Um Estudo Axiomático Comparativo entre Modelos de Bancos de Dados

Mauricio Pereira de Oliveira

TESE APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Marcelo Finger**

Durante a elaboração deste trabalho o autor recebeu apoio financeiro do CNPq

– São Paulo, agosto de 2000 –

Um Estudo Axiomático Comparativo entre Modelos de Bancos de Dados

Este exemplar corresponde à redação
final da tese devidamente corrigida
e defendida por Mauricio Pereira de Oliveira
e aprovada pela comissão julgadora.

São Paulo, 2 de agosto de 2000.

Banca examinadora:

- Prof. Dr. Marcelo Finger (IME-USP)
- Profa. Dra. Ana Cristina Vieira de Melo (IME-USP)
- Prof. Dr. Ruy Guerra de Queiroz (UFPE)

Para Veronica,
Therézio e Milena.

Resumo

Há 30 anos foi apresentado o mais popular modelo de bancos de dados utilizado até hoje, o modelo relacional. Com o passar dos anos, no entanto, ele se tornou insuficiente para expressar uma série de novos conceitos decorrentes de novas demandas, motivando assim o surgimento de diversos outros modelos.

Desses novos modelos surgiu a necessidade de se poder compará-los, de preferência de uma maneira formal, buscando saber se um é mais ou menos expressivo que outro, ou se a combinação de conceitos de um dado modelo pode ou não expressar um determinado conceito em outros modelos.

Neste sentido, o presente trabalho tem por objetivo propor e testar um *ambiente* formal onde diversos modelos de bancos de dados possam ser comparados relativamente às suas expressividades. Ele se divide em duas partes. A primeira parte se destina a estruturar o problema da comparação entre modelos de dados e a desenvolver um ambiente apropriado a essas comparações. A segunda parte se destina a testar o ambiente recém desenvolvido. Para isso realizamos um estudo de caso, onde analisamos de que forma e sob quais condições um modelo relacional pode exprimir os conceitos estruturais de um modelo orientado a objetos.

Abstract

The relational model was first presented thirty years ago. Despite being the most popular database model and still largely used, it has become insufficient to express a string of new concepts demanded by new types of applications, motivating the appearance of many data models.

Then, there appeared the need for comparing those new models, preferably in a formal way, in order to discover whether one model is more or less expressive than another, or whether a combination of concepts of a given model can express a particular concept in other models.

This work aims at proposing and testing a formal environment where several database models could be compared concerning their expressivenesses. It is divided into two parts. The first one is devoted to structure the matter of database model comparisons and develop an appropriate environment for such comparisons. The second part is devoted to test the environment developed in part one. With this aim, a case study was performed and it was analysed in which way and under which conditions a relational model can express the structural concepts of an object-oriented model.

Sumário

Capítulo 1

Introdução

Há 30 anos foi apresentado o mais popular modelo de bancos de dados utilizado até hoje, o modelo relacional, proposto por Codd em [CODD70]. Com o passar dos anos, no entanto, o modelo relacional tornou-se insuficiente para expressar uma série de novos conceitos, decorrentes de novas demandas, dando assim a motivação necessária para o surgimento de diversos outros modelos.

Esses modelos, em geral, evoluiram a partir de diferentes paradigmas, tais como paradigmas orientados a objetos, dedutivos, relacionais, ativos, dedutivos orientados a objetos, dentre outros. E foram descritos também através de diferentes representacões, com diferentes graus de formalização, tais como representações gráficas, descritivas, lógicas, etc.

Com esses modelos surgiu também a necessidade de se poder compará-los, de preferência de uma maneira formal, buscando saber se um modelo é mais ou menos expressivo que outro, ou se a combinação de conceitos de um dado modelo pode ou não expressar um determinado conceito em outros modelos. Este tipo de análise é muito importante quando precisamos especificar a troca de informações entre bancos de dados heterogêneos ou selecionar, dentre diversos modelos, um mais adequado a alguma aplicação particular.

Neste sentido, o presente trabalho tem por objetivo propôr e testar um *ambiente* onde diversos modelos de bancos de dados possam ser comparados relativamente às suas expressividades. Este ambiente deve apresentar, entre suas características mais importantes, um embasamento teórico bem sólido e definido, e possibilitar algum tipo de automatização e metodologia que facilitem o processo de eventuais comparações.

Assim este trabalho se divide em duas partes.

A primeira parte se destina a estruturar o problema da comparação entre modelos de dados e a desenvolver um ambiente apropriado a essas comparações. Inicialmente apresentamos diversos modelos de dados, com diferentes paradigmas e representações, visando fornecer assim os ingredientes necessários à motivação deste estudo. A seguir fornecemos toda uma teoria que embasa e justifica esta comparação, e propomos também ferramentas e metodologias para que a prática dessa comparação possa ser realizada. E no decorrer dessas explicações,

alguns problemas e desafios de cunho tanto teórico quanto prático são também discutidos.

A segunda parte se destina a colocar em prática o exposto na parte anterior, ou seja, testar o ambiente recém desenvolvido. Assim, tentamos realizar um estudo de caso onde dois modelos de dados, um orientado a objetos e outro relacional, são apresentados e posteriormente comparados. Tentamos, mais especificamente, estudar de que forma e sob quais condições um modelo relacional pode exprimir os conceitos estruturais de um modelo orientado a objetos.

Parte I

Estrutura do problema

Capítulo 2

Modelos de Dados

2.1 Importância

Qual a importância de um modelo de dados? Nas palavras de W. Kim, em [KIM90], "um modelo de dados determina uma Linguagem de Banco de Dados, que por sua vez determina a implementação de um Sistema de Banco de Dados". Qualquer sistema de banco de dados é projetado sobre algum modelo de dados, da mesma forma que uma casa é construída sobre o seu alicerce. O sucesso ou fracasso de tal sistema é diretamente dependente da expressividade e eficiência do modelo de dados sobre o qual foi projetado. Daí a importância de se buscar formalizações teóricas matematicamente bem fundadas para estes modelos.

2.2 Composição

Três componentes de um modelo de dados podem ser distinguidos[1]:

2.2.1 Componente Estrutural

Este componente se caracteriza pela modelagem dos objetos ou entidades do universo real em termos de alguma estrutura de dados e por vários relacionamentos entre esses objetos. Aqui devem ser incluídos conceitos como agregação, agrupamento, identidade de objetos, objetos complexos, relacionamentos ISA, herança, etc. Este componente constitui o foco principal de nosso estudo.

2.2.2 Componente de Manipulação de Dados

Constitui a Linguagem de Manipulação de Dados, que se subdivide em :

Linguagem de Definição de Dados. Permite a especificação do esquema de um banco de dados.

[1] Como descrito em [ABITE87].

Linguagem de Gerenciamento de Dados. Permite a consulta, criação, atualização e exclusão de instâncias individuais de um banco de dados.

Linguagem de Controle de Dados. Permite a especificação de transações e a autorização e gerenciamento de métodos de acesso.

2.2.3 Componente de Especificação de Integridade

Permite a especificação de restrições de integridade, com o objetivo de restringir as instâncias permitidas pelo esquema de banco de dados.

2.2.4 Outros Componentes

Na verdade outros componentes ainda podem ser imaginados como fazendo parte de um modelo de dados, tal como o controle de versões, um aspecto que tem ganhado muita importância nos últimos anos.

2.3 Modelos Estudados

Nas seções a seguir expomos alguns modelos encontrados na literatura. A idéia aqui não é de se fazer uma resenha do que existe, mas sim de se dar um testemunho da diversidade de modelos que podem ser encontrados. Além disso, foram descritos apenas aqueles modelos cuja formalização teórica pudesse ser extraída, seja de forma implícita ou explícita, na sua totalidade ou parcialmente. Não descrevemos aqui os modelos de rede e hierárquico, por serem modelos antigos, obsoletos e pouco usados atualmente.

Dessa forma, apresentamos o modelo relacional, alguns modelos semânticos, o modelo orientado a objetos, o modelo dedutivo, o modelo de banco de dados ativo e, finalmente, o modelo dedutivo orientado a objetos.

2.4 Modelos Relacionais

O modelo relacional pode não ter sido o primeiro modelo de banco de dados criado, mas com certeza foi o mais importante, simples e popular até hoje concebido. Foi o primeiro a ter uma forte formalização teórica e serviu como base principal para o desenvolvimento da teoria de banco de dados.

Este modelo foi inicialmente proposto por E.F.Codd em [CODD70]. Neste trabalho o autor define o Modelo de Dados Relacional através de um componente estrutural (o *Relational View of Data*), onde os dados são organizados em tuplas pertencentes a relações, e de um componente de manipulação de dados, que seria a Álgebra Relacional, totalmente fundada no Cálculo de Predicados de Primeira Ordem.

Esta uniformidade do modelo relacional, no entanto, é verdadeira apenas no que se refere aos seus conceitos mais básicos. Atualmente, a definição do que

seja o modelo relacional, em seu sentido mais amplo, é um tanto inexata, como mostra o trecho a seguir extraído de [ABITE95]:

> *O termo modelo relacional é, hoje, bastante vago. Como introduzido no artigo seminal de Codd, este termo se refere a um modelo de dados específico com relações como estrutura de dados, uma álgebra para especificar consultas, e nenhum mecanismo para expressar atualizações ou restrições. Artigos subseqüentes de Codd introduziram uma segunda linguagem de consulta baseada no cálculo de predicados da lógica de primeira ordem, mostrando isso ser equivalente à álgebra, e introduziu as primeiras restrições de integridade para o modelo relacional, ou seja, as dependências funcionais. Daí em diante, pesquisadores em sistemas de bancos de dados implementaram linguagens baseadas na álgebra e cálculo, estenderam para incluir operadores de atualização, aritméticos e de agregação. Além disso, uma rica teoria sobre restrições de integridade emergiu. O termo modelo relacional, portanto, se refere a uma grande classe de modelos que tem as relações como estrutura de dados e que incorporam alguns ou todos os recursos de consulta, atualização ou de restrições de integridade.*

Com esta conceituação, muitos dos modelos de BDs ativos e dedutivos podem ser considerados como sendo pertencentes à classe dos relacionais. No entanto, por sua importância isolada e características particulares, os mesmos são descritos em seções próprias, mais adiante, neste trabalho.

2.5 Modelos Semânticos

Modelos semânticos são modelos conceituais que visam fornecer uma representação formal, gráfica e de alto nível, abstrata, mas bem definida, para aquilo que se quer modelar. São muito úteis na macroespecificação de dados, e também no estudo teórico de diferentes paradigmas. Talvez o mais conhecido deles seja o modelo entidade-relacionamento.

2.5.1 Modelo Entidade-Relacionamento

O Modelo E-R (ver figura 2.1) foi originalmente criado por Chen em [CHEN76], e posteriormente melhorado de forma a estender sua expressividade. Baseia-se, como o próprio nome diz, em entidades detentoras de atributos, possivelmente ligadas umas às outras por diferentes tipos de relacionamentos.

Este modelo, como descrito em [BATI91], foi durante muitos anos amplamente utilizado, principalmente na análise do Modelo Relacional, pois ambas teorias possuem entre si uma tradução quase que direta. Para a análise OO, o modelo E-R pôde retratar, com alguma profundidade, aspectos importantes tais como Classes e Hierarquia, mas foi incapaz de modelar, por exemplo, Identidade de objetos e Objetos Complexos, dois conceitos fundamentais deste paradigma.

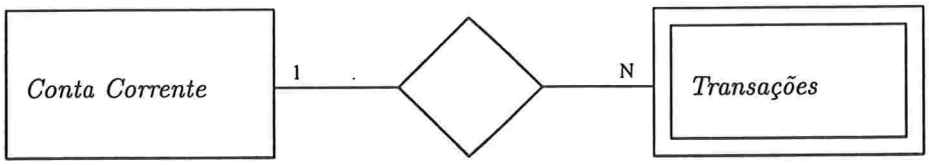


Figura 2.1: Exemplo de Diagrama E-R com a entidade *Conta Corrente*, a entidade fraca *Transações*, e um relacionamento *1-N*.

2.5.2 IFO - Um Modelo de Banco de Dados Formal

Tentando suprir as deficiências levantadas em modelos anteriores, principalmente no relacional, Abiteboul e Hull propuseram em [ABITE87] um novo modelo chamado IFO (ver figura 2.2) que englobava, relativo ao aspecto estrutural, os modelos Relacional e E-R, entre outros. Além disso, forneceu uma nova estrutura para a caracterização de Objetos Complexos, através de construtores de objetos provenientes da Teoria dos Conjuntos, ou seja, operadores que possibilitavam a construção de objetos a partir de outros objetos, a saber, o produto cartesiano, o operador potência e o operador união. Um esquema de banco de dados neste modelo é uma árvore direcionada, cujas folhas representam dados e cujos nós internos, ou operadores, representam conexões entre objetos que podem ser dados e/ou outros nós.

No que se refere a Classes e Hierarquia, no entanto, este modelo é bastante pobre, não conseguindo dar um tratamento tão estruturado quanto ao dado aos Objetos Complexos. Na verdade, neste aspecto, o IFO não acrescentou muito ao que já era encontrado no Modelo E-R, na versão de [BATI91].

Além disso, não foram apresentados tratamentos para outros componentes dos Modelos de Dados, principalmente no que se refere ao componente de Linguagem de manipulação de Dados.

Apesar disso tudo, o IFO tornou-se um modelo importante, servindo como base para diversos trabalhos posteriores, entre eles o LDM, visto a seguir.

2.5.3 LDM - The Logical Data Model

O Modelo de Dados Lógico (ver figura 2.3), ou simplesmente LDM, foi descrito por Kuper e Vardi em [KUPER93]. Essencialmente, este modelo herda a estrutura do IFO, mas, diferentemente deste, que só admitia estruturas em forma de árvore, o LDM passa a permitir estruturas em forma de grafos direcionados com ciclos e insere o conceito de Identidade de Objetos. Esta situação permitiu ao LDM, segundo os autores, a modelar outros paradigmas de dados, tais como os Modelos de Rede e Hierárquico, e também muitos aspectos do modelo OO.

Mas talvez a maior contribuição do LDM tenha sido a proposição de uma Linguagem de Manipulação de Dados incorporada ao Modelo. Além disso, os

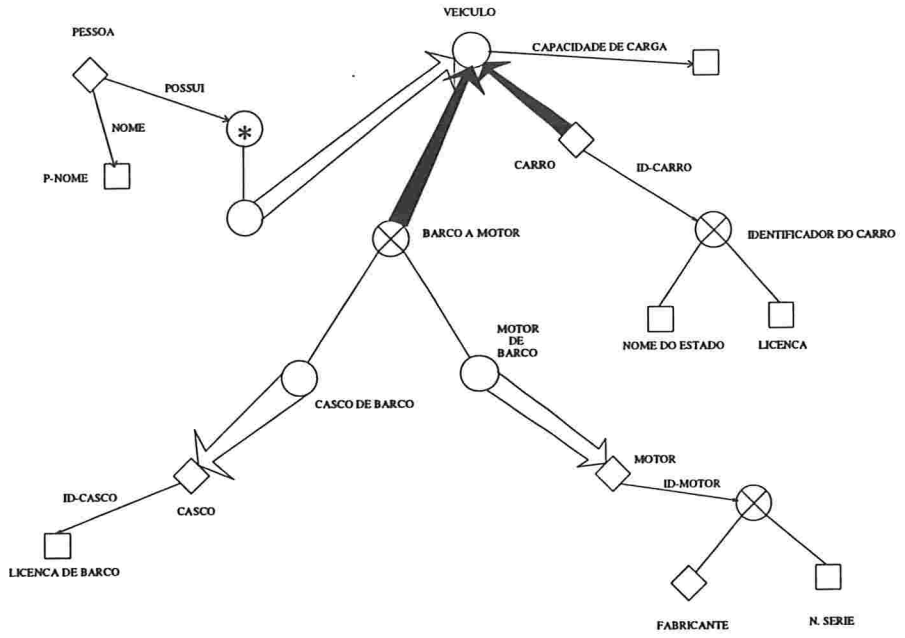


Figura 2.2: Exemplo de Grafo IFO de veículos.

autores propõem uma álgebra de manipulação de dados que é provada ser equivalente, ou completa, em relação à Linguagem de Manipulação de Dados.

Ainda assim, mesmo com todos esses importantes avanços, o LDM não chega a tratar outros aspectos do paradigma OO tais como Classes, Hierarquia, etc.

2.6 Modelos Orientados a Objetos

No caso dos Modelos Orientados a Objetos, o consenso em torno de uma única formalização teórica não existe, apesar da existência de muitas propostas (ver [ATKIN89]). Mesmo assim, muitos sistemas têm sido desenvolvidos *ad-hoc*, e provavelmente, e talvez infelizmente, o modelo de dados aceito será aquele resultante de uma verdadeira seleção natural "darwiniana" influenciada por interesses econômicos, sistemas lançados primeiramente no mercado, propaganda, etc.

Esta falta de padronização é principalmente devida a não haver uma definição clara do que seja um sistema orientado a objetos, do qual os sistemas de Bancos de Dados OO herdam seus conceitos mais básicos. Isso faz com que cada modelo criado tenha características distintas visando um domínio particular de aplicação, e não a generalidade.

Mas esta indefinição tem ainda uma outra razão de ser. A formalização teórica buscada hoje deve ser extremamente poderosa, no sentido de ser capaz de expressar *todos* os componentes de um modelo de dados, como descrito an-

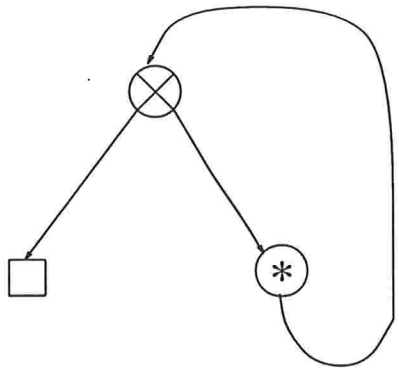


Figura 2.3: Exemplo de esquema LDM correspondente ao modelo hierárquico. Notar a presença de um ciclo no grafo.

teriormente, e, além disso, poder expressar aspectos dinâmicos dos bancos de dados (tais como atualizações de dados, etc.), poder unificar numa única linguagem tanto a linguagem de programação como a linguagem de manipulação de dados (unificação esta que, de longe, não ocorre no Modelo Relacional), e até mesmo poder gerenciar *informações implícitas* (ou seja, dados dedutíveis logicamente de outros dados), como encontrado nos sistemas de bancos de dados dedutivos orientados a objetos (DOOD). A formalização do Modelo Relacional é, neste sentido, muitíssimo restrita, e portanto, muito mais fácil de ser concebida.

2.6.1 O *Manifesto* dos Sistemas de Bancos de Dados OO

E nesta verdadeira "fogueira das vaidades" em que se transformaram os modelos OO, alguns importantes pesquisadores resolveram se unir e definir quais características deveria possuir um Sistema de Banco de Dados OO, e dessas pesquisas originaram-se os chamados *Manifestos*. Desses *Manifestos* destaca-se o de [ATKIN89]. Nele, as características obrigatórias que um tal sistema deveria prover são:

1. Objetos Complexos.
2. Identidade de Objetos.
3. Encapsulamento de Objetos.
4. Tipos ou Classes.
5. Hierarquia de Tipos ou Classes.
6. *Overriding, Overloading* e *Late Binding.*
7. Completude Computacional.

8. Extensibilidade.

9. Características comuns a um DBMS: persistência, gerenciamento de memória secundária, concorrência, recuperação de dados, facilidade de consulta *ad hoc*.

2.6.2 Modelos OO Existentes

Neste contexto, muitos modelos foram propostos, com diferentes enfoques e diferentes paradigmas. A seguir damos uma breve descrição de um desses modelos.

The *Core Data Model* - O modelo de dados do sistema ORION

Saindo um pouco dos modelos puramente teóricos para aqueles mais aplicados, temos o modelo que embasou o Sistema ORION, como descrito em [KIM90]. Usaremos a sigla CDM para identificá-lo.

Apesar do CDM ter sido descrito de uma maneira um tanto informal na referência indicada, este modelo apresentou um tratamento bastante interessante quanto ao aspecto de Classes e Hierarquia. O CDM trata as classes como se fossem elas próprias objetos, com métodos e atributos próprios. A Hierarquia de Classes, por sua vez, está representada por relacionamentos tipo *ISA* entre esses objetos. Além disso, as classes, por serem objetos que devem ser criados ou destruídos por métodos, devem pertencer a outras classes, as metaclasses. Analogamente as metaclasses devem pertencer a metametaclasses, e assim sucessivamente.

Esta estratificação do conceito de classe garantiu um tratamento uniforme no manuseio das mensagens e na definição de objetos, além de possibilitar situações tais como alterar a definição de uma classe em tempo de execução ou fazer consultas sobre quais classes possuem determinados atributos, por exemplo. (Na verdade, o ORION foi implementado possibilitando apenas um nível simplificado de metaclasse).

Além disso, o CDM modelou outros conceitos, como Hierarquia de Composição de Classes, análogo aos Objetos Complexos e ortogonal à Hierarquia de Classes, e Identidade de Objetos, entre outros.

2.7 Modelos de Banco de Dados Ativos

Segundo [ZANIO97] os bancos de dados ativos oferecem uma facilidade, fortemente integrada ao software de sistema de banco de dados, para criar e executar regras de produção. Essas regras seguem o paradigma *Evento-Condição-Ação*. Elas, de forma autônoma, reagem aos eventos ocorridos nos dados e executam uma reação sempre que a avaliação de uma determinada condição produz um valor verdadeiro.

Quando um banco de dados exibe um comportamento reativo, uma grande fração da semântica que é normalmente codificada dentro das aplicações pode ser expressa por meio de regras ativas, dando às aplicações de banco de dados

uma nova dimensão de independência, chamada *independência de conhecimento* : as aplicações são liberadas da necessidade de expressarem o conhecimento sobre o processamento reativo, que é codificado na forma de regras ativas. Conseqüentemente, o conhecimento é codificado uma única vez no esquema e é automaticamente compartilhado por todos os usuários; a modificação do conhecimento é gerenciada mudando-se o conteúdo das regras e não as aplicações.

Entre os produtos/protótipos conhecidos detentores desse paradigma, podemos citar os bancos de dados relacionais *Starburst, Oracle, DB2,* e o banco de dados OO *Chimera.*

Do ponto de vista teórico não existe um modelo universal para os bancos de dados ativos, apesar da existência de muitos protótipos de pesquisa e de produtos comerciais.

As características fundamentais comuns aos modelos de bancos de dados ativos são:

Evento: Os eventos típicos considerados pelas regras ativas, ou seja, inserções, exclusões e atualizações de dados, são primitivas para as mudanças de estado do banco de dados; vários sistemas podem também monitorar consultas aos dados, alguns sistemas monitoram eventos relacionados ao tempo (Ex: às 5:00 PM, toda sexta-feira), e alguns sistemas monitoram eventos externos, causados por aplicações.

Condição: A condição é um predicado do banco de dados ou uma consulta. Uma condição retorna um valor verdade: uma consulta é interpretada como uma condição verdadeira se ela contém pelo menos uma tupla, e falsa, caso contrário.

Ação: A ação é um programa arbitrário de manipulação de dados; pode incluir comandos de transação (tal como o comando *Rollback*) ou comandos de manipulação de regras (tal como o ativamento ou desativamento de regras ativas ou de grupos de regras ativas), e algumas vezes pode ativar procedimentos definidos externamente.

2.8 Modelos de Banco de Dados Dedutivos

Da mesma forma que o artigo [CODD70] serviu como pedra fundamental para os bancos de dados relacionais, tendo como base o paradigma teórico de modelos da LPO, o artigo [REITER84] teve o mesmo papel em relação aos bancos de dados dedutivos, por sua vez se baseando no paradigma teórico de prova da LPO.

Do ponto de vista da teoria de modelos, o autor define um banco de dados como sendo a tripla (R, I, IC). $R = (A, W)$ é uma linguagem relacional, definida como sendo uma LPO sem símbolos funcionais, onde A é um alfabeto contendo um número finito de constantes e predicados, entre os quais existe um predicado de igualdade e possivelmente alguns predicados unários chamados *tipos simples*; e W é um conjunto de fbf's (fórmulas bem formadas) construídas usando-se

os símbolos de A . $I = (D, K, E)$ é uma interpretação relacional, onde D é um domínio, K é uma bijeção entre as constantes de A e os elementos de D (portanto D deve ser finito), e E é a extensão do predicado de igualdade para o domínio D. Finalmente IC é um conjunto de fórmulas chamadas *restrições de integridade*. Essas restrições são consideradas satisfeitas sse I for um modelo para IC.

Uma vez definido o que seria um banco de dados em termos da teoria de modelos, o autor reconstrói logicamente este banco de dados em termos da teoria de prova. Ou seja, da tripla (R, I, IC), obtém-se a tripla (R, T, IC) . R e IC são definidos como acima. T, por sua vez, é uma teoria relacional, que é um conjunto de fórmulas constituído por[2]:

- **Fatos atômicos**

  As tabelas da interpretação I são transformadas em fatos atômicos (Ex: homem(marcos) e mulher(ana)).

- **AFD - Axioma do Fecho de Domínio**

  Em geral, se I tem domínio $c_1, c_2, ..., c_n$, então o AFD p/ I é:
  $$\forall x((x \approx c_1) \vee (x \approx c_2) \vee ... \vee (x \approx c_n)).$$

- **AUN - Axioma da Unicidade dos Nomes**

  Para cada par de constantes distintas c, c', temos:
  $$(c \not\approx c').$$

- **AC - Axiomas da Completação**

  Para cada predicado P de R, sua extensão $E(P)$, em I, é um conjunto finito de tuplas, possivelmente vazio. É necessário um axioma denotando que essas tuplas são os únicos elementos de $E(P)$.

  Este grupo de axiomas deve ser entendido mais facilmente através de um exemplo. Digamos que os únicos fatos para o predicado *pai* sejam:

  pai(antonio,jose).
  pai(jose,carlos).
  pai(carlos,maria).

  Então, o axioma de completude CA para o predicado *pai* é:

  $$\forall xy(pai(x, y) \rightarrow ((x \approx antonio) \wedge (y \approx jose)) \vee$$
  $$((x \approx jose) \wedge (y \approx carlos)) \vee ((x \approx carlos) \wedge (y \approx maria))).$$

- **Axiomas da Igualdade**

  Para especificar a reflexividade, comutatividade e transitividade da igualdade, e para especificar o princípio da substituição de termos iguais de Leibnitz.

[2]Assumir, durante a presente seção, que os elementos do domínio de I têm os mesmos nomes das constantes correspondentes do alfabeto de R.

Portanto, com a teoria T montada dessa forma, o seguinte teorema pode ser provado:

Teorema 2.8.1 Se T é uma teoria relacional de R, então T tem um um único modelo I, o qual é uma interpretação relacional para R. E, se I é uma interpretação relacional para R então existe uma teoria relacional T de R tal que I é o único modelo de T.

■

E o corolário a seguir pode então ser obtido:

Corolário 2.8.1 Suponha que T seja uma teoria relacional de uma linguagem relacional R, e que I seja um modelo de T. Então para cada fbf w de R, w é verdadeira em I sse $T \vdash w$.

■

Do corolário acima deduzimos que uma consulta ao banco de dados (R, T, IC) pode ser representada por uma fórmula a ser provada pela teoria T.

O autor ainda aponta vantagens do paradigma de prova sobre o de modelo, tais como nos tratamentos de informação incompleta em bancos de dados, informação disjuntiva, semântica de valores nulos e outros.

2.9 Modelos de Bancos de Dados Dedutivos Orientados a Objetos

Segundo [KIFER95], uma das forças impulsionadoras por trás do interesse em linguagens OO em bancos de dados é a promessa que elas mostram em superar o problema do desacoplamento entre linguagens de programação para escrever aplicações e linguagens para extração de dados. Ao mesmo tempo, um enfoque diferente, dedutivo, ganhou enorme popularidade. Desde que Lógica pode ser usada como um formalismo computacional e também como uma linguagem de especificação de dados, defensores do paradigma de programação dedutiva têm argumentado que este enfoque também supera o problema do desacoplamento. Entretanto, na presente forma, ambos os enfoques têm falhas. Um dos principais problemas com o enfoque OO é a falta de uma semântica lógica que, tradicionalmente, tem exercido um papel importante em linguagens de programação de bancos de dados. Por outro lado, bancos de dados dedutivos se baseiam em um modelo de dados plano, e não suportam estruturação de dados. Portanto, a combinação dos dois paradigmas poderia trazer grandes benefícios. Modelos que buscam esta combinação são denominados modelos dedutivos OO.

2.9.1 F-logic

Frame Logic, ou abreviadamente F-logic, como apresentado em [KIFER95] segue a linha dos bancos de dados dedutivos orientados a objetos (DOOD's). F-logic constitui um formalismo lógico de primeira ordem, cujo objetivo maior é

suplantar o *Impedance mismatch* (desacoplamento), como descrito acima, e além disso unir, sob uma estrutura comum, o paradigma de orientação a objetos com o paradigma dedutivo de programação.

F-logic possui uma teoria de prova composta de 13 regras de inferência e 1 axioma. O modelo de dados, apesar de não estar explicitamente descrito nesta linguagem, está implícito nessas regras de inferência.

Assim seja, por exemplo, um conceito comumente encontrado no componente estrutural da maioria dos modelos OO:

> "Se P é um elemento da classe Q, e Q é uma subclasse da classe R, então P é também um elemento da classe R."

Tal conceito encontra-se implícito, em F-logic, na seguinte regra de inferência:

Inclusão de Subclasse: Segundo a sintaxe de F-logic, $A : B$ significa que A é uma instância da classe B e $B :: C$ significa que B é uma subclasse da classe C. Assim seja $W = (P : Q) \vee C$ e $W' = (Q' :: R') \vee C'$ e θ um m.g.u. de Q e Q', então:

De W e W' deriva $\theta((P : R') \vee C \vee C')$.

Finalmente, vale notar que F-logic poderia ser um sério candidato a um *framework* comum para o estudo comparativo de diferentes modelos de dados.

2.9.2 O modelo do sistema *ROCK & ROLL*

ROCK & ROLL é um DOOD desenvolvido a partir de axiomatizações dos conceitos OO, como exposto na tese [FERNAN95]. Na verdade esta tese serviu como base para os trabalhos de axiomatização de modelos da presente pesquisa, como mostrado mais adiante.

Neste trabalho, o autor propõe um tratamento axiomático ao modelamento de Bancos de Dados Dedutivos Orientados a Objetos (DOOD's). O modelo do sistema ROCK & ROLL é muito semelhante à F-logic, porém, em vez de se concentrar nas regras de inferência, procurou enriquecer o conjunto de axiomas. Isto confere flexibilidade ao sistema, uma vez que melhorias ou alterações no mesmo podem ser efetuadas simplesmente modificando-se o conjunto de axiomas, sem precisar alterar a *máquina dedutiva*, composta pelas regras de inferência. Uma outra vantagem é que esta *máquina dedutiva*, segundo o autor, pode ser a mesma utilizada nos tradicionais Bancos de Dados Dedutivos Relacionais, reaproveitando todo o trabalho de otimização feito no desenvolvimento desses sistemas.

Resumidamente, o modelo do sistema ROCK & ROLL, ou abreviadamente modelo RR, constitui um conjunto de 36 axiomas, divididos em dois grupos.

Um primeiro grupo de 10 axiomas tem o objetivo de gerar informações implícitas. Esta geração é realizada a partir do conteúdo explícito e através de conseqüências lógicas. Por exemplo:

Axioma 2. Inclusão de classes é uma relação transitiva. Formalmente:

$$\forall x \forall y \forall z (is\_a(x,y) \wedge is\_a(y,z) \Rightarrow is\_a(x,z)).$$

E um segundo grupo, de 26 axiomas, tem o objetivo de restringir quais dessas informações geradas são válidas. Por exemplo:

Axioma 15. Inclusão de classes é uma relação antissimétrica. Formalmente:

$$\forall x \forall y (is\_a(x,y) \wedge is\_a(y,x) \Rightarrow x \approx y).$$

2.10 Outros Modelos de Dados

Outros modelos de dados merecem também ser mencionados. A saber, as relações N1NF (*Non First Normal Form*) e as *V-relations*, como descrito em [PARED87], e também os modelos dos sistemas comerciais O_2 e $ODMG$.

2.11 O nosso trabalho

A grande diversidade de modelos existentes, como visto até o momento, a sensação de incerteza e imprecisão quanto à escolha de um melhor modelo ou quanto a comparações meramente genéricas e intuitivas encontradas na bibliografia em geral, e as interessantes idéias levantadas em [FERNAN95] e [REITER84] motivaram a presente proposta de trabalho.

Devido a esta diversidade, pareceu ser interessante dispor de uma maneira estruturada e formal que permitisse comparar um modelo com outro, no sentido de buscar deficiências e virtudes de cada um. Este mesmo desejo é evidenciado no trecho a seguir de [REITER84]:

> *Há uma necessidade dentro da comunidade de bancos de dados de se estender o modelo relacional para acomodar mais conhecimento do mundo real, e muitas das extensões requeridas não podem ser acomodadas no paradigma teórico dos bancos de dados relacionais. Uma enorme variedade de propostas têm sido apresentadas em resposta a essa necessidade. No entanto, há dois problemas com esse grande número de propostas:*
>
> 1. *Como alguém pode começar a compará-los? Em que senso formal poderia alguém conclamar que duas tais propostas têm os mesmos* **poderes** *de representação, ou que uma é uma generalização da outra. A maioria dessas propostas envolvem diferentes linguagens de representação e diferentes (e em alguns casos não especificadas) semânticas, fazendo o mapeamento entre elas virtualmente impossível. ...*

Assim, a proposta deste trabalho se resume a:

1. Prover um ambiente formal onde comparações entre modelos de dados possam ser efetuadas.
2. Definir dentro deste ambiente uma teoria das comparações, onde os vários aspectos deste problema possam ser discutidos formalmente.
3. Discutir alguns problemas teóricos e práticos e as limitações que eventualmente existam na realização de tais comparações.
4. Discutir e utilizar ferramentas existentes para realizar essas comparações.
5. Propor uma metodologia para aumentar o sucesso e a eficiência dessas comparações.
6. Realizar um estudo de caso, fazendo efetivamente uma comparação entre dois modelos de dados existentes, utilizando-se os recursos e idéias discutidos até então.

Para providenciar o ambiente formal pedido no primeiro item acima, seria necessário existir um *framework* comum com poder suficiente para expressar esses modelos. Elegemos então a Lógica de Primeira Ordem (LPO) como sendo este *framework*. A suficiência da LPO no tratamento de modelos é defendida por diversos autores, tais como [KIFER95] e [FERNAN95].

Mas talvez a melhor justificativa para esta escolha se encontre também em [REITER84]:

> *Meu propósito é indicar como um* **framework** *lógico pode aliviar esses problemas. Especificamente, eu devo argumentar que os tipos de conhecimento do mundo real que esses modelos de dados estendidos tentam capturar têm representações naturais como fórmulas de primeira ordem. Segue que tais modelos de dados não lógicos podem ser equivalentemente formalizados por convenientes classes restritas de teorias de primeira ordem... Posto que este mapeamento de um modelo de dados não lógico para um modelo lógico pode ser feito, nós desfrutaríamos de um número de benefícios imediatos:*
>
> 1. *A semântica do modelo de dados não lógico seria precisamente definida por sua tradução lógica.*
> 2. *Dois modelos de dados não lógicos poderiam ser comparados (com respeito a seus* **poderes** *de representação), comparando-se suas traduções.*

Dessa forma, uma vez escolhida a LPO como base formal, o próximo passo é traduzir, para esta base, os modelos de interesse, para que possamos posteriormente compará-los. A este processo de tradução damos o nome de **axiomatização**. Iniciamos então a axiomatização desses modelos. Para cada modelo

estudado, levantamos os seus conceitos fundamentais ou semânticos, e escrevemos esses conceitos em forma de um conjunto Φ de fórmulas[3] de LPO.

Por exemplo, para o conceito OO já visto anteriormente:

> "Se P é um elemento da classe Q, e Q é uma subclasse da classe R, então P é também um elemento da classe R."

Geramos uma axiomatização sua em LPO:

$$\forall p \forall q \forall r(elemento(p,q) \wedge subclasse(q,r) \Rightarrow elemento(p,r)).$$

Para que alguma comparação possa ser realizada, é preciso uma axiomatização de, ao menos, dois modelos arbitrários. Sejam, por ora, esses dois modelos identificados por A e B. A axiomatização de A gera um conjunto Φ_A de sentenças, ou axiomas, em LPO. Analogamente, B gera um conjunto Φ_B de axiomas. Diversas comparações podem ser então logicamente analisadas de posse dessas axiomatizações, seja manualmente ou utilizando-se, por exemplo, um provador de teoremas. As propostas abaixo refletem alguns desses resultados que eventualmente possam ser obtidos. Para isso assumimos aqui a seguinte notação: $\Phi_A \vdash \Phi_B$ se e somente se $\Phi_A \vdash \alpha$, ou Φ_A *deduz* α para todo $\alpha \in \Phi_B$[4].

1. **Se Φ_A for um conjunto inconsistente**

   Φ_A é inconsistente se e somente se for insatisfazível. Podemos verificar isso, simplesmente aplicando o provador de teoremas ao conjunto Φ_A e obtendo uma resposta afirmativa para a refutação. Se este for o caso, significa que não existe nenhuma instância de banco de dados que se *encaixe* neste modelo de dados. Ou seja, concluímos que o modelo A foi erroneamente concebido. Analogamente, o mesmo teste pode ser feito para Φ_B.

2. **Se $\Phi_A \vdash \alpha$, onde $\alpha \in \Phi_B$**

   Isto significa que o conceito semântico expresso por α no modelo B está também implicitamente expresso no modelo A.

3. **Se $\Phi_A \vdash \Phi_B$**

   Significa que tudo que é dedutível de Φ_B é dedutível de Φ_A, ou seja, a expressividade de A engloba a de B.

4. **Se $\Phi_A \vdash \Delta$, onde $\Delta \subset \Phi_B$**

   Significa que tudo que é dedutível de Δ é dedutível de Φ_A, ou seja, a expressividade de A engloba parte da de B. Aqui, um interessante estudo pode ser feito. Vamos procurar um conjunto Φ' de fórmulas tal que $\{\Phi' \cup \Phi_A\}$ seja consistente e $\{\Phi' \cup \Phi_A\} \vdash \Phi_B$. Intuitivamente, o significado de Φ' é de um conjunto de conceitos faltantes ao modelo A para que este englobe a expressividade de B.

[3] Sendo mais preciso, são sentenças, ou seja, fórmulas sem variáveis livres.

[4] Na verdade, como Φ_A e Φ_B pertencem a linguagens distintas, é improvável que $\Phi_A \vdash \alpha$ para algum $\alpha \in \Phi_B$. Precisamos antes de uma extensão conservativa por definições Φ'_A de Φ_A, tal que $\Phi'_A \vdash \alpha$. Isso será explicado e justificado nos próxinos capítulos. Por ora, procurando favorecer o entendimento em detrimento da precisão, diremos $\Phi_A \vdash \alpha$ em lugar de $\Phi'_A \vdash \alpha$.

5. **Se $\Phi_C \supseteq \Phi_A \cup \Phi_B$**

   Neste caso, poderíamos pensar em Φ_C como a base de um modelo que englobaria os modelos A e B. Mas para afirmar isto, primeiro teríamos que testar se Φ_C é ou não um conjunto consistente.

6. **Se $\Phi_A \vdash \Phi_B$ e $\Phi_B \vdash \Phi_A$**

   Este é um caso interessante, pois toda instância de Banco de Dados que tem uma representação num dos modelos, possui forçosamente uma representação correspondente no outro. Isso significa, por exemplo, que é possível construir um tradutor entre esses modelos, que teria grande aplicação na área de Bancos de Dados Heterogêneos.

7. **Outros Resultados**

   Muitos outros testes poderiam ser realizados. Por exemplo, combinando alguns dos testes expostos acima. Assim, dados Φ_A e Φ_B quaisquer, poderíamos achar superconjuntos deles Φ'_A e Φ'_B, respectivamente, de forma que esses novos conjuntos fossem logicamente equivalentes, como no teste anterior. O significado disso seria: dados dois modelos A e B, que conceitos extras devemos acrescentar a cada um deles para que os modelos resultantes sejam semanticamente equivalentes.

2.12 Comentários Finais

Nos próximos capítulos desta primeira parte discutiremos com grande profundidade diversos aspectos teóricos e práticos envolvidos nesta comparação formal entre modelos. Apresentaremos um provador de teoremas chamado *Otter*, sua sintaxe e seu funcionamento. E discutiremos também algumas propostas metodológicas que emergiram durante a execução das comparações.

Nos capítulos da segunda parte tentaremos realizar efetivamente uma comparação entre dois modelos de dados, a saber, um modelo relacional e um modelo OO.

Capítulo 3

Uma disciplina para axiomatização

3.1 Introdução

Este capítulo discute alguns aspectos teóricos e metodológicos envolvidos no processo de axiomatização de modelos de BDs, fornecendo a base necessária para as axiomatizações realizadas na segunda parte deste trabalho.

3.2 Lógica Utilizada

Vimos no capítulo anterior que uma escolha para um *framework* adequado para o presente estudo é a lógica de primeira ordem, ou LPO. Porém alguma discussão sobre esta escolha se faz ainda necessária. Gostaríamos que esta lógica fosse suficiente para modelar se não todos, pelo menos uma classe grande e representativa de modelos de dados. Vimos que segundo Reiter em [REITER84], LPO teria esta suficiência. Além disso, para o estudo de BDs mais modernos, tais como bancos de dados dedutivos OO, ou DOODs, apesar da adequação de LPO ter sido posta em questão por muitos autores — argumentando que uma lógica de segunda ordem ou uma lógica modal seriam requeridas — a maioria dos trabalhos em DOOD adotam LPO, mais ou menos explícita e diretamente[1].

Um outro fator muito importante que favorece a utilização de LPO é a sua solidez, no sentido que nenhuma outra lógica foi tão bem estudada e possui mecanismos de prova tão sedimentados e otimizados. Não daremos aqui explicações sobre LPO, que podem ser obtidas em bons textos matemáticos, tais como [BELL77] ou [ENDER72].

Não vamos precisar no entanto, para o fim deste estudo, de toda a expressividade de LPO. De fato, utilizaremos uma LPO que tenha, em sua linguagem, o predicado da igualdade juntamente com os axiomas da igualdade, e que permita

[1] ver [FERNAN95]

somente símbolos funcionais com aridade igual a zero, ou seja, possua apenas *constantes* como símbolos funcionais. O não uso de símbolos funcionais, além das constantes, é comum nas formalizações de BDs, uma vez que, em geral, não se utiliza operadores sobre dados nesses modelos. Denominaremos de LPO' uma LPO com essas restrições.

Um último fator fundamental a ser comentado é a complexidade dos procedimentos de prova em LPO'. LPO', assim como LPO, é um sistema lógico *semidecidível*. Portanto, para o problema de decidir a satisfatibilidade ou não de um conjunto de fórmulas em LPO', o melhor que podemos conseguir é um algoritmo que tenha como entrada tal conjunto e que sempre responda *sim* em tempo finito, caso o mesmo seja insatisfatível, ou que responda *não* em tempo finito ou simplesmente rode indefinidamente sem jamais responder, caso contrário. É de algum consolo que para o presente trabalho precisemos provar apenas a insatisfatibilidade de fórmulas em LPO', como será visto mais adiante. Porém, mesmo para as entradas *decidíveis*, este algoritmo tem complexidade com tempo não polinomial, ou seja, seu tempo de resposta cresce pelo menos exponencialmente com o tamanho do conjunto de entrada.

Este estado de coisas causou grande impacto nas provas realizadas nesta dissertação. Algumas delas demoraram até alguns dias para fornecerem uma resposta. Isso era inaceitável, uma vez que a comparação entre modelos mostrou ser um processo iterativo e não modular, significando que as fórmulas a serem provadas eram constantemente alteradas. Por isso algumas metodologias bastante simples foram desenvolvidas para que viabilizassem a realização das mesmas num tempo ao menos razoável. Elas são expostas no capítulo 5.

3.3 Axiomatização

O passo seguinte é a realização da **axiomatização** de primeira ordem, i.e., escolhido um modelo particular, seus conceitos semânticos são levantados, se possível de alguma forma metodológica e formal, e em seguida traduzidos para fórmulas de LPO, (aqui, de LPO'), que por sua vez determinam uma teoria de primeira ordem.

Iremos abaixo apresentar um método que facilitará a axiomatização. Porém não podemos omitir que a axiomatização é um processo que demanda uma boa dose de subjetividade e intuição. Primeiro pela própria dificuldade de se extrair, de uma forma completa, todos os conceitos fundamentais do objeto que se deseja axiomatizar, muitas vezes devido a própria falta de uma formalização prévia adequada deste objeto. Segundo porque, mesmo com o conceito à mão, é fácil cometer o erro de, quando de sua tradução para uma fórmula lógica, esta não exprimir exatamente a semântica daquele. E terceiro porque, matematicamente falando, à medida que a axiomatização origina um conjunto, supostamente finito, de sentenças lógicas que representam uma certa teoria e que podem existir infinitos conjuntos de fórmulas que representam uma mesma teoria, podem então existir infinitas axiomatizações retratando esta teoria. Com alguma sorte podemos conseguir extrair um desses conjuntos corretamente.

Com isso em mente, utilizaremos uma metodologia que, se não gera deterministicamente uma axiomatização, serve ao menos como linha mestra para orientar este processo. Na verdade esta metodologia pode ser dividida em duas fases complementares:

1. Utilização do **método de Bourbaki** com o objetivo de caracterizar modelos de BDs como estruturas matemáticas dentro de uma teoria de conjuntos.

2. Tradução dessas estruturas matemáticas para uma lógica de primeira ordem, processo conhecido como **reconstrução lógica**.

Durante a axiomatização real dos modelos na segunda parte deste trabalho, a distinção entre essas duas fases existirá apenas implicitamente. Isso porque a aplicação desses processos é bastante direta e simples, não necessitando uma apresentação explícita de cada fase da axiomatização. Assim, durante as axiomatizações, apresentaremos um conceito extraído da semântica do modelo seguido imediatamente de sua formulação lógica. No entanto, a compreensão detalhada de cada uma dessas fases é fundamental para o sucesso da axiomatização.

Vale salientar novamente que nem todas as axiomatizações possíveis são igualmente adequadas para o objetivo buscado nesta dissertação. Somente axiomatizações de primeira ordem sem símbolos funcionais são consideradas como sendo alternativas apropriadas. Por outro lado, a axiomatização deve representar o mais próximo possível a visão geral do significado dos conceitos do modelo de dados em estudo. No caso do modelo OO, esta visão pode ser obtida em [ATKIN89]. Um exemplo de aplicação do primeiro princípio acima: o uso de construtores de tipo é restrito. Assim, não é permitido aninhar construtores em expressões de tipo tais como *X é um conjunto de conjuntos de Y*. Em vez disso, deve-se nomear o(s) tipo(s) intermediário(s), quebrando assim uma expressão de tipo aninhada em duas ou mais expressões de tipo normais[2], p. ex., *X é um conjunto de Zs e Z é um conjunto de Ys*. Isto é necessário para garantir que esta linguagem permaneça de primeira ordem e sem símbolos funcionais.

3.3.1 Caracterização dos MBDs como estruturas matemáticas

A caracterização de modelos de BDs, abreviadamente MBDs, como estruturas matemáticas definidas dentro de uma teoria de conjuntos é obtida seguindo-se, em seu sentido mais amplo, a versão *Bourbakiana* do método axiomático para a caracterização de uma estrutura matemática, tal como descrita em [FERNAN95] ou em [BOURB68].

A motivação de usar o estilo particular de formulação matemática brevemente descrita abaixo é o fato dela permitir que uma *teoria lógica de primeira ordem* seja derivada a partir da *teoria matemática* num número de passos reduzido e simplificado. Esta derivação é conhecida como *reconstrução lógica* na literatura de BDs.

[2] *flat* em inglês.

3.3.2 Definição de estruturas matemáticas

Uma *estrutura matemática* ([BOURB68]) é uma descrição unificada de objetos matemáticos usando somente os conceitos *conjunto* e *relação*. Mais exatamente, uma estrutura matemática $\mathcal{E}$ é um par $\langle \Sigma, \Gamma \rangle$ consistindo de um conjunto de símbolos, ou *assinatura*, $\Sigma = \{\Xi_1, \ldots, \Xi_m, \xi_1, \ldots, \xi_n\}$, que compreende *símbolos de conjuntos base* $(\Xi_1, \ldots, \Xi_m)$ e *símbolos de relações básicas* $(\xi_1, \ldots, \xi_n)$, mais um axioma Γ sobre Σ.

Adotando Bourbaki em estilo e terminologia, os passos do *método axiomático* são expostos a seguir. Descrevemos paralelamente um exemplo onde concebemos uma estrutura matemática para um modelo OO muito simplificado que apenas retrata o conceito de inclusão entre duas ou mais classes.

Passo 1

Os *conjuntos base principais e auxiliares* são listados exaustivamente, i.e., o conjunto $(\Xi_1, \ldots, \Xi_m)$ é inicialmente definido.

Exemplo. Seja $\mathbf{C}$ $(= \Xi_1)$ um conjunto base, representando o conjunto de **classes.**

Passo 2

Uma lista exaustiva de *conjuntos potência de produtos cartesianos* é exibida. Cada elemento do qual é formado somente a partir dos conjuntos base. Elementos desses conjuntos potência são nomeados exaustivamente como *relações básicas* de interesse $(\xi_1, \ldots, \xi_n)$, o que dá origem a uma *caracterização típica.*

Mais formalmente, a conjunção de todas as fórmulas $\mathcal{F}_i$ definidas por

$$\mathcal{F}_i : \xi_i \in \mathcal{P}(\Xi_{i_1} \times \cdots \times \Xi_{i_k}) \text{ onde } \begin{cases} k > 0 \\ 1 \leq i \leq n \\ \Xi_{i_1}, \ldots, \Xi_{i_k} \in \{\Xi_1, \ldots, \Xi_m\} \\ \mathcal{P} = \text{conjunto potência,} \end{cases}$$

é chamada de *caracterização típica* de $\mathcal{E}$, representada por $\mathcal{T}(\mathcal{E})$.

Exemplo. Seja $\mathcal{F}_1$ dada por:

$$\mathcal{F}_1 : \textbf{inclusão } (= \xi_1) \in \mathcal{P}(\mathbf{C} \times \mathbf{C}).$$

Passo 3

Definimos um *sistema axiomático*, i.e., um conjunto de axiomas que é produzido para impor estrutura aos conjuntos base e às relações básicas. O propósito crucial do sistema axiomático é definir as propriedades básicas desses conjuntos e relações. Este conjunto de axiomas representa as propriedades descritas como p proposições sobre Σ denotadas por

$$P_i|\Xi_1,\ldots,\Xi_m,\xi_1,\ldots,\xi_n| \text{ onde } \begin{cases} p \geq 0 \\ 0 \leq i \leq p. \end{cases}$$

Então $\mathcal{S}(\mathcal{E}) = \{P_1,\ldots,P_p\}$, onde $\mathcal{S}(\mathcal{E})$ é o *sistema axiomático* de $\mathcal{E}$. A conjunção da caracterização típica $\mathcal{T}(\mathcal{E})$ com a conjunção dos elementos de $\mathcal{S}(\mathcal{E})$ é o *axioma* Γ.

Exemplo. Seja a propriedade $P_1|\mathbf{C}, \mathbf{inclusão}|$, descrita por:

> *A relação* **inclusão** *é transitiva sobre* **C**. Assim, p. ex., se $\langle \mathtt{a}, \mathtt{b}\rangle \in$ **inclusão** e $\langle \mathtt{b}, \mathtt{c}\rangle \in$ **inclusão**, então $\langle \mathtt{a}, \mathtt{c}\rangle \in$ **inclusão**.

Passo 4

Uma *prova* é dada que o conjunto de modelos da estrutura matemática assim caracterizada é não vazio e, dessa forma $\mathcal{E}$ é consistente.

Com este propósito define-se $A_1,\ldots,A_m$ como sendo os *conjuntos base* (no lugar de $\Xi_1,\ldots,\Xi_m$). As *relações básicas* (ou *conceitos básicos*) $\alpha_1,\ldots,\alpha_n$ (no lugar de $\xi_1,\ldots,\xi_n$) são dados como elementos dos conjuntos que são finitamente gerados a partir de $A_1,\ldots,A_m$, i.e., elementos dos conjuntos obtidos através de um número finito de aplicações das operações de conjuntos para formar um produto Cartesiano e um conjunto potência sobre $A_1,\ldots,A_m$. As *propriedades básicas* ($P_1|\Xi_1,\ldots,\Xi_m,\xi_1,\ldots,\xi_n|$, ..., $P_n|\Xi_1,\ldots,\Xi_m,\xi_1,\ldots,\xi_n|$) são dadas como proposições $P_i|A_1,\ldots,A_m,\ \alpha_1,\ldots,\alpha_n|$. Tomadas juntas, os *conjuntos base*, as *relações básicas*, e as *propriedades básicas* determinam um *sistema matemático*.

Quando se substitui os símbolos $\Xi_1,\ldots,\Xi_m$ e $\xi_1,\ldots,\xi_n$ pelos conjuntos $A_1,\ldots,A_m$ e pelas relações $\alpha_1,\ldots,\alpha_n$, respectivamente, e Γ é avaliado como sendo verdadeiro, $(A_1,\ldots,A_m,\alpha_1,\ldots,\alpha_n)$ torna-se um *sistema matemático com a estrutura matemática* $\mathcal{E} = \langle\Sigma,\Gamma\rangle$, ou, equivalentemente, um *modelo da estrutura* $\mathcal{E}$.

Se ao menos um desses modelos existir, então o conjunto de todos os modelos de $\mathcal{E}$ é não vazio, e portanto $\mathcal{E}$ é consistente. Assim a prova buscada se resume à procura de um desses modelos. Este modelo, se encontrado, é também conhecido como *testemunha*.

Exemplo. Uma testemunha para a estrutura definida nos passos anteriores poderia ser:

> Seja $\mathbf{C} = \{\mathtt{cão}, \mathtt{animal}, \mathtt{ser\_vivo}\}$.
> Seja **inclusão** $= \{\langle \mathtt{cão}, \mathtt{animal}\rangle, \langle \mathtt{animal}, \mathtt{ser\_vivo}\rangle, \langle \mathtt{cão}, \mathtt{ser\_vivo}\rangle\}$.
> Vemos que **C** e **inclusão** satisfazem a propriedade P_1.

3.3.3 Reconstrução lógica de MBD para TMBD

A caracterização de um MBD como uma estrutura matemática já seria por si só suficiente como uma formalização teórica do modelo. Porém a necessidade de reconstruí-lo logicamente vem do fato de um sistema lógico diferir de uma estrutura matemática exatamente no ponto em que traz o requerimento de formalização até a noção de *prova*. Portanto as *estruturas matemáticas* obtidas na seção anterior são reconstruídas agora como *sistemas lógicos*, i.e., estruturas matemáticas equipadas com uma noção de prova formal.

Como exemplo de analogia, o conteúdo técnico do método Bourbakiano é análogo, em intenção, ao trabalho original de Codd sobre os modelos relacionais em [CODD70], o qual lançou os bancos de dados como assunto de estudo teórico. Entretanto o trabalho de Codd é apenas uma construção teórica de modelo e não suporta qualquer noção de dedução formal. Esta transformação para uma teoria lógica ocorreria apenas com Reiter em [REITER84], transformação esta que ficaria a partir de então conhecida na literatura de BDs como *reconstrução lógica*.

3.3.4 Caracterização de TMBD a partir de MBD

Esta seção descreve como um MBD pode ser representado como um sistema axiomático em LPO', culminando na caracterização de MBD como uma teoria de modelo de banco de dados, ou TMBD, i.e., um conjunto de axiomas numa linguagem de primeira ordem, ou, equivalentemente, como um cálculo de primeira ordem.

Definição de uma linguagem para TMBD

Vamos chamar a linguagem de TMBD como LMBD (linguagem de MBD). Neste contexto, os símbolos próprios de LMBD são determinados da seguinte forma:

Símbolos funcionais. Os axiomas de TMBD determinam o conjunto C de constantes individuais (símbolos funcionais 0-ários) de LMBD. O conjunto de constantes individuais contém aquelas que realmente ocorrem nos axiomas que descrevem TMBD e somente aquelas. O conjunto de símbolos funcionais de LMBD com aridade maior que zero é vazio.

Predicados. Da mesma forma, os axiomas de TMBD determinam o conjunto P de símbolos predicados de LMBD.

Variáveis. Seja V um conjunto contável infinito de variáveis individuais.

Então, o **alfabeto (não lógico)** de LMBD é dado pela tripla:

$$\mathsf{A} = \langle \mathsf{C}, \mathsf{P}, \mathsf{V} \rangle.$$

O conjunto de termos de LMBD sobre A é dada pela união:

$$\mathsf{T} = \mathsf{C} \cup \mathsf{V}.$$

O processo de reconstrução lógica é, nesta altura, um tanto direto. As expressões de $\mathcal{E}$ são agora traduzidas para fórmulas de primeira ordem que, por sua vez, definirão TMBD. Dàmos aqui uma noção deste processo através de uma continuação do exemplo da seção anterior:

Exemplo de reconstrução lógica. Sejam $x, y, z, \ldots$ variáveis sobre T. Sejam $\hat{x}, \hat{y}, \hat{z}, \ldots$ variáveis sobre $\mathcal{E}$. Seja $\equiv$ um símbolo que denote a relação *é-substituido-por* entre expressões em $\mathcal{E}$ e expressões em LMBD. A reconstrução lógica de $\mathcal{E}$ em TMBD é realizada através das substituições descritas abaixo:

$$\begin{aligned}
\hat{x} \in \mathbf{C} &\equiv classe(x). \\
\langle \hat{x}, \hat{y} \rangle \in \mathbf{inclusão} &\equiv is_a(x, y). \\
\mathbf{inclusão} \in \mathcal{P}(\mathbf{C} \times \mathbf{C}) &\equiv \forall xy(is_a(x, y) \Rightarrow classe(x) \wedge classe(y)). \\
P_1|\mathbf{C}, \mathbf{inclusão}| &\equiv \forall xyz(is_a(x, y) \wedge is_a(y, z) \Rightarrow is_a(x, z)). \\
\ldots &\equiv \ldots
\end{aligned}$$

Estruturação da axiomatização de MBD

A teoria TMBD gerada a partir da aplicação da reconstrução lógica sobre $\mathcal{E}$ é representada por um conjunto de axiomas que, no caso da axiomatização de modelos de BDs, pode ser dividido em subconjuntos que auxiliam em sua melhor estruturação e são fundamentais para seu entendimento e verificação de sua correção.

Assim, o conjunto de *axiomas de modelo de banco de dados* que caracterizam uma TMBD divide-se nos seguintes grupos e subgrupos de axiomas:

1. Um conjunto dos axiomas de representação $RAx = EqAx$:

   (a) $EqAx$ é o conjunto dos *axiomas da igualdade* que definem o predicado da igualdade.

   Os axiomas de representação são necessários na visão teórica de prova de MBDs, independente do modelo de dados associado. Em $\mathcal{E}$ representam propriedades da teoria que embasa $\mathcal{E}$, ou seja, da teoria de conjuntos.

2. Um conjunto $MBDAx$ de axiomas que determinam um particular MBD. $MBDAx = T \cup C \cup E$ é o conjunto dos *axiomas de modelagem*, os quais são invariantes com respeito aos domínios de aplicação, com os seguintes subconjuntos disjuntos:

   (a) T é o conjunto de *axiomas de tipo* e caracteriza uma TMBD *tipada*. Os elementos em T são implicitamente conjugados e T é referenciado como o *axioma de tipo* das TMBDs. Em $\mathcal{E}$, T formaliza a caracterização típica $\mathcal{T}(\mathcal{E})$.

(b) C é o conjunto de *axiomas de completação*. $C \cup T$ caracteriza uma TMBD *completa*. Para entendermos o papel dos axiomas de C, é mais fácil pensarmos na sua função em bancos de dados dedutivos, onde são responsáveis pela geração, através de conseqüência lógica, de toda a informação implícita positiva a partir da informação explícita declarada. Os elementos em C são implicitamente conjugados e C é referenciado como o *axioma de completação* (ou *positivo*) das TMBDs. Exemplo: Num modelo OO, inclusão de classes é uma relação transitiva. Formalmente:

$$\forall x \forall y \forall z (is\_a(x,y) \wedge is\_a(y,z) \rightarrow is\_a(x,z)).$$

(c) E é o conjunto de *axiomas de exceção*. $MBDAx = T \cup C \cup E$ caracteriza uma TMBD *(restrita)*. Analogamente a C, em BDs dedutivos, os axiomas de E têm a função de gerar através de conseqüência lógica toda a informação implícita negativa a partir das informações explícitas declaradas e implícitas positivas geradas por C, ou seja, eles restringem qual informação explícita ou implícita é realmente válida. Os elementos em E são implicitamente conjugados e E é referenciado como o *axioma de exceção* (ou *negativo*) das TMBDs. Exemplo: Num modelo OO, inclusão de classes é uma relação antissimétrica. Formalmente:

$$\forall x \forall y (is\_a(x,y) \wedge is\_a(y,x) \rightarrow x \approx y).$$

Em $\mathcal{E}$, $C \cup E$ formaliza o sistema axiomático $\mathcal{S}(\mathcal{E})$ e a conjunção dos axiomas de $MBDAx$ representa o axioma Γ. Em resumo os axiomas de $MBDAx$ expressam os conceitos semânticos do modelo estudado.

Axiomatização de um BD

Apesar de não ser o objetivo do presente trabalho, vale a pena neste momento abrirmos um pequeno parêntese e analisarmos como ficaria esta divisão de axiomas caso estivéssemos axiomatizando não apenas um modelo de dados, mas um banco de dados. Ou seja, ao axiomatizarmos um banco de dados, devemos axiomatizar não só o modelo de dados que representa este BD, mas também o esquema e a instância associados ao BD. Nesta situação, dois outros grupos de axiomas compostos apenas por fatos devem ser acrescentados aos expostos anteriormente:

1. Ao conjunto RAx devemos incluir o conjunto dos *axiomas de particularização* PAx, com $PAx = AC \cup AFD \cup AUN$[3], onde:

   (a) AC é o conjunto dos *axiomas da completação*.

[3] Ver seção 2.8 para maiores detalhes.

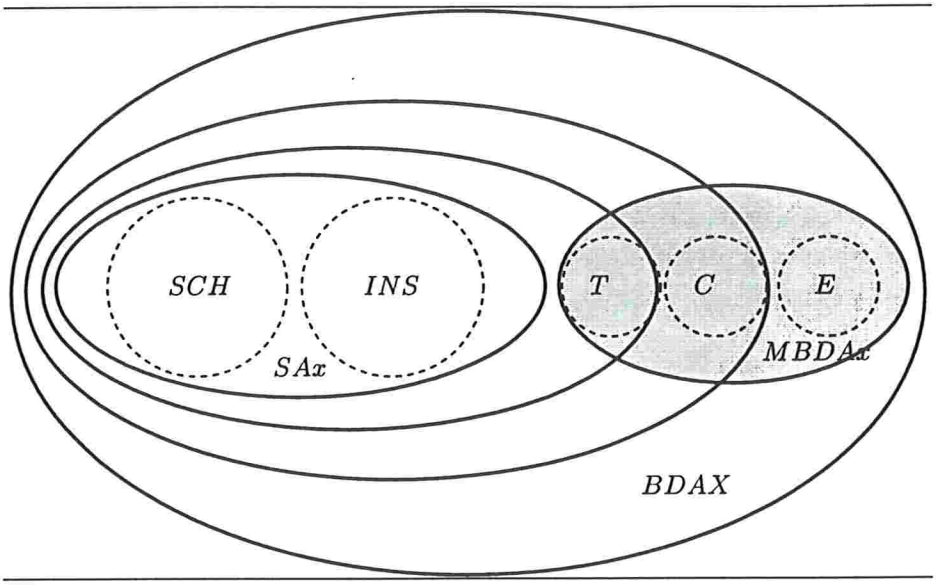


Figura 3.1: Diagrama de Venn representando os axiomas de $MBDAx$ e $BDAx$.

(b) AFD é o *axioma do fecho de domínio.*

(c) AUN é o conjunto dos *axiomas de unicidade dos nomes.*

2. Devemos unir $MBDAx$ com SAx para obtermos o conjunto maior $BDAx$ de axiomas de BDs, que determina uma teoria de BD particular. $SAx = SCH \cup INS$ é o conjunto dos axiomas de estado que capturam o esquema de um domínio de aplicação e uma de suas instâncias válidas (i.e., um particular estado do domínio). Todos os axiomas em SAx são fatos atômicos na linguagem de BD. Os axiomas em SAx associam uma teoria de BD a um domínio de aplicação particular num estado também particular. Em $\mathcal{E}$, SAx representa um modelo de $\mathcal{E}$, ou um sistema matemático com a estrutura matemática $\mathcal{E}$.

3.3.5 Escopo deste trabalho relativo às axiomatizações

Para o estudo de caso realizado na segunda parte deste trabalho, duas atividades descritas neste capítulo não serão realizadas, por simplificação e conveniência. A saber,

1. Não nos preocuparemos neste trabalho com a realização do passo 4 da aplicação do método de Bourbaki (ver seção 3.3.2), ou seja, com a busca de uma testemunha para verificar se a TMBD obtida no processo de axiomatização de cada MBD é realmente uma teoria consistente.

2. Ao gerarmos os axiomas de $MBDAx$ não faremos uma separação explícita entre C e E, pois apesar dessa divisão ser conceitualmente importante durante o processo de axiomatização, ela se torna incoveniente como uma forma de apresentação dos axiomas durante as comparações, onde uma classificação dos axiomas por cada predicado nos é mais favorável.

Capítulo 4

Teoria das comparações

4.1 Introdução

Este capítulo apresenta a adaptação às necessidades deste trabalho da parte do estudo da lógica de primeira ordem conhecida como *Interpretação entre teorias* tal como exposto em [ENDER72], que servirá de justificativa teórica principal para todas as provas entre teorias de BDs realizadas nos capítulos subseqüentes.

A *Interpretação entre teorias* como descrita em [ENDER72] corresponde às seções 4.4.1, 4.4.2 e 4.4.3. Da seção 4.4.4 até o final do capítulo, o material apresentado foi todo desenvolvido pelo autor deste trabalho.

4.2 Objetivo

O objetivo primordial deste capítulo é descrito resumidamente a seguir.

Vimos no capítulo anterior que, dados dois modelos A e B podemos axiomatizá-los obtendo dois conjuntos de fórmulas ϕ_A e ϕ_B, que por sua vez determinam duas teorias T_A e T_B, respectivamente. Vamos provar que, para descobrirmos se o modelo B é pelo menos tão expressivo quanto o modelo A, basta construirmos um conjunto *especial* de fórmulas Δ, tal que a relação abaixo seja verdadeira.

$$\boxed{\sigma_i \in \phi_A \Rightarrow \{\phi_B \cup \Delta\} \vdash \sigma_i, \text{ para todo } \sigma_i \in \phi_A.}$$

Este resultado é equivalente à fórmula (4.44) provada ao final do capítulo. As fórmulas em Δ definem todos os símbolos extralógicos (símbolos predicados e funcionais) da linguagem de T_A a partir de símbolos lógicos e extralógicos da linguagem de T_B.

Assim, p. ex., se T_A for uma teoria dos números naturais contendo em sua linguagem o predicado unário *natural* e T_B uma teoria dos números inteiros contendo em sua linguagem o predicado unário *inteiro*, o predicado binário $\geq$

e a constante *zero*, uma das definições em Δ pode ser dada por

$$\forall x(natural(x) \leftrightarrow inteiro(x) \wedge (x \geq zero)).$$

4.3 Algumas convenções

Antes de iniciarmos, vamos convencionar algumas notações e relembrar alguns conceitos de lógica de primeira ordem:

1. Para interpretar termos e fórmulas numa linguagem L, é necessário fixar uma estrutura $\mathcal{B}$ para L consistindo dos seguintes ingredientes:

   (a) Uma classe não vazia $|\mathcal{B}|$ chamada *universo de discurso* (ou, abreviadamente, *universo* ou *domínio*) de $\mathcal{B}$. Os membros de $|\mathcal{B}|$ são chamados *indivíduos*.

   (b) Um mapeamento que associa a cada símbolo funcional f de L uma operação $f^{\mathcal{B}}$ sobre $|\mathcal{B}|$, tal que se f é um símbolo funcional n-ário, $f^{\mathcal{B}}$ é uma operação n-ária sobre $|\mathcal{B}|$. Em particular, se c é uma constante, então $c^{\mathcal{B}}$ é um indivíduo.

   (c) Um mapeamento que associa a cada símbolo predicado P de L uma relação $P^{\mathcal{B}}$ sobre $|\mathcal{B}|$, tal que se P é um símbolo predicado n-ário, $P^{\mathcal{B}}$ é uma relação n-ária sobre $|\mathcal{B}|$.

2. Uma fórmula que não tem variáveis livres é chamada de *sentença*. Se uma estrutura $\mathcal{B}$ satisfaz uma sentença α, ou $\mathcal{B} \models \alpha$, ou ainda $\models_{\mathcal{B}} \alpha$, então dizemos que $\mathcal{B}$ é um *modelo* de α. Se $\mathcal{B} \models \phi$ para todo ϕ num conjunto Φ de sentenças, dizemos que $\mathcal{B}$ é um *modelo* de Φ.

3. T é uma teoria numa linguagem L se e somente se T for um conjunto de sentenças tal que para qualquer sentença σ em L,

$$T \models \sigma \Rightarrow \sigma \in T.$$

4. Para toda fórmula φ apresentada nas seções seguintes, subentende-se que as variáveis livres que eventualmente apareçam em φ estejam entre as variáveis $v_1, v_2, \ldots, v_n$, pertencentes, por sua vez, a um conjunto contável infinito V de variáveis.

5. Quanto à substituição de termos, convencionaremos que

$$\varphi(t) = \varphi_t^{v_1},$$
$$\varphi(t_1, t_2) = (\varphi_{t_1}^{v_1})_{t_2}^{v_2},$$

onde $\varphi_t^{v_1}$ é a fórmula obtida a partir da fórmula φ ao se substituir a variável

v_1, sempre que ela aparecer livre em φ, pelo termo t. Se nós escrevermos $\varphi(x)$, não estamos muito preocupados se v_1 é ou não substituível por x em φ. Se não for, nós fazemos $\varphi(x)$ como sendo $\psi_x^{v_1}$, onde ψ é uma variante alfabética apropriada de φ. Com essa convenção, é verdade, p. ex., que $\varphi = \varphi(v_1) = \varphi(v_1, v_2)$.

6. Definimos $\mathrm{Cn}(\Sigma)$, onde Σ é um conjunto de sentenças de primeira ordem, como sendo o conjunto de todas as sentenças logicamente satisfeitas por Σ, ou

$$\mathrm{Cn}(\Sigma) = \{\sigma : \Sigma \models \sigma\}.$$

É imediato que $\mathrm{Cn}(\Sigma)$ é uma teoria, e $\mathrm{Cn}(\Sigma)$ é a teoria *representada* por Σ.

7. Seja uma estrutura fixa $\mathcal{B}$. Então para os elementos $a_1, \ldots, a_n$ de $|\mathcal{B}|$,

$$\models_{\mathcal{B}} \varphi[\![a_1, \ldots, a_n]\!],$$

significa que $\mathcal{B}$ satisfaz φ com alguma (e daí com qualquer) função $s : V \to |\mathcal{B}|$, onde $s(v_i) = a_i, 1 \leq i \leq n$.

8. Seja uma estrutura fixa $\mathcal{B}$ e uma função $s : V \to |\mathcal{B}|$. Então

$$\models_{\mathcal{B}} \varphi[s],$$

significa que $\mathcal{B}$ satisfaz φ com o mapeamento s. A função $s(y|b)$ é exatamente igual a s, com exceção de que para a variável y ela assume o valor b. Assim a relação abaixo é verdadeira

$$\models_{\mathcal{B}} \varphi[\![a_1, \ldots, a_n]\!] \Leftrightarrow \models_{\mathcal{B}} \varphi[s(v_1|a_1)\ldots(v_n|a_n)], \text{ para todo } s : V \to |\mathcal{B}|.$$

4.4 Interpretação entre teorias

4.4.1 Introdução

Em alguns casos uma teoria T_1 pode ser mostrada como sendo tão poderosa como uma outra teoria T_0. Este é o caso se as duas teorias estão na mesma linguagem e $T_0 \subseteq T_1$. Mas mesmo se as teorias estão em linguagens diferentes, deve existir um modo de traduzir de uma linguagem para outra de uma maneira que membros de T_0 sejam traduzidos como membros de T_1.

4.4.2 Interpretações

É possível para uma teoria ser tão poderosa quanto uma outra teoria em uma outra linguagem. Por exemplo, a teoria dos conjuntos axiomática é pelo menos tão poderosa quanto a teoria dos números naturais com zero e sucessor,i.e., $(\mathbb{N}, 0, S)$. Qualquer sentença nesta última linguagem pode ser traduzida de uma maneira natural numa sentença da teoria de conjuntos. Se a sentença original for verdadeira em $(\mathbb{N}, 0, S)$, então sua tradução será uma conseqüência dos axiomas da teoria de conjuntos.

Seja então L_0 uma linguagem e T_1 uma teoria numa linguagem L_1 com igualdade, tal que L_1 pode ou não ser igual a L_0.

Definição 4.4.1 Uma *interpretação* π de L_0 para T_1 é uma função sobre o conjunto de parâmetros (símbolos lógicos e extralógicos) de L_0, tal que

1. π associa ao símbolo lógico $\forall$ uma fórmula $\pi_\forall$ de L_1 na qual no máximo v_1 ocorre livre, e tal que

$$T_1 \models \exists v_1 \pi_\forall. \tag{4.1}$$

   A idéia é que em qualquer modelo de T_1, a fórmula $\pi_\forall$ deveria definir um conjunto não vazio para ser usado como um universo de uma estrutura de L_0.

2. π associa a cada símbolo predicado n-ário P uma fórmula π_P de L_1 na qual no máximo as variáveis $v_1, \ldots, v_n$ ocorrem livres.

3. π associa a cada símbolo funcional n-ário f uma fórmula π_f de L_1 na qual no máximo as variáveis $v_1, \ldots, v_n, v_{n+1}$ ocorrem livres, e tal que para todo $\vec{v}$ no conjunto definido por $\pi_\forall$, há um único x tal que $\pi_f(\vec{v}, x)$ e x também pertence ao conjunto definido por $\pi_\forall$. Formalmente,

$$\begin{aligned} T_1 \models & \forall v_1 \ldots \forall v_n (\pi_\forall(v_1) \to \cdots \to \pi_\forall(v_n) \\ & \to \exists x (\pi_\forall(x) \wedge \forall v_{n+1} (\pi_f(v_1, \ldots, v_{n+1}) \leftrightarrow v_{n+1} \approx x))). \end{aligned} \tag{4.2}$$

   A idéia é assegurar que em qualquer modelo de T_1, π_f defina uma função no universo definido por $\pi_\forall$. No caso de um símbolo constante c, nós temos $n = 0$ e (4.2) torna-se

$$T_1 \models \exists x (\pi_\forall(x) \wedge \forall v_1 (\pi_c(v_1) \leftrightarrow v_1 \approx x)). \tag{4.3}$$

   Em outras palavras, π_c define um conjunto de um único elemento que também pertence ao conjunto definido por $\pi_\forall$

■

Um caso particular importante para nossa presente análise é quando $L_0 \subseteq L_1$, onde podemos usar a interpretação identidade π, para a qual

$$\begin{aligned} \pi_\forall &= (v_1 \approx v_1). \\ \pi_P &= Pv_1 \dots v_n. \\ \pi_f &= fv_1 \dots v_n \approx v_{n+1}. \\ \pi_c &= (c \approx v_1). \end{aligned} \tag{4.4}$$

Nesta situação, as condições (4.1) e (4.2) são satisfeitas para qualquer teoria T_1.

Definição 4.4.2 (Definição de $^\pi\mathcal{B}$ e $\pi^{-1}[T_1]$) Seja agora π uma interpretação e $\mathcal{B}$ um modelo de T_1. Há uma maneira natural de se extrair de $\mathcal{B}$ uma estrutura $^\pi\mathcal{B}$ para L_0. Assim, sejam

$$\begin{aligned} |{}^\pi\mathcal{B}| &= \text{o conjunto definido em } \mathcal{B} \text{ por } \pi_\forall. \\ P^{{}^\pi\mathcal{B}} &= \text{a relação definida em } \mathcal{B} \text{ por } \pi_P, \text{ restrita a } |{}^\pi\mathcal{B}|, \\ &\quad \text{i.e., } \langle d_1, \dots, d_m \rangle \in P^{{}^\pi\mathcal{B}} \text{ sse } \models_{\mathcal{B}} \pi_P[\![d_1, \dots, d_m]\!] \\ &\quad \text{e } \{d_1, \dots, d_m\} \subseteq |{}^\pi\mathcal{B}|. \\ f^{{}^\pi\mathcal{B}}(a_1, \dots, a_n) &= \text{o único } b \text{ tal que } \models_{\mathcal{B}} \pi_f[\![a_1, \dots, a_n, b]\!], \\ &\quad \text{onde } a_1, \dots, a_n \text{ estão em } |{}^\pi\mathcal{B}|. \end{aligned} \tag{4.5}$$

Pela condição (4.1) na definição de interpretações, $|{}^\pi\mathcal{B}| \neq \emptyset$. E, pela condição (4.2), a definição de $f^{{}^\pi\mathcal{B}}$ faz sentido, i.e., existe um único b que satisfaz a condição acima. Daí $^\pi\mathcal{B}$ é realmente uma estrutura para a linguagem L_0.

Definimos o conjunto $\pi^{-1}[T_1]$ de sentenças em L_0 pela equação

$$\begin{aligned} \pi^{-1}[T_1] = \{\sigma : \sigma &\text{ é uma sentença em } L_0 \text{ verdadeira em toda estrutura } {}^\pi\mathcal{B} \\ &\text{obtida de um modelo } \mathcal{B} \text{ de } T_1\}. \end{aligned} \tag{4.6}$$

$\pi^{-1}[T_1]$ assim definido é uma teoria e é satisfatível sse T_1 é satisfatível.

■

Tradução sintática

Dada uma fórmula φ em L_0, podemos achar uma fórmula φ^π em L_1 que corresponda, em algum sentido, exatamente a φ. Nós então definimos agora φ^π através de recursão em φ .

Primeiro, consideramos uma fórmula atômica α em L_0. A definição de α^π pode ser estabelecida pelo uso de recursão no número de locais nos quais símbolos

funcionais ocorrem em α. Se este número é zero, então α é $Px_1 \ldots x_n$ e α^π é $\pi_P(x_1, \ldots, x_n)$. Caso contrário, tome o local mais à direita no qual um símbolo funcional g ocorre. Se g é um símbolo n-ário, então este local inicia um segmento $gx_1 \ldots x_n$. Substitua este segmento por alguma nova variável y, obtendo uma fórmula que nós podemos chamar de $\alpha_y^{gx_1 \ldots x_n}$. Então α^π é

$$\forall y(\pi_g(x_1, \ldots, x_n, y) \to (\alpha_y^{gx_1 \ldots x_n})^\pi). \tag{4.7}$$

Por exemplo,

$$\begin{aligned}(Pfgx)^\pi &= \forall y(\pi_g(x, y) \to (Pfy)^\pi) \\ &= \forall y(\pi_g(x, y) \to \forall z(\pi_f(y, z) \to (Pz)^\pi)) \\ &= \forall y(\pi_g(x, y) \to \forall z(\pi_f(y, z) \to \pi_P(z))).\end{aligned} \tag{4.8}$$

A interpretação de uma fórmula não atômica é definida na forma mais óbvia. $(\neg\varphi)^\pi$ é $(\neg\varphi^\pi)$, $(\varphi \to \psi)^\pi$ é $(\varphi^\pi \to \psi^\pi)$ e $(\forall x\varphi)^\pi$ é $\forall x(\pi_\forall(x) \to \varphi^\pi)$ (Ou seja, os quantificadores são *relativizados* com relação a $\pi_\forall$).

O sentido no qual φ^π *diz a mesma coisa* que φ é feito preciso no próximo lema básico.

Lema 4.4.1 Seja π uma interpretação de L_0 para T_1, seja $\mathcal{B}$ um modelo de T_1. Para qualquer fórmula φ de L_0 e qualquer mapeamento s de variáveis para $|{}^\pi\mathcal{B}|$,

$$\models_{{}^\pi\mathcal{B}} \varphi[s] \text{ sse } \models_{\mathcal{B}} \varphi^\pi[s]. \tag{4.9}$$

Prova Nós usaremos indução em φ, mas somente o caso de uma fórmula atômica α é não trivial. Para α, nós usamos indução nos números de lugares nos quais símbolos funcionais ocorram. É imediato se este número é zero. Do contrário, assumindo g como um símbolo funcional unário, sem perder generalidade para os símbolos n-ários,

$$\alpha^\pi = \forall y(\pi_g(x, y) \to \beta^\pi),$$

onde $\beta_{gx}^y = \alpha$. Seja

$$\begin{aligned}b = &\text{ o único } b \text{ tal que } \models_{\mathcal{B}} \pi_g[\![s(x), b]\!] \\ = &\, g^{{}^\pi\mathcal{B}}(s(x)).\end{aligned}$$

Então

$$\begin{aligned}
\models_{\mathcal{B}} \alpha^{\pi}[s] &\Leftrightarrow \models_{\mathcal{B}} \beta^{\pi}[s(y|b)] \\
&\Leftrightarrow \models_{\pi\mathcal{B}} \beta[s(y|b)] \text{ (pela hipótese de indução)} \\
&\Leftrightarrow \models_{\pi\mathcal{B}} \beta^{y}_{gx}[s] \text{ (pelo lema da substituição)} \\
&\Leftrightarrow \models_{\pi\mathcal{B}} \alpha[s].
\end{aligned}$$

■

Corolário 4.4.1 Sejam L_0 e L_1 duas linguagens onde $L_0 \subseteq L_1$. Seja T_1 uma teoria em L_1 e π_i uma interpretação identidade de L_0 para T_1. Então toda fórmula φ em L_0, e conseqüentemente em L_1, é logicamente equivalente à sua tradução φ^{π}. Ou seja, $\models \varphi \leftrightarrow \varphi^{\pi}$, ou $\varphi \Leftrightarrow \varphi^{\pi}$.

Prova Primeiro provamos o corolário para o caso de φ ser uma fórmula atômica. Para isso utilizamos indução no número de símbolos funcionais que ocorrem em φ. Caso não haja símbolos funcionais, $\varphi = Pv_1 \ldots v_n$. Então $\varphi^{\pi} = \pi_P(v_1 \ldots v_n) = Pv_1 \ldots v_n = \varphi$. Caso contrário, tome o local mais à direita onde um símbolo funcional g ocorre,

$$\begin{aligned}
\varphi^{\pi} &= \forall y(\pi_g(x_1, \ldots, x_n, y) \rightarrow (\varphi_y^{gx_1 \ldots x_n})^{\pi}) && \text{(pela definição de tradução)} \\
&= \forall y(g(x_1, \ldots, x_n) \approx y \rightarrow (\varphi_y^{gx_1 \ldots x_n})^{\pi}) && \text{(pela interpretação identidade)} \\
&\Leftrightarrow \forall y(g(x_1, \ldots, x_n) \approx y \rightarrow \varphi_y^{gx_1 \ldots x_n}) && \text{(pela hipótese de indução)} \\
&\Leftrightarrow \varphi.
\end{aligned}$$

Provamos agora para o caso de φ ser qualquer fórmula. Utilizamos indução no número de símbolos lógicos. Assim resumidamente,

$$\begin{aligned}
(\neg\varphi)^{\pi} &= (\neg\varphi^{\pi}) \Leftrightarrow (\neg\varphi). \\
(\varphi \rightarrow \phi)^{\pi} &= (\varphi^{\pi} \rightarrow \phi^{\pi}) \Leftrightarrow (\varphi \rightarrow \phi). \\
(\forall x\varphi)^{\pi} &= \forall x(\pi_{\forall}(x) \rightarrow \varphi^{\pi}) = \forall x((x \approx x) \rightarrow \varphi^{\pi}) \Leftrightarrow \forall x(\varphi^{\pi}) \Leftrightarrow (\forall x\varphi).
\end{aligned}$$

■

O próximo corolário justifica nossa escolha de notação para $\pi^{-1}[T_1]$.

Corolário 4.4.2 Para uma sentença σ em L_0,

$$\sigma \in \pi^{-1}[T_1] \text{ sse } \sigma^{\pi} \in T_1. \tag{4.10}$$

Prova Relembre-se que por definição

$$\begin{aligned}\sigma \in \pi^{-1}[T_1] &\Leftrightarrow \text{ para todo modelo } \mathcal{B} \text{ de } T_1,\ \models_{\pi\mathcal{B}} \sigma \\ &\Leftrightarrow \text{ para todo modelo } \mathcal{B} \text{ de } T_1,\ \models_{\mathcal{B}} \sigma^{\pi} \text{ (pelo lema 4.4.1)} \\ &\Leftrightarrow T_1 \models \sigma^{\pi}.\end{aligned}$$

■

Definição 4.4.3 Uma *interpretação* π de uma teoria T_0 para uma teoria T_1 é uma interpretação π da linguagem de T_0 para T_1 tal que

$$T_0 \subseteq \pi^{-1}[T_1]. \tag{4.11}$$

Em outras palavras, é necessário que para qualquer sentença σ em L_0,

$$\sigma \in T_0 \Rightarrow \sigma^{\pi} \in T_1, \tag{4.12}$$

de onde podemos concluir que, nessa condição, T_1 é tão poderosa quanto T_0. Além disso, $\pi^{-1}[T_1]$ é a maior teoria a qual π interpreta para T_1. Se $T_0 = \pi^{-1}[T_1]$, então nós temos

$$\sigma \in T_0 \Leftrightarrow \sigma^{\pi} \in T_1. \tag{4.13}$$

Neste caso π é dito como sendo uma *interpretação fiel*[1] de T_0 para T_1.

■

4.4.3 Definição de predicados e funções

Veremos mais adiante que, para podermos comparar duas teorias T_A e T_B, nos será útil acrescentar a uma dessas teorias, digamos T_B, algumas novas sentenças que servirão para definir símbolos funcionais ou predicados da linguagem da outra teoria, no caso T_A.

Da mesma forma, em matemática, muitas vezes é útil introduzir definições de novas funções. Por exemplo, em teoria de conjuntos alguém pode definir a operação $\mathcal{P}$ de conjuntos potência por uma sentença do tipo: *Seja $\mathcal{P}x$ o conjunto cujos membros sejam os subconjuntos de x.* Ou por uma sentença numa linguagem formal:

[1] *faithful* em inglês

$$\forall v_1 \forall v_2 [\mathcal{P} v_1 \approx v_2 \leftrightarrow \forall u (u \in v_2 \leftrightarrow u \subseteq v_1)].$$

Definições não são como teoremas ou axiomas. Diferentemente de teoremas, definições não são coisas a serem provadas. E, diferentemente de axiomas, não esperamos que as definições acrescentem informação. Esperamos que uma definição contribua para nossa conveniência, não para nosso conhecimento.

Dessa forma, uma definição deve ser concebida de uma maneira razoável. Como exemplo de uma definição não razoável em teoria dos números, vamos supor que introduzimos um novo símbolo funcional f através da definição: $f(x) = y$ *se e somente se* $x < y$. Ou pela sentença numa linguagem formal:

$$\forall v_1 \forall v_2 (f v_1 \approx v_2 \leftrightarrow v_1 < v_2).$$

Desde que sabemos que $1 < 2$, vemos que $f(1) = 2$. Mas também $1 < 3$, então nós temos $f(1) = 3$. E então chegamos à conclusão (a qual não envolve f !) que $2 = 3$. Obviamente esta definição de f foi, de alguma maneira, muito ruim. Ela nos permitiu concluir que $2 = 3$, conclusão que não teríamos obtido sem a presença desta definição.

Quando acrescentamos uma nova sentença a uma teoria existente, dizemos que estamos realizando uma *extensão* dessa teoria. Quando essa sentença é uma definição de um novo símbolo (funcional ou predicado) não existente na linguagem da teoria original, dizemos que a extensão é uma *extensão por definição*. Quando essa extensão acrescenta novas conclusões na linguagem da teoria original, não pertencentes à teoria original, como no caso anterior, dizemos que a extensão é *não conservativa*. Caso contrário, dizemos que a extensão é *conservativa*[2].

Nesta subseção nós iremos considerar as condições sob as quais nós podemos estar seguros que uma definição seja satisfatória, i.e., que a extensão por definição assim gerada seja conservativa. Além disso, iremos provar algumas propriedades sobre essas extensões por definição que nos serão úteis nas justificativas de nossas futuras comparações.

Para simplificar a notação, nós consideraremos apenas a definição de um predicado binário P e de um símbolo funcional unário f, mas as mesmas provas se aplicam analogamente a outros símbolos funcionais e predicados n-ários da mesma forma.

Para efeito dos teoremas desta seção, considere uma teoria T em uma linguagem L ainda não contendo o símbolo funcional unário f e nem o predicado binário P. Suponha que uma entre as duas situações abaixo ocorra:

I. Adicionamos f à linguagem L, obtendo a nova linguagem L_f, introduzindo-o através da definição:

[2]Ver [TURSK87] para maiores detalhes.

$$\delta = \forall v_1 \forall v_2 (f v_1 \approx v_2 \leftrightarrow \varphi), \tag{4.14}$$

onde φ é uma fórmula em L (i.e., uma fórmula não contendo f) na qual somente v_1 e v_2 podem ocorrer livres. φ deve ser escolhida de tal forma que a sentença ε abaixo pertença à teoria T. ε é uma abreviação de uma fórmula mais longa, que significa *para todo v_1 existe um e um único v_2 tal que φ é verdadeiro.*

$$\varepsilon = \forall v_1 \exists! v_2 \varphi. \tag{4.15}$$

II. Adicionamos P à linguagem L, obtendo a nova linguagem L_P, introduzindo-o através da definição:

$$\delta = \forall v_1 \forall v_2 (P v_1 v_2 \leftrightarrow \phi), \tag{4.16}$$

onde ϕ é uma fórmula em L (onde somente v_1 e v_2 podem ocorrer livres).

Teorema 4.4.1 Qualquer das definições δ em (I) ou (II) acima é não criativa, ou seja, a extensão gerada por δ é conservativa, i.e., para toda sentença σ na linguagem L vale a relação:

$$T; \delta \models \sigma \Rightarrow T \models \sigma. \tag{4.17}$$

Prova Caso ocorra a situação (I), seja $\mathcal{U}$ um modelo de T. ($\mathcal{U}$ é uma estrutura para a linguagem L). Se $d \in |\mathcal{U}|$, seja $F(d)$ o único $e \in |\mathcal{U}|$ tal que $\models_{\mathcal{U}} \varphi[d, e]$. Há um único e porque $\models_{\mathcal{U}} \varepsilon$. Seja $(\mathcal{U}, F)$ a estrutura para a linguagem aumentada L_f que concorda com $\mathcal{U}$ nos parâmetros originais e que associa F ao símbolo f. Então é fácil ver que $(\mathcal{U}, F)$ é um modelo de δ. Além disso, $\mathcal{U}$ e $(\mathcal{U}, F)$ satisfazem as mesmas sentenças da linguagem original L.. Em particular $(\mathcal{U}, F)$ é um modelo de T. Daí

$$\begin{aligned} T; \delta \models \sigma &\Rightarrow \models_{(\mathcal{U},F)} \sigma \\ &\Rightarrow \models_{\mathcal{U}} \sigma. \end{aligned}$$

Para o caso da situação (II), a prova é semelhante: seja $\mathcal{U}$ um modelo de T. Se $d, e \in |\mathcal{U}|$, seja o par $\langle d, e\rangle \in P'$, tal que $\models_{\mathcal{U}} \phi[d, e]$. $(\mathcal{U}, P')$ é modelo de δ e é modelo de T. Daí segue $T; \delta \models \sigma \Rightarrow \models_{(\mathcal{U},P')} \sigma \Rightarrow \models_{\mathcal{U}} \sigma$.

■

Vale lembrar agora que a linguagem de primeira ordem que utilizaremos (LPO') possui apenas símbolos funcionais 0-ários, ou seja, constantes. Assim a fórmula δ em (4.14) toma a forma:

$$\delta = \forall v_1(c \approx v_1 \leftrightarrow \varphi), \tag{4.18}$$

onde, apenas por questão de notação, o símbolo f foi adequadamente substituído pelo símbolo c. Se, além disso, sempre fizermos φ como sendo $\varphi = (v_1 \approx c_1)$ onde c_1 é uma constante pertencente a L então δ transforma-se novamente em

$$\delta = \forall v_1(c \approx v_1 \leftrightarrow v_1 \approx c_1). \tag{4.19}$$

Com isso, ε em (4.15) torna-se logicamente equivalente a

$$\exists v_1(v_1 \approx c_1 \wedge \forall v_2(v_2 \approx c_1 \rightarrow v_1 \approx v_2)). \tag{4.20}$$

Afortunadamente, a sentença (4.20) é válida e portanto ε também é válida. Assim ε pertence a qualquer teoria T em L. Concluindo, qualquer definição δ de um novo predicado ou de uma nova constante, nos formatos descritos respectivamente em (4.16) e (4.19), são suficientes para garantir (4.17).

Suponha novamente que uma das situações (I) ou (II) ocorra, queremos agora construir uma interpretação π:

i. Para a situação (I), seja π uma interpretação de L_f para T. π é a interpretação identidade em todos os parâmetros exceto f. A fórmula π_f é φ. O fato que $T \models \varepsilon$ é exatamente o que precisamos para verificar que π é realmente uma interpretação. Para qualquer modelo $\mathcal{U}$ de T, ${}^{\pi}\mathcal{U}$ é uma estrutura previamente chamada $(\mathcal{U}, F)$, que é um modelo de $T; \delta$.

ii. Para a situação (II), seja π uma interpretação de L_P para T. π é a interpretação identidade em todos os parâmetros exceto P. A fórmula π_P é ϕ. Para qualquer modelo $\mathcal{U}$ de T, ${}^{\pi}\mathcal{U}$ é uma estrutura previamente chamada $(\mathcal{U}, P')$, que é um modelo de $T; \delta$.

Teorema 4.4.2 Para qualquer dos dois casos (i) e (ii) acima, vale a relação

$$\pi^{-1}[T] = \mathrm{Cn}(T;\delta). \tag{4.21}$$

Prova Primeiro observe que qualquer modelo $\mathcal{B}$ de $T;\delta$ é igual a ${}^{\pi}\mathcal{U}$, onde $\mathcal{U}$ é a restrição de $\mathcal{B}$ para a linguagem de T. Daí para uma sentença σ em L_f (ou em L_P),

$$\begin{aligned}\sigma \in \pi^{-1}[T] &\Leftrightarrow \models_{{}^{\pi}\mathcal{U}} \sigma \text{ para todo modelo } \mathcal{U} \text{ de } T\\ &\Leftrightarrow \models_{\mathcal{B}} \sigma \text{ para todo modelo } \mathcal{B} \text{ de } T;\delta\\ &\Leftrightarrow T;\delta \models \sigma.\end{aligned}$$

■

Corolário 4.4.3 Segundo o teorema 4.4.2, π é uma interpretação fiel de $\mathrm{Cn}(T;\delta)$ para T, uma vez que

$$\pi^{-1}[T] = \mathrm{Cn}(T;\delta). \tag{4.22}$$

■

Nós podemos agora delinear uma conclusão adicional; a definição é *eliminável.*

Teorema 4.4.3 Assuma que nós temos uma das situações (I) ou (II) descritas na página 40. Então para qualquer σ em L_f (ou em L_P) nós podemos achar a sentença σ^{π} na linguagem original L tal que

1. $T;\delta \models (\sigma \leftrightarrow \sigma^{\pi})$.
2. $T;\delta \models \sigma \Leftrightarrow T \models \sigma^{\pi}$.
3. Se f não ocorre em σ (ou se P não ocorre em σ), então $\models (\sigma \leftrightarrow \sigma^{\pi})$.

Prova Parte (3) segue do fato que π é a interpretação identidade em todos os parâmetros exceto f (ou exceto P). Parte (2) ratifica que π é uma interpretação fiel de $\mathrm{Cn}(T;\delta)$ para T. Desde que π é fiel, para (1) é suficiente mostrar que $T \models (\sigma \leftrightarrow \sigma^{\pi})^{\pi}$. Isto segue de (3), uma vez que $(\sigma \leftrightarrow \sigma^{\pi})^{\pi}$ é $(\sigma^{\pi} \leftrightarrow \sigma^{\pi\pi})$, a qual é válida.

■

4.4.4 Composição de definições

Esta seção visa ampliar os teoremas da seção anterior para o caso em que estendemos conservativamente uma teoria através de várias definições. Denominaremos essa situação como sendo uma *composição de definições.* Essa composição será utilizada nas provas da segunda parte deste trabalho.

Além disso, para simplificar os teoremas desta seção, introduzimos a seguir uma definição que objetiva nomear e descrever todos os parâmetros de uma composição de definições que servirá de base para as provas desses teoremas.

Definição 4.4.4 (Composição de definições e $\bar{\pi}^n$) Seja a inclusão de mais de uma definição δ caracterizando uma *composição de definições.* Seja δ_1 a primeira definição adicionada de um novo símbolo k_1 (funcional ou predicado), utilizando símbolos da linguagem L de uma teoria T e gerando $\mathrm{Cn}(T;\delta_1)$. Seja δ_2 a segunda definição adicionada, utilizando símbolos da linguagem $\{L;k_1\}$ de $\mathrm{Cn}(T;\delta_1)$ e gerando $\mathrm{Cn}((T;\delta_1);\delta_2)$. Observe que a definição de δ_2 pode conter o símbolo k_1 recém introduzido por δ_1. Isso é uma situação que ocorrerá durante as provas deste trabalho, a qual denominaremos de *definições em camadas.* E assim sucessivamente até obtermos uma linguagem $\{L;k_1;\ldots;k_n\}$ e uma teoria $\mathrm{Cn}(((..(T;\delta_1);\delta_2)\ldots);\delta_n)$.

Vamos construir analogamente n interpretações, seguindo o procedimento descrito nos itens (i) e (ii) na página 42, e obtendo a tabela abaixo:

A	B	C
k_1 / δ_1 / π^1 / $\pi^1_{k_1} = \psi_1$.	π^1 de $\{L;k_1\}$ para T.	$(\pi^1)^{-1}[T] = \mathrm{Cn}(T;\delta_1)$.
k_2 / δ_2 / π^2 / $\pi^2_{k_2} = \psi_2$.	π^2 de $\{L;k_1;k_2\}$ para $\mathrm{Cn}(T;\delta_1)$.	$(\pi^2)^{-1}[\mathrm{Cn}(T,\delta_1)] = \mathrm{Cn}((T;\delta_1);\delta_2)$.
k_3 / δ_3 / π^3 / $\pi^3_{k_3} = \psi_3$.	π^3 de $\{L;k_1;k_2;k_3\}$ para $\mathrm{Cn}((T;\delta_1);\delta_2)$.	$(\pi^3)^{-1}[\mathrm{Cn}(T,\delta_1,\delta_2)] = \mathrm{Cn}(((T;\delta_1);\delta_2);\delta_3)$.
...	...	...
k_n / δ_n / π^n / $\pi^n_{k_n} = \psi_n$.	π^n de $\{L;k_1;\ldots;k_n\}$ para $\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-1})$.	$(\pi^n)^{-1}[\mathrm{Cn}(T,\delta_1,\ldots,\delta_{n-1})] = \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n)$.

onde as colunas significam

A= k_i é o símbolo predicado ou funcional que foi incluído através da definição δ_i. π^i é a interpretação identidade em todos os parâmetros exceto k_i, onde $\pi^i_{k_i} = \psi_i$. $\psi_i = \varphi_i$ para o caso de k_i ser um símbolo funcional e $\psi_i = \phi_i$ para o caso de k_i ser um símbolo predicado. φ_i ou ϕ_i é a fórmula encontrada na definição δ_i e ambas equivalem, respectivamente, a φ e ϕ nas situações (I) e (II) na seção 4.4.3.

B= π^i é a interpretação da linguagem $\{L;k_1;\ldots;k_i\}$ obtida com a inclusão do símbolo k_i para a teoria $\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{i-1})$ sem a definição δ_i.

C= Nessas condições e de acordo com o teorema 4.4.2, são válidas as igualdades $(\pi^i)^{-1}[\mathrm{Cn}(T,\delta_1,\ldots,\delta_{i-1})] = \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_i)$.

É natural que, numa composição de definições como apresentada acima, queiramos construir uma interpretação da linguagem final $\{L; k_1; \ldots; k_n\}$ para a teoria original (T). Seja então $\bar{\pi}^n$ esta interpretação, construída da seguinte forma: $\bar{\pi}^n$ é a interpretação identidade para todos os parâmetros, exceto para os símbolos $k_1, \ldots, k_n$. Para esses símbolos, definimos

$$\begin{aligned}
\bar{\pi}^n_{k_1} &= \psi_1 (= \pi^1_{k_1}). \\
\bar{\pi}^n_{k_2} &= (\psi_2)^{\pi_1}. \\
\bar{\pi}^n_{k_3} &= ((\psi_3)^{\pi_2})^{\pi_1}. \\
\ldots &= \ldots \\
\bar{\pi}^n_{k_n} &= (..((\psi_n)^{\pi_{n-1}})\cdots)^{\pi_1}.
\end{aligned}$$

Em outras palavras, queremos que $\bar{\pi}^n_{k_i}$ seja definido como a tradução para a teoria original T da fórmula ψ_i, que por sua vez define exatamente $\pi^i_{k_i}$.

■

Lema 4.4.2 $\bar{\pi}^n$ é realmente uma interpretação.

Prova Basta provar que $\bar{\pi}^n$ satisfaz as condições descritas na definição 4.4.1. Já vimos que para os parâmetros em que $\bar{\pi}^n$ é a interpretação identidade, essas condições são automaticamente satisfeitas. Resta-nos então verificar se as condições são satisfeitas para a interpretação dos símbolos $k_1, \ldots, k_n$.

Seja então k_i um desses símbolos, com $1 < i \leq n$. Queremos verificar se $\bar{\pi}^n_{k_i}$ atende a essas condições. Se $i = 1$, temos que $\bar{\pi}^n_{k_1} = \psi_1 = \pi^1_{k_1}$. Como $\pi^1_{k_1}$ atende às condições então $\bar{\pi}^n_{k_1}$ também atende. Para o caso de $i > 1$, temos que

$$\bar{\pi}^n_{k_i} = (..((\psi_i)^{\pi^{i-1}})\cdots)^{\pi^1}. \tag{4.23}$$

Caso k_i seja um símbolo predicado, a validade de $\bar{\pi}^n_{k_i}$ é imediata. Caso k_i seja um símbolo funcional unário, sem perder generalidade para outras aridades, temos que verificar se

$$T \models \forall v_1 \exists! v_2 (..((\psi_i)^{\pi^{i-1}})\cdots)^{\pi^1}. \tag{4.24}$$

Para isso temos que em π^i, $\pi^i_{k_i} = \psi_i$ e como, por hipótese, π^i é uma interpretação de $\{L; k_1; \ldots; k_i\}$ para $\mathrm{Cn}((..(T; \delta_1)\ldots); \delta_{i-1})$, então

$$
\begin{aligned}
& (..(T;\delta_1)\ldots);\delta_{i-1} \models \forall v_1 \exists! v_2 \psi_i \\
\Leftrightarrow\ & (..(T;\delta_1)\ldots);\delta_{i-2} \models (\forall v_1 \exists! v_2 \psi_i)^{\pi^{i-1}} && \text{(pelo teorema 4.4.3)} \\
\Leftrightarrow\ & (..(T;\delta_1)\ldots);\delta_{i-2} \models \forall v_1 \exists! v_2 (\psi_i)^{\pi^{i-1}} && \text{(pois } \pi_{\forall}^{i-1} \text{ é identidade)} \\
\Leftrightarrow\ & \ldots \models \ldots \\
\Leftrightarrow\ & T \models \forall v_1 \exists! v_2 (..((\psi_i)^{\pi^{i-1}})\cdots)^{\pi^1} && \text{(eliminando todos os } \delta\text{'s).}
\end{aligned}
$$

■

Teorema 4.4.4 Seja $\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n)$ uma composição de definições como descrita na definição 4.4.4 e seja σ uma sentença na linguagem original L de T. Então

$$((..(T;\delta_1)\ldots);\delta_n) \models \sigma \Rightarrow T \models \sigma. \tag{4.25}$$

Prova Vamos provar por indução no número n de novas definições. Assim, para apenas uma definição, vale (4.17). Para o caso de n definições ($n > 1$), temos

$$
\begin{aligned}
& ((..(T;\delta_1)\ldots);\delta_n) \models \sigma \\
& \quad \Leftrightarrow (T';\delta_n) \models \sigma \text{ (onde } T' = \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-1})\ ) \\
& \quad\quad \Rightarrow T' \models \sigma \text{ (por (4.17))} \\
\Rightarrow\ & ((..(T;\delta_1)\ldots);\delta_{n-1}) \models \sigma \\
& \quad\quad \Rightarrow T \models \sigma \text{ (pela hipótese de indução).}
\end{aligned}
$$

■

Teorema 4.4.5 Seja $\bar{\pi}^n$ a interpretação da linguagem $L;k_1;\ldots;k_n$ para a teoria T, tal como descrito na definição 4.4.4. Então

$$(\bar{\pi}^n)^{-1}[T] = \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n). \tag{4.26}$$

Prova Inicialmente provaremos por indução em n que, se $\mathcal{B}$ é um modelo de T, então a relação a seguir é verdadeira

$$^{\pi^n}(\cdots(^{\pi^1}(\mathcal{B}))..) = {}^{\bar{\pi}^n}(\mathcal{B}). \tag{4.27}$$

Para $n = 1$, $\pi^1 = \bar{\pi}^1$ e (4.27) é imediata. Nossa hipótese de indução é então

$$^{\pi^{n-1}}(\cdots(^{\pi^1}(\mathcal{B}))..) = {}^{\bar{\pi}^{n-1}}(\mathcal{B}). \tag{4.28}$$

É fácil verificar que $\bar{\pi}^n$ concorda com todos os parâmetros de $\bar{\pi}^{n-1}$, com a única exceção de que $\bar{\pi}^n$ possui a mais a interpretação para o parâmetro k_n, tal que $\bar{\pi}^n_{k_n} = (..((\psi_n)^{\pi_{n-1}})\cdots)^{\pi_1}$. Portanto $^{\bar{\pi}^n}(\mathcal{B})$ difere de $^{\bar{\pi}^{n-1}}(\mathcal{B})$ apenas pelo fato de que o primeiro possui uma relação a mais que o segundo, que é justamente a relação associada ao símbolo k_n [3]. Pelo mesmo motivo $^{\pi^n}(\cdots(^{\pi^1}(\mathcal{B}))..)$ difere de $^{\pi^{n-1}}(\cdots(^{\pi^1}(\mathcal{B}))..)$.

Devido a este fato e juntamente à hipótese de indução, para provarmos (4.27) basta provarmos que as duas relações associadas à k_n nas duas estruturas em (4.27) são exatamente iguais.

Fazendo uso do mesmo argumento utilizado no teorema 4.4.2, temos

$$\begin{aligned}
&\models_{\mathcal{U}} \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n) \\
\Leftrightarrow\ &\models_{\mathcal{W}} \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-1}) && \text{onde } \mathcal{U} = {}^{\pi^n}(\mathcal{W}) \\
\Leftrightarrow\ &\models_{\mathcal{X}} \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-2}) && \text{onde } \mathcal{W} = {}^{\pi^{n-1}}(\mathcal{X}) \\
\Leftrightarrow\ &\models \ldots \\
\Leftrightarrow\ &\models_{\mathcal{Z}} \mathrm{Cn}(T;\delta_1) && \text{onde } \ldots \\
\Leftrightarrow\ &\models_{\mathcal{B}} T && \text{onde } \mathcal{Z} = {}^{\pi^1}(\mathcal{B}).
\end{aligned} \tag{4.29}$$

Assim, sendo $\mathcal{B}$ um modelo de T e juntamente ao resultado (4.29), podemos aplicar recursivamente o lema 4.4.1 e obter

$$\begin{aligned}
\models_{^{\pi^{n-1}}(\cdots(^{\pi^1}(\mathcal{B}))..)} \psi_n[s] &\Leftrightarrow \models_{^{\pi^{n-2}}(\cdots(^{\pi^1}(\mathcal{B}))..)} (\psi_n)^{\pi^{n-1}}[s] \\
&\Leftrightarrow \models_{^{\pi^{n-3}}(\cdots(^{\pi^1}(\mathcal{B}))..)} ((\psi_n)^{\pi^{n-1}})^{\pi^{n-2}}[s] \\
&\Leftrightarrow \ldots \\
&\Leftrightarrow \models_{\mathcal{B}} (..((\psi_n)^{\pi_{n-1}})\cdots)^{\pi_1}[s].
\end{aligned} \tag{4.30}$$

A relação (4.30) é igual a

$$\models_{^{\pi^{n-1}}(\cdots(^{\pi^1}(\mathcal{B}))..)} \pi^n_{k_n}[s] \Leftrightarrow \models_{\mathcal{B}} \bar{\pi}^n_{k_n}[s]. \tag{4.31}$$

[3] Aqui, apenas para simplificar a prova, podemos pensar em uma relação associada à k_n, mesmo que este último seja um símbolo funcional. Neste caso, a tupla $\langle a_1, \ldots, a_m, b\rangle$ pertence a esta relação se e somente se $b = k_n^{^{\bar{\pi}^n}(\mathcal{B})} a_1, \ldots, a_m$.

Segundo a definição 4.4.2, todos os mapeamentos s em que $\models_{\pi^{n-1}(\cdots(\pi^1(\mathcal{B}))..)} \pi^n_{k_n}[s]$ definem a relação associada a k_n na estrutura $\pi^n(\cdots(\pi^1(\mathcal{B}))..)$. Da mesma forma todos os mapeamentos s em que $\models_{\mathcal{B}} \bar{\pi}^n_{k_n}[s]$ definem a relação associada ao símbolo k_n na estrutura $\bar{\pi}^n\mathcal{B}$. Mas de (4.31) essas relações são iguais. E portanto a relação (4.27) é verdadeira.

Provamos a seguir que

$$(\pi^n)^{-1}[\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-1})] = (\bar{\pi}^n)^{-1}[T]. \tag{4.32}$$

De fato,

$\sigma \in (\pi^n)^{-1}[\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-1})]$

$\Leftrightarrow \models_{\pi^n(\mathcal{U})} \sigma$ para todo $\mathcal{U}$ modelo de $\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-1})$

$\Leftrightarrow \models_{\pi^n(\cdots(\pi^1(\mathcal{B}))..)} \sigma$ para todo $\mathcal{B}$ modelo de T (do resultado (4.29))

$\Leftrightarrow \models_{\bar{\pi}^n(\mathcal{B})} \sigma$ do resultado (4.27)

$\Leftrightarrow \sigma \in (\bar{\pi}^n)^{-1}[T]$.

E pelo teorema 4.4.2, $(\pi^n)^{-1}[\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_{n-1})] = \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n)$, de onde obtemos finalmente que

$$(\bar{\pi}^n)^{-1}[T] = \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n).$$

■

Corolário 4.4.4 Segundo o teorema 4.4.5, $\bar{\pi}^n$ é uma interpretação fiel de $\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n)$ para T uma vez que

$$(\bar{\pi}^n)^{-1}[T] = \mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n).$$

■

Teorema 4.4.6 Seja uma composição de definições tal como descrita na definição 4.4.4. Então para qualquer sentença σ pertencente à linguagem $\{L; k_1; \ldots; k_n\}$, nós podemos achar a sentença $\sigma^{\bar{\pi}^n}$ na linguagem original L de T tal que

1. $((..(T;\delta_1)\ldots);\delta_n) \models (\sigma \leftrightarrow \sigma^{\bar{\pi}^n})$.
2. $((..(T;\delta_1)\ldots);\delta_n) \models \sigma \Leftrightarrow T \models \sigma^{\bar{\pi}^n}$.
3. Se σ pertence à linguagem L de T, então $\models (\sigma \leftrightarrow \sigma^{\bar{\pi}^n})$.

Prova Parte (3) segue do fato que $\bar{\pi}^n$ é a interpretação identidade em todos os parâmetros exceto $k_1, \ldots, k_n$. Parte (2) ratifica que $\bar{\pi}^n$ é uma interpretação fiel de $\mathrm{Cn}((..(T;\delta_1)\ldots);\delta_n)$ para T. Desde que $\bar{\pi}^n$ é fiel, para (1) é suficiente mostrar que $T \models (\sigma \leftrightarrow \sigma^{\bar{\pi}^n})^{\bar{\pi}^n}$. Isto segue de (3), uma vez que $(\sigma \leftrightarrow \sigma^{\bar{\pi}^n})^{\bar{\pi}^n}$ é $(\sigma^{\bar{\pi}^n} \leftrightarrow \sigma^{\bar{\pi}^n\bar{\pi}^n})$, a qual é válida.

■

4.5 Justificativa de nossas provas

4.5.1 Introdução

Voltando agora ao nosso problema, o que queremos com todos esses teoremas? Vimos que dados dois modelos de dados A e B, podemos axiomatizá-los, gerando assim duas teorias T_A e T_B. Seguindo os conceitos descritos neste capítulo, para descobrirmos se T_B é pelo menos tão poderosa quanto T_A, precisamos determinar uma interpretação π, caso ela exista, da linguagem L_A da teoria T_A para a teoria T_B, tal que para todo σ em L_A,

$$\sigma \in T_A \Rightarrow \sigma^{\pi} \in T_B. \tag{4.33}$$

Adicionalmente, se possível, gostaríamos que π fosse uma interpretação fiel, ou seja,

$$\sigma \in T_A \Leftrightarrow \sigma^{\pi} \in T_B. \tag{4.34}$$

Para atingir este objetivo, basta-nos definir uma função interpretação π entre as duas linguagens, traduzir cada axioma de T_A para L_B usando π, e verificar, utilizando um provador de teoremas, se para todos estes axiomas e suas respectivas traduções, a relação (4.33) é válida. Caso não seja, precisamos tentar determinar uma outra interpretação π e refazermos esses testes, fazendo isso repetidamente até encontrarmos um π satisfatório ou abandonarmos a procura.

Este processo, no entanto, não é nada trivial. Temos diversos problemas a enfrentar, tais como a complexidade do algoritmo de prova, a não modularidade do processo, um grande número de axiomas, etc, que dificultam sobremaneira a obtenção da meta (ver seção 5.2 na página 58 para maiores detalhes). Assim, qualquer facilitação em todo este processo é bem-vinda e, neste sentido, a forma de procedimento descrita acima apresenta alguns problemas, a saber:

1. O processo de tradução de um fórmula envolve um trabalho que o provador automático não pode ajudar muito, pois não é projetado para isso. Assim este processo deve ser ou feito manualmente ou utilizando algum programa externo. Isso é ruim, pois seria desejável que todos os procedimentos necessários pudessem ser tratados dentro do escopo de uma única ferramenta, no caso o provador de teoremas.

2. Quando se faz uma tradução de uma fórmula, a fórmula resultante é em geral muito mais complexa e portanto muito menos intuitiva que a original. Perde-se assim o conceito original. Isso é lamentável, uma vez que até mesmo a própria lógica matemática possui como seus símbolos principais conceitos mundanos e humanos, o que facilita em muito seu estudo e evolução. Para o presente estudo, a situação não é diferente.

Uma maneira de se eliminar os problemas descritos nos itens anteriores seria que as duas teorias pertencessem, de alguma maneira, a uma mesma linguagem. Mas a princípio estamos tratando com duas linguagens totalmente diferentes. Como isso poderia ser feito?

Uma alternativa de se fazer isso seria enriquecer uma das linguagens com todos os parâmetros da outra. Resumidamente, criamos uma definição para cada símbolo da linguagem de T_A utilizando nesta os símbolos da linguagem de T_B, tal como descrito nas seções anteriores. Dessa forma estendemos T_B e L_B com todas essas novas definições e obtemos uma nova teoria T'_B numa nova linguagem L'_B. Por construção, $L_A \subseteq L'_B$, e assim T_A e T'_B pertencem à mesma linguagem L'_B. Em seguida utilizamos T'_B para tentar provar os axiomas de T_A diretamente, sem a necessidade de se executar nenhuma tradução. O insucesso dessas provas levar-nos-ía a rever as definições até que uma prova completa fosse atingida. A pergunta que nos resta responder é: O sucesso dessa prova assim executada significa que o objetivo original (4.33) foi atingido? Nós vamos provar a seguir que *sim*.

4.5.2 Construção da comparação

Suponha que L_A possua todos os seus m símbolos extra-lógicos entre os n símbolos extra-lógicos $k_1, \ldots, k_n$, onde $m \leq n$, numerados de alguma forma apropriada. Construiremos uma composição de definições tal como exposto na definição 4.4.4. Assim, iniciamos definindo *adequadamente* k_1 a partir de uma fórmula ψ_1 em L_B, tal como descrito na seção 4.4.3 e de tal maneira que a sentença δ_1 assim originada tenha uma das formas expostas nas equações (4.16) e (4.19) (pgs. 41 e 42). Isso é possível pois trabalhamos com uma LPO sem símbolos funcionais além de constantes. Em seguida definimos, analogamente, k_2 a partir de uma fórmula ψ_2 em $\{L_B; k_1\}$, gerando δ_2. Fazemos isso sucessivamente até definirmos finalmente k_n. Ao final teremos:

$$L'_B = \{L_B; k_1; \ldots; k_n\}. \tag{4.35}$$

$$T'_B = \mathrm{Cn}((..(T_B; \delta_1)\ldots); \delta_n). \tag{4.36}$$

Seja $\bar{\pi}^n$ a interpretação fiel de T'_B para T_B tal como descrita na definição 4.4.4 (veja também o corolário 4.4.4). Além disso temos, por construção, que $L_A \subseteq L'_B$, e portanto, como visto na seção 4.4.2, podemos utilizar a interpretação identidade π_i de L_A para T'_B como exposto em (4.4).

Vamos verificar agora se π_i é uma interpretação de T_A para T'_B. Para isso, da definição 4.4.3, basta verificarmos se

$$\sigma \in T_A \Rightarrow \sigma^{\pi_i} \in T'_B. \tag{4.37}$$

Mas como $\models \sigma^{\pi_i} \leftrightarrow \sigma$, segundo o corolário 4.4.1, então é verdade que

$$\sigma^{\pi_i} \in T'_B \Leftrightarrow \sigma \in T'_B, \tag{4.38}$$

e a tarefa de verificar (4.37) passa a ser a tarefa de verificar

$$\sigma \in T_A \Rightarrow \sigma \in T'_B. \tag{4.39}$$

Do teorema 4.4.6, $T'_B \models \sigma \Leftrightarrow T_B \models \sigma^{\bar{\pi}^n}$, de onde se conclui que (4.39) é verdade se e somente se for verdade

$$\sigma \in T_A \Rightarrow \sigma^{\bar{\pi}^n} \in T_B. \tag{4.40}$$

$\bar{\pi}^n$ é uma interpretação da linguagem L'_B para T_B. Seja π a restrição de $\bar{\pi}^n$ aos símbolos lógicos e não lógicos da linguagem L_A. É fácil verificar que π é uma interpretação de L_A para T_B e, como σ é uma sentença de L_A, então $\sigma^{\pi} = \sigma^{\bar{\pi}^n}$ e (4.40) torna-se

$$\sigma \in T_A \Rightarrow \sigma^{\pi} \in T_B, \tag{4.41}$$

onde vemos que π é também uma interpretação da teoria T_A para T_B. Mas (4.41) é exatamente a relação (4.33) buscada no início. Podemos então concluir que para verificarmos (4.33) basta verificarmos (4.39), ou

$$\sigma \in T_A \Rightarrow \sigma^{\pi} \in T_B \text{ se e somente se } \sigma \in T_A \Rightarrow \sigma \in T'_B. \tag{4.42}$$

A mesma prova serve para a verificação de (4.34). Assim

$$\sigma \in T_A \Leftrightarrow \sigma^{\pi} \in T_B \text{ se e somente se } \sigma \in T_A \Leftrightarrow \sigma \in T'_B. \tag{4.43}$$

Do resultado (4.42), para concluirmos que T_B é tão poderosa quanto T_A, basta provarmos que para todo $\sigma \in T_A$, é verdade que $T'_B \models \sigma$. Isso é equivalente a verificar, utilizando-se um provador de teoremas, se

$$\begin{aligned} &\boxed{\sigma_i \in \phi_A \Rightarrow \phi'_B \vdash \sigma_i,} \\ &\text{onde } \phi_A = \{\sigma_1, \ldots, \sigma_p\} \text{ tal que } T_A = \mathrm{Cn}(\phi_A), \\ &\quad \phi'_B = \phi_B \cup \{\delta_1, \ldots, \delta_n\} \text{ tal que } T'_B = \mathrm{Cn}(\phi'_B), \\ &\quad \text{e } \phi_B = \{\gamma_1, \ldots, \gamma_q\}. \end{aligned} \tag{4.44}$$

Em nosso estudo, $\phi_A = \{\sigma_1, \ldots, \sigma_p\}$ nada mais é que o conjunto dos axiomas gerados durante o processo de axiomatização de algum modelo de dados A, como descrito no capítulo anterior. Da mesma forma, $\phi_B = \{\gamma_1, \ldots, \gamma_q\}$ são os q axiomas gerados durante a axiomatização de um segundo modelo B. T_A e T'_B são as teorias induzidas respectivamente por ϕ_A e ϕ'_B, onde ϕ'_B é a união dos axiomas de ϕ_B com as definições $\{\delta_1, \ldots, \delta_n\}$.

O sucesso dessa verificação depende da construção correta e apropriada das definições $\delta_1, \ldots . \delta_n$. A concepção dessas definições não é totalmente objetiva. Ou seja, precisamos conceber essas definições apoiados em nosso bom senso e paralelamente testar se $\phi'_B \vdash \sigma_i$ para todo $\sigma_i \in \phi_A$. A não verificação desta relação para algum desses σ_i nos força a alterar algumas (ou eventualmente todas) definições. O prejuízo disso é que na maioria das vezes precisamos verificar novamente (4.44) para os σ_i's que já tinham sido verificados anteriormente. Essa necessidade de se rever as provas feitas anteriormente sempre que houver uma alteração nas definições caracteriza a verificação como sendo um processo *não modular*. Algumas metodologias para minimizar o efeito dessa não modularidade foram adotadas e serão descritas com detalhes no capítulo seguinte.

4.5.3 Escopo deste trabalho relativo às comparações

Analogamente ao capítulo sobre as axiomatizações, queremos também deixar claro no presente capítulo qual o escopo que será adotado durante as comparações na segunda parte deste trabalho.

Procuraremos determinar uma interpretação entre duas teorias de modelos de BDs através da construção de um conjunto de definições $\{\delta_1, \ldots, \delta_n\}$ e da verificação da relação (4.44), exatamente como aqui exposto. Não procuraremos, apesar de desejável, encontrar uma interpretação entre essas teorias que seja fiel. Mais que isso, apontaremos até contra-exemplos mostrando que a interpretação encontrada não é fiel.

Fica também, assim como no caso das axiomatizações, a procura de interpretações fiéis um problema para futuras pesquisas.

Capítulo 5

Provador e Metodologia

5.1 OTTER: um provador de teoremas

5.1.1 Introdução

Precisávamos, neste momento, de alguma maneira estruturada, de preferência automatizada, de executar as provas entre os modelos. Optamos assim pelo uso de algum provador de teoremas que realizasse este trabalho. Este provador, como já visto, deveria ser capaz de realizar provas em LPO' (i.e., LPO com igualdade e sem símbolos funcionais). Poderíamos tentar utilizar PROLOG para este fim, por causa de sua eficiência, porém o fato de o mesmo ser limitado apenas às cláusulas de Horn, inviabilizou o seu uso, uma vez que muitas das fórmulas que usaríamos eram conclusões disjuntivas ($A \Rightarrow (B \vee C)$), não sendo portanto cláusulas de Horn.

Chegamos então a desenvolver um provador de teoremas, escrito em PROLOG, cujo mecanismo de prova era baseado no método do *Tableau* (ver [BELL77]). Este método fornece uma maneira estruturada, e quase algorítmica, para verificar se um conjunto de fórmulas em LPO é insatisfatível. O grande problema está na complexidade não polinomial do algorítmo de prova, como exposto na seção 3.2. Este fato acabou inviabilizando o uso de *nosso* provador de teoremas, uma vez que o mesmo era eficiente apenas na avaliação de conjuntos muito reduzidos de fórmulas.

Partimos então para a escolha de um provador de teoremas já existente que satisfizesse nossas exigências. O mesmo teria que ser expressivo o suficiente para tratar nosso problema, possuir rotinas internas de otimização que ao menos atenuassem o problema da complexidade, e ser comprovadamente confiável, de preferência gratuito e de fácil uso. Escolhemos então o provador de teoremas chamado OTTER, o qual descrevemos a seguir.

5.1.2 Introdução ao OTTER

OTTER (Organized Techniques for Theorem-proving and Effective Research) é

um provador de teoremas que utiliza resolução em suas provas. Ele oferece as regras de inferência resolução binária, hiperresolução, UR-resolução e paramodulação binária. OTTER se aplica a expressões escritas em lógica de primeira ordem com igualdade, sendo portanto adequado às nossas necessidades. No entanto, ele não é indicado na solução de problemas que requeiram provas por indução ou funções de ordem superior.

iniciando o OTTER

OTTER é essencialmente um programa não interativo. Em geral, ele lê de um arquivo de entrada padrão e escreve num arquivo de saída padrão:

`otter <` *arquivo-de-entrada* `>` *arquivo-de-saída*

Nenhuma opção de linha de comando é aceita; todas as opções devem ser fornecidas diretamente no arquivo de entrada.

Sintaxe

OTTER reconhece dois tipos básicos de expressões: cláusulas e sentenças. Cláusulas são disjunções simples, cujas variáveis são quantificadas universal e implicitamente, não existindo portanto quantificadores existenciais. As buscas de provas de OTTER operam sempre sobre cláusulas. Sentenças são fórmulas de primeira ordem sem variáveis livres—todas as variáveis são explicitamente quantificadas. Quando sentenças são introduzidas, OTTER imediatamente as traduz para cláusulas. Neste trabalho utilizamos apenas sentenças como entrada, devido à sua maior clareza.

Nomes de variáveis, constantes, símbolos funcionais e símbolos predicados são identificados em geral como cadeias de caracteres alfanuméricos. Alguns outros caracteres de importância são:

- . (ponto) — termina cada expressão de entrada.
- % — inicia um comentário (o qual termina com o final da linha).
- , () [] {} — são símbolos de pontuação e agrupamento.

Variáveis. Determinar se um um termo simples é uma constante ou é uma variável depende do contexto do termo. Um termo simples numa sentença é uma variável sse ele for limitado por um quantificador.

Igualdade. Entre os nomes reservados de OTTER destacamos aqueles associados à igualdade. O nome = (e qualquer nome que comece com `eq`, `EQ` ou `Eq`), quando usado como um símbolo predicado binário, é reconhecido como um predicado de igualdade pelos processos de demodulação e paramodulação. Além disso, =, como um predicado binário, pode ser escrito na forma infixa. Assim, por exemplo, `=(a,b)` pode ser escrito como `a=b`. Da mesma forma, a desigualdade pode ser escrita nas três formas seguintes que são equivalentes: `-(=(a,b))` ou `-(a=b)` ou ainda `a!=b`.

Com exceção dos predicados relativos a igualdade, para efeito do presente trabalho, todos os demais átomos serão escritos na forma prefixa. Por exemplo, o predicado binário `is_a` sempre será escrito na forma `is_a(x,y)` e nunca na forma `x is_a y`.

Sentenças

A tabela 5.1 lista os símbolos lógicos para a construção de sentenças.

Tabela 5.1: Símbolos lógicos

negação	`-`
disjunção	`|`
conjunção	`&`
implicação	`->`
equivalência	`<->`
quantificador existencial	`exists`
quantificador universal	`all`

Sentenças são usualmente escritas e abreviadas de uma maneira natural. Abaixo alguns exemplos.

uso padrão	sintaxe do OTTER (abreviada)
$\forall x P(x)$.	`all x P(x).`
$\forall xy \exists z (P(x,y,z) \vee Q(x,z))$.	`all x y exists z (P(x,y,z) | Q(x,z)).`
$\forall x (P(x) \wedge Q(x) \wedge R(x) \rightarrow S(x))$.	`all x (P(x) & Q(x) & R(x) -> S(x)).`

Note que, se uma sentença tem uma cadeia de quantificadores idênticos, todos além do primeiro podem ser descartados. Por exemplo, `all x all y all z p(x,y,z)` pode ser abreviado para `all x y z p(x,y,z)` . Em expressões envolvendo as operações associativas `&` e `|`, parênteses extras podem ser removidos. Além disso, uma precedência padrão nos símbolos lógicos nos permite descartar mais parênteses: `<->` tem a mesma precedência que `->`, e para os outros símbolos lógicos, em ordem decrescente de precedência, temos `->`, `|`, `&`, `-`. Por exemplo, `p | -q & r -> -s | t` representa `(p | (-(q) & r)) -> (-(s) | t)`.

Arquivo de entrada

Um dos principais usos de um provador de teoremas como OTTER é: dado como entrada um conjunto de sentenças Ψ, encontrar uma refutação para Ψ, ou seja, tentar provar que Ψ é um conjunto inconsistente. Por outro lado, o escopo deste trabalho, relativo a buscas em LPO, se resume a verificar se dados um conjunto Φ de sentenças e uma sentença α, α é dedutível de Φ, ou seja, se $\Phi \vdash \alpha$.

É sabido que, em LPO[1]:

[1] ver [BELL77]

$\Phi \vdash \alpha \Leftrightarrow$

$\Phi \models \alpha \Leftrightarrow$

$\{\Phi, \neg\alpha\}$ é insatisfatível $\Leftrightarrow$

Existe uma refutação para $\{\Phi, \neg\alpha\}$.

Assim, para determinarmos se $\Phi \vdash \alpha$, basta utlizarmos OTTER para encontrar uma refutação para $\Psi = \{\Phi, \neg\alpha\}$.

Isso é feito, como já mencionado, através de um arquivo de entrada, um arquivo texto, cujo formato é esquematizado logo a seguir, seguido de um exemplo real retratando uma de nossas futuras provas. O conjunto de sentenças $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ e a sentença $\neg\alpha$ são dispostas no espaço reservado para a lista `usable`, que é delimitada pelas assertivas `formula_list(usable)` e `end_of_list`. No início do arquivo são colocadas as opções de comando utilizadas para parametrizar OTTER, tais como escolha de regras de inferência, tempo máximo de busca, etc. Na maioria das provas deste trabalho, utilizamos apenas a opção `set(auto)`, que coloca OTTER no modo autônomo, e a opção `assign(max_seconds,100000)`, que prolonga o tempo máximo de busca para 100000 segundos (aproximadamente 28 horas!). No modo autônomo, o usuário insere um conjunto de cláusulas e/ou sentenças, e OTTER faz uma simples análise sintática e decide quais regras de inferência e estratégias usar.

Esquema de um arquivo de entrada:

```
set(auto).
assign(max_seconds,100000).

formula_list(usable).
```

ϕ_1
ϕ_2
$\ldots$
ϕ_n
$(\neg\alpha)$

```
end_of_list.
```

Exemplo de arquivo de entrada:

```
set(auto).
assign(max_seconds,100000).

formula_list(usable).

all x a (c_representa(x,a) -> classe_especifica(x) & relacao(a)).
all x y ( is_a(x,y) <-> ((classe_especifica(x) & classe_especifica(y) &
                (all v (c_representa(y,v) -> c_representa(x,v))))
```

```
                |
                (classe_generica(x) & classe_generica(y) & (x=y))
                )).
all x y ( (classe_especifica(x) & classe_especifica(y) &
        (all a (c_representa(x,a) <-> c_representa(y,a)))) -> x=y).

- (
all x y ((is_a(x,y) & is_a(y,x)) -> x=y)
).

end_of_list.
```

Arquivo de saída

OTTER envia a maior parte de seu resultado para uma *saída padrão*, a qual é geralmente um arquivo de saída. A primeira parte do arquivo de saída é uma repetição da maior parte do arquivo de entrada e algumas informações adicionais, incluindo números de identificação para cláusulas e descrições de alguns processos de entrada. A segunda parte do arquivo de saída reflete a busca. A parte final lista estatísticas de vários eventos e tempos para várias operações.

Nosso interesse maior é sobre a segunda parte, ou seja, a busca, que mostramos logo abaixo através da prova do exemplo anterior: ao lado esquerdo de cada linha temos um número que serve para identificar a cláusula à sua direita. No meio, entre chaves, uma seqüência que mostra como a presente cláusula foi obtida, a partir de quais outras cláusulas a mesma se originou e quais regras de inferência foram utilizadas para obtê-la. Se esta lista for vazia, significa que a cláusula é uma entrada. Finalmente, à direita, a cláusula propriamente dita. Notem que a última linha mostra, através do $F, a refutação.

Segunda parte do arquivo de saída para o exemplo anterior:

```
...
...
---------------- PROOF ----------------

5 [] -is_a(x,y)|classe_especifica(x)|x=y.
8 [] -is_a(x,y)|classe_especifica(y)|x=y.
11 [] -is_a(x,y)| -c_representa(y,z)|c_representa(x,z)|x=y.
15 [] -classe_especifica(x)| -classe_especifica(y)|c_representa(x,$f2(x,y))|
      c_representa(y,$f2(x,y))|x=y.
16 [] -classe_especifica(x)| -classe_especifica(y)| -c_representa(x,$f2(x,y))|
      -c_representa(y,$f2(x,y))|x=y.
17 [] $c2!=$c1.
23 [] is_a($c2,$c1).
24 [] is_a($c1,$c2).
```

```
25 [hyper,23,8,unit_del,17] classe_especifica($c1).
26 [hyper,23,5,unit_del,17] classe_especifica($c2).
40 [hyper,26,15,25,unit_del,17] c_representa($c1,$f2($c1,$c2))|
     c_representa($c2,$f2($c1,$c2)).
171 [hyper,40,11,23,unit_del,17,factor_simp] c_representa($c2,$f2($c1,$c2)).
172 [hyper,40,11,24,unit_del,17,factor_simp] c_representa($c1,$f2($c1,$c2)).
215 [hyper,172,16,25,26,171,flip.1] $c2=$c1.
217 [binary,215.1,17.1] $F.

------------ end of proof -------------

Search stopped by max_proofs option.

============ end of search ============
...
...
```

5.2 Metodologias de Otimização

5.2.1 Por que usar metodologias

Tínhamos até aqui todo material e recursos necessários para realização das comparações. No entanto, tínhamos dois problemas interdependentes ainda a enfrentar.

O primeiro problema, já comentado anteriormente, era o fato de LPO' (ou LPO) ser um sistema lógico semidecidível com complexidade não polinomial e, mesmo com a utilização de um provador de teoremas otimizado como OTTER, o problema continuava, porém reduzido. Algumas provas ainda levavam dois ou três dias para concluírem e outras nem terminavam neste período.

O segundo problema era o fato das provas que estávamos realizando terem um aspecto *não modular* (ver seção 4.5.2 na página 52). Assim, conforme o processo de comparação avançava, provas realizadas posteriormente demandavam certas alterações nas definições entre as teorias que acabavam invalidando algumas provas realizadas anteriormente que dependiam dessas definições. Essa prática de alteração mostrou ser muito comum e conseqüentemente a necessidade de se realizar a mesma prova diversas vezes era inevitável.

Esses dois problemas tinham um efeito sinérgico negativo sobre o tempo de prova, tornando a comparação entre duas teorias um processo demasiado longo e quase impraticável.

Isso acabou motivando a adoção de algumas práticas que deram origem a metodologias simples que viabilizaram a realização dessas comparações. Essas metodologias serviram, por um lado, para estruturar e organizar o processo de prova, diminuindo assim o problema da não modularidade. E, por outro lado, para diminuir o tempo de cada prova, diminuindo também o problema da complexidade em LPO'. Esses procedimentos apresentavam no entanto a desvan-

tagem de demandar uma maior interferência humana no processo, restringindo a busca de uma total automatização do mesmo.

5.2.2 Metodologia do escopo reduzido

A *metodologia do escopo reduzido* se baseia no seguinte resultado do cálculo de primeira ordem: sejam α uma fórmula em LPO' e ϕ e Φ dois conjuntos de fórmulas em LPO' onde $\phi \subseteq \Phi$. Nessas condições, se $\phi \vdash \alpha$ então $\Phi \vdash \alpha$. Dessa forma a busca da prova $\Phi \vdash \alpha$ se restringe à busca de um subconjunto ϕ de Φ, se possível minimal, tal que $\phi \vdash \alpha$ (pode ocorrer, no entanto, que $\phi = \Phi$).

Já dissemos anteriormente que encontrar uma refutação para o conjunto $\Psi = \{\Phi, \neg\alpha\}$ de fórmulas lógicas em LPO' é um problema de complexidade não polinomial. Isso significa que o tempo para achar uma refutação para Ψ aumenta pelo menos exponencialmente com o *tamanho* de Ψ. O *tamanho* de Ψ aqui pode ser interpretado como sendo o número de conectivos lógicos e quantificadores pertencentes a Ψ. Isso significa que, se pudermos obter um conjunto ϕ *menor* que Φ, o tempo de prova também cairá exponencialmente, o que é muito desejável.

O desafio passou então a ser a escolha do ϕ apropriado, o que não foi muito difícil, apesar desse procedimento ser bastante subjetivo. Isso em parte porque o objetivo não era necessariamente tentar encontrar o ϕ minimal, i.e., o ϕ que contivesse apenas as fórmulas que participariam da prova, mas sim um ϕ suficientemente pequeno que contivesse essas fórmulas e eventualmente outras que não teriam participação naquela.

Com isso os resultados foram excelentes, uma vez que não raro conseguíamos reduzir o tamanho de Φ para ϕ em cerca de 90%, o que significava uma estimativa de queda do tempo de prova em torno de 99,8%! Cabe lembrar no entanto que para cada prova p_i realizada demandava a escolha de um ϕ_i particular.

A outra vantagem dessa metodologia foi sobre a questão da não modularidade. Se estivéssemos utilizando Φ em vez de ϕ para realizar as provas, toda vez que houvesse uma alteração de alguma definição δ, todas as provas p_i feitas anteriormente teriam que ser executadas novamente, uma vez que $\delta \in \Phi$. No caso de utilizarmos ϕ no lugar de Φ, tínhamos que executar apenas aquelas provas p_i em que δ pertencesse ao correspondente ϕ_i. Como conseqüência, isso diminuiu muito o número de provas que tinham que ser refeitas, aliviando portanto o problema da não modularidade.

5.2.3 Metodologia das subprovas (ou da *lematização*)

Em matemática é comum, durante a prova de teoremas mais complexos, provar inicialmente alguns lemas que posteriormente servirão de base para a prova do teorema principal. Cabe à intuição e experiência do matemático escolher apropriadamente quais lemas intermediários ele irá provar.

Usando o mesmo conceito, quando desejamos provar $\phi_i \vdash \alpha$ utilizando um provador de teoremas, podemos antes provar alguns *lemas intermediários*, ou

fórmulas, escolhidos apropriadamente. Este procedimento caracteriza a *metodologia das subprovas*, que tende a diminuir muito o tempo final de prova e minimizar o problema da não modularidade. Estas vantagens são justificadas a seguir.

Como visto na seção 5.1.2, devemos buscar uma refutação para o conjunto $\{\phi_i, \neg\alpha\}$ utilizando um provador de teoremas, caso queiramos provar $\phi_i \vdash \alpha$. Podemos enxergar o funcionamento deste provador como uma geração sucessiva de conjuntos, da seguinte forma: o conjunto inicial θ^0 é o conjunto de fórmulas sobre o qual o provador inicia seu trabalho, ou seja, $\theta^0 = \{\phi_i, \neg\alpha\}$. Para qualquer $s \geq 0$, o conjunto de fórmulas θ^s *dá origem* a um novo conjunto θ^{s+1}, simbolicamente $\theta^s \hookrightarrow \theta^{s+1}$, quando o provador aplica alguma regra de inferência sobre fórmulas de θ^s obtendo novas fórmulas que, somadas à θ^s, compõem o conjunto θ^{s+1}. Como podem existir diversas regras de inferência e cada regra pode ser aplicada a diferentes conjuntos de fórmulas de θ^s, podem existir inúmeros conjuntos subseqüentes θ^{s+1}. Dessa forma, enumerando-os, teremos n possíveis conjuntos subseqüentes $\theta_1^{s+1}, \ldots, \theta_n^{s+1}$ partindo de θ^s. Todo este processo de geração se encerra quando é originado um conjunto θ_i^j possuindo entre suas fórmulas a refutação \$F , caracterizando assim o sucesso da prova, ou quando todos os conjuntos possíveis de serem gerados forem obtidos e não constar em nenhum deles a refutação \$F, caracterizando assim o fracasso da prova.

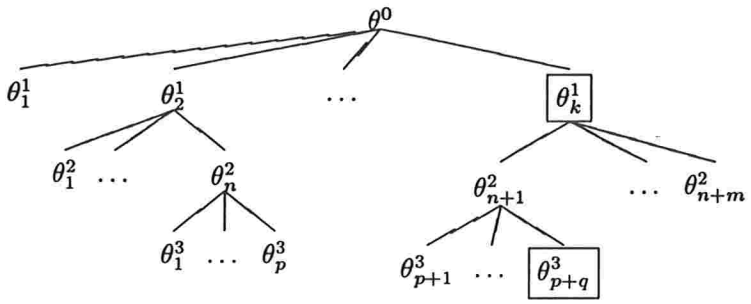


Figura 5.1: Exemplo do funcionamento de um provador de teoremas

Um exemplo de prova com este paradigma é mostrado na figura 5.1. O provador encerra seu trabalho quando gera o conjunto θ_{p+q}^3 que possui a refutação. O caminho desta prova é $\theta^0 \hookrightarrow \theta_k^1 \hookrightarrow \theta_{n+1}^2 \hookrightarrow \theta_{p+q}^3$.

Seja agora a fórmula β um dos lemas intermediários utilizados em alguma dedução particular de α a partir de ϕ_i. Ainda na figura 5.1, suponhamos que ao gerar o conjunto θ_k^1 o provador deu origem a β. Sabemos então que na subárvore abaixo de θ_k^1 existe uma refutação, pois β faz parte de uma dedução de α. Porém o provador desconhece que ele esteja no caminho correto e pode então decidir gerar novos conjuntos em outras ramificações, por exemplo a partir de θ_2^1 ou mesmo de θ^0. Caso não existam outras refutações nesses outros locais, essa decisão acabará por atrasá-lo. Se pudéssemos, no entanto, interferir em seus

próximos passos, poderíamos forçá-lo a esquecer todos as outras ramificações e a seguir somente o caminho a partir de θ_k^1. Estaríamos então, com grandes chances, diminuindo seu tempo de prova.

Na prática, uma maneira aproximada de realizar essa intervenção utilizando um provador de teoremas como OTTER é provar inicialmente $\phi_i \vdash \beta$ e em seguida $\{\phi_i', \beta\} \vdash \alpha$, onde $\phi_i' \subseteq \phi_i$. Uma característica importante dessa metodologia é sua capacidade de composição. Assim, se precisarmos agora provar $\phi_i \vdash \beta$ podemos, utilizando o mesmo raciocínio, achar um lema δ que faça parte da dedução de β, e provarmos $\phi_i \vdash \delta$ e em seguida $\{\phi_i'', \delta\} \vdash \beta$, onde $\phi_i'' \subseteq \phi_i$. A composição de várias dessas subprovas tende a diminuir exponencialmente o tempo final de prova.

Assim, utilizando esta metodologia, uma grande prova pode ser subdividida em um grande número de pequenas provas. Se em algum momento alguma definição entre as teorias for alterada, será necessário executar novamente, dentre essas pequenas provas, apenas aquelas que dependam diretamente dessa definição. Isso portanto diminui também o efeito da não modularidade.

5.2.4 Metodologia das definições em camadas

Como já visto na seção 4.4.4 na página 44, podemos compor várias extensões por definição sem perder suas propriedades. Isso significa, em particular, que em vez de fazermos uma extensão por definição de uma teoria T_1, visando definir diretamente um predicado de uma teoria T_2, podemos, numa primeira extensão por definição de T_1, definir alguns predicados intermediários e posteriormente, numa segunda extensão por definição, definir o predicado alvo de T_2 utilizando esses predicados intermediários e, eventualmente, outros predicados próprios de T_1.

Isso pode ser melhor entendido através do seguinte exemplo: definimos, na seção 7.3.7 na página 94, o predicado `made_of_ast` a partir do predicado `cond_made_of` que, como `made_of_ast`, também representa uma extensão por definição e não pertence originalmente à linguagem de REL_E. Assim, para definirmos o predicado `made_of_ast` em REL_E, precisamos definir primeiramente o predicado `cond_made_of` a partir dos predicados próprios de REL_E. Mostramos abaixo apenas uma parte da definição de `cond_made_of`, por ser nuito longa:

```
% DFN15
all ax y z (cond_made_of(ax,y,z) <->
                (atributo(ax) & classe_especifica(y) &
                ((z=association)| (z=aggregation)) &
                ((z=association) ->
                    (all ay (ar_classe(ay,y) -> ri(ax,ay,normal_1n))) &
                    (all ay ((-ar_classe(ay,y)) -> (-ri(ax,ay,normal_1n)))) &
                    (all ay (atri_raiz(ay,y) -> (- ri(ay,ax,normal_1n)) &
                                                (- ri(ax,ay,normal_11)) )) &
                    .....
```

.....
.....

Em seguida, definimos o predicado `made_of_ast` a partir do predicado `cond_made_of` e outros predicados de REL_E:

```
% DFN14
all x y z (made_of_ast(x,z,y) <->
                (classe_especifica(x) & classe_especifica(y) &
                (exists ax (atri_raiz(ax,x) & cond_made_of(ax,y,z))))
        ).
```

Este procedimento foi adotado porque, sendo a definição de `cond_made_of` um tanto complexa e na ausência deste predicado, toda esta complexidade estaria presente diretamente na definição de `made_of_ast`. A vantagem desse procedimento é que algumas provas que dependiam da definição de `made_of_ast` não dependiam dos complexos detalhes da definição de `cond_made_of`, ou seja, a definição de `cond_made_of` não precisava constar dos ϕ_i's dessas provas, o que diminuia muito seus tempos de resposta, reduzindo o problema da complexidade do algorítmo de prova. Dois exemplos dessas provas são mostrados a seguir. Notem que na lista de cada um dos ϕ_i's dessas provas consta apenas a definição de `made_of_ast` (DFN14), não aparecendo a definição de `cond_made_of` (DFN15).

```
% PROOF X19 (Axioma RR29)(DEP PROOF X06, DFN14, DFN6d, DFN6e, DFN16)
all x y z (made_of_ast(x,z,y) -> made_of(x,y)).

% PROOF X20 (Axioma RR30)(PROOF X02, DFN6a, PROOF X11.07, DFN14, DFN16)
all x y z (made_of(x,y) & is_a(z,x) -> made_of(z,y)).
```

Parte II

Um estudo de caso

Capítulo 6

Modelos Estudados

6.1 Introdução aos Modelos Rock & Roll e Relacional

Nosso trabalho, a partir deste ponto, basicamente gira em torno da axiomatização e da comparação de dois modelos de dados: o primeiro, o modelo OO denominado Rock & Roll, como descrito em [FERNAN95], e o segundo, o modelo relacional, como descrito em [CODD70].

Este capítulo tem como objetivo descrever estes dois modelos e suas correspondentes axiomatizações. A apresentação dessas axiomatizações seguirá os três primeiros passos do método de Bourbaki, tal como descrito na seção 3.3.2.

6.2 Modelo RR (Rock & Roll)

6.2.1 Breve introdução ao modelo do sistema Rock & Roll

Utilização de um modelo derivado

Na realidade utilizamos, para nossa análise, um modelo derivado do modelo OO do sistema Rock & Roll. Este modelo derivado é, ao mesmo tempo, uma restrição, uma simplificação e uma correção do modelo original. Ele é uma restrição do modelo original, uma vez que somente consideramos o componente estrutural deste último, deixando de lado as características comportamentais[1] do mesmo, tais como operações e métodos. É uma simplificação de alguns conceitos originais com o objetivo de diminuir a complexidade das futuras comparações e provas, facilitar o entendimento por parte do leitor e impedir que o trabalho se tornasse muito extenso. E, por último, é uma correção, pois ao menos um dos axiomas teve que ser alterado, uma vez que estava no modelo original concebido erroneamente. O axioma original era:

[1] *behaviour em [FERNAN95]*

Axioma RR34.

$$\forall x\, y\, y'\, y''\, z\, z'\, z''\, (y \not\approx z \wedge is\_a(x,y) \wedge is\_a(x,z) \\ \wedge made\_of\_ast(y,y',y'') \wedge made\_of\_ast(z,z',z'') \\ \rightarrow y' \approx z' \wedge y'' \approx z'').$$

O axioma acima informalmente deveria dizer que se uma classe x é subclasse de duas classes diferentes y e z, então y e z, caso fossem classes compostas, teriam composições iguais. Mas na realidade diz também outra coisa, que y e z só podem ser compostas de uma única classe. Por exemplo, seja y equivalente à classe `livro` e z equivalente à classe `manual`. Caso `livro` seja composta por agregação das classes `nome` e `autor`, ou `made_of_ast(livro,aggregation,nome)` e `made_of_ast(livro,aggregation,autor)`, então queremos que `manual` também seja composta dessas mesmas classes, ou seja, `made_of_ast(manual,aggregation,nome)` e `made_of_ast(manual,aggregation,autor)`. Do axioma acima, chegamos à conclusão errônea de que `nome` = `autor`. O axioma corrigido e correto é exposto abaixo.

Axioma RR34.

$$\forall x\, y\, z\, w\, v\, (y \not\approx z \wedge is\_a(x,y) \wedge is\_a(x,z) \\ \rightarrow (made\_of\_ast(y,w,v) \leftrightarrow made\_of\_ast(z,w,v))).$$

A existência de poucos erros como este na tese em questão, no entanto, mostra a boa qualidade da axiomatização descrita na mesma, devido à grande propensão a erros inerente a este processo.

Em suma, descreveremos aqui já o modelo derivado. Informações completas sobre o modelo original do sistema Rock & Roll podem ser obtidas em [FERNAN95].

Para facilidade de compreensão, vamos designar de modelo RR_o o modelo original e de modelo RR o modelo derivado.

6.2.2 *Passo 1* - conjuntos base existentes

Inicialmente definimos os dois tipos básicos de conjuntos: **classes** e **objects**. **Classes** têm o principal propósito conceitual de modelar coleções de objetos, ou **objects**, de acordo com suas características estruturais (e comportamentais) compartilhadas. Como exemplo, a classe (**class**) `integer` modela a coleção dos números inteiros, a qual pertence o objeto (**object**) `1`. Dizemos também que o objeto `1` é uma *instância* da classe `integer`. Da mesma forma, à **class** `livro` pertence, por exemplo, o **object** `!1` (Neste trabalho, um símbolo, cujo primeiro caractere é '!', denota um *símbolo de entidade*[2] em algum domínio de aplicação, e não deve ser confundido com um elemento de qualquer dos tradicionais tipos de dados, tais como inteiros ou caracteres).

[2] *entity token* em [FERNAN95]

Predicados: `class(x)` e `object(y)`.

Cada um desses dois tipos básicos é particionado em dois subconjuntos, qualificados como **specific** (no sentido de ser dependente de uma aplicação específica) e **generic** (no sentido de ser independente de qualquer aplicação específica). Daí, os seguintes quatro tipos são definidos: **specific_classes** e **generic_classes**, e **specific_objects** e **generic_objects**.

Por exemplo, `livro`, `livraria` denotam elementos de **specific_classes** , enquanto `integer` e `string` denotam elementos de **generic_classes**. `!1`, `!9` denotam elementos de **specific_objects**, enquanto `1` e `'Fundamentos de LPO'` denotam elementos de **generic_objects**.

Elementos de um tipo específico não são interpretáveis a priori, isto é, eles não são conhecidos fora dos limites de um domínio de aplicação particular. Ao contrário, elementos de um tipo genérico são conhecidos e entendidos fora dos limites de qualquer domínio de aplicação.

Tradicionalmente, os **generic_objects** são comumente chamados de *valores primitivos (atômicos)*, ou seja, pertencentes a *tipos de dados primitivos (atômicos)* (como são conhecidas as **generic_classes**). Da mesma forma, **specific_objects** são também conhecidos como *objetos complexos* no paradigma de orientação a objetos.

Predicados:

`specific_class(x)`, `generic_class(y)`, `specific_object(z)` e `generic_object(w)`.

Classes são organizadas em hierarquias de herança baseadas num relacionamento de inclusão sobre sua extensão, isto é, o conjunto de todas as suas instâncias. Por exemplo, a classe `livro_de_bolso` é uma subclasse da classe `livro`, ou, informalmente, um `livro_de_bolso` é um `livro`. Isto implica que toda instância de `livro_de_bolso` é também uma instância de `livro`.

Além disso, construtores que modelam objetos estruturalmente são de dois tipos. No primeiro tipo, um objeto é estruturalmente modelado por *propriedades* que são *atribuídas* às classes das quais o objeto é uma instância. Por exemplo, `título_do_livro` nomeia (a extensão de) uma propriedade que é uma atributo das instâncias da classe `book`. No segundo tipo, um objeto complexo (isto é, uma instância de uma classe composta) é modelado pelos *componentes* de sua *construção*. Por exemplo, um relacionamento tal como `publicado_por`, entre um `autor` e uma `editora`, pode ser representado por uma classe composta construída por agregação das classes relacionadas. Assim, cada instância de `publicado_por` terá em sua construção um par ordenado ligando um autor e uma editora. Por exemplo, a instância `!101` de `publicado_por` poderia ser aquela que liga a instância de `autor` cujo nome é `'J.W. Lloyd'` à instância de `editora` cujo nome é `'Springer Verlag'`. Além das agregações (que são estruturas de registros), consideraremos também neste modelo as associações (que são estruturas de conjuntos). Assim, precisaremos de um conjunto extra, além das classes e dos objetos, para transmitir a natureza precisa do relacionamento de composição entre classes. Mais especificamente, símbolos são necessários

para indicar se a composição é por *agregação* ou por *associação*. A semântica associada a cada um desses *modos de composição* é diferente. Esses símbolos são os chamados **composition_modes**.

Predicado: `composition_mode(x)`.

Nas seções a seguir formalizamos com mais detalhes todos esses relacionamentos entre os conjuntos até aqui introduzidos.

6.2.3 *Passo 2* - definição da caracterização típica $\mathcal{T}(\mathcal{E})$, ou os relacionamentos entre esses conjuntos

1. **inclusão**, a qual denota uma relação binária em **classes**, dá a noção de *superclasse* e *subclasse* de uma dada classe. Como exemplo: um `livro_de_bolso` é um tipo de `livro`, ou seja, a classe `livro_de_bolso` é uma subclasse da classe `livro`, ou, equivalentemente, `livro` é uma superclasse de `livro_de_bolso`, um `artigo` é um tipo de `publicação`, etc.

   Predicado: `is_a(x,y)` (onde `class(x)` e `class(y)`).

2. **atribuição**, que denota uma relação binária entre **specific_classes** e **classes**, dá a noção de propriedades descritivas ou *atributos*, possuídos por todos objetos de uma classe. Por exemplo, uma `editora` tem um `nome`, um `endereço`, um `catálogo`, etc.

   Predicado: `has(x,y)` (onde `specific_class(x)` e `class(y)`).

3. **composição**, que denota uma relação binária em **specific_classes**, dá a noção de *classes compostas*, ou seja, classes cujas instâncias são objetos construídos a partir de outros objetos. Por exemplo, uma instância de `livros_emprestados` é construída como uma coleção, ou *conjunto*, de instâncias de `livros`, uma instância de `endereço` é construída como uma tripla, ou *agregação*, consistindo de um `local`, uma `cidade` e um `CEP`.

   Predicado: `made_of(x,y)`
   (onde `specific_class(x)` e `specific_class(y)`).

4. **classificação**, denota uma relação binária entre **specific_objects** e **specific_classes** , dá a noção de um objeto particular sendo uma instância de uma ou mais classes, dependendo da estrutura (e comportamento) deste objeto. Por exemplo, o objeto denotado por `!9` é uma instância da classe `livraria`, a qual o relaciona estruturalmente a instâncias de atributos, tais como `estoque` e `endereço`.

   Predicado: `instance_of(x,y)`
   (onde `specific_object(x)` e `specific_class(y)`).

5. **descrição**, denota uma relação ternária entre **specific_objects**, **classes** e **objects**, dá a noção de valores para as propriedades, ou atributos, de um objeto. Por exemplo, a instância `!1` de `livro` tem o valor `'Fundamentos de LPO'` para sua propriedade `nome_do_livro`.

   Predicado: `property(x,y,z)`
   (onde `specific_object(x)`, `class(y)` e `object(z)`).

6. **construção**, que denota uma relação ternária entre **specific_objects**, **indexicals** e **specific_objects**, dá a noção de elementos de construção de objetos complexos. Por exemplo, uma instância de `comitê_editorial` refere-se a (daí, é construído como) uma coleção de objetos, cada um dos quais uma instância de `autor`.

   Predicado: `component(x,y,z)`
   (onde `specific_object(x)`, `indexical(y)` e `specific_object(z)`).

7. **tipificação**[3] , denota uma relação binária entre **generic_objects** e **generic_classes**, e é análogo à **classificação**, no caso de objetos independentes de aplicação e classes.

   Predicado: `of_type(x,y)`
   (onde `generic_object(x)` e `generic_class(y)`).

Outros relacionamentos entre conjuntos

Os relacionamentos **atribuição** e **composição** abrangem toda informação do esquema que é específico ao domínio sendo modelado. No entanto, eles mascaram a distinção entre a informação que é explicitamente declarada e a informação implícita que é obtida por inferência. Esta distinção é essencial para que possamos aplicar restrições à herança estrutural.

Para diferenciarmos entre informação declarada e inferida, faremos uso de dois novos relacionamentos: **atribuição*** e **composição***, correspondentes, respectivamente, aos relacionamentos básicos **atribuição** e **composição**.

1. **atribuição***, difere de **atribuição** no sentido que o primeiro somente relaciona uma classe com suas propriedades atribuídas, enquanto o segundo adicionalmente relaciona uma classe com suas propriedades herdadas.

   Por exemplo, suponha que uma `editora_científica` é subclasse de `editora`. Suponha também que, a uma `editora` é atribuída, entre outras, a propriedade `nome` e que, a uma `editora_científica` é atribuída, entre outras, a propriedade `especialidade` (ex: `medicina`, `matemática`, etc.). Então, o par `<editora_científica,nome>` está na extensão de **inclusão**, mas não na de **inclusão***. Ou seja, `nome` é uma propriedade herdada por, ao contrário de atribuída a, `editora_científica`. Ao contrário, o par

[3] *typing* em [FERNAN95]

<editora_científica,especialidade> está na extensão de **inclusão***, pois especialidade foi atribuída a editora_científica, e também está na extensão de **inclusão**, pois toda propriedade atribuída a uma classe é, trivialmente, herdada por esta classe.

Predicado: has_ast(x,y) (onde specific_class(x) e class(y)).

2. **composição***, difere de **composição** ao ter um terceiro argumento que determina o modo de composição que existe entre duas classes, a saber, os modos de associação e de agregação. Além disso, analogamente ao item anterior, **composição** estende-se sobre informações inferidas e declaradas, enquanto **composição*** somente se estende sobre o último.

   Por exemplo, associação modela coleções de objetos homogêneos: uma biblioteca é uma associação de seções. E agregação modela coleções de tuplas: um evento empréstimo é uma agregação de data_saída, leitor e data_retorno.

   Predicado: made_of_ast(x,y,z)
   (onde specific_class(x), composition_mode(y) e specific_class(z)).

Resumindo, os axiomas que compõem a caracterização típica $\mathcal{T}(\mathcal{E})$ são:

Axiomas RR1 até RR9. O conjunto de axiomas de $\mathcal{T}(\mathcal{E})$.

```
all x y (is_a(x,y) -> class(x) & class(y)).
all x y (has_ast(x,y) -> specific_class(x) & class(y)).
all x y (has(x,y) -> specific_class(x) & class(y)).
all x y u (made_of_ast(x,y,u) ->
          specific_class(x) & composition_mode(y) & specific_class(u)).
all x y (made_of(x,y) ->
         specific_class(x) & specific_class(y)).
all x y (instance_of(x,y) -> specific_object(x) & specific_class(y)).
all x y (of_type(x,y) -> generic_object(x) & generic_class(y)).
all x y u (property(x,y,u) -> specific_object(x) & class(y) & object(u)).
all x y u (component(x,y,u) ->
          specific_object(x) & indexical(y) & specific_object(u)).
```

6.2.4 *Passo 3* - definição do sistema axiomático $\mathcal{S}(\mathcal{E})$, ou o que é dito sobre os conjuntos e seus relacionamentos

Sobre os conjuntos

Axioma RR10 até RR13. Um indivíduo no domínio de aplicação deve pertencer a exatamente um conjunto. Isto significa que o presente modelo não é uniforme. Mais precisamente, todos os conjuntos são dois a dois disjuntos.

```
all x (specific_class(x) -> (-generic_class(x)) & (-specific_object(x)) &
      (-generic_object(x)) & (-composition_mode(x)) ).

all x (generic_class(x) -> (-specific_object(x)) & (-generic_object(x)) &
      (-composition_mode(x)) ).

all x (specific_object(x)) -> (-generic_object(x)) & (-composition_mode(x)) ).

all x (generic_object(x)) -> (-composition_mode(x)) ).
```

Axiomas RR14, RR15 e RR16. Existe um conjunto **classes** (respectivamente, **objects**; **indexicals**) que é a união de **generic_classes** e **specific_classes** (respectivamente, **generic_objects** e **specific_objects**; **specific_classes** e **specific_objects**).

```
all x ( class(x) <-> (generic_class(x) | specific_class(x) )).
all x ( object(x) <-> (generic_object(x) | specific_object(x) )).
all x ( indexical(x) <-> (specific_class(x) | specific_object(x) )).
```

Axioma RR17. Não há outros modos de composição senão `associação` e `agregação`.

```
all x (composition_mode(x) -> (x=association | x=aggregation)).
```

Sobre *inclusão* (predicado `is_a`)

Axiomas RR18 e RR19. Num relacionamento de **inclusão** somente uma classe específica pode ser incluída numa classe específica e somente uma classe genérica pode ser incluída numa classe genérica. Mais precisamente, **inclusão** relaciona elementos no mesmo subconjunto de seu domínio.

```
all x y ((is_a(x,y) & specific_class(y)) -> specific_class(x)).
all x y ((is_a(x,y) & generic_class(y)) -> generic_class(x)).
```

Axioma RR20. Se uma classe é subclasse de outra classe, e a segunda também é subclasse da primeira, então as duas classes são iguais. Mais precisamente, **inclusão** é antissimétrica.

```
all x y ((is_a(x,y) & is_a(y,x)) -> x=y).
```

Axiomas RR21 e RR22. Toda classe é uma subclasse dela própria. Também, uma classe tem como superclasses todas as superclasses de suas próprias superclasses. Mais precisamente, **inclusão** é reflexiva sobre o conjunto **classes** e transitiva.

```
all x ( class(x) -> is_a(x,x)).
all x y z ( is_a(x,y) & is_a(y,z) -> is_a(x,z)).
```

Axioma RR23. Em configurações de herança múltipla, classes distintas possuindo uma subclasse comum devem, elas próprias, possuirem uma superclasse comum. Mais precisamente, **inclusão** é fracamente direcionada.

```
all x y z ((is_a(x,y) & is_a(x,z) & ( - (y=z)))
                    -> ( exists w (is_a(y,w) & is_a(z,w)))).
```

Sobre *classificação* e *tipificação* (predicados `instance_of` e `of_type`)

Axiomas RR24 e RR25. Todo objeto deve pertencer a alguma classe. Mais especificamente, todo elemento de **specific_objects** é relacionado por **classificação** a algum elemento de **specific_classes** e todo elemento de **generic_objects** é relacionado por **tipificação** a algum elemento de **generic_classes**.

```
all x (specific_object(x) -> ( exists y (instance_of(x,y)))).
all x (generic_object(x) -> ( exists y (of_type(x,y)))).
```

Axioma RR26. Toda instância de uma classe é também uma instância de suas superclasses. Mais precisamente, **classificação** propaga-se através da **inclusão**.

```
all x y z ((instance_of(x,y) & is_a(y,z)) -> instance_of(x,z)).
```

Sobre *atribuição* (predicados `has_ast` e `has`)

Axioma RR27. Toda propriedade atribuída a uma classe é uma propriedade desta classe. Mais precisamente, **atribuição*** é englobada por **atribuição**, daí distinguindo-se propriedades atribuídas das propriedades herdadas.

```
all x y ( has_ast(x,y) -> has(x,y)).
```

Axioma RR28. Toda propriedade de uma classe é herdada por suas subclasses. Mais precisamente, **atribuição** propaga-se através da **inclusão**.

```
all x y z ( has(x,y) & is_a(z,x) -> has(z,y)).
```

Sobre made_of_ast e made_of

Axioma RR29. Se uma classe é composta de alguma classe componente por algum modo de composição, então ela é composta daquela classe. Mais precisamente, **composição** é induzida a partir de **composição*** descartando-se o modo de composição.

```
all x y z ( made_of_ast(x,y,z) -> made_of(x,z)).
```

Axioma RR30. A composição de uma classe é herdada por suas subclasses. Mais precisamente, **composição** propaga-se através da **inclusão**.

```
all x y z ( made_of(x,y) & is_a(z,x) -> made_of(z,y)).
```

Axioma RR31. Se duas classes são relacionadas por **composição***, então elas devem ser assim relacionadas por exatamente um modo de composição. Mais precisamente, **composição*** determina um relacionamento funcional do primeiro e terceiro argumentos para o segundo.

```
all x u v z (made_of_ast(x,u,z) & made_of_ast(x,v,z) -> (u=v)).
```

Axioma RR32. Instâncias de uma classe que seja uma **composição*** por `associação` devem ser homogêneas com respeito aos tipos de seus componentes. Mais precisamente, **composição*** por `associação` determina um relacionamento funcional do primeiro e segundo argumentos para o terceiro.

```
all x y u v (made_of_ast(x,y,u) & (y=association) &
             made_of_ast(x,y,v) -> (u=v)).
```

Axioma RR33. Composição é uma relação assimétrica. Uma consequência disso é que uma classe não pode ser composta dela mesma.

```
all x y (made_of(x,y) -> (- made_of(y,x))).
```

Axioma RR34. Numa configuração de herança múltipla, uma subclasse comum não deve herdar classes componentes distintas nem ter modos de composição conflitantes. Como exemplo, se `coleção_temática` e `antologia` têm `publicação_especializada` como uma subclasse comum então, se elas são composições, elas devem ser feitas dos mesmos componentes e da mesma forma. por exemplo, uma `associação` de `artigos`.

```
all x y z w v
    ((y!=z) & is_a(x,y) & is_a(x,z) ->
    (made_of_ast(y,w,v) <-> made_of_ast(z,w,v))).
```

Sobre *descrição* (predicado `property`)

Axioma RR35 e RR36. Cada instância de uma classe que tem uma propriedade deve referir-se a uma instância da classe que nomeia a propriedade. Como exemplo, se um `livro` tem como atributo um `nome_de_livro`, então a instância de `livro` denotada por `!109` deve referir-se a uma instância de `nome_de_livro`, por exemplo, `'O Velho e o Mar'`.

```
all a x y ((instance_of(a,x) & has(x,y) & specific_class(y)) ->
          (exists b (property(a,y,b) & instance_of(b,y)))).
all a x y ((instance_of(a,x) & has(x,y) & generic_class(y)) ->
          (exists b (property(a,y,b) & of_type(b,y)))).
```

Sobre *construção* (predicado `component`)

Axioma RR37. Cada instância de uma classe que é composta por outra por `associação` somente se refere, em sua construção, a instâncias da classe componente com um índice neutro, ou seja, que não induz qualquer ordem. Como exemplo, se uma `biblioteca` é uma `associação` de `seções`, então se a `biblioteca` denotada por `!451` faz referência aos objetos `!562` e `!673` como seus elementos de construção, então `!562` e `!673` devem denotar `seções` e as referências a eles devem ter índices `!562` e `!673` para os objetos de denotação idêntica.

```
all a b bl x y z ((instance_of(a,x) & made_of_ast(x,y,z) & (y=association) &
                  component(a,bl,b)) -> (instance_of(b,z) & (bl=b))).
```

Axioma RR38. Cada instância de uma classe que é composta por outra por `agregação` deve referir-se a uma instância de cada uma de suas classes componentes. Cada uma de tais referências deve ser indexada pelo nome da classe componente, daí induzindo uma noção de coordenada. Como exemplo, se um `empréstimo` é uma `agregação` de `data` e `leitor`, então se `!514` denota um `empréstimo`, ele faz referência a objetos tais como `010799` e `!847`, como seus elementos de construção, onde `010799` e `!847` denotam, e são indexados por, uma `data` e um `leitor`, respectivamente.

```
all a x y z ((instance_of(a,x) & made_of_ast(x,y,z) & (y=aggregation)) ->
                  (exists b (component(a,z,b) & instance_of(b,z)))).
```

6.3 Modelo Relacional

6.3.1 Introdução

A axiomatização do modelo relacional apresentada nesta seção é baseada fundamentalmente no modelo relacional descrito por Codd em [CODD70]. Adicionalmente, utilizamos em nosso modelo um conceito bastante simplificado de restrição de integridade, muito comum até nos menos sofisticados BD's relacionais comerciais (tais como Microsoft Access, Inprise Paradox, etc.).

Vale notar também que a presente axiomatização do modelo relacional foi toda desenvolvida para este trabalho pelo seu autor, em vez de ter sido concebida a partir de alguma axiomatização pré-existente. Uma das razões para isso é que, na realidade, precisávamos de uma axiomatização do *metamodelo* relacional, para que pudéssemos viabilizar as futuras comparações. Assim, por exemplo, enquanto na bibliografia em geral cada relação é representada na forma de um predicado, em nossa teoria cada relação é representada como sendo um conjunto. Veja o exemplo a seguir.

Seja a relação `livro`, cujos atributos são `nome` e `autor`, e que possua a tupla `<'Dom Casmurro', 'M. Assis'>`. Em geral, na bibliografia encontraremos uma axiomatização que represente a relação acima de uma forma semelhante à descrita abaixo:

```
livro(.nome='Dom Casmurro',.autor='M. Assis').
```

Em nosso modelo relacional, ou metamodelo, a mesma relação teria a seguinte representação:

```
relacao(livro).
atributo(nome).
atributo(autor).
tem_ast(livro,nome).
tem_ast(livro,autor).
tupla(ta).
instancia_de(ta,livro).
propriedade(ta,nome,'Dom Casmurro').
propriedade(ta,autor,'M. Assis').
```

6.3.2 *Passo 1* - conjuntos base existentes

Da mesma forma que em [CODD70], o termo **relação** é definido aqui em seu sentido matemático. Dados os conjuntos (ou **tipos**) $S_1, S_2, ...S_n$ (não necessariamente distintos), R é uma relação nesses n conjuntos se R é um conjunto de n-**tuplas**, cada uma das quais tendo seu primeiro elemento de S_1, seu segundo elemento de S_2, e assim sucessivamente.

Pensando em uma representação matricial para uma relação, esta deverá ter as seguintes propriedades:

1. Cada linha representa uma tupla de R.
2. A ordem dessas linhas é irrelevante.
3. Todas as linhas são distintas.
4. A ordem das colunas é semanticamente relevante - ela corresponde à ordem $S_1, S_2, ...S_n$ dos domínios nos quais R é definida.

A propriedade 4 acima torna-se inconveniente quando o número de colunas, ou grau da relação, torna-se muito grande. Portanto, é desejável que não trabalhemos com relações que tenham domínios ordenados, mas sim com relações[4] cujos domínios não sejam ordenados.

Para se conseguir isso, os domínios devem ser unicamente identificáveis, ao menos dentro de cada relação. Utilizaremos assim o conceito de **atributo**, para que forneça esta identificação única. Ou seja, um mesmo atributo só pode aparecer uma única vez numa relação. Forçaremos ainda, sem perder expressividade, que um mesmo atributo não se repita em mais de uma relação. Cada atributo é associado a um tipo (ou conjunto) S. É possível, no entanto, que diversos atributos estejam associados a um mesmo tipo.

[4]Sendo matematicamente rigoroso, o termo correto aqui seria *relacionamento*, que é uma classe de equivalência daquelas relações que são equivalentes sob a permutação de domínios. Neste trabalho, porém, não distinguiremos entre os termos relação e relacionamento.

Cada instância pertencente a um tipo qualquer é um elemento atômico, tal como um número inteiro ou uma cadeia de caracteres. A este elemento atômico damos o nome de **objeto_genérico**.

Finalmente, temos as restrições de integridade, ou simplesmente ri's, que são dependências funcionais entre tuplas de duas relações. Para a conceituação das ri's, que detalharemos melhor ao fim deste capítulo, precisamos de um novo conjunto para podermos distinguir entre alguns de seus tipos diferentes, no que concerne ao seu aspecto funcional. Aos elementos desse conjunto denominamos **modos de restrição de integridade**, ou simplesmente `modo_ri`'s.

Sumarizando então, apresentamos os predicados unários que representam os conjuntos básicos de nosso modelo relacional:

```
relacao(x).
tipo(y).
tupla(w).
atributo(z).
objeto_generico(v).
modo_ri(k).
```

6.3.3 *Passo 2* - definição da caracterização típica $\mathcal{T}(\mathcal{E})$, ou os relacionamentos entre esses conjuntos

1. **atribuição**, que denota uma relação binária entre **relações** e **atributos**, dá a noção de propriedades descritivas ou *atributos*, possuídos por todas as tuplas de uma relação. Por exemplo, um `livro` tem um `nome`, um `autor`, etc.

   Predicado: `tem_ast(x,y)` (onde `relacao(x)` e `atributo(y)`).

2. **associação**, denota uma relação binária entre **atributos** e **tipos**, dá a noção de atributos associados a tipos ou domínios. Esta associação define e restringe o domínio de cada atributo. Por exemplo, o atributo `numero_de_paginas` tem como domínio o tipo `inteiro`.

   Predicado: `tem_tipo(x,y)` (onde `atributo(x)` e `tipo(y)`).

3. **tipificação**, denota uma relação binária entre **objetos_genéricos** e **tipos**, dá a noção de um objeto particular sendo uma instância de um determinado tipo. Por exemplo, a cadeia de caracteres `'M. Assis'` pertence ao tipo `string`.

   Predicado: `de_tipo(x,y)` (onde `objeto_generico(x)` e `tipo(y)`).

4. **extensão**, denota uma relação binária entre **tuplas** e **relações**, dá a noção de uma tupla particular pertencendo à extensão de uma determinada relação. Por exemplo, a tupla denotada por `!5` é uma instância da relação `manuais`.

Predicado: `instancia_de(x,y)` onde `tupla(x)` e `relacao(y)`).

5. **descrição**, denota uma relação ternária entre **tuplas**, **atributos** e **objetos_genéricos**, dá a noção de valores para as propriedades de uma tupla. Por exemplo, a tupla `!1`, que é uma instância da relação `editora`, tem o valor `'J. Wiley'` para o atributo (de `editora`) `nome_da_editora`.

   Predicado: `propriedade(x,y,z)`
   (onde `tupla(x)`, `atributo(y)` e `objeto_generico(z)`).

6. **restrição**, denota uma relação ternária entre **atributo**, **atributo** e **modo_ri**, dá a noção de dependência funcional entre os valores de dois atributos. A definição desta dependência é fornecida por **modo_ri**. Por exemplo...

   Predicado: `ri(x,y,z)`
   (onde `atributo(x)`, `atributo(y)` e `modo_ri(z)`).

Resumindo, os axiomas que compõem a caracterização típica $\mathcal{T}(\mathcal{E})$ são:

Axiomas REL1 até REL6. O conjunto de axiomas de $\mathcal{T}(\mathcal{E})$.

```
all x y (tem_ast(x,y) -> relacao(x) & atributo(y)).
all x y (tem_tipo(x,y) -> atributo(x) & tipo(y)).
all x y (de_tipo(x,y) -> objeto_generico(x) & tipo(y)).
all x y (instancia_de(x,y) -> tupla(x) & relacao(y)).
all x y z (propriedade(x,y,z) -> tupla(x) & atributo(y) & objeto_generico(z)).
all x y z (ri(x,y,z) -> atributo(x) & atributo(y) & modo_ri(z)).
```

6.3.4 *Passo 3* - definição do sistema axiomático $\mathcal{S}(\mathcal{E})$, ou o que é dito sobre os conjuntos e seus relacionamentos

Sobre os conjuntos

Axioma REL7 até REL11. Um indivíduo no domínio de aplicação deve pertencer a exatamente um conjunto. Isto significa que o presente modelo não é uniforme. Mais precisamente, todos os conjuntos são dois a dois disjuntos.

```
all x (relacao(x) ->  (-tipo(x)) &  (-objeto_generico(x)) & (-atributo(x)) &
      (-tupla(x)) & (-modo_ri(x)) ).

all x (tipo(x) -> (-objeto_generico(x)) & (-atributo(x)) & (-tupla(x)) &
      (-modo_ri(x)) ).

all x (objeto_generico(x) -> (-atributo(x)) & (-tupla(x)) & (-modo_ri(x)) ).

all x (atributo(x) -> (-tupla(x)) & (-modo_ri(x)) ).

all x (tupla(x) -> (-modo_ri(x)) ).
```

Sobre *tipificação* (predicado `de_tipo`)

Axiomas REL12 e REL13. Todo objeto genérico pertence a um e somente um tipo. Mais precisamente, **tipificação** representa uma função do conjunto dos **objetos_genericos** para o conjunto dos **tipos**.

```
all x (objeto_generico(x) -> (exists y (de_tipo(x,y)))).
all x y z (de_tipo(x,y) & de_tipo(x,z) -> (y=z)).
```

Sobre *atribuição* (predicado `tem_ast`)

Axioma REL14. Toda relação tem ao menos um atributo. Mais precisamente, todo elemento de **relações** é relacionado por **atribuição** a algum elemento de **atributos**.

```
all x (relacao(x) -> (exists y (tem_ast(x,y)))).
```

Axiomas REL15 e REL16. Todo atributo pertence a uma e somente uma relação. Mais precisamente, **atribuição** representa uma função do conjunto dos **atributos** para o conjunto das **relações**.

```
all x (atributo(x) -> (exists y (tem_ast(y,x)))).
all x y z (tem_ast(x,y) & tem_ast(z,y) -> (x=z)).
```

Sobre *associação* (predicado `tem_tipo`)

Axioma REL17 e REL18. Todo atributo tem um e somente um tipo. Mais precisamente, **associação** representa uma função do conjunto dos **atributos** para o conjunto dos **tipos**.

```
all x (atributo(x) -> (exists y (tem_tipo(x,y)))).
all x y z (tem_tipo(x,y) & tem_tipo(x,z) -> (y=z)).
```

Sobre *extensão* (predicado `instancia_de`)

Axioma REL19 e REL20. Toda tupla pertence a uma e somente uma relação. Mais precisamente, **extensão** representa uma função do conjunto das **tuplas** para o conjunto das **relações**.

```
all x (tupla(x) -> (exists y (instancia_de(x,y)))).
all x y z (instancia_de(x,y) & instancia_de(x,z) -> (y=z)).
```

Sobre *descrição* (predicado `propriedade`)

Axioma REL21. Toda instância de uma relação que tem uma propriedade, ou atributo, deve referir-se a um objeto do mesmo tipo daquele que nomeia esta propriedade. Ou, em outras palavras, se uma tupla a pertence a uma relação x, e x tem um atributo y, então a tupla a possui um valor para o atributo y, cujo tipo é igual ao do atributo y.

```
all a x y
(       (instancia_de(a,x) & tem_ast(x,y)) ->
        (exists b z (tipo(z) & tem_tipo(y,z) & de_tipo(b,z) &
        propriedade(a,y,b)))
).
```

Axioma REL22. Toda tupla possui um valor único para cada atributo. Mais precisamente, **descrição** determina um relacionamento funcional do primeiro e segundo argumentos para o terceiro.

```
all a b c d (propriedade(a,b,c) & propriedade(a,b,d) -> (c=d)).
```

Axioma REL23. Se uma tupla possui um valor qualquer para um determinado atributo, então esta tupla deve ser uma instância de uma relação que tem este atributo.

```
all a b c ra rb (propriedade(a,b,c) & instancia_de(a,ra) & tem_ast(rb,b) ->
        (ra=rb)).
```

Sobre *restrições de Integridade* (predicado `ri`)

Os axiomas a seguir descrevem as restrições de integridade que definimos para nosso modelo relacional. A restrição de integridade é representada pelo predicado `ri(x,y,z)`, onde `x`, que denominamos pai, e `y`, filho, são dois atributos quaisquer, e `z` é o modo, ou tipo, de restrição que desejamos representar.

Axioma REL24. Os modos, ou tipos, de restrição de integridade são representados pelo predicado unário `modo_ri`. Existe um conjunto finito de modos de `ri`.

```
all x (modo_ri(x) -> (x=normal_11) | (x=normal_1n)).
```

Axioma REL25. O axioma abaixo representa a definição de chave candidata. Um atributo `x` é uma chave candidata (`chave_cand(x)`) sse dada uma tupla qualquer pertencente à mesma relação `r` de `x`, o valor da propriedade desta tupla em `x` é único, ou seja, ele não se repete em `x` para nenhuma outra tupla pertencente a `r`. Notem que não introduzimos em *REL* a noção de *chave primária*. Na verdade a *chave primária* de uma relação seria uma das chaves candidatas desta relação, escolhida arbitrariamente.

```
all x (chave_cand(x) <-> atributo(x) &
        (all a b c d (propriedade(a,x,c) & propriedade(b,x,d)
        & (a!=b) -> (c!=d))) ).
```

Axioma REL26. O atributo pai numa `ri` tem que ser uma chave candidata.

```
all x y z (ri(x,y,z) -> chave_cand(x)).
```

Axioma REL27. Seja a restrição `ri(x,y,z)`. Caso exista uma tupla a com um determinado valor `c` no atributo filho `y`, deverá então existir uma tupla `al`, cujo valor no atributo pai `x` seja igual a `c`. Informalmente, não pode existir uma tupla filho sem uma tupla pai correspondente.

```
all a y c x z
(ri(x,y,z) & propriedade(a,y,c)  -> (exists al (propriedade(al,x,c)))).
```

Axioma REL28. Na mesma linha do axioma anterior, numa ri, cujo modo seja 1 p/ 1, para cada tupla pai existe no máximo uma tupla filho correspondente.

```
all x y a c b d z
     (ri(x,y,z) & propriedade(a,x,c) & propriedade(b,y,c) &
      propriedade(d,y,c) & (z=normal_11) -> b=d ).
```

Axioma REL29. Pode somente haver um `modo_ri` entre dois atributos numa `ri`.

```
all x1 y1 z1 x2 y2 z2
(ri(x1,y1,z1) & ri(x2,y2,z2) & (x1=x2) & (y1=y2) -> (z1=z2)).
```

Capítulo 7

Comparação de modelos: provando RR a partir de REL

7.1 Introdução

Infelizmente, como era possível de se esperar, não conseguimos provar RR a partir de REL sem que fizéssemos primeiro uma extensão não conservativa deste último. Isto porque talvez seja improvável que haja uma extensão conservativa de REL que deduza RR . De fato, é difícil imaginarmos que possamos provar RR sem ao menos estendermos REL através da inclusão de algumas instâncias fixas de alguns de seus conjuntos. Assim, p. ex., como proposto por alguns autores, seria natural querermos estender REL através da inclusão de uma relação binária fixa chamada `is_a` onde a mesma retratasse a relação de **inclusão** entre duas classes. A inclusão desta relação, no entanto, já caracterizaria uma extensão não conservativa de REL, uma vez que a existência de uma relação (formalmente: $\exists x(relação(x))$) seria dedutível logicamente desta nova teoria, o que não é verdade em REL. Não estamos com isso argumentando que uma extensão conservativa não exista, mas sim que a concepção de tal extensão deva ser muito difícil e até mesmo injustificável, devido à pouca expressividade de REL e à grande diferença entre os paradigmas de REL e RR.

Isto posto, nossa missão tornou-se a de encontrar uma extensão não conservativa que alterasse o mínimo possível a estrutura do modelo original. Neste sentido, a solução proposta acima de se incluir algumas instâncias fixas de conjuntos de REL parecia ser a mais razoável. No entanto, da maneira que REL foi concebido, retratando o metamodelo relacional, a inclusão de instâncias fixas seria feita através da inclusão de um grande número de novos axiomas, fatos atômicos, etc. E assim o esforço demandado no processo de comparação dos modelos, e conseqüentemente o tamanho desta dissertação, seriam seguramente

o dobro, afastando este trabalho de seu escopo principal. E para piorar a situação, não sabíamos de antemão quais instâncias fixas deveriam constar nesta extensão, o que dificultaria ainda mais nossa análise.

Sendo assim optamos por acrescentar verdadeiramente alguns conceitos novos à REL e tentar realizar as provas. Após um longo processo de tentativas e erros, finalmente conseguimos realizá-las, basicamente ao acrescentarmos dois novos conjuntos ao modelo relacional, um primeiro que retratava um conjunto de relações, que denominamos **classe_específica**, e um segundo que retratava um conjunto de tuplas de diferentes relações que denominamos **objeto_específico**. E esta é exatamente a solução exposta neste capítulo e até onde nós conseguimos alcançar.

O passo seguinte seria, aí sim, tentar definir todos os conceitos introduzidos por estes dois novos conjuntos a partir dos conjuntos e axiomas originais de REL, seja utilizando-se instâncias fixas ou fazendo uso de algum outro recurso. Uma outra solução que emerge a partir da conclusão da necessidade desses dois novos conjuntos, é a utilização de um modelo relacional um pouco mais expressivo que REL, mas ainda de acordo com [CODD70], que contivesse, por exemplo, conjuntos que representassem *views* ou *queries* (consultas) sobre esquemas de BD. Isso porque poderíamos pensar numa consulta como um conjunto de relações, e no resultado de algumas dessas consultas como conjuntos de tuplas de diferentes relações, que é exatamente o que necessitamos. Isso demonstra que o processo de comparação é iterativo e suscetível a constantes refinamentos, e que o estágio que conseguimos atingir é válido e é parte importante deste processo. Além disso, mostra também que o uso de uma estrutura teórica e metodológica tal como a montada no presente trabalho é fundamental para esta análise.

Mas independente desses futuros refinamentos, os resultados obtidos até aqui podem ser considerados por si só satisfatórios, uma vez que os objetivos propostos no início do trabalho, ou seja, de enfrentar todas as dificuldades inerentes ao processo de comparação e a partir daí fundamentar uma teoria e uma metodologia que possibilitasse o exercício do mesmo dentro de uma base formal, foram atingidos (ver seção 2.11). Também quanto à comparação propriamente dita, conseguimos definir um conjunto de novos conceitos que devam ser acrescentados a REL para que este expresse o modelo RR. Esses conceitos poderiam ser implementados através de alguns recursos de programação e acoplados ao modelo relacional. Talvez para alguns isso já poderia ser uma solução suficiente às suas necessidades particulares.

A este modelo obtido a partir da extensão não conservativa de REL, denominaremos REL_E. Esta extensão será feita através da alteração e inclusão de algumas sentenças em REL. Posteriormente buscaremos uma interpretação (não fiel) da teoria de RR para a teoria de REL_E, seguindo o proposto na seção 4.5.2.

7.2 REL_E - uma extensão não conservativa de REL

A seguir introduzimos as sentenças que deverão ser alteradas e acrescentadas às já existentes em REL para compor REL_E. Identificaremos os novos axiomas com o prefixo **PRP** (de *proposição*), apenas para podermos distinguí-los dos axiomas de REL, já identificados com o prefixo **REL**. Para simplificar a descrição dessas proposições, adiantaremos aqui o enunciado de algumas definições que fazem parte, na realidade, da extensão conservativa de REL_E descrita na seção 7.3 logo a seguir. Em todas as definições utilizaremos o prefixo **DFN**. Finalmente introduziremos também dois fatos atômicos, identificados com o prefixo **FAT**.

7.2.1 Os novos conjuntos base de REL_E

Para a prova do modelo RR a partir do modelo REL_E vamos necessitar de dois novos conjuntos com características de segunda ordem em REL. Assim, necessitamos em REL_E de um conjunto que represente conjuntos *especiais* de relações, e de um segundo conjunto que represente conjuntos *especiais* de tuplas de diferentes relações (uma vez que o conjunto **relação** já representa, ele próprio, um conjunto de tuplas). O termo *especial* neste contexto significa que não estamos interessados em quaisquer conjuntos de relações (ou de tuplas), e sim em conjuntos de relações (ou tuplas) com características bem definidas, as quais serão declaradas mais tarde. Ao primeiro conjunto daremos o nome de **classe_específica**, e ao segundo o nome de **objeto_específico**.

Predicados: `classe_especifica(x)` e `objeto_especifico(y)`.

7.2.2 Definição de $\mathcal{T}(\mathcal{E})$ de REL_E

O conjunto **classe_específica** representa um conjunto de relações. Precisamos portanto de um novo predicado que represente uma relação binária de pertinência entre elementos do conjunto de relações e elementos do conjunto de classes específicas. A este predicado denominaremos de **c_representa**.

Predicado: `c_representa(x,y)` (onde `classe_especifica(x)` e `relacao(y)`).

Analogamente a **classe_específica**, precisamos também de um predicado que represente uma relação binária de pertinência entre elementos do conjunto de tuplas e elementos do conjunto de objetos específicos. A este predicado denominaremos de **o_representa**.

Predicado: `o_representa(x,y)` (onde `objeto_especifico(x)` e `tupla(y)`).

Assim introduzimos os novos axiomas da caracterização típica $\mathcal{T}(\mathcal{E})$ para os predicados **c_representa** e **o_representa**:

```
% PRP1
all x a (c_representa(x,a) -> classe_especifica(x) & relacao(a)).

% PRP2
all x tx (o_representa(x,tx) -> objeto_especifico(x) & tupla(tx)).
```

7.2.3 Definição de $\mathcal{S}(\mathcal{E})$ de REL_E

Axiomas de REL alterados

Vamos agora aumentar o número de modos de restrição de integridade de dois para quatro. Os dois novos modos são: `isa_11` e `has_11`. Na verdade estes modos têm exatamente a mesma semântica do modo `normal_11` de REL. Porém sua utilidade está no fato de podermos classificar os relacionamentos em quatro conjuntos distintos. Informalmente, os modos `isa_11`, `has_11`, `normal_11` e `normal_1n` auxiliarão mais adiante na caracterização dos predicados de RR `is_a`, `has_ast`, `made_of_ast` com modo de composição `aggregation` e `made_of_ast` com modo de composição `association`, respectivamente.

Assim, os axiomas **REL24** e **REL28** de REL devem ser alterados, com o intuito de considerar esses novos modos de `ri`:

Axioma REL24*.

```
all x (modo_ri(x) ->     (x=normal_11) | (x=normal_1n) |
                         (x=isa_11) | (x=has_11) ).
```

Axioma REL28*.

```
all x y a c b d z
(ri(x,y,z) & propriedade(a,x,c) & propriedade(b,y,c)
           & propriedade(d,y,c)
           & ((z=normal_11) | (z=isa_11) | (z=has_11)) -> b=d ).
```

Axiomas adicionados a REL

Mencionamos que as relações que compõem cada **classe_específica** possuem características *especiais*, o mesmo ocorrendo com os conjuntos de tuplas representados por **objetos_específicos**. A maioria dos axiomas que devem ser adicionados aos já existentes em REL para se obter REL_E formalizam exatamente estas características especiais.

Juntamente aos axiomas REL7 a REL11, que definem que em REL os conjuntos são dois a dois disjuntos, devem ser acrescentados dois novos axiomas (PRP3 e PRP4) que garantem que os novos conjuntos **classe_específica** e **objeto_específico** também sejam disjuntos entre si e relativamente aos conjuntos pré-existentes:

```
% PRP3
all x (
```

```
classe_especifica(x) -> (-objeto_especifico(x)) & (-relacao(x))
        & (-tipo(x)) &  (-objeto_generico(x)) & (-atributo(x))
        & (-tupla(x)) & (-modo_ri(x))).

% PRP4
all x (
objeto_especifico(x) -> (-relacao(x)) & (-tipo(x))
        & (-objeto_generico(x)) & (-atributo(x)) & (-tupla(x))
        & (-modo_ri(x))).
```

Definição de raiz

Vamos agora definir o que é uma **raiz**:

Definição de ri_1n Existe um relacionamento **ri_1n** de um relação x para uma relação y sse existir uma **ri** (restrição de integridade) com **modo_ri** `isa_11` de um atributo de x para um atributo de y (ver **DFN1** abaixo).

Definição de raiz r é uma **raiz** sse r for uma relação em que não exista nenhum relacionamento **ri_1n** de qualquer relação y para r (ver **DFN2** abaixo).

```
% DFN1
all x y (ri_1n(x,y) <->
        (exists ax ay (tem_ast(x,ax) & tem_ast(y,ay)
        & ri(ax,ay,isa_11)))).

% DFN2
all x (raiz(x) <-> relacao(x) & (all t ((- ri_1n(t,x)))) ).
```

Como já exposto anteriormente, se x é uma **classe_específica**, x representa, ou **c_representa**, um certo número n de relações. Vamos impor que $n > 0$. Temos dois casos a analisar:

1. **Se** $n = 1$
   A única relação w que x representa, ou **c_representa**, deve ser uma **raiz**. Neste caso, definimos que x representa unicamente w, ou `rep_unic(x,w)` (ver **DFN3** abaixo).

   ```
   % DFN3
   all x w (rep_unic(x,w) <-> classe_especifica(x) & raiz(w)
           & c_representa(x,w) &
           (all v (c_representa(x,v) -> (v=w) ))).
   ```

2. **Se** $n > 1$
   Dentre as n relações que x c_representa, uma delas deve ser uma raiz. Seja w esta raiz. Para as demais $n-1$ relações, existe um relacionamento **ri_1n** de w para cada uma delas. Portanto, pela própria definição de raiz, essas $n-1$ relações não podem, por sua vez, serem raízes.

Das considerações acima, deduzimos que se x é uma **classe_específica**, então x c_representa uma e uma única raiz. (ver **PRP5** abaixo)

```
% PRP5
all x (classe_especifica(x) -> (exists w (raiz(w)
            & c_representa(x,w) &
            (all v (raiz(v) & c_representa(x,v) -> (v=w) )))) ).
```

Para concluir esta seção, impomos inversamente que se w for uma raiz, então existe ao menos uma **classe_específica** x que a c_representa unicamente. (ver **PRP6** abaixo). E que duas classes específicas são iguais sse c_representarem exatamente as mesmas relações (ver **PRP7** abaixo).

```
% PRP6
all w (raiz(w) -> (exists x (classe_especifica(x) & rep_unic(x,w)))).

% PRP7
all x y ( (classe_especifica(x) & classe_especifica(y) &
       (all a (c_representa(x,a) <-> c_representa(y,a)))) -> x=y).
```

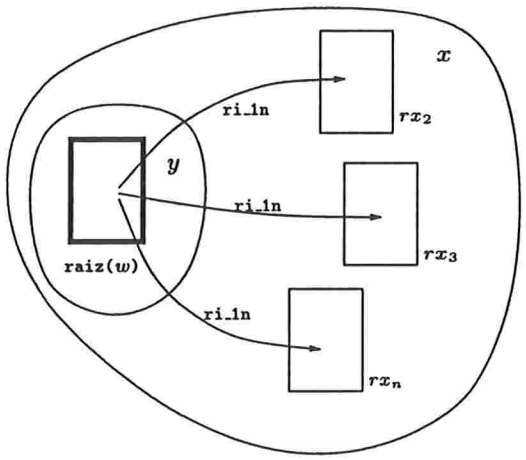


Figura 7.1: Exemplo dos predicados rep_unic e c_representa.

atri_raiz e ar_classe

Vimos acima que numa **classe_específica** x, quando $n > 1$, existe um relacionamento **ri_1n** entre a raiz w c_representada por x e cada uma das demais $n-1$ relações rx_i c_representadas por x. Ou seja, existe uma **ri** com modo isa_11 entre um atributo da raiz w e um atributo de rx_i, para todo i, tal que $2 \leq i \leq n$. Vamos impor adicionalmente que essas ri's se originem sempre de um mesmo atributo aw da raiz w. Nessas condições definimos o atributo aw

como sendo um *atributo raiz* da classe específica x, ou atri_raiz(aw,x) (ver **DFN4** e **PRP8** mais adiante). Observem adicionalmente que aw tem que ser uma chave candidata, ou chave_cand(aw) segundo o axioma **REL26**.

```
% DFN4
all aw x (
atri_raiz(aw,x) <-> atributo(aw) & chave_cand(aw) &
                classe_especifica(x) &
                (exists rx (raiz(rx) &
                tem_ast(rx,aw) & c_representa(x,rx) &
                (all r (relacao(r) & (r!=rx) & c_representa(x,r) ->
                        (exists ar ((ar!=aw) & atributo(ar) &
                        tem_ast(r,ar) & ri(aw,ar,isa_11))
                )))
                )) ).

% PRP8
all x (classe_especifica(x) ->
                (exists aw rx (atributo(aw) & raiz(rx) &
                tem_ast(rx,aw) & c_representa(x,rx) &
                atri_raiz(aw,x))) ).
```

Queremos também que exista apenas um atributo a_{rx_i} em cada relação rx_i para o qual se destinam essas ri's. Dizemos então que cada um desses a_{rx_i} é um *atributo de restrição* da classe específica x, ou simplesmente ar_classe(a_{rx_i},x). Para completar, estabelecemos que se aw é um *atributo raiz* de x então aw é também um *atributo de restrição* de x, ou se atri_raiz(aw,x) então ar_classe(aw,x) (ver **DFN5** abaixo).

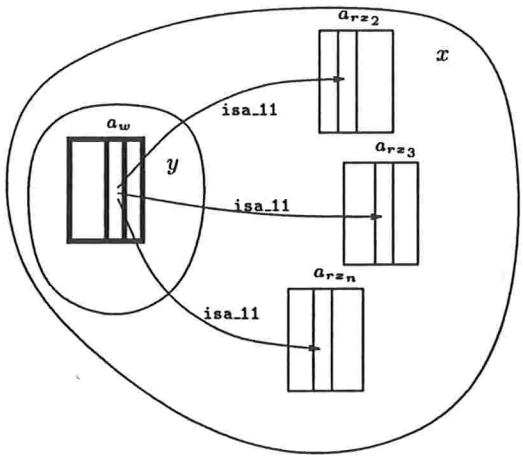
```
% DFN5
all ax x (
ar_classe(ax,x) <-> atributo(ax) & classe_especifica(x) &
                (atri_raiz(ax,x)
                |
                (exists ay ry rx (
                atributo(ay) & raiz(ry) & relacao(rx) &
                tem_ast(ry,ay) & c_representa(x,ry) &
                tem_ast(rx,ax) & c_representa(x,rx) &
                (ry!=rx) & (ay!=ax) &
                atri_raiz(ay,x) &
                ri(ay,ax,isa_11)))
                ) ).
```

Para o caso de $n = 1$, como visto anteriormente, a única relação c_representada pela **classe_específica** x deve ser uma raiz. Conseqüentemente, pode não haver ri's com modo isa_11 desta raiz para outras relações. Assumiremos, assim, que esta raiz deva possuir apenas um atributo aw como sendo uma chave

candidata, ou chave_cand(aw). E que aw seja o único atributo raiz de x, ou atri_raiz(aw,x), e, conseqüentemente, ar_classe(aw,x) (ver **DFN4** e **PRP8** acima).

Em resumo, desejamos garantir que haja, para qualquer $n > 0$, um e somente um atributo por relação c_representada por alguma **classe_específica**, que seja **ar_classe** desta **classe_específica** (ver **PRP9**).

```
% PRP9
all x y z r (
        classe_especifica(x) & atributo(y) & atributo(z) &
        relacao(r) & tem_ast(r,y) & tem_ast(r,z) & c_representa(x,r) &
        ar_classe(y,x) & ar_classe(z,x) -> (y=z)).
```



chave_cand(a_w)

atri_raiz(a_w,x) → ar_classe(a_w,x)

ar_classe(a_{rx_2},x)

.....

ar_classe(a_{rx_n},x)

atri_raiz(a_w,y) → ar_classe(a_w,y)

Figura 7.2: Exemplo dos predicados atri_raiz e ar_classe.

Sobre os objetos_específicos

Analogamente à **classe_específica**, um **objeto_específico** é um conjunto especial de tuplas. Informalmente, um **objeto_específico** deve representar, ou o_representar, uma tupla em toda relação c_representada por alguma classe_específica x.

Para formalizar isso adequadamente precisamos de alguns conceitos ou axiomas. Porém, para o escopo de nossas futuras provas, dois desses axiomas são suficientes:

1. Dados uma **classe_específica** x e um **objeto_genérico** c. Se para todo atributo aw pertencente a alguma relação c_representada por x , tal que ar_classe(aw,x), existir uma tupla tx tal que a propriedade de tx em aw

seja c, ou `propriedade(tx,aw,c)`, então existe um **objeto_específico** b que o_representa todas essas tuplas (ver **PRP10** abaixo).

2. Um **objeto_específico** o_representa ao menos uma tupla de alguma raiz, mas não pode o_representar tuplas de duas raízes diferentes (ver **PRP11** abaixo).

```
% PRP10
all x c (
(classe_especifica(x) & objeto_generico(c) &
(all ax (atributo(ax) & ar_classe(ax,x) ->
        (exists tx (tupla(tx) & propriedade(tx,ax,c))))))
        ->
        (exists b (objeto_especifico(b) &
        (all ax tx (atributo(ax) & tupla(tx) & ar_classe(ax,x) &
                propriedade(tx,ax,c) -> o_representa(b,tx))))) ).

% PRP11
all x (objeto_especifico(x) ->
                (exists tx r (raiz(r) & tupla(tx) &
                instancia_de(tx,r) & o_representa(x,tx) &
                        (all ty ry (raiz(ry) & tupla(ty) &
                        instancia_de(ty,ry) & o_representa(x,ty) ->
                        (ry=r)
                        ))
                )) ).
```

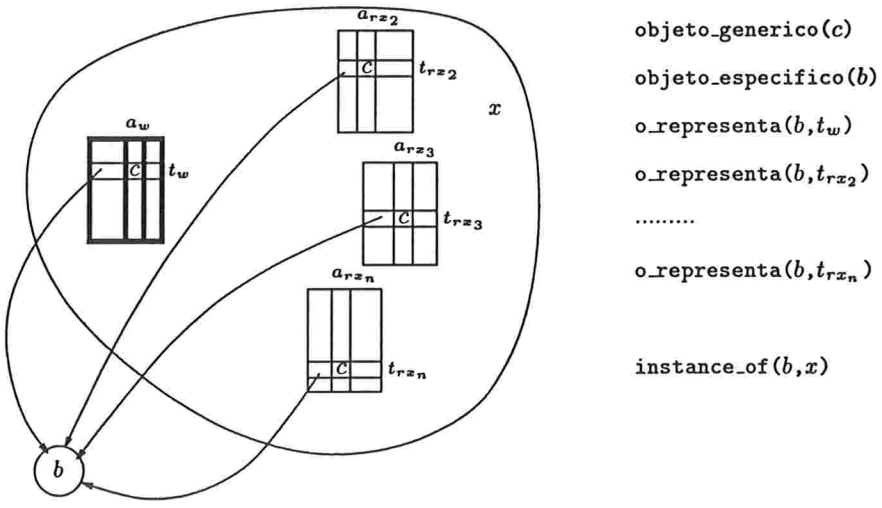


Figura 7.3: Exemplo dos predicados o_representa e instance_of.

associacao e agregacao

Para finalizar a extensão não conservativa de REL, iremos precisar de duas constantes em REL_E para podermos definir futuramente as constantes `association` e `agreggation` de RR. Para este fim vamos reservar *quaisquer* dois elementos do conjunto **objeto_generico**, p.ex., os elementos `associacao` e `agregacao` (ver **FAT1** e **FAT2**).

```
% FAT1
objeto_generico(associacao).

% FAT2
objeto_generico(agregacao).
```

A única condição que impomos é que estes dois elementos sejam escolhidos de tal forma que nunca apareçam em nenhuma tupla de nenhuma relação (ver **PRP12** e **PRP13**).

```
% PRP12
all tx ax (-propriedade(tx,ax,associacao)).

% PRP13
all tx ax (-propriedade(tx,ax,agregacao)).
```

7.3 Definições entre linguagens

Nesta seção apresentamos uma extensão conservativa por definição através de uma composição de definições a partir da teoria T_{REL_E}, originando assim uma nova teoria T'_{REL_E}, cuja linguagem contém a linguagem de T_{RR}. A composição de definições $\Delta = \{\delta_1, \ldots, \delta_n\}$ desenvolvida aqui retrata exatamente a construção descrita na seção 4.5.2 (página 50), cujo objetivo é, portanto, o de demonstrar que T_{REL_E} é tão poderosa quanto T_{RR}.

A seguir descrevemos as definições que compõem Δ.

7.3.1 Definições de apoio a REL_E

As primeiras definições introduzidas são as definições **DFN1** até **DFN5**, já descritas na seção anterior.

7.3.2 Definição dos conjuntos de RR

As definições dos conjuntos de RR (classes, objetos, etc.) falam por si mesmas. Vale notar, porém, que definimos implicitamente o conjunto **generic_class** a partir da união dos conjuntos **atributo** e **tipo** (através da definição intemediária de **classe_genérica**). Notem também que definimos o conjunto **generic_object** como sendo igual ao conjunto **objeto_generico** a menos de dois de seus elementos, a saber, os elementos `associacao` e `agregacao`. Isto porque

queremos reservar estes dois elementos para a posterior definição do conjunto **composition_mode** e garantir a disjunção entre este último e o conjunto **generic_object**.

```
% DFN6a
all x (specific_class(x) <-> classe_especifica(x)).
% DFN6b
all x (generic_class(x) <-> classe_generica(x)).
% DFN6c
all x (classe_generica(x) <-> (atributo(x) | tipo(x))).
% DFN6d
all x (classe(x) <-> (classe_especifica(x) | classe_generica(x))).
% DFN6e
all x (class(x) <-> classe(x)).
% DFN6f
all x (generic_object(x) <-> objeto_generico(x)
                        & (x!=associacao) & (x!=agregacao)).
% DFN6g
all x (specific_object(x) <-> objeto_especifico(x)).
% DFN6h
all x (objeto(x) <-> (objeto_especifico(x) | generic_object(x))).
% DFN6i
all x (object(x) <-> objeto(x)).
```

7.3.3 Predicado *is_a*

Dividimos a definição do predicado **is_a** em duas situações. **Is_a** é um relacionamento entre duas **classes_específicas** ou entre duas **classes_genéricas**.

No primeiro caso dizemos que a **classe_específica** x é uma (**is_a**) **classe_específica** y sse todas as relações c_representadas por y forem também c_representadas por x. No segundo caso, para simplificar nossas provas, definimos que a **classe_genérica** x é uma (**is_a**) **classe_genérica** y sse elas simplesmente forem iguais. Esta simplificação no entanto tem seu preço. Com a definição DFN7 abaixo, é fácil verificar que a teoria T'_{REL_E} construída até o momento deduz logicamente a sentença pertencente à linguagem de T_{RR}

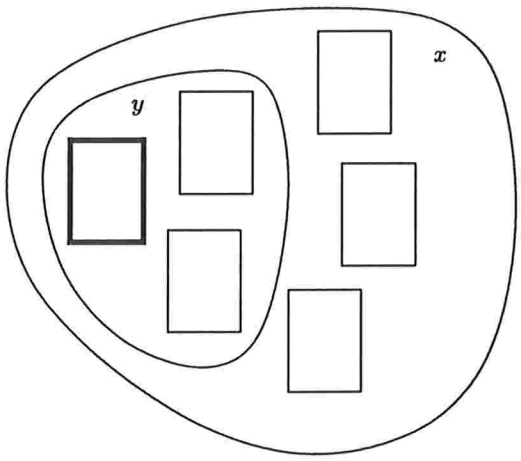
$$\forall xy(generic\_class(x) \wedge generic\_class(y) \wedge is\_a(x,y) \rightarrow x \approx y).$$

Por outro lado, é também fácil verificar que esta sentença não é dedutível de T_{RR}. Isto portanto é suficiente para provar, como contra-exemplo, que a interpretação π de T_{RR} para T_{REL_E}, que estamos implicitamente construindo através das definições Δ, não é fiel[1].

```
% DFN7
```

[1] ver (4.43) na página 52 para maiores detalhes.

```
all x y ( is_a(x,y) <-> ((classe_especifica(x) & classe_especifica(y) &
                (all v (c_representa(y,v) -> c_representa(x,v))))
                |
                (classe_generica(x) & classe_generica(y) & (x=y))
                )).
```



is_a(x,y)

(c_representa(y,r) → c_representa(x,r))

Figura 7.4: Exemplo do predicado is_a(x,y), quando x e y são **classes_específicas**.

7.3.4 Predicados *instance_of* e *of_type*

instance_of

Teremos x como sendo uma instância de y, ou instance_of(x,y), sse as duas condições abaixo forem satisfeitas (ver figura 7.3):

1. x é um **objeto_específico** e y é uma **classe_específica**.
2. Existe um **objeto_genérico** c, tal que para todo atributo ay, onde ar_classe(ay,y), existe uma tupla ty, cujo valor em ay é c, ou propriedade(ty,ay,c), e x o_representa ty.

```
% DFN8
all x y
(instance_of(x,y) <-> (
        objeto_especifico(x) & classe_especifica(y) &
        (exists c (objeto_generico(c) &
                (all ay (atributo(ay) & ar_classe(ay,y) ->
                        (exists ty (tupla(ty) &
                        propriedade(ty,ay,c) & o_representa(x,ty)))
```

```
                    ))
          ))
)).
```

of_type

O termo x será do tipo y, ou `of_type(x,y)`, sse as condições abaixo forem satisfeitas:

1. x é um **generic_object**, tal como descrito na definição `DFN6f` da seção 7.3.2.
2. Uma das seguintes condições for satisfeita:
   (a) y é um atributo e existe um tipo t tal que y tenha tipo t e x pertença a t.
   (b) y é um tipo e x pertence a y.

```
% DFN9
all x y (of_type(x,y) <-> ((generic_object(x) & atributo(y) &
                        (exists t (tipo(t) & tem_tipo(y,t) & de_tipo(x,t))))
                        |
                        (generic_object(x) & tipo(y) & de_tipo(x,y)))
                        ).
```

7.3.5 Predicados *has_ast* e *has*

has_ast

O termo x tem a propriedade atribuída y, ou `has_ast(x,y)`, sse *um* dos conjuntos de condições abaixo for satisfeito:

1. Primeiro conjunto:
   (a) x e y são ambas **classes_específicas**.
   (b) Existe uma **relação** rx, c_representada por x, que possui um atributo ax. E, para todo atributo ay, tal que `ar_classe(ay,y)`, existe uma ri com modo `has_11` de ay para ax.
2. Segundo conjunto:
   (a) x é uma **classe_específica** e y é um **atributo**.
   (b) Existe uma **relação** rx, c_representada por x, que possui o atributo y.

```
% DFN10
all x y
(has_ast(x,y) <-> (classe_especifica(x) & classe_especifica(y) &
```

```
                (exists rx ax (relacao(rx) & atributo(ax) &
                c_representa(x,rx) & tem_ast(rx,ax) &
                        (all ay (ar_classe(ay,y) -> ri(ay,ax,has_11)))
                )))
                |
                (classe_especifica(x) & atributo(y) &
                (exists rx (relacao(rx) & c_representa(x,rx) & tem_ast(rx,y))))
).
```

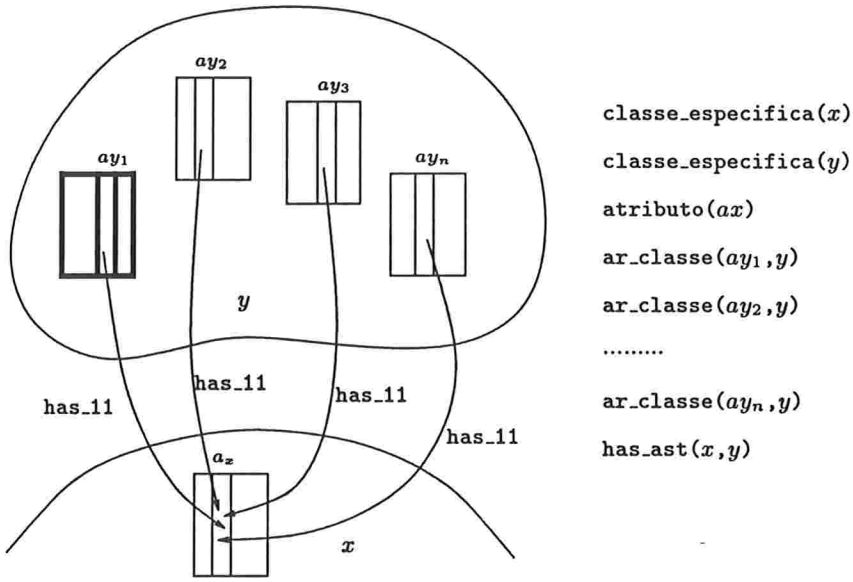


Figura 7.5: Exemplo do predicado has_ast(x,y), quando y é uma **classe_específica**.

has

O termo x tem a propriedade herdada y, ou has(x,y), sse existir uma **classe_específica** v, tal que v tenha a propriedade atribuída y, ou has_ast(v,y), e x seja uma subclasse de v, ou is_a(x,v).

```
% DFN11
all x y (has(x,y) <-> (exists v (has_ast(v,y) & is_a(x,v)))).
```

7.3.6 Constantes *association* e *aggregation*

Precisamos definir neste momento as constantes association e aggregation da linguagem de *RR*. Elas farão parte das definições dos predicados da próxima seção. Para definirmos essas constantes, utilizaremos o formato proposto na

seção 4.19 (página 42) que, como já verificado, garante que tenhamos uma extensão conservativa por definição. Para este fim utilizaremos os elementos associacao e agregacao pertencentes ao conjunto **objeto_generico** da linguagem de REL_E. Assim obtemos as duas sentenças abaixo.

```
% DFN12
all x (x=association <-> x=associacao).

% DFN13
all x (x=aggregation <-> x=agregacao).
```

7.3.7 Predicados *made_of_ast*, *made_of* e *composition_mode*

A definição para o predicado **made_of_ast** é a mais complexa. Por isso, dividimos a mesma em duas subdefinições. Na primeira definimos **made_of_ast** em função de um predicado ternário chamado **cond_made_of**. E na segunda definimos o que é o relacionamento **cond_made_of** propriamente dito. Ao final da seção definimos os predicados **made_of** e **composition_mode**.

made_of_ast

O termo x é composto por y através do modo de composição z, ou `made_of_ast(x,z,y)`, sse x e y forem ambas **classes_específicas** e existir um atributo ax, tal que ax seja um atributo raiz de x, ou `atri_raiz(ax,x)`, e houver um relacionamento **cond_made_of** entre ax, y e z, ou `cond_made_of(ax,y,z)`.

```
% DFN14
all x y z (made_of_ast(x,z,y) <->
                (classe_especifica(x) & classe_especifica(y) &
                (exists ax (atri_raiz(ax,x) & cond_made_of(ax,y,z))))
        ).
```

cond_made_of

Existe um relacionamento `cond_made_of(ax,y,z)` sse:

1. ax é um **atributo** e y é uma **classe_específica**.
2. z pode assumir dois valores, a saber, `association` ou `aggregation`.
3. no caso de z ser igual a `association`, as seguintes condições devem ser satisfeitas:
   (a) Para todo atributo ay, tal que `ar_classe(ay,y)`, existe uma ri com modo `normal_1n` de ax para ay.
   (b) Para todo atributo ay, em que não seja verdade que `ar_classe(ay,y)`, não pode existir uma ri com modo `normal_1n` de ax para ay.

(c) Para todo atributo *ay*, tal que `atri_raiz(ay,y)`, não pode existir uma ri com modo `normal_1n` de *ay* para *ax*, nem existir uma ri com modo `normal_11` de *ax* para *ay*.

(d) Para todo atributo *ay* e toda **classe_específica** *yl*, tal que `atri_raiz(ay,yl)`, não pode existir uma ri com modo `normal_11` de *ay* para *ax*.

4. no caso de *z* ser igual a `aggregation`, as seguintes condições devem ser satisfeitas:

(a) Para todo atributo *ay*, tal que `ar_classe(ay,y)`, existe uma ri com modo `normal_11` de *ay* para *ax*.

(b) Para todo atributo *ay*, tal que `atri_raiz(ay,y)`, não pode existir uma ri com modo `normal_11` de *ax* para *ay*, nem existir uma ri com modo `normal_1n` de *ay* para *ax*.

(c) Para todo atributo *ay* e toda **classe_específica** *yl*, tal que `atri_raiz(ay,yl)`, não pode existir uma ri com modo `normal_1n` de *ax* para *ay*.

```
% DFN15
all ax y z (cond_made_of(ax,y,z) <->
                (atributo(ax) & classe_especifica(y) &
                ((z=association)| (z=aggregation)) &
                ((z=association) ->
                        (all ay (ar_classe(ay,y) -> ri(ax,ay,normal_1n))) &
                        (all ay ((-ar_classe(ay,y)) -> (-ri(ax,ay,normal_1n)))) &
                        (all ay (atri_raiz(ay,y) -> (- ri(ay,ax,normal_1n)) &
                                                    (- ri(ax,ay,normal_11)) )) &
                        (all ay yl
                                (atri_raiz(ay,yl) -> (- ri(ay,ax,normal_11)) ))
                )
                &
                ((z=aggregation) ->
                        (all ay (ar_classe(ay,y) -> ri(ay,ax,normal_11))) &
                        (all ay (atri_raiz(ay,y) -> (- ri(ax,ay,normal_11)) &
                                                    (- ri(ay,ax,normal_1n)) )) &
                        (all ay yl
                                (atri_raiz(ay,yl) -> (- ri(ax,ay,normal_1n)) ))
                )
        )).
```

made_of

O termo *x* é composto por *y*, ou `made_of(x,y)`, sse existir uma **classe_específica** *v*, tal que *v* seja composta por *y* através de um modo de composição z, ou `made_of_ast(v,z,y)`, e *x* seja subclasse de *v*, ou `is_a(x,v)`.

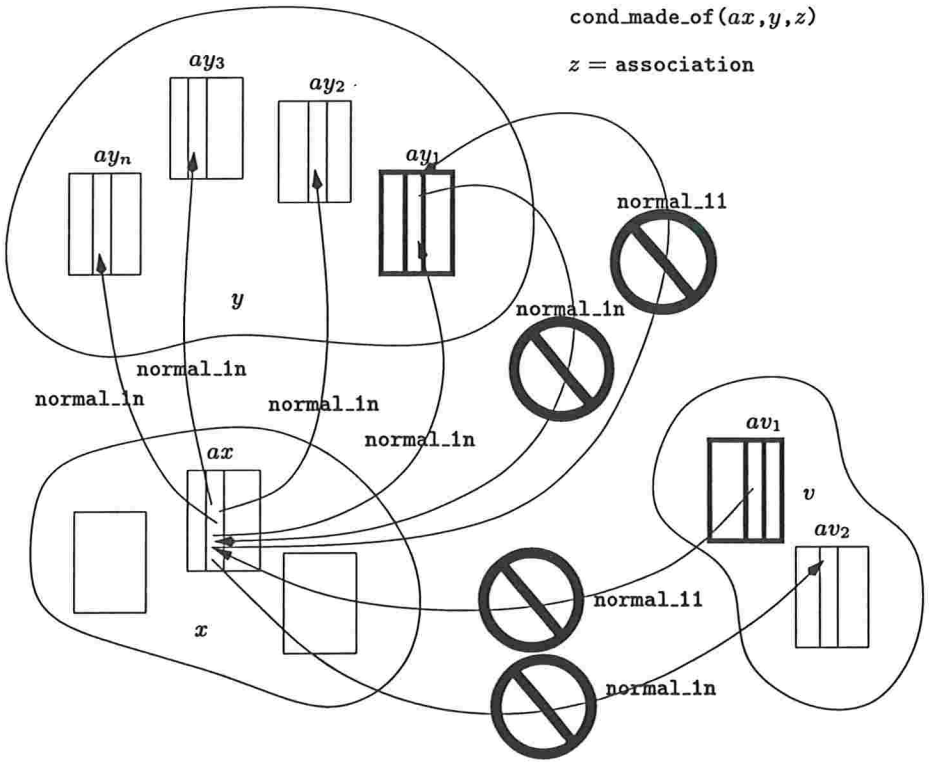


Figura 7.6: Exemplo do predicado cond_made_of(ax,y,z), quando z é igual a association.

```
% DFN16
all x y z (made_of(x,y) <-> (exists v z (made_of_ast(v,z,y) & is_a(x,v)))).
```

composition_mode

O termo x é um modo de composição, ou composition_mode(x), sse x assumir os valores association ou aggregation.

```
% DFN17
all x (composition_mode(x) <-> ((x=association) | (x=aggregation))).
```

7.3.8 Predicado *property*

Dizemos que x possui o valor z para a propriedade y, ou property(x,y,z), sse as condições abaixo forem satisfeitas:

1. x é um **objeto_específico**.
2. Uma das duas condições a seguir devem ser satisfeitas:

(a) y é uma **classe_específica** e z é um **objeto_específico** e existe um **objeto_genérico** c, uma **tupla** tx e um **atributo** ax, tal que x o_representa tx, e tx tem o valor c para o propriedade ax. Além disso, para todo atributo ay, tal que `ar_classe(ay,y)`, existe uma tupla ty, tal que z o_representa ty e ty tem o valor c para a propriedade ay.

(b) y é uma **atributo** e z é um **objeto_genérico** e existe uma **tupla** tx, tal que x o_representa tx, e tx tem o valor z para a propriedade y.

```
% DFN18
all a y b
(property(a,y,b) <->
        objeto_especifico(a) &
                (
                  (classe_especifica(y) & objeto_especifico(b) &
                    (exists c tx ax (objeto_generico(c) & tupla(tx) &
                    atributo(ax) & o_representa(a,tx) & propriedade(tx,ax,c) &
                        (all ay (atributo(ay) & ar_classe(ay,y) ->
                                (exists ty (tupla(ty) & o_representa(b,ty) &
                                propriedade(ty,ay,c)))
                        ))
                    ))
                  )


                |
                  (atributo(y) & objeto_generico(b) &
                    (exists tx (tupla(tx) & o_representa(a,tx) & propriedade(tx,y,b)
                    ))
                  )
                )

).
```

7.3.9 Predicados *component* e *indexical*

component

Dizemos que a é construído de b com **indexical** z, ou `component(a,z,b)` sse as condições abaixo forem satisfeitas:

1. a e b são **objetos_ específicos.**
2. Uma das duas condições a seguir devem ser satisfeitas:
   (a) z é um **objeto_específico** e z é igual a b. Além disso, existem x e y, **classes_específicas**, tal que a é uma instância de x, ou

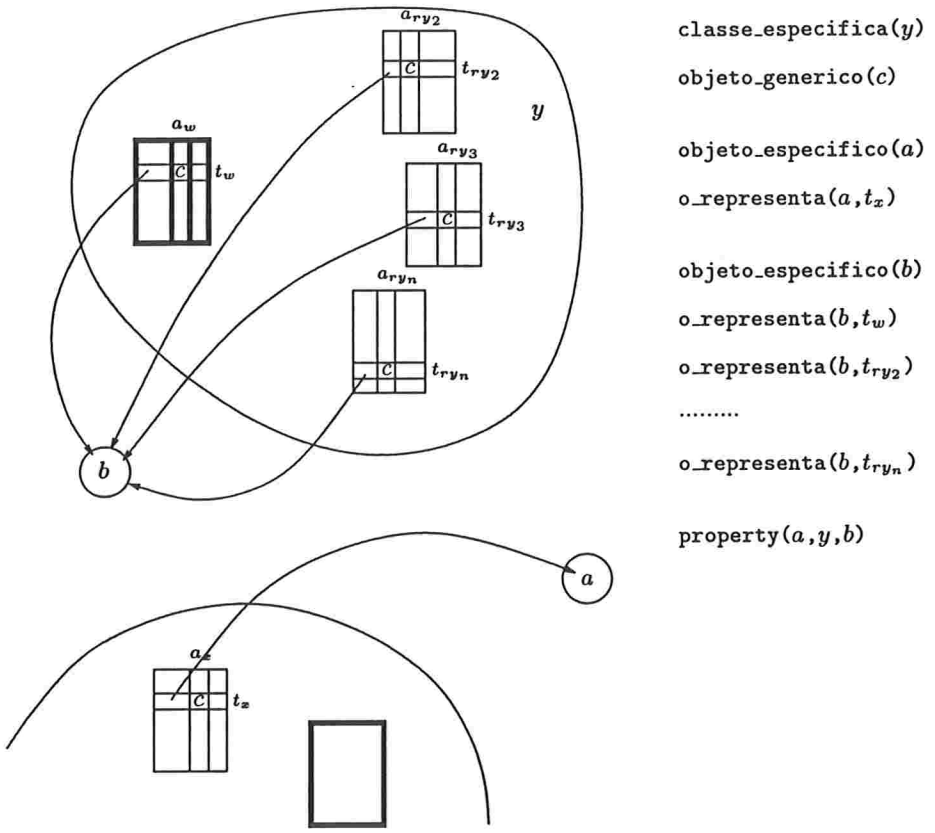


Figura 7.7: Exemplo do predicado property(a,y,b), quando y é uma **classe_específica**.

instance_of(a,x), e b é uma instância de y, ou instance_of(b,y) e x é composto de y por association.

(b) z é um **classe_específica** e existe x, também uma **classe_específica**, tal que a é uma instância de x, ou instance_of(a,x), e b é uma instância de z, ou instance_of(b,z) e x é composto de z por aggregation.

```
% DFN19
all a z b
(component(a,z,b) <->
        (objeto_especifico(a) & objeto_especifico(b) &
        objeto_especifico(z) &  (z=b) &
        (exists x y (classe_especifica(x) & classe_especifica(y) &
                instance_of(a,x) & instance_of(b,y) &
                made_of_ast(x,association,y))))
```

```
        |
        (objeto_especifico(a) & objeto_especifico(b) &
        classe_especifica(z) &
        (exists x (classe_especifica(x) &
                instance_of(a,x) & instance_of(b,z) &
                made_of_ast(x,aggregation,z))))
).
```

indexical

O termo x é um **indexical**, ou `indexical(x)`, sse x for uma **specific_class** (`specific_class(x)`) ou x for um **specific_object** (`specific_object(x)`).

```
% DFN20
all x (indexical(x) <-> (specific_class(x) | specific_object(x))).
```

7.4 Provas entre os modelos

No restante deste capítulo introduzimos as provas de $(REL_E \cup \Delta \vdash RR)$. Comentaremos inicialmente a sintaxe dessas provas e a seguir apresentaremos todas as provas com a semântica de algumas delas.

7.4.1 Sintaxe das provas

A apresentação das provas seguirá um padrão específico, como mostrado pelo exemplo abaixo.

```
% PROOF X04 (Axioma RR20)(DEP. PRP1, DFN7, PRP7)
all x y ((is_a(x,y) & is_a(y,x)) -> x=y).
```

onde:

- `all x y ((is_a(x,y) & is_a(y,x)) -> x=y).` é a sentença provada.
- `PROOF X04` é a identificação da sentença provada. Todas as provas realizadas possuem o prefixo `PROOF`.
- `(Axioma RR20)`. Quando esta identificação aparece, a sentença provada corresponde a um axioma do modelo RR.
- `(DEP. PRP1, DFN7, PRP7)` é a lista das sentenças que deduzem a sentença provada. Esta lista corresponde ao conjunto ϕ_i de sentenças descrito na *metodologia do escopo reduzido* na seção 5.2.2 (página 59). Estas sentenças podem ser axiomas de REL_E, definições de Δ, e provas ou subprovas já realizadas anteriormente.

Há uma outra espécie de prova, um pouco mais complexa, que é a `PROVA COM SUBPROVAS`, como descrito na *metodologia das subprovas* na seção 5.2.3 (página 59). Um exemplo dessa prova é dado a seguir.

```
% PROVA COM SUBPROVAS DO AXIOMA RR31

% PROOF X22.01 (DEP. DFN4, DFN5)
all x y (atri_raiz(x,y) -> ar_classe(x,y)).
% PROOF X22.02 (DEP. PRP8, PROOF X22.01, DFN15) (3.5 min)
all ax x y (atri_raiz(ax,x) & cond_made_of(ax,y,association)
              -> (-cond_made_of(ax,y,aggregation))).
% PROOF X22.03 (DEP. DFN15)
all ax y z (cond_made_of(ax,y,z)
              -> ((z=association)| (z=aggregation))).
% PROOF X22.04 (DEP. PROOF X22.03, PROOF X22.02)
all ax x y w z (atri_raiz(ax,x) & cond_made_of(ax,y,w)
              & cond_made_of(ax,y,z) -> (z=w)).
% PROOF X22 (Axioma RR31)(DEP. PROOF X11.05, PROOF X22.04, DFN14)
all x u v z (made_of_ast(x,u,z) & made_of_ast(x,v,z)
              -> (u=v)).
```

Aqui PROOF X22 (ou axioma RR31) é a prova que estamos perseguindo. Para isso, antes realizamos as subprovas PROOF X22.01 a PROOF X22.04. Notem que na subprova PROOF X22.02 informamos o tempo que o provador de teoremas levou para concluí-la, no caso 3.5 minutos. Este tempo de prova só é informado quando for superior a 1 minuto. É claro que este tempo depende de fatores tais como hardware e sistema operacional. Em nosso caso, o hardware utilizado foi um computador com procesador Intel PENTIUM 200 Mhz, memória RAM de 32 Mb e rodando o sistema operacional Red Hat LINUX.

Para completar a sintaxe das provas, em algumas delas consta o símbolo (S/I), que significa *sem igualdade.* Nessas provas os axiomas da igualdade foram desconsiderados, pois sabíamos de antemão que não seriam necessários para o processo de dedução. Assim utilizávamos um comando do OTTER, set(tptp_eq)., no início do arquivo de entrada, que fazia com que a máquina dedutiva não utilizasse esses axiomas, tornando o comportamento do predicado da igualdade igual ao de qualquer outro predicado, e como conseqüência trazendo uma redução dramática sobre o tempo de prova.

7.4.2 Apresentação das provas

Tentamos manter as provas na mesma ordem em que os axiomas de *RR* foram apresentados no capítulo anterior, sem muito sucesso. A apresentação desses axiomas está na verdade na ordem em que foram efetivamente provados, o que é mais fiel à realidade. Notem que muitos resultados obtidos em provas de determinados axiomas são usados em provas de axiomas posteriores. E que a prova de muitos axiomas foi dividida em *subprovas* para que a prova principal fosse concebida num tempo relativamente curto pelo provador de teoremas. Mesmo assim algumas dessas *subprovas* tomaram ainda um bom tempo, tal como a

prova PROOF X24.01, que levou cerca de 10 horas para encerrar.

Para a maioria das provas mais complexas introduzimos comentários que visam explicá-las de uma maneira informal e procuram mostrar apenas os aspectos mais importantes de suas semânticas, deixando os detalhes em segundo plano. Queremos ser, com isso, mais didáticos do que exatos. Esses comentários em geral estarão sempre dispostos logo após a prova comentada.

Resultados envolvendo relacionamentos *is_a*

```
% PROOF X01 (Axioma RR14)(DEP. DFN6a - DFN6f)
all x (class(x) <-> (generic_class(x) | specific_class(x))).
```

```
% PROOF X02 (Axioma RR18)(DEP. DFN7, DFN6a, DFN6b,
% DFN6c E PRP3) (S/I)
all x y ((is_a(x,y) & specific_class(y)) -> specific_class(x)).
```

```
% PROOF X03 (Axioma RR19)(DEP. DFN7, DFN6a, DFN6b,
% DFN6c E PRP3) (S/I)
all x y ((is_a(x,y) & generic_class(y)) -> generic_class(x)).
```

```
% PROOF X04 (Axioma RR20)(DEP. PRP1, DFN7, PRP7)
all x y ((is_a(x,y) & is_a(y,x)) -> x=y).
```

```
% PROOF X05 (Axioma RR22)(DEP. PRP1, DFN7, PRP7) (7.5 min)
all x y z (is_a(x,y) & is_a(y,z) -> is_a(x,z)).
```

Comentário 1 Ou x, y e z são todas classes genéricas ou são todas classes específicas, segundo a definição de is_a (DFN7). No primeiro caso, também da definição de is_a, $x = y$ e $y = z$. Portanto $x = z$ e x é subclasse de z. No segundo caso, todas as relações c_representadas por y são c_representadas por x, todas as relações c_representadas por z são c_representadas por y, e portanto todas as relações c_representadas por z são c_representadas por x, de onde se conclui que x é subclasse de z.

```
% PROOF X06 (Axioma RR21)(DEP. DFN7, DFN6d, DFN6e,
% + sentença válida (all x (x=x).) )
all x (class(x) -> is_a(x,x)).
```

```
% PROVA COM SUBPROVAS DO AXIOMA RR23

% PROOF X07.01 (DEP. PRP5, PRP1, DFN7)
all x y rx ry (classe_especifica(x) & classe_especifica(y) &
        is_a(x,y) & raiz(rx) & raiz(ry) & c_representa(x,rx) &
        c_representa(y,ry) -> (rx=ry)).

% PROOF X07.02 (DEP. PROOF X07.01, PRP5, DFN7, PRP1)
all x y z (is_a(x,y) & is_a(x,z) & (y!=z) ->
```

```
        (exists w (raiz(w) & c_representa(x,w) & c_representa(y,w)
        & c_representa(z,w)))).

% PROOF X07.03 (DEP. DFN7, PRP1, DFN3)
all x y w (c_representa(x,w) & rep_unic(y,w) -> is_a(x,y)).

% PROOF X07.04 (DEP. PROOF X07.03, PRP6 )
all x w (raiz(w) & c_representa(x,w) ->
        (exists z (rep_unic(z,w) & is_a(x,z)))).

% PROOF X07.05 (DEP. DFN3, PRP7) (1.5 min)
all x z w (rep_unic(x,w) & rep_unic(z,w) -> (x=z)).

% PROOF X07 (Axioma RR23)(DEP. PROOF X07.02, PROOF X07.04,
% PROOF X07.05)
all x y z ((is_a(x,y) & is_a(x,z) & ( - (y=z)))
                    -> ( exists w (is_a(y,w) & is_a(z,w)))).
```

Comentário 2 Ou x, y e z são todas classes genéricas ou são todas classes específicas, segundo a definição de `is_a` (DFN7). No primeiro caso, também da definição de `is_a`, y não pode ser diferente de z, e portanto a premissa da sentença é sempre falsa e a sentença é trivialmente verdadeira. No segundo caso, se x é subclasse de y, então, de `is_a` novamente, todas as relações c_representadas por y são c_representadas por x. Como toda raiz é uma relação e toda classe específica c_representa uma única raiz, então x e y c_representam uma mesma raiz r. Se x também é subclasse de z, com z diferente de y, então x, y e z c_representam a mesma raiz r. Como para toda raiz r existe uma classe específica w que c_representa unicamente r, então toda relação c_representada por w é também c_representada por x, y e z. Então da definição de `is_a`, x, y e z são todas subclasses de w.

```
% PROOF X08 (Axioma RR1)(DEP. DFN7, DFN6d, DFN6e)
all x y (is_a(x,y) -> class(x) & class(y)).
```

Resultados envolvendo relacionamentos *instance_of* e *of_type*

```
% PROOF X09 (Axioma RR25)(DEP. DFN6f, DFN9, REL12, REL3)
all x (generic_object(x) -> ( exists y (of_type(x,y)))).
```

```
% PROOF X10 (Axioma RR7)(DEP. DFN6bcf, DFN9, REL3)
all x y (of_type(x,y) -> generic_object(x) & generic_class(y)).
```

```
% PROVA COM SUBPROVAS DO AXIOMA RR26

% PROOF X11.01 (DEP. PRP3, DFN6C, DFN7, DFN4) (S/I)
all x y at ( is_a(x,y) & atri_raiz(at,x) ->
```

```
            ((classe_especifica(x) & classe_especifica(y) &
            (all v (c_representa(y,v) -> c_representa(x,v)))) )).

% PROOF X11.02 (DEP. PRP3, DFN6c, DFN7, DFN4) (S/I)
all x y at ( is_a(x,y) & atri_raiz(at,y) ->
            ((classe_especifica(x) & classe_especifica(y) &
            (all v (c_representa(y,v) -> c_representa(x,v)))) )).

% PROOF X11.03 (DEP. REL16, REL15)
all rx ry ax ay (tem_ast(rx,ax) & tem_ast(ry,ay) & (rx!=ry)
                                          -> (ax!=ay)).

% PROOF X11.04 (DEP. PROOF X11.01, PROOF X07.01, DFN4, PRP8,
% PROOF X11.03) (1 hs)
all x y at ( is_a(x,y) & atributo(at) & atri_raiz(at,x)
                                    -> atri_raiz(at,y)).

% PROOF X11.05 (DEP. PRP5, PRP9, DFN2, DFN4, DFN5, REL16)
% (16 min)
all x y ax ay
(classe_especifica(x) & atri_raiz(ax,x) & atri_raiz(ay,x)
                                        -> (ax=ay)).

% PROOF X11.06 (DEP. DFN4, PROOF X11.04, PROOF X11.05)
all x y ax ay
(is_a(x,y) & atri_raiz(ax,x) & atri_raiz(ay,y) -> (ax=ay)).

% PROOF X11.07 (DEP. PRP8, PROOF X11.02, PROOF X11.06)
all x y at ( is_a(x,y) & atri_raiz(at,y) -> atri_raiz(at,x)).

% PROOF X11.08 (DEP. PRP3, DFN6c, DFN7, PROOF X11.07) (S/I)
all x y ( is_a(x,y) & classe_especifica(x) ->
        classe_especifica(y) &
        (all ry (c_representa(y,ry) -> c_representa(x,ry))) &
        (all ay (atri_raiz(ay,y) -> atri_raiz(ay,x)))
        ).

% PROOF X11.09 (DEP. DFN5, PROOF X11.08) (S/I)
all x y ( is_a(x,y) & classe_especifica(x) -> (( classe_especifica(y)
        & (all at (atributo(at) & ar_classe(at,y) -> ar_classe(at,x))))))).


% PROOF X11 (Axioma RR26)(DEP. PROOF X11.09, PRP1, PRP2, DFN8)
all x y z (instance_of(x,y) & is_a(y,z) -> instance_of(x,z)).
```

Comentário 3 Da definição de `instance_of` (DFN8), basicamente `instance_of(x,y)` é válido sse x for um objeto específico, y for uma classe específica e x o_representar uma tupla em cada relação c_representada por y. Como y é uma subclasse de z, da definição de `is_a` (DFN7), z é também classe específica e toda

relação c_representada por z é c_representada por y. Portanto x o_representa uma tupla em toda relação c_representada por z. Assim, também é verdade que instance_of(x,z).

```
% PROOF X12 (Axioma RR15)(DEP. DFN6a-DFN6i)
all x (object(x) <-> (generic_object(x) | specific_object(x))).
```

```
% PROOF X13 (Axioma RR6)(DEP. DFN6a-DFN6i, DFN8)
all x y (instance_of(x,y) -> specific_object(x) & specific_class(y)).
```

```
% PROVA COM SUBPROVAS DO AXIOMA RR24

% PROOF X14.01 (DEP. PRP6, PRP11, DFN6g)
all x (specific_object(x) -> (exists y ry ty (
        classe_especifica(y) & raiz(ry) & tupla(ty) &
        rep_unic(y,ry) & instancia_de(ty,ry) & o_representa(x,ty)))).

% PROOF X14.02 (DEP. DFN4, DFN5)
all x y (atri_raiz(x,y) -> ar_classe(x,y)).

% PROOF X14.03 (DEP. PRP8, PROOF X14.02, REL21, REL3, DFN3,
% PROOF X14.01)
all x (specific_object(x) -> (exists y ry ty ay c (
        classe_especifica(y) & raiz(ry) & tupla(ty) &
        atributo(ay) & objeto_generico(c) &
        ar_classe(ay,y) & rep_unic(y,ry) & instancia_de(ty,ry) &
        propriedade(ty,ay,c) & o_representa(x,ty)))).

% PROOF X14.04 (DEP. REL1, REL15, DFN2, DFN4, DFN5) (23 min)
all ay y
(ar_classe(ay,y) -> (exists ry (relacao(ry) & c_representa(y,ry)
                & tem_ast(ry,ay)))).

% PROOF X14.05 (DEP. REL1, DFN2, DFN3, PROOF X14.04)
all y ry ay (rep_unic(y,ry) & ar_classe(ay,y) -> tem_ast(ry,ay)).

% PROOF X14.06 (DEP. DFN2, DFN5, DFN3, PROOF X14.05,
% PRP9)   (5 min)
all y ry ax ay
(rep_unic(y,ry) & ar_classe(ax,y) & ar_classe(ay,y) -> (ax=ay)).

% PROOF X14 (Axioma RR24)(DEP. DFN6g, PROOF X14.03, DFN8, PROOF X14.06)
all x (specific_object(x) -> ( exists y (instance_of(x,y)))).
```

Comentário 4 De PRP11, se x é um objeto específico, x o_representa ao menos uma tupla em alguma raiz w. De PRP6, existe uma classe específica y que c_representa unicamente w. Portanto x o_representa uma tupla em toda relação c_representada por y, uma vez que a única relação representada por y é a raiz w. Isso, da definição DFN8, é basicamente suficiente para garantir que

```
instance_of(x,y).
```

Resultados envolvendo relacionamentos *has_ast* e *has*

```
% PROVA COM SUBPROVAS DO AXIOMA RR28

% PROOF X15.01 (DEP. DFN7, PRP3, DFN6a - DFN6c)
% (S/I)
all x y ( is_a(x,y) & classe_especifica(y) -> ((classe_especifica(x) &
                (all v (c_representa(y,v) -> c_representa(x,v)))) )).

% PROOF X15.02 (DEP. DFN11, PRP3, DFN10)
all x y
(has(x,y) -> (classe_especifica(y) & (-atributo(y))) |
                ((-classe_especifica(y)) & atributo(y))
).

% PROOF X15.03 (DEP. PROOF X15.01, PROOF X15.02, DFN10, DFN11)
all x y z (has(x,y) & classe_especifica(y) & is_a(z,x) -> has(z,y)).

% PROOF X15.04 (DEP. PROOF X15.01, PROOF X15.02, DFN10, DFN11)
all x y z (has(x,y) & atributo(y) & is_a(z,x) -> has(z,y)).

% PROOF X15 (Axioma RR28)(DEP. PROOF X15.02, PROOF X15.03, PROOF X15.04)
all x y z (has(x,y) & is_a(z,x) -> has(z,y)).
```

Comentário 5 Das definições de has_ast e has (DFN10 e DFN11), has(x,y) é verdadeiro sse x for uma classe específica e possuir uma superclasse v, tal que has_ast(v,y). Portanto toda subclasse z de x, da definição de is_a (DFN7), é também classe específica e tem v como superclasse e conseqüentemente has(z,y) é válido.

```
% PROOF X16 (Axioma RR27)(DEP. PROOF X06, DFN10, DFN6d, DFN6e, DFN11)
all x y (has_ast(x,y) -> has(x,y)).
```

Comentário 6 De PROOF X06, toda classe é subclasse dela mesma. Assim, caso has_ast(x,y), então x é classe, e x é subclasse de x. Da definição de has, temos então que has(x,y) é válido.

```
% PROOF X17 (Axioma RR2)(DEP. DFN10, DFN6a-DFN6i)
all x y (has_ast(x,y) -> specific_class(x) & class(y)).
```

```
% PROOF X18 (Axioma RR3)(DEP. DFN10, DFN6a-DFN6i, PROOF X02, DFN11)
all x y (has(x,y) -> specific_class(x) & class(y)).
```

Resultados envolvendo *made_of_ast* e *made_of*

```
% PROOF X19 (Axioma RR29)(DEP. PROOF X06, DFN14, DFN6d, DFN6e, DFN16)
all x y z (made_of_ast(x,z,y) -> made_of(x,y)).
```

Comentário 7 O comentário desta prova é análogo ao comentário da prova PROOF X16.

```
% PROOF X20 (Axioma RR30)(PROOF X02, DFN6a, PROOF X11.07, DFN14, DFN16)
all x y z (made_of(x,y) & is_a(z,x) -> made_of(z,y)).
```

Comentário 8 O comentário desta prova é análogo ao comentário da prova PROOF X15.

```
% PROVA COM SUBPROVAS DO AXIOMA RR34

% PROOF X21.01 (DEP. DFN7)
all x y z
((y!=z) & is_a(x,y) & is_a(x,z) ->
        classe_especifica(x) & classe_especifica(y) & classe_especifica(z)).

% PROOF X21.02 (DEP. PRP8, DFN6a, PROOF X11.04, PROOF X11.07, PROOF X21.01)
all x y z w v
((y!=z) & is_a(x,y) & is_a(x,z) ->
        (exists ax (atri_raiz(ax,x) & atri_raiz(ax,y) & atri_raiz(ax,z)))).

% PROOF X21 (Axioma RR34)(DEP. PRP8, PROOF X11.05, PROOF X21.01,
% PROOF X21.02, DFN14) (11 min)
all x y z w v
((y!=z) & is_a(x,y) & is_a(x,z) ->
        (made_of_ast(y,w,v) <-> made_of_ast(z,w,v))).
```

Comentário 9 A maior parte das próximas provas desta seção depende principalmente da definição do predicado `cond_made_of` (DFN15). Da definição DFN14, temos que `made_of_ast(x,z,y)` é válido sse x e y forem classes específicas e `cond_made_of(ax,y,z)` for válido, onde ax é o atributo raiz de x. Da definição DFN15,

1. z pode assumir dois valores, a saber, `association` ou `aggregation`.
2. no caso de z ser igual a `association`, as seguintes condições devem ser satisfeitas:
   (a) Para todo atributo ay, tal que `ar_classe(ay,y)`, existe uma `ri` com modo `normal_1n` de ax para ay.
   (b) Para todo atributo ay, em que não seja verdade que `ar_classe(ay,y)`, não pode existir uma `ri` com modo `normal_1n` de ax para ay.

(c) Para todo atributo ay, tal que `atri_raiz(ay,y)`, não pode existir uma `ri` com modo `normal_1n` de ay para ax, nem existir uma `ri` com modo `normal_11` de ax para ay.

(d) Para todo atributo ay e toda **classe_específica** yl, tal que `atri_raiz(ay,yl)`, não pode existir uma `ri` com modo `normal_11` de ay para ax.

3. no caso de z ser igual a `aggregation`, as seguintes condições devem ser satisfeitas:

(a) Para todo atributo ay, tal que `ar_classe(ay,y)`, existe uma `ri` com modo `normal_11` de ay para ax.

(b) Para todo atributo ay, tal que `atri_raiz(ay,y)`, não pode existir uma `ri` com modo `normal_11` de ax para ay, nem existir uma `ri` com modo `normal_1n` de ay para ax.

(c) Para todo atributo ay e toda **classe_específica** yl, tal que `atri_raiz(ay,yl)`, não pode existir uma `ri` com modo `normal_1n` de ax para ay.

- Os itens 2a e 2b juntos formam a base para provar que, através de associação, uma classe específica só pode ser composta por uma única classe especifica. Ou seja, formam a base para provar `PROOF X24`.
- Os itens 1, 2a, 2c, 3a e 3b juntos formam a base para provar que, se x é composta por y, então y não pode ser composta por x. Ou seja, formam a base para provar `PROOF X23`.
- Os itens 1, 2a, 2d, 3a e 3c juntos formam a base para provar que x só pode ser composta por y por apenas um modo de composição. Ou seja, formam a base para provar `PROOF X22`.

```
% PROVA COM SUBPROVAS DO AXIOMA RR31

% PROOF X22.01 (DEP. DFN4, DFN5)
all x y (atri_raiz(x,y) -> ar_classe(x,y)).

% PROOF X22.02 (DEP. PRP8, PROOF X22.01, DFN15) (3.5 min)
all ax x y
(atri_raiz(ax,x) & cond_made_of(ax,y,association) ->
                        (-cond_made_of(ax,y,aggregation))).

% PROOF X22.03 (DEP. DFN15)
all ax y z
(cond_made_of(ax,y,z) -> ((z=association)| (z=aggregation))).

% PROOF X22.04 (DEP. PROOF X22.03, PROOF X22.02)
all ax x y w z
(atri_raiz(ax,x) & cond_made_of(ax,y,w) & cond_made_of(ax,y,z) -> (z=w)).
```

```
% PROOF X22 (Axioma RR31)(DEP. PROOF X11.05, PROOF X22.04, DFN14)
all x u v z (made_of_ast(x,u,z) & made_of_ast(x,v,z) -> (u=v)).
```

```
% PROVA COM SUBPROVAS DO AXIOMA RR33

% PROOF X23.01 (4 PROVAS) (DEP. PROOF X22.01, DFN15) (MÁXIMO DE 5.5 min)
all ax x ay y
(atri_raiz(ax,x) & atri_raiz(ay,y) & cond_made_of(ax,y,association) ->
                                    (-cond_made_of(ay,x,association))).
all ax x ay y
(atri_raiz(ax,x) & atri_raiz(ay,y) & cond_made_of(ax,y,association) ->
                                    (-cond_made_of(ay,x,aggregation))).
all ax x ay y
(atri_raiz(ax,x) & atri_raiz(ay,y) & cond_made_of(ax,y,aggregation) ->
                                    (-cond_made_of(ay,x,association))).
all ax x ay y
(atri_raiz(ax,x) & atri_raiz(ay,y) & cond_made_of(ax,y,aggregation) ->
                                    (-cond_made_of(ay,x,aggregation))).


% PROOF X23.02 (2 PROVAS) (DEP. PROOF X23.01, PROOF X22.03)
% (MÁXIMO DE 3 min)
all ax x ay y z
(atri_raiz(ax,x) & atri_raiz(ay,y) & cond_made_of(ax,y,z) ->
                                     (-cond_made_of(ay,x,association))).
all ax x ay y z
(atri_raiz(ax,x) & atri_raiz(ay,y) & cond_made_of(ax,y,aggregation) ->
                                     (-cond_made_of(ay,x,z))).

% PROOF X23.03 (DEP. PROOF X22.03, PROOF X23.02)
all ax x ay y z w
(atri_raiz(ax,x) & atri_raiz(ay,y) & cond_made_of(ax,y,w) ->
                                          (-cond_made_of(ay,x,z))).

% PROOF X23.04 (DEP. PROOF X23.03, DFN14)
all x y z w (made_of_ast(x,z,y) -> (-made_of_ast(y,w,x))).

% PROOF X23 (Axioma RR33)(DEP. PROOF X23.04, DFN16, PROOF X02,
% DFN6a, PROOF X11.07, DFN14)
all x y (made_of(x,y) -> (- made_of(y,x))).
```

```
% PROVA COM SUBPROVAS DO AXIOMA RR32

% PROOF X24.01 (DEP. DFN2, DFN4, DFN5, PRP8, REL1, REL14, REL16, REL15)
% (10 HS !)
all x rx (
classe_especifica(x) & relacao(rx) & c_representa(x,rx) ->
        (exists ax (tem_ast(rx,ax) & ar_classe(ax,x))) ).
```

```
% PROOF X24.02 (DEP. PRP1, PROOF X14.04, PROOF X24.01, REL16)
all x y (classe_especifica(x) & classe_especifica(y) &
        (all at (ar_classe(at,y) -> ar_classe(at,x))) ->
                (all r (c_representa(y,r) -> c_representa(x,r))) ).

% PROOF X24.03 (DEP. PROOF X24.02, PRP7)
all x y (classe_especifica(x) & classe_especifica(y) &
        (all at (ar_classe(at,x) <-> ar_classe(at,y))) -> (x=y) ).


% PROOF X24.04 (DEP. DFN15) (S/I)
all ax y z (cond_made_of(ax,y,z) & (z=association) ->
                (atributo(ax) & classe_especifica(y) &
                        (all ay (ar_classe(ay,y) -> ri(ax,ay,normal_1n))) &
                        (all ay ((-ar_classe(ay,y)) -> (-ri(ax,ay,normal_1n))))
                )
        ).

% PROOF X24.05 (DEP. PROOF X24.04, PROOF X24.03)
all at x y z
(cond_made_of(at,x,z) & cond_made_of(at,y,z) & (z=association) -> (x=y))

% PROOF X24 (Axioma RR32)(DEP. PROOF X24.05, PROOF X11.05, DFN14)
all x y u v
(made_of_ast(x,y,u) & (y=association) & made_of_ast(x,y,v) -> (u=v)).
```

```
% PROOF X25 (Axioma RR5)(DEP. PROOF XO2, DFN6a, DFN14, DFN16)
all x y (made_of(x,y) -> specific_class(x) & specific_class(y)).
```

```
% PROOF X26 (Axioma RR4)(DEP. PROOF X22.03, DFN14, DFN17, DFN6a)
all x y u (made_of_ast(x,y,u) ->
        specific_class(x) & composition_mode(y) & specific_class(u) ).
```

```
% PROOF X27 (Axioma RR17) (DEP. DFN17)
all x (composition_mode(x) -> (x=association | x=aggregation)).
```

Resultados envolvendo relacionamentos *property*

```
% PROVA COM SUBPROVAS DO AXIOMA RR35

% PROOF X28.01 (DEP. REL1, REL4, REL5, REL15, REL20 , REL19, REL23,
% DFN4, DFN5) (2 HS)
all y ay ty c (classe_especifica(y) & atributo(ay) &  ar_classe(ay,y)
        & tupla(ty) & propriedade(ty,ay,c) -> (exists ry (relacao(ry)
        & c_representa(y,ry) &  tem_ast(ry,ay) &
        instancia_de(ty,ry)))).
```

```
% PROOF X28.02 (DEP. REL16, PROOF X28.01, DFN8, PROOF X24.01)
all x y
(instance_of(x,y) -> (
        objeto_especifico(x) & classe_especifica(y) &
        (all ry (relacao(ry) & c_representa(y,ry) ->
                (exists ty (tupla(ty) & instancia_de(ty,ry) & o_representa(x,ty)))
        ))
        )).

% PROOF X28.03 (DEP. DFN10, PROOF X15.01, PROOF X15.02, DFN11, DFN6a)
all x y
(has(x,y) & specific_class(y) -> (classe_especifica(x) & classe_especifica(y) &
                (exists rx ax (relacao(rx) & atributo(ax) &
                c_representa(x,rx) & tem_ast(rx,ax) &
                        (all ay (ar_classe(ay,y) -> ri(ay,ax,has_11)))
                )))

).

% PROOF X28.04 (DEP. PROOF X28.03, PROOF X28.02, REL21, REL27, REL5, PRP10)
all x y a
(has(x,y) & specific_class(y) & instance_of(a,x) ->
        classe_especifica(x) & classe_especifica(y) & objeto_especifico(a) &
        (exists b c tx ax (objeto_especifico(b) & objeto_generico(c) &
        tupla(tx) & atributo(ax) & o_representa(a,tx) & propriedade(tx,ax,c) &
                (all ay ((atributo(ay) & ar_classe(ay,y)) ->
                        (exists ty (tupla(ty) & propriedade(ty,ay,c) &
                        o_representa(b,ty)))
                ))
                )
        )
).

% PROOF X28.05 (DEP. DFN8)
all x y
((
        objeto_especifico(x) & classe_especifica(y) &
        (exists c (objeto_generico(c) &
                (all ay (atributo(ay) & ar_classe(ay,y) ->
                        (exists ty (tupla(ty) &
                        propriedade(ty,ay,c) & o_representa(x,ty)))
                ))
        ))
) -> instance_of(x,y)).


% PROOF X28.06 (DEP. DFN18)
all x y z
( (objeto_especifico(x) & classe_especifica(y) & objeto_especifico(z) &
                    (exists c tx ax (objeto_generico(c) & tupla(tx) &
```

```
                    atributo(ax) & o_representa(x,tx) & propriedade(tx,ax,c) &
                        (all ay (atributo(ay) & ar_classe(ay,y) ->
                            (exists ty (tupla(ty) & o_representa(z,ty) &
                            propriedade(ty,ay,c)))
                        ))
                    ))
  )
 -> property(x,y,z)).

% PROOF X28 (Axioma RR35)(DEP. PROOF X28.05, PROOF X28.06,
% PROOF X28.04) (3 min)
all x y a (has(x,y) & specific_class(y) & instance_of(a,x)  -> (exists
b (instance_of(b,y) & property(a,y,b) ))).
```

Comentário 10 Se `has(x,y)` e y é classe específica, das definições DFN10 e DFN11, existe uma `ri` com modo `has_11` de cada atributo ay de y, tal que `ar_classe(ay,y)`, para um único atributo ax de x. Além disso, se `instance_of(a,x)`, de DFN8, o objeto específico a o_representa uma tupla em cada relação c_representada por x. Em particular, a o_representa uma tupla tx na relação que possui o atributo ax. Assim, de REL21, tx tem um valor c no atributo ax, ou `propriedade(tx,ax,c)`. Do axioma REL27 e do apresentado no início deste comentário, deve existir uma tupla ty em cada relação c_representada por y, tal que ty tem o valor c para cada atributo ay ar_classe de y, ou `propriedade(ty,ay,c)`. De PRP10, existe então um objeto específico b que o_representa essas tuplas ty de y. Isso, junto a DFN8, garantem que `instance_of(b,y)`. E da definição DFN18, também é verdade que `property(a,y,b)`.

```
% PROVA COM SUBPROVAS DO AXIOMA RR36

% PROOF X29.01 (DEP. DFN10, PROOF X15.01, PROOF X15.02, DFN11, DFN6a)
all x y  (has(x,y) & generic_class(y) -> (classe_especifica(x) &
atributo(y) & (exists rx (relacao(rx) & c_representa(x,rx) &
tem_ast(rx,y)))) ).

% PROOF X29.02 (DEP. PROOF X29.01, PROOF X28.02, REL21, REL5)
all x y a
(has(x,y) & generic_class(y) & instance_of(a,x) ->
        classe_especifica(x) & atributo(y) & objeto_especifico(a) &
        (exists b tx z (objeto_generico(b) & tupla(tx) & o_representa(a,tx) &
        propriedade(tx,y,b) & tipo(z) & tem_tipo(y,z) & de_tipo(b,z)
        ))
).

% PROOF X29 (Axioma RR36)(DEP. DFN6f, PRP12, PRP13,
% DFN18, DFN9, PROOF X29.02)
all x y a
(has(x,y) & generic_class(y) & instance_of(a,x)  ->
        (exists b (of_type(b,y) & property(a,y,b) ))).
```

Comentário 11 Se has(x,y) e y é classe genérica, das definições DFN10 e DFN11, y é um atributo de alguma relação c_representada por x. Da mesma forma que no comentário 10, o objeto específico a o_representa uma tupla tx na relação c_representada por x que possui o atributo y. Assim, de REL21, tx tem um valor b em y, ou propriedade(tx,y,b), e existe um tipo t tal que de_tipo(b,t) e tem_tipo(y,t). De DFN9 é verdade que of_type(b,y) e de DFN18 é verdade que property(a,y,b).

```
% PROOF X30 (Axioma RR8)(DEP. DFN18, DFN6a-DFN6i, PRP12, PRP13)
all x y u (property(x,y,u) -> specific_object(x) & class(y) & object(u)).
```

Resultados envolvendo relacionamentos *component*

```
% PROVA COM SUBPROVAS DO AXIOMA RR38

% PROOF X31.01 (DEP. DFN4)
all ax x ( atri_raiz(ax,x) -> atributo(ax) & classe_especifica(x) &
                (exists rx (raiz(rx) & tem_ast(rx,ax) & c_representa(x,rx) ))
        ).

% PROOF X31.02 (DEP. DFN15) (S/I)
all ax y z (cond_made_of(ax,y,z) & (z=aggregation) ->
                atributo(ax) & classe_especifica(y) &
                        (all ay (ar_classe(ay,y) -> ri(ay,ax,normal_11)))
                ).

% PROOF X31.03 (DEP. PROOF X31.01, DFN2, DFN14, PROOF X31.02)
all x z y
(made_of_ast(x,z,y) & (z=aggregation) -> (classe_especifica(x) & classe_especifica(y) &
                (exists rx ax (relacao(rx) & atributo(ax) &
                c_representa(x,rx) & tem_ast(rx,ax) &
                        (all ay (ar_classe(ay,y) -> ri(ay,ax,normal_11)))
                )))

).

% PROOF X31.04 (DEP. PROOF X31.03, PROOF X28.02, REL21, REL27, REL5, PRP10)
all x z y a
(made_of_ast(x,z,y) & (z=aggregation) & instance_of(a,x) ->
        classe_especifica(x) & classe_especifica(y) & objeto_especifico(a) &
        (exists b c tx ax (objeto_especifico(b) & objeto_generico(c) &
        tupla(tx) & atributo(ax) & o_representa(a,tx) & propriedade(tx,ax,c) &
                (all ay ((atributo(ay) & ar_classe(ay,y)) ->
                        (exists ty (tupla(ty) & propriedade(ty,ay,c) &
                        o_representa(b,ty)))
                ))
```

```
                )
        )
).

% PROOF X31.05 (DEP. DFN19)
all x y z w
        (objeto_especifico(x) & classe_especifica(y) & objeto_especifico(z) &
        instance_of(x,w) & made_of_ast(w,aggregation,y) &
                    (exists c tx ax (objeto_generico(c) & tupla(tx) &
                    atributo(ax) & o_representa(x,tx) & propriedade(tx,ax,c) &
                        (all ay (atributo(ay) & ar_classe(ay,y) ->
                                (exists ty (tupla(ty) & o_representa(z,ty) &
                                propriedade(ty,ay,c)))
                        ))
                    ))
 -> component(x,y,z)).

% PROOF X31 (Axioma RR38)(DEP. PROOF X28.05, PROOF X31.05, PROOF X31.04)
% (3 min)
all a x y z ((instance_of(a,x) & made_of_ast(x,y,z) & (y=aggregation)) ->
                (exists b (component(a,z,b) & instance_of(b,z)))).
```

Comentário 12 Esta prova é muito semelhante à prova PROOF X28. Se x é composta por z através de agregação, ou `made_of_ast(x,aggregation,z)`, das definições DFN14 e DFN15, x e z são classes específicas e existe uma `ri` com modo `normal_11` de cada atributo az de z, tal que `ar_classe(az,z)`, para o único atributo raiz ax de x. Além disso, se `instance_of(a,x)`, de DFN8, o objeto específico a o_representa uma tupla em cada relação c_representada por x. Em particular, a o_representa uma tupla tx na relação que possui o atributo ax. Assim, de REL21, tx tem um valor c no atributo ax, ou `propriedade(tx,ax,c)`. Do axioma REL27 e do apresentado no início deste comentário, deve existir uma tupla tz em cada relação c_representada por z, tal que tz tem o valor c para cada atributo az ar_classe de z, ou `propriedade(tz,az,c)`. De PRP10, existe então um objeto específico b que o_representa essas tuplas tz de z. Isso, junto a DFN8, garantem que `instance_of(b,y)`. E da definição DFN19, também é verdade que `component(a,z,b)`.

```
% PROVA COM SUBPROVAS DO AXIOMA RR37

% PROOF X32.01 (DEP. DFN7, DFN6a, DFN6b, DFN6c E PRP3) (S/I)
all x y ((is_a(x,y) & specific_class(x)) -> specific_class(y)).

% PROOF X32.02  (DEP. DFN4, PROOF X32.01, DFN6a, PROOF X11.04, DFN14)
all x y z w (made_of_ast(x,z,y) & is_a(x,w) -> made_of_ast(w,z,y)).

% PROOF X32.03 (DEP. DFN2, PROOF X28.02, PRP5)
all x y
(instance_of(x,y) -> (
        objeto_especifico(x) & classe_especifica(y) &
```

```
                (exists ry ty (raiz(ry) & c_representa(y,ry) & tupla(ty) &
                instancia_de(ty,ry) & o_representa(x,ty)))
        )).

% PROOF X32.04 (DEP. PROOF X32.03, PRP11)
all x y z (instance_of(x,y) & instance_of(x,z) ->
        (exists r (raiz(r) & c_representa(y,r) & c_representa(z,r)))).

% PROOF X32.05 (DEP. PROOF X32.04, PROOF X07.04, PROOF X07.05, DFN3)
all x y z (instance_of(x,y) & instance_of(x,z) ->
        (exists w (classe_especifica(w) & is_a(y,w) & is_a(z,w))))

% PROOF X32.06 (DEP. PRP8, DFN15, PROOF X22.01) (3.5 min)
all ax x y w
(atri_raiz(ax,x) & cond_made_of(ax,y,association) ->
                        (-cond_made_of(ax,w,aggregation))).

% PROOF X32.07 (DEP. PROOF X11.05, PROOF X32.06, DFN14)
all x y w
(made_of_ast(x,association,w) -> (- made_of_ast(x,aggregation,y))).

% PROOF X32.08 (DEP. PROOF X32.05, PROOF X32.02, DFN19, PROOF X32.07)
all a b bl x y z ((instance_of(a,x) & made_of_ast(x,y,z) & (y=association) &
                  component(a,bl,b)) -> objeto_especifico(bl) ).

% PROOF X32.09 (DEP. DFN19, PRP3, PROOF X32.08) (2.5 min)
all a b bl x y z ((instance_of(a,x) & made_of_ast(x,y,z) & (y=association) &
                  component(a,bl,b)) ->
                        (objeto_especifico(a) & objeto_especifico(b) &
                        objeto_especifico(bl) & (bl=b) &
                        (exists xl yl (classe_especifica(xl) & classe_especifica(yl) &
                                instance_of(a,xl) & instance_of(b,yl) &
                                made_of_ast(xl,association,yl))))
                ).

% PROOF X32 (Axioma RR37)(DEP. PROOF X32.05, PROOF X32.02, PROOF X24,
% PROOF X32.09)
all a b bl x y z ((instance_of(a,x) & made_of_ast(x,y,z) & (y=association) &
                  component(a,bl,b)) -> (instance_of(b,z) & (bl=b)))
```

Comentário 13 Voltando ao comentário 10, os itens 1, 2a, 2d, 3a e 3c provam um resultado mais forte que PROOF X22. Juntos formam a base para provar que se uma classe específica x é composta através de associação de alguma classe específica, então x não pode ser composta por agregação de qualquer classe específica, e vice-versa. Isso é provado em PROOF X32.07. Deste resultado, e sendo válidos `made_of_ast(x,association,z)`, `component(a,bl,b)` e `instance_of(a,x)`, é também válida a primeira parte da disjunção da definição DFN19, ou seja, é verdade que b e bl são objetos específicos e $b = bl$, e também que existe uma classe específica y, tal que `instance_of(b,y)` e

made_of_ast(x,association,y). Como made_of_ast(x,association,z) e de PROOF X24, deduzimos que $y = z$, e conseqüentemente que instance_of(b,z).

```
% PROOF X33 (Axioma RR16)(DEP. DFN20)
all x (indexical(x) <-> (specific_class(x) | specific_object(x))).
```

```
% PROOF X34 (Axioma RR9)(DEP. DFN19, DFN20, DFN6a, DFN6g)
all x y u (component(x,y,u) -> specific_object(x) & indexical(y) &
                               specific_object(u)).
```

Resultados envolvendo a disjunção dos conjuntos base

Comentário 14 As provas dos próximos axiomas são um tanto quanto diretas. Em geral cada conjunto em RR é definido como sendo um conjunto em REL_E, cujos conjuntos, por sua vez, são também disjuntos entre si. Duas exceções merecem atenção. A primeira é que o conjunto generic_class em RR é definido como sendo a união dos conjuntos atributo e tipo de REL_E, disjuntos entre si e relativamente ao outros conjuntos de REL_E. Portanto, a união dos mesmos será também disjunta. A outra exceção é que generic_object e composition_mode são ambos definidos como sendo cada um uma partição do conjunto objeto_generico de REL_E, e são, portanto, também disjuntos entre si.

```
% PROOF X35 (Axioma RR10)(DEP. DFN6a-DFN6i,
% REL7-REL11, PRP3, PRP4, FAT1, FAT2,
% DFN12, DFN13, DFN17). (8 min)
all x ( specific_class(x) -> (- generic_class(x)) & (- specific_object(x)) &
                (- generic_object(x)) & (- composition_mode(x))
).
```

```
% PROVA COM SUBPROVAS DO AXIOMA RR11

% PROOF X36.01 (DEP. DFN6bcfg, REL8, REL9, PRP4, FAT1, FAT2, DFN12,
% DFN13, DFN17)
all x (
tipo(x)  -> (- specific_object(x)) & (- generic_object(x)) &
                (- composition_mode(x))).

% PROOF X36.02 (DEP. DFN6bcfg, REL8, REL9, PRP4, FAT1, FAT2, DFN12,
% DFN13, DFN17)
all x (
atributo(x)  -> (- specific_object(x)) & (- generic_object(x)) &
                (- composition_mode(x))).

% PROOF X36 (Axioma RR11)(DEP. DFN6bc, PROOF X36.01, PROOF X36.02)
```

```
all x (
generic_class(x)  -> (- specific_object(x)) & (- generic_object(x)) &
                (- composition_mode(x)))
```

```
% PROOF X37 (Axioma RR12)(DEP. DFN6fg, REL9, PRP4, FAT1, FAT2, DFN12,
% DFN13, DFN17)
all x (
specific_object(x) -> (- generic_object(x)) & (- composition_mode(x))
).
```

```
% PROOF X38 (Axioma RR13)(DEP. DFN6f, FAT1, FAT2, DFN12, DFN13, DFN17)
all x (
generic_object(x) -> (- composition_mode(x))
).
```

Capítulo 8

Conclusões e outros usos

A análise comparativa formal entre modelos de bancos de dados demonstrou, com este trabalho, ser uma tarefa um tanto difícil de realizar. Esta dificuldade vai desde a geração de axiomatizações corretas dos modelos até os problemas de complexidade e modularidade dos processos de prova entre os axiomas gerados. Mas demonstrou também que, devido à dificuldade de comparar formalmente dois modelos de dados, torna-se duvidoso crer na exatidão de várias comparações informais existentes na literatura.

Ao longo deste trabalho deixamos também algumas atividades como pontos de partida para futuras pesquisas. Dentre elas podemos destacar:

- tentar encontrar interpretações fiéis entre duas teorias de modelos de bancos de dados;
- comparar não apenas aspectos estruturais dos modelos, mas também aspectos comportamentais;
- evoluir o estudo comparativo de modelos de dados para o estudo comparativo de bancos de dados, como descrito na página 29. Isto seria feito através da inclusão dos novos grupos de axiomas SAx e PAx aos grupos de axiomas que caracterizam os modelos de dados que, por sua vez, embasam estes bancos. Este procedimento teria uma aplicação direta na área de bancos de dados heterogêneos.

Além dessas atividades, entendemos que este tipo de estudo poderia também ser aplicado a outras áreas da computação, tais como:

- no estudo teórico e na avaliação de modelos conceituais;
- na evolução de modelos de dados já existentes no sentido de adquirirem maior expressividade;
- na especificação de protocolos de comunicação de dados;

- trilhando um caminho mais utópico, na solução do problema da medição da *aderência* de pacotes ERP, tais como SAP, Oracle ou Peoplesoft, à realidade de empresas ou instituições. *Aderência* é a capacidade que um pacote desses tem de modelar diretamente as necessidades funcionais dessas instituições. Ou seja, quanto maior a aderência, menor a necessidade de customizações, e conseqüentemente menor o tempo e o custo de implantação desses pacotes. Tanto esses pacotes como o funcionamento das instituições em que os mesmos são implantados podem ser axiomatizados em teorias lógicas, se possível de primeira ordem. Assim, a aderência de um desses pacotes a uma dessas instituições pode ser medida em função da complexidade das possíveis interpretações da teoria que representa o funcionamento da instituição para a teoria que representa o pacote.

Para finalizar, de uma maneira geral o presente estudo demandou a aplicação de conceitos multidisciplinares, tais como complexidade de algoritmos, conceitos de programação, provadores de teoremas, lógica matemática, teoria de bancos de dados e até de alguns aspectos filosóficos associados à concepção dos modelos de dados.

Referências Bibliográficas

[ABITE87] S. Abiteboul e R. Hull. *IFO: A Formal Semantic Database Model.* ACM TODS V.12, 1987.

[ABITE90] S. Abiteboul et al. *New Hope on Data Models and Types.* ACM SIGMOD V.19, 1990.

[ABITE91] S. Abiteboul e A. Bonner. *Objects and Views.* ACM SIGMOD V.20, 1991.

[ABITE95] S. Abiteboul, R. Hull e V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[FERNAN95] A. A. A. Fernandes. *An Axiomatic Approach to Deductive Object-Oriented Databases.* Tese de Doutorado, Heriot-Watt University, 1995.

[ATKIN89] Atkinson et al. *The Object Oriented Database System Manifesto.* Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD), pages 40-57, 1989.

[BANCI86] F. Bancilhon. *A Calculus for Complex Objects.* ACM PODS, 1986.

[BATI91] C. Batini, S. Ceri e S. B. Navathe. *Conceptual Database Design: An Entity-Relatioship Approach.* Addison-Wesley, 1991.

[BELL77] J. Bell e M. Machover. *A Course in Mathematical Logic.* North-Holland, 1977.

[BOURB68] N. Bourbaki. *Elements of Mathematics: Theory of Sets.* Addison-Wesley, 1968.

[BRODIE84] M. L. Brodie, J. Mylopoulos e J. W. Schmidt. *On Conceptual Modelling - Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Springer-Verlag,1984.

[BUDD91] T. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[CHEN76] P. P. Chen. *The entity-relationship model. Toward a unified view of data.* ACM-TODS, 1976.

[CLOCK87] W. F. Clocksin e C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1987.

[CODD70] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks.* Communications of ACM V.13, 1970.

[ENDER72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[HALMO74] P. Halmos. *Naive Set Theory.* Springer-Verlag, 1974.

[HANNA95] M. S. Hanna. *A Close Look at the IFO Data Model.* ACM SIGMOD V.24, 1995.

[HAREL92] D. Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley, 1992.

[HOGG84] C. J. Hogger. *Introduction to Logic Programming.* Academic Press, 1984.

[KIFER95] M. Kifer et al. *Logical Foundations of Object-Oriented and Frame-Based Languages.* Journal of the ACM, 1995.

[KIM90] W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[KUPER93] G. Kuper e M. Vardi *The Logical Data Model.* ACM TODS V.18, 1993.

[MCUNE94] W. W. McCune. *Otter 3.0 Reference Manual and Guide.* Argonne National Laboratory, 1994.

[PARED87] J. Paredaens. *Databases.* ILSCS, 1987.

[REITER80] R. Reiter. *Databases: A Logical Perspective.* Em SIGPLAN Notices, Vol.16, n.1, 1981, pgs 174-176.

[REITER84] R. Reiter. *Towards a Logical Reconstruction of Relational Database Theory.* pgs. 191-233 em [BRODIE84].

[SOLOV92] V. Soloviev. *An Overview of Three Commercial OODBMS: ONTOS, ObjectStore, e O_2.* ACM SIGMOD V.21, 1992.

[SMULL94] R. M. Smullyan. *First-order Logic.* Dover Publications, Inc., 1994.

[TURSK87] W. Turski e T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.

[ZANIO97] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V. Subrahmanian, R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.