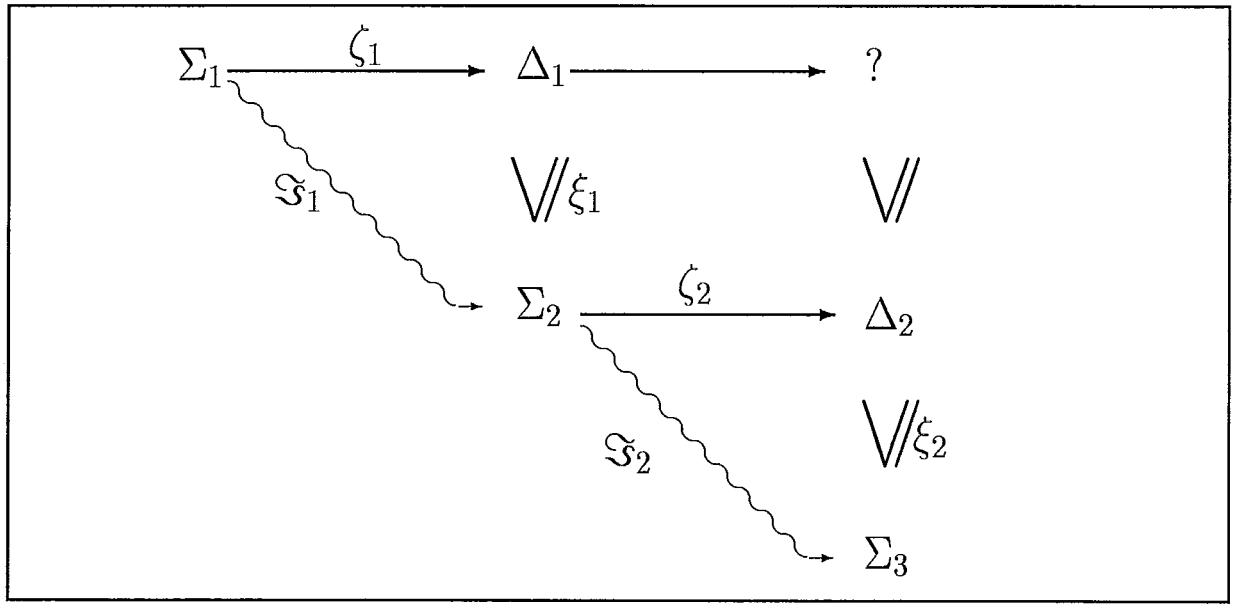
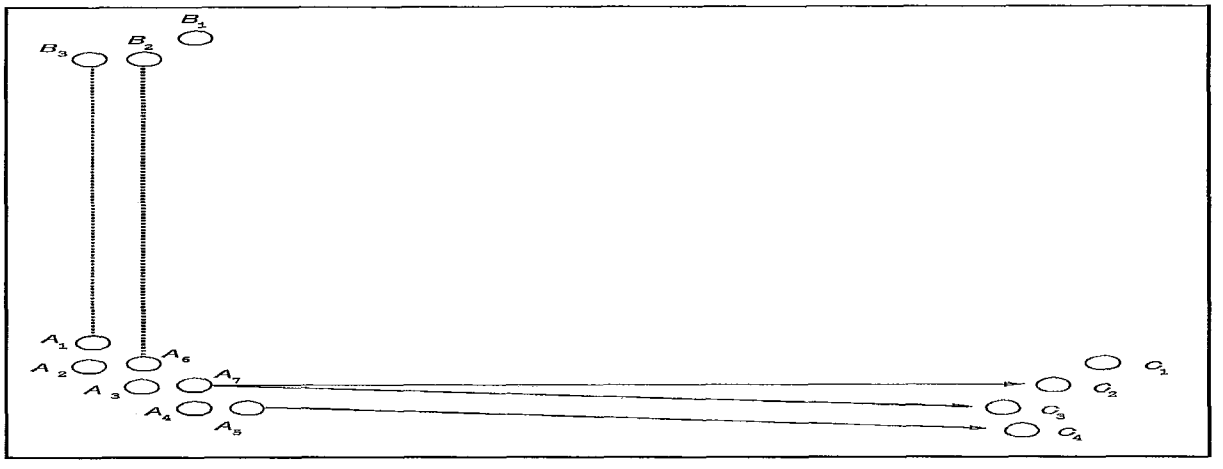


Figura V.2.2: A tentativa de compor $\Im_1$ e $\Im_2$ pode não resultar em um Passo Canônico

Supondo que o Teorema da Modularização esteja sendo aplicado entre os dois passos canônicos consecutivos $\Im_1$ e $\Im_2$, então por definição (V.2.1), tanto Σ_2 quanto Δ_2 geram extensões *default*. Logo, os possíveis problemas reduzem-se a duas situações, de acordo com a demonstração :

- Quando as extensões *default* da implementação declarativa Δ_1 estão baseadas em um mecanismo lingüístico (representado nas extensões *default* de Σ_2 na imagem de f), que não coincide, de modo algum, com a extensões *default* de Σ_2 (pertencendo a imagem de r), escolhidas para continuar o desenvolvimento, e sendo implementadas em termos de extensões *default* para a implementação declarativa Δ_2. Por instância, na figura seguinte seja $\{B_1, B_2, B_3\}$ as extensões *default* de Δ_1, e $\{C_1, C_2, C_3, C_4\}$ as de Δ_1. Embora B_1 e B_2 estejam definidas em termos de A_1 e A_6, foram as extensões *default* A_5 e A_7 as escolhidas para continuar a implementação. De $\Im_1$ para $\Im_2$

foi perdido o seqüenciamento da implementação, esquecendo completamente o que realmente precisaria ser implementado.



- Ou mesmo quando algumas propriedades relativas a uma extensão *default* $\mathcal{B}$ de Δ_1 talvez sejam conclusões apoiadas na ausência de certos detalhes a respeito da extensão *default* $f(\mathcal{B})$ de Σ_2, que são assumidos como inválidos porque não há confirmação explícita de sua veracidade. Este padrão de raciocínio é bastante comum em *defaults*. Porém nada impede que a interpretação de $f(\mathcal{B})$ ($\mathcal{C}$ r $f(\mathcal{B})$ para alguma extensão *default* $\mathcal{C}$ de Δ_2) introduza justamente alguns entre estes detalhes. Resumindo, $\mathcal{B}$ e $\mathcal{C}$ podem tirar conclusões contraditórias a respeito das mesmas informações não disponíveis em $\mathcal{C}$ r $f(\mathcal{B})$, e portanto não parece ser coerente admitir a conjunção de propriedades em $\mathcal{C}$ com as conclusões de $\mathcal{B}$ derivadas a partir da indefinição destas mesmas propriedades. Nesta circusntância optou-se por descartar inteiramente as extensões *default* para $\Delta_{2,1}$ onde isto aconteceria. Logo, sendo pessimista, possivelmente há situações em que não resta a $\Delta_{2,1}$ nenhuma extensão *default*, porque sempre houve alguma interferência indesejada de conclusões pertencentes às extensões *default* de Δ_2 com as hipóteses assumidas na geração das extensões *default* de Δ_1.

Na primeira situação, o Teorema da Modularização poderia apresentar como resultado aceitável exclusivamente uma teoria *default* sem extensões *default*, bloqueando assim a continuidade do desenvolvimento, pois não é possível compor, deste modo, estes dois passos de implementação. A tentativa de implementação direta de Σ_1 sobre Σ_3 não constitue, por definição um passo canônico.

Quando surge na prática, este impedimento, há uma clara indicação que algo foi feito incorretamente, o último passo canônico não atende aos objetivos para prosseguir o desenvolvimento, e portanto deve ser tentado outra vez. A proposição seguinte confirma que dentro das definições apresentadas, uma teoria *default* trivial é o único resultado admissível para o Teorema da Modularização nestas circunstâncias.

Proposição V.2.1 *Seja* $\Delta_{2,1}$ *uma teoria* default *fechada consistente, que completaria o diagrama do Teorema da Modularização, conforme abaixo,*

$$\begin{array}{ccc} \Delta_1 & \xrightarrow{\zeta_2} & \Delta_{2,1} \\ \mathbb{V} & & \mathbb{V} \\ \Sigma_2 & \xrightarrow{\zeta_1} & \Delta_2 \end{array}$$

Considere uma relação Ω *determinada sobre os conjunto de extensões* default *de* Δ_1 *e* Δ_2, *(* $\Omega \subseteq ExtDef(\Delta_1) \times ExtDef(\Delta_2)$ *), consoante com a construção utilizada para a demonstração do Teorema da Modularização, versão para Lógica Default,*

$$\langle \mathcal{B}, \mathcal{C} \rangle \in \Omega \quad \Leftrightarrow \quad \mathcal{C}\ r\ f(\mathcal{B})$$

onde $r \in ExtDef(\Delta_2) \times ExtDef(\Sigma_2)$ *é uma relação total, e que estabelece a interpretação* $\Delta_2 \xrightarrow{\zeta_1} \Sigma_2$; *igualmente* $f : ExtDef(\Delta_1) \longmapsto ExtDef(\Sigma_2)$ *define a extensão conservativa* $\Delta_1 \leq \Sigma_2$. *Então,*

$$\Omega = \emptyset \quad \Rightarrow \quad ExtDef(\Delta_{2,1}) = \emptyset$$

Prova : A princípio suponha que a relação Ω esteja realmente vazia, nenhum par de extensões *default* satisfaz a sua definição. Isto ocorre quando um dos conjuntos sobre o qual Ω está definida é também vazio, ou ainda caso nenhuma *extensão default* para Δ_1 esteja associada por f com outra em Σ_2 a qual já pertencesse também a r, relacionada com alguma *extensão default* para Δ_2. Mais formalmente, $\Omega = \emptyset$

- $ExtDef(\Delta_1) = \emptyset$
- $ExtDef(\Delta_2) = \emptyset$
- $(\forall\ \mathcal{B} \in ExtDef(\Delta_1))(\forall\ \mathcal{C} \in ExtDef(\Delta_2)) \setminus \langle \mathcal{B}, \mathcal{C} \rangle \notin \Omega$

A fim de demonstrar a proposição, será estabelecida antes a validade de sua contra-posição, mais precisamente,

$$ExtDef(\Delta_{2,1}) \neq \emptyset \;\Rightarrow\; \Omega \neq \emptyset$$

portanto, suponha inicialemente que existe uma tal teoria *default* $\Delta_{2,1}$ apresentando pelo menos uma *extensão default*, por exemplo $\mathcal{G}$, de tal modo que $ExtDef(\Delta_{2,1}) \neq \emptyset$.

Neste caso encontra-se, por definição uma relação total $\mathbf{h}_r$ e uma função $\mathbf{h}_f$, conforme abaixo:

- $\mathbf{h}_r \subseteq ExtDef(\Delta_{2,1}) \times ExtDef(\Delta_1)$ onde se $\mathcal{G}\ \mathbf{h}_r\ \mathcal{B}$ então $\mathcal{B} \xrightarrow{\zeta_2} \mathcal{G}$;
- $\mathbf{h}_f : ExtDef(\Delta_{2,1}) \longmapsto ExtDef(\Delta_2)$ onde se $\mathbf{h}_f(\mathcal{C}) \leq \mathcal{C}$.

Assim seja :

1. Uma vez que $\Delta_2 \leq \Delta_{2,1}$, necessariamente tem-se que

$$\mathbf{h}_f(\mathcal{G}) \in ExtDef(\Delta_2)$$

logo $ExtDef(\Delta_2) \neq \emptyset$.

2. Similarmente, visto que $\Delta_1 \xrightarrow{\zeta_2} \Delta_{2,1}$, implicaria que existe $\mathcal{B}$,

$$\mathcal{B}_\mathcal{G} \in ExtDef(\Delta_1)$$

tal que $\mathcal{G}\ \mathbf{h}_r\ \mathcal{B}$, seja $\mathcal{B}$ uma destas extensões *default* relacionadas com $\mathcal{G}$ por $\mathbf{r}$, consequentemente, $ExtDef(\Delta_1) \neq \emptyset$.

3. Resta confirmar agora que a existência de extensões *default* em $\Delta_{2,1}$ requer que haja pares correspondentes de extensões *default*, por exemplo $\langle \mathcal{B}_i, \mathcal{C}_i \rangle$, pertencentes a Ω. Posto isto, $\Omega \neq \emptyset$.

Novamente, considere $\mathcal{G} \in ExtDef(\Delta_{2,1})$, por intermédio da relação total $\mathbf{h}_r$, existe $\mathcal{B} \in ExtDef(\Delta_1)$ e tem-se que $\mathcal{G}\ \mathbf{h}_r \mathcal{B}_\mathcal{G}$, e pela função f, chega-se a $f(\mathcal{B}_\mathcal{G})$ como uma *extensão default* para Σ_2. Mas por definição, $f(\mathcal{B}_\mathcal{G}) \leq \mathcal{B}_\mathcal{G}$ e portanto, para qualquer $\alpha \in \mathcal{L}(\Sigma_2)$,

$$\mathcal{B}_\mathcal{G} \models \alpha \text{ se e somente se } f(\mathcal{B}_\mathcal{G}) \models \alpha$$

e por construção da relação total $\mathbf{h}_r$, $\mathcal{B}_\mathcal{G} \xrightarrow{\zeta_2} \mathcal{G}$, logo pelo teorema da tradução,

$$\text{Se } \mathcal{B}_\mathcal{G} \models \alpha \text{ então } \mathcal{G} \models \alpha^{\zeta_2}$$

então, resumindo,

$$\text{Se } f(\mathcal{B}_\mathcal{G}) \models \alpha \text{ então } \mathcal{G} \models \alpha^{\zeta_2}$$

Por outro lado, $\mathbf{h}_f$ estabelece uma *extensão default* em Δ_2, imagem de $\mathbf{h}_f(\mathcal{G})$, tal que, $\mathbf{h}_f(\mathcal{G}) \leq \mathcal{G}$ porém, a função de tradução ζ_1 interpreta a linguagem $\mathcal{L}(\Sigma_2)$ sobre a teoria Δ_2, e portanto a tradução de sua linguagem está contida na linguagem $\mathcal{L}(\Delta_2)$. Assim é possível concluir que,

$$\mathcal{G} \models \alpha^{\zeta_2} \text{ se e somente se } \mathbf{h}_f(\mathcal{G}) \models \alpha^{\zeta_2}$$

consequentemente,

$$\text{Se } f(\mathcal{B}_\mathcal{G}) \models \alpha \text{ então } \mathbf{h}_f(\mathcal{G}) \models \alpha^{\zeta_2}$$

Além do que, ζ_2 estende naturalmente ζ_1, ou seja, $\zeta_1[\Sigma_2] = \zeta_2[\Sigma_2]$, logo $\alpha^{\zeta_1} = \alpha^{\zeta_2}$, consequentemente,

$$\text{Se } f(\mathcal{B}_\mathcal{G}) \models \alpha \text{ então } \mathbf{h}_f(\mathcal{G}) \models \alpha^{\zeta_1}$$

então há uma interpretação de $f(\mathcal{B}_\mathcal{G})$ em $\mathbf{h}_f(\mathcal{G})$, segundo ζ_1, ou seja,

$$f(\mathcal{B}_\mathcal{G}) \xrightarrow{\zeta_1} \mathbf{h}_f(\mathcal{G})$$

e consequentemente, o par $\langle \mathbf{h}_f(\mathcal{G}), f(\mathcal{B}_\mathcal{G}) \rangle$ deverá pertencer, por construção a relação total $\mathbf{r}$ (i.e. $\mathbf{h}_f(\mathcal{G})\,\mathbf{r}\, f(\mathcal{B}_\mathcal{G})$) e finalmente $\langle \mathcal{B}_\mathcal{G}\,,\ \mathbf{h}_f(\mathcal{G}) \rangle$ estaria por definição em Ω, logo $\Omega \neq \emptyset$.

Concluindo, nenhuma das três possíveis condições para que Ω seja vazia verifica-se, caso seja assumido a existência de uma teoria *default* com um conjunto não vazio de *extensões* default, para satisfazer o enunciado do Teorema da Modularização. Isto torna válida a contra-positva da proposição acima.

Por conseguite, está plenamente justificada a escolha de $\Delta_{2,1}$ sem extensões *default*, no caso que Ω é vazia, durante a demonstração do Teorema da Modularização.

□

O segundo caso enumerado, também sugere erros generalizados nas decisões de refinamento ou detalhes. Ambos os passos precisam ser revistos para identificar a fonte dos problemas.

Entretanto, a teoria *default* $\Delta_{2,1}$ não trivial sugerida pela demonstração gera extensões *default* só e somente quando os resultados deduzidos estão coerentes com as hipótese assumidas em Δ_1. Cada possível extensão *default* para $\Delta_{2,1}$ é constituida realizando a união de um par de extensões *default* de Δ_1 e Δ_2, e é resultante exclusivamente da aplicação de um *default*, já que o consequente do *default* disparado sempre nega o primeiro termo na justificativa dos demais.

O artifício que determina a racionalidade das conclusões representadas, está expresso no segundo termo da justificativa dos *defaults* que compõem D_4 (**Coerencia**($\mathcal{B}_i$)). Se o consequente de um *default* contradiz sua própria justificativa, é porque alguma conclusão inferida na extensão *default* para Δ_2 não é consistente com as suposições representadas nas justificativas dos *defaults* que geram a extensão *default* para Δ_1.

A presença deste dispositivo é sustentada porque o consequente de um *default* não normal pode não carregar informação sobre as hipóteses levantadas em suas justificativas, e meramente proceder a conjunção das axiomatizações pode produzir conclusões que não deveriam ser conjuntamente aceitas. Portanto, o mais indicado é mesmo eliminar as extensões *default* para $\Delta_{2,1}$ incoerentes.

O oposto nunca ocorre, portanto não está previsto na construção da teoria *default* $\Delta_{2,1}$. Qualquer interferência fica restrita a linguagem de Σ_2, e como $\Sigma_2 \leq \Delta_1$, nenhuma extensão *default* poderia derivar propriedades em $\mathcal{L}(\Sigma_2)$ que não pertençam as conseqüências de sua imagem em Σ_2, logo, já estariam devidamente traduzidas na extensão *default* de Δ_2. Portanto, Δ_1 não é capaz de deduzir qualquer sentença em $\mathcal{L}(\Sigma_1)$, cuja a tradução Δ_2 já não fosse concluida em Δ_2.

De qualquer modo, o Teorema da Modularização conduz a composição de implementações subsequentes conforme esperado, fornecendo resultados viáveis apenas quando há neles sentido real. Mesmo quando não permite a composição de passos canônicos mostra indicações de falhas ou enganos cometidos durante o desenvolvimento.

V.2.3.2 Teorema da Construtibilidade

Finalmente, o Teorema da Construtibilidade merece uma readaptação à *Lógica Default.*

Teorema V.2.2 (Teorema da Construtibilidade (*Lógica Default*)) *Sejam teorias* default *consistentes* Σ_1, Σ_1', *e também* Σ_2, Σ_2', *e* Δ_1 *(estas últimas envolvendo conjuntos finitos de axiomas e conjuntos finitos de* defaults *fechados). Se*

$$\begin{array}{lll} \Sigma_2 & & \\ \Vdash & & \\ \Sigma_1 & \xrightarrow{\zeta_1} & \Delta_1 \\ & & \Vdash \\ & & \Sigma_1' \quad \subseteq \quad \Sigma_2' \end{array}$$

Então, existe pelo menos uma teoria default Δ_2, *cujo o conjunto de extensões* default *não é vazio, tal que* $\Delta_1 \subseteq \Delta_2$ *e*

$$\begin{array}{lll} \Sigma_2 & \xrightarrow{\zeta_2} & \Delta_2 \\ \Vdash & & \\ \Sigma_1 & \xrightarrow{\zeta_1} \Delta_1 & \Vdash \\ & \Vdash & \\ & \Sigma_1' \quad \subseteq \quad \Sigma_2' & \end{array}$$

Prova : Assim, será construida uma teoria *default* Δ_2, que seja uma interpretação para Σ_2, contenha Δ_1 e então mostra-se que a teoria *default* Δ_2 escolhida, também estende conservativamente Σ_2', ou seja,

i) $\Sigma_2 \xrightarrow{\zeta_2} \Delta_2$

ii) $\Delta_1 \subseteq \Delta_2$

iii) $\Sigma_2' \leq \Delta_2$

onde interpretação, extensão e extensão conservativa estão em acordo com as definições V.1.1, V.1.2 e V.1.3 respectivamente.

Todavia, a partir de uma candidata a teoria default Δ_2, é possivel reestabelecer *i*) *ii*) e *iii*) alternativamente por intermédio de relações totais $\mathbf{h}_r$, $\mathbf{h}_s$ e uma função $\mathbf{h}_f$, ambas sobre o conjunto de extensões de Δ_2, definidas de modo a satisfazer:

i) $\mathbf{h}_r \subseteq ExtDef(\Delta_2) \times ExtDef(\Sigma_2)$ onde se $\mathcal{G}\ \mathbf{h}_r\ \mathcal{B}$ então $\mathcal{B} \xrightarrow{\zeta_2} \mathcal{G}$

ii) $\mathbf{h}_s \subseteq ExtDef(\Delta_2) \times ExtDef(\Delta_1)$ onde se $\mathcal{G}\ \mathbf{h}_r\ \mathcal{D}$ então $\mathcal{D} \subseteq \mathcal{G}$

iii) $\mathbf{h}_f : ExtDef(\Delta_2) \longmapsto ExtDef(\Sigma_2')$ onde $\mathbf{h}_f(\mathcal{G}) \leq \mathcal{G}$

Para tanto, seja uma função de tradução ζ_2 sobre a linguagem de Σ_2, estendendo ζ_1 naturalmente, tal que $\zeta_2[\Sigma_1] = \zeta_1[\Sigma_1]$. É viável definir ζ_2 de forma que:

$$\begin{aligned} \mathcal{L}(\zeta_2[\Sigma_2]) \cap \mathcal{L}(\Delta_1 \cup \Sigma_2') &\subseteq \mathcal{L}(\zeta_1[\Sigma_1]) \\ \mathcal{L}(\Delta_1) \cap \mathcal{L}(\Sigma_2') &\subseteq \mathcal{L}(\Sigma_1') \end{aligned}$$

Mas por definição, conforme V.1.1,

$\Sigma_1 \xrightarrow{\zeta_1} \Delta_1$ existe pelo menos uma relação total, por exemplo r, do conjunto de extensões *default* de Δ_1 sobre as extensões *default* de Σ_1, isto é $\mathbf{r} \in ExtDef(\Delta_1) \times ExtDef(\Sigma_1)$;

Igualmente, por V.1.3

$\Sigma_1' \leq \Delta_1$ existe pelo menos uma função, por instância f, do conjunto de extensões *default* de Δ_1 sobre as extensões *default* de Σ_1, isto é $f : ExtDef(\Delta_1) \longmapsto ExtDef(\Sigma_1')$;

$\Sigma_1 \leq \Sigma_2$ existe pelo menos uma função, por instância g, do conjunto de extensões *default* de Σ_2 sobre as extensões *default* de Σ_1, isto é $g : ExtDef(\Sigma_2) \longmapsto ExtDef(\Sigma_1)$;

E segundo V.1.2,

$\Sigma_1' \subseteq \Sigma_2'$ existe pelo menos uma relação total, por exemplo s, do conjunto de extensões *default* de Σ_2' sobre as extensões *default* de Σ_1', isto é, $\mathbf{s} \in ExtDef(\Sigma_1') \times ExtDef(\Sigma_2')$.

Seja uma relação auxiliar Ξ definida sobre o produto cartesiano das extensões *default* de Σ_2, Δ_1 e Σ_2', mais formalmente

$$\Xi \subseteq ExtDef(\Sigma_2) \times ExtDef(\Delta_1) \times ExtDef(\Sigma_2')$$

onde $\langle \mathcal{B}, \mathcal{D}, \mathcal{C} \rangle \in \Xi \Leftrightarrow \{ \mathcal{D}\, \mathbf{r}\, g(\mathcal{B}) \wedge \mathcal{C}\, \mathbf{s}\, f(\mathcal{D}) \}$.

Deste modo, pertencem a relação Ξ, triplas de extensões *default* para Σ_2, Δ_1 e Σ_2 que estejam associados, respectivamente com as mesmas *extensão default* em Σ_1 e Σ_1'.

Existem dois casos a serem considerados, dependendo da relação Ξ, a saber:

Primeiro Caso : $\Xi = \emptyset$

Isto significa que o conjunto de extensões *default* de Σ_2 ou de Δ_1 ou ainda de Σ_2' é vazio, ou mesmo que nenhuma tripla de *extensão default* satisfaz a Ξ.

Em qualquer caso seria suficiente tomar a teoria *default* $\Delta_2 = \langle W_4, D_4 \rangle$ de maneira que não hajam extensões *default*, ou seja por instância,

$$W_4 = \emptyset$$
$$D_4 = \left\{ \frac{: \mathbf{true}}{\mathbf{false}} \right\}$$

Neste caso verifica-se vacuamente o Teorema da Construtibilidade, tomando como $\mathbf{h}_r$ $\mathbf{h}_s$ relações vazias, e da mesma forma $\mathbf{h}_f$ uma função também vazia.

Embora seja igualmente possível de recorrer a uma teoria sem extensões *default* satisfazendo trivialmente ao teorema, também não costuma ser satisfatório ou adequado utilizá-la. Quando possível é desejável exibir uma teoria *default* com extensões *default* para completar os diagramas. Se Ξ é não vazia isto pode ser possível, conforme será comprovado a seguir.

Segundo Caso : $\Xi \neq \emptyset$

De acordo com B.3.1, todas as extensões *default* para Σ_2, Δ_1 e Σ_2' são finitamente axiomatizáveis. Suponha $\mathbf{Ax}(\mathcal{B}_j)$, $\mathbf{Ax}(\mathcal{D}_j)$ e $\mathbf{Ax}(\mathcal{C}_j)$ respectivamente a conjunção de axiomatizações finitas para Σ_2, Δ_1 e Σ_2'.

Caso existam triplas de extensões *default* satisfazendo a Ξ, procura-se construir um conjunto de teorias clássicas, que se constituirá no conjunto de extensões *default* para a teoria *default* Δ_2 pretendida.

A princípio, considere para cada tripla de extensões *default* em Ξ, um *default* da seguinte forma:

$$d(\mathcal{B}_i, \mathcal{D}_i, \mathcal{C}_i) = \left\{ \frac{: \bigwedge^{j \neq i} \neg(\zeta_2[\mathbf{Ax}(\mathcal{B}_j)] \wedge \mathbf{Ax}(\mathcal{D}_j) \wedge \mathbf{Ax}(\mathcal{C}_j)) \wedge \mathbf{Coerencia}(\mathcal{B}_i) \wedge \mathbf{Coerencia}(\mathcal{D}_i)}{\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)} \right\}$$

onde,

$$\mathbf{Coerencia}(\mathcal{B}_i) \equiv \zeta_2[\mathbf{Justificativas}(\mathbf{GD}(\mathcal{B}_i, \Sigma_2))]$$
$$\mathbf{Coerencia}(\mathcal{D}_i) \equiv \mathbf{Justificativas}(\mathbf{GD}(\mathcal{D}_i, \Delta_1))$$

Então, por intermédio de Ξ, constrõe-se a teoria *default* $\Delta_2 = \langle W_4, D_4 \rangle$ da seguinte forma

$$W_4 = \emptyset$$
$$D_4 = \{ d(\mathcal{B}_i, \mathcal{D}_i, \mathcal{C}_i) \setminus \langle \mathcal{B}_i, \mathcal{D}_i, \mathcal{C}_i \rangle \in \Xi \}$$

Uma vez definida a nova teoria *default* Δ_2, resta comprovar que existem as relações totais $\mathbf{h}_r$, $\mathbf{h}_s$ e uma função $\mathbf{h}_f$, definida sobre $ExtDef(\Delta_2)$, e que ainda satisfazem a *i*), *ii*) e *iii*). Mas, analizando uma *extensão default* $\mathcal{G}$ particular de Δ_2, é por definição (B.3.2),

$$\mathcal{G} = \mathbf{Cn}(W_4 \cup \mathbf{Consequentes}\,(\mathbf{GD}\,(\mathcal{G}, \Delta_2)))$$

mas $W_4 = \emptyset$, e o primeiro termo na justificativa presente em cada *default* $d(\mathcal{B}_i, \mathcal{D}_i, \mathcal{C}_i)$, garante que somente um seria aplicável na geração de cada *extensão default*. Talvez nem todos os defaults sejam capazes de gerar uma *extensão* default, porque o segundo e o terceiro termos asseguram a coerência da *extensão default*, pois o *default* só seria disparado caso seu consequente não fosse capaz de inferir a negação da tradução de justificativas assumidas pelos geradores de $\mathcal{B}_i$ e $\mathcal{D}_i$.

Deste modo, se houver a *extensão default* gerada, onde foi aplicado o *default* $d(\mathcal{B}_i, \mathcal{D}_i, \mathcal{C}_i)$, esta será exatamente a definida por:

$$\mathbf{Cn}(\ \zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)\ )$$

Ora trivialmente obtem-se que,

$$\begin{array}{lcl} \mathcal{B}_i & \xrightarrow{\zeta_2} & \mathbf{Cn}(\ \zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)\ ) \\ \mathcal{D}_i & \subseteq & \mathbf{Cn}(\ \zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)\ ) \\ \mathcal{C}_i & \subseteq & \mathbf{Cn}(\ \zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)\ ) \end{array}$$

Finalmente, resta apenas confirmar a conservatividade desta última extensão, contudo, por construção de Ξ, está definido que $\mathcal{D}_i\,\mathbf{r}\ g(\mathcal{B}_i)$, onde $\mathcal{D}_i \in ExtDef(\Sigma_1)$, e também $\mathcal{C}_i\,\mathbf{s}\ f(\mathcal{D}_i)$, onde $\mathcal{C}_i \in ExtDef(\Sigma_1')$, e logo é possível estabelecer os diagramas a seguir, para qualquer *extensão default* de Δ_2,

$$\begin{array}{ccccc} \mathcal{B}_i & & \xrightarrow{\zeta_2} & & \mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)) \\ \mathbb{V} & & & & \\ g(\mathcal{B}_i) & \xrightarrow{\zeta_1} & \mathcal{D}_i & & \cup\!| \\ & & \mathbb{V} & & \\ & & f(\mathcal{D}_i) & \subseteq & \mathcal{C}_i \end{array}$$

porém foi assumido que

$$\begin{array}{rcl} \mathcal{L}(\zeta_2[\Sigma_2]) \cap \mathcal{L}(\Delta_1 \cup \Sigma_2') & = & \mathcal{L}(\zeta_1[\Sigma_1]) \\ \mathcal{L}(\Delta_1) \cap \mathcal{L}(\Sigma_2') & = & \mathcal{L}(\Sigma_1') \end{array}$$

e então, recorrendo a versão clássica para o Teorema da Construtibilidade, pode ser determinada a conservatividade de $\mathcal{C}_i \leq \mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i))$

Mas conforme visto, qualquer *extensão default* para Δ_2, se acaso existir alguma, estará construida como a união de uma *extensão default*, $\mathcal{B}_i$ para Σ_2, devidamente traduzida por ζ_2, uma, $\mathcal{D}_i$, para Δ_1, e outra $\mathcal{C}_i$ para Σ_2' relaciondas em consoância com Ξ, portanto, será sempre da forma:

$$\mathbf{Cn}(\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i))$$

Então, é diretamente possível encontrar uma relação total $\mathbf{h}_r$, de tal modo que:

$$\begin{aligned}(\forall\, \mathcal{G} \in \mathit{ExtDef}(\Delta_2))\, \mathcal{G} \;\; \mathbf{h}_r \;\; \mathcal{B}_i \quad &\Leftrightarrow \\ \mathcal{G} \;\; = \;\; &\mathbf{Cn}(\,\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)\,)\end{aligned}$$

Uma relação total $\mathbf{h}_s$ tal que:

$$\begin{aligned}(\forall\, \mathcal{G} \in \mathit{ExtDef}(\Delta_2))\, \mathcal{G} \;\; \mathbf{h}_s \;\; \mathcal{D}_i \quad &\Leftrightarrow \\ \mathcal{G} \;\; = \;\; &\mathbf{Cn}(\,\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)\,)\end{aligned}$$

da mesma forma, definimos a função $\mathbf{h}_f$ também como:

$$\begin{aligned}(\forall\, \mathcal{G} \in \mathit{ExtDef}(\Delta_2))\, \mathbf{h}_f(\mathcal{G}) \;\; = \;\; \mathcal{C}_i \quad &\Leftrightarrow \\ \mathcal{G} \;\; = \;\; &\mathbf{Cn}(\,\zeta_2[\mathbf{Ax}(\mathcal{B}_i)] \wedge \mathbf{Ax}(\mathcal{D}_i) \wedge \mathbf{Ax}(\mathcal{C}_i)\,)\end{aligned}$$

Ainda que Δ_2 não apresente nenhuma extensão *default* (embora Ξ esteja sendo considerada não vazia), as relações total $\mathbf{h}_r$ $\mathbf{h}_s$ e a função $\mathbf{h}_f$ estão apropriadamente definidas. Assim portanto conclue-se diretamente que $\mathbf{h}_r, \mathbf{h}_s$ e $\mathbf{h}_f$ atendem respectivamante a *i*), *ii*) e *iii*), conforme desejado. Portanto, vale também a versão para Lógica *Default* do Teorema da Construtibilidade.

□

Repetindo o comportamento encontrado na seção anterior, o Teorema da Construtibilidade também aceita uma teoria *default* Δ_2 sem extensões *default*, satisfazendo-o por vacuidade. Por isso, não há convicção que um novo passo canônico (nos moldes da definição V.2.1) invariavelmente resulta da tentativa de estender o nível abstrato e/ou concreto de um passo canônico subjacente. Logo, nem sempre é factível proceder a composição com uma implementação subjacente.

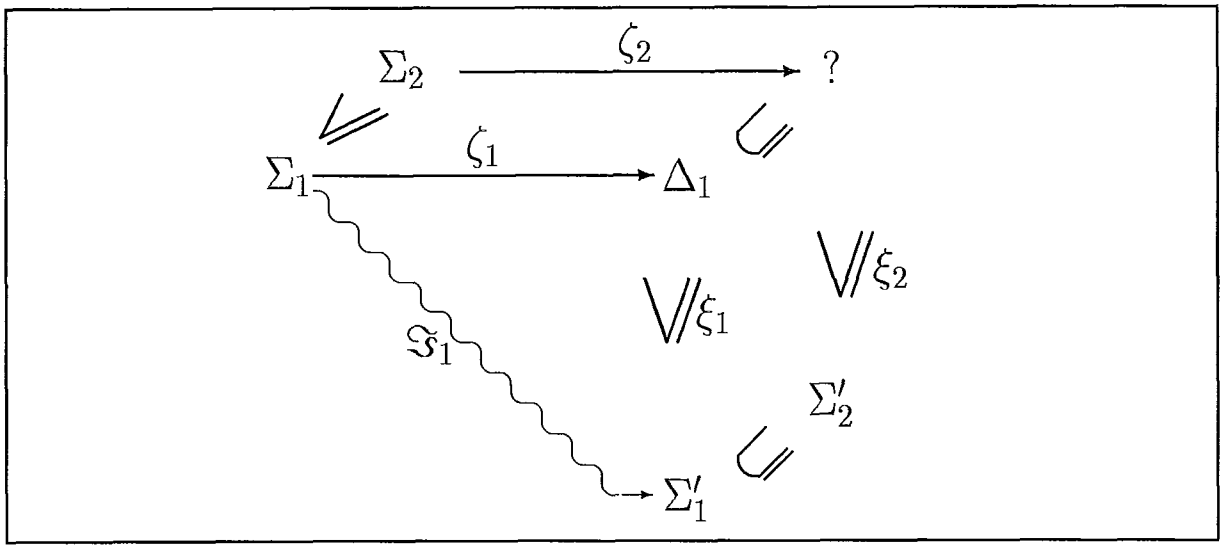


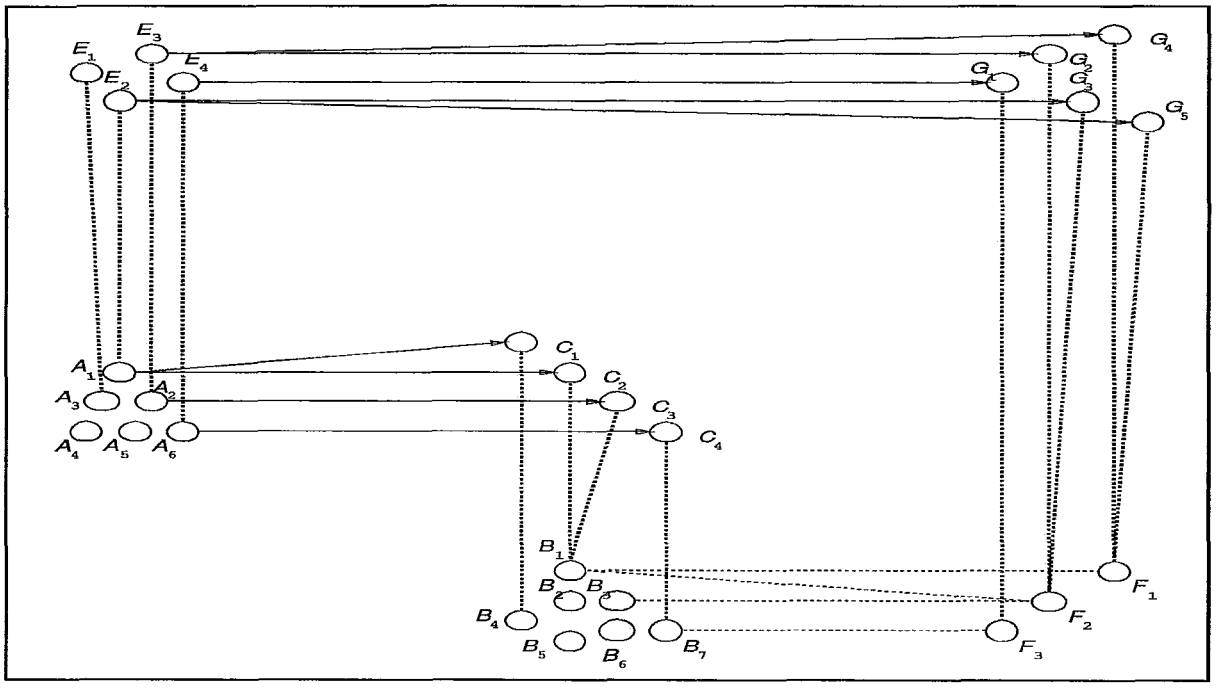
Figura V.2.3: Tentar implementar Σ_2 em Σ'_2 pode não resultar em um Passo Canônico

Novamente, há meios para identificar quais situações deveriam ser evitadas na realidade, observando como as extensões *default* para Σ_2 e Σ'_2 relacionam-se por intermédio de extensões *default* para Δ_1. As relações totais e funções envolvidas nas interpretações e extensões conservativas fornecem algumas informações necessárias para antecipar a impossibilidade de composição. Além disto, restaria confirmar se as extensões *default* de Σ'_2, que estão propriamente relacionadas, são capazes de deduzir conseqüências incoerentes com as suposições, assumidas nas extensões *default* correspondentes em Σ_2 ou Δ_1. Logo, quando o Teorema da Construtibilidade não admite compor um novo passo canônico por sobre um outro subjacente, existe uma explicação intuitiva que sustenta este resultado, em última análise, a ligação entre os passos está perdida com a composição.

Exemplificando, se tanto Σ_2 quanto Σ'_2 e Δ_1 possuem extensões *default*, e contudo não satisfazem a relação auxiliar Ξ, ou acontece que Σ_2 está sendo enriquecido, mas as propriedades acrescentadas não se referem a nenhuma das extensões *default* de Σ_1 efetivamente implementadas no passo canônico subjacente, ou ao contrário, as decisões de refinamento e detalhes representandos em Σ'_2 não dizem respeito a implementação declarativa Δ_1. Em qualquer das situações, o novo passo canônico que se pretende compor nada tem haver com o passo subjacente. Mesmo que Ξ seja não vazia, se uma possível extensão *default* para Δ_2 é anulada então seriam incoerentes suas conclusões, mas uma vez o teorema funciona conforme as expectativas.

Na figura a seguir, uma composição de um novo passo canônico está representada pelas extensões *default*. Ilustrando como a implementação pré-existente de A_3 em

B_1 seviu como base para implementar E_3 em termos de F_1 e F_2, dando origem respectivamente às extensões *default* G_3 e G_4 de Δ_2 resultante da aplicação do Teorema da Construtibilidade.



V.3 Referências Bibliográficas

Uma excelente referência para entender a intuição que fundamenta padrões de raciocínio não monotônicos e os formalismo que os definem é [Luk90]. A definição dos princípais conceitos da *Lógica Default,* utilizada aqui, pode ser encontrada em [Rei80], de acordo com a sua proposição original, contudo, ainda em [Luk90], há um capítulo dedicado à *Lógica Default*, suficiente para esclarecer o que foi assumido neste trabalho.

Uma primeira tentativa de explorar a *Lógica Default* para a implementação de tipos abstratos de dados está em [AV93a], também em [AV93b], propondo definições para os conceitos de interpretação e extensão conservativa no novo contexto, e segundo estas, demonstrando o Teorema da Modularização. Comparativamente, aqui foram propostas definições mais flexíveis para interpretação e extensão conservativa entre teorias *defaults*, e fundado em novas definições, os dois mecanismos de composição de implementações (apresentados no capítulo III) estão agora reestabelecidos.

Em [AV93a] e em [AV93b] está condicionada a existência de uma função bijetiva para relacionar o conjunto de extensões *default* de uma teoria *default* com o correspondente conjunto de extensões *default* da teoria que a interpreta (ou a estende). Deste modo, a existência de uma intepretação (ou de uma extensão conservativa) fica limitada a teorias *default* com exatamente o mesmo número de extensões *default*. As definições assim propostas são restritivas para as possíveis aplicações práticas. Portanto, está justificada a necessidade de relaxar as definições anteriores de interpretação e extensão conservativa apresentadas em [AV93a] e depois em [AV93b] e transferir para a definição do passo canônico a responsabilidade de manter coerentes os resultados de um passo de implementação.

Capítulo VI

Aplicações da Composição de Implementações no Contexto da *Lógica Default*

A aplicação prática de formalismos não monotônicos, a *Lógica Default* entre eles, à especificação e ao desenvolvimento de programas ainda é incipiente, porém mostra-se promissora.

Situado devidamente o contexto da *Lógica Default* , algumas seções do capítulo IV serão agora revisadas, abordando os mesmos aspectos sob ângulos mais liberais, e procurando tratar algumas questões apontadas superficialmente antes.

A primeira seção revê a estratégia de desenvolvimento construtivo, enfatizando o que é ganho na troca de contexto. Também a capacidade de absorver alterações *a posteriori* é extremamente favorecida com a perda da monotonicidade, segundo descreve a seção VI.2.

A seção VI.3 esboça idéias sobre o tratamento de exceção e condições de erro, propositalmente esquecido no contexto clássico, quando não havia muito a acrescentar. E finalizando, a última seção volta a falar brevemente sobre reusabilidade.

VI.1 Desenvolvimento Construtivo no Contexto da *Lógica Default*

Filosoficamente, a ótica do desenvolvimento acaba modificada por completo, no contexto não monotônico. Em lugar de tentar identificar e representar todas as propriedades que caracterizam singularmente um conceito, toma-se como ponto de partida um determinado número de conjuntos, constituidos por propriedades supostamente cabíveis, isto é, algumas representações hipoteticamente aceitáveis para descrevê-lo sob formas alternativas.

Eventualmente, estas representações guardam em comum, algumas propriedades que indiscutivelmente são válidas, e sobre estas, é que algumas representações alternativas podem ser postuladas, explorando hipóteses consistentes para complementar os pontos ainda lacônicos. Associa-se claramente o papel dos axiomas e dos *defaults* para especificar TAD's neste contexto.

Especificada a visão inicial do conceito, ou melhor o conjunto de visões possíveis, a implementação segue procurando encontrar aquelas conjecturas que não seriam realmente adequadas, pois cedo ou tarde, elas revelam-se contraditórias sob a perspectiva de novas informações apreendidas. As propriedades, quando se manifestam inadequadas, são eliminadas, apenas descartando-se todos os conjuntos que as consideraram antes como crenças coerentes. Assim, a implementação indiretamente seleciona entre as representações aceitas de início, aquelas que merecem atingir uma apresentação final, à medida que vão sendo elucidadas as dúvidas restantes sobre o conceito.

Isto não implica que um conjunto particular de crenças permaneça inalterado, existem, ao contrário, mecanismos capazes de enriquecê-lo. Enquanto a implementação é conduzida, suposições adicionais, na forma de detalhes ou refinamentos, ajudam a caracterizar melhor as representações possíveis, ou desconsiderar nos passos subsequentes ou subjacentes, as hipóteses anteriores que não são mais satisfatórias.

Este processo de escolhas decide ao final, que representação, ou mesmo representações, são efetivamente a implementação para o conceito. Assim, o desenvolvimento abandonaria um paradigma extritamente sintético, em favor de um paradigma de certo modo analítico, favorecendo abordagens evolutivas ou a prototipagem, de acordo com [Som89].

Não é difícil imaginar como a *Lógica Default* encaixa-se a estas idéias. As extensões *default* são justamente aqueles conjuntos de sentenças consistentes, representando

os modelos possíveis para um dado conjunto de axiomas, de acordo com as hipóteses que estão expressas por meio de *defaults.*

Convencidos que teorias defualt especificam apropriadamente TAD's, cabe discutir o seu comportamente durante o processo de implementação. Há basicamente dois pontos de vista a serem avaliados. O primeiro leva em consideração o conjunto de extensões *default* externamente, isto é:

- aumentar o número de extensões *default* de uma especificação, à medida que esta é implementada, implicaria em mais representações alternativas para o mesmo conceito, caracterizando propriedades distintas e possívelmente contraditórias, mas que deveriam agora prosseguir na implementação;
- diminuir o conjunto de extensões *defaults* significa proceder escolhas, decidindo quais as representações para o conceito parecem estar melhor apropriadas para continuar a implementação, de acordo com crenças antigas refutadas.

Apenas quando a intenção é particularizar ou individualizar múltiplas possibilidades de implementação, a partir de uma extensão *default* original, pretendendo destacar novas propriedades, é que se justifica aumentar o conjunto de extensões *default.* Mesmo assim, isto fornece indícios que há pontos duvidosos na implementação.

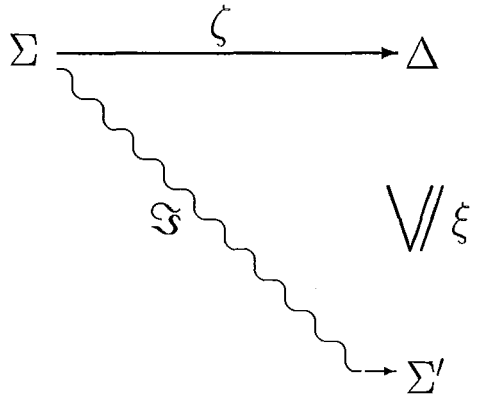
Por outro lado, observadas as extensões *default* individualmente, seria possível acontecer durante a implementação:

- um crescimento nas conseqüências inferíveis, porque mais propriedades foram identificadas e assumidas como coerentes, e neste caso, a representação torna-se mais precisa ou menos permissiva; ou o oposto,
- uma redução nas conseqüências derivadas, condicionando uma generalização de suas representações, ou seja, uma ampliação na classe de modelos associada.

Mas esta última situação não parece razoável, já que se opõe diretamente ao sentido da implementação, que objetiva sempre uma maior especificidade.

Resumindo, no contexto da *Lógica Default,* implementar condiciona fundamentamente reduzir o número de extensões *default,* enquanto as restantes são progressivamente enriquecidas com novas propriedades.

A implementação declarativa Δ, que media um passo canônico implementando Σ em Σ', deve ser definida em conformidade com estas idéias. Assim, na construção de Δ está envolvida a escolha, entre os mecanismos lingüísticos descritos por extensões *default* em Σ', de quais serão refinados para representar algumas entre as extensões *default* de Σ, decididas pela interpretação ζ, para uma efetiva implementação. Na construção do par ξ e ζ há espaço para decisões de projeto.



A necessidade de descrever classes de pilhas cujo crescimento é arbitrariamente limitado, será revista agora no contexto não monotônico, como uma forma de estabelecer uma comparação entre a espressividade de cada um.

Agora, há meios para explorar duas alternativas bem diferentes para o comportamento de pilha :

- no primeiro caso, após atingir o limite arbitrado, a pilha deve permanecer inalterada em relação a novas tentativas de empilhamento (exatamente a mesma idéia, já desenvolvida no exemplo da seção IV.3.1.2);
- no segundo caso, tudo transcorre normalmente, até o limite ser alcançado, quando então, a próxima tentativa de empilhar provoca a retirada do elemento mais antigo na fila, como forma de ceder lugar ao novo elemento sendo colocado.

Obviamente, não há como especificar ambos os casos em uma mesma teoria clássica. Portanto, o comportamento comum do TAD $BoundedStack[Char]$ será representado por axiomas de primeira ordem, e as opções para o comportamento da pilha quando o limite é atingido estarão descritas como hipóteses distinta e incompatíveis,

gerando duas possíveis extensões *default* para caracterizar os dois comportamentos esperados.

ADT $BoundedStack[Char]$

Syntax

Import $Elem$

Function

$create : \longmapsto BoundedStack;$

$push : BoundedStack \times Elem \longmapsto BoundedStack;$

$pop : BoundedStack \longmapsto BoundedStack;$

$removebottom : BoundedStack \longmapsto BoundedStack;$

$top : BoundedStack \longmapsto Elem$

Predicate $empty(BoundedStack);$

$reachbound(BoundedStack);$

Axiomatization

$(\forall b : BoundedStack)$

$b \stackrel{S}{=} create \vee (\exists b' : BoundedStack)(\exists e : Elem) b \stackrel{S}{=} push(b', e)$ (1)

$empty(create)$ (2)

$(\forall b : BoundedStack)(\forall e : Elem) \neg empty(push(b, e))$ (3)

$(\forall b : BoundedStack)(\forall e : Elem)$

$\neg reachbound(b) \Rightarrow pop(push(b, e)) \stackrel{S}{=} b$ (4)

$(\forall b : BoundedStack)(\forall e : Elem)$

$\neg reachbound(b) \Rightarrow top(push(b, e)) \stackrel{C}{=} e$ (5)

$(\forall b : BoundedStack) \neg reachbound(removebottom(b))$ (6)

Hypothesis

$$\frac{reachbound(b) : push(b, e) \stackrel{S}{=} b}{push(b, e) \stackrel{S}{=} b} \quad (7)$$

$$\frac{reachbound(b) : push(b, e) \stackrel{S}{=} push(removebottom(b), e)}{push(b, e) \stackrel{S}{=} push(removebottom(b), e)} \quad (8)$$

End

Embora, em ambos as extensões *default* estejam sendo definidas pilhas limitadas, como conseqüência da compacidade (ver [End72]) as duas extensões *default* adimitem realizações, em cujo o domínio há instâncias de cardinalidade infinita, ainda que $Char$

seja sempre um domínio finito de vinte e seis constantes. Logo, para determinar as extensões *default* é preciso observar o fechamento dos *default* por todas as instâncias válidas do domínio de pilha, logo a declaração **Hypothesis** esconde na verdade, um conjunto infinito de *defaults* para gerar cada extensão *default* (ver [Rei80]).

$BoundedStack[Char]$ foi apresentado aqui para exemplificar uma série de aspectos interessantes na utilização de teorias *default* para especificar tipos abstratos de dados. Além da utilização de *defaults* abertos, o que notadamente, deve pressupor um cuidado especial, por mais uma vez o TAD $Array[Indice, Char]$ será preferido para suportar a implementação. Isto é viável porque uma teoria de primeira ordem é sempre uma teoria *default* onde paradoxalmente não há *defaults*. Neste caso, apenas uma extensão *default* seria gerada, o próprio conjunto de conseqüências lógicas da especificação de $Array[Indice, Char]$.

Isto pretende mostrar que teorias clássicas e teorias *default* podem ser combinadas no desenvolvimento de uma implementação, contudo, estas últimas tem a preferência. Na presença de teorias *default*, as teorias clássicas são vistas como teorias *default* sem *defaults*.

O fato da especificação para pilha envolver, na verdade, um número infinito de *defaults* fechados causaria algumas complicações, caso a sua implementação fosse tentada construtivamente, pois a aplicação do Teorema da Construtibilidade requer que as teorias apresentem conjunto de *defaults* fechados e finitos. Ainda assim, é estabelecida sem maiores problemas a implementação declarativa esperada, para mediar este passo canônico, sem necessitar de *defaults* abertos:

ADT $BStack^*[Char]$

Syntax

Import $Elem$

Function

$create^*$	:	$\longmapsto BStack^*$;
$push^*$	:	$BStack^* \times Elem \longmapsto BStack^*$;
pop^*	:	$BStack^* \longmapsto BStack^*$;
$removebottom^*$	:	$BStack^* \longmapsto BStack^*$;
top^*	:	$BStack^* \longmapsto Elem$

Predicate $empty^*(BStack^*)$;
$reachbound^*(BStack^*)$;

HidenSymbols $CList$;

$head : CList \longmapsto Indice$;

$rear : CList \longmapsto Indice$;

$array : CList \longmapsto Array$;

$bstack : CList \longmapsto BStack^*$;

Axiomatization

$$(\forall c : CList)(\exists a : Array)(\exists i, i' : Indice)$$

$$head(c) \stackrel{I}{=} i \ \wedge \ rear(c) \stackrel{I}{=} i' \ \wedge \ array(c) \stackrel{A}{=} a \tag{1}$$

$$(\forall c, c' : CList)\neg c \stackrel{CL}{=} c' \Rightarrow$$

$$\{head(c) \stackrel{I}{=} head(c') \ \wedge \ rear(c) \stackrel{I}{=} rear(c') \ \wedge \ array(c) \stackrel{A}{=} array(c')\} \tag{2}$$

$$(\forall c, c' : CList)bstack(p) \stackrel{S}{=} bstack(p') \Leftrightarrow (\forall i : Indice)$$

$$inbetween(i, head(c), rear(c)) \Rightarrow$$

$$value(array(c), i) \stackrel{C}{=} value(array(c'), offset(i, head(c), head(c'))) \tag{3}$$

$$(\forall b : BStack^*)(\exists c : CList)bstack(c) \stackrel{S}{=} b \tag{4}$$

$$(\forall b : BStack^*)b \stackrel{S}{=} create^* \Leftrightarrow (\exists c : CList)$$

$$\{bstack(c) \stackrel{S}{=} b \ \wedge \ rear(c) \stackrel{I}{=} head(c) \ \wedge \ array(c) \stackrel{A}{=} inicialize\} \tag{5}$$

$$(\forall b : BStack^*)empty^*(b) \Leftrightarrow$$

$$(\exists c : CList)bstack(c) \stackrel{S}{=} b \ \wedge \ rear(c) \stackrel{I}{=} head(c) \tag{6}$$

$$(\forall b : BStack^*)reachbound^*(b) \Leftrightarrow$$

$$(\exists c : CList)bstack(c) \stackrel{S}{=} b \ \wedge \ rear(c) \stackrel{I}{=} inc(head(c)) \tag{7}$$

$$(\forall b : BStack^*)(\exists b' : BStack^*)b' \stackrel{S}{=} removebottom^*(b) \Leftrightarrow$$

$$(\exists c, c' : CList)bstack(c) \stackrel{S}{=} b \ \wedge \ bstack(c') \stackrel{S}{=} b' \Rightarrow$$

$$head(c') \stackrel{I}{=} head(c) \wedge rear(c') \stackrel{I}{=} inc(rear(c)) \wedge array(c') \stackrel{A}{=} array(c) \tag{8}$$

$$(\forall s : BStack^*)(\forall r : Char)p \stackrel{C}{=} top^*(s) \Leftrightarrow$$

$$(\exists c : CList)bstack(c) \stackrel{S}{=} b \Rightarrow value(array(c), head(c)) \stackrel{C}{=} r \tag{9}$$

$$(\forall b : BStack^*)(\exists b' : BStack^*)b' \stackrel{S}{=} pop^*(b) \Leftrightarrow$$

$$(\exists c, c' : CList)\{bstack(c) \stackrel{S}{=} b \ \wedge \ bstack(c') \stackrel{S}{=} b'\} \Rightarrow$$

$$inc(head(c')) \stackrel{I}{=} head(c) \wedge rear(c') \stackrel{I}{=} rear(c) \wedge array(c') \stackrel{A}{=} array(c) \tag{10}$$

Hypothesis

$$\frac{\begin{array}{l}(\forall b : BStack^*)(\forall r : Char)(\exists b' : BStack^*)\neg reachbound(b) \Rightarrow \{b' \stackrel{S}{=} push^*(b,r) \Leftrightarrow \\ : \ (\exists c, c' : CList)[bstack(c) \stackrel{S}{=} b \ \wedge \ bstack(c') \stackrel{S}{=} b'] \Rightarrow [head(c') \stackrel{I}{=} inc(head(c)) \\ \wedge \ rear(c') \stackrel{I}{=} rear(c) \ \wedge \ array(c') \stackrel{A}{=} attrib(array(c), inc(head(c)), c)]\}\end{array}}{\begin{array}{l}(\forall b : BStack^*)(\forall r : Char)(\exists b' : BStack^*)\neg reachbound(b) \Rightarrow \{b' \stackrel{S}{=} push^*(b,r) \Leftrightarrow \\ (\exists c, c' : CList)[bstack(c) \stackrel{S}{=} b \ \wedge \ bstack(c') \stackrel{S}{=} b'] \Rightarrow [head(c') \stackrel{I}{=} inc(head(c)) \\ \wedge \ rear(c') \stackrel{I}{=} rear(c) \ \wedge \ array(c') \stackrel{A}{=} attrib(array(c), inc(head(c)), c)]\}\end{array}} \quad (11)$$

$$\frac{\begin{array}{l}(\forall b : BStack^*)(\forall r : Char)(\exists b', b'' : BStack^*) reachbound(b) \Rightarrow \\ : \ b' \stackrel{S}{=} push^*(b,r) \Leftrightarrow b'' = removebottom(b) \ \wedge \ b' = push^*(b'', r)\end{array}}{\begin{array}{l}(\forall b : BStack^*)(\forall r : Char)(\exists b', b'' : BStack^*) reachbound(b) \Rightarrow \\ b' \stackrel{S}{=} push^*(b,r) \Leftrightarrow b'' = removebottom(b) \ \wedge \ b' = push^*(b'', r)\end{array}} \quad (12)$$

$$\frac{: (\forall b : BStack^*)(\forall r : Char) b \stackrel{S}{=} push^*(b,r) \Leftrightarrow reachbound(b)}{(\forall b : BStack^*)(\forall r : Char) b \stackrel{S}{=} push^*(b,r) \Leftrightarrow reachbound(b)} \quad (13)$$

End

A partir da teoria *default* especificada por $BoundedStack^*[Char]$ são geradas duas extensões *defaults* distintas. Em uma, onde são disparados os *defaults* (11) e (12), a pilha é implementada como se fosse uma lista circular, para possibilitar a operação de remover o caractere mais antigo na pilha. Seja, por instância $\Re$ um modelo para $BoundedStack^*[Char]$, que realiza o sorte *Indice* como o conjunto dos números naturais, e o símbolo *inc*, herdado também da linguagem de *Indice*, poderia realizado como :

$$\Re|inc(i)| = (i+1)\text{modulo}(maxpos+1)$$

onde + denota a soma algébrica entre números, e modulo é a operação que retorna o resto da divisão de um número por outro. A função *inc* nesta realização claramente se comportaria incrementando o seu argumento, fornecendo um resultado sempre restrita entre 0 (zero) e o valor realizado para a constante *maxpos*.

A segunda extensão *default* de $BoundedStack^*[Char]$, gerada por (11) e (13), não precisa ser uma lista circular, funciona simplemente como uma lista limitada, onde nenhum valor pode ser acrescentado depois que o limite *maxpos* for atingido. Em um modelo para esta extensão, tomando novamente *Indice* sobre o domínio dos números naturais, a função *inc* poderia ser realizada, desta vez, como somar uma unidade ao valor de seu argumento, contanto que não seja igual a constante *maxpos*, quando é indiferente sua definição.

Todavia aproveitando que $Char$ representa um domínio sempre finito, nada impede que um passo canônico seja superpostos por sobre o passo $BoundedStack[Char] \rightsquigarrow Array[Indice, Char]$, restringindo as realizações tratadas na implementação subjacente para instâncias finitas, com uma "altura" máxima fixa, arbitrada em n. Ou seja, isto implica em especializar as realizações para o TAD $BoundedStack[Char]$.

Lançando mão de um artifício, a "altura" da pilha pode ser controlada, associando-a a um sorte (e.g. $Height$) cujas as realizações tem sempre um domínio finito, de cardinalidade n, e n não é uma constante sujeita a interpretação (seu valor é definido, apenas foi omitido, mas poderia ser por exemplo 100). Seja a função

$$height : BoundedStack \longmapsto Height$$

que mapeia innstâncias de pilha em seu tamanho, de acordo com os seguintes axiomas:

$$(\forall b, b' : BoundedStack) empty(b) \Rightarrow \{height(b) \stackrel{H}{=} height(b') \Rightarrow empty(b')\}$$

$$(\forall b, b' : BoundedStack) height(b) \stackrel{H}{=} height(b') \Leftrightarrow$$
$$(\exists s, s' : BoundedStack)(\exists r, r : Char)$$
$$\{b \stackrel{B}{=} push(s, r) \ \wedge \ b' \stackrel{B}{=} push(s', r') \ \wedge \ height(s) \stackrel{H}{=} height(s')\}$$

Finalmente, o axioma,

$$(\exists h_1, h_2, \ldots, h_n : Height)(\forall h : Height)$$
$$\neg h_1 \stackrel{H}{=} h_2 \ \wedge \ \neg h_1 \stackrel{H}{=} h_3 \ \wedge \ \ldots \ \wedge \ \neg h_1 \stackrel{H}{=} h_n \ \wedge$$
$$\neg h_2 \stackrel{H}{=} h_3 \ \wedge \ \ldots \ \wedge \ \neg h_2 \stackrel{H}{=} h_n \ \wedge \ \ldots \ \wedge$$
$$\neg h_{n-1} \stackrel{H}{=} h_n \ \wedge$$
$$h \stackrel{H}{=} h_1 \ \vee \ h \stackrel{H}{=} h_2 \ \vee \ \ldots \ \vee \ h \stackrel{H}{=} h_n$$

define a cardinalidade n do sorte $Height$. Como todas as pilhas estão associadas a uma "altura", e está é finita e limitada, o sorte $BoundedStack$ só admitiria modelos com domínios de pilhas finitas, (de "altura" nunca superior a n). Já que $Char$ é também finito, há por conseqüência, uma número finito de instâncias de pilha.

Introduzido, não conservativamente, em $Indice$ a sentença:

$$\underbrace{inc(inc(\ldots inc}_{n}(maxpos)\ldots)) \stackrel{I}{=} maxpos$$

fica igualmente estabelecido o "tamanho" n da fila circular que representa o objeto abstrato $BoundedStack$. Falta ainda, encontrar uma repesentação $Height^*$, adequada para

o novo sorte $Height$,

$$(\forall h : Height^*)(\exists i : Indice)$$
$$h \stackrel{I}{=} i \Leftrightarrow \{\neg i < inc(maxpos) \ \wedge \ \neg maxpos < i\}$$

A nova implementação declarativa definida estendendo $BoundedStack^*[Char]$, ainda gera duas extensões *default*, contendo as geradas por $BoundedStack^*[Char]$, mas em ambas é sempre possível derivar propriedades relacionadas com a limitação da "altura" a n.

No contexto da *Lógica Default,* com mais ênfase, a definição de uma implementação declarativa, Δ, é crítica, como visto através do exemplo anterior. É ainda mais conveniente recorrer à estratégia construitiva, para determinar Δ, como a superposição de passos canônicos. Desta maneira, a construção final para Δ resultaria de uma sucessão de extensões $\delta_1, \delta_2, \ldots, \delta_n$, onde em cada uma ($\delta_i \subseteq \delta_{i+1}$) vai sendo delineado seu conjunto de extensões *default*, acrescentando gradativamente novas propriedades e eliminando outras.

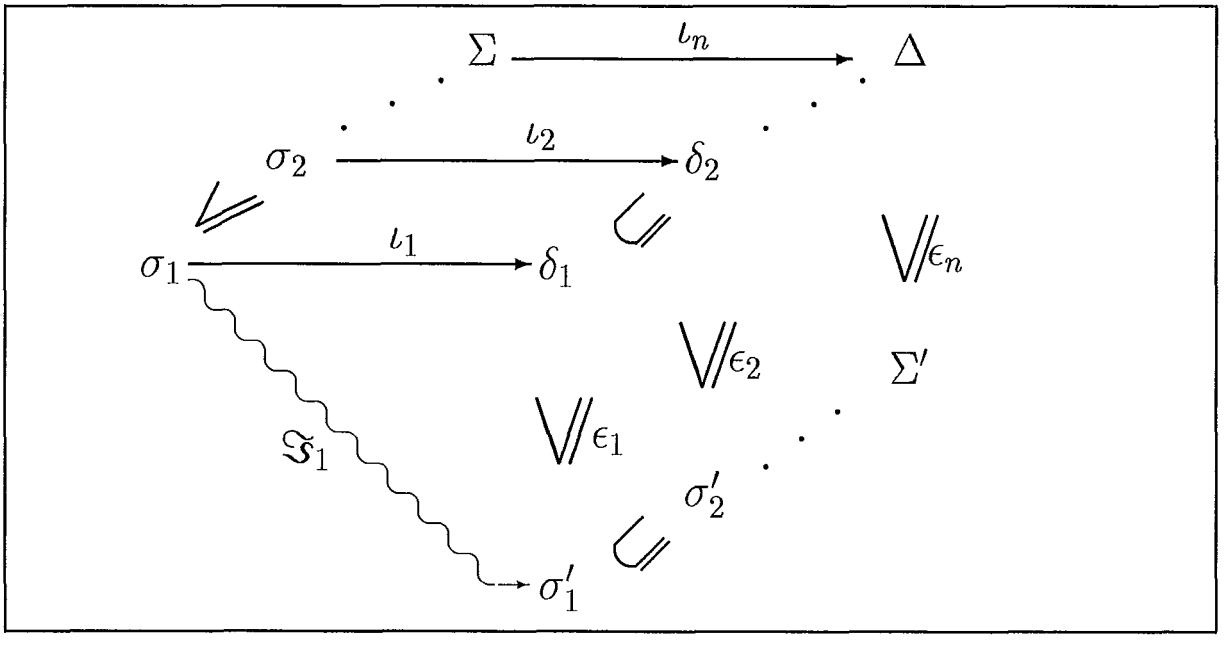


Figura VI.1.1: Desenvolvimento Construtivo da Especificação Δ

Limitando o foco de atenção, provavelmente será menos complexo acompanhar os efeitos causados pela adição de *defaults* ou axiomas. Ou ao menos, detectar erros antecipadamente, minimizando sua abrangência.

De acordo com o Teorema da Construtibilidade (V.2.2), a implementação declarativa Δ, presumivelmente definida conforme uma estratégia construtiva (a exemplo

da seção IV.1), incorpora em suas extensões *default* todas as propriedades que foram presumidas durante a construção de $\delta_1,\delta_2,\ldots,\delta_n$, mas não foram refutadas e continuam coerentes. Assim, a construção de Δ reflete a cada momento, o conhecimento racional e presente, e deixa sempre possibilidade de revisá-lo em um próximo passo superposto.

Como exemplo, seja ρ uma relação binária sobre um domínio não vazio, cuja especificação deveria ser construida para suportar a implementação de uma teoria *default* Σ. Inicialmente, seja $\sigma_1' = \langle W_1, D_1 \rangle$, uma teoria *default* onde :

$$W_1 = \{reflexiva(\rho), transitiva(\rho)\}$$
$$D_1 = \left\{ \frac{: simetrica(\rho)}{simetrica(\rho)}, \frac{: \neg simetrica(\rho)}{\neg simetrica(\rho)} \right\}$$

σ_1 descreve dois possíveis comportamentos de ρ, pois são geradas duas extensões *default*:

$$\mathcal{A}_1 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), simetrica(\rho)\})$$
$$\mathcal{B}_1 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), \neg simetrica(\rho)\})$$

A primeira, $\mathcal{A}_1$, especifica ρ como uma relação de equivalência, valendo a reflexibilidade, a transitividade e a simetria, enquanto $\mathcal{B}_1$ define uma ordem parcial para ρ.

Superpondo o próximo passo, uma nova propriedade de ρ poderia ser definida, incluindo D_1 os *default*:

$$\left\{ \frac{\neg simetrica(\rho) : total(\rho)}{total(\rho)}, \frac{\neg simetrica(\rho) : \neg total(\rho)}{\neg total(\rho)} \right\}$$

Neste caso, a nova teoria *default* σ_2' formada apresentaria três extensões *default* :

$$\mathcal{A}_2 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), simetrica(\rho)\})$$
$$\mathcal{B}_2 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), \neg simetrica(\rho), \neg total(\rho)\})$$
$$\mathcal{C}_2 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), \neg simetrica(\rho), total(\rho)\})$$

A extensão *default* $\mathcal{C}_2$ denota agora uma relação de ordem total. A teoria *default* σ_2' é por definição uma extensão de σ_1', segundo a definição V.1.2.

Presumindo que no passo seguinte haja motivos para não aceitar ρ como uma relação de equivalência, a extensão *default* $\mathcal{A}_2$ logo precisaria ser eliminada. Para tanto a sentença $\neg simetrica(\rho)$, adicionada entre os axiomas de σ_2', automaticamente a descarta, restando apenas as outras duas como extensões *default* da nova teoria *default* σ_3' construida.

As relações totais podem ser diferenciadas por serem discretas ou contínuas, e portanto outros *defaults* serão incluidos em σ'_3, definido a teoria *default* $\sigma'_4 = \langle W_4, D_4 \rangle$, onde :

$$W_4 = \{reflexiva(\rho), transitiva(\rho), \neg simetrica(\rho), discreta(\rho) \Rightarrow \neg continua(\rho)\}$$

$$D_4 = \left\{ \frac{: simetrica(\rho)}{simetrica(\rho)}, \frac{: \neg simetrica(\rho)}{\neg simetrica(\rho)}, \frac{\neg simetrica(\rho) : total(\rho)}{total(\rho)}, \right.$$

$$\left. \frac{\neg simetrica(\rho) : \neg total(\rho)}{\neg total(\rho)}, \frac{\neg total(\rho) : discreta(\rho)}{discreta(\rho)}, \frac{\neg total(\rho) : continua(\rho)}{continua(\rho)} \right\}$$

O conjunto definitivo de extensões *default* para σ'_4, é gerado resultando em:

$$\mathcal{B}_4 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), \neg simetrica(\rho), \neg total(\rho)\})$$

$$\mathcal{C}_4 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), \neg simetrica(\rho), total(\rho), discreta(\rho)\})$$

$$\mathcal{D}_4 = \mathbf{Cn}(\{reflexiva(\rho), transitiva(\rho), \neg simetrica(\rho), total(\rho), continua(\rho)\})$$

Durante a construção de σ'_4 foi preservado sempre uma relação de continência de forma que:

$$\sigma'_1 \subseteq \sigma'_2 \subseteq \sigma'_3 \subseteq \sigma'_4$$

Este processo de construção da teoria *default* primitiva influencia diretamente a constução da implementação declarativa Δ correspondente.

Embora, o desenvolvimento construtivo tenha sido visto apenas do ângulo da especificação mais primitiva, este exemplo visa ilustrar como a composição de um passo canônico com um outro subjacente, permite modificar o conjunto das extensões *default* de uma teoria *default,* sendo especificada. Após um número suficiente de composições, um resultado definitivo pode ser finalmente obtido.

Mesmo que a implementação esteja apropriadamente "concluida", os resultados produzidos não perdem o *status* de crença, ou seja, hipótese não refutada. Positivamente, uma implementação está sempre sujeita a atualização, porque não existe programas permanentemente estáveis. Este é o caráter não monotônico da implementação construtiva.

Inegavelmente, há um ganho em flexibilidade, pois, levantar hipóteses não teria conseqüências tão drásticas quanto decidir antecipadamente a válidade de fatos.

Também a consecução de passos canônicos propicia o decréscimo no conjunto de extensões *default*, decidindo quais ainda merecem ser implementadas, além de enriquecê-las, pois o nível de detalhes aumenta conforme a implementação aproxima-se de linguagens mais primitivas.

Na Composição de $\Sigma_1 \rightsquigarrow \Sigma_2$ com $\Sigma_2 \rightsquigarrow \Sigma_3$, o Teorema da Modularização (V.2.1) produz uma implementação declarativa implementando Σ_1 em Σ_3, contanto que hajam extensões *default* de Σ_1 propriamente implementadas em extensões *default* de Σ_2, que por sua vez, estejam também implementadas em extensões *default* para Σ_3, isto é, caso seja possível compor a implementação, em ao menos um par de extensões *default* de Σ_1 e Σ_3. A cada passo canônico, há chance para escolher que representações devem continuar a implementação e quais deveriam ser abandonadas.

A implementação novamente requer atenção, porque não é difícil perder completamente o rumo, continuando a implementar em $\Im_2$ representações sem qualquer conexão com o conceito original, em Σ_1.

A capacidade de lidar adequadamente com conjecturas confere mais um recurso, para apreender propriedades a respeito de um conceito. Não se exige certeza *a prioiri,* que uma propriedade informada está incontestavelmente correta, antes de adicioná-la a especificação. Uma propriedade poderia ser aceita, como mais uma crença racional, desde que seja concebível uma representação consistente adicionando-a àquilo que presentemente é acreditado, nesta representação.

Todavia o novo conjunto de extensões *default* formado deveria, em princípio, preservar alguma relação com o conjunto de extensões *default* anterior, isto é, a adição deveria dar-se através de um mecanismo de composição. Posteriormente, a continuidade da implementação ou coaduna com a hipótese levantada, ou encarrega-se de refutá-la, descartando as supostas representações que esta propriedade deu origem.

Daí porque estes mecanismos adaptam-se naturalmente a situações onde o conceito sendo implementado está parcialmente compreendido, ou há alguma indefinição quanto aos objetivos globais do desenvolvimento. Por exemplo, se a solução procurada para um dado problema não está plenamente consolidada, é possível especificar algumas opções e refiná-las enquanto a implementação progride, até atingir uma formulação final que satisfaz o problema. Adiando a necessidade de confirmação, é factível conjecturar e testar algumas idéias para implementar uma resolução para o problema.

Todavia, esta seção não defende um total descontrole, onde propriedades são arbitrariamente colocadas e retiradas, até que por tentativa-e-erro, após um incontável número de passos, seja atingida uma representação final com aparência acidentalmente satisfatória. Agindo como se não importasse o modo que foi conduzida a implementação, desde que o resultado condiga com o esperado.

Este extremo é igualmente inaceitável, a ausência completa de disciplina não ajuda, ao contrário, compromete a implementação. Apenas concordando com a sistematização imposta pelos mecanismos de composição, che ga-se a resultados com confiança e eficiência. A liberalidade não monotônica é favorável se corretamente disciplinada. Embora, isto soe paradoxal, representa o equilíbrio sensato para o bom termo da implementação.

Estratégias estilo *'Unfold-fold'*, ou a especificação e implementação de protótipos são favorecidas com o relaxamento da monotonicidade. Qualquer situação, onde a expressão da critividade demanda uma maior liberalidade, pode valer-se destes mecanismos não monotônicos.

Por outro lado, certa prudência deve anteceder qualquer entusiasmo. O custo para liberar as restrições inerentes à monotonicidade envolve perder algum conforto,(e.g. a composição de implementações pode não resultar formalmente em um passo de implementação). Mesmo uma simples modificação na especificação, como a adição de um *default*, pode provocar alterações radicais em várias extensões *default*. Consequentemente, é bem complicado antever os efeitos desencadeados e a influência de certas decisões no processo de implementação.

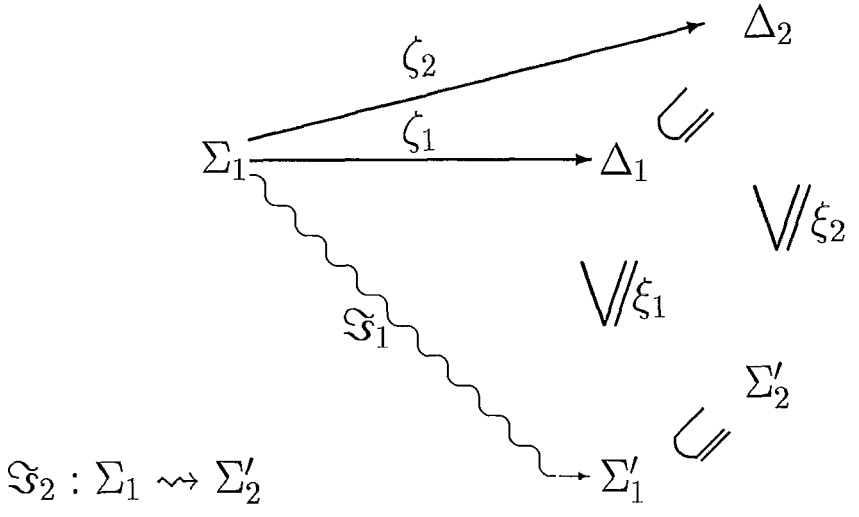
VI.2 Alterações no Contexto da *Lógica Default*

A monotonicidade, característica da Lógica de Primeira Ordem, é que impedia um tratamento mais satisfatório para a alteração. Agora, pensar realisticamente em dar seqüência a implementação, ainda que haja surgido a necessidade de alterar, eliminando da especificação propriedades não mais adequadas, não é totalmente absurdo. Os benefícios que esta facilidade retorna são claros, o trabalho realizado até então não seria inutilizado, nem mesmo em parte, não se faria preciso refazer ou reconstruir passos canônicos.

No contexto da *Lógica Default*, a retirada de propriedades realiza-se novamente, por descarte de extensões *default*, que as apresentam como crenças coerentes. Invariavelmente, consegue-se estabelecer extensões entre teorias *default*, pelo mero bloqueio de extensões *default* da primeira.

Assim, na extensão de Σ_1' para Σ_2' podem ser descartadas algumas extensões *default* e logo, a implementação declarativa Δ_2 não conteria mais no conjunto de extensões *default*, as propridades completamente eliminadas e suas conseqüências. A al-

teração transcorre assim, a contento, conforme o diagrama a seguir.



Portanto, a versão *default* do Teorema da Construtibilidade é capaz de proceder as alterações, superpondo um novo passo canônico $\Im_2$, capaz de descartar extensões *default* de Δ_1 indesejáveis, desde que não sejam desconsideradas todas aquelas de interesse. Se nenhuma extensão *default* realmente haja sido deixada em Δ_2, não haveria por definição, um passo canônico de Σ_1 em Σ'_2.

Assim, é preciso parar e identificar o que não saiu a contento, e por que. Se for o caso, talvez seja aconselhável esquecer $\Im_2$ e tentar novamente um outro passo de implementação por sobre $\Im_1$.

Apenas a disciplina de acesso diferencia uma fila de uma pilha. Caso persista a dúvida quanto a disciplina mais apropriada, a *Lógica Default* permite representar ambas as disciplinas, em uma mesma estrutura dando origem a extensões *default* distintas,

ADT *UnKnown*[*Elem*]

Syntax

Import *Elem*

Function *create* : $\longmapsto$ *UnKnown*;
insere : *UnKnown* $\times$ *Elem* $\longmapsto$ *UnKnown*;
remove : *UnKnown* $\longmapsto$ *UnKnown*;
value : *UnKnown* $\longmapsto$ *Elem*

Predicate $empty(UnKnown)$;
$LIFO$;
$FIFO$;

Axiomatization

$$(\forall u : Unknown)$$
$$u \stackrel{S}{=} create \ \vee \ (\exists u' : Unknown)(\exists e : Elem) u \stackrel{S}{=} insere(u', e) \quad (1)$$
$$empty(create) \quad (2)$$
$$(\forall u : Unknown)(\forall e : Elem) \neg empty(insere(u, e)) \quad (3)$$
$$(\forall u : Unknown) empty(u) \Rightarrow empty(remove(u)) \quad (4)$$
$$(\forall u : Unknown)(\forall e : Elem) empty(u) \Rightarrow value(insere(u, e)) \stackrel{E}{=} e \quad (5)$$

Hypothesis

$$\frac{: LIFO \ \wedge \ \neg FIFO}{LIFO} \quad (6)$$

$$\frac{: FIFO \ \wedge \ \neg LIFO}{FIFO} \quad (7)$$

$$\frac{FIFO : \begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow value(insere(u, e)) \stackrel{E}{=} value(u)\end{array}}{\begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow value(insere(u, e)) \stackrel{E}{=} value(u)\end{array}} \quad (8)$$

$$\frac{FIFO : \begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow remove(insere(u, e)) \stackrel{S}{=} insere(remove(u), e)\end{array}}{\begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow remove(insere(u, e)) \stackrel{S}{=} insere(remove(u), e)\end{array}} \quad (9)$$

$$\frac{LIFO : \begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow value(insere(u, e)) \stackrel{E}{=} e\end{array}}{\begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow value(insere(u, e)) \stackrel{E}{=} e\end{array}} \quad (10)$$

$$\frac{LIFO : \begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow remove(insere(u, e)) \stackrel{S}{=} u\end{array}}{\begin{array}{l}(\forall u : Unknown)(\forall e : Elem) \\ \neg empty(u) \Rightarrow remove(insere(u, e)) \stackrel{S}{=} u\end{array}} \quad (11)$$

End

Entre as extensões *default* possívelmente geradas, uma comporta-se de forma *LIFO,* enquanto a outra atua como se houvesse sido especificada inicialmente uma estrutura *FIFO.*

Exemplificando a extrema capacidade de alteração de uma implementação no contexto não monotônico, a disciplina de acesso mais cabível é escolhido facilmente, descartando a outra, pela simples introdução do predicado correspondente, como axioma. Assim $Unknown[Elem]$ pode ser alterada, conforme a situação exija um comportamento *LIFO,* ou *FIFO.*

Embora o emprego da *Lógica Default* não seja exatamente milagroso, revela facilidade antes não imaginadas, capazes de contornar problemas inerentemente relacionados às restrições de monotonicidade, como este.

VI.3 Exceções e Condições de Erro

A implementação de exceções e condições de erro, bastante usuais na especificação de operações, não havia merecido sequer alusão no contexto clássico, mas agora oportunamente, junta mais uma aplicação para os formalismo não monotônicos. O esquecimento anterior explica-se mesmo pela falta de recursos mais apropriados para tratar esta situação.

Ao especificar um certo elemento de um TAD, uma função por exemplo, as vezes fica evidenciado que seu comportamento difere em circunstâncias particulares. Como representar então uma variedade de comportamentos que caracteriza a função ? Em geral, apenas uma circunstância é relevante para a implementação, e neste caso uma especificação liberal envolvendo somente as propriedades de interesse, custuma ser suficiente.

Mas quando convém uma especificação mais precisa, ou está confusa a noção precisa de relevância, é a *Lógica Default* quem permite representar esta multiplicidade de comportamentos, por meio de extensões *default.* Inclusive caso sejam identificadas várias possíveis situações distintas, e não esteja perfeitamente claro que todas verificam-se realmente, a mesma idéia vale, na dúvida, todas são representadas, e a confirmação definitiva fica adiada para a implementação. Em caso negativo, é só descartar, através de mecanismos de composição, aquelas representações para as circunstâncias irrealizáveis.

Por convenção, a situação normal representa o conjunto de propriedades que se presume verificar com maior freqüência, todos os demais conjuntos de crenças ainda factíveis são as execeções. Já uma condição de erro seria aproximadamente uma exceção

cuja ocorrência julga-se indesejável ou inpertinente.

Seja uma função n-ária $f(v_1,v_2,\ldots,v_n)$ que na situação onde vale ϕ, e não há nenhuma comprovação a respeito de μ, comporta-se de modo a satisfazer a

$$(\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\, \psi[v/f(v_1, v_2, \ldots, v_n)]$$

Considerando ϕ e ψ fórmulas da linguagem, e ψ com apenas a variável livre v de mesmo sorte que f. O *default*,

Default VI.1

$$\frac{\phi : (\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\, \psi[v/f(v_1, v_2, \ldots, v_n)] \wedge \neg\mu}{(\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\, \psi[v/f(v_1, v_2, \ldots, v_n)]}$$

representa exatamente esta situação.

Se for conhecido o comportamento de f, quando acontece a exceção μ, por instância satisfazendo a propriedade $(\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\, \gamma[v/f(v_1,v_2,\ldots,v_n)]$, onde γ é outra fórmula da linguagem, com v livre, especifica-se a exceção através do *default*,

Default VI.2

$$\frac{\mu : (\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\, \gamma[v/f(v_1, v_2, \ldots, v_n)]}{(\forall v_1 : S_1, v_2 : S_2, \ldots, v_n : S_n)\, \gamma[v/f(v_1, v_2, \ldots, v_n)]}$$

Evidentemente, nunca será gerada uma extensão *default* em que vale os consequentes de VI.1 e VI.2, simultaneamente, contudo, a mesma teoria *default* resume a especificação de ambas as situações. Acrescentado o axioma $\phi \Rightarrow \neg\mu$, fica enfatizado que a situação normal e a execessão são mutuamente exclusivas. Talvez μ configure uma condição de erro.

Por ilustração, se não há absoluta certeza que as instâncias de uma seqüência devem permitir ou não a repetição de elementos, a especificação para o TAD $RSeq[Elem]$ poderia descrevê-lo, considerando normal a ocorrência de repetições, e ocasionalmente uma exceção quando a repetição de elementos deve ser evitada.

ADT $RSeq[Elem]$

Syntax

Import $Elem$

Function $create : \longmapsto RSeq;$
$insere : RSeq \times Elem \longmapsto RSeq;$
$tail : RSeq \longmapsto RSeq;$
$head : RSeq \longmapsto Elem$

Predicate $on(RSeq, Elem);$
$isnull(RSeq);$
REP

Axiomatization

$$(\forall s : RSeq)$$
$$s \stackrel{S}{=} null \ \vee \ (\exists s' : RSeq)(\exists e : Elem) s \stackrel{S}{=} insere(s', e) \tag{1}$$
$$isnull(create) \tag{2}$$
$$(\forall s : RSeq)(\forall e : Elem) isnull(s) \Rightarrow \neg on(s, e) \tag{3}$$
$$(\forall s : RSeq)(\forall e : Elem)$$
$$on(s, e) \Leftrightarrow \neg isnull(s) \ \wedge \ (\exists s' : RSeq)\{s \stackrel{S}{=} insere(s', e) \vee on(s', e)\} \tag{4}$$

Hypothesis

$$\frac{: (\forall s : RSeq)(\forall e : Elem) \neg on(s, e) \Rightarrow head(insere(s, e)) \stackrel{E}{=} e \ \wedge \ \neg REP}{(\forall s : RSeq)(\forall e : Elem) \neg on(s, e) \Rightarrow head(insere(s, e)) \stackrel{E}{=} e} \tag{5}$$

$$\frac{: (\forall s : RSeq)(\forall e : Elem) on(s, e) \Rightarrow head(insere(s, e)) \stackrel{E}{=} head(s) \wedge \neg REP}{(\forall s : RSeq)(\forall e : Elem) on(s, e) \Rightarrow head(insere(s, e)) \stackrel{E}{=} head(s)} \tag{6}$$

$$\frac{: (\forall s : RSeq)(\forall e : Elem) \neg on(s, e) \Rightarrow tail(insere(s, e)) \stackrel{S}{=} s \ \wedge \ \neg REP}{(\forall s : RSeq)(\forall e : Elem) \neg on(s, e) \Rightarrow tail(insere(s, e)) \stackrel{S}{=} s} \tag{7}$$

$$\frac{: (\forall s : RSeq)(\forall e : Elem) on(s, e) \Rightarrow tail(insere(s, e)) \stackrel{S}{=} tail(s) \ \wedge \ \neg REP}{(\forall s : RSeq)(\forall e : Elem) on(s, e) \Rightarrow tail(insere(s, e)) \stackrel{S}{=} tail(s)} \tag{8}$$

$$\frac{REP : (\forall s : RSeq)(\forall e : Elem) head(insere(s, e)) \stackrel{E}{=} e}{(\forall s : RSeq)(\forall e : Elem) head(insere(s, e)) \stackrel{E}{=} e} \tag{9}$$

$$\frac{REP : (\forall s : RSeq)(\forall e : Elem) tail(insere(s, e)) \stackrel{S}{=} s}{(\forall s : RSeq)(\forall e : Elem) tail(insere(s, e)) \stackrel{S}{=} s} \tag{10}$$

End

O predicado REP serve para indicar se a repetição de elementos é desejável ou não. Conjecurando que $RSeq[Elem]$ esteja sendo construida para implementar um conjuntos de elementos, então é necessária a opção pela caso excepcional. A adição não conservativa de $\neg REP$ em $RSeq[Elem]$ imediatamente causa o bloqueio, no passo su-

perposto, da extensão *default* de $RSeq[Elem]$ que não importava-se com a ocorrência de execeções.

Se na verdade, sucede a situação normal, sempre, ϕ (no caso REP) pode ser acrescentado como axioma a teoria *default*, em um passo canônico superposto, eliminando automaticamente todas as extensões *default* onde se verificava o consequente de VI.2. Caso contrário ambas, a situação normal e a exceção, continuaram a ser implementadas.

Este mesmo raciocínio pode ser utilizado para especificar e implementar quantas forem as situações distintas encontradas na representação de uma função. A *Lógica Default* presta-se para resolver, de modo elegante, mais uma limitação de expressividade, comum aos mecanismos clássicos.

VI.4 Reusabilidade

No contexto não monotônico, as idéias a respeito de reusabilidade são genericamente as mesmas apontadas antes na seção IV.4 A única novidade advem de uma maior flexibilidade nos recursos de composição que possibilita uma especificação ser reutilizada, ainda quando esta não se apresenta perfeitamente adequada à situação. Pois, reajustando-a não só por acréscimo de novas propriedades, como já acontecia no contexto clássico, mas também por eliminação de propriedades desnecessárias, construindo uma outra especificação definitivamente aplicável ao conceito.

A versão default para o Teorema da Construtibilidade desempenha papel análogo ao antes cumprido pela sua versão clássica, assegurando a capacidade de reutilizar especificações pré-existentes.

ADT $Vertex$

Syntax

Predicate $arc(Vertex, Vertex)$;
$edge(Vertex, Vertex)$;
$path(Vertex, Vertex)$;
$connected(Vertex, Vertex)$;
$sink(Vertex)$;
$source(Vertex)$;

Axiomatization

$(\forall u, v : Vertex)\{connected(u, v) \Leftrightarrow$

$$(edge(u,v) \vee (\exists x : Vertex) edge(u,x) \wedge connected(x,v))\} \quad (1)$$

$$(\forall u,v : Vertex)\{connected(u,v) \Leftrightarrow connected(v,u)\} \quad (2)$$

$$(\forall u,v : Vertex)\{path(u,v) \Leftrightarrow$$

$$(arc(u,v) \vee (\exists x : Vertex) arc(u,x) \wedge path(x,v))\} \quad (3)$$

$$(\forall u,v : Vertex)\{arc(u,v) \wedge arc(v,u) \Rightarrow edge(u,v)\} \quad (4)$$

$$(\forall u : Vertex)\{source(u) \Leftrightarrow (\forall v : Vertex) \neg u \stackrel{V}{=} v \Rightarrow path(u,v))\} \quad (5)$$

$$(\forall u : Vertex)\{sink(u) \Leftrightarrow (\forall v : Vertex) \neg u \stackrel{V}{=} v \Rightarrow path(v,u))\} \quad (6)$$

Hypothesis

$$\frac{: (\forall u,v : Vertex) arc(u,v) \Rightarrow arc(v,u)}{(\forall u,v : Vertex) arc(u,v) \Rightarrow arc(v,u)} \quad (7)$$

$$\frac{: (\exists u,v : Vertex) arc(u,v) \wedge \neg arc(v,u)}{(\exists u,v : Vertex) arc(u,v) \wedge \neg arc(v,u)} \quad (8)$$

End

A especificação para grafos, descrita acima, gera duas extensões default, uma onde estão representados os grafos direcionados, gerada pelo *default* (8), e outra onde este conceito de direcionamento é irrelevante, pois de acordo com o *default* (7), a relação *arc* torna-se transitiva. Nesta segunda extensão default é possível inferir propriedades que não são satisfatíveis na primeira, como por exemplos,

$$(\forall u : Vertex) sink(u) \quad \Leftrightarrow \quad source(u)$$

pois para um grafo não direcionado, os conceitos de *sink* e *source* não tem qualquer distinção no seu significado. Esta especificação pode ser construida e apropriadamente implementada, e sempre que for necessário, é bastante simples sua adaptação para uma ou outra circunstância.

VI.5 Referências Bibliográficas

Considerações sobre o tratamento de exceções e erros podem ser encontradas em [Vel87].

Capítulo VII

Epílogo

Este capítulo encerra fazendo uma restrospectiva do estudo desenvolvido aqui, destacando os resultados mais importantes deste trabalho, e por fim procura sugerir algumas direções para novas investigações futuras tomando esta como ponto de partida.

VII.1 Contribuições

Este trabalho pode ser sinteticamente resumido enumerando os principais resultados obtidos pelo estudo descrito até aqui:

- No capítulo III, foi concebida uma forma de composição de implementações cuja a aplicabilidade completa os resultados já possíveis através do Teorema da Modularização, e consiste em determinar um novo passo canônico sempre compondo-o com outro subjacente. Por analogia a seção III.1.1, em III.1.3 é estabelecida a corretude deste novo mecanismo de composição, enunciando o Teorema da Construtibilidade, e só então, está confirmada a sua aplicabilidade real;

- Dispondo de dois mecanismos de composição complementares, o método de desenvolvimento de programas com tipos abstratos de dados é favorecido com a proposição de uma estratégia de caráter construtivo, capaz de definir formalmente a implementação declarativa que especifica o módulo de implementação, como uma seqüência de passos sucessivos;

- O desenvolvimento construtivo de implementações ganha quando são mostrados meios capazes de contornar restrições teóricas relativas a conservatividade, conforme

foi disposto em IV.1.2 e IV.1.3;

- Na seção IV.2, foi argumentada a viabilidade de proceder também construtivamente a verificação de corretude de uma implementação;

- Novamente, apesar das limitações impostas pela conservatividade, há alternativas para tratar a necessidade de evoluir, manifestada em especificações implementadas *a priori*. Assim, o passo canônico é estendido, em resposta a novos elementos que se fazem presentes em uma ou outra das especificações participantes;

- Na seção IV.4, são formalizados critérios para a reusabilidade, empregando os mecanismos de composição para orientar a reutilização de especificações e/ou implementações;

- A seção IV.5 elucida melhor como ocorre a implementação quando estão envolvidos tipos abstratos de dados parametrizados;

- O conceito de sorte para a *Lógica Default*, está proposto em V.1, modificando a sintaxe da linguagem visando adaptá-la a especificação de tipos abstratos de dados;

- O capítulo V objetiva redefinir, no contexto da *Lógica Default*, noções de interpretação, extensão e extensão conservativa, de modo que as novas definições assimilem a mesma intuição representada em seus correpondentes clássicos, mas ainda sejam flexíveis o suficiente para permitir a transposição efetiva dos mecanismos de composição para a *Lógica Default*;

- Também, a seção V.1.2 sugere como obter um extensão conservativa, nos moldes da definição V.1.3, de uma teoria *default* normal, meramente introduzindo mais um *default* para definir o significado de um símbolo de função não pertencente antes a linguagem;

- O capítulo V conclue a transposição do Teorema da Modularização e o Teorema da Constuitibilidade, condição necessária e suficiente para estabelecer os mecanismos de composição apresentados, no contexto da *Lógica Default*. Assim, ao desenvolvimento de programas é permitido desconsiderar a última limitação importante relacionada com o formalismo clássico : a monotonicidade. A transição para a *Lógica Default* automaticamente libera os problemas encontrados na monotonicidade, embora isto implique em um cuidado multiplicado para lidar com um formalismo mais flexível;

- Comprovando que é factível implementar dentro de um contexto não monotônico, herda-se a habilidade para lidar com situações imprecisas ou parcialmente compreendidas, dispondo de novos padrões de raciocínio para levantar hipóteses que podem ser futuramente corroboradas ou refutadas.

VII.2 Sugestões para a Continuidade do Estudo

O presente trabalho não tem pretensões de ser conclusivo, mas propôs-se a despertar possibilidades a serem aprofundadas em futuras investigações. Para tanto, direções antevistas durante este tabalho são sugeridas como o seu prosseguimento natural :

- Aproveitando que o método de *Desenvolvimento de Programas com Tipos Abstratos de Dados* é favorecido com estratégias de natureza construtiva, é necessário incorporá-las definitivamente às etapas do método, detalhando os procedimentos envolvidos no desenvolvimento e verificação de uma implementação;

- Explorar meios para abordar a implementação através de *transformações que preservem a corretude.* Isto significa formalizar um conjunto de regras genéricas a serem aplicadas para definir a extensão conservativa e a interpretação para cada passo canônico. Ou seja, a proposta é conceber um cálculo dedutivo correto e completo para executar o passo canônico, propiciando um paradigma de desenvolvimento automaticamente sintético, aonde a verificação não aparece como uma etapa independente, mas já está embutida na própria construção.

- Transpor para o contexto da *Lógica Modal*, as noções lógicas de interpretação e extensão conservativa, e então, reestabelecer o Teorema da Modularização e o Teorema da Construtibilidade entre teorias modais. Deste modo, todas as idéias de composição de passos de implementação passam a ser observadas em aplicações que se encaixam naturalmente a *Lógica Modal*, como linguagem de especificação (e.g. construção de sistemas distribuídos, descrição de processos comunicantes e modelagem de transições de estado);

- Generalizar os mecanismos de composição em um nível ainda mais "alto", abstraindo-se de qualquer formalismo lógico particular, em favor de *Teoria da Categorias*;

- Investigar novas aplicações práticas para a composição de implementações, como por exemplo, em paradigmas *orientados a objetos*, desenvolvimento rápido de Protótipos

e outros.

Especificamente relacionados com estes trabalho, alguns aspectos aguardam ser futuramente completados :

- Procurar príncipios que orientem a derivação de uma especificação suficiente para o TAD mais abstrato, exclusivamente a partir das propriedades essenciais à computação do programa abstrato;
- Formalizar critérios melhor definidos para executar a etapa de simplificação (IV.1.1.1), que sejam adequados a futura recombinação dos resultados;
- Estabelecer critérios para indicar quando uma especificação já está suficientemente simples para uma primeira tentativa de implementação na etapa de trivialização (IV.1.1.2);
- Definir critérios para enfatizar o desacoplamento da implementação declarativa, já durante a sua construção;
- Ainda formalizar extensões por definição utilizando *defaults*, para símbolos de sorte, predicados ou funções, de preferência envolvendo *defaults* não normais;
- Associar aos mecanismos de composição de implementações apresentados no contexto da *Lógica Default*, um mecanismo eficiente de revisão de crenças, objetivando beneficiar o desenvolvimento construtivo e incorporação de alterações.

Apêndice A

Lógica de Primeira Ordem

A.1 Conceitos Básicos

No presente trabalho são adotadas as definições usuais de Lógica de Primeira Ordem, para os conceitos fundamentais de linguagem, teoria e modelo, conforme [End72, Sho67], e também consultando a [vD89] e [EFT84].

A.1.1 Linguagem

Definição A.1.1 (Linguagem) *uma linguagem $\mathcal{L}$ é o conjunto de fórmulas bem formadas (wffs) construídas sobre um alfabeto A (de símbolos de sorte, predicado e função), variáveis e a partir das constantes lógicas true e false, dos símbolos lógicos ($\neg$, $\wedge$, $\vee$,$\Rightarrow$ e $\Leftrightarrow$), dos quantificadores ($\forall$ e $\exists$), do símbolo de igualdade ($\stackrel{S}{=}$) para cada sorte S em A.*

Definição A.1.2 (Sublinguagem e Extensão de Linguagens) *Uma linguagem $\mathcal{L}$ é sublinguagem de $\mathcal{L}'$, denotado por $\mathcal{L} \subseteq \mathcal{L}'$, se e somente se o alfabeto de $\mathcal{L}$ está contido no alfabeto de $\mathcal{L}'$, neste caso, $\mathcal{L}'$ é dita uma extensão de $\mathcal{L}$.*

A.1.2 Modelo

Definição A.1.3 (Estruturas) *Formalmente, uma estrutura $\Re$, para uma dada linguagem de primeira ordem $\mathcal{L}$, é uma função cujo domínio é o conjunto de parâmetros, tais que:*

i) $\Re$ *atribui a cada sorte* S *em* $\mathcal{L}$ *um conjunto não vazio,* $|\Re|^S$*, chamado de domínio do sorte* S*;*

ii) $\Re$ *atribui a cada símbolo de predicado n-ário, e.g.* $p(S_1,S_2,\ldots,S_n)$ *em* $\mathcal{L}$*, uma relação n-ária,* $\Re(p) \subseteq |\Re|^{S_1} \times \ldots \times |\Re|^{S_n}$*; isto é,* $\Re(p)$ *é um conjunto de tuplas respectivamente sobre os domínios de* $S_1,S_2,\ldots,S_n$*;*

iii) $\Re$ *atribui a cada símbolo de função n-ária, e.g.* $f(S_1,S_2,\ldots,S_n)$ *de sorte* S*, em* $\mathcal{L}$*, uma operação n-ária,* $\Re(f) : |\Re|^{S_1} \times \ldots \times |\Re|^{S_n} \longmapsto |\Re|^S$.

Definição A.1.4 (Modelo) *Simplificadamente, uma estrutura* $\Re$ *para a linguagem* $\mathcal{L}$*, é um modelo (ou uma possível realização) para uma sentença* α *se* $\Re$ *satisfaz a* α*, denotando por* $\models_{\Re} \alpha$ *ou* $\Re \models \alpha$*.* $\Re$ *é um modelo para um conjunto de sentenças* Σ *se* $\Re$ *satisfaz a todas as sentenças de* Σ*, denotado por* $\models_{\Re} \Sigma$.

Definição A.1.5 (Classe de Modelos) *A Classe de Modelos que satisfaz a um conjunto de sentenças* Σ*, denotada por* $\boldsymbol{Mod}(\Sigma)$*, é:*

$$\boldsymbol{Mod}(\Sigma) = \{\Re \setminus \models_{\Re} \Sigma\}$$

Definição A.1.6 (Equivalência Elementar) *Duas estruturas na linguagem* $\mathcal{L}$*, são elementarmente equivalentes, denotado por* $\Re \equiv \aleph$*, se e somente se satisfazem as mesmas sentenças, isto é, para* $\alpha \in \mathcal{L}$*:*

$$\models_{\Re} \alpha \quad \textit{se e somente se} \quad \models_{\aleph} \alpha$$

Teorema A.1.1 *Se dois modelos são elementarmente equivalentes, então pertencem a classe de modelos da mesma teoria :*

$$\textit{Se} \quad \Re \equiv \aleph \quad e \; \Re \in \boldsymbol{Mod}(\Sigma) \quad \textit{então} \quad \aleph \in \boldsymbol{Mod}(\Sigma)$$

Definição A.1.7 (Modelos Isomórficos) *Dois Modelos são isomorfos, denotado por* $\Re \cong \aleph$*, se e somente se existe um isomorfismo de um para o outro.*

Definição A.1.8 (Expansão de Modelos) *Seja* $\mathcal{L} \subseteq \mathcal{L}'$*. Uma estrutura* $\Re'$ *para a linguagem* $\mathcal{L}'$ *é uma expansão de uma estrutura* $\Re$ *para uma linguagem* $\mathcal{L}$*, se* $\Re$ *e* $\Re'$ *concordam nas atribuições dadas aos símbolos do alfabeto de* $\mathcal{L}$*. Ao contrário,* $\Re$ *é o reduto de* $\Re'$ *pela linguagem* $\mathcal{L}$*.*

A.1.3 Teoria

Definição A.1.9 (Teoria) *uma teoria Σ é um conjunto de fórmulas bem formadas (wffs) pertencentes a linguagem $\mathcal{L}(\Sigma)$, que é fechado sob conseqüência lógica ($\models$), ou seja, para qualquer $\alpha \in \mathcal{L}(\Sigma)$, tem-se que,*

$$\Sigma \models \alpha \text{ se e somente se } \alpha \in \Sigma$$

Definição A.1.10 (Equivalência) *Duas Teorias são equivalentes, denotado por $\Sigma \equiv \Delta$, se e somente se suas classe de modelos são iguais.*

Definição A.1.11 (Conseqüência Lógica) *O conjunto de Conseqüências Lógicas de uma teoria Σ, denotada por $\boldsymbol{Cn}(\Sigma)$, é:*

$$\boldsymbol{Cn}(\Sigma) = \{\alpha \in \mathcal{L}(\Sigma) \setminus \Sigma \models \alpha\}$$

Definição A.1.12 (Especificação (ou Apresentação)) *especificação ou apresentação para uma teoria Σ é o par formado pela linguagem, denotado por $\mathcal{L}(\Sigma)$, e pela axiomatização, descrita por $\boldsymbol{Ax}(\Sigma)$. Onde a axiomatização é um conjunto de fómulas bem formadas pertencentes a Σ, tais que para qualquer $\alpha \in \mathcal{L}(\Sigma)$, tem-se que,*

$$\Sigma \models \alpha \text{ se e somente se } \boldsymbol{Ax}(\Sigma) \models \alpha$$

Preferencialmente, o conjunto de fórmulas $\boldsymbol{Ax}(\Sigma)$ deve ser finito, quando isto é possível.

Lema A.1.1 (Lema da Interpolação de Craig) *Assumindo linguagens de primeira ordem tais que, $\mathcal{L}_\Sigma = \mathcal{L}_{\Sigma'} \cap \mathcal{L}_\Delta$, e considerando conjuntos de sentenças de primeira ordem Σ', em $\mathcal{L}_{\Sigma'}$, e Δ, em $\mathcal{L}_\Delta$, tais que Σ' é consistente. Dada uma sentença ϕ de $\mathcal{L}_\Delta$ tal que,*

$$\Sigma' \cup \Delta \models \phi$$

existe um conjunto de sentenças I em $\mathcal{L}_\Sigma$ (chamado de Intepolantes) tais que:

i) $\Sigma' \models \psi$ *para toda sentença* $\psi \in I$; *e*

ii) $\Delta \cup I \models \phi$

Teorema A.1.2 (Teorema da Consistência de Robinson) *Caso Σ seja uma teoria consistente e Σ', Δ sejam extensões consistentes de Σ, onde pelo menos uma delas é conservativa, e $\mathcal{L}_\Sigma = \mathcal{L}_{\Sigma'} \cap \mathcal{L}_\Delta$ então $\Sigma' \cup \Delta$ é também consistente.*

A.2 Interpretação

A noção lógica de Interpretação, segundo adaptada as circunstâncias da implementação entre tipos abstratos de dados, foi retirada de [Vel87].

Definição A.2.1 (Função de Tradução) *Sejam duas linguagens de primeira ordem* $\mathcal{L}, \mathcal{L}'$. *Uma função* ζ *é uma função de tradução entre linguagens se e somente se* $\zeta : \mathcal{L} \longmapsto \mathcal{L}'$ *é um mapeamento dos símbolos de sorte, função e predicado de* $\mathcal{L}$ *sobre os símbolos de* $\mathcal{L}'$, *onde*

- *cada símbolo de sorte em* $\mathcal{L}$ *é mapeado em um símbolo de sorte de* $\mathcal{L}'$;
- *cada símbolo de função n-ário em* $\mathcal{L}$ *é mapeado em um símbolo de função também n-ário de* $\mathcal{L}'$; *e*
- *cada símbolo de predicado n-ário de* $\mathcal{L}$ *é mapeado em um símbolo de predicado n-ário de* $\mathcal{L}'$.

Usando a função de tradução ζ, define-se a tradução dos termos formados com o alfabeto de $\mathcal{L}$ em termos formados no alfabeto de $\mathcal{L}'$. As fórmulas de $\mathcal{L}$ podem ser igualmente traduzidas em fórmulas de $\mathcal{L}'$. Se $\alpha \in \mathcal{L}$ denota-se a tradução de α segundo ζ por α^{ζ}, definida indutivamente na estrutura das fórmulas, conforme abaixo:

i) Se α é uma fórmula atômica da forma $p(t_1, \ldots, t_n)$, onde p é um símbolo de predicado do alfabeto de $\mathcal{L}$, e $t_1, \ldots, t_n$ são termos em $\mathcal{L}$, então

$$(p(t_1, t_2, \ldots, t_n))^{\zeta} = p^{\zeta}((t_1)^{\zeta}, \ldots, (t_n)^{\zeta})$$

ii) Se α tem a forma de um dos esquemas de fórmulas abaixo sua tradução é dada do seguinte modo:

$$\begin{aligned}
(B \vee C)^{\zeta} &= B^{\zeta} \vee C^{\zeta} \\
(B \wedge C)^{\zeta} &= B^{\zeta} \wedge C^{\zeta} \\
(B \Rightarrow C)^{\zeta} &= B^{\zeta} \Rightarrow C^{\zeta} \\
(B \Leftrightarrow C)^{\zeta} &= B^{\zeta} \Leftrightarrow C^{\zeta} \\
(\neg B)^{\zeta} &= \neg((B)^{\zeta})
\end{aligned}$$

iii) Se α é da forma $(\forall v B)$ ou $(\exists v B)$, então:

$$\{(\forall v : S)B\}^{\zeta} = \{(\forall \zeta(v)) : S^{\zeta})B^{\zeta}\}$$
$$\{(\exists v : S)B\}^{\zeta} = \{(\exists \zeta(v)) : S^{\zeta})B^{\zeta}\}$$

Definição A.2.2 (Tradução de uma Teoria) *Denota-se como* $\zeta[\Sigma]$ *a teoria resultante da tradução de todas as fórmulas em* Σ*, segundo* ζ*.*

$$\zeta[\Sigma] = \{\alpha^{\zeta} \setminus \Sigma \models \alpha\}$$

Definição A.2.3 (Modelo Induzido) *Seja uma função de tradução* $\zeta : \mathcal{L} \longmapsto \mathcal{L}'$*. Considere* $\Re$ *uma estrutura para* $\mathcal{L}'$*. A estrutura para* $\mathcal{L}$ *induzida via* ζ*, é denotado por* $\Re^{\zeta^{-1}}$*, é construida do seguinte modo:*

i) *o domínio de cada sorte* S*, denotado por* $\Re^{\zeta^{-1}}|S|$ *é o mesmo domínio de* $\Re|\zeta(S)|$*;*

ii) $\Re^{\zeta^{-1}}|f| = \Re|\zeta(f)|$*, para cada símbolo de função* f *pertecente a* $\mathcal{L}$*;*

iii) $\Re^{\zeta^{-1}}|p| = \Re|\zeta(p)|$*, para cada símbolo de predicado* p *pertecente a* $\mathcal{L}$*;*

onde $\Re|s|$ *representa a realização do símbolo* s*, na estrutura* $\Re$*.*

Teorema A.2.1 (Teorema da Tradução) *Considere um função de tradução* ζ *da Linguagem* $\mathcal{L}_1$ *para a linguagem* $\mathcal{L}_2$*. Dado uma teoria de primeira ordem na linguagem de* $\mathcal{L}_1$ *e uma realização* $\Re$ *para* $\mathcal{L}_2$*, qualquer sentença* ϕ*, de* $\mathcal{L}_1$*, tem-se que os dois enunciados abaixo são válidos e equivalentes:*

i) $\Re^{\zeta^{-1}} \models \phi$ *se e somente se* $\Re \models \phi^{\zeta}$

ii) $\Sigma \models \phi$ *se e somente se* $\zeta[\Sigma] \models \phi^{\zeta}$

onde $\Re^{\zeta^{-1}}$ *é a realização induzida para* $\mathcal{L}_1$*, por* ζ*; e* ϕ^{ζ} *é a tradução de* ϕ *segundo* ζ*; e* $\zeta[\Sigma]$ *a tradução de todas as fómulas em* Σ *segundo* ζ*. i) representa a versão semantica enquanto ii) a estabelece sintaticamente.*

Definição A.2.4 (Interpretação) *Considere teorias* Σ *e* Δ. *Diz-se que existe uma interpretação de* Σ *em* Δ, *denotado por* $\Sigma \xrightarrow{\zeta} \Delta$, *para alguma função de tradução* ζ *da linguagem de* Σ *sobre a linguagem de* Δ ($\zeta : \mathcal{L}(\Sigma) \longmapsto \mathcal{L}(\Delta)$), *se*

$$\Sigma \models \alpha \text{ implica em } \Delta \models \alpha^{\zeta}$$

Como conseqüência direta do Teorema da Tradução (A.2.1) e da definição de Interpretação (A.2.4), seja o teorema que dá significado a definição de Intepretação.

Teorema A.2.2 (Teorema da Interpretação) *Considere teorias* Σ *e* Δ. *Uma função de tradução* ζ *da linguagem de* Σ *sobre a linguagem de* Δ ($\zeta : \mathcal{L}(\Sigma) \longmapsto \mathcal{L}(\Delta)$), *existe uma interpretação de* Σ *em* Δ, *denotado por* $\Sigma \xrightarrow{\zeta} \Delta$, *se e somente se :*

i) *Se* $\Sigma \models \alpha$ *então* $\Delta \models \alpha^{\zeta}$

ii) *Se* $\Re \in \boldsymbol{Mod}(\Delta)$ *então* $\Re^{\zeta^{-1}} \in \boldsymbol{Mod}(\Sigma)$

Teorema A.2.3 (Modelos Induzidos) *Se* $\Sigma \xrightarrow{\zeta} \Delta$, *é uma interpretação, então para qualquer modelo* $\Re$ *de* Δ, $\Re^{\zeta^{-1}}$ *é um modelo de* Σ. *Isto é, a interpretação garante que os modelos de* Δ *induzem modelos para* Σ.

Teorema A.2.4 *Se* $\Gamma \xrightarrow{\zeta} \Gamma'$ *então para cada modelo* $\Re \in \boldsymbol{Mod}(\Gamma')$ *existe um modelo* $\aleph \in \boldsymbol{Mod}(\Gamma)$, *onde* $\Re \cong \aleph$.

Prova : Simplesmente tomando como $\aleph$ a realização induzida por $\Re$ em Γ, ou seja, $\aleph = \Re^{\eta}$, por construção de modelo induzido (veja a definição A.2.3) existe um isomorfismo natural de $\Re$ em $\Re^{\eta}$, dado pela identidade ([End72]). Isto poderia ser verificado detalhadamente por indução em fórmulas. □

Teorema A.2.5 *Sejam* Γ *e* Γ' *duas teorias. Se para cada* $\Re \in \boldsymbol{Mod}(\Gamma')$ *existe um modelo isomórfico* $\aleph \in \boldsymbol{Mod}(\Gamma)$, *então* $\Gamma \longrightarrow \Gamma'$.

Prova : Considere ζ, uma função de tradução entre as linguagens de ambas (i.e. $\zeta : \mathcal{L}(\Gamma) \longmapsto \mathcal{L}(\Gamma')$) construida de forma que, para cada símbolo s' em $\mathcal{L}(\Gamma')$, e s em $\mathcal{L}(\Gamma)$:

$$\zeta(s) = s'$$

se e somente se para cada $\aleph \in \mathbf{Mod}(\Gamma)$ e $\Re \in \mathbf{Mod}(\Gamma')$, onde $\Re \cong \aleph$, $\Re|s| = \aleph|s'|$.

Isto é sempre possível, pois $\Re$ e $\aleph$ são isomórficos, e ζ está sendo construido de modo que $\aleph$ seja o modelo induzido por $\Re$ segundo ζ. Então, diretamente, para qualquer $\alpha \in \mathcal{L}(\Sigma)$, $\aleph \models \alpha$ se e somente se $\Re \models \alpha^{\zeta}$, logo pelo Teorema da Tradução, $\Gamma \xrightarrow{\zeta} \Gamma'$. □

Teorema A.2.6 $\Sigma \xrightarrow{\zeta} \Delta$ *é uma interpretação, se e somente se para todo axioma* γ *na especificação (axiomatização) de* Σ *(ou seja,* $\gamma \in \boldsymbol{Ax}(\Sigma)$ *),* $\Delta \models \gamma^{\zeta}$

Teorema A.2.7 (Transitividade) *A interpretação é transitiva, ou seja,*

$$\{\Sigma_1 \xrightarrow{\zeta_1} \Sigma_2 \ \wedge \ \Sigma_2 \xrightarrow{\zeta_2} \Sigma_3\} \ \Rightarrow \ \Sigma_1 \xrightarrow{\zeta_2 \cdot \zeta_1} \Sigma_3$$

Teorema A.2.8 (Associatividade) *A interpretação é associativa, ou seja,*

$$\zeta_1[\Sigma_1] \xrightarrow{\zeta_2} \Sigma_3 \ \Leftrightarrow \ \Sigma_1 \xrightarrow{\zeta_1} \zeta_2[\Sigma_2] \ \Leftrightarrow \ \Sigma_1 \xrightarrow{\zeta_1} \Sigma_2 \xrightarrow{\zeta_2} \Sigma_3$$

A.3 Extensões

Definição A.3.1 (Extensão) *Considere teorias* Σ *e* Σ'*. Diz-se que* Σ' *estende* Σ*, denotado por* $\Sigma \subseteq \Sigma'$*, se e somente se* $\mathcal{L}(\Sigma) \subseteq \mathcal{L}(\Sigma')$*, e para qualquer fórmula* $\alpha \in \mathcal{L}(\Sigma)$

Se $\Sigma \models \alpha$ *então* $\Sigma' \models \alpha$

Definição A.3.2 (Extensão Conservativa) *Considere as teorias* Σ *e* Σ'*.* Σ' *estende* Σ *conservativamente, denotado por* $\Sigma \leq \Sigma'$*, se e somente se* $\Sigma \subseteq \Sigma'$*, e para qualquer fórmula* $\alpha \in \mathcal{L}(\Sigma)$

Se $\Sigma' \models \alpha$ *então* $\Sigma \models \alpha$

Teorema A.3.1 (Consistência de Robinson Estendido) *Seja* Σ*,* Σ' *e* Δ *teorias de primeira ordem, onde* Σ' *é consistente e* $\mathcal{L}_{\Sigma} = \mathcal{L}_{\Sigma'} \cap \mathcal{L}_{\Delta}$*. Se*

$$\begin{array}{ccc} \Delta & & \\ \mathbb{V} & & \\ \Sigma & \subseteq & \Sigma' \end{array}$$

Então,

$$\begin{array}{ccc} \Delta & \subseteq & \Sigma' \cup \Delta \\ \mathbb{V} & & \mathbb{V} \\ \Sigma & \subseteq & \Sigma' \end{array}$$

Definição A.3.3 (Extensão Expansiva) *Considere as teorias Σ e Σ'. Diz-se que Σ' estende Σ expansivamente, denotado por $\Sigma \trianglelefteq \Sigma'$, se e somente se $\Sigma \subseteq \Sigma'$, e qualquer modelo para Σ se expande a um modelo para Σ'.*

$$\Re|_{\mathcal{L}(\Sigma)} \in \boldsymbol{Mod}(\Sigma) \Rightarrow \Re \in \boldsymbol{Mod}(\Sigma')$$

Teorema A.3.2 *Se a extensão $\Sigma \trianglelefteq \Sigma'$ é expansiva, então é conservativa.*

A.3.1 Extensões por Definição

As definições e os teoremas apresentados a seguir, foram consultados em [VV90] e [VV91].

Definição A.3.4 (Definição de Símbolo de Predicados) *Considere a linguagem $\mathcal{L}$ incluindo os símbolos de sortes S_1,S_2,...,S_n, e seja p um símbolo não pertencente a esta linguagem. Considere ainda, π uma fórmula em $\mathcal{L}$ sem variáveis livres, com execessão de v_1,v_2,...,v_n. Seja β_p a seguinte sentença,*

$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)\{\pi \Leftrightarrow p(v_1, v_2, \ldots, v_n)\}$$

Esta sentença é chamada Definição do Símbolo de Predicado *p.*

Definição A.3.5 (Extensão por Definição de Predicados) *Seja a teoria Σ e $\mathcal{L}$ a sua linguagem. Considere a nova teoria Σ', obtida a partir de Σ, introduzindo o símbolo p na linguagem $\mathcal{L}$, como um símbolo de predicado sobre os sortes S_1,S_2,...,S_n, em $\mathcal{L}$, ou seja, $p(S_1,S_2,\ldots,S_n)$. E também acrescentando às sentenças pertencentes à axiomatização de Σ, uma definição do símbolo de predicado p (conforme o axioma A.3.4). Então, diz-se que Σ' é uma* Extensão de Σ por Definição do Símbolo de Predicado *p.*

Teorema A.3.3 *Seja uma teoria Σ' uma extensão de uma teoria Σ por definição de símbolo de um predicado p, e a fórmula β_p define o predicado p (veja o axioma A.3.4). Então $\Sigma \leq \Sigma'$*

Definição A.3.6 (Definição de Símbolo de Função) *Considere a linguagem $\mathcal{L}$ incluindo os símbolos de sortes $S_1,S_2,\ldots,S_n,S$, e seja f um símbolo não pertencente a esta linguagem. Considere ainda, ϕ uma fórmula em $\mathcal{L}$ sem variáveis livres, além de $v_1,v_2,\ldots,v_n$ e v. Seja β_f a seguinte sentença:*

$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)(\forall v : S)\{\phi \Leftrightarrow f(v_1, v_2, \ldots, v_n) = v\}$$

Esta sentença β_f é chamada Definição do Símbolo de Função *f.*

Definição A.3.7 (Extensão por Definição de Funções) *Seja uma teoria Σ e $\mathcal{L}$ a sua linguagem. Seja a nova teoria Σ', obtida a partir de Σ, introduzindo o símbolo f na linguagem $\mathcal{L}$, como um símbolo de função sobre os sortes $S_1,S_2,\ldots,S_n$, S, em $\mathcal{L}$, ou seja, $f(S_1,S_2,\ldots,S_n) \Rightarrow S$. E também acrescentando às sentenças pertencentes à axiomatização de Σ, uma definição do símbolo de função f (conforme o axioma A.3.6). Então, diz-se que Σ' é uma* Extensão de Σ por Definição do Símbolo de Função *f.*

Teorema A.3.4 *Seja uma teoria Σ' uma extensão de uma teoria Σ por definição de símbolo de um função f, e a fórmula β_f define a função f, de acordo com o axioma A.3.6. Considere as fórmulas ϵ_f e υ_f abaixo, e as variáveis u e u' não em $\mathcal{L}$,*

$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)(\exists u : S)\{\phi[v/u]\}$$
$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)(\forall u, u' : S)\{\phi[v/u] \wedge \phi[v/u'] \Rightarrow u = u'\}$$

As fórmulas ϵ_f e υ_f são chamadas de condição de existência e de unicidade respectivamente. Se ϵ e υ pertencem as conseqüências de Σ, então Σ' é uma extensão conservativa de Σ.

$$[(\epsilon_f \wedge \upsilon_f) \in Cn(\Sigma)] \Rightarrow \Sigma \leq \Sigma'$$

Definição A.3.8 (Definição Liberal por Comportamento de E/S) *Seja a linguagem $\mathcal{L}$ incluindo os símbolos de sortes $S_1,S_2,\ldots,S_n$, S, e seja f um símbolo não pertencente a esta linguagem. Considere ainda, ϕ uma fórmula em $\mathcal{L}$ sem variáveis livres, além de $v_1,v_2,\ldots,v_n$. Considere ψ uma fórmula em $\mathcal{L}$ sem variáveis livres, além das $v_1,v_2,\ldots,v_n$ e v. Seja λ_f a seguinte sentença,*

$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_n : s_n)\{\phi \Rightarrow \psi[v/f(v_1, v_2, \ldots, v_n)]\}$$

Esta sentença λ_f é chamada Definição Liberal do Símbolo de Função *f.*

Definição A.3.9 (Extensão por Definição Liberal por Comportamento de E/S) *Seja Σ uma teoria e $\mathcal{L}$ a sua linguagem. Considere a nova teoria Σ', obtida a partir de Σ, introduzindo o símbolo f em sua linguagem $\mathcal{L}$, como um símbolo de função sobre os sortes $S_1,S_2,\ldots,S_n$, em $\mathcal{L}$, ou seja, $f(S_1,S_2,\ldots,S_n) \Rightarrow S$. E também acrescentando às sentenças pertencentes à axiomatização de Σ, uma definição liberal do símbolo de função f (conforme o axioma A.3.8). Então, diz-se que Σ' é uma* Extensão de Σ por Definição Liberal do Símbolo de função f por Comportamento de E/S .

Teorema A.3.5 *Seja uma teoria Σ' uma extensão de uma teoria Σ por definição liberal de símbolo de um função f, e a fórmula λ_f define liberalmente a função f, conforme o axioma A.3.8. Considere a fórmula λ_f abaixo, e a variável $u : S$ não em $\mathcal{L}$,*

$$(\forall v_1 : s_1, v_2 : s_2, \ldots, v_k : s_k)(\exists u : S)\{\phi \Rightarrow \psi[v/u]\}$$

A fórmula λ_f é chamada de condição de possibilidade. Representa a condição de conservatividade para λ_f. Se λ_f pertence as conseqüências de Σ, então Σ' é uma extensão conservativa de Σ.

$$[\lambda_f \in Cn(\Sigma)] \Rightarrow \Sigma \leq \Sigma'$$

A.3.2 Extensões por Definição de Sortes

Resumindo sucintamente [MV92], são apresentados abaixo três formas para estender teoria de primeira ordem, pela introdução de novos sortes na linguagem, de maneira que a teoria resultante seja sempre uma extensão expansiva da anterior.

Definição A.3.10 (Produto Cartesiano) *Considere a linguagem $\mathcal{L}$ incluindo os símbolos de sortes A e B, e seja uma nova linguagem $\mathcal{L}'$, acrescentando em $\mathcal{L}$, um novo sorte S, $p_A : S \longmapsto A$ e $p_B : S \longmapsto B$ dois símbolos de função não pertencentes a $\mathcal{L}$. Considere ainda, os axiomas :*

$$(\forall a : A)(\forall b : B)(\exists s : S)\{p_A(s) \stackrel{A}{=} a \ \wedge \ p_B(s) \stackrel{B}{=} b\}$$
$$(\forall s, s' : S)\{(p_A(s) \stackrel{A}{=} p_A(s') \ \wedge \ p_B(s) \stackrel{B}{=} p_B(s')) \Rightarrow s \stackrel{S}{=} s'\}$$

Estas sentença definem o produto cartesiano de $A \times B$, onde p_A e p_B são as respectivas projeções.

Definição A.3.11 (Subsorte) *Considere a linguagem $\mathcal{L}$ incluindo o símbolo de sorte A, e uma relação r sobre A, e seja uma nova linguagem $\mathcal{L}'$, acrescentando em $\mathcal{L}$, um novo sorte S, $i : S \longmapsto A$ um símbolo de função não pertencente a $\mathcal{L}$. Considere ainda, os axiomas :*

$$(\forall a : A)(\exists s : S)\{r(a) \quad \Leftrightarrow \quad a \stackrel{A}{=} i(s)\}$$
$$(\forall s, s' : S)\{(i(s) \stackrel{A}{=} i(s') \quad \Rightarrow \quad s \stackrel{S}{=} s'\}$$

Estas sentença definem o sobsorte S de A, pelo predicado de relativisação i .

Definição A.3.12 (Quociente) *Considere a linguagem $\mathcal{L}$ incluindo o símbolo de sorte A, e uma relação q de equivalência em $A \times A$, e seja uma nova linguagem $\mathcal{L}'$, acrescentando em $\mathcal{L}$, um novo sorte S, o símbolo de função $p : A \longmapsto S$ não pertencente a $\mathcal{L}$. Considere ainda, os axiomas :*

$$(\forall a, a' : A)\{q(a, a') \quad \Leftrightarrow \quad p(a) \stackrel{S}{=} p(a')\}$$
$$(\forall s : S)(\exists a : A)\{p(a) \stackrel{S}{=} s\}$$

Estas sentença definem o sorte S quociente de A pela relação de equivalência q.

A extensão da linguagem de uma teoria de conformidade com alguma das três definições acima, e a adição dos respectivos axiomas γ_S e γ'_S, resulta em uma extensão expansiva. No segundo caso A.3.11, porém a teoria extendida já deveria derivar que a rela¸ao r é não vazia.

Apêndice B

Lógica *Default*

Uma formalização do raciocínio por *defaults* foi proposta por Raymond Reiter em [Rei80], e posteriormente revisada e refinada em [RC83]. Esta proposta se apresenta bastante adequada a formalização do raciocínio de senso comum devido a sua simplicidade conceitual.

Nesta proposta, as regras de inferência não monotônica são representadas por expressões linguísticas especiais chamadas de *defaults*.

Em lógica *default* o raciocínio de senso-comum sobre o universo sendo modelado é representado como uma teoria *default*. Isto é, um par que consiste de um conjunto de sentenças de primeira ordem, chamadas de axiomas da teoria, e um conjunto de *defaults*. Intuitivamente, os axiomas de uma teoria representam o conhecimento logicamente válido, mas geralmente incompleto sobre o mundo, enquanto os *defaults* permitem extender estas informações através conclusões plausíveis, porém não necessariamente verdadeiras.

Qualquer conjunto de crenças suportadas racionalmente sobre o mundo sendo modelado por uma teoria *default* é chamado de extensão da teoria. Intuitivamente, uma extensão contém todas as sentenças que podem ser derivados dos axiomas por inferência clássica de primeira ordem ou por aplicação dos *defaults*.

Agora, formaliza-se cada um dos três conceitos fundamentais apresentados informalmente acima.

B.1 Defaults

Definição B.1.1 (***Default***) *é qualquer expressão da forma :*

$$\frac{\alpha(\bar{x}) : \beta_1(\bar{x}), ..., \beta_n(\bar{x})}{\omega(\bar{x})}$$

onde $\alpha(\bar{x}), \beta_1(\bar{x}), ..., \beta_n(\bar{x}), \omega(\bar{x})$ *são fórmulas de primeira ordem poli-sortida, cuja as variáveis estão entre* $\bar{x} = (x_1, ..., x_k)$. $\alpha(\bar{x})$ *é chamado de Pré-requisito, enquanto* $\beta_1(\bar{x}), ..., \beta_n(\bar{x})$ *são chamados de Justificativas, e* $\omega(\bar{x})$ *é o Consequente do* default.

Lê-se *para todo* $\bar{x}$ *se* $\alpha(\bar{x})$ *é válido, e é consistente acreditar em* $\beta_1(\bar{x}), ..., \beta_n(\bar{x})$ *então é possível acreditar em* $\omega(\bar{x})$

Definição B.1.2 (***Default* Fechado**) *Um* default

$$\frac{\alpha : \beta_1, \beta_2, \ldots, \beta_n}{\omega}$$

é dito fechado se e somente se nenhum entre α, β_1,...,β_n *e* ω *contém variávies livres.*

B.2 Teorias *Default*

Definição B.2.1 *(Teoria* Default*) Uma teoria* default *é um par* $\Sigma = \langle W, D \rangle$, *onde* W *é um conjunto de fórmulas de primeira ordem poli-sortida, os axiomas de* Σ, *e* D *um conjunto de* defaults.

Definição B.2.2 *Uma teoria* default $\Sigma = \langle W, D \rangle$ *é considerada fechada se todos os* defaults *pertencente a* D *são fechados, e todos os axiomas estão propriamente quantificados.*

B.3 Extensões *Default*

Existem três condições que podem ser razoavelmente impostas para que um conjunto de sentenças $\mathcal{E}$ seja considerado uma extensão default de uma teoria default $\Sigma = \langle W, D \rangle$:

- $\mathcal{E}$ deve ser fechado com respeito a dedução clássica : $\mathcal{E} = \mathbf{Cn}(\mathcal{E})$.

- $\mathcal{E}$ deve conter os axiomas da teoria Σ : $W \subseteq \mathcal{E}$.
- $\mathcal{E}$ deve conter o conjunto maximal de conclusões que podem ser obtidas pela aplicação dos *defaults* de D.

Assim, formalizando,

Definição B.3.1 (Extensão *Default* de uma Teoria *Default* Fechada) *Considere uma teoria* default *fechada* $\Sigma = \langle W, D \rangle$ *sob uma linguagem* $\mathcal{L}$. *Para qualquer conjunto* $\mathcal{E} \subseteq \mathcal{L}$, *seja* $\Gamma(\mathcal{E})$ *o menor conjunto de* $\mathcal{L}$ *satisfazendo:*

- $\mathcal{E}$ *deve ser fechado em relação à dedução clássica,* $\Gamma(\mathcal{E}) = \boldsymbol{Cn}(\Gamma(\mathcal{E}))$;
- $\mathcal{E}$ *deve conter os axiomas da teoria* Σ : $W \subseteq \Gamma(\mathcal{E})$.
- *Se* $\frac{\alpha:\beta_1,\beta_2,\ldots,\beta_n}{\omega}$ *pertence a* D *e* $\alpha \in \Gamma(\mathcal{E})$ *e a negação das justificativas,* $\neg\beta_1, \ldots, \neg\beta_n$ *não pertence a* $\mathcal{E}$ *então* $\omega \in \Gamma(\mathcal{E})$.

o conjunto $\mathcal{E} \subseteq \mathcal{L}$ *é uma extensão se de* Σ, *se somente se,* $\mathcal{E} = \Gamma(\mathcal{E})$ *(onde* $\mathcal{E}$ *é o ponto fixo do operador* Γ*).*

Teorema B.3.1 *Se* $\Sigma = \langle W, D \rangle$ *é uma teoria* default *fechada, então um conjunto de sentenças* $\mathcal{E}$ *é uma extensão de* Σ *se e somente se:*

$$\mathcal{E} = \bigcup_{i=0}^{\infty} \mathcal{E}_i$$

onde

$$\begin{aligned} \mathcal{E}_0 &= W \\ \mathcal{E}_{i+1} &= \boldsymbol{Cn}(\mathcal{E}_i) \cup \{\omega \setminus \tfrac{\alpha:\beta_1,\beta_2,\ldots,\beta_n}{\omega} \in D \\ &\qquad \text{onde } \alpha \in \mathcal{E}_i \text{ e } \neg\beta_1, \neg\beta_2, \ldots, \neg\beta_n \notin \mathcal{E}'\} \end{aligned}$$

Teorema B.3.2 *Seja* $\Sigma = \langle W, D \rangle$ *é uma teoria* default *fechada, então qualquer extensão* default $\mathcal{C}$, *para* Σ, *é tal que:*

$$\mathcal{C} \equiv W \cup \boldsymbol{Consequentes}\,(\boldsymbol{GD}\,(\mathcal{C}, \Sigma))$$

onde $\boldsymbol{GD}\,(\mathcal{C}, \Sigma))$ *representa o conjunto de* defaults *geradores, isto é, o conjunto de todos os* defaults *em* D *aplicados na geração de* $\mathcal{C}$, *de acordo com B.3.1, e* $\boldsymbol{Consequentes}\,(A)$ *denota a conjunção do consequente de todos os* defaults *em um conjunto de* defaults A.

Corolário B.3.1 *Uma teoria* default *fechada* $\Sigma = \langle W, D\rangle$ *tem uma extensão inconsistente se e somente se* W *é inconsistente.*

Corolário B.3.2 *Uma teoria* default *fechada* $\Sigma = \langle W, D\rangle$ *tem uma extensão inconsistente então esta é a sua única extensão.*

Teorema B.3.3 (Minimalidade das Extensões) *Se* $\mathcal{E}$ *e* $\mathcal{F}$ *são extensões de uma teoria* default *e* $\mathcal{E} \subseteq \mathcal{F}$ *então* $\mathcal{E} = \mathcal{F}$.

Definição B.3.2 *Um teoria* default *fechada* $\Sigma = (D, W)$ *é finitamente axiomatizável, se* D *é finito e* W *é finitamente axiomatizável.*

Proposição B.3.1 *Se* $\Sigma = (D, W)$ *é uma teoria* default *finitamente axiomatizável, então cada extensão* default $\mathcal{E}$ *para* Σ *é uma teoria finitamente axiomatizável. Mais ainda,* $ExtDef(\Sigma)$, *o conjunto de todas as extensões* default *de* Σ, *é finito.*

Prova : ... ver [AV93b]. □

B.3.1 Teorias *Default* Normais

Definição B.3.3 (*Default* Normal) *é qualquer* default *da forma:*

$$\frac{\alpha(\bar{x}) : \omega(\bar{x})}{\omega(\bar{x})}$$

Definição B.3.4 (Teoria Normal) *uma teoria* $\Sigma = \langle W, D\rangle$ *é considerada normal se e somente se todos os* defaults *em* D *são normais.*

Há outro critério para determinar se um determinado conjunto de crenças é na verdade uma extensão *default* de uma teoria *default* normal. Antes, sejam as definições :

Definição B.3.5 (*Default* Aplicável) *Seja* $\frac{\alpha:\omega}{\omega}$ *um* default *normal fechado, suponha que* $\mathcal{E}$ *seja um conjunto de sentenças qualquer.* $\frac{\alpha:\omega}{\omega}$ *é aplicável em relação a* $\mathcal{E}$ *se e somenete se* $\mathcal{E} \models \alpha$ *e* $\mathcal{E} \not\models \neg\omega$.

Definição B.3.6 (Mapeamento) *Seja $\mathcal{E}$ um conjunto de sentenças. Para cada* default $\frac{\alpha:\omega}{\omega}$ *é associado o mapeamento, denotado por d, dado como :*

$$d(\mathcal{E}) = \begin{cases} \boldsymbol{Cn}(\mathcal{E} \cup \{\omega\}) & \text{se } \frac{\alpha:\omega}{\omega} \text{ é aplicável com relação a } \mathcal{E}. \\ \mathcal{E} & \text{caso contrário} \end{cases}$$

Definição B.3.7 (Estabilidade) *Seja $\mathcal{E}$ um conjunto de sentenças, e suponha que D seja um conjunto de* defaults *normais. $\mathcal{E}$ é estável com relação a D se e somente se $d(\mathcal{E}) = \mathcal{E}$ para qualquer* default *em D.*

Definição B.3.8 (Acessbilidade) *Seja W um conjunto de sentenças, e suponha que D seja um conjunto de* defaults *normais, e assuma ainda que $\mathcal{E}$ é um conjunto de senteças. $\mathcal{E}$ é acessível de W com relação a D se e somente se para cada conjunto de sentenças $\mathcal{S}$ tal que $W \subseteq \mathcal{S} \subset \mathcal{E}$, existe algum* default *em D de forma que $\mathcal{S} \subset d(\mathcal{S}) \subseteq \mathcal{E}$.*

Teorema B.3.4 (Extensão *Default* de uma Teoria *Default* Normal) *Considere $\Sigma = \langle W, D \rangle$ uma teoria* default *normal, e seja $\mathcal{E}$ um conjunto de sentenças. Então $\mathcal{E}$ é uma extensão* default *para Σ se e somente se :*

i) $W \subseteq \mathcal{E}$

ii) *E é estável com relação a D*

iii) *E é acessível de W com relação a D*

As teorias *default* normais apresentam três propriedades desejáveis representadas pelos teoremas a seguir:

Teorema B.3.5 *Toda teoria* default *normal tem pelo menos uma extensão.*

Teorema B.3.6 (Ortogonalidade das Extensões) *Se uma teoria* default *normal tem estensões distintas $\mathcal{E}$ e $\mathcal{F}$, então $\mathcal{E} \cup \mathcal{F}$ é inconsistente.*

Teorema B.3.7 (Semi-Monotonicidade) *Sejam D_1 e D_2 conjuntos de* defaults *normais fechados tais que $D_1 \subseteq D_2$. Seja $\mathcal{E}_1$ uma extensão de $\Sigma_1 = \langle W, D_1 \rangle$ e suponha que $\Sigma_2 = \langle W, D_2 \rangle$. Então Σ_2 tem uma extensão $\mathcal{E}_2$ tal que $\mathcal{E}_1 \subseteq \mathcal{E}_2$.*

B.3.2 Teorias *Default* Semi-Normais

Para evitar essas inferências não intuitivas, faz-se necessário a utilização de teorias *default* semi-normais. Estas regras, que modelam exceções podem ser representadas siplesmente por *defaults* semi-normais.

Definição B.3.9 (*Default* Semi-Normal) *é qualquer* default *da forma:*

$$\frac{\alpha(\bar{x}) : \omega(\bar{x}) \wedge \mu(\bar{x})}{\omega(\bar{x})}$$

Definição B.3.10 (Teoria Semi-Normal) *uma teoria* $\Sigma = \langle W, D \rangle$ *é considerada semi-normal sse todos os* defaults *em* D *são semi-normais.*

Qualquer situação do senso comum que poder ser modelada usando *defaults* com uma só justificativa, pode ser modelada simplesmente por uma teoria semi-normal.

Regras que em isolado podem naturalmente ser representadas por *defaults* normais, quando consideradas em algum contexto podem ter circunstâncias excepcionais que façam a sua aplicação inaceitável.

As teorias semi-normais nem sempre apresentam extensões *default*, e em geral, não atendem a semi-monotonicidade.

A aplicação de um *default* aplicável pode ocasionar o bloqueio da extensão por qualquer uma das três seguintes razões:

i) O consequente de um *default*, junto com os axiomas e os consequentes de outros *defaults* já aplicados, contradiz uma de suas próprias justificativas.

ii) O consequente de um *default* juntamente como os axiomas e os consequentes dos outros *defaults* já aplicados, negam a justificativas de algum dos *defaults* já aplicados.

iii) O consequente do *default* contradiz alguma sentença derivável dos axiomas e dos consequentes de outros *defaults* já aplicados.

Bibliografia

[AV93a] José E. C. Araújo e Sheila R. M. Veloso. Properties of default logic and its application to formal specification of programs. Research Report ES 276/93, Prog. Sistemas e Computação/COPPE – Univ. Federal do Rio de Janeiro, Rio de Janeiro, 1993.

[AV93b] José E. C. Araújo e Sheila R. M. Veloso. Some monotonic properties of default logic enable it to be used in the formal specification of programs. *Forthcoming*, 1993.

[EFT84] Heinz-Dieter Ebbinghaus, J. Flum, e W. Thomas. *Mathematical Logic.* Springer Verlag, Berlin, 1984.

[End72] Herbert F. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[ER83] D. Etherington e Raymond Reiter. On inheritance hierarchies with exceptions. Em *Proceedings of AAAI-83*, páginas 104–108, 1983.

[Hoa85] Charles A. R. Hoare. *Communicating Sequential Process.* Prentice Hall, New York, 1985.

[Luk90] Wiltold Lukaszewicz. *Non-Monotonic Reasoning.* Ellis Harwood, 1990.

[Mai86] Thomas S.E. Maibaum. The role of abstraction in program development. Em H.-J. Kugler, editor, *Information Processing '86*, páginas 135–142. North Holland, 1986.

[MFS90] Thomas S.E. Maibaum, Jose Fiadeiro, e Martin Sadler. Stepwise program development in π-institutions. Draft, Depto. Computing - Imperial College of Science, Technology and Medicine, London, 1990.

[MS85] Thomas S.E. Maibaum e Martin Sadler. Axiomatizing specifications theory. Em H.-J. Kreowski, editor, *Recent Trends in Data Type Specification*, páginas 171–177. Springer Verlag, Berlin, 1985.

[MS90] Thomas S.E. Maibaum e Martin Sadler. Requirements on logics for specifications. Draft, Depto. Computing - Imperial College of Science, Technology and Medicine, London, 1990.

[MT84] Thomas S.E. Maibaum e Wladislaw M. Turski. On what exactly is going on when a software is developed step-by-step. Em *Proceedings of the 7th International Conference on Software Engeneering*, páginas 528–533. IEEE Computer Society, 1984.

[MV81] Thomas S.E. Maibaum e Paulo A. S. Veloso. A logic theory of data types motived by programming. Technical Report 81/28, Depto. Computing - Imperial College of Science, Technology and Medicine, London, 1981.

[MV92] Claudia Meré e Paulo A. S. Veloso. On extensions by sorts. Monografia MCC 38/92, Depto. Informática - Pontifícia Universidade Católica do Rio de Janeiro, dez. 1992.

[MVS84] Thomas S.E. Maibaum, Paulo A. S. Veloso, e Martin Sadler. Logical specification and implementation. Em M. Joseph e R. Shyamasundar, editores, *Foundations of Software Technology and Theoretical Computer Science LNCS 181*, páginas 13–30. Springer Verlag, 1984.

[MVS85] Thomas S.E. Maibaum, Paulo A. S. Veloso, e Martin R. Sadler. A theory of abstract data types for program development : bridging the gap. Em H. Ehrig, C. Floyd, M. Nivat, e J. Thatcher, editores, *Formal Methods and Software Development - vol 2 - Colloquium on Software Engeneering*, páginas 214–230. Springer Verlag, Berlin, 1985.

[PV83] Francisco E. P. Pessoa e Paulo A. S. Veloso. Introdução a programação com tipos abstratos de dados. Monografia MCC 1/83, Depto. Informática - Pontifícia Universidade Católica do Rio de Janeiro, 1983.

[RC83] Raymond Reiter e G. Criscuolo. Some representational issues on default reasoning. *Int. J. Computer and Mathematics*, 9:1–13, 1983.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial Inteligence*, 13:81–132, 1980.

[Sav90] Pedro Savadovsky. Algorithms for implementation between specifications. *Draft*, 1990.

[Sho67] Joseph E. Shoenfield. *Mathematical Logic.* Addison Wesley, Reading, 1967.

[Som89] Ian Sommerville. *Software Engeneering.* Addison Wesley, Reading, 1989.

[TM87] Wladislaw M. Turski e Thomas S.E. Maibaum. *The Specification of Computer Programs.* Addison Wesley, Wokingham, 1987.

[vD89] Dirk van Dalen. *Logic and Structure.* Springer Verlag, Berlin, 1989.

[Vel80] Paulo A. S. Veloso. Divide and conquer via data types. Em *Annales de la 8o Conf. LatinoAmericana de Informatica*, páginas 530–539, Lima, 1980.

[Vel83] Paulo A. S. Veloso. Problem solving via divide and conquer and abstract data types. Monografia MCC 1/83, Depto. Informática - Pontifícia Universidade Católica do Rio de Janeiro, 1983.

[Vel85a] Paulo A. S. Veloso. On the role of (data) abstraction in program development and problem solving. Monografia MCC 6/85, Depto. Informática - Pontifícia Universidade Católica do Rio de Janeiro, 1985.

[Vel85b] Paulo A. S. Veloso. Program development as theory manipulations. Monografia MCC 4/85, Depto. Informática - Pontifícia Universidade Católica do Rio de Janeiro, 1985.

[Vel87] Paulo A. S. Veloso. *Especificação e Verificação de Programas com Tipos de Dados.* Ed. Edgar Blucher, São Paulo, 1987.

[Vel88] Paulo A. S. Veloso. Problem solving by interpretation of theories. Em W. Carnielli e L. Alcantara, editores, *Methods and Applications of Mathematical Logic Contemporary Mathematics vol. 69*, páginas 241–250. American Mathematical Society, 1988.

[Vel91] Paulo A. S. Veloso. A computer-like example of conservative and non expansive extensions. Technical Report 91/36, Depto. Computing - Imperial College of Science, Technology and Medicine, London, 1991.

[Vel92a] Paulo A. S. Veloso. The modularization theorem for logical specification. Em *International Federation for Information Processing (IFIP) - WG 2.1 Meeting*, Agsburg - Alemanha, 1992.

[Vel92b] Paulo A. S. Veloso. Notes on interpretations of logical specifications. Research Report ES 277/93, COPPE /UFRJ - Universidade Federal do Rio de Janeiro, 1992.

[Vel92c] Paulo A. S. Veloso. On the modularization theorem for logical specification : its role and proof. Monografia MCC 17/92, Depto. Informática - Pontifícia Universidade Católica do Rio de Janeiro, 1992.

[Vel93] Paulo A. S. Veloso. A new, simpler proof of the modularization theorem. *Bulletin of the IGPL (Interest Group on Propositional and Predicate Logic*, jun, 1993.

[VMS85] Paulo A. S. Veloso, Thomas S.E. Maibaum, e Martin R. Sadler. Program development and theory manipulations. Em *Proc. of 3rd Intern. Workshop on Software Specification and Design*, páginas 155–162, Los Angeles, 1985.

[VV81] Paulo A. S. Veloso e Sheila R. M. Veloso. Problem decomposition and reduction : applicability, soundness and completeness. Em R. Trappl, J. Klir, e F. Pinchler, editores, *Progress in Cybernetic and Systems Research*, volume VIII. Hemisphere, 1981.

[VV90] Paulo A. S. Veloso e Sheila R. M. Veloso. On extensions by functions symbols : Conservativiness and comparation. Research report, COPPE /UFRJ - Universidade Federal do Rio de Janeiro, 1990.

[VV91] Paulo A. S. Veloso e Sheila R. M. Veloso. Some remarks on conservative extensions: A socratic dialogue. *Bulletim of EATCS (European Association for Theoretical Computer Science)*, 43:189–198, 1991.