Implementação e Automação de Sistemas de Dedução

Jorge Adriano Branco Aires

Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática
2004

Dissertação apresentada à Faculdade de Ciências e Tecnologia da Universidade de Coimbra, para satisfação parcial dos requisitos do programa de Mestrado em Engenharia Informática, na especialidade de Computação Automática e Evolutiva.

Agradecimentos

Gostaria de deixar uma palavra de agradecimento aos meus orientadores Professor Doutor Pedro Quaresma de Almeida e Professor Doutor Ernesto Jorge Fernandes Costa, a todos os que de alguma forma me acompanharam ou incentivaram durante a realização deste trabalho e, a nível institucional, ao Departamento de Matemática da Universidade de Coimbra, ao Departamento de Engenharia Informática da Universidade de Coimbra, e ao Centro de Informática e Sistemas da Universidade de Coimbra.

Jorge Adriano Branco Aires

Conteúdo

Introdução

Sistemas lógicos permitem descrever e raciocinar de forma rigorosa acerca dos mais variados domínios, sejam eles teorias matemáticas abstractas como álgebra, análise ou geometria, ou áreas mais aplicadas como especificação de *software*, verificação de *hardware* ou processamento de linguagens naturais. Este rigor é conseguido à custa da utilização de linguagens bem definidas, que podem ser interpretadas de forma precisa sobre os domínios que pretendemos estudar. Assim, construíndo frases nestas linguagens lógicas, que designamos por *proposições* ou *fórmulas*, expressamos propriedades nos respectivos domínios, que poderão ser satisfeitas sempre, nunca, ou apenas mediante determinadas circunstâncias. Interessa-nos conseguir raciocinar acerca destas propriedades, ser capazes de determinar se são ou não sempre verificadas no nosso domínio de estudo, ou de um modo mais geral, se são verificadas mediante determinadas condições, ou seja, como consequência da verificação de certas propriedades no domínio. Tomando como exemplo a área da aritmética de inteiros, podemos querer tentar determinar se *"qualquer número inteiro pode ser escrito como a soma de dois quadrados perfeitos"*, ou não. Uma representação comum desta frase num sistema lógico é dada por $\forall k.\, \exists m.\, \exists n.\ k = m^2 + n^2$. Para que este tipo de estudo seja efectuado de forma rigorosa, é necessário não só a representação e interpretação precisa destas propriedades, mas uma base formal, perfeitamente definida, que sirva de suporte aos raciocínios efectuados durante esse mesmo estudo. Os *sistemas lógicos dedutivos* formam essa base. Tomando como ponto de partida relações de consequência elementares entre fórmulas (*axiomas*) e regras básicas que permitem inferir novas conclusões a partir de consequências conhecidas ou assumidas válidas num determinado contexto (*regras de inferência*), é possível criar e apresentar raciocínios de forma clara e concisa acerca dos objectos de estudo, ou seja, efectuar *demonstrações* no sistema. O objectivo deste trabalho passa por estudar *sistemas lógicos dedutivos*, com especial ênfase na sua *implementação* numa linguagem de programação (Haskell), tendo em vista a *automação* do processo de demonstração.

Este estudo é iniciado com uma abordagem teórica, que visa definir conceitos, expôr fundamentos e estabelecer resultados que serão, necessários ao longo do trabalho desenvolvido. No capítulo 1, começamos por apresentar as componentes sintáctica e semântica de sistemas lógicos, o que nos permite compreender a simbologia utilizada e desenvolver intuição acerca da mesma. Apresentamos dois tipos de linguagens lógicas, *linguagens proposicionais*, em que o domínio de estudo são simplesmente *valores de verdade*, representando por exemplo acontecimentos, e de seguida *linguagens de predicados (de primeira ordem)*, em que os valores de verdade são definidos sobre propriedades de *indivíduos* de um domínio de estudo independente

da lógica considerada. São apresentados, tanto para o caso proposicional como para o caso das linguagens de predicados, *modelos semânticos clássicos*, a sua interpretação mais comum. O estudo teórico prossegue com a análise de vários sistemas de dedução para as lógicas anteriores, acompanhado de alguns exemplos de demonstrações nos mesmos, apresentadas na forma de *árvores de demonstração*, que são caracterizadas por terem como raiz a consequência entre fórmulas (*sequente*) que pretendemos demonstrar e como folhas consequências trivias (*axiomas*) trivialmente demonstráveis, sendo que cada nó da árvore é obtido através dos seus filhos por aplicação de *regras de inferência* do sistema considerado. São estudados três tipos distintos de sistemas de dedução, *sistemas de Hilbert*, *sistemas de Dedução Natural* e *sistemas de Gentzen (Cálculo de Sequentes)*. Destes três, prosseguimos com o terceiro tipo de sistemas após destacarmos algumas das suas propriedades que simplificam a automação das regras de inferência que os definem. Nesta fase, já estudados os tratamentos semânticos e dedutivos de sistemas lógicos, exploramos a ligação entre ambos. Em particular são apresentados os conceitos de *idoneidade*, *completude*, *decidibilidade* e *semi-decidibilidade*, bem como resultados positivos e negativos envolvendo os mesmos para as lógicas clássicas. Como resultado positivo destacamos que os sistemas apresentados para as lógicas proposicional e de primeira ordem clássicas são idóneos e completos, isto é, que demonstrações nestes sistemas dedutivos estão correctas a nível semântico e que qualquer consequência semântica entre fórmulas pode ser demonstrada. Como resultado negativo, nota-se que, ao contrário do que acontece para lógica proposiconal, não existe nenhum sistema decidível para lógica de primeira ordem, isto é, não é possível no caso geral, criar um algoritmo que consiga, num número finito de passos, decidir se existe ou não uma consequência lógica entre fórmulas de primeira ordem. Este resultado é muito importante pois estabelece um limite a nível das capacidades de demonstração automática em lógica de primeira ordem. Notamos, no entanto, que lógica de primeira ordem clássica é semi-decidível, isto é, nos casos em que uma demonstração existe ela pode ser determinada num número finito de passos. O estudo prossegue com a definição de *teorias* sobre sistemas lógicos, ilustrado com a axiomatização da estrutura algébrica *grupo* e demonstração de algumas propriedades desta estrutura matemática na teoria construída. O primeiro capítulo termina com a apresentação de *substituições*, ou seja, correspondências (mapeamentos) definidas de variáveis em termos. É dada especial atenção a *unificadores* de pares de termos, isto é, substituições que quando aplicadas a dois termos distintos resultam numa *instância* comum a ambos. Substituições e unificadores desempenham um papel muito importante em sistemas de dedução automática e como tal o seu estudo irá ser aprofundado mais tarde, no capítulo 3.

O conteúdo do capítulo 2 é mais técnico, neste debruçamo-nos principalmente sobre a implementação das estruturas que estão na base de um sistema de demonstração, nomeadamente *termos* e *fórmulas*. Para isso, começamos por apresentar brevemente algumas das características mais importantes de Haskell, uma linguagem de programação funcional fortemente tipada, pura e não estrita, utilizada ao longo deste trabalho para implementar muitas das técnicas apresentadas. De seguida é descrita uma biblioteca de operações auxiliares, implementadas nesta linguagem, definida sobre uma classe de estruturas recursivas, bem como a sua generalização para estruturas mutuamente recursivas. As operações implementadas baseiam-se apenas na estrutura

(recursiva) dos tipos de dados a que são aplicadas, ou seja, nos métodos das classes definidas, que são essencialmente a *projecção* e *actualização* dos *subdados* das estruturas consideradas. Entre as funções disponibilizadas encontram-se *paramorfismos*, *mapeamentos*, *esmagamentos* generalizações de *zips* e as suas variantes *monádicas*. Estas bibliotecas foram utilizadas durante a realização deste trabalho para manipular as estruturas de dados *termos* e *fórmulas*, bem como *multitermos* e *multiequações* utilizados por alguns métodos de unificação descritos no terceiro capítulo. Antes de definirmos a implementação das estruturas de dados *termo* e *fórmula* analisamos parcialmente o processo de demonstração automática para definir alguns detalhes de representação. As demonstrações automáticas serão efectuadas partindo da consequência entre fórmulas que pretendemos demonstrar e aplicando as regras de inferência no sentido inverso por forma a construir a árvore de demonstração da raiz (sequente original) para as folhas (axiomas). Esta aplicação inversa das regras, no caso da lógica de primeira ordem, poderá não ser univocamente definida, e requerer que sejam calculados termos que permitam terminar o processo com sucesso. São então introduzidos, para auxiliar esse cálculo *metavariáveis*, que representam termos ainda não definidos, e *parâmetros* para associar informação extra às variáveis livres. Apresentamos ainda a notação de *de Bruijn* que será utilizada na representação de variáveis ligadas para facilitar alguns algoritmos. O cálculo dos valores para as metavariáveis irá ter por base métodos de unificação, e é este o assunto a que nos dedicamos no capítulo seguinte.

No capítulo 3 são estudados algoritmos de unificação e estruturas que permitem simplificar ou optimizar estes métodos. Começamos por apresentar o algoritmo de Robinson, o mais simples dos algoritmos de unificação, baseado na travessia por recursão estrutural dos termos a unificar. São focados a forma como é calculado o unificador, em particular a necessidade da aplicação das substituições intermédias calculadas aos subtermos dos termos a unificar, e como são verificadas eventuais *incompatibilidades estruturais* dos termos a unificar ou a impossibilidade de unificação devido à *ocorrência* de variáveis no interior de subtermos com os quais teriam que ser associadas. De seguida descrevemos uma implementação deste algoritmo em Haskell, onde são abordados aspectos como o *tratamento dos erros* que podem surgir durante o processo de unificação através de estruturas *monádicas*, o controlo eficiente da aplicação das substituições calculadas aos subtermos dos termos a unificar, e variações na forma como são aplicados os testes *estruturais* e de *ocorrência*. Dedicamo-nos ainda à derivação e optimização do cálculo da composição de substituições. A optimização passa por uma representação das susbtituições sob a forma da lista de substituições elementares que a determinam (*forma triangular*). É também descrita a implementação desta representação e de operações sobre a mesma. Na secção seguinte apresentamos o algoritmo de unificação de Martelli e Montanari. Trata-se de um algoritmo mais complexo que o de Robinson, em que o problema da unificação de termos é modelado através de um sistema de equações e resolvido de forma eficiente através de representações compactas das equações que o compõem (*multiequações*) bem como dos termos de cada equação (*multitermos*), e de um desenvolvimento inteligente do sistema que permite evitar alguns dos passos mais complexos do algoritmo de Robinson, nomeadamente a instanciação de subtermos com as substituições calculadas e a verificação de ocorrências. Para terminar este capítulo apresentamos métodos de indexação de termos, que visam facilitar sua unificação com um outro termo (deno-

minado *query term*). Estas indexações permitem selecionar rapidamente termos potencialmente unificáveis com o *query term*, selecção essa que, dependendo do método, pode ser perfeita (se o resultado for exactamente o conjunto de termos unificáveis), ou não. São apresentados os métodos de *indexação por discriminação*, na sua variante perfeita e imperfeita, que é baseado em partilha de prefixos comuns, *indexação por abstracção* que tira partido da relação de instanciação entre termos e substituições, e *indexação por substituição* que combina os dois métodos anteriores.

No capítulo 4 continuamos a desenvolver as técnicas de demonstração automática cujo estudo teve início no segundo capítulo. Numa primeira fase analisamos as regras de inferência do sistema de dedução. Começamos por verificar a necessidade de duplicação de fórmulas durante o processo de aplicação inversa das regras de inferência e estudar as consequências desta duplicação a nível de terminação do algoritmo. Exploramos também a selecção das regras base a serem utilizadas, que pode influênciar a complexidade de demonstrações. Este aspecto é ilustrado com a introdução de uma regra específica para a conectiva de equivalência ($\Leftrightarrow$). Aprofundamos ainda a representação de fórmulas baseada em metavariáveis e parâmetros através da adição de informação acerca destes ao nível dos sequentes. A modificação resultante na forma como são aplicadas as regras de inferência é conhecida por *cálculo de sequentes restrictos*. Após este estudo, são descritos detalhes de implementação da representação das regras de inferência e representação dos sequentes. Numa segunda fase debruçamo-nos sobre a construção da demonstração propriamente dita. Discutimos dois aspectos essenciais, como escolher a fórmula de um sequente à qual iremos aplicar uma regra de inferência (*fórmula principal*), e por que ordem desenvolver os nós da árvore de demonstração. A nível da escolha das fórmulas principais são tidos em conta aspectos como a complexidade das regras de inferência correspondentes, a necessidade de garantir que todas as fórmulas são eventualmente escolhidas, e as vantagens da criação de novas fórmulas atómicas por aplicação das regras de inferência. Relativamente ao desenvolvimento dos nós da árvore de demonstração são tidos em conta essencialmente aspectos relacionados com a quantidade de memória necessária e completude de do processo. Apresentamos três estratégias básicas, *construção em largura*, um método completo mas nada prático devido ao elevado consumo de memória, *construção em profundidade* um método incompleto mas cujo consumo de memória é mais aceitável, e *construção heurística* baseado em avaliações heurísticas dos nós da árvore e que apresenta algumas vantagens sobre o primeiro método apesar do consumo de memória poder ser também proibitivo. São analisadas ainda duas variantes dos métodos anteriores, *construção heurística em profundidade* que procura um equilíbrio entre a utilização de avaliações heurística dos nós (reduzindo a incompletude do processo) e um consumo de memória controlado conseguido através de uma estratégia de construção predominantemente em profundidade, e *construção em profundidade iterada*, uma estratégia completa que consiste em sucessivas construções em profundidade até a um nível de profundidade máximo cada vez mais elevado. Estudamos ainda a forma como são tratadas as aplicações dos unificadores calculados durante a demonstração, bem como as possíveis consequências de serem considerdas apenas parte das possíveis instanciações. Este capítulo termina com a referência a alguns detalhes de implementação do processo de construção da árvore de demonstração.

Capítulo 1

Fundamentos

Neste capítulo são apresentados os fundamentos teóricos dos sistemas de dedução. Na secção 1.1 definimos linguagens lógicas proposicionais e linguagens de predicados de primeira ordem, (a componente sintáctica de um sistema lógico). Mostramos ainda como associar semântica a essas mesmas linguagens, através da definição de modelos, que permitem interpretar e avaliar fórmulas sobre determinados domínios, e definimos sistemas lógicos semânticos com base na relação de consequência semântica entre fórmulas num modelo. As lógicas apresentadas são os casos clássicos, proposicional e de primeira ordem. Na secção seguinte, 1.2, são definidos sistemas lógicos dedutivos, onde a consequência entre fórmulas é definida de forma construtiva, através de regras. São apresentados os sistemas dedutivos de Hilbert, de Dedução Natural, e de Gentzen, para a lógica clássica, e é observado que os últimos são os que permitem uma mecanização mais simples do processo de demonstração. Na secção 1.3 aprofundamos a relação entre sistemas lógicos semânticos e dedutivos. Em particular, são definidos os conceitos de idoneidade, completude e decidibilidade. Na secção 1.4 definimos Teorias, que permitem modelar ou axiomatizar domínios com propriedades associadas. Finalmente, na secção 1.5, definimos substituições, unificadores e outros conceitos associados.

1.1 Linguagens Lógicas e Modelos

Começamos, nesta secção, por apresentar o tipo de linguagem lógica mais simples, a *linguagem lógica proposicional*.

1.1.1 Linguagens Proposicionais

As linguagens proposicionais definem estruturas (palavras) denominadas proposições, com base em símbolos, (constantes ou variáveis), proposicionais, e conectivas que os permitem combinar. Vejamos a sua definição formal.

Definição 1.1 (Linguagem Proposicional, Proposições) *Uma Linguagem Proposicional $\mathcal{L}$ é definida por um par (P, C), com P um conjunto de variáveis ditas proposicionais, e $C = \{C_n\}_{n\in\mathbb{N}_0}$ uma família de conjuntos de símbolos lógicos (conectivas) de aridade n. O conjunto*

$Prop_{\mathcal{L}}$ das proposições de L é o conjunto C-gerado sobre P [8]*, ou seja, é definido indutivamente por:*

- *se $p \in P$ então $p \in Prop_{\mathcal{L}}$,*
- *se $k \in C_0$ então $k \in Prop_{\mathcal{L}}$,*
- *se $c \in C_n$, $n > 0$ e $A_1, \ldots, A_n \in Prop_{\mathcal{L}}$, então $c(A_1, \ldots, A_n) \in Prop_{\mathcal{L}}$.*

Os elementos de $P \cup C_0$ dizem-se proposições atómicas.

É comum a utilização de operadores em *prefixo* ou *posfixo* para conectivas unárias, e de operadores *infixo* para conectivas binárias.

Neste caso, para evitar ambiguidade nas expressões, é necessário o uso de parêntesis. Para tornar a notação mais leve, é habitualmente definida uma prioridade para cada conectiva, e uma associatividade (esquerda ou direita) para as conectivas binárias. O caso clássico, mais comum, é apresentado de seguida.

Definição 1.2 (Linguagem Proposicional Clássica) *A linguagem proposicional clássica é definida por $\mathcal{L}_{pc} = (P, C)$, com variáveis proposicionais $P = \{a, \ldots, z, a_1, \ldots, z_1, a_2, \ldots, z_2, \ldots\}$, e conectivas $C_0 = \{\bot\}$, $C_1 = \{\neg\}$, $C_2 = \{\wedge, \vee, \Rightarrow\}$, e $C_n = \emptyset$, para $n > 2$.*

As conectivas denominam-se *falso* ($\bot$), *negação* ($\neg$), *conjunção* ($\wedge$), *disjunção* ($\vee$) e *implicação* ($\Rightarrow$). São escritas em notação *prefixa* ($\neg$) e *infixa* ($\wedge$,$\vee$,$\Rightarrow$) e estão aqui apresentadas por ordem decrescente de prioridade. As conectivas binárias, por omissão, associam à direita. São exemplos de proposições nesta linguagem,

1. $\bot$
2. p
3. $p \vee (r \Rightarrow \bot)$

4a. $(p \vee r) \Rightarrow \bot$

4b. $p \vee r \Rightarrow \bot$

4. $\neg(p \vee q) \Rightarrow \neg p \wedge \neg q$.

As expressões 4a e 4b representam a mesma proposição. As proposições 1 e 2 são atómicas, e todas as restantes não atómicas.

Proposições são apenas construções sintáticas, sem nenhum significado particular. A associação de semântica a proposições obtém-se por *interpretação* das conectivas de aridade 0 como constantes, e das conectivas de aridade superior a 0 como funções, num determinado domínio. Os elementos desse domínio denominam-se *valores de verdade*. A *avaliação* de proposições com base numa interpretação é efectuada *atribuindo* valores de verdade do domínio considerado às variáveis usadas. Em seguida são definidos de forma rigorosa os conceitos que acabámos de descrever.

Definição 1.3 (Atribuição, Interpretação, Avaliação) *Considere-se uma linguagem proposicional $\mathcal{L} = (P, C)$ e um conjunto M de elementos denominados valores de verdade. Uma atribuição $a : P \to M$, faz corresponder a cada elemento $p \in P$ um elemento $a(p) \in M$. Uma interpretação $v : C \to M$, faz corresponder a cada constante proposicional $k \in C_0$ um valor de*

verdade $M_k \in M$ e a cada conectiva $c \in C_n$, com $n > 0$, uma função $M_c : M^n \to M$. Fixadas uma atribuição $a : V + P \to I + M$ e uma interpretação $v : O + C \to I + M$, a sua extensão $v_a : Prop_L \to M$, denominada avaliação, faz corresponder, a cada proposição, um valor de verdade, e é indutivamente definida por:

- $v_a(p) = a(p)$ *para,* $p \in P$,
- $v_a(k) = v(k)$ *para,* $k \in C_0$,
- $v_a(c(A_1, \ldots, A_n)) = v(c)(v_a(t_1), \ldots, v_a(t_n))$, *para* $c \in C_n$ *com* $n > 0$ *e* $t_1, \ldots, t_n \in Prop_{\mathcal{L}}$.

Fixados um domínio de valores de verdade e uma interpretação sobre o mesmo, dizemos que temos um *Modelo* ou *Sistema de Avaliação*.

Definição 1.4 (Modelo, Sistema de Avaliação) *Um Modelo, ou Sistema de Avaliação, para uma Linguagem Proposicional $\mathcal{L} = (P, C)$ consiste num tuplo $\mathcal{M} = (\mathcal{L}, M, D, v)$, formado por uma linguagem $\mathcal{L}$, um conjunto M de valores de verdade, um conjunto $D \subseteq M$ de valores designados, e uma interpretação $v : C \to M$. Avaliações no Modelo $\mathcal{M}$ são avaliações da forma v_a.*

O subconjunto D de valores designados é contituído pelos valores de verdade considerados satisfatórios. Nos casos mais simples temos apenas dois valores de verdade, *falso* (f) e *verdadeiro* (t), e um único valor designado, o *verdadeiro*. É o que acontece no modelo clássico para a linguagem proposicional clássica.

Definição 1.5 (Modelo Clássico) *O modelo clássico para lógica proposicional clássica é definido por $\mathcal{M}_{pc} = (\mathcal{L}_{pc}, \{f, t\}, \{t\}, v)$, onde $v(c) = \mathcal{M}_c$ para toda a conectiva c, com $\mathcal{M}_c$ definido pelas tabelas:*

$\mathcal{M}_\bot$	
	f

$\mathcal{M}_\neg$	
$\boldsymbol{f}$	t
$\boldsymbol{t}$	f

$\mathcal{M}_\wedge$	$\boldsymbol{f}$	$\boldsymbol{t}$
$\boldsymbol{f}$	f	f
$\boldsymbol{t}$	f	t

$\mathcal{M}_\vee$	$\boldsymbol{f}$	$\boldsymbol{t}$
$\boldsymbol{f}$	f	t
$\boldsymbol{t}$	t	t

$\mathcal{M}_\Rightarrow$	$\boldsymbol{f}$	$\boldsymbol{t}$
$\boldsymbol{f}$	t	t
$\boldsymbol{t}$	f	t

Portanto o modelo clássico para lógica proposicional clássica é baseado no domínio bivalor já descrito, e numa interpretação v que ao falso (sintático, $\bot$) associa o falso (semântico, f), à negação uma operação unária que inverte um valor de verdade, à conjunção ($\wedge$) uma operação binária que devolve verdadeiro apenas quando ambos os argumentos são verdadeiros, à disjunção ($\vee$) uma operação que devolve verdadeiro se pelo menos um dos argumentos o for, e à implicação ($\Rightarrow$) uma operação que, para devolver verdadeiro, exige que o segundo argumento seja verdadeiro quando o primeiro o for. Como exemplo, no modelo descrito, a avaliação de $\bot \vee \neg\neg\bot$, é,

$$\begin{aligned} & v(\bot \vee \neg\neg\bot) \\ = \; & \mathcal{M}_\vee(\mathcal{M}_\bot, \mathcal{M}_\neg(\mathcal{M}_\neg(\mathcal{M}_\bot))) \\ = \; & \mathcal{M}_\vee(\mathrm{f}, \mathcal{M}_\neg(\mathcal{M}_\neg(\mathrm{f}))) \\ = \; & \mathcal{M}_\vee(\mathrm{f}, \mathcal{M}_\neg(\mathrm{t})) \\ = \; & \mathcal{M}_\vee(\mathrm{f}, \mathrm{f}) \\ = \; & \mathrm{f} \end{aligned}$$

Dada uma atribuição a, a avaliação da expressão $p \vee \neg\neg q$, será,

$$\begin{aligned} & v_a(p \vee \neg\neg q) \\ = \; & \mathcal{M}_\vee(a(p), \mathcal{M}_\neg(\mathcal{M}_\neg(a(q)))) \end{aligned}$$

Se a for tal que $a(p) = \mathrm{t}$ e $a(q) = \mathrm{f}$, ter-se-á,

$$\begin{aligned} & \mathcal{M}_\vee(\mathrm{t}, \mathcal{M}_\neg(\mathcal{M}_\neg(\mathrm{f}))) \\ = \; & \mathcal{M}_\vee(\mathrm{t}, \mathcal{M}_\neg(\mathrm{t})) \\ = \; & \mathcal{M}_\vee(\mathrm{t}, \mathrm{f}) \\ = \; & \mathrm{t} \end{aligned}$$

Avaliações podem ser representadas, de forma mais conveniente, através de *tabelas de verdade*. Apresentamos o valor das atribuições para as variáveis proposicionais, bem como das avaliações dos subtermos da proposição considerada, e finalmente da proposição que queremos avaliar.

$a(p)$	$a(q)$	$v_a(\neg q)$	$v_a(\neg\neg q)$	$v_a(p \vee \neg\neg q)$
t	f	t	f	t

Tabela 1.1: tabela de verdade

Por vezes são consideradas outras conectivas em lógica proposicional clássica, como equivalência ($\Leftrightarrow$) e disjunção exclusiva ($\dot\vee$). Estas satisfazem $v_a(p \Leftrightarrow q) = v_a(p \Rightarrow q \wedge q \Rightarrow p)$ e $v_a(p \mathbin{\dot\vee} q) = v_a(\neg(p \Leftrightarrow q))$. Ou seja, as conectivas $\Leftrightarrow$ e $\dot\vee$ são interpretadas como funções que devolvem, verdadeiro quando os seus argumentos tiverem o mesmo valor de verdade, e valores de verdade distintos, respectivamente.

1.1.2 Linguagens de Predicados

Linguagens proposicionais servem para modelar domínios, cujos os elementos são eles próprios valores de verdade, e portanto não são adequadas para formular propriedades de domínios genéricos. Neste último caso precisamos de trabalhar com dois tipos de entidades, os valores de verdade e os elementos do domínio que queremos estudar, e os primeiros devem ser de alguma forma obtidos a partir dos últimos. Quando é esse o nosso objectivo utilizamos linguagens de predicados.

Definição 1.6 (Linguagem de Predicados, Fórmulas) *Uma Linguagem de Predicados (de primeira ordem) $\mathcal{L}$ é definida por um tuplo (V, O, P, C, Q), onde V é um conjunto de variáveis, $O = \{O_n\}_{n\in\mathbb{N}_0}$ uma família de conjuntos de operações de aridade n, $P = \{P_n\}_{n\in\mathbb{N}_0}$ uma família de conjuntos de predicados de aridade n, $C = \{C_n\}_{n\in\mathbb{N}_0}$ uma família de conjuntos de conectivas de aridade n, e Q um conjunto de quantificadores. Define-se então o conjunto $Term_\mathcal{L}$ dos termos de $\mathcal{L}$ como sendo o conjunto O-gerado sobre V, ou seja:*

- *se* $x \in V$ *então* $x \in Term_{\mathcal{L}}$,
- *se* $a \in C_0$ *então* $a \in Term_{\mathcal{L}}$,
- *se* $o \in O_n$, $n > 0$ *e* $t_1, \ldots, t_n \in Term_{\mathcal{L}}$, *então* $o(t_1, \ldots, t_n) \in Term_{\mathcal{L}}$.

O conjunto $Form_L$, *das fórmulas de* $\mathcal{L}$, *é definido indutivamente por:*

- *se* $p \in P_0$ *então* $p \in Form_{\mathcal{L}}$,
- *se* $p \in P_n$, $n > 0$ *e* $t_1, \ldots, t_n \in Term_{\mathcal{L}}$, *então* $p(t_1, \ldots, t_n) \in Form_{\mathcal{L}}$,
- *se* $k \in C_0$ *então* $k \in Form_{\mathcal{L}}$,
- *se* $c \in C_n$, $n > 0$ *e* $A_1, \ldots, A_n \in Form_{\mathcal{L}}$, *então* $c(A_1, \ldots, A_n) \in Form_{\mathcal{L}}$,
- *se* $q \in Q$, $x \in V$ *e* $A \in Form_{\mathcal{L}}$, *então* $q\ x.A \in Form_{\mathcal{L}}$.

Os termos em $V \cup O_0$ *dizem-se atómicos. As fórmulas em* C_0, *em* P_0, *ou da forma* $p(t_1, \ldots t_n)$ *com* $p \in P_n$, *dizem-se atómicas.*

Uma linguagem de predicados, define então elementos de duas espécies distintas, os *termos*, que são construídos através das variáveis em V e constantes e operadores em O, e as *fórmulas*, obtidas na sua forma mais elementar por aplicação de predicados a termos, ou por constantes lógicas (conectivas de aridade 0), e que podem ser combinadas entre si para dar origem a outras mais complexas por aplicação de conectivas e quantificadores. A definição limita-se a linguagens de predicados ditas de primeira ordem, ou seja, em que a quantificação ocorre apenas sobre variáveis.

Definição 1.7 (Linguagem de Predicados de Primeira Ordem Clássica) *Linguagens de predicados de primeira ordem clássica são da forma* $\mathcal{L}_{poc} = (V, O, P, C, Q)$, *onde* $C_0 = \{\bot\}$, $C_1 = \{\neg\}$, $C_2 = \{\wedge, \vee, \Rightarrow\}$, *e* $C_n = \emptyset$, *para* $n > 2$, $Q = \{\forall, \exists\}$, *e* V, O, P *são quaisquer conjuntos de variáveis, famílias de conjuntos de operadores e famílias de conjuntos de predicados, respectivamente.*

O que define uma linguagem de primeira ordem clássica são as conectivas, que são as utilizadas na linguagem proposicional clássica, e os quantificadores, que são $\forall$ (universal), e $\exists$ (existencial). As expressões seguintes são fórmulas sobre $\mathcal{L}_{poc}$, com variáveis $\{x, y, z\} \subseteq V$, operações $e \in O_0$, $f \in O_1$ e $g \in O_2$, e predicados $Q \in P_1$ e $R \in P_2$.

1. $R(x, y)$
2. $Q(g(x, y)) \Rightarrow Q(z)$
3. $\forall x.\, R(x, f(y))$
4. $\forall x.\, Q(x) \Rightarrow \exists y.\, Q(y)$
5. $(\forall x.\, Q(x)) \Rightarrow (\exists y.\, Q(y))$
6. $\forall x.\, \exists y.\, R(g(x, y), e)$.

A prioridade e associatividade das conectivas é a mesma que no caso proposicional. Quanto aos quantificadores, estes são definidos como tendo prioridade mínima sobre o segundo argumento. Ou seja, a menos que se utilizem parêntesis, a fórmula, argumento do quantificador,

é composta por tudo o que se segue à variável quantificada. Como exemplo, nas fórmulas anteriores, no caso 4, $\forall x$ é aplicado a $Q(x) \Rightarrow \exists y.\, Q(y)$ e, no caso 5, $\forall x$ é aplicado apenas a $Q(x)$.

O *alcance* de um quantificador é toda a porção da fórmula, argumento desse mesmo quantificador, que não seja argumento de um outro quantificador aplicado a uma variável igual à do primeiro. Por exemplo, em $\forall x.\, Q(x) \wedge \exists y.\, R(x,y)$, o *alcance* do primeiro quantificador é $Q(x) \wedge \exists y.\, R(x,y)$, mas em $\forall x.\, Q(x) \wedge \exists x.\, R(x,y)$ apenas $Q(x)$ está sobre o alcance do primeiro quantificador. No primeiro caso as variáveis x em $Q(x)$ e em $R(x,y)$ dizem-se *ligadas* ao quantificador $\forall$, e a variável y a $\exists$. No segundo caso a variável x em $Q(x)$ está *ligada* ao quantificador $\forall$, a variável x em $R(x,y)$ está *ligada* ao quantificador $\exists$, e a variável y em $R(x,y)$ diz-se *livre*, pois não está no alcance de nenhum quantificador aplicado a uma variável y.

Definição 1.8 (Fórmulas Abertas e Fechadas) *Uma fórmula A diz-se fechada se não contiver nenhuma variável livre, caso contrário diz-se aberta.*

Estão definidos os aspectos sintácticos da linguagem de predicados de primeira ordem, em particular o caso clássico. A associação de semântica a estas linguagens é semelhante ao caso proposicional. Para linguagens de primeira ordem, interpretações e atribuições serão definidas a dois níveis, o dos termos, sobre um domínio I de indivíduos cujas propriedades queremos estudar, e o nível das fórmulas, sobre um domínio M de valores de verdade. Os operadores e conectivas serão interpretados como funções internas em I e M, respectivamente. A ligação entre ambos os domínios é dada pelos predicados e quantificadores. A cada predicado será atribuida uma função que vai de indívíduos num valor de verdade, ou seja, predicados representam propriedades dos indivíduos. Quantificadores permitem definir propriedades sobre colecções de termos, considerando variáveis 'genéricas' (as variáveis ligadas). Estes conceitos são formalmente definidos de seguida.

Definição 1.9 (Atribuição, Avaliação, Interpretação (linguagem de predicados)) *Sejam $\mathcal{L} = (V, O, P, C, Q)$ uma linguagem de predicados, M um conjunto de elementos denominados valores de verdade, e I um conjunto de elementos denominados indivíduos. Uma atribuição $a : V + P \to I + M$ faz corresponder a cada $x \in V$ um indivíduo $a(x) \in I$ e a cada predicado $p \in P_n$, uma função $a(p) : I^n \to M$. Uma interpretação $v : O + C \to I + M$ faz corresponder a cada operador $o \in O_n$ uma função $I_o : I^n \to I$, a cada constante proposicional $k \in C_0$ um valor de verdade $M_k \in M$, a cada conectiva $c \in C_n$, com $n > 0$, uma função $M_c : M^n \to M$ e a cada quantificador $q \in Q$ uma função $M_q : \wp(M) \to M$.*

Fixadas uma atribuição $a : P \to M$ e uma avaliação $v : P \to M$, a sua extensão $v_a : Term_{\mathcal{L}} + Form_{\mathcal{L}} \to I + M$, denominada interpretação, é definida indutivamente para elementos de espécie termo, por:

- $v_a(k) = I_k$, *para* $k \in I_0$,
- $v_a(x) = a(x)$, *para* $x \in V$,
- $v_a(o(t_1, \ldots, t_n)) = v(o)(v_a(t_1), \ldots, v_a(t_n))$, *para* $o \in I_n$ *com* $n > 0$ *e* $t_1, \ldots, t_n \in Term_{\mathcal{L}}$.

Para elementos de espécie fórmula, a definição indutiva é:

- $v_a(p) = a(p)$, *para* $p \in P$,
- $v_a(k) = v(k)$, *para* $k \in C_0$,
- $v_a(c(A_1, \ldots, A_n)) = v(c)(v_a(A_1), \ldots, v_a(A_n))$, *para* $c \in C_n$ *com* $n > 0$ *e* $A_1, \ldots, A_n \in Form_L$,
- $v_a(q\, x.\ A) = v(q)(\{v'_a(A)\ |a' \sim_x a\})$, *para* $q \in Q$, $x \in V$ *e* $A \in Form_L$.

Escrevemos $a' \sim_x a$ *se e só se,* $a'(p) = a(p)$, *para todo o predicado* p, *e* $a(y) = a'(y)$ *para toda a variável* $y \neq x$ [33].

De forma análoga ao caso proposicional, define-se um modelo $\mathcal{M}$ para uma dada linguagem de predicados $\mathcal{L}$, fixando domínios I e M, um conjunto de valores designados $D \subseteq M$ e, uma avaliação v sobre os mesmos. Repare-se que modelos proposicionais estão 'contidos' em modelos clássicos de predicados de primeira ordem. As proposições são dadas por fórmulas sem quantificadores e apenas com predicados de aridade 0.

Definição 1.10 (Modelo Clássico para Lógica de Primeira Ordem) *O modelo clássico para lógica de predicados clássica é definido por* $\mathcal{M}_{lpc} = (\mathcal{L}_{pc}, I, \{f, t\}, \{t\}, v)$, *onde* $v(c) = \mathcal{M}_c$ *para toda a conectiva* c *(ver definição 1.5), e* $v(\forall) = M_\forall$ *e* $v(\exists) = M_\exists$, *com,*

$$M_\forall(X) = \begin{cases} \text{f} & \text{se} \quad \text{f} \in X \\ \text{t} & \text{se} \quad \text{f} \notin X \end{cases} \qquad M_\exists(X) = \begin{cases} \text{t} & \text{se} \quad \text{t} \in X \\ \text{f} & \text{se} \quad \text{t} \notin X \end{cases}$$

A definição de modelo clássico para lógica de primeira ordem está parametrizada sobre o conjunto de indivíduos I e a interpretação dos termos. Podemos ter lógicas de primeira ordem definidas sobre vários conjuntos de indivíduos, e de várias formas. Tomemos como exemplo, a fórmula $P(g(x,k),k)$, e o modelo clássico com $I = \mathbb{N}_0$, $v(k) = 0$ e $v(g) = +$. Intuitivamente a expressão será interpretada como, *"a soma de x com 0 satisfaz uma propriedade* $P(x,0)$*"*. Formalmente temos,

$$\begin{aligned} & v_a(P(g(x,k),k)) \\ = \ & a(P)(a(x) + 0, 0) \\ = \ & a(P)(a(x), 0) \end{aligned}$$

Se a atribuição escolhida for tal que $a(P)$ seja a relação de desigualdade $>$ e $a(x) = 4$, ficamos com $4 > 0$, e portanto a avaliação é verdadeira. Por outro lado, será falsa se escolhermos a tal que $a(P)$ seja, como anteriormente, a relação $>$, mas $a(x) = 0$, pois não se tem $0 > 0$. Consideremos agora a fórmula $\forall x.\, P(g(x,k),k)$, e o modelo anterior. Intuitivamente a expressão será interpretada como, *"para todo o número natural x, a sua soma com 0 satisfaz a propriedade*

$P(x, 0)$". Formalmente temos,

$$\begin{aligned} & v_a(\forall x.\, P(g(x,k),k)) \\ = \;& M_\forall(v'_a(\forall x.\, P(g(x,k),k)) \mid a' \sim_x a\}) \\ = \;& M_\forall(\{a'(P)(a'(x)+0,0) \mid a' \sim_x a\}) \\ = \;& M_\forall(\{a'(P)(a'(x),0) \mid a' \sim_x a\}) \\ = \;& M_\forall(\{a(P)(y,0) \mid y \in \mathbb{N}_0\}) \end{aligned}$$

Se tal como anteriormente considerarmos uma avaliação a tal que $a(P)$ seja a relação de desigualdade $>$ então teremos,

$$M_\forall(\{y > 0 \mid y \in \mathbb{N}_0\})$$

que, tal como esperavamos, não depende da atribuição que consideramos para x pois este está quantificado, e é falso pois o conjunto, argumento de $M_\forall$, inclui pelo menos um valor falso (uma vez que para $y = 0$ se tem $0 > 0$). Se em vez de $\forall$ tivessemos usado o quantificador $\exists$ a expressão considerada seria verdadeira, pois existem valores $y \in \mathbb{N}_0$ tais que $y > 0$.

1.1.3 Consequência Semântica

Tendo como base um modelo $\mathcal{M}$ para uma linguagem proposicional ou de primeira ordem $\mathcal{M}$, podemos querer saber se, quando um conjunto de proposições ou fórmulas $A_1, \ldots, A_n$ são verdadeiras segundo uma avaliação v_a arbitrária, também o é uma determinada fórmula A. Ou seja se a fórmula A, é uma consequência das fórmulas $A_1, \ldots, A_n$. Este conceito de consequência de fórmulas num determinado modelo é definido formalmente através da relação $\vDash$, dita *consequência semântica*.

Definição 1.11 (Consequência Semântica, $\Gamma \vDash A$) *Dado um modelo $\mathcal{M}$ diz-se que uma fórmula A é consequência semântica do conjunto de fórmulas Γ em $\mathcal{M}$, e escreve-se $\Gamma \vDash_{\mathcal{M}} A$, se para toda a avaliação v_a tal que $v_a(F) \in D$ para todo o $F \in \Gamma$, se tem $v_a(A) \in D$. A relação $\vDash_{\mathcal{M}}$, denomina-se relação de consequência semântica.*

Os conjuntos Γ serão denotados por sequências de fórmulas separadas por vírgulas, em particular o conjunto vazio será denotado por uma sequência vazia. Vejamos alguns exemplos, no modelo clássico. Temos que $P(x), Q, R(x,y) \vDash P(x)$, é uma consequência semântica trivial pois qualquer avaliação das fórmulas do membro esquerdo de $\vDash$ que seja verdadeira, o será em particular para $P(x)$. Outro exemplo trivial é $\bot \vDash Q$, uma vez que não existe nenhuma avaliação para a qual Q seja verdadeiro. São consequências semânticas não triviais, por exemplo, $P \Rightarrow Q(x), Q(x) \Rightarrow R(x,y) \vDash P \Rightarrow R(x,y)$, e também $\forall x.\, Q(x) \vDash Q(k)$, que podem ser confirmadas por análise da definição da interpretação das conectivas $\Rightarrow$ e $\forall$.

Definição 1.12 (Tautologia, $\models_{\mathcal{M}} A$) *Uma fórmula A diz-se uma tautologia em $\mathcal{M}$ se $v_a(A) \in D$, para toda a avaliação v_a sobre $\mathcal{M}$, ou seja, se $\models_{\mathcal{M}} A$.*

São exemplos de tautologias, ou seja, fórmulas que são consequências de um conjunto vazio de fórmulas, $P \Rightarrow P$, bem como $\forall x.\, Q(x) \Rightarrow \neg\exists x.\, Q(x)$.

Definição 1.13 (Sistema Lógico Semântico) *Um sistema lógico semântico $\mathcal{S}$ é um par $(\mathcal{L}, \models)$, onde $\mathcal{L}$ é uma linguagem lógica e $\models$ uma relação de consequência semântica sobre essa linguagem.*

Sistemas lógicos semânticos são utilizados quando o foco é estabelecer relações entre a validade de fórmulas num determinado modelo.

1.2 Sistemas de Dedução

O nosso objectivo é conseguir determinar se uma fórmula A é ou não consequência de um conjunto de fórmulas Γ num determinado sistema lógico. Consideremos como exemplo a definição anterior para Lógica Proposicional Clássica, usando um sistema de avaliação. Determinar se $\Gamma \models A$, pode ser conseguido através de *tabelas de verdade*. O método consiste em testar todas as atribuições resultantes das combinações de valores de verdade para as variáveis proposicionais em $A \cup \Gamma$. Se as avaliações correspondentes que devolvam verdadeiro para todas as fórmulas em Γ, devolverem também verdadeiro para A, então $\Gamma \models A$ (tabela 1.2).

$a(p)$	$a(q)$	$a(r)$	$v_a(p \Rightarrow q)$	$v_a(q \Leftrightarrow r)$	$v_a(\neg p \vee r)$
f	f	f	**t**	**t**	**t**
f	f	t	t	f	t
f	t	f	t	f	t
f	t	t	**t**	**t**	**t**
t	f	f	f	t	f
t	f	t	f	f	t
t	t	f	t	f	f
t	t	t	**t**	**t**	**t**

Tabela 1.2: Tabela de verdade com demonstração de $p \Rightarrow q, q \Leftrightarrow r \models \neg p \vee r$.

Se, para o caso de Lógica Proposicional Clássica, gerar as avaliações necessárias é um processo extremamente ineficiente, uma vez que o número de casos a testar cresce exponencialmente com o número de variáveis, para outros sistemas lógicos isso é mesmo impossível. É, por exemplo, o caso de Lógica de Primeira Ordem. De seguida iremos construir sistemas lógicos que, em vez de terem por base uma relação de consequência semântica (resultante de interpretações e testada com avaliações), se baseiam em *relações de dedução*. Estas relações, denotadas por $\vdash$, são definidas de forma construtiva [7] a partir de *axiomas* e *regras de inferência*. Os *axiomas* são um subconjunto de pares $\Gamma \vdash A$ pertencentes à relação em causa, as *regras de inferência* permitem calcular novos pares da relação a partir de outros já conhecidos. Todas as relações

de dedução que iremos considerar partilham um axioma e uma regra fundamental, *reflexividade* e *monotonia*, respectivamente. O axioma da reflexividade diz-nos que qualquer fórmula é trivialmente deduzida a partir de um conjunto de fórmulas que a contenha,

$$\Gamma \vdash A \text{ se } A \in \Gamma \text{ (Axioma da Reflexividade).}$$

A regra de inferência da monotonia diz-nos que se uma fórmula A é deduzida de um conjunto Γ, então também é deduzida a partir de qualquer outro conjunto que a contenha. A notação utilizada para a representar é,

$$\frac{\Gamma \vdash A}{\Delta \vdash A} \text{ se } \Gamma \subseteq \Delta \text{ (Regra de Inferência da Monotonia).}$$

Repare-se que o axioma da reflexividade define na verdade uma infinidade de axiomas, sendo *esquema axiomático* um termo mais preciso para o descrever, no entanto é comum algum abuso de linguagem sem perigo de ambiguidade. Da mesma forma, regras são também definidas com base em esquemas axiomáticos. De seguida é apresentado um sistema de dedução proposto por Hilbert, e que iremos designar por *Hilbertiano*.

1.2.1 Sistemas de Dedução de Hilbert

Os sistemas de dedução introduzidos por David Hilbert [33], ditos *Hilbertianos*, são caracterizados por possuírem apenas uma regra própria (que não a da monotonia), a regra *Modus Ponens*. O sistema de dedução de Hilbert para lógica proposicional clássica em seguida descrito é uma adaptação do definido por Kleene em [15].

Definição 1.14 (Sistema de Hilbert para Lógica Proposicional Clássica) *Os axiomas próprios do Sistema de Hilbert para lógica proposicional clássica $\mathcal{H}$ são dados pelos seguintes esquemas axiomáticos,*

$\mathcal{A}_1$. $\vdash_{\mathcal{H}} A \Rightarrow (B \Rightarrow A)$,

$\mathcal{A}_2$. $\vdash_{\mathcal{H}} (A \Rightarrow B) \Rightarrow ((A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \Rightarrow C))$

$\mathcal{A}_3$. $\vdash_{\mathcal{H}} A \Rightarrow (B \Rightarrow A \wedge B)$,

$\mathcal{A}_4$. $\vdash_{\mathcal{H}} A \Rightarrow A \vee B$,

$\mathcal{A}_5$. $\vdash_{\mathcal{H}} A \Rightarrow B \vee A$,

$\mathcal{A}_6$. $\vdash_{\mathcal{H}} A \wedge B \Rightarrow A$,

$\mathcal{A}_7$. $\vdash_{\mathcal{H}} A \wedge B \Rightarrow B$,

$\mathcal{A}_8$. $\vdash_{\mathcal{H}} (A \Rightarrow B) \Rightarrow ((C \Rightarrow B) \Rightarrow (A \vee C \Rightarrow B))$,

$\mathcal{A}_9$. $\vdash_{\mathcal{H}} (A \Rightarrow B) \Rightarrow ((A \Rightarrow \neg B) \Rightarrow \neg A)$,

$\mathcal{A}_{10}$. $\vdash_{\mathcal{H}} (\neg\neg A \Rightarrow A)$.

A única regra de inferência deste sistema, denominada Modus Ponens ($\mathcal{MP}$), é definida por,

$$\frac{\Gamma \vdash_{\mathcal{H}} A \quad \Delta \vdash_{\mathcal{H}} A \Rightarrow B}{\Gamma, \Delta \vdash_{\mathcal{H}} B}\ \mathcal{MP}$$

Tanto a proposição $p \Rightarrow (q \Rightarrow p)$, como $p \vee q \Rightarrow (\neg r \Rightarrow p \vee q)$, são axiomas em $\mathcal{H}$, instâncias do esquema axiomático $\mathcal{A}_1$. A proposição $p \Rightarrow p \vee p$ é também um axioma, instância de ambos, $\mathcal{A}_4$ e $\mathcal{A}_5$. Na tabela 1.3 mostramos que a proposição $p \Rightarrow p$ pode ser deduzida de um conjunto vazio de fórmulas, ou seja, que se tem $\vdash_{\mathcal{H}} p \Rightarrow p$.

$$\dfrac{\dfrac{\overbrace{\vdash p \Rightarrow ((p \Rightarrow p) \Rightarrow p)}^{\mathcal{A}_1} \qquad \dfrac{\overbrace{\vdash p \Rightarrow (p \Rightarrow p)}^{\mathcal{A}_1} \qquad \overbrace{\vdash (p \Rightarrow (p \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))}^{\mathcal{A}_2}}{\vdash (p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p)}\ \mathcal{MP}}{\vdash p \Rightarrow p}\ \mathcal{MP}$$

Tabela 1.3: Árvore de demonstração para $\vdash p \Rightarrow p$, em $\mathcal{H}$.

Definição 1.15 (Teorema, $\vdash A$) *Dado um sistema de dedução (L,$\vdash$), uma fórmula A diz-se um Teorema se for dedutível do vazio, ou seja, se $\vdash A$.*

Como acabámos de ver, a fórmula $p \Rightarrow p$ é um teorema em $\mathcal{H}$. A definição de teorema num sistema de dedução é análoga à de tautologia num sistema de avaliação. A relação entre ambos será esclarecida na secção seguinte. Na tabela 1.4, mostramos que se pode deduzir $\neg r$ a partir das fórmulas $p \wedge q$ e $p \Rightarrow (q \Rightarrow \neg r)$ (a que damos o nome de *hipóteses*).

$$\dfrac{\dfrac{\overbrace{p \wedge q}^{\text{Hip 1}} \vdash p \wedge q \qquad \overbrace{\vdash p \wedge q \Rightarrow q}^{\mathcal{A}_7}}{p \wedge q \vdash q}\ \mathcal{MP} \qquad \dfrac{\dfrac{\overbrace{p \wedge q}^{\text{Hip 1}} \vdash p \wedge q \qquad \overbrace{\vdash p \wedge q \Rightarrow p}^{\mathcal{A}_6}}{p \wedge q \vdash p}\ \mathcal{MP} \qquad \overbrace{p \Rightarrow (q \Rightarrow \neg r)}^{\text{Hip 2}} \vdash p \Rightarrow (q \Rightarrow \neg r)}{p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash q \Rightarrow \neg r}\ \mathcal{MP}}{p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r}\ \mathcal{MP}$$

Tabela 1.4: Árvore de demonstração para $p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r$, em $\mathcal{H}$.

Como foi referido anteriormente, para mostrarmos que uma fórmula A é deduzida a partir de um conjunto de fórmulas Γ, partimos de axiomas (folhas das árvores apresentadas) e através de várias aplicações de regras de inferência deduzimos sucessivamente novos pares na relação $\vdash$, até obtermos o par desejado (raiz da árvore). A este processo dá-se o nome de *demonstração*.

As apresentações aqui utilizadas, em forma de árvore, têm o nome de *árvores de demonstração.* Uma definição mais precisa de árvore de demonstração, requer a introdução de alguns conceitos auxiliares, e uma definição mais concisa de conceitos já descritos.

Definição 1.16 (Sequente, Antecedente, Consequente) *Um sequente é um par* $\Gamma \vdash A$. *Ao primeiro membro do par, o conjunto de fórmulas* Γ, *dá-se o nome de antecedente, e ao segundo, a fórmula* A, *de consequente. Cada fórmula de* Γ *diz-se uma hipótese.*

Note-se que sequentes serão redefinidos mais tarde (definição 1.21) para permitir considerar conjuntos de fórmulas também no consequente.

Portanto, o sequente $s,\ p \Rightarrow q \vdash \neg\neg r$, tem antecedente $s,\ p \Rightarrow q$, e consequente $\neg\neg r$. A fórmula s é uma das duas hipóteses consideradas. Conjuntos de sequentes podem ser definidos a partir de *esquemas de sequentes*, ou seja, formas gerais. Quando interpretado como *esquema*, $\Delta, A \Rightarrow B \vdash C$, representa todos os sequentes que possuem uma fórmula do tipo $A \Rightarrow B$ como uma das hipóteses, onde A e B são fórmulas arbitrárias. São instâncias desse esquema $p \Rightarrow q \vdash r$, assim como $s \vee r, p \Rightarrow s \wedge r, q \vdash r$. Esquemas são também denominados sequentes, desde que não haja perigo de ambiguidade.

Definição 1.17 (Regra de Inferência, Premissa, Conclusão) *Regras de inferência são da forma* $\frac{\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n}{\Gamma_0 \vdash \Delta_0}$, *onde cada par* $\Gamma_i \vdash A_i$ *é um esquema de sequentes. Os sequentes superior da regra denominam-se premissas, e o sequente inferior denomina-se conclusão.*

A regra de inferência do sistema $\mathcal{H}$, *Modus Ponens*, $\frac{\Gamma \vdash A \quad \Delta \vdash A \Rightarrow B}{\Gamma, \Delta \vdash B}$, tem como premissas $\Gamma \vdash A$ e $\Delta \vdash A \Rightarrow B$, e como conclusão $\Gamma, \Delta \vdash B$. Árvores de inferência e árvores de demonstração, obtidas por encadeamento de regras de inferência, são definidas em seguida.

Definição 1.18 (Árvore de Inferência, Árvore de Demonstração) *Árvores de inferência, são árvores cujos nós contêm (são etiquetados por) sequentes, e que satisfazem a seguinte definição indutiva:*

- *sequentes, são árvores de inferência (constituídas por um só nó);*
- *se* $t_1, \ldots, t_n$ *são árvores de inferência com raízes* $\Gamma_i \vdash \Delta_i$, *e* $\frac{\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n}{\Gamma_0 \vdash \Delta_0}$ *é uma instância de uma regra de inferência, então também* $\frac{t_1, \ldots, t_n}{\Gamma_0 \vdash \Delta_0}$ *é uma árvore de inferência.*

Árvores de demonstração, são árvores de inferência em que todas as folhas são axiomas.

As árvores apresentadas nas tabelas 1.3 e 1.4 são ambas árvores de demonstração. Tomemos a subárvore da segunda, apresentada na tabela 1.5. Esta é apenas uma árvore de inferência, mas não de demonstração, pois nem todas as folhas são axiomas.

1.2.2 Sistemas de Dedução Natural

As demonstrações em sistemas do tipo *Hilbertiano* são normalmente bastante complexas. A simplicidade a nível do conjunto de regras de inferência que os definem faz com que as escolha dos

$$\dfrac{\dfrac{\overbrace{p \wedge q}^{\text{Hip 1}} \vdash p \wedge q \qquad \overbrace{\vdash p \wedge q \Rightarrow q}^{\mathcal{A}_7}}{p \wedge q \vdash q}\ \mathcal{MP} \qquad \dfrac{p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash q \Rightarrow \neg r}{p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r}\ \mathcal{MP}}{}$$

Tabela 1.5: Árvore de inferência (mas não de demonstração), em $\mathcal{H}$.

axiomas a utilizar em cada situação não seja nada óbvia, nem fácil de calcular. Para simplificar o processo de demonstração, Gerard Gentzen introduziu os sistemas de dedução natural, que têm por objectivo permitir apresentações mais próximas das demonstrações informais, normalmente efectuadas [33]. Estes sistemas, ao contrário dos anteriores, possuem apenas o axioma fundamental (da reflexividade) e várias regras de inferência. Têm ainda a característica particular de definirem duas regras de inferência para cada conectiva e quantificador, uma para a sua introdução, e outra para a sua remoção.

Definição 1.19 (Sistema de Dedução Natural para Lógica Proposicional Clássica) *As regras de inferência próprias do Sistema de Dedução Natural $\mathcal{N}$, para lógica proposicional clássica, são:*

$$\dfrac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}\ \wedge_i \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge_{e_1} \quad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge_{e_2}$$

$$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_{i_1} \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_{i_2} \qquad \dfrac{\Gamma \vdash A \vee B \quad \Delta, A \vdash C \quad \Theta, B \vdash C}{\Gamma, \Delta, \Theta \vdash C}\ \vee_e$$

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow_i \qquad \dfrac{\Gamma \vdash A \Rightarrow B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}\ \Rightarrow e$$

$$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i \qquad \dfrac{\Gamma \vdash A \quad \Delta \vdash \neg A}{\Gamma, \Delta \vdash \bot}\ \neg_e$$

$$\ldots \bot_i \qquad \dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_e$$

A demonstração do teorema $p \Rightarrow p$ no sistema $\mathcal{N}$ é trivial, e é apresentada na tabela 1.6. A tabela 1.7 contém a demonstração de $p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r$.

O sistema $\mathcal{N}$, para lógica de predicados de primeira ordem é de seguida apresentado.

Definição 1.20 (Sistema de Dedução Natural para Lógica de Predicados) *As regras de*

$$\dfrac{p \vdash p}{\vdash p \Rightarrow p}\ \Rightarrow_i$$

Tabela 1.6: Árvore de demonstração para $\vdash p \Rightarrow p$, em $\mathcal{N}$.

$$\dfrac{\dfrac{\dfrac{p \wedge q \vdash p \wedge q}{p \wedge q \vdash p \wedge q}\ \wedge_e}{p \wedge q \vdash q} \qquad \dfrac{\dfrac{p \wedge q \vdash p \wedge q}{p \wedge q \vdash p}\ \wedge_e \qquad p \Rightarrow (q \Rightarrow \neg r) \vdash p \Rightarrow (q \Rightarrow \neg r)}{p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash q \Rightarrow \neg r}\ \Rightarrow_e}{p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r}\ \Rightarrow_e$$

Tabela 1.7: Árvore de demonstração para $p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r$, em $\mathcal{N}$.

inferência próprias do Sistema de Dedução Natural $\mathcal{N}$, para lógica de predicados (de primeira ordem), são todas as apresentadas na definição 1.19, e ainda:

$$\dfrac{\Gamma \vdash A(y)}{\Gamma \vdash \forall x.\, A}\ \forall_i \qquad \dfrac{\Gamma \vdash \forall x.\, A}{\Gamma \vdash A(t)}\ \forall_e$$

$$\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x.\, A}\ \exists_i \qquad \dfrac{\Gamma \vdash \exists x.\, A \quad \Delta, A(y) \vdash B}{\Gamma, \Delta \vdash B}\ \exists_e$$

Em $\forall_i$, a variável y não ocorre livre em Γ ou $A(x)$. Em $\exists_e$, a variável y não ocorre livre em Γ, $A(x)$ ou B, e o termo t é qualquer.

Nesta definição, as fórmulas do tipo $A(s)$ são obtidas tomando a fórmula A quantificada, presente na regra em causa, e substituindo a variável ligada pelo termo s. Vejamos a demonstração de $\forall x.\, P(x) \vdash \neg\exists x.\, \neg P(x)$ no sistema $\mathcal{N}$ para lógica de predicados clássica, apresentada na tabela 1.8. A leitura da demonstração é bastante simples,

> "Tomemos como hipótese que se tem $P(x)$ para todo o x, daí podemos concluir $P(y)$ para um determinado y arbitrário ($\forall_e$). Suponhamos agora que temos também $\neg P(y)$, de ambas as hipóteses surge uma contradição ($\neg_e$). Como a variável y considerada é arbitrária, então basta existir um qualquer x que satisfaça $P(x)$ para obter a contradição ($\exists_e$). Da contradição conclui-se que, se para todo o x se tem $P(x)$, então podemos deduzir que não existe um x que verifique $\neg P(x)$.".

Este tipo de sistemas pode ser facilmente adaptado para utilização em sistemas de demonstração interactivos (*proof assistents*). O sistema HOL, é um exemplo, de um demonstrador

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, P(x) \vdash \forall x.\, P(x)}{\forall x.\, P(x) \vdash P(y)}\,\forall_e \qquad \neg P(y) \vdash \neg P(y)}{\forall x.\, P(x), \neg P(y) \;\vdash\; \bot}\,\neg_e \qquad \exists x.\, \neg P(x) \vdash \exists x.\, \neg P(x)}{\forall x.\, P(x), \exists x.\, \neg P(x) \;\vdash\; \bot}\,\exists_e}{\forall x.\, P(x) \;\vdash\; \neg\exists x.\, \neg P(x)}\,\neg_i$$

Tabela 1.8: Árvore de demonstração para $\forall x.\, P(x) \vdash \neg\exists x.\, \neg P(x)$, em $\mathcal{N}$.

interactivo baseado em dedução natural [29]. No entanto, as demonstrações em dedução natural não são facilmente mecanizáveis, a selecção de regras para a introdução e eliminação de conectivas e hipóteses é normalmente conseguida de forma intuitiva, intuição essa difícil de traduzir para algoritmos.

1.2.3 Sistemas de Dedução de Gentzen

Sistemas de dedução de Gentzen, também conhecidos por Cálculo de Sequentes, são semelhantes aos de Dedução Natural, no sentido em que também não possuem axiomas para além do fundamental, e se baseiam em pares de regras para cada conectiva. Desta vez, os pares de regras não servem, no entanto, para introduzir e eliminar conectivas de um consequente, mas sim apenas para as introduzir em ambos, antecedente e consequente. Para apresentarmos o sistema de dedução de Gentzen para lógica clássica, necessitamos de redefinir *sequentes*, por forma a que permitam várias fórmulas no consequente, e não apenas uma, como definido anteriormente.

Definição 1.21 (Sequente, Antecedente, Consequente) *Um sequente é um par* $\Gamma \vdash \Delta$. *Ao primeiro membro do par, o conjunto de fórmulas* Γ*, dá-se o nome de antecedente, e ao segundo, o conjunto de fórmulas* Δ*, consequente.*

Intuitivamente, um sequente da forma $\Gamma \vdash \Delta$ pode ser interpretado como sendo possível deduzir, a partir da conjunção das fórmulas em Γ, a disjunção das fórmulas de Δ. Desta alteração na definição de sequente, resulta a necessidade de adaptação das regras fundamentais, definidas na secção 1.2, em particular passamos a ter duas regras de monotonia, a já existente para antecedentes, e uma nova para consequentes. Passamos a ter então, como axioma da reflexividade,

$$\Gamma \vdash \Delta \text{ se } \Gamma \cap \Delta \neq \emptyset \quad \text{(Axioma da Reflexividade).}$$

e como regras de monotonia à esquerda e à direita,

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Theta} \text{ se } \Delta \subseteq \Theta, \quad \frac{\Gamma \vdash \Delta}{\Theta \vdash \Delta} \text{ se } \Gamma \subseteq \Theta \quad \text{(Regras da Monotonia).}$$

Definição 1.22 (Sistema de Gentzen para Lógica Proposicional Clássica) *As regras de inferência próprias do Sistema de Gentzen $\mathcal{G}$, para lógica proposicional clássica, são:*

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_l \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge_r$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee_l \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_r$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow_l \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow_r$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg_l \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, \vdash \neg A, \Delta} \neg_r$$

Este sistema de Gentzen em particular apresenta propriedades adicionais bastante interessantes. Em cada regra é criada uma nova fórmula na conclusão (*fórmula principal*), a partir de fórmulas das premissas (*fórmulas laterais*), por aplicação de uma conectiva. As regras definem uma relação de *antecessor* nas fórmulas da demonstração. No caso da fórmula principal, o seu *antecessor directo* são as fórmulas laterais, no caso das restantes fórmulas (*fórmulas secundárias*) são as próprias fórmulas que estão também presentes nas premissas. Os antecessores de uma fórmula são dados pelo fecho transitivo da relação de antecessor directo. Para além disso, por construção, os antecessores de um fórmula são subfórmulas da mesma. Estas propriedades são conhecidas por *propriedade de antecessor e subfórmula*, e são bastante úteis para a mecanização de demonstrações [34, 5].

Consideremos novamente o sequente $p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r$, a sua demonstração no sistema $\mathcal{G}$ (tabela 1.9) é obtida, partindo da raiz até às folhas, escolhendo uma fórmula (*fórmula principal*) que será decomposta (nas suas *fórmulas laterais*). Cada ramo termina quando a intersecção entre antecedente e consequente da premissa calculada for diferente do vazio. O processo de demonstração termina quando todos os ramos terminarem.

As regras de inferência para lógica de predicados são um pouco mais complexas. Neste caso não se irá obter uma subfórmula por remoção do quantificador, de qualquer modo o resultado verifica a propriedade semelhante, a fórmulas laterais resultantes são *instâncias* (conceito definido mais à frente, na secção 1.5) de subfórmulas da fórmula principal.

Definição 1.23 (Sistema de Gentzen para Lógica de Predicados Clássica) *As regras de inferência próprias do Sistema de Gentzen $\mathcal{G}$, para lógica de predicados (de primeira ordem), são todas as apresentadas na definição 1.22, e ainda:*

$$\dfrac{\dfrac{r,\ p,\ \mathbf{q} \vdash \mathbf{q} \qquad \dfrac{\mathbf{r},\ p,\ q \vdash \mathbf{r}}{r,\ p,\ q, \neg r \vdash}\ \neg_l}{r,\ p,\ q,\ q \Rightarrow \neg r \vdash}\ \Rightarrow_l \qquad \dfrac{\dfrac{\dfrac{r,\ p,\ q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash}{p,\ q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r}\ \neg_r}{p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r}\ \wedge_l}{r,\ \mathbf{p},\ q \vdash \mathbf{p}}\ \Rightarrow_l$$

Tabela 1.9: Árvore de demonstração para $p \wedge q,\ p \Rightarrow (q \Rightarrow \neg r) \vdash \neg r$ em $\mathcal{G}$.

$$\frac{\Gamma, A(t) \vdash \Delta}{\Gamma, \forall x.\, A(x) \vdash \Delta}\ \forall_l \qquad \frac{\Gamma \vdash A(y), \Delta}{\Gamma \vdash \forall x.\, A(x), \Delta}\ \forall_r$$

$$\frac{\Gamma, A(y) \vdash \Delta}{\Gamma, \exists x.\, A(x) \vdash \Delta}\ \exists_l \qquad \frac{\Gamma \vdash A(t), \Delta}{\Gamma \vdash \exists x.\, A(x), \Delta}\ \exists_r$$

onde, em $\forall_r$ *e* $\exists_l$*, a variável* y *não ocorre livre em* Γ*,* Δ*, ou* $A(x)$*.*

Na tabela 1.10 é apresentada a demonstração de $\forall x.\, P(x) \vdash \neg\exists x.\, \neg P(x)$ no sistema $\mathcal{G}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{P(y)} \ \vdash\ \mathbf{P(y)}}{\forall x.\, P(x) \ \vdash\ P(y)}\ \forall_l}{\forall x.\, P(x),\ \neg P(y) \ \vdash}\ \neg_l}{\forall x.\, P(x),\ \exists x.\, \neg P(x) \ \vdash}\ \exists_l}{\forall x.\, P(x) \ \vdash\ \neg\exists x.\, \neg P(x)}\ \neg_r$$

Tabela 1.10: Árvore de demonstração para $\forall x.\, P(x) \vdash \neg\exists x.\, \neg P(x)$, em $\mathcal{G}$.

Como se pode observar, grande parte da demonstração consiste apenas em 'decompor fórmulas'. Convém notar, no entanto, que ainda nada foi referido acerca de como mecanizar a escolha dos termos utilizados, quando removidos os quantificadores.

1.3 Semântica e Demonstrações

Referimos até agora duas formas de criar sistemas lógicos, a primeira semântica ($\models$), e a segunda dedutiva ($\vdash$). Tanto a consequência semântica como a dedutiva são casos particulares de relações de consequência.

Definição 1.24 (Relação de Consequência) *Uma relação $\Vdash$ diz-se relação de consequência se verificar as seguintes propriedades,*

reflexividade: *se $A \in \Gamma$ então $\Gamma \Vdash A$,*

monotonia: *se $\Gamma \subseteq \Delta$ e $\Gamma \Vdash A$ então $\Delta \Vdash A$,*

transitividade (corte): *se $\Gamma \Vdash A$ e $\Delta, A \Vdash B$ então $\Gamma, \Delta \Vdash B$.*

No caso de relações de consequência semântica estas propriedades resultam directamente da definição de $\models$ e das avaliações v_a. No caso das relações de dedução, as duas primeiras são equivalentes aos respectivos axiomas e regras fundamentais apresentadas, a última porém está dependente da construção de cada sistema, isto é, das suas regras e axiomas. Pode demonstrar-se que qualquer um dos sistemas apresentados satisfaz a propriedade do corte [8].

Sistemas lógicos podem ser definidos à custa de uma linguagem lógica e uma qualquer relação de consequência definida sobre a mesma. Dados dois sistemas lógicos sobre uma mesma linguagem podemos querer compará-los. De um modo geral essa comparação é efectuada entre uma lógica semântica e uma lógica dedutiva. Definimos então os conceitos de idoneidade e completude entre dois sistemas.

Definição 1.25 (Idoneidade, Completude) *Sejam $\models$ uma relação de consequência semântica e $\vdash$ uma relação de dedução. Diz-se que $\vdash$ é idóneo relativamente a $\models$ se para todo o Γ, A tal que $\Gamma \vdash A$, também $\Gamma \models A$. Diz-se que $\vdash$ é completo relativamente a $\models$ se para todo Γ, A tal que $\Gamma \models A$, também $\Gamma \vdash A$.*

Todos os sistemas de dedução para lógica proposicional e lógica de primeira ordem clássicas, apresentados são idóneos e completos. Em particular, para todo o teorema existe uma demonstração, e fórmulas deduzidas do vazio são teoremas.

Definição 1.26 (Decidibilidade, Semi-Decidibilidade) *Um sistema lógico $(L, \models, \vdash)$ completo e idóneo diz-se decidível se existir um algoritmo que, para toda a fórmula A consiga determinar num número finito de passos, se $\Gamma \vdash A$ ou $\Gamma \nvdash A$. Um sistema diz-se semi-decidível se for decidível na restrição do sistema às fórmulas A tais que $\Gamma \models A$.*

1.4 Teorias

Definição 1.27 (Teoria) *Uma teoria $\mathcal{T}$, definida sobre um sistema lógico $S = (L, \Vdash_S)$ através de um conjunto de fórmulas Γ, é um sistema lógico $S + \mathcal{T} = (L, \Vdash_{S+\mathcal{T}})$, onde a relação de consequência é definida por $\Delta \Vdash_{S+\mathcal{T}} A$ se e só se $\Gamma, \Delta \Vdash_S A$.*

Por outras palavras, uma teoria é uma extensão de um sistema lógico. Da definição tem-se que, se o sistema em causa for semântico, a extensão é construída a partir da restrição das avaliações v_a a atribuições a que tornem as fórmulas em Γ verdadeiras. Ou seja, a relação $\models_{S+\mathcal{T}}$ é a menor extensão a $\models_S$ onde as fórmulas de Γ são tautologias. Se se tratar de um sistema de dedução, as fórmulas de Γ definem axiomas $\{ \vdash_{S+\mathcal{T}} A \mid A \in \Gamma\}$ a partir dos quais a nova relação de dedução é gerada. É comum a referência às fórmulas de Γ como sendo os axiomas de $\mathcal{T}$.

A formulação de determinados domínios como teorias, permite demonstrar propriedades dos mesmos, de forma rigorosa, utilizando um sistema lógico como base. Tomemos como exemplo a estrutura algébrica *Grupo*. Dizem-se *Grupos* as estruturas matemáticas definidas por um conjunto de elementos, munido de uma operação (interna) associativa, para a qual existe um elemento neutro, e onde todo o elemento possui inverso [35].

Definição 1.28 *Um grupo $\mathcal{G} = (G, \oplus)$ consiste num conjunto G e numa operação interna binária $\oplus : G \times G \to G$, que verifica as seguintes propriedades,*

$\mathcal{G}_1$. *(Associatividade) Para todo o $a, b, c \in G$,*

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c,$$

$\mathcal{G}_2$. *(Existência de elemento neutro) Existe um $e \in G$ tal que para todo o $a \in G$,*

$$e \oplus a = a \oplus e = a,$$

$\mathcal{G}_3$. *(Existência de inversos) para todo o $a \in G$ existe um $a' \in G$ tal que*

$$a \oplus a' = a' \oplus a = e.$$

O par $(\mathbb{Z}, +)$, formado por números inteiros e adição, define um grupo, com elemento neutro 0, e onde o inverso de cada $a \in \mathbb{Z}$ é $-a$. Outro exemplo é o par $(\mathbb{R}\backslash\{0\}, \times)$, de números reais excepto o zero, e multiplicação. Neste caso tem-se 1 como elemento neutro e $\frac{1}{a}$ como o inverso de cada $a \in \mathbb{R}\backslash\{0\}$. Para criar uma teoria de grupos, é necessário axiomatizar o domínio que acabámos de descrever, ou seja, traduzir as propriedades apresentadas para fórmulas no nosso sistema lógico. Em lógica de primeira ordem clássica podemos escrever,

$\mathcal{A}^1_{\mathcal{G}}$. $\forall x. \forall y. \forall z.\ x \oplus (y \oplus z) = (x \oplus y) \oplus z,$

$\mathcal{A}^2_{\mathcal{G}}$. $\exists e. \forall x.\ e \oplus x = x \ \wedge\ x \oplus e = x,$

$\mathcal{A}^3_{\mathcal{G}}$. $\forall x. \exists x'.\ x \oplus x' = e \ \wedge\ x' \oplus x = e.$

Nesta formulação, o símbolo $\oplus$ é um operador (função) binário, escrito em notação *infix*, e o simbolo $=$ denota um predicado, também binário, escrito em notação *infix*. Apesar de muito semelhante à definição de $\mathcal{G}$ apresentada anteriormente, esta axiomática não é suficiente para definir uma teoria de grupos sobre lógica de primeira ordem clássica. A apresentação inicial assume que é conhecida a definição de igualdade e, no nosso sistema, nada é dito acerca de $=$, o predicado correspondente. Antes de mais, a igualdade é uma relação de equivalência, ou seja, é reflexiva, simétrica, e transitiva [35]. Essas propriedades são captadas pela seguinte formulação, em lógica de primeira ordem,

$\mathcal{A}^1_{\mathcal{E}q}$. $\forall x.\ x = x$,

$\mathcal{A}^2_{\mathcal{E}q}$. $\forall x. \forall y.\ x = y\ \Rightarrow\ y = x$,

$\mathcal{A}^3_{\mathcal{E}q}$. $\forall x. \forall y. \forall z.\ x = y\ \wedge\ y = z\ \Rightarrow\ x = z$.

Por fim, falta ainda dizer que $(\mathcal{G}, \oplus)$, é uma congruência sobre $=$, ou seja, que a operação $\oplus$ preserva a igualdade entre elementos de $\mathcal{G}$ [35].

$\mathcal{A}^4_{\mathcal{G}}$. $\forall x. \forall y. \forall z.\ x = y\ \Rightarrow z \oplus x = z \oplus y$,

$\mathcal{A}^5_{\mathcal{G}}$. $\forall x. \forall y. \forall z.\ x = y\ \Rightarrow x \oplus z = y \oplus z$.

Estamos então em condições de definir uma teoria de equivalência $\mathcal{E}q$ sobre lógica de primeira ordem clássica (ou seja um sistema $FOL + \mathcal{E}q$). Sobre a teoria $\mathcal{E}q$ podemos definir uma teoria de grupos $\mathcal{G}$ (ou seja um sistema $FOL + \mathcal{E}q + \mathcal{G}$). Nesta teoria podemos demonstar resultados sobre o domínio em causa, no entanto, é importante notar que, caso desejemos efectuar demonstrações que envolvam outras funções que não a operação $\oplus$, é necessário explicitar que estas também preservam a igualdade, através da introdução de hipóteses extra semelhantes aos axiomas $\mathcal{A}^4_{\mathcal{G}}$ e $\mathcal{A}^5_{\mathcal{G}}$. Por exemplo, para uma função unária g ter-se-ía,

$$\forall x. \forall y.\ x = y\ \Rightarrow g(x) = g(y).$$

Da mesma forma, demonstrações com predicados extra requerem que se explicite que é indiferente satisfazer um elemento x ou um elemento y que seja igual ao primeiro. Para um predicado unário teríamos,

$$\forall x. \forall y.\ x = y\ \wedge P(x) \Rightarrow P(y).$$

O ideal seria conseguir expressar estas propriedades, de uma só vez, para funções e predicados arbitrários, no entanto tal não é possível em lógica de primeira ordem. A quantificação é efectuada apenas sobre variáveis, e não sobre funções ou predicados. Outro problema da axiomatização apresentada é a elevada complexidade dos novos axiomas introduzidos em $\mathcal{G}$, devido à sua dependência dos axiomas de $\mathcal{E}q$. Porque igualdade se trata de uma propriedade especial, esta é normalmente tratada de forma especial. Em vez de ser definida a nível do sistema lógico, através de um predicado normal, a igualdade entre termos é definida ao nível dos próprios termos, termos iguais entre si são considerados o mesmo. Na prática, esta abordagem requer recurso a técnicas como *paramodulação*, para substituir termos por outros iguais, ou *demodulação*, para reduzir a representação de termos a formas canónicas. Tais técnicas não serão abordadas neste trabalho, pelo que, teorias equacionais (com igualdade) irão ser tratadas apenas ao nível lógico.

Podemos, no entanto, tentar construir uma axiomática menos complexa, (i.e., que simplifique o processo de demonstração), que não seja baseada no predicado de igualdade. Para o efeito introduzimos o predicado ternário P, cuja aplicação $P(x, y, z)$ deverá significar x *operado com* y *através de* $\oplus$ *resulta em* z. As propriedades de grupos, reformuladas com base neste predicado, são [39],

$\mathcal{A}^1_{\mathcal{G}}$. $\forall x.\, P(e,x,x) \wedge P(x,e,x)$,

$\mathcal{A}^2_{\mathcal{G}}$. $\forall x.\, \exists x'.\, P(x,x',e) \wedge P(x',x,e)$,

$\mathcal{A}^{3a}_{\mathcal{G}}$. $\forall x.\, \forall y.\, \forall z.\, \forall u.\, \forall v.\, \forall w.\, (P(x,y,u) \wedge P(y,z,v)) \Rightarrow (P(u,z,w) \Rightarrow P(x,v,w))$,

$\mathcal{A}^{3b}_{\mathcal{G}}$. $\forall x.\, \forall y.\, \forall z.\, \forall u.\, \forall v.\, \forall w.\, (P(x,y,u) \wedge P(y,z,v)) \Rightarrow (P(x,v,w) \Rightarrow P(u,z,w))$.

A primeira fórmula define um elemento neutro e, a segunda estabelece a existência de elementos inversos, e as duas últimas a propriedade associativa da operação do grupo. Se as duas primeiras são triviais, já a axiomática definida para a propriedade associativa poderá requerer alguma explicação. O predicado P apenas permite considerar operação de elementos dois a dois, portanto necessitamos de elementos auxiliares para denotar o resultado de cada uma das operações binárias que aparecem nas fórmulas $x \oplus (y \oplus z)$ e $(x \oplus y) \oplus z$. Definimos então u, v, e w, como sendo,

$$\underbrace{x \oplus \overbrace{(y \oplus z)}^{v}}_{w} = \underbrace{\overbrace{(x \oplus y)}^{u} \oplus z}_{w}.$$

Nas fórmulas $\mathcal{G}_{3a}$ e $\mathcal{G}_{3b}$ os lados esquerdos da implicação (as conjunções) definem u e v. O lado direito de $\mathcal{G}_{3b}$ diz que se de $x \oplus v$ resultar w então também de $u \oplus z$ resulta w, e o lado direito de $\mathcal{G}_{3b}$ o recíproco[1].

Porque ter de utilizar elementos auxiliares para expressar a operação de vários elementos não é muito conveniente, podemos agora, com base no predicado P, definir a operação do grupo,

$\mathcal{A}^4_{\mathcal{G}}$. $\forall x.\, \forall y.\ P(x,y,x \oplus y)$.

Necessitamos ainda de igualdade para garantir que o resultado da operação de dois elementos é único, e que elementos iguais dão o mesmo resultado. As duas fórmulas seguintes são suficientes,

$\mathcal{A}^5_{\mathcal{G}}$. $\forall x.\, \forall y.\, \forall u.\, \forall v.\ P(x,y,u) \wedge P(x,y,v) \Rightarrow u = v$,

$\mathcal{A}^6_{\mathcal{G}}$. $\forall x.\, \forall y.\, \forall u.\, \forall v.\ P(x,y,u) \wedge u = v \Rightarrow P(x,y,v)$.

Podemos, no entanto, se assim desejarmos, introduzir propriedades adicionais da igualdade para simplificar algumas demonstrações. Finalmente, comparemos esta nova axiomática com a primeira que definimos. O papel do predicado P é permitir a definição das propriedades de $\mathcal{G}$ directamente, sem recurso a uma teoria de equivalência, o que irá permitir obter demonstrações mais simples.

Vejamos agora como efectuar demonstrações com base numa teoria. Uma questão interessante diz respeito às propriedades da axiomática definida. Por exemplo, será que existe redundância na formulação da teoria? Será que algumas das propriedades pode ser relaxada sem que a teoria sofra alterações? Como iremos ver, a resposta é afirmativa. A propriedade $\mathcal{G}_1$, traduzida pelo axioma $\mathcal{A}^1_{\mathcal{G}}$, estabelece a existência de um elemento e, neutro à direita e à esquerda. É possível demonstrar que basta considerar e como sendo neutro de um dos lados, a neutralidade do lado

[1] Se introduzirmos no sistema lógico a conectiva de equivalência ($\Leftrightarrow$), os axiomas da associatividade, $\mathcal{A}^{3a}_{\mathcal{G}}$ e $\mathcal{A}^{3b}_{\mathcal{G}}$, podem ser fundidos na seguinte fórmula, $\forall x.\, \forall y.\, \forall z.\, \forall u.\, \forall v.\, \forall w.\, (P(x,y,u) \wedge P(y,z,v)) \Rightarrow (P(u,z,w) \Leftrightarrow P(x,v,w))$.

oposto é consequência dos restantes axiomas. Tomemos neutralidade à esquera como hipótese, uma possível demonstração informal para a neutralidade à direita é,

$$\begin{aligned} & e \oplus a = a \\ \equiv \quad & (a \oplus a') \oplus a = a \\ \equiv \quad & a \oplus (a' \oplus a) = a \\ \equiv \quad & a \oplus e = a \ . \end{aligned}$$

Na tabela 1.4 apresentamos a árvore de demonstração para $\vdash_{\mathcal{G}} \forall x.\, P(x, e, x)$, após substituição de $A^1_{\mathcal{G}}$ pelo axioma $\mathcal{A}^{1'}_{\mathcal{G}}$, definido apenas pela fórmula $\forall x.\, P(e, x, x)$. Os passos iniciais correspondem à introdução explícita dos axiomas de $\mathcal{G}$, procedida da eliminação dos quantificadores, que na prática pode ser interpretado como uma selecção dos elementos de G sobre os quais queremos aplicar as propriedades em causa. No final, são aplicadas regras de inferência sobre essas mesmas propriedades, obtendo assim axiomas (da reflexividade) nas folhas, completando a demonstração.

Para terminar, vejamos como seria efectuada a demonstração da identidade à direita, se em vez de $P(a, e, a)$, partíssemos da forma mais habitual, $a \oplus e = a$. A ideia geral da demonstração seria a redução ao primeiro caso, demonstramos $P(a, e, a)$, o axioma $\mathcal{A}^1_{\mathcal{G}}$ garante-nos $P(a, e, a \oplus e)$, e obtemos a fórmula desejada a partir destes e do $\mathcal{A}^5_{\mathcal{G}}$ na forma $P(a, e, a \oplus e) \wedge P(a, e, a) \Rightarrow a \oplus e = a$. Parte da árvore de demonstração encontra-se na tabela 1.12, a demonstração completa seria obtida por concatenação desta com a anterior.

1.5 Substituição e Unificação

Para terminar este capítulo, apresentamos os fundamentos de *substituições*, *unificadores* e alguns conceitos relacionados. Estes serão necessários para a mecanização do processo de demonstração.

Definição 1.29 (Substituição) *Uma substituição σ consiste num mapeamento de um conjunto de variáveis V num conjunto de termos $T \supseteq V$, sendo a sua aplicação a $x \in V$ denotada por $x\sigma$. É habitual representar substituições por conjuntos de associações 'variável' $\mapsto$ 'termo', $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ mapeia x_i em t_i, para $i \in \{1, \ldots, n\}$, e x em si próprio, para todo $x \in V \backslash \{x_1, \ldots, x_n\}$. A substituição identidade, ou seja $\emptyset$, irá ser denotada por ε.*

A aplicação de substituições pode ser estendida para termos e fórmulas e consiste no mapeamento de todas as ocorrências de variáveis livres no termo ou fórmula em causa. Como exemplo, para $\sigma = \{x \mapsto y, y \mapsto w, z \mapsto g(y)\}$, $t = f(x, y, g(z))$ e $A = \forall z.\, P(x, z)$, temos $t\sigma = f(y, w, g(g(y)))$ e $A\sigma = \forall z.\, P(y, z)$. No restante texto desta subsecção iremos considerar a aplicação de substituições apenas a termos, sem perda de generalidade para simplificar a exposição.

Definição 1.30 (Domínio e Codomínio) *O domínio de uma substituição σ é definido por $Dom(\sigma) = \{x \in V \mid x\sigma \neq x\}$ e o seu codomínio por $Cod(\sigma) = \{x\sigma \mid x \in Dom(\sigma)\}$.*

A substituição $\sigma = \{x \mapsto y, y \mapsto w, z \mapsto g(y)\}$, considerada no exemplo anterior, tem domínio $Dom(\sigma) = \{x, y, z\}$ e codomínio $Cod(\sigma) = \{y, w, g(y)\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{P}(\mathbf{a},\mathbf{b},\mathbf{e}) \wedge \mathbf{P}(\mathbf{b},\mathbf{a},\mathbf{e}),\ P(e,a,a) \vdash_{\mathcal{G}} \mathbf{P}(\mathbf{a},\mathbf{b},\mathbf{e}) \wedge \mathbf{P}(\mathbf{b},\mathbf{a},\mathbf{e}),\ P(a,e,a)}\ \dfrac{P(e,a,a),\ \mathbf{P}(\mathbf{a},\mathbf{e},\mathbf{a}),\ \Delta_1 \vdash_{\mathcal{G}} \mathbf{P}(\mathbf{a},\mathbf{e},\mathbf{a}) \qquad \Delta_2,\ \mathbf{P}(\mathbf{e},\mathbf{a},\mathbf{a}) \vdash_{\mathcal{G}} \mathbf{P}(\mathbf{e},\mathbf{a},\mathbf{a}),\ P(a,e,a)}{P(e,a,a) \Leftrightarrow P(a,e,a),\ P(a,b,e) \wedge P(b,a,e),\ P(e,a,a) \vdash_{\mathcal{G}} P(a,e,a)}\ \Leftrightarrow_l}{(P(a,b,e) \wedge P(b,a,e)) \Rightarrow (P(e,a,a) \Leftrightarrow P(a,e,a)),\ P(a,b,e) \wedge P(b,a,e),\ P(e,a,a) \vdash_{\mathcal{G}} P(a,e,a)}\ \Rightarrow_l}{\forall x.\forall y.\forall z.\forall u.\forall v.\forall w.\,(P(x,y,u) \wedge P(y,z,v)) \Rightarrow (P(u,z,w) \Leftrightarrow P(x,v,w)),\ P(a,b,e) \wedge P(b,a,e),\ P(e,a,a) \vdash_{\mathcal{G}} P(a,e,a)}\ (\forall_l)^*}{P(a,b,e) \wedge P(b,a,e),\ P(e,a,a) \vdash_{\mathcal{G}} P(a,e,a)}\ \mathcal{A}^3_{\mathcal{G}}}{\exists x'.\,P(a,x',e) \wedge P(x',a,e),\ P(e,a,a) \vdash_{\mathcal{G}} P(a,e,a)}\ \exists_l}{\forall x.\,\exists x'.\,P(x,x',e) \wedge P(x',x,e),\ P(e,a,a) \vdash_{\mathcal{G}} P(a,e,a)}\ \forall_l}{P(e,a,a) \vdash_{\mathcal{G}} P(a,e,a)}\ \mathcal{A}^2_{\mathcal{G}}}{\forall x.\,P(e,x,x) \vdash_{\mathcal{G}} P(a,e,a)}\ \forall_l}{\vdash_{\mathcal{G}} P(a,e,a)}\ \mathcal{A}^1_{\mathcal{G}}}{\vdash_{\mathcal{G}} \forall x.\,P(x,e,x)}\ \forall_r$$

Tabela 1.11: Árvore de demonstração da fórmula da identidade à direita, em $FOL + \mathcal{G}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{P(a,e,a\oplus e)} \vdash_{\mathcal{G}} \mathbf{P(a,e,a\oplus e)},\ a\oplus e = a}{\forall x.\, P(x,e,x\oplus e) \vdash_{\mathcal{G}} P(a,e,a\oplus e),\ a\oplus e = a}\,\forall_l}{\vdash_{\mathcal{G}} P(a,e,a\oplus e),\ a\oplus e = a}\,\mathcal{A}^1_{\mathcal{G}} \qquad \dfrac{\vdots}{\vdash_{\mathcal{G}} P(a,e,a),\ a\oplus e = a}}{\vdash_{\mathcal{G}} P(a,e,a\oplus e) \wedge P(a,e,a),\ a\oplus e = a}\,\wedge_r \qquad \mathbf{a\oplus e = a} \vdash_{\mathcal{G}} \mathbf{a\oplus e = a}}{P(a,e,a\oplus e) \wedge P(a,e,a) \Rightarrow a\oplus e = a,\ \vdash_{\mathcal{G}}\ a\oplus e = a}\,\wedge_l}{\forall x.\forall y.\forall u.\forall v.\ P(x,y,u) \wedge P(x,y,v) \Rightarrow u = v\ \vdash_{\mathcal{G}}\ a\oplus e = a}\,\forall_l}{\vdash_{\mathcal{G}} a\oplus e = a}\,\mathcal{A}^5_{\mathcal{G}}}{\vdash_{\mathcal{G}} \forall x.\, x\oplus e = x}\,\forall_r$$

Tabela 1.12: Árvore de demonstração de $\forall x.\, x \oplus e = x$, em $FOL + \mathcal{G}$.

Definição 1.31 (Composição, Idempotência) *A composição de σ com τ denota-se por $\sigma\tau$ e define-se como sendo $\{x \mapsto (x\sigma)\tau \mid x \in V\}$. Uma substituição diz-se idempotente se $\sigma\sigma = \sigma$.*

Da definição, resulta que $x(\sigma\tau) = (x\sigma)\tau$, para todo o $x \in V$, para além disso é simples verificar que a composição de substituições é associativa. Destas observações concluimos que se pode escrever simplesmente $x\sigma\tau$, omitindo parêntesis, sem perigo de ambiguidade. Para ilustrar este conceito dadas as substituições, $\sigma = \{x \mapsto y, y \mapsto w, z \mapsto g(y)\}$ e $\tau = \{y \mapsto f(b), z \mapsto x\}$ temos $\sigma\tau = \{x \mapsto f(b), y \mapsto w, z \mapsto g(f(b)), z \mapsto x\}$ e $\tau\sigma = \{x \mapsto y, y \mapsto f(b), z \mapsto y\}$. Na subsecção 3.1.2 do capítulo 3 são apresentados métodos para cálculo destas composições. Relativamente ao conceito de idempotência, chama-se a atenção para o facto de uma substituição σ ser idempotente se e só se $Dom(\sigma) \cap Var(Cod(\sigma)) = \emptyset$, onde $Var(X)$ indica o conjunto de todas as variáveis livres, presentes em X. Como exemplo, a substituição σ não é idempotente uma vez que $Dom(\sigma) \cap Var(Cod(\sigma)) = \{x, y, z\} \cap \{y, w, g(y)\} = \{y\}$. Para confirmar que $\sigma\sigma \neq \sigma$ basta comparar o resultado das respectivas aplicações a x (que é mapeado em y por σ), e de facto $x\sigma\sigma = \sigma y = w \neq y = x\sigma$. Já a substituição τ é idempotente uma vez que $Dom(\tau) \cap V(Cod(\tau)) = \emptyset$.

Definição 1.32 (Instância, Generalização) *A relação $\precsim$, dita de instanciação, definida sobre um conjunto de termos, é dada por $s \precsim t$ se existe σ tal que $s = t\sigma$. Podemos, de forma análoga, definir uma relação de instanciação entre substituições, neste caso tem-se $\tau \precsim \theta$ se existe σ tal que $\tau = \theta\sigma$. Para os termos (substituições) utilizados nas definições anteriores diz-se que s é uma instância de t (τ é uma instância de θ) e que t é uma generalização de s (θ é uma generalização de τ).*

Como exemplo, dados os termos $s = g(f(x), z)$ e $t = g(y, x)$ tem-se $s \precsim t$, uma vez que $s = t\{y \mapsto f(x), x \mapsto z\}$. De forma análoga, dados $\tau = \{v \mapsto g(f(x)), w \mapsto z\}$ e $\theta = \{v \mapsto y, w \mapsto x\}$ tem-se $\tau \precsim \theta$, uma vez que $\tau = \theta\{y \mapsto f(x), x \mapsto z\}$. A relação $\precsim$ é uma pré-ordem, ou seja, é

reflexiva (pois $t = t\varepsilon$ e $\sigma = \sigma\varepsilon$) e transitiva (se $r = s\sigma$ e $s = t\sigma'$ então $r = t\sigma'\sigma$, analogamente para substituições).

A utilização do mesmo símbolo para representar ambas as relações de instanciação (em termos e em substituições) é motivada pelo paralelismo (patente nas suas definições) e forte ligação que existe entres os dois conceitos. Repare-se que quanto mais geral for uma substituição, mais gerais serão os termos obtidos através da sua aplicação, isto é, se $\tau \precsim \theta$ então $t\tau \precsim t\theta$, para todo termo t, uma vez que existe σ tal que $t\tau = t(\theta\sigma) = (t\theta)\sigma$. Da mesma forma, a relação de instanciação entre dois termos é preservada pela aplicação de uma mesma substituição, ou seja se $s \precsim t$ então também $s\tau \precsim t\tau$, pois existe σ tal que $s\tau = (t\sigma)\tau = t(\sigma\tau)$.

Definição 1.33 (Unificador, Unificador Mais Geral) *Diz-se que σ é um unificador de $s, t \in T$ se $s\sigma = t\sigma$. Um unificador σ de s e t diz-se mais geral (mgu) se, para qualquer outro unificador τ de s e t se tem $\tau \precsim \sigma$.*

Nem todos os pares de termos possuem um unificador, são exemplos de termos não unificáveis $f(x)$ e $g(x)$, e também x e $f(x)$. No entanto, para os pares de termos que possuam um unificador é sempre possível determinar um unificador mais geral. Consideremos os termos $s = f(g(x), y, z)$ e $t = f(u, h(u), v)$. São unificadores de s e t, as seguintes substituições:

$$\begin{aligned}
\sigma_1 &= \{u \mapsto g(x), y \mapsto h(g(x)), v \mapsto z\},\\
\sigma_2 &= \{u \mapsto g(x), y \mapsto h(g(x)), z \mapsto v\},\\
\sigma_3 &= \{u \mapsto g(a), y \mapsto h(g(a)), z \mapsto v\},\\
\sigma_4 &= \{u \mapsto g(a), y \mapsto h(g(a)), z \mapsto f(b)\}
\end{aligned}$$

Destas substituições, são *mgu*s apenas σ_1 e σ_1. Temos que $\sigma_2 = \sigma_1\{z \mapsto v\}$, $\sigma_3 = \sigma_1\{x \mapsto a\}$ e $\sigma_3 = \sigma_1\{x \mapsto a, v \mapsto f(b)\}$. Chama-se ainda a atenção para a substituição $\{z \mapsto v\}$, que permite obter o *mgu* σ_2 por composição com σ_1. Esta substituição diz-se uma *renomeação de variáveis*, por conter apenas variáveis no domínio e codomínio e ser injectiva. Unificadores mais gerais são únicos a menos de composição com este tipo de substituições.

Para terminar notamos que todo o capítulo 3.1.2 é dedicado ao cálculo de unificadores mais gerais, uma vez que se trata se uma parte muito importante do processo de dedução automática.

Capítulo 2

Estruturas de Base

Neste capítulo são apresentadas bibliotecas e estruturas, que estão na base das implementações efectuadas durante a realização deste trabalho. Na secção 2.1 damos a conhecer, muito brevemente, algumas características de Haskell, a linguagem utilizada. Na secção 2.2 é descrita a implementação de uma biblioteca de operações sobre estruturas recursivas, utilizada, entre outras coisas, para manipular termos e fórmulas. Por fim, na secção 2.3, após analisarmos melhor o processo de dedução em sistemas de Gentzen, implementamos estruturas de dados para representar termos e fórmulas.

2.1 A Linguagem de Programação Haskell

A implementação do sistema de demonstração foi efectuada na linguagem de programação Haskell. Trata-se de uma linguagem funcional pura, não estrita (*lazy*), e fortemente tipada. De seguida irão ser apresentadas estas e outras características importantes da linguagem em causa, por forma a tornar o texto mais auto-contido, e motivar as abordagens seguidas na implementação de alguns dos algoritmos descritos ao longo do mesmo. Esta secção não pretende de modo algum servir como tutorial de Haskell, para o efeito existe uma extensa bibliografia disponível[1]. Leitores familiarizados com esta ou outras linguagens funcionais, poderão ignorar esta secção ou parte dela, e consultá-la apenas em caso de necessidade. Durante o texto, caixas rectangulares serão utilizadas para apresentar código de programação, a menos de nota em contrário. O símbolo $\equiv$ será usado para denotar igualdade semântica entre expressões na linguagem.

Programação funcional

Haskell é uma *linguagem funcional*, como tal *funções são entidades de primeira ordem*, em particular podem ser passadas como parâmetros, e podem ser devolvidas por outras funções. Uma forma comum de construir novas funções é através da aplicação parcial de funções já definidas, no exemplo a seguir, tem-se $g = f\ 0 \equiv g\ y = f\ 0\ y \equiv g\ y = 2y$.

```
f       :: Int → Int → Int          g :: Int → Int
f x y =  x+2∗y                      g = f 0
```

[1] `http://www.haskell.org/bookshelf/`

Funções que recebem outras como argumentos, dizem-se de *ordem superior*. Como exemplo, a função **map** recebe como argumento uma função h e uma lista de elementos, e devolve a lista resultante da aplicação de h a cada um dos elementos da lista original.

```
map h [x1, x2, ..., xn] = [h x1, h x2, ..., h xn]
```

Tipos de Dados

A linguagem em causa é *fortemente tipada*, o que significa que todos os elementos têm um tipo bem definido, funções inclusive. Os tipos podem ser *atómicos*, como **Int**, **Char**, ou **Double**, ou *não atómicos*, no sentido em que são construídos a partir de tipos mais simples por produtos (registos), coprodutos (reuniões disjuntas), ou exponenciais (construção de funções). Os tipos Pair e CoPair são, respectivamente, produtos e coprodutos de inteiros por caracteres. São do tipo Pair os elementos, Prod 1 'a' e Prod 5 '!', e do tipo CoPair os elementos, CoInt 1 e CoChar '!'.

```
data Pair   = Prod  Int Char
data CoPair = CoInt Int | CoChar Char
```

Podemos definir tipos mais complexos através de definições recursivas. Em seguida é apresentada uma lista de inteiros, ListInt. São elementos deste tipo, Empty, e também Node (1 Node (2 (Node 0 Empty))).

```
data List = Empty | Node Int List
```

Finalmente, o tipo *Função* é construído aplicando (→) a outros tipos. Por exemplo, a função g anteriormente definida tem tipo **Int→Int**. A função f, por sua vez, tem tipo **Int→(Int→Int)**, isto é, dado um inteiro, devolve uma nova função (**Int→Int**), que dado um segundo inteiro, devolve um inteiro. Porque (→) é associativo à direita podemos escrever simplesmente **Int→Int→Int**. Portanto f 0 6 ≡ 0+2∗6 ≡ 12 e, como já vimos, f 0 ≡ g y = 2y.

Polimorfismo e Classes

Usando *tipos variáveis* podem ser declaradas estruturas polimórficas. Os exemplos anteriores, de produto, coproduto e listas, podem ser generalizados para quaisquer tipos, e não apenas inteiros e caracteres.

```
data Pair   a b = Prod  a b
data CoPair a b = CoInt a   | CoChar b
data List   a   = Empty     | Node a (List a)
```

Desta forma podemos ter Prod 1 1 do tipo Prod **Int Int**, como Prod 'a' 'b' do tipo Pair **Int Int**, ou até mesmo Prod (CoInt 'a') (Node 1 Empty) do tipo Pair (CoPair **Char** b) (**List Int**). Estes tipos de dados estão pré-definidos pela linguagem, com uma sintaxe um pouco diferente.

```
data (a,b)          = (a,b)
data Either a b = Left a | Right b
data [a]            = [] | a : [a]
```

Como exemplo, temos que (1,2) é um par (**Int**,**Int**), **Left** 'a' um copar **Either Char** b, e 1:3:4:[ ] uma lista de inteiros. Esta última pode ser escrita na forma [1,3,4] . São exemplos de funções polimórficas, **fst** que devolve o primeiro elemento de um par, isLeft que verifica se um copar é o caso **Left** ou não, e a função **map**, anteriormente utilizada, que mapeia uma lista com uma função.

```
fst    :: (a,b)→a
isLeft :: Either a b→Bool
map    :: (a→b)→[a]→[b]
```

Existem também funções que actuam sobre uma *classe* restrita de elementos. É o caso dos operadores de igualdade e desigualdade, que só podem ser utilizados com tipos que sejam membros da classe **Eq**, ou de operadores de comparação, máximo e mínimo, que só estão disponíveis para membros da classe **Ord**. A estas funções e operadores damos o nome de *métodos*. Como podemos ver na declaração da classe **Ord**, esta exige que o tipo de dados para que é definida pertença à classe **Eq**, desta forma é definida uma *hierarquia de classes*.

```
class  Eq a where
   (==) :: a → a → Bool
   (/=) :: a → a → Bool
```

```
class Eq a ⇒ Ord a where
   compare                  :: a → a → Ordering
   (<), (≤), (≥), (>)       :: a → a → Bool
   max, min                 :: a → a → a
```

A definição de pertença de um tipo de dados a uma classe é efectuada através da criação de *instâncias*. Consideremos a classe de *Functores*, de todos os tipos mapeáveis e a sua instância para árvores binárias.

```
class  Functor f  where
   fmap :: (a → b) → f a → f b
```

```
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)

instance Functor BinTree  where
   fmap f (Leaf x)       = Leaf (f x)
   fmap f (Node s x t) = Node (fmap f s) f x (fmap f t)
```

O seu único método, fmap, é uma generalização da função **map** para outros tipos de dados que não listas. Comparemos as suas assinaturas.

```
fmap :: Functor f ⇒ (a → b) → f a → f b
map  ::             (a → b) → [a] → [b]
```

Avaliação *Lazy*

Uma das características mais particulares de Haskell é o método de avaliação utilizado, avaliação normalmente denominada *lazy*, em que os valores das estruturas de dados serão calculados apenas quando necessário. Por exemplo, no código seguinte, os segundos membros dos tuplos nunca são avaliados, em particular, no segundo exemplo não ocorre erro de divisão por zero, pois esta nunca será efectuada.

```
fst (h x, h y)
fst (h x, 1/0)
```

Esta característica permite definir e trabalhar com estruturas infinitas. No exemplo seguinte é definida uma lista infinita, xs, com todos os inteiros pares positivos, através do mapeamento de [1,2..] , a lista infinita de todos os inteiros positivos, com a operação (2∗). A função h mapeia cada elemento da lista xs no seu dobro e devolve os n primeiros elementos. Se for necessário avaliar o resultado de h 20 (o que acontece quando executamos **print** (h 20), por exemplo), apenas os 20 primeiros elementos de xs serão avaliados.

```
xs :: [Int]                    h    :: Int → [Int]
xs = map (2∗) [1,2..]          h n  = take n xs
```

Gestão de Memória e Partilha de Valores

As estruturas, uma vez avaliadas, permanecem nesse estado até deixarem de ser necessárias, altura em que são descartadas através de *garbage collection*. No exemplo seguinte, considerando a lista xs anteriormente definida, serão necessários os 1000 primeiros elementos de xs para calcular uma soma, e em seguida os 4 primeiros elementos da mesma lista para calcular o produto. Durante o cálculo do produto, nenhum elemento será reavaliado.

```
k :: Int
k = sum(take 1000 xs) + product(take 4 xs)
```

Repare-se que, apesar dos ganhos a nível de eficiência temporal, neste caso são conseguidos à custa de uma pior gestão de memória. Todos os 1000 primeiros elementos de xs ficarão em memória até o produto ter sido completamente calculado. Em muitos casos, a retenção cumulativa de elementos partilhados tem um impacto muito grande a nível de consumo de memória. Essa situação é conhecida por fuga de espaço *(space leak)*. Por outro lado, a possibilidade de partilhar estruturas em memória, permite também poupar recursos na definição de estruturas.

A seguinte função permite construir uma árvore binária completa, de altura n, em tempo e espaço linear, uma vez que os filhos de cada nó são partilhados.

```
linBin :: Int → BinTree Int
linBin 0 = Leaf 0
linBin n = Node t n t where t = linBin (n-1)
```

Convém notar que esta partilha se refere apenas à representação em memória. As subárvores esquerda e direita são completamente independentes. Um simples mapeamento da árvore com uma qualquer função, eliminaria toda a partilha de valores, pois na definição fmap para árvores binárias as novas subárvores calculadas não partilham o mesmo valor.

Estruturas e Avaliação Puras, e Monads

Haskell diz-se uma linguagem *pura* no sentido em que não existe a possíbilidade de efeitos colaterais. Uma função, quando aplicada a um mesmo argumento em pontos diferentes do programa, devolve sempre o mesmo valor. Esta característica implica a não existência de variáveis (globais ou locais), em particular todas as estruturas de dados são imutáveis (ou seja, o seu valor, uma vez definido, não se altera). A implementação em Haskell, de algoritmos que, em linguagens não puras, se baseiam em efeitos colaterais, varia de caso para caso. Podemos no entanto destacar um tipo de estruturas, muitas vezes associadas a este tipo de algoritmos, referimo-nos a *Monads*. Estruturas monádicas são normalmente interpretadas como computações encapsuladas (computações com estado, não determinísticas, etc). A classe *Monad* define operações básicas para as manipular. São elas[2] **return**, que consiste numa computação que se limita a devolver um resultado, a operação *bind*, denotada por $\gg=$, que permite sequenciar duas computações transferindo o resultado da primeira para a segunda, e $\gg$, que se limita a encadear duas computações, ignorando o valor calculado pela primeira.

```
class Monad m where
    return :: a → m a
    (>>=)  :: m a → (a → m b) → m b
    (>>)   :: m a → m b → m b
```

O tipo de dados **Maybe** é exemplo de uma estrutura mónadica muito simples. Neste contexto elementos de **Maybe** a são interpretados como computações que podem terminar com sucesso e devolver um resultado do tipo a, ou seja (**Just** a), ou falhar (**Nothing**). Apresentamos de seguida a sua instância da classe **Monad**.

```
data Maybe a = Nothing | Just a
```

[2] É omitido o método **fail** para simplificar a apresentação. O tratamento de erros será efectuado com base nos métodos da classe *Monad Error*.

```
instance Monad Maybe where
    (Just x) >>= k  =  k x
    Nothing  >>= k  =  Nothing
    return          =  Just
```

Tomemos com exemplo as funções reduce e increase. A primeira, dado um inteiro, falha se este não for positivo, caso contrário devolve com sucesso o seu predecessor. A função increase devolve sempre com sucesso o sucessor do número recebido.

```
reduce n | n≤0 = Nothing          increase n = Just (n+1)
         | n >0 = Just (n−1)
```

Tem-se então **return** 2 ≫=reduce ≫=reduce ≫=increase ≡ **Just** 1, e **return** 2 ≫=reduce ≫=reduce ≫= reduce ≫=increase ≡ **Nothing**. Repare-se que, a partir do momento em que se obtém um **Nothing**, qualquer sequenciação de computações posterior falha, ou seja devolve **Nothing**.

Outros *Monads* comuns, para além de *Maybe*, são (*Either e*) que modela computações com vários tipos de erro possíveis (do tipo e), [ ] (listas) interpretadas como computações não deterministas, *State* que modela computações com estados, e IO para *inputs* e *outputs*.

2.2 Operações Genéricas sobre Estruturas Recursivas

Durante uma primeira fase de implementação dos algoritmos apresentados neste trabalho foram sendo identificados vários padrões a nível de processamento (alterações estruturais, travessias, etc.) de algumas estruturas de dados, nomeadamente *Termos* e *Fórmulas*. Implementaram-se então funções auxiliares (e na sua maioria) de ordem superior, que permitem reproduzir esses mesmos padrões. Observou-se posteriormente que as operações auxiliares sobre *Termos* e *Fórmulas* eram bastante idênticas, e facilmente adaptáveis a quaisquer tipos de dados simplesmente recursivos. Assim sendo foram generalizadas para operações genéricas sobre uma classe de estruturas de dados simplesmente recursivas. Na base desta definição está uma classe SubData de tipos de dados que contém subdados. Os dois métodos principais, subData e updSubData, permitem aceder e actualizar os subdados. Os métodos mapSubData e isAtomData estão definidos por omissão a partir dos dois anteriores, e permitem mapear subdados e verificar se um elemento é atómico (isto é, se não contêm subdados), respectivamente.

```
class SubData a b | a→b where
    subData     :: a→[b]
    updSubData  :: a→[b]→ a
    isAtomData  :: a→Bool
    mapSubData  :: (b→b)→a→a

    isAtomData      = null ∘ subData
    mapSubData f a = updSubData a (map f (subData a))
```

Para ilustrar os conceitos introduzidos consideremos árvores de inteiros, definidas da forma habitual. A sua instanciação na classe de estruturas recursivas requer apenas a definição dos métodos de acesso e actualização dos subdados, que consistem na projecção e injecção do segundo campo (a lista de subárvores), respectivamente.

```
data TreeInt = Node Int [TreeInt]

instance RecData TreeInt          where
instance SubData TreeInt TreeInt where
    subData    (Node _ ts)    = ts
    updSubData (Node n _ ) ts = Node n ts
```

2.2.1 Classe de Estruturas Recursivas

São estruturas (simplesmente) recursivas todas aquelas cujos subdados sejam do mesmo tipo que a estrutura original. Ou seja, todas as da forma SubData a a.

```
class SubData a a ⇒ RecData a where
```

Apesar de se tratar de uma classe muito simples, os poucos métodos de que dispõe permitem definir uma grande variedade de funções genéricas. Uma das principais, paraData, recebe uma função binária $f :: a \to [b] \to b$ e usa-a para operar uma dada estrutura a com a lista de valores resultantes da aplicação recursiva (de paraData) aos seus subdados. O nome utilizado deve-se ao facto de este tipo de transformação ser conhecido por *paramorfismo* [2].

```
paraData    :: RecData a ⇒ (a→[b]→b)→a→ b
paraData f d = f d (map (paraData f) (subData d))
```

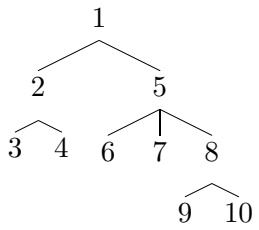

Figura 2.1: Árvore t_1

Como exemplo, para a árvore t_1 aqui apresentada (figura 2.1), e dada a função f definida por[3] f Node n bs =n / **product** bs, obter-se-ia,

$$\text{paraData f } t_1 = \frac{1}{\frac{2}{3\times 4} \times \frac{5}{6\times 7\times \frac{8}{9\times 10}}}.$$

Estão definidas, à custa da função paraData funções estruturais como o cálculo da complexidade, ou seja do número de nós de uma estrutura (complexityData), profundidade máxima (maxDepthData), e profundidade mínima (minDepthData).

```
complexityData :: RecData a ⇒ a → Int
complexityData = paraData (\x ys→ 1+sum ys)
```

[3]Uma implementação correcta requer a conversão explícita do inteiro n para real, aqui omitida para simplificar a exposição.

```
maxDepthData :: RecData a ⇒ a → Int
maxDepthData = paraData (\x ys→if isAtomData x then 0 else 1+maximum ys)

minDepthData :: RecData a ⇒ a → Int
minDepthData = paraData (\x ys→if isAtomData x then 0 else 1+minimum ys)
```

O mapeamento da informação contida em cada um dos nós de uma estrutura simplesmente recursiva pode também ser exprimido através da função paraData. A função mapNodesData implementa este caso particular. Em cada passo recursivo é aplicada a função de mapeamento à estrutura em causa, e os seus subdados actualizados com o resultado do mapeamento dos subdados da estrutura original. Esta função pode devolver uma estrutura recursiva com um tipo diferente do original – é possível, por exemplo, transformar uma árvore de inteiros numa árvores de caracteres – por outro lado, a configuração da estrutura não é alterada.

```
mapNodesData :: (RecData a,RecData b) ⇒ (a→b)→a→b
mapNodesData f = paraData (\a bs→ updSubData (f a) bs)
```

Outro tipo de mapeamento, que iremos designar por *mapeamento ascendente* (botUpMapData), consiste em recursivamente mapear os subdados, actualizar a estrutura original com o resultado obtido, e em seguida aplicar a função de mapeamento. A diferença relativamente ao procedimento anterior é bastante subtil, consiste apenas na ordem pela qual é aplicada a função de mapeamento e são actualizados os subdados.

```
botUpMapData    :: RecData a ⇒ (a→a)→a→a
botUpMapData f = paraData (\a as→f (updSubData a as))
```

Neste caso a estrutura resultante tem sempre o mesmo tipo da original, no entanto a configuração obtida pode ser completamente diferente. Como exemplo, se efectuarmos o mapeamento ascendente de uma árvore com uma função reverseSubData que inverta a ordem dos subdados de uma estrutura recursiva, iremos obter uma árvore simétrica (figura 2.2).

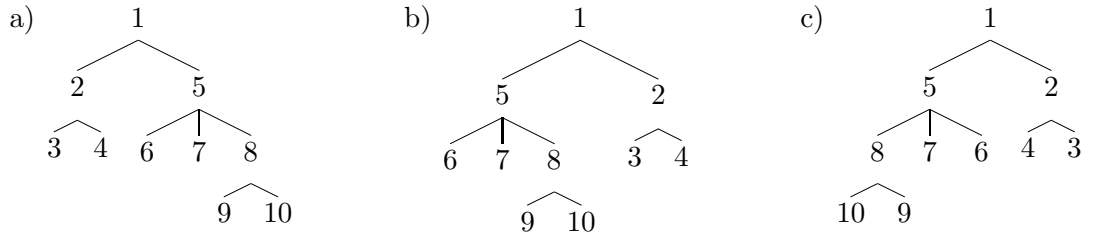

Figura 2.2: a) Árvore t_1 b) reverseSubData t_1 c) botUpMapData reverseSubData t_1

Uma terceira possibilidade consiste em mapear a estrutura começando pela raiz, e avançando posteriormente para os subdados da estrutura resultante, ou seja, um *mapeamento descendente* (topDownMapData). Os subdados sobre os quais é efectuada a recursão não são os da estrutura original, e portanto, ao contrário das anteriores, esta função não é um caso particular de paraData.

```
topDownMapData      :: RecData a ⇒ (a→a)→a→a
topDownMapData f d = mapSubData (topDownMapData f) (f d)
```

Se para o exemplo anterior um mapeamento descendente iria devolver o mesmo resultado, tal não acontece no caso geral. Um mapeamento ascendente de uma árvore com uma função que acrescenta um novo filho (folha) à raiz duplicaria o número de nós da original, o mapeamento descendente correspondente resultaria numa árvore de profundidade infinita, com um número infinito de nós.

Outro padrão comum, para além de recursão estrutural uniforme, consiste em operar subdados segundo uma determinada travessia (pré-ordem, pós-ordem...). A função foldrData recebe uma operação binária $\oplus$, um elemento e e uma estrutura recursiva a, e devolve $s_0 \oplus (s_1 \oplus (\ldots s_{n-1} \oplus (s_n \oplus e)\ldots))$, onde s_i é a i-ésima subestrutura de a segundo uma travessia em pré-ordem, e $s_0 = a$. Foram também implementadas, foldlData e foldlData', variantes com associação à esquerda, ou seja $(\ldots((e \oplus s_0) \oplus s_1)\ldots s_{n-1}) \oplus s_n$, sendo a última baseada em **foldl'** e portanto estrita.

```
foldrData :: RecData a ⇒ (a→b→b)→b→a→b
foldrData f e a
    | isAtomData d = f a e
    | otherwise    = f a (foldr (flip $ foldrData f) e (subData a))
```

```
foldlData, foldlData' :: RecData a ⇒ (b→a→b)→b→a→b
```

Uma aplicação simples de foldrData consiste em listar todos os subdados de uma estrutura, em pré-ordem. Esta função denomina-se listSubData e satisfaz a identidade, foldrData $\oplus$ e = **foldr** $\oplus$ e $\circ$ listSubData, e analogamente à esquerda, foldlData $\oplus$ e = **foldl** $\oplus$ e $\circ$ listSubData.

```
listSubData :: RecData a ⇒ a → [a]
listSubData = foldrData (:) []
```

Convém acrescentar que, a listagem de todos os subdados não implica a existência de múltiplas cópias dos mesmos em memória, a estrutura inicial é partilhada por todas as posições da lista que se limitam a 'apontar' para a subestrutura adequada. Esta partilha de valores comuns ocorre naturalmente em *Haskell*. De referir também que esta função é um bom produtor, ou seja, o resultado é uma *stream*, isto é, apenas os elementos consumidos são produzidos (avaliados) [30]. A função complexityData pode ser implementada de forma bastante eficiente (bem mais do que a anteriormente descrita) como complexityData =**length** $\circ$listSubData.

As funções até agora apresentadas limitam-se a processar apenas uma estrutura recursiva. A funçao zipWithData, combina duas estruturas recursivas estruturalmente, com base numa função binária, e devolve uma terceira estrutura recursiva.

```
zipWithData        :: (RecData a,RecData b,RecData c)⇒(a→b→c)→a→b→c
zipWithData f s t = updSubData (f s t)
                        (zipWith (zipWithData' f) (subData s) (subData t))
```

Foram implementadas muitas outras funções, nomeadamente as variantes de mapeamento com acumulação de dados topDownMapAccumData, botUpMapAccumRData e botUpMapAccumLData, as funções anyData e allData que verificam se uma determinada propriedade é satisfeita por alguma das subestruturas (no primeiro caso) ou por todas (no segundo), zipWithData', semelhante a zipWithData mas que devolve um resultado linear, bem como algumas variantes monádicas de funções já referidas, entre elas paraDataM, foldMData, foldM_Data, zipWithMData, zipWithM_Data, zipWithMData' e zipWithM_Data'.

```
topDownMapAccumData :: RecData a ⇒ (acc→a→(acc,a))→acc→a→a
botUpMapAccumRData  :: RecData a ⇒ (acc→a→(acc,a))→acc→a→(acc, a)
botUpMapAccumLData  :: RecData a ⇒ (acc→a→(acc,a))→acc→a→(acc, a)
anyData             :: RecData a ⇒ (a→Bool)→a→Bool
allData             :: RecData a ⇒ (a→Bool)→a→Bool
zipWithData'        :: (RecData a,RecData b)⇒(a→b→c)→a→b→[c]
```

```
paraDataM       :: (Monad m, RecData a) ⇒ (a→[b]→m b)→a→ m b
foldMData       :: (Monad m, RecData a) ⇒ (b→a→m b)→b→a→m b
foldM_Data      :: (Monad m, RecData a) ⇒ (b→a→m b)→b→a→m ()
zipWithMData'   :: (RecData a,RecData b,Monad m)⇒(a→b→m c)→a→b→m [c]
zipWithM_Data'  :: (RecData a,RecData b,Monad m)⇒(a→b→m())→a→b→m ()
zipWithDataM    :: (RecData a,RecData b,RecData c,Monad m)
                ⇒ (a→b→m c)→a→b→m c
```

São exportadas instâncias de Rec para os tipos de dados Lista (Data.**List**.[a]) e Árvores Generalizadas (Data.Tree.Tree a). As suas definições são de seguida apresentadas.

```
instance RecData [a] where
instance SubData [a] [a] where
    subData []     = []
    subData (x:xs) = [xs]
    updSubData [] _          = []
    updSubData (x:xs) [ys]   = (x:ys)
```

```
instance RecData (Tree a) where
instance SubData (Tree a) (Tree a) where
    subData    (Node _ ts)    = ts
    updSubData (Node x _ ) ts = Node x ts
```

2.2.2 Classe de Estruturas Mutuamente Recursivas

Para além da biblioteca de operações sobre estruturas simplesmente recursivas, foi implementada a sua generalização para estruturas mutuamente recursivas. Estas foram particularmente úteis na manipulação de multiequações e multitermos, estruturas de dados utilizadas na imple-

mentação do algoritmo de unificação de Martelli e Montanari (secção 3.2).

Estamos perante estruturas mutuamente recursivas, se os subdados tiverem tipos alternados.

```
class    (SubData a b, SubData b a) ⇒ MRecData a b where
instance (SubData a b, SubData b a) ⇒ MRecData a b where
```

Um exemplo simples de estruturas deste tipo, são árvores com elementos de tipos alternados.

```
data TreeA a b = TreeA a [TreeB b a]
data TreeB b a = TreeB b [TreeA a b]
```

A adaptação das operações de ordem superior para este caso, de um modo geral, é trivial. Em vez de ser dada uma função como argumento, são dadas duas, uma para cada tipo de subdados. Como exemplo, vejamos a generalização das funções paraData, e botUpMapMRData.

```
paraMRData :: MRecData a b ⇒ (a→[d]→c)→(b→[c]→d)→a→c
paraMRData f g d = f d (map (paraMRData g f) (subData d))


botUpMapMRData    :: MRecData a b ⇒ (a→a)→(b→b)→a→a
botUpMapMRData f g = paraMRData (\a bs→f (updSubData a bs))
                                (\b as→g (updSubData b as))
```

Consideremos uma árvore do tipo TreeA **Int Char**, com elementos do tipo inteiro e caracter. A figura 2.3 mostra o resultado do seu mapeamento com a função reverseSubData para TreeA **Int Char** e **id** para TreeB **Char Int**.

Figura 2.3: a) Árvore t_1 b) botUpMapMRData reverseSubData **id** t_1

A função listSubData :: RecData a⇒a→[a], é exemplo de uma operação que não é trivialmente generalizável para o caso mutuamente recursivo, pois não são permitidas listas heterogéneas (com elementos de mais do que um tipo). A solução adoptada foi devolver uma lista de coprodutos.

```
listSubMRData :: MRecData a b ⇒ a → [Either a b]
listSubMRData = foldrMRData (\a e→Left a:e) (\b e→Right b:e) []
```

Por uma questão de conveniência, são também disponibilizadas funções que listam apenas as subestruturas de um dos dois tipos.

```
listSubMRDataA :: MRecData a b ⇒ a → [a]
listSubMRDataA = foldrMRData (:) (\b as→as) []

listSubMRDataB :: MRecData a b ⇒ a → [b]
listSubMRDataB = foldrMRData (\a bs→bs) (:) []
```

2.2.3 Considerações

A abordagem seguida revelou-se muito vantajosa a médio prazo. Tomando como base as funções auxiliares aqui apresentadas, raciocínios sobre vários algoritmos e consequente implementação foram bastante simplificados. Tratam-se de funções com um poder expressivo considerável, e que, para além disso, de um modo geral implementam 'estratégias' já estudadas, com propriedades conhecidas [13, 2], algumas vezes generalizações simples de funções muito comuns em contextos mais restritos (por exemplo listas) [14]. A abstracção destas estratégias para um contexto genérico de estruturas simplesmente recursivas simplificou de facto a sua implementação. Permitiu ignorar as características particulares de cada uma das estruturas, e que são redundantes neste contexto. Evitou duplicação de código, diminuindo assim o esforço na sua criação e manutenção. Permitiu que o código desenvolvido fosse utilizado não só nas estruturas para as quais foi originalmente previsto, mas também noutras, o que, por exemplo, simplificou a criação de testes de correcção.

Este estilo de programação, a nível da estrutura dos tipos de dados, denomina-se *programação genérica*. Algumas linguagens derivadas (extensões) de *Haskell*, como *Generic Haskell* [22] e *PolyP* [25] têm como base este paradigma. Mais recente, o *Atribute Grammar System*[6, 4] consiste num pré-processador que permite gerar automaticamente funções semânticas, *catamorfismos*, e outras definições adicionais para os tipos de dados utilizados. Existem também bibliotecas de programação genérica que podem ser directamente importadas como *Strafunski* [19] e *Scrap Your Boilarplate* [20], esta última já incluída em versões recentes do *Glasgow Haskell Compiler* no módulo *Data.Generics*. Poderá valer a pena explorar algumas destas ferramentas, em particular *Scrap Your Boilarplate*.

2.3 Estruturas de dados Termos e Fórmulas

2.3.1 Metavariáveis e Parâmetros

Como vimos anteriormente (subsecção 1.2.3), demonstrações em sistemas de Gentzen podem ser efectuadas partindo do sequente a demonstrar e aplicando regras de inferência no sentido inverso, sucessivamente. No caso das regras $\forall_l$ e $\exists_r$, é necessária a escolha de um termo para substituir as variáveis ligadas ao quantificador em causa. Esta escolha deve ser feita por forma a conseguirmos (se possível) vir a obter fórmulas comuns no antecendentes e consequentes dos sequentes calculados, para criar uma árvore de demonstração. Como na altura em que essas regras são aplicadas a escolha do termo não é, de um modo geral, óbvia, adiamos a decisão para mais tarde. Assim substituímos as variáveis ligadas, não por um termo, mas por *metavariáveis*

(representadas aqui por identificadores precedidos do símbolo '?', e.g. '$?x$') que serão instanciadas quando descobrirmos valores para as mesmas que satisfaçam o nosso objectivo. Na tabela 2.1 apresentamos novamente a demonstração de $\forall x.\, P(x) \;\vdash\; \neg\exists x.\, \neg P(x)$, e a sua construção usando uma metavariável $?x$. Neste caso, após a decomposição do sequente incial, é óbvio que para se obter uma demonstração, basta que fazer $?x \mapsto y$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{P(y)} \;\vdash\; \mathbf{P(y)}}{P(y),\ \neg P(y) \vdash}\,\neg_l}{\forall x.\, P(x),\ \neg P(y) \;\vdash}\,\forall_l}{\forall x.\, P(x),\ \exists x.\, \neg P(x) \;\vdash}\,\exists_l}{\forall x.\, P(x) \;\vdash\; \neg\exists x.\, \neg P(x)}\,\neg_r \qquad \dfrac{\dfrac{\dfrac{\dfrac{\mathbf{P(?x)} \;\vdash\; \mathbf{P(y)}}{P(?x),\ \neg P(y) \vdash}\,\neg_l}{\forall x.\, P(x),\ \neg P(y) \;\vdash}\,\forall_l}{\forall x.\, P(x),\ \exists x.\, \neg P(x) \;\vdash}\,\exists_l}{\forall x.\, P(x) \;\vdash\; \neg\exists x.\, \neg P(x)}\,\neg_r$$

Tabela 2.1: Árvore de demonstração (esq), e a sua construção usando metavariáveis (dir).

Vejamos mais um exemplo, um pouco mais complexo, onde a escolha dos termos a utilizar é menos óbvia (tabela 2.2). O sequente a demonstrar é $\forall x.\, P(x, f(x)) \;\vdash\; \forall x.\, \exists y.\, P(g(x), y)$. Por aplicação das regras no sentido inverso, reduzimos a demonstração ao cálculo de uma instância (relativamente às metavariáveis) comum dos predicados $P(?x, f(?x))$ e $P(g(z), ?y)$. Uma solução é dada pelo unificador (mais geral) $\{?x \mapsto g(z), ?y \mapsto f(g(z))\}$. O estudo e implementação de processos de cálculo de unificadores é apresentado no capítulo 3.

$$\dfrac{\dfrac{\dfrac{\mathbf{P(g(z), f(g(z)))} \;\vdash\; \mathbf{P(g(z), f(g(z)))}}{\forall x.\, P(x, f(x)) \;\vdash\; P(g(z), f(g(z)))}\,\forall_l}{\forall x.\, P(x, f(x)) \;\vdash\; \exists y.\, P(g(z), y)}\,\exists_r}{\forall x.\, P(x, f(x)) \;\vdash\; \forall x.\, \exists y.\, P(g(x), y)}\,\forall_r \qquad \dfrac{\dfrac{\dfrac{\mathbf{P(?x, f(?x))} \;\vdash\; \mathbf{P(g(z), ?y)}}{\forall x.\, P(x, f(x)) \;\vdash\; P(g(z), ?y)}\,\forall_l}{\forall x.\, P(x, f(x)) \;\vdash\; \exists y.\, P(g(z), y)}\,\exists_r}{\forall x.\, P(x, f(x)) \;\vdash\; \forall x.\, \exists y.\, P(g(x), y)}\,\forall_r$$

Tabela 2.2: Árvore de demonstração para $\forall x.\, P(x, f(x)) \;\vdash\; \forall x.\, \exists y.\, P(g(x), y)$.

É importante notar que, se não houver cuidados adicionais, os termos escolhidos para as regras $\forall_l$ e $\exists_r$ poderão violar as condições das regras $\forall_r$ e $\exists_l$ que hajam sido aplicadas. Como exemplo consideremos a seguinte demonstração incorrecta do sequente $\forall x.\, P(x) \;\vdash\; \neg\exists x.\, \neg P(x)$.

O problema da demonstração está na aplicação da regra $\exists_r$. Após a instanciação da metavariável $?x$, a inferência resultante é,

$$\dfrac{P(y),\ \neg P(y) \vdash}{P(y),\ \exists x.\, \neg P(x) \;\vdash}\,\exists_l$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{P(y)} \vdash \mathbf{P(y)}}{P(y),\ \neg P(y) \vdash}\neg_l}{P(y),\ \exists x.\neg P(x) \vdash}\exists_l}{P(y),\ \vdash\ \neg\exists x.\neg P(x)}\neg_r}{\forall x.\, P(x)\ \vdash\ \neg\exists x.\neg P(x)}\forall_l \qquad \dfrac{\dfrac{\dfrac{\dfrac{\mathbf{P(?x)} \vdash \mathbf{P(y)}}{P(?x),\ \neg P(y) \vdash}\neg_l}{P(?x),\ \exists x.\neg P(x) \vdash}\exists_l}{P(?x)\ \vdash\ \neg\exists x.\neg P(x)}\neg_r}{\forall x.\, P(x)\ \vdash\ \neg\exists x.\neg P(x)}\forall_l$$

Tabela 2.3: Árvore de demonstração (esq), e a sua construção usando metavariáveis (dir).

que viola as condições da aplicação dessa regra, pois a variável livre y, escolhida para substituir a variável ligada x na premissa, aparece livre na conclusão. Intuitivamente, a premissa diz-nos que ter ambos, $P(y)$ e $\neg P(y)$, é uma contradição, e daí concluímos (erradamente) que, se tivermos $P(y)$ e existir um x para o qual se verifique $\neg P(x)$, também estamos perante uma contradição.

Para evitar estes problemas, quando removemos um quantificador, associamos à variável livre criada todas as metavariáveis presentes no sequente correspondente, a estrutura resultante é denominada *parâmetro*. Esta informação será utilizada para evitar instanciar metavariáveis com termos que contenham parâmetros dependentes da metavariável em causa. O exemplo anterior, é apresentado na tabela 2.4, desta vez utilizando um parâmetro. Agora é simples verificar que $?x$ não pode ser instanciado com $y\{?x\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{P(?x)} \vdash \mathbf{P(y\{?x\})}}{P(?x),\ \neg P(y\{?x\}) \vdash}\neg_l}{P(?x),\ \exists x.\neg P(x) \vdash}\exists_l}{P(?x)\ \vdash\ \neg\exists x.\neg P(x)}\neg_r}{\forall x.\, P(x)\ \vdash\ \neg\exists x.\neg P(x)}\forall_l$$

Tabela 2.4: Árvore de demonstração (esq), e a sua construção usando metavariáveis (dir).

A criação automática de árvores de demonstração é estudada com mais profundidade no capítulo 4 deste trabalho.

2.3.2 Notação de *de Bruijn*

Enquanto que cada variável livre é univocamente definida pelo seu identificador numa determinada fórmula ou termo, no caso das variáveis ligadas o mesmo identificador pode ser usado por variáveis distintas. Como exemplo, tanto em $(\forall x.\, P(x)) \wedge (\exists x.\, Q(x))$ como em $\forall x.\, (P(x) \wedge \exists x.\, Q(x))$

a variável x aparece uma vez para se referir ao quantificador $\forall$ e outra para se referir ao quantificador $\exists$. No primeiro caso, as variáveis x aparecem em subfórmulas quantificadas disjuntas, mas no segundo ambas as variáveis fazem parte da subfórmula quantificada por $\forall$, acontece que o alcance deste quantificador é bloqueado por $\exists$. Para evitar este segundo caso, que pode levar a algoritmos sobre as fórmulas e termos mais complicados, foi empregue a notação de *de Bruijn*, sugerida em [28] e utilizada, por exemplo, pelo demonstrador FAUST [29]. Nesta notação, as variáveis ligadas são identificadas por números inteiros não negativos que indicam o número de quantificadores entre a sua ocorrência e o quantificador que lhes está associado. Assim $\forall x.\, P(x)$ será representado por $\forall P(0)$, $\forall x.\, \exists y.\, Q(x,y)$ por $\forall\exists Q(1,0)$, e $\forall x.\, (P(x) \wedge \exists x.\, Q(x))$ por $\forall(P(0) \wedge \exists Q(0))$. Uma propriedade interessante desta representação é o facto de cada fórmula fechada ter representação única.

Para determinar quais as variáveis associadas a um determinado quantificador, basta, partindo do mesmo, percorrer a fórmula (descendo na sua estrutura) contando o número de outros quantificadores que apareçam pelo caminho até atingir os predicados. O número calculado durante essa travessia é o número de *de Bruijn* que devemos procurar nos termos dos respectivos predicados. Outro tipo de operação comum envolvendo fórmulas consiste em relacionar informação associada aos quantificadores com as variável ligadas correspondentes. Neste caso a grande diferença é que não nos interessa apenas identificar o número de *de Bruijn* para um determinado quantificador, mas sim decidir a qual dos quantificadores uma fórmula pertence. Uma forma simples de o fazer consiste em efectuar a travessia descendente da fórmula, acumulando a informação de cada quantificador à cabeça de uma lista (inicialmente vazia). O número de *de Bruijn* de uma variável ligada irá indicar a posição da lista em que se encontra a informação do quantificador a que esta está ligada. A figura 2.4 ilustra este método para o caso em que a informação associada aos quantificadores são identificadores de variáveis [4].

2.3.3 Implementação das Estruturas de Dados *Termo* e *Fórmula*

Iremos considerar então três tipos de termos atómicos, as *metavariáveis* (Var), cujos identificadores (VarId) serão *strings*, os parâmetros (Par) identificados por *strings* (Parid) e com um conjunto de identificadores de variáveis associados (Set VarId), e *variáveis ligadas* (Bnd) identificadas por números de *de Bruijn* (BndId). Um termo não atómico é uma função (Fun), representada pela *string* (FunId) e aridade (FunArity) que a identifica, aplicada a uma lista de subtermos ([Term]).

```
data Term = Var{varId   :: VarId}
          | Par{parId   :: ParId, parDeps   :: (Set VarId)}
          | Bnd{bndBrj  :: BndId}
          | Fun{funId   :: FunId, funArity :: Arity, funTerms :: [Term]}
```

Definimos para esta estrutura funções de projecção e actualização dos subtermos, que, no caso de termos elementares, são apenas a constante lista vazia e a função identidade, respectivamente, e, no caso de termos não elementares (funções), são respectivamente a projecção e actualização

[4] Este método é usado, por exemplo, para imprimir fórmulas que utilizam internamente a notação de *de Bruijn*, na sua notação habitual.

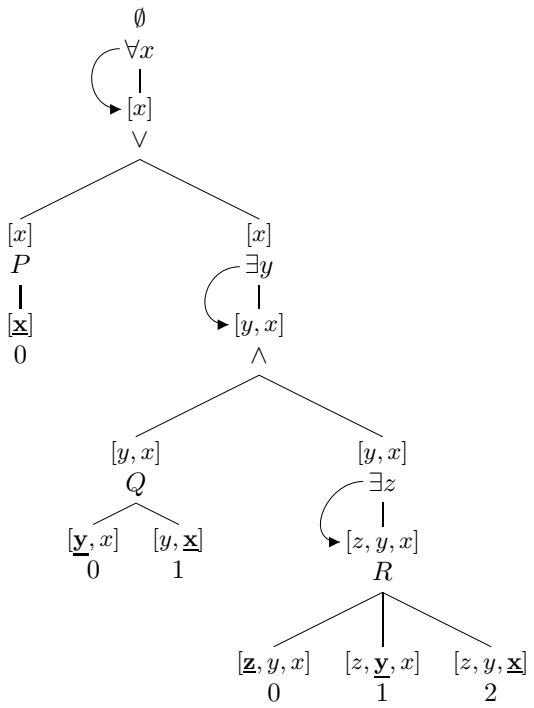

Figura 2.4: Associação das variáveis ligadas da expressão $\forall x.\, Q(0) \vee \exists y.\, P(0,1) \wedge \exists z.\, R(0,1,2)$ aos números de *de Bruijn* correspondentes.

da lista de subtermos.

```
subTerms                          :: Term → [Term]
subTerms (Fun{funTerms = ts}) = ts
subTerms _                        = []

updSubTerms             :: Term→[Term]→Term
updSubTerms (t@Fun{}) ts = t{funTerms = ts}
updSubTerms t         ts = t
```

Estas funções definem termos como instâncias da classe Rec de estruturas simplesmente recursivas.

```
instance RecData Term where
instance SubData Term Term where
    subData    = subTerms
    updSubData = updSubTerms
```

São exemplos de operações definidas à custa desta instância a verificação da inexistência de variáveis ligadas num termo[5] (isProperTerm), ou a recolha de (identificadores de) metavariáveis (getTermVarIds).

[5]Esta propriedade pode ser testada antes do processo de unificação dos termos de predicados, durante o processo de dedução, pois nessa altura já todos os quantificadores devem ter sido removidos.

```
isSemiTerm , isProperTerm :: Term → Bool
isSemiTerm    = anyData isBnd
isProperTerm = not ∘ isSemiTerm

getTermVarIds    :: Term→[VarId]
getTermVarIds t = [varId s | s←subData t, isVar s]
```

A recolha da lista de metavariáveis com parâmetro acumulação, útil para encadear recolhas sem necessidade de concatenação das listas resultantes, pode ser implementada com base num foldrData e numa função addVarId :: Term−[VarId]→[VarId] que adicione identificadores de variáveis a uma lista. Esta função satisfaz a especificação (addTermsVarIds t [ ] =getTermVarIds).

```
addTermVarIds       :: Term→[VarId]→[VarId]
addTermVarIds t xs = foldrData addVarId xs t
```

As fórmulas podem ser *predicados* (Prd), o único tipo de fórmulas atómico, aos quais estão associados uma aridade e uma lista de termos. Fórmulas não atómicas podem ser *conectivas* (Con), definidas pelo símbolo que as representa e pela sua aridade, aplicadas a uma lista de fórmulas, ou então *quantificadores*, definidos apenas pelo seu símbolo, aplicados a uma fórmula (que quantificam) e aos quais será associado um identificador de variáveis apenas para efeitos de impressão de fórmulas.

```
data Form = Prd{prdId   :: PrdId,   prdArity :: Arity, prdTerms :: [Term]}
          | Con{conSym  :: ConSym,  conArity :: Arity, conForms :: [Form]}
          | Qnt{qntSym  :: QntSym,  qntVarId :: String, qntForm  :: Form}
            deriving Show
```

A projecção e actualização das subfórmulas para predicados e conectivas é semelhante à dos subtermos para termos atómicos e funções respectivamente. Os quantificadores têm apenas a particularidade de terem associados uma única subfórmula e não uma lista de subfórmulas.

```
subForms :: Form → [Form]
subForms Prd{}             = []
subForms Con{conForms=as} = as
subForms Qnt{qntForm=a}    = [a]

updForms                  :: Form→[Form]→Form
updForms (a@Prd{}) bs    = a
updForms (a@Con{}) bs    = a{conForms=bs}
updForms (a@Qnt{}) [b] = a{qntForm=b}
```

Também estas funções definem fórmulas como instâncias da classe de estruturas simplesmente recursivas.

```
instance RecData Form       where
instance SubData Form Form where
    subData    = subForms
    updSubData = updForms
```

Para além do acesso a subfórmulas, é importante poder aceder aos termos contidos em fórmulas. As funções prdTerms, updPrdTerms, e mapPrdTerms, implementam a projecção, actualização e mapeamento dos argumentos de um predicado.

```
prdTerms    :: Form → [Term]
updPrdTerms :: Form → [Term] → Form
mapPrdTerms :: (Term → Term) → Form → Form
```

A recolha de termos de uma fórmula consiste em percorrer essa fórmula, adicionando os termos em cada predicado a uma lista.

```
getFormTerms    :: Form → [Term]
getFormTerms = foldrData addTerm []
```

A recolha das metavariáveis de uma fórmula é conseguida, recolhendo os seus termos, e depois as metavariáveis contidas nesses mesmos termos.

```
getFormVarIds :: Form → [VarId]
getFormVarIds = foldr addTermVarIds [] ∘ getFormTerms
```

Note-se que estas funções são completamente *lazy*, o resultado é uma *stream* de metavariáveis, ou seja, estas vão sendo recolhidas apenas à medida que necessárias para serem consumidas.

No caso das fórmulas interessa-nos saber se todas as variáveis são ligadas, isto é, se a fórmula é fechada. Mais uma vez, estendemos este cálculo aos termos (isOpenTerm), e efectuamos o teste sobre a lista de termos que recolhemos da fórmula (isOpenForm e isClosedForm).

```
isOpenTerm :: Term → Bool
isOpenTerm = anyData (\t → isPar t || isVar t)

isOpenForm, isClosedForm :: Form → Bool
isOpenForm   = any isOpenTerm ∘ getFormTerms
isClosedForm = not ∘ isOpenForm'
```

Para terminar, vejamos como implementar a substituição de uma variável ligada, por uma outra fórmula. Este procedimento é efectuado durante a aplicação das regras de inferência para quantificadores. Numa primeira fase efectuamos a travessia descendente da fórmula, até aos predicados, contando pelo caminho o número de quantificadores. De seguida, substituímos em todos os termos de cada predicado, as ocorrências do número de *de Bruijn* calculado pela fórmula desejada.

```
substBndForm :: Term → BndId → Form → Form
substBndForm t = topDownMapAccumData stepdown
    where
    stepdown n (a@Qnt{}) = (n+1,a)
    stepdown n (a@Con{}) = (n,  a)
    stepdown n (a@Prd{}) = (n,  mapPrdTerms (substBndTerm t n) a)
```

A implementação da substituição a nível do termo, é trivial e pode ser implementada com qualquer um dos mapeamentos disponíveis.

```
substBndTerm      :: Term → BndId → Term → Term
substBndTerm t n = botUpMapData (\s→if mkBnd n == s then t else s)
```

2.3.4 Considerações

Muitos demonstradores utilizam outro tipo de representação de termos e fórmulas. Uma estratégia comum é a utilização de *grafos acíclicos orientados* (DAGs), para representar estas estruturas. Uma vantagem desta representação é o facto de ser possível criar partilha de subtermos comuns (que podem ser calculados em tempo linear) [3]. Esta partilha permite reduzir o espaço de memória utilizado, e diminuir o tempo de travessia de termos, por exemplo, em algoritmos de unificação. A implementação efectuada, na verdade já funciona como um DAG, no sentido em que a representação interna é de facto um grafo orientado acíclico, e como iremos ver, é mesmo efectuada alguma partilha de termos comuns. No entanto, como já vimos anteriormente (secção 2.1), esta partilha é difícil de controlar e é transparente. Este último aspecto faz com que não sirva para optimização temporal de algorítmos, pois não há forma de saber se um determinado caminho do DAG já foi percorrido ou não. Por estes motivos poderá valer a pena explorar a implementação de fórmulas e termos como DAGs, de forma explícita.

Capítulo 3

Métodos de Unificação

O processo de unificação é parte fundamental dos sistemas de demonstração automática [37, 39]. Como o número de termos a unificar vai crescendo durante a pesquisa efectuada, um método de unificação lento pode abrandar consideravelmente a pesquisa efectuada (construção da árvore de demonstração). Neste capítulo começamos por descrever o método de unificação de Robinson [3] (secção 3.1), um método simples para determinar um unificador mais geral (*mgu*) de dois termos por recursão estrutural, e uma implementação do mesmo. Na secção 3.2 debruçamo-nos sobre o algoritmo de Martelli e Montanari [23], um algoritmo mais eficiente baseado na resolução de sistemas de multiequações. Por fim apresentamos técnicas de indexação de termos [31], que permitem seleccionar em paralelo vários candidatos para unificação com um *query term*, ou mesmo efectuar a própria unificação em paralelo.

3.1 Algoritmo de Robinson

A abordagem mais directa para calcular o unificador de dois termos consiste em construir essa substituição, à medida que se percorrem simultaneamente as estruturas que definem ambos os termos, começando pelas respectivas raízes. Vejamos alguns exemplos para ganhar alguma intuição sobre os detalhes do processo. Para $s = f(x, g(h(c)))$ e $t = f(a, g(z))$, é simples determinar o unificador $\sigma = \{x \mapsto a, z \mapsto h(c)\}$. Consideremos agora $s' = f(x, g(h(x)))$, note-se que a substituição $\sigma^* = \{x \mapsto a, z \mapsto h(x)\}$ não é um unificador de s' e t, pois $s\sigma^* = f(a, g(h(a))) \neq f(a, g(h(x))) = t\sigma^*$. Se começarmos por criar o mapeamento $\{x \mapsto a\}$, quando processamos os subtermos z e $h(x)$ devemos associar z a $h(a) = h(x)\{x \mapsto a\}$, e portanto o unificador resultante será $\sigma' = \{x \mapsto a, z \mapsto h(a)\}$. Ou seja:

> Durante o procedimento de unificação, o mapeamento até então calculado será aplicado a cada par de subtermos considerados.

Se começarmos por processar z e $h(x)$, então obtemos em primeiro lugar o mapeamento $\{z \mapsto h(x)\}$, ao qual será em seguida 'adicionado' $\{x \mapsto a\}$. Dessa 'adição' resultará $\sigma' = \{z \mapsto h(a), x \mapsto a\} = \{z \mapsto h(a)\}\{x \mapsto a\}$. Ou seja:

> Durante o procedimento de unificação, o mapeamento até então calculado será actualizado através da composição de novas substituições unitárias.

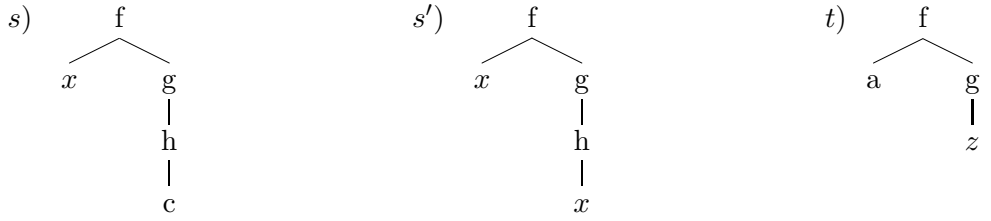

Figura 3.1: Termos s, s' e t. Os pares s e t, e s' e t são unificáveis.

A unificação de dois termos pode no entanto não ser possível. São exemplos de pares de termos trivialmente não unificáveis, $f(x, y)$ e $g(x, y)$, $f(x, y)$ e $h(x, y, z)$, e $f(x, a)$ e $f(x, g(y))$. Em qualquer dos casos, os termos considerados possuem numa mesma posição funções (possivelmente constantes) distintas, o que impossibilita à partida qualquer tipo de unificação. Termos nesta situação dizem-se *incompatíveis*. Consideremos agora o par de termos $f(x, a, c)$ e $f(b, x, c)$, estes compatíveis, logo potencialmente unificáveis. Tentemos achar um unificador, começamos por considerar x e a, originando o mapeamento $\{x \mapsto a\}$, mapeamento esse que deverá ser aplicado aos dois subtermos seguintes, b e x, resultando em b e a, esses sim, incompatíveis, portanto os termos originais não são unificáveis. Quando unificações falham pelos motivos descritos, diz-se que a causa foi um *erro estrutural* (*pattern error*). A verificação de compatibilidade entre as raízes de dois (sub)termos designa-se por *verificação estrutural* (*pattern check*).

Figura 3.2: Pares de termos não unificáveis por erros estruturais, sendo os da esquerda incompatíveis e os da direita compatíveis.

Pode ocorrer ainda um outro problema. Apesar de não existirem erros estruturais, os termos x e $f(x)$ não são unificáveis. Como $f(x)$ contém a variável x, $f(x)\sigma$ irá conter (no sentido estrito) $x\sigma$, para toda a substituição σ. Durante o processo de unificação é então necessário verificar se $x \notin t$, antes de criar uma nova associação $x \mapsto t$. A este passo dá-se o nome de *verificação de ocorrência* (ou *occurs check*). Quando $x \in t$ dizemos que o processo falha com um *erro de ocorrência* (*occurs error*).

Com base nas observações anteriores, passamos a descrever de forma mais precisa um algoritmo de unificação, baseado em recursão estrutural, conhecido por *algoritmo de unificação de Robinson* (ver algoritmo 3.1). Apesar de bastante simples, verifica algumas propriedades muito importantes, nomeadamente um unificador calculado por este processo é sempre um unificador mais geral e idempotente.

De acordo com a descrição do algoritmo 3.1 e com as observações anteriormente efectuadas, a aplicação da substituição σ é efectuada a todo o par de subtermos considerado, porém é fácil verificar que esse passo é, em alguns casos, redundante e bastante dispendioso. Considere-se, por exemplo, que s e t são ambos funções. Nesse caso necessitamos apenas de efectuar uma verificação estrutural e, eventualmente, prosseguir com a unificação dos seus subtermos, a aplicação da substituição poderá ser efectuada mais tarde. O algoritmo 3.2 é equivalente ao

Algoritmo 3.1 *Algoritmo de Unificação de Robinson*

$\sigma \leftarrow \varepsilon$ *(global)*

Unificar(s, t)

```
Caso (sσ, tσ) Seja
 (x ∈ V, y ∈ V)                : Se x ≠ y  Então σ ← σ{x ↦ y}
 (x ∈ V, t')                   : Se x ∉ t' Então σ ← σ{x ↦ t'} Senão erro de ocorrência
 (t', x ∈ V)                   : Se x ∉ t' Então σ ← σ{x ↦ t'} Senão erro de ocorrência
 (f(s1,...,sm), g(t1,...,tm))  : Se f = g  Então Para i ← m Faz Unificar(si, ti)
                                           Senão erro estrutural
```

FimUnificar

primeiro, mas optimizado por forma a evitar substituições redundantes.

Algoritmo 3.2 *Algoritmo de Unificação de Robinson (melhoramento)*

$\sigma \leftarrow \varepsilon$ *(global)*

Unificar(s, t)

```
Se s ∈ V  Então s ← sσ
Se t ∈ V  Então t ← sσ
Caso (s, t) Seja
 (x ∈ V, y ∈ V)                : Se x ≠ y  Então σ ← σ{x ↦ y}
 (x ∈ V, t)                    : Se x ∉ tσ Então σ ← σ{x ↦ tσ} Senão erro de ocorrência
 (t, x ∈ V)                    : Se x ∉ tσ Então σ ← σ{x ↦ tσ} Senão erro de ocorrência
 (f(s1,...,sm), g(t1,...,tm))  : Se f = g  Então Para i ← m Faz Unificar(si, ti)
                                           Senão erro estrutural
FimCaso
```

FimUnificar

3.1.1 Implementação do Algoritmo de Robinson

A implementação deste algoritmo em Haskell levanta, à partida, duas questões: como modelar os erros de unificação e a existência de uma variável global (a substituição σ). Comecemos por analisar o primeiro caso. O resultado de uma unificação pode ser um erro ou uma substituição, portanto a abordagem mais simples consiste em considerar o tipo do resultado como sendo a reunião disjunta de dois tipos, *Erro de Unificação* e *Substituição*.

```
data UniError = Occurs | Pattern
type Uni      = Either UniError Subst
```

Este é um caso particular de reunião disjunta de dois tipos, utilizada aqui para 'encapsular' uma computação cujo resultado será uma substituição. Convém notar que o procedimento de unificação pode, no entanto, ser composto por outros procedimentos (computações) auxiliares,

e que o resultado de cada um destes não tem, necessariamente, que ser uma substituição. Esta observação sugere a generalização do tipo Uni, atrás definido para a reunião disjunta de UniError com qualquer outro tipo.

```
type Uni a = Either UniError a
```

Uni a representa então uma computação, parte de um algoritmo de Unificação, que devolve um resultado de tipo a e que pode falhar devido a um erro estrutural ou de ocorrência. Em Haskell, o 'encapsulamento' de computações é normalmente realizado através de Monads, e, de facto, aplicações parciais da forma **Either** b, em particular Uni = **Either** UniError, são instâncias da classe **Monad** e da sua subclasse ErrorMonad. A primeira disponibiliza, entre outros, os métodos $\gg$ e $\gg=$ para sequenciar computações e **return** para devolver resultados computados. A classe ErrorMonad, inclui o método throwError que permite terminar computações devolvendo erros (que neste caso serão estruturais ou de ocorrência). Em seguida são apresentadas as assinaturas dos métodos referidos, quando instanciados para o **Monad** Uni.

```
instance Monad Uni where
  (>>=)  :: Uni a -> (a -> Uni b) -> Uni b;
  (>>)   :: Uni a -> Uni b -> Uni b
  return :: a -> Uni a

instance MonadError Uni where
  throwError :: UniError -> Uni a
```

Adicionalmente poderão ser usadas em computações de unificações todas as funções exportadas pelas bibliotecas *Control.Monad* e *Control.Monad.Error*.

Consideremos, por exemplo, as funções de verificação estrutural e de ocorrência. Foram implementadas duas versões para cada uma dessas funções, uma onde é devolvido um *booleano*, indicando se foi ou não encontrado um erro, e outra que é uma computação de unificação. Neste segundo caso o cálculo não devolve nenhum valor, limita-se a terminar com sucesso ou com o erro apropriado.

```
patternCheck  :: Term ->Term->Bool
patternCheckU :: Term ->Term->Uni()
occursCheck   :: VarId->Term->Bool
occursCheckU  :: VarId->Term->Uni()
```

Concentremo-nos agora na implementação da verificação de ocorrência. Como já sabemos, $x \in y$ se $x = y$, $x \in a_{\mathcal{R}}$ se $x \in \mathcal{R}$ e $x \in f(t_1, \ldots, t_n)$ se $x \in t_1$ ou ... ou $x \in t_n$. A versão monádica efectua esse mesmo teste e, se o resultado for verdadeiro, termina a computação com um erro de ocorrência.

```
occursCheck                              :: VarId→Term→Bool
occursCheck xid (Var{varId=yid})     = xid == yid
occursCheck xid (Par{parDeps=deps}) = xid `elementOf` deps
occursCheck xid (Fun{funTerms=ts})   = any (occursCheck xid) ts

occursCheckU        :: VarId→Term→Uni()
occursCheckU xid t = if occursCheck xid t then throwError Occurs else return()
```

A verificação de ocorrência é efectuada na fase de unificação de uma variável com outro termo, ou seja antes de actualizar a substituição com uma nova associação. Se terminar com sucesso, a nova substituição é devolvida, senão também este procedimento é cancelado.

```
unifyVarId :: Subst→VarId→Term→Uni Subst
unifyVarId sigma xid (Var{varId=yid})| yid==xid=return sigma
unifyVarId sigma xid t = occursCheckU  xid t  ≫return (addBind sigma xid t)
```

O processamento de cada par de nós dos termos considerados consiste em, caso pelo menos um dos termos seja uma variável, prosseguir com a tentativa de actualização da substituição usando o procedimento anterior, caso contrário consiste numa verificação estrutural e eventual devolução da substituição inalterada.

```
processNode :: Term→Term→Subst→Uni Subst
processNode (Var{varId=xid}) t sigma = unifyVarId sigma xid t
processNode t (Var{varId=xid}) sigma = unifyVarId sigma xid t
processNode s t                sigma = patternCheckU s t ≫ return sigma
```

Analisemos finalmente o procedimento principal do processo de unificação. Como se pode ver é composto por uma sequência de duas computações. A primeira (processNode), já descrita, diz respeito a todas as operações efectuadas sobre os nós (as raízes dos termos) que estamos a considerar, pode incluir verificação estrutural, verificação de ocorrência e cálculo de uma nova substituição. A segunda computação recebe como argumento a substituição actualizada pela primeira (caso esta termine com sucesso) e continua o processo de unificação, sobre os subtermos dos termos dados.

```
unifyTerms            :: Term→Term→Subst→Uni Subst
unifyTerms s t sigma = processNode s' t' sigma
                    ≫= unifyListTerms (subData news) (subData newt)
    where
    s'    = applySubst sigma s
    t'    = applySubst sigma t
    news = whenVar s (const s')
    newt = whenVar t (const t')
```

Menos óbvia é a forma como as substituições são, ou não, aplicadas aos termos s e t que tentamos unificar. Apesar de poder não parecer o caso, está a ser seguido o critério descrito no algoritmo 3.2 (*Robinson melhorado*). Os termos s' e t' são $s\sigma$ e $t\sigma$, respectivamente, e os termos

news e newt resultam da aplicação condicional de σ a s e t (isto é, apenas no caso em que são variáveis). O passo recursivo é efectuado sobre s e t condicionalmente expandidos por aplicação de σ, e portanto coincide com o algoritmo melhorado. O processamento dos nós recebe sempre como argumento $s\sigma$ e $t\sigma$, e de facto nem sempre é necessário o cálculo da aplicação destas substituições. Analisemos melhor esta situação. Se pelo menos um de entre $s\sigma$ e $t\sigma$ for uma variável, então de acordo com o algoritmo 3.2 ambas as substituições foram de facto necessárias, pois mesmo que um dos termos não fosse inicialmente variável teria de ser substituído por causa da verificação de ocorrência. Se $s\sigma$ e $t\sigma$ não forem ambos variáveis então o cálculo da aplicação de σ pode ser desnecessário. Convém então lembrar que se está a trabalhar com avaliação *lazy*, o que significa que o resultado dessa aplicação será avaliado apenas quando necessário. Como neste caso o processamento dos nós consistirá apenas numa verificação estrutural entre funções ou parâmetros, que tem em conta apenas a raiz destes termos, a aplicação da substituição na prática não é calculada. Note-se também que, pelo mesmo motivo, na verificação de ocorrência $x \in \sigma t$ referida no algoritmo melhorado, a aplicação de σ a t vai sendo calculada apenas à medida que o termo é percorrido, o que evita cálculos desnecessários.

Resta então examinar como é efectuada a unificação da lista de subtermos. A função unifyListTerms combina as duas listas de subtermos, componente a componente, com a aplicação parcial da função UnifyTerms, do que resulta uma lista de funções mónadicas, [Subst → Uni Subst], que dada uma substituição computam a sua actualização (através da unificação dos subtermos correspondentes). A função composeM compõe estas funções monádicas.

```
unifyListTerms :: [Term] → [Term] → Subst → Uni Subst
unifyListTerms ss ts = composeM (zipWith unifyTerms ss ts)
```

Ou seja, o resultado será a seguinte sequência de computações[1], onde s_i e t_i são os i-ésimos subtermos considerados,

$$\mathit{unifyTerms}\ s_1\ t_1\ \sigma \gg\!= \mathit{unifyTerms}\ s_2\ t_2 \gg\!= \ldots \gg\!= \mathit{unifyTerms}\ s_n\ t_n.$$

O processo de unificação termina com um erro, ou então com sucesso, devolvendo um unificador, quando não houver mais subtermos para unificar (caso em que pelo menos uma das listas de subtermos será vazia).

Verificação Prévia de Compatibilidade

Uma variante comum deste processo de unificação consiste em efectuar em primeiro lugar uma verificação de compatibilidade entre os dois termos. Como não envolve a aplicação de substituições nem a verificação de ocorrência, é um processo bastante leve, e a sua aplicação prévia pode compensar, principalmente se grande parte dos termos forem incompatíveis (o que, de um modo geral, é comum). A implementação da verificação de compatibilidade consiste apenas em associar nós correspondentes de ambos os termos com a operação de verificação estrutural, e

[1] Na verdade, por uma questão de eficiência, a composição monádica é implementada com associação à direita, e portanto a expressão exacta é: $(\mathit{unifyTerms}\ s_1\ t_1 \gg\!= (\lambda\theta.\ \mathit{unifyTerms}\ s_2\ t_2 \gg\!= (\ \ldots \gg\!= (\lambda\theta.\ \mathit{unifyTerms}\ s_n\ t_n)\ldots)))\ \sigma$.

sequenciar estas computações. A associação dos termos nó a nó com uma dada função mónadica e sequenciação das computações resultantes é implementada pela função genérica zipWithM_Data.

```
checkCompat :: Term→Term→Uni()
checkCompat =  zipWithM_Data patternCheckU
```

A variante do processo de unificação com verificação de compatibilidade, resulta da sequenciação da verificação com a função de unificação original.

```
unifyTerms'              :: Term→Term→Subst→Uni Subst
unifyTerms' s t sigma = checkCompat s t ≫ unifyTerms' s t sigma
```

Nos casos em que os termos são compatíveis, este procedimento acaba por ser obviamente menos eficiente. Toda a verificação de compatibilidade prévia é trabalho a mais, e isto acontece porque não é recolhida qualquer informação sobre os termos durante a travessia efectuada. Podemos, durante essa travessia prévia, devolver os pares subtermos comparados em que, pelo menos um deles, é uma variável. Para o efeito, bastaria alterar as funções patternCheckU e checkCompat.

```
patternCheckU :: Term→Term→Uni (Maybe(VarId,Term))
checkCompat   :: Term→Term→Uni [(VarId,Term)]
```

Após uma verificação de compatibilidade com sucesso, bastaria unificar sequencialmente os pares de termos na lista devolvida.

3.1.2 Representação de Substituições

Vejamos agora como implementar substituições e operações entre substituições, particularmente a sua composição. Da definição 1.31 podemos deduzir uma regra simples para determinar a composição de substituições, quando representadas sob a forma de conjuntos de associações (definição 1.29). No desenvolvimento seguinte, apesar das operações serem efectuadas entre conjuntos, a igualdade é definida entre as substituições que estes representam.

$$
\begin{aligned}
& \sigma\tau \\
= \;& \{x \mapsto (x\sigma)\tau \,|\, x \in V\} \\
= \;& \{x \mapsto (x\sigma)\tau \,|\, x \in Dom(\sigma) \cup Dom(\tau)\} \\
= \;& \{x \mapsto (x\sigma)\tau \,|\, x \in Dom(\sigma)\} \cup \{x \mapsto x\tau \,|\, x \in Dom(\tau) \backslash Dom(\sigma)\} \\
= \;& \{x \mapsto t\tau \mid (x \mapsto t) \in \sigma\} \cup \tau_{\restriction Dom(\tau)\backslash Dom(\sigma)} \\
= \;& \{x \mapsto t\tau \mid (x \mapsto t) \in \sigma,\ x \neq t\tau\} \cup \tau_{\restriction Dom(\tau)\backslash Dom(\sigma)}
\end{aligned}
$$

O segundo passo, restringe a definição de composição às variáveis pertencentes aos domínios de σ ou τ, pois para todas as restantes o mapeamento continuará a ser a identidade. Para variáveis x não pertencentes ao domínio de σ basta calcular $x\tau$ em vez de $x\sigma\tau$. O penúltimo passo

observa que os conjuntos da expressão anterior podem ser obtidos por aplicação de τ aos termos no $Cod(\sigma)$, e por restrição de τ a variáveis não pertencentes ao $Dom(\sigma)$. Por fim nota-se que podem ser eliminadas associações redundantes do primeiro conjunto. Resulta então o seguinte algoritmo.

Algoritmo 3.3 *Composição de substituições*

Compor ($\sigma = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$, $\tau = \{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}$)
  $\theta \leftarrow \varepsilon$
  `Para` $i \leftarrow 1$ `Até` m `Faz`
    $s'_i \leftarrow s_i\tau$
    `Se` $x_i \neq s'_i$ `Então` $\theta \leftarrow \theta \cup \{x_i \mapsto s'_i\}$
  `Para` $i \leftarrow 1$ `Até` n `Faz`
    `Se` $y_i \notin Dom(\sigma)$ `Então` $\theta \leftarrow \theta \cup \{y_i \mapsto t_i\}$
  `Devolve` θ
FimCompor

No contexto particular de unificação de *Robinson*, o processo de composição de substituições pode ser um pouco optimizado. Durante o cálculo do unificador $\sigma = \sigma_1 \ldots \sigma_n$, com $\sigma_i = \{x_i \mapsto s_i\}$, de cada vez que é determinada uma associação $x_i \mapsto s_i$, a variável x_i passará a ser substituída em eventuais ocorrências posteriores. Isso significa que para qualquer outra associação $x_j \mapsto s_j$, com $j > i$, se terá $x_j \neq x_i$ (e portanto $x_j \notin Dom(\sigma_1 \ldots \sigma_{j-1})$) e $x_i \notin s_j$ (logo $x_i \neq s'_i = s_i x_j \mapsto s_j$). As duas verificações do algoritmo de composição podem então ser ignoradas.

Algoritmo 3.4 *Composição de substituições (optimizada para algoritmo de Robinson)*

Compor ($\sigma = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$, $\tau = \{y \mapsto t\}$)
  $\theta \leftarrow \varepsilon$
  `Para` $i \leftarrow 1$ `Até` m `Faz`
    $\theta \leftarrow \theta \cup \{x_i \mapsto s_i\tau\}$
  $\theta \leftarrow \theta \cup \{y \mapsto t\}$
  `Devolve` θ
FimCompor

Tratam-se de optimizações pequenas, em termos de tempo, pois o passo mais pesado é a aplicação de uma das substituições a todos os termos da outra. Em termos de espaço podem

ocorrer alguns problemas. Considere-se $\sigma = \sigma_1 \dots \sigma_n$, com $\sigma_i = \{x_i \mapsto f(x_{i+1}, x_{i+1})\}$.

$$\begin{aligned}
& \sigma_1\sigma_2\sigma_3 \dots \\
= \; & \{x_0 \mapsto f(x_1, x_1)\}\ \{x_1 \mapsto f(x_2, x_2)\}\ \{x_2 \mapsto f(x_3, x_3)\}\ \dots \\
= \; & \{x_0 \mapsto f(f(x_2, x_2), f(x_2, x_2)),\ x_1 \mapsto f(x_2, x_2)\}\ \{x_2 \mapsto f(x_3, x_3)\} \dots \\
= \; & \{x_0 \mapsto f(f(f(x_3, x_3), f(x_3, x_3)), f(f(x_3, x_3), f(x_3, x_3))) \\
& x_1 \mapsto f(f(x_3, x_3), f(x_3, x_3)),\ x_2 \mapsto f(x_3, x_3)\} \dots
\end{aligned}$$

Como se pode ver, os termos podem crescer exponencialmente. Uma forma comum de parcialmente contornar o problema é evitar calcular a composição das substituições elementares explicitamente, e em vez disso manter uma lista com as várias associações determinadas, $[x_1 \mapsto s_1, \dots, x_n \mapsto s_n]$. A esta representação dá-se o nome de *forma triangular*. Para o caso anterior, $\sigma_1 \dots \sigma_n$ seria simplesmente $[x_1 \mapsto f(x_1, x_1), \dots, x_n \mapsto f(x_n, x_n)]$.

Numa linguagem *lazy* e com partilha de valores, como *Haskell*, as coisas não se passam de forma assim tão simples. Consideremos a composição de substituições unitárias $\sigma = \sigma_1 \dots \sigma_n$, com $\sigma_i = \{x_i \mapsto s_i\}$, e representação $\sigma = \{x_1 \mapsto s'_1, \dots, x_n \mapsto s'_n\}$. A substituição σ será avaliada quando, e apenas à medida que for sendo necessária. Até lá estará guardada em memória como sendo a composição $\sigma = \{x_1 \mapsto s_1\} \dots \{x_n \mapsto s_n\}$, ou seja, algo muito semelhante à *forma triangular*. Cada termo s'_i do codomínio de σ será calculado quando for necessário avaliar uma aplicação $x_i\sigma$ (durante um processo de unificação, por exemplo), e do algoritmo 3.4 sabemos que irá resultar $s'_i = s_i\sigma_{i+1} \dots \sigma_n$. Ou seja, avaliar $x_i\sigma$ consistirá em avaliar a aplicação sequencial das substituições unitárias $x_i\{x_i \mapsto s_i\}\{x_{i+1} \mapsto s_{i+1}\} \dots \{x_n \mapsto s_n\}$, mais uma vez semelhante à utilização de uma *forma triangular*. Quanto ao espaço utilizado, $x_i\sigma$ e s'_i irão partilhar a mesma representação. Mais que isso, múltiplas ocorrências de $x_i\sigma$ partilharão todas o mesmo valor s'_i que terá de ser avaliado uma única vez. Utilização de uma forma triangular para calcular $x_i\sigma$ resultaria num termo que ocupa o mesmo espaço, termo esse que não seria guardado na representação de σ, e que portanto teria que ser recalculado para novas aplicações de $x_i\sigma$, resultando em mais cálculos e espaço ocupado.

Vistas as coisas desta forma, pode parecer não haver motivo para usar a *forma triangular*, de facto existem algumas desvantagens em o fazer. No entanto, a excessiva partilha de termos resultante da representação por conjuntos pode ser problemática. A partir do momento em que uma ocorrência $x_i\sigma = s'_i$ for avaliada, persistirá em memória enquanto houver pelo menos uma referência à substituição σ ou a quaisquer outras ocorrências de $x_i\sigma$ (mesmo que nunca necessitem de ser avaliadas). Perante este tipo de cenário, a forma como a memória é gerida torna-se menos previsível, e o aparecimento de *fugas de espaço* mais provável. Optou-se então por uma implementação baseada na *forma triangular* descrita, com a ressalva de que, dependendo do contexto em que for usada, uma implementação baseada em conjuntos poderá ser mais vantajosa.

Implementação em Haskell: A aplicação de uma substituição $\sigma = [x_1 \mapsto s_1, \dots, x_n \mapsto s_n]$ a um termo t consiste na aplicação sequencial de cada uma das substituições unitárias que o

compõem.

$$t\sigma = \begin{cases} t & \text{se } \sigma = [\,] \\ (x\{x_1 \mapsto s_1\})[x_2 \mapsto s_2, \ldots, x_n \mapsto s_n] & \text{se } t = x \in V \text{ e } \sigma = [x_1 \mapsto s_1, \ldots, x_n \mapsto s_n] \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{se } t = f(t_1, \ldots, t_n) \end{cases}$$

O caso do meio, aplicação da substituição a uma variável, pode ser visto como uma pesquisa linear da variável x no domínio da lista de associações. Começando à cabeça vão sendo removidos elementos até encontrar uma associação $x \mapsto t$, que é de seguida aplicada. Composição de novas substituições unitárias é efectuada adicionando a respectiva associação no final da lista, portanto esta estrutura funciona como uma fila (*fifo*). Implementações funcionais puras de filas não são muito eficientes. As estruturas mais comuns apresentam complexidade linear (pior caso) para inserções [30], outras mais elaboradas apresentam complexidade logarítmica [27]. Uma alternativa à utilização de uma fila, é usar uma estrutura que permita acesso rápido a associações usando variáveis como chave, e ignorar a remoção das associações introduzidas antes da pesquisada. Estas remoções podem não ser efectuadas pois sabemos que para as substituições criadas $x_i \notin s_j$ para $i < j$. Como contrapartida, o tamanho da substituição não vai diminuindo à medida que for sendo aplicada. O primeiro caso base pode ser verificado, por exemplo, se a estrutura mantiver a informação de qual o último elemento introduzido (a substituição fica vazia quando a última substituição unitária é aplicada)[2].

Optou-se então por implementar a *representação triangular* de substituições através de um *mapeamento finito* (Finite Map), baseado em árvores balanceadas [1], e portanto com acessos logarítmicos e inserções logarítmicas. As primitivas de construção e acesso a substituições são directamente baseadas nas correspondentes para *mapeamentos finitos*. Para construção temos emptySubst para a substituição identidade, unitSubst para substituições unitárias, addBind para adicionar uma associação, addBindsList para acrescentar uma lista de associações, e addBinds_EC que adiciona uma lista de associações de variáveis que apontam para um mesmo termo[3].

```
emptySubst   :: Subst
addBind      :: Subst → VarId → Term → Subst
addBindList  :: Subst → [(VarId, Term)] → Subst
addBinds_EC  :: Subst → [VarId] → Term → Subst
```

As Primitivas de acesso são boundTo que devolve o termo a que uma variável está associada (ou a indicação de que não há associação, se for o caso), e isBoundTo que verifica se existe associação.

```
boundTo    :: Subst → VarId → Maybe Term
isBoundTo  :: Subst → VarId → Bool
```

Uma substituição pode conter cadeias de associações entre variáveis, ou seja, podemos ter $x_{k_1} \mapsto x_{k_2}, x_{k_1} \mapsto x_{k_2}, \ldots, x_{k_{r-1}} \mapsto x_{k_r}, x_{k_r} \mapsto t$. Nestes casos pode dar jeito saber, caso

[2] A verificação deste caso base é necessária apenas por questões de eficiência, pois evita que se percorra o termo todo, e ainda não foi implementada.

[3] O EC em BindsList_EC significa classe de equivalência.

exista, qual o termo não variável, que termina a cadeia (denominado *termo indirectamente apontado* pelas variáveis da cadeia), e qual a última variável da cadeia (denominada *representante* das variáveis dessa cadeia). A função auxiliar chase, devolve o representante de, e o termo indirectamente apontado por, uma variável (se existir). A função selectBindTarget permite escolher o termo final (variável ou não) de uma cadeia de associações a partir de um par construído pela função anterior.

```
chase            :: Subst → VarId → (VarId, Maybe Term)
selectBindTarget :: (VarId, Maybe Term) → Term
```

Podemos então definir funções representId e chaseVarId que devolvem respectivamente o representante e o termo final da cadeia de uma variável, e chaseTerm, a extensão de chaseVarId para termos, que para não variáveis é simplemente a identidade.

```
representId        :: Subst → VarId → VarId
representId sigma = fst ∘ chase sigma

chaseVarId         :: Subst → VarId → Term
chaseVarId sigma = selectBindTarget ∘ chase sigma

chaseTerm          :: Subst → Term → Term
chaseTerm sigma t = whenVar t (chaseVarId sigma)
```

Substituições poderão ser aplicadas a termos, fórmulas, e outras estruturas. Definiu-se portanto uma classe Substitute de tipos que podem ser substituídos.

```
class Substitute a where
    applySubst :: Subst → a → a
```

Substituir um termo consiste em percorrer o mesmo da raiz para as folhas, substituindo variáveis e, caso o resultado seja uma função, substituindo também os seus subtermos, ou seja, em efectuar um mapeamento descendente com a função chaseTerm. Substituir uma fórmula consiste em substituir todos os termos dessa mesma fórmula.

```
instance Substitute Term where
   applySubst sigma = topDownMapData (chaseTerm sigma)

instance Substitute Form where
   applySubst sigma =  mapFormTerms (applySubst sigma))
```

Para efeitos de verificação estão implementadas funções que permitem testar se uma substituição é um unificador. Apesar de neste momento ainda só ser necessária a sua instância para *Termos*, a implementação isUnifierOf contempla qualquer tipo da classe Substitute para o qual esteja definido o operador igualdade, e a sua variante isUnifierOfEq qualquer tipo da classe Substitute desde que fornecida uma função de igualdade.

```
isUnifierOf   :: (Substitute a, Eq a) ⇒ Subst→a→a→ Bool
isUnifierOfEq :: Substitute a         ⇒ Subst→(a→a→Bool)→a→a→Bool
```

Outras funções: Caso haja uma forte probabilidade de ocorrência de grandes cadeias de variáveis, pode ser útil eliminá-las, colocando todas as variáveis a apontar para o termo final da sua cadeia. Para esse efeito é disponibilizada a função compactSubst.

```
compactSubst        :: Subst → Subst
compactSubst sigma = mapFM (\x t→chaseVarId sigma x) sigma
```

Outra possibilidade consiste em fundir o processo de eliminação de cadeias, com as funções de acesso que por definição já percorrem as cadeias de variáveis. As variantes das funções chase, chaseVarId, representId e chaseTerm têm assinaturas e definições idênticas às originais, mas devolvem uma nova substituição, equivalente à inicial, mas com a cadeia percorrida eliminada.

```
chase_S       :: Subst → VarId → (Subst,(VarId,Maybe Term))
chaseVarId_S  :: Subst → VarId → (Subst,Term)
representId_S :: Subst → VarId → (Subst,VarId)
chaseTerm_S   :: Subst → Term  → (Subst,Term)
```

Para além das funções aqui referidas, estão ainda implementadas e são exportadas algumas funções estruturais para o tipo de dados *Substituição*, como por exemplo, acesso a domínio, codomínio e tamanho de substituição, entre outras.

3.2 Algoritmo de Martelli e Montanari

O processo de unificação pode ser descrito com base num sistema (conjunto) de equações entre termos, e regras que permitem resolver esse mesmo sistema.

$$\{t \stackrel{?}{=} x\} \cup P; \sigma \longrightarrow \{x \stackrel{?}{=} t\} \cup P; \sigma \tag{3.1}$$

$$\{x \stackrel{?}{=} x\} \cup P; \sigma \longrightarrow P; \sigma \tag{3.2}$$

$$\{f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n)\} \cup P; \sigma \longrightarrow \{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\} \cup P; \sigma \tag{3.3}$$

$$\{f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n)\} \cup P; \sigma \stackrel{f \neq g}{\longrightarrow} \textit{erro estrutural} \tag{3.4}$$

$$\{x \stackrel{?}{=} t\} \cup P; \sigma \stackrel{x \in t}{\longrightarrow} \textit{erro de ocorrência} \tag{3.5}$$

$$\{x \stackrel{?}{=} t\} \cup P; \sigma \stackrel{x \notin t}{\longrightarrow} P\{x \mapsto t\}; \sigma\{x \mapsto t\} \tag{3.6}$$

Nas regras anteriores o símbolo $\longrightarrow$ denota uma transição, eventualmente condicionada à expressão que lhe é associada. Os membros de cada transição são da forma $\langle$*sistema de equações* , *substituição calculada*$\rangle$. A *regra da orientação*, 3.1, indica apenas que podemos sem perda de generalidade considerar que variáveis aparecem do lado esquerdo das equações, e a *regra*

trivial, 3.2, que podemos eliminar equações triviais. A regra 3.3, da *decomposição*, consiste no passo estruturalmente indutivo da unificação. As regras, 3.4 e 3.5, são as regras de *verificações estrutural* e de *ocorrência*. Finalmente, a regra 3.6, dita de *eliminação de variáveis*, consiste na remoção de uma equação da forma $x = t$, na criação de uma nova associação $x \mapsto t$, e actualização das restantes equações, bem como da substituição já calculada, por aplicação com, e composição com, $\{x \mapsto t\}$, respectivamente. Unificar dois termos t_1, t_2 consiste em resolver a equação $t_1 = t_2; \varepsilon$.

O algoritmo descrito por Martelli e Montanari [23], tem em vista a optimização do processo de resolução destes sistemas através de representações eficientes das equações que o compõem *multiequações* e dos termos de cada multiequação *multitermos*, optimização das regras, e uma escolha inteligente das regras a aplicar que permite evitar alguns passos normalmente efectuados em algoritmos de unificação, como verificação de ocorrência e instanciação de subtermos. Iremos de seguida descrever estas optimizações.

Multiequações

Equações com termos comuns podem ser reformuladas como apenas uma *multiequação*. Por exemplo, o sistema de equações,

$$\begin{cases} x_1 \stackrel{?}{=} x_2 \\ x_1 \stackrel{?}{=} f(x_3, h(g(x_5))) \\ x_2 \stackrel{?}{=} f(x_4, h(x_3)) \\ x_2 \stackrel{?}{=} f(g(b), h(x_3)) \\ x_3 \stackrel{?}{=} x_4 \\ x_4 \stackrel{?}{=} g(b) \\ x_6 \stackrel{?}{=} f(f(x_1, x_2), f(x_1, x_2)) \end{cases}$$

é equivalente ao seguinte sistema de multiequações,

$$\begin{cases} x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(g(x_5))) \stackrel{?}{=} f(x_4, h(x_3)) \stackrel{?}{=} f(g(b), h(x_3)) \\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \\ x_6 \stackrel{?}{=} f(f(x_1, x_2), f(x_1, x_2)) \end{cases}.$$

Multitermos

A representação das multiequações pode ser optimizada, compactando a representação de todos os termos não variáveis numa só estrutura. Assumindo que todos os termos não variáveis são estruturalmente compatíveis, o símbolo na raiz da estrutura que os representa é comum, e portanto pode ser partilhado. Queremos verificar se os seus subtermos são unificáveis, portanto estes podem ser representados através de multiequações. Ou seja, podemos representar a primeira equação do sistema anterior, na forma,

$$x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b)\ ,\ h(g(x_5)) \stackrel{?}{=} h(x_3)\ ).$$

Aplicando este método recursivamente podemos ainda simplificar $h(g(x_5)) \stackrel{?}{=} h(x_3)$, como $h(\ g(x_5) \stackrel{?}{=} x_3\ )$,

$$x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b)\ ,\ h(g(x_5) \stackrel{?}{=} x_3)\ ).$$

Estas multiequações são normalmente representadas como um conjunto $\langle X, T\rangle$, onde X é um conjunto de variáveis e T um multitermo. Porque uma multiequação pode não ter termos não variáveis, é necessário considerar a existência de um multitermo vazio, que será denotado por $\emptyset$ (terá portanto a mesma representação que um conjunto vazio de variáveis). Nesta notação a multiequação anterior escreve-se como (ver figura 3.3, para apresentação em forma de árvore),

$$\langle\{x_1, x_2\}\ ,\ f(\ \langle\{x_3, x_4\}, g(\langle\emptyset, b\rangle)\rangle\ ,\ \langle\emptyset, h(\langle\{x_3\}, g(\langle\{x_5\}, \emptyset\rangle)\rangle)\rangle\ )\rangle.$$

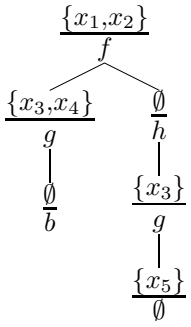

Figura 3.3: Representação em árvore de uma multiequação baseada em multitermos.

Obtemos assim uma representação mutuamente recursiva de multiequações e multitermos, compacta. Equações em que haja termos incompatíveis não podem ser transformadas em multiequações baseadas em multitermos. Nesse caso o algoritmo deverá falhar com um erro estrutural.

Decomposição de Multiequações e Multitermos

A decomposição de termos não variáveis (regra 3.3) pode ser melhorada, actuando em maior profundidade. Para isso necessitamos de definir dois conceitos auxiliares, *parte comum* e *fronteira* de termos. Para apenas um par de termos, temos:

$$\text{comum}(s,t) = \begin{cases} x & se \quad s = x \in V \\ x & se \quad t = x \in V \\ f(c_1, \ldots, c_n) & se \quad s = f(s_1, \ldots, s_n),\ t = f(t_1, \ldots, t_n),\ \text{e} \\ & \quad\ c_i = \text{comum}(s_i, t_i), i = 1, \ldots, n \end{cases}$$

$$\text{fronteira}(s,t) = \begin{cases} \{x \stackrel{?}{=} t\} & se \quad s = x \in V \\ \{x \stackrel{?}{=} s\} & se \quad t = x \in V \\ \bigcup_{i=1}^{n} \text{fronteira}(s_i, t_i) & se \quad s = f(s_1, \ldots, s_n) \text{ e } t = f(t_1, \ldots, t_n) \end{cases}.$$

Ou seja, intuitivamente, dados dois termos compatíveis, a sua parte comum é dada pela porção das suas estruturas que, quando sobrepostas, são comums, e a sua fronteira pelas porções que não o são, associadas aos nós terminais (variáveis). Estes conceitos são trivialmente generalizáveis para um número arbitrário de termos.

O cálculo da fronteira permite uma decomposição rápida de termos não variáveis. No caso da primeira multiequação do sistema considerado, a fronteira de $f(x_3, h(g(x_5)))$, $f(x_4, h(x_3))$, $f(g(b), h(x_3))$ é dada por $\{\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b)\ ,\ g(x_5) \stackrel{?}{=} x_3 \stackrel{?}{=} x_4\}$. Numa multiequação podemos ter simultaneamente termos não variáveis que queiramos decompor, e variáveis. Estas últimas poderão formar uma nova multiequação com a parte comum dos termos decompostos. Neste caso a parte comum dos termos considerado seria dada por, $f(x_3, h(x_3))$. Resumindo a multiequação, $x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(g(x_5))) \stackrel{?}{=} f(x_4, h(x_3)) \stackrel{?}{=} f(g(b), h(x_3))$ pode ser decomposta nas multiequações,

$$x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(x_3))$$
$$x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b)$$
$$x_3 \stackrel{?}{=} g(x_5)$$

A adaptação para multiequações com multitermos é simples. Neste caso, em vez de percorrermos várias estruturas (termos) simultaneamente, percorremos apenas uma (multiequação ou multitermo). As fronteiras são dadas pelas submultiequações com um conjunto de variáveis não vazio, e a parte comum é obtida substituindo, em submultiequações com parte variável não vazia, o multitermo por um vazio e o conjunto de variáveis por um outro singular, com apenas uma das variáveis do original . Repare-se que, como o multitermo da multiequação para a parte comum tem em cada posição apenas uma variável ou nenhuma variável e um multitermo, ele representa apenas um termo (não variável), sem qualquer ambiguidade. Os algoritmos 3.5 e 3.6 calculam a fronteira e a parte comum de multiequações com multitermos, respectivamente. A figura 3.4 apresenta a decomposição anterior quando utilizados multitermos, com representação em árvore.

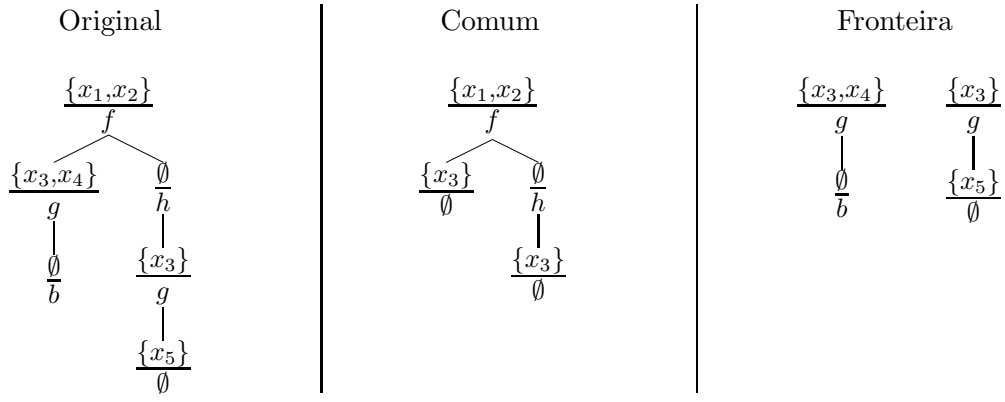

Figura 3.4: Decomposição de uma multiequação em parte comum e fronteira.

Compactação de multiequações e fusão de multitermos

Através da decomposição da primeira equação do sistema que considerámos originalmente,

$$\begin{cases} x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(g(x_5))) \stackrel{?}{=} f(x_4, h(x_3)) \stackrel{?}{=} f(g(b), h(x_3)) \\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) x_6 \stackrel{?}{=} f(f(x_1, x_2), f(x_1, x_2)) \end{cases}$$

Algoritmo 3.5 *Cálculo da fronteira de multiequações e multitermos*

FronteiraMultiEq $(\langle X, T\rangle)$

Se $X \neq \emptyset$
  Então Devolve $\{\langle X, T\rangle\}$
  Senão Devolve **FronteiraMultiTermo**(T)
FimFronteiraMultiEq

FronteiraMultiTermo(T)

Caso T Seja
 $\emptyset$: Devolve $\emptyset$
 $f(E_1, \ldots, E_n)$: Devolve $\bigcup_{i=1}^{n}$**FronteiraMultiEq**(E_i)
FimCaso
FimFronteiraMultiTermo

Algoritmo 3.6 *Cálculo da parte comum de multiequações e multitermos*

ComumMultiEq$(\langle X, T\rangle)$

Se $X \neq \emptyset$
  Então Devolve $\langle\{x\}, T\rangle$, com $x \in X$
  Senão Devolve $\langle\emptyset,$ **ComumMultiTermo**$(T)\rangle$
FimComumMultiEq

ComumMultiTermo(T)

Caso T Seja
 $\emptyset$: Devolve $\emptyset$
 $f(E_1, \ldots, E_n)$: Devolve $f(C_1, \ldots, C_n)$, com $C_i =$ **ComumMultiEq**(E_i)
FimCaso
FimComumMultiTermo

Obteríamos um novo sistema,

$$\begin{cases} x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(x_3)) \\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \\ x_3 \stackrel{?}{=} g(x_5) \\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \\ x_6 \stackrel{?}{=} f(f(x_1, x_2), f(x_1, x_2)) \end{cases}$$

Após este passo, faz todo o sentido reagrupar multiequações com termos em comum. Da aplicação dessa transformação, denominada *compactação do sistema*, resultaria,

$$\begin{cases} x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(x_3)) \\ x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \stackrel{?}{=} g(x_5) \\ x_6 \stackrel{?}{=} f(f(x_1, x_2), f(x_1, x_2)) \end{cases}$$

Esse reagrupamento significa que vamos juntar na mesma multiequação termos não variáveis que estavam até então em multiequações distintas. Isso significa que, no caso de uma representação com multitermos, os teremos que *fundir* num só. Se não for possível a fusão, estamos perante um erro estrutural e o sistema (a unificação) não tem solução. O processo de fusão é dado pelo algoritmo 3.7. A figura 3.5 apresenta a fusão anterior com multitermos.

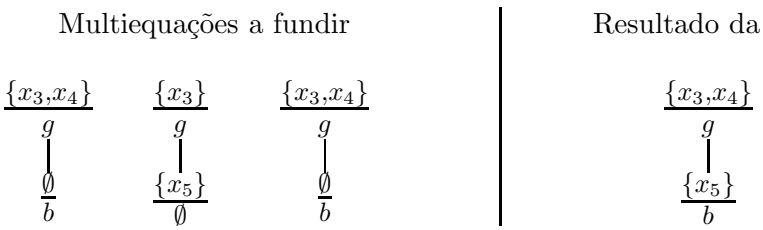

Figura 3.5: Fusão de multiequações e multitermos.

Algoritmo 3.7 *Fusão de multiequações e multitermos*

FusãoMultiEq$(\langle X, S\rangle, \langle Y, T\rangle)$

`Devolve` $\langle X \cup Y, \textbf{FusãoMultiTermo}(\mathrm{S,T})\rangle$

FimFusãoMultiEq

FusãoMultiTermo(S, T)

`Caso` (S, T) `Seja`

$(\emptyset, T)$	: `Devolve` T
$(S, \emptyset)$	: `Devolve` S
$(f(D_1, \ldots, D_n), f(E_1, \ldots, E_n))$	: `Devolve` $f(F_1, \ldots, F_n)$, com $F_i = \textbf{FusãoMultiEq}(D_i, E_i)$
`Senão`	: *erro estrutural*

`FimCaso`

FimFusãoMultiTermo

Construção da Substituição

De acordo com a regra 3.6, da eliminação de variáveis, podemos construir uma nova associação para a substituição a ser calculada, sempre que tivermos uma equação da forma $x \stackrel{?}{=} t$, tal que $x \notin V(t)$, o que pode ser obviamente generalizado para multiequações da forma $x_1 \stackrel{?}{=} \ldots \stackrel{?}{=} x_n \stackrel{?}{=} t$, mas não para multiequações com mais do que um termo não variável, pois não podemos criar mais do que uma associação por variável. No caso em que temos mais do que um termo não variável podemos, como já vimos, obter por decomposição uma nova multiequação (a parte comum) com apenas um termo não variável. No sistema que estamos a considerar, após a decomposição, estamos em condições de criar as novas associações, $x_1 \mapsto f(x_3, h(x_3))$ e $x_2 \mapsto f(x_3, h(x_3))$, pois nem x_1 nem x_2 pertencem ao termo em causa. O passo seguinte seria remover esta equação do sistema e aplicar às restantes as substituições unitárias calculadas. O resultado seria,

$$\begin{cases} x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \stackrel{?}{=} g(x_5) \\ x_6 \stackrel{?}{=} f(f(f(x_3, h(x_3)), f(x_3, h(x_3))), f(f(x_3, h(x_3)), f(x_3, h(x_3)))) \end{cases}$$

A última multiequação ficou bastante mais complexa. Esse problema já foi referido aquando da implementação do algoritmo de Robinson. O ideal seria que as variáveis das multiequações às quais aplicamos a regra da eliminação não ocorressem em nenhum dos termos não variáveis das equações restantes, desta forma não seria efectuada nenhuma substituição. De facto, se o sistema tiver solução, existe (ou é possível obter por decomposição de outras) uma multiequação nessas condições, pois quando tal não acontece isso significa que existe uma dependência cíclica entre os termos do sistema, e portanto há um erro de ocorrência. Voltemos ao sistema no seu estado anterior. O número associado a cada multiequação indica o número total de ocorrências dos seus termos variáveis nas restantes multiequações.

$$\begin{cases} [4] & x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(x_3)) \\ [2] & x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \stackrel{?}{=} g(x_5) \\ [0] & x_6 \stackrel{?}{=} f(f(x_1, x_2), f(x_1, x_2)) \end{cases}$$

Neste caso deveria ser escolhida para eliminação a última equação, pois é a única com contagem de ocorrência de variáveis a 0. Dessa aplicação obter-se-iam a associação $x_6 \mapsto f(f(x_1, x_2), f(x_1, x_2))$, e o sistema,

$$\begin{cases} [0] & x_1 \stackrel{?}{=} x_2 \stackrel{?}{=} f(x_3, h(x_3)) \\ [2] & x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \stackrel{?}{=} g(x_5) \end{cases}$$

É importante reparar na alteração dos números de ocorrências associados ao sistema devido à eliminação efectuada. A próxima multiequação a ser eliminada seria a primeira, dando origem às associações $x_1 \mapsto f(x_3, h(x_3))$ e $x_2 \mapsto f(x_3, h(x_3))$, e ao sistema,

$$\begin{cases} [0] & x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(b) \stackrel{?}{=} g(x_5) \end{cases}$$

Como a única multiequação contém mais do que um termo não variável, é necessário aplicar

decomposição. O resultado é um novo sistema, composto pela parte comum e fronteira,

$$\begin{cases} [1] & x_3 \stackrel{?}{=} x_4 \stackrel{?}{=} g(x_5) \\ [0] & x_5 \stackrel{?}{=} b \end{cases}$$

A eliminação da segunda multiequação do sistema dá origem a $x_5 \mapsto b$, e por fim, da eliminação da única multiequação restante, resultam as associações $x_3 \mapsto g(x_5)$, e $x_4 \mapsto g(x_5)$. O sistema inicial está então resolvido com sucesso e o resultado é um unificador mais geral, definido pelo mapeamento,

$$\begin{aligned} &x_6 \mapsto f(f(x_1, x_2), f(x_1, x_2)) \\ &x_1 \mapsto f(x_3, h(x_3)) \\ &x_2 \mapsto f(x_3, h(x_3)) \\ &x_5 \mapsto b \\ &x_3 \mapsto g(x_5) \\ &x_4 \mapsto g(x_5) \end{aligned}$$

Resumo do Algoritmo de Martelli e Montanari

Resumindo, no algoritmo de Martelli e Montanari a unificação de dois termos s e t consiste na resolução de um sistema de multiequações, partindo de $s \stackrel{?}{=} t$. A representação das equações é efectuada através de multitermos, o que permite obter uma representação mais compacta do sistema, e tornar mais eficiente o processo de verificação estrutural, bem como a aplicação das regras de transformação do sistema. A decomposição das (multi)equações é efectuada em profundidade, através do cálculo de parte comum e fronteira. O sistema é mantido na sua forma compacta por reagrupamento de equações com termos (variáveis) comuns e fusão de multitermos após cada decomposição. A selecção de cada (multi)equação a decompor (caso necessário) e eliminar é efectuada por forma a que as variáveis da mesma não ocorram nas restantes do sistema, evitando assim a aplicação de substituições. A manutenção da contagem de ocorrências das variáveis de cada multiequação nos multitermos das restantes, permite também evitar a realização de testes de ocorrência explícitos.

3.2.1 Implementação do Algoritmo de Martelli e Montanari

A implementação dos tipos de dados *Multiequação* e *Multitermo* é tradução quase directa do que acabámos de descrever. Multiequações são um par com um conjunto de variáveis e um multitermo, e multitermos poderão ser multitermos vazios, funções com multiequações como argumentos, ou ainda parâmetros. Como estas estruturas são apenas usadas durante a unificação, os termos a considerar não contêm variáveis ligadas, pelo que estas não são consideradas.

```
data ME = ME{meL :: Set VarId, meR :: MT}

data MT = EmptyMT
        | Par{parId :: ParId, parDeps :: (Set VarId)}
        | Fun{funId :: FunId, funArity :: Arity, funMEs :: [ME]}
```

Estas estruturas são mutuamente recursivas, e portanto instâncias da classe MRecData descrita na secção 2.2. As multiequações têm um único subdado, o seu multitermo. É importante notar que, apesar de ser só um, esse subdado é devolvido numa lista (unitária), tal como especifica a assinatura da função subData. Multitermos por sua vez, têm como subdados a lista de multiequações que contêm.

```
instance SubData ME MT where
subData     (ME{meR=mt}) = [mt]
    updSubData (me@ME{}) [mt] = me{meR=mt}
    updSubData (me@ME{}) _    = error "must update ME with a single MT"

instance SubData MT ME where
    subData (Fun{funMEs=mes}) = mes
    subData _                 = []
    --
    updSubData (f@Fun{}) mes = f{funMEs=mes}
    updSubData mt        _   = mt

instance MRecData ME MT where
instance MRecData MT ME where
```

Decomposição de Multitermos e Multiequações

O cálculo da fronteira de um multitermo, pode ser expresso à custa de um paramorfismo. A operação ao nível do tipo multitermo limita-se a concatenar as fronteiras de cada um dos seus subdados. Note-se que tanto o multitermo vazio como os parâmetros não têm subdados, e portanto nesse caso o resultado é uma lista vazia. Ao nível do tipo multiequação, se esta não contiver variáveis, então o resultado é a fronteira do seu multitermo, senão é a lista singular com a própria multiequação. A parte comum é obtida por aplicação de um mapeamento que altera apenas as multiequações que não têm conjunto de variáveis vazio, removendo-lhes o seu multitermo e todas as variáveis excepto uma (função pickvar).

```
frontierMT :: MT → [ME]
frontierMT = paraMRData (\mt frs  → concat frs)
                        (\me [fr] → if isEmptyMEL me then fr else [me])

commonMT :: MT → MT
commonMT = botUpMapMRData id (\me→ if isEmptyMEL me then me else pickvar me)
```

O cálculo da fronteira e parte comum podem ser fundidos numa só função. Sabemos que a função botUpMapMRData é um caso particular de um paraMRData, para além disso dois paraMRData são facilmente fundidos num só, por emparelhamento das funções aplicadas a cada subdado. Combinar ambas as funções num só paraMRData pode ser feito directamente, de forma intuitiva, no entanto determinar primeiro a implementação alternativa de commonMT como um paraMRData simplifica o raciocínio e pode ser efectuado de forma quase mecânica.

```
        commonMT

(1) ≡ botUpMapMRData
          id
          (\me→if isEmptyMEL me then me else pickvar me)

(2) ≡ paraMRData
          (\a cs→id (updSubData a cs))
          (\b cs→(\me→if isEmptyMEL me then me else pickvar me) (updSubData b cs))

(3) ≡ paraMRData
          (\mt cs→id (updSubData mt cs))
          (\me cs→(if isEmptyMEL (updSubData me cs)
                    then (updSubData me cs) else pickvar (updSubData me cs)))

(4) ≡ paraMRData
          (\mt cs→updSubData mt cs)
          (\me cs→if isEmptyMEL me then (updSubData me cs) else pickvar me)

(5) ≡ paraMRData
          (\mt cs →updSubData mt cs)
          (\me [c]→if isEmptyMEL me then (updSubData me [c]) else pickvar me)
```

Os passos são bastante simples, partindo da definição de commonMT (1) efectuamos o *inlining* da função botUpMapMRData (2), no passo seguinte (3) aplicamos apenas uma redução-β (aplicação da função de mapeamento para multitermos ao seu argumento) e uma renomeação-α (alteração dos nomes das variáveis a e b para mt e me, respectivamente), por fim a expressão é simplificada (4) com base no facto de nem isEmptyMEL (verificação da existência ou não de variáveis numa multiequação) nem pickvar dependerem dos subtermos da multiequação em causa, e explicitamos o facto dos subdados de uma multiequação serem uma lista singular. A construção de uma função dec :: MT→([ME],ME) de decomposição tal que decMT mt =(frontierMT mt , commonMT mt) é agora bastante mais simples.

```
        decMT

(1) ≡ paraMRData
          (\mt (frs_cs)→let (frs,cs)=unzip frs_cs in (concat frs, updSubData mt cs))
          (\me (frs_cs)→let ([fr],[c])=unzip frs_cs in
                          (if isEmptyMEL me then fr else [me],
                           if isEmptyMEL me then (updSubData me [c]) else pickvar me))
(2) ≡ paraMRData
          (\mt (frs_cs)→let (frs,cs)=unzip frs_cs in (concat frs, updSubData mt cs))
          (\me (frs_cs)→let ([fr],[c])=unzip frs_cs in
                          if isEmptyMEL me then (fr,updSubData me [c])
                                           else ([me],pickvar me))
```

Neste caso combinamos multitermos e multiequações com listas de pares fronteira e parte comum. Começamos por (1) simplesmente combinar as funções de ambos os paraMRData, recuperamos a lista de fronteiras e a lista das partes comums da lista de pares fronteira e parte comum, e devolvemos um par com a nova fronteira e parte comum, calculada a partir das funções anteriores. No segundo passo (2), simplificamos a função que actua sobre as multiequações, combinando as duas verificações da existência ou não de variáveis na multiequação.

```
decMT = paraMRData
          (\mt (frs_cs)→let (frs,cs)=unzip frs_cs in
                        (concat frs, updSubData mt cs))

          (\me (frs_cs)→let ([fr],[c])=unzip frs_cs in
                        if isEmptyMEL me then (fr,updSubData me [c])
                                         else ([me],pickvar me))
```

Seria desejável evitar as concatenações efectuadas durante a recolha das fronteiras. Convém notar que não é possível exprimir (de forma simples) a função frontierMT através de foldrMRData ou foldlMRData. Estas funções efectuam uma linearização implícita da estrutura, o que de facto evitaria concatenações, mas essa linearização não permite evitar que se recolham também as fronteiras das fronteiras calculadas. Seria útil acrescentar à biblioteca de funções (mutuamente) recursivas *folds* condicionais (isto é, que recorram a determinados subdados apenas mediante a satisfação de uma determinada condição).

Fusão de Multiequações

O processo de fusão de duas multiequações é capturado pela função zipWithMRData. Fundir duas multiequações implica combiná-las de alguma forma e de seguida combinar os seus subdados. A nível dos multitermos é efectuado apenas um teste de compatibilidade, sendo que o resultado da combinação de termos vazios com não vazios é o termo não vazio. Como este processo pode falhar com um erro de Pattern, deve estar encapsulado no *monad* Uni. Também por esse motivo é utilizada a variante *monádica* zipWithMRDataM.

```
mergeME :: ME → ME → Uni ME
mergeME = zipWithMRDataM f g
    where
    --
    f me1 me2 = (return ∘ mkMEVars) (union (meL me1) (meL me2))
    --
    g EmptyMT    mt                                          = return mt
    g mt         EmptyMT                                     = return mt
    g (ms@Par{})(mt@Par{}) |parId ms    == parId mt          = return ms
    g (ms@Fun{})(mt@Fun{}) |funId ms    == funId mt
                           && funArity ms == funArity mt     = return ms
    g _          _                                           = throwError Pattern
```

Sistemas de Multiequações

Associado a cada multiequação de um sistema estará um inteiro, que indica o número de variáveis dessa multiequação presentes no interior dos multitermos do sistema.

```
data MEInfo = MEI {meVarCount :: Int, getME :: ME}
```

As multiequações e contagem de variáveis associadas, serão indexadas por um identificador único (posição) num mapeamento finito, (campo meSys do sistema). As soluções (multiequações eliminadas) são guardadas numa lista, (campo meSol) para posteriormente serem convertidas num unificador. Para simplificar as operações é mantida uma base de dados de informação acerca de cada uma das variáveis presentes no sistema, com a posição da multiequação que a contém como membro, e o número de vezes que ocorre no interior dos multitermos do sistema. As posições das multiequações com contagem associada a 0 são guardadas numa lista, (campo posAt0). O sistema contém ainda um gerador de posições (nextPos).

```
data MESys = MES{meSys     :: FiniteMap Int MEInfo,
                 varInfoDB :: FiniteMap VarId (Pos,Count)
                 meSol     :: [ME],
                 posAt0    :: [Int],
                 nextPos   :: !Int,
                }
```

Para esta estrutura estão definidas várias operações primitivas de acesso, ou actualização de dados. Como exemplo, para a base de dados de informação de variáveis, temos a possibilidade de aceder ou actualizar a informação de uma variável, em particular apenas a sua posição ou contagem no sistema, assim como remover essa informação da base de dados.

```
varInfo     :: MESys → VarId → (Pos,Count)
varPos      :: MESys → VarId → Pos
varCount    :: MESys → VarId → Count
delVarInfo  :: MESys → VarId → MESys
setVarInfo  :: MESys → (VarId,(Pos,Count)) → MESys
setVarPos   :: MESys → (VarId,Pos) → MESys
setVarCount :: MESys → (VarId,Count) → MESys
```

Entre as primitivas sobre o sistema propriamente dito, estão incluídas funções que permitem remover, adicionar, ou alterar informação sobre equações.

```
delFromMES    :: MESys → Int → MESys
addMEI        :: MESys → Pos → MEInfo → MESys
setMeVarCount :: MESys → (Pos,Count) → MESys
```

Note-se que, como a própria designação indica (primitivas), estas funções não garantem de forma alguma que o sistema resultante é consistente. Servem apenas para permitir a abstração das estruturas utilizadas na implementação do tipo de dados *Sistema*, e facilitar a sua manipulação.

Resolução de Sistemas de Multiequações

Uma sistema de multiequações é resolvido por aplicações sucessivas da função stepSys, até ser obtido um sistema vazio, sem mais multiequações para eliminar.

```
stepSys :: MESys →Uni MESys
stepSys sys = do
              n         ← pickME  sys
              sys       ← return (choiceOk sys)
              me        ← return (takeME sys n)
              (fr ,com) ← return (decME  me)
              sys       ← return (removeME sys n (frVars fr))
              sys       ← return (addSol sys com)
              sys       ← foldM  insertME sys fr
              return sys
```

Em cada passo é escolhida uma multiequação com contagem de variáveis a zero. A função pickME :: MESys →Uni **Int**, devolve a posição de uma das multiequações nessas condições caso exista. Se não existir, a computação termina com um erro de ocorrência. Se a computação não for interrompida, é confirmada a escolha (é removida a posição escolhida da lista de variáveis a 0), recolhida a multiequação escolhida do sistema, e calculadas a fronteira e a parte comum. Em seguida é removida do sistema a multiequação escolhida, adicionada à lista de soluções a parte comum calculada e inseridas as fronteiras recolhidas no sistema. Também durante esta inserção, efectuada através de sucessivas aplicações da função insertME :: MESys →ME →Uni MESys, a computação pode falhar com um erro estrutural. Os passos não triviais do processo que acabámos de descrever são a remoção da multiequação escolhida do sistema, e a inserção das fronteiras calculadas. Estes passos obrigam a várias actualizações de informação na estrutura para a manter consistente.

Compactação do Sistema

Comecemos por analizar a função insertME.

```
insertME :: MESys → ME → Uni MESys
insertME sys me =
        do
        --
        varsNew     ← return (meL me)
        posToMerge  ← return (mapSet (varPos sys) varsNew)
        mesToMerge  ← return (map (takeME sys) (setToList posToMerge))
        varsOld     ← return (unionManySets (map meL mesToMerge))
        varsAll     ← return (union varsNew varsOld)
        --
        meMerged    ← mergeMEs me mesToMerge
        (sys ,pos)  ← return (reInsertSys sys  meMerged)
        newPosList  ← return (zip  (setToList varsAll) [pos..])
        sys         ← return (foldl' setVarPos sys newPosList)
        --
        return sys
```

Estão destacadas duas fases distintas, uma primeira de recolha de informação, e uma segunda com a inserção propriamente dita. Na primeira fase são determinadas as posições de todas as multiequações do sistema que partilham pelo menos um termo não variável com a nova

multiequação a inserir (posToMerge), através de consulta à base de dados com a informação das variáveis do sistema para cada uma das variáveis da nova multiequação. Essas posições são utilizadas para recolher as multiequações do sistema (mesToMerge). Uma vez recolhida a informação passamos à segunda fase. As multiequações recolhidas são fundidas com a nova através da função mergeMEs que não é mais que a aplicação da função mergeME que funde pares de multiequações, falhando com erro estrutural caso tal não seja possível. O resultado da fusão é introduzido de novo no sistema pela função reInsertSys :: MESys →ME →(MESys,**Int**), o que implica também determinar e associar a contagem de variáveis a esta multiequação. A função anterior devolve, para além do sistema alterado, a posição onde foi efectuada a inserção, esta posição é utilizada para alterar a informação da localização das variáveis das multiequações fundidas. Por fim, o sistema resultante é devolvido.

Remoção de Multiequações

A remoção de uma multiequação do sistema implica também que se altera informação relativa a variáveis que deixam de fazer parte de multitermos na equação, ou seja os termos variáveis da fronteira recolhida. Essa lista de variáveis é recolhida no procedimento stepSys descrito anteriormente, e passada à função removeME juntamente com a posição da multiequação a remover.

```
removeME :: MESys → Int → MESys
removeME sys n = newsys
    where
    meToRm = takeME sys n
    vars   = setToList (meL meToRm)
    sys'   = delFromMES sys n
    newsys = foldl' removeVar sys vars
```

A remoção propriamente dita é simples, a alteração da informação das variáveis é mais complexa. Essa alteração é efectuada pela função removeVar :: MESys →VarId →MESys ao nível da base de dados de informação das variáveis, por um lado, e ao nível da informação associada a cada multiequação, por outro.

```
removeVar          :: MESys → VarId → MESys
removeVar sys xid = runIdentity $
        do
        let (xpos,xcount) = varInfo sys xid
        let mecount       = meVarCount sys xpos
        --
        sys ← return (setVarCount   sys (xid,xcount-1))
        sys ← return (setMeVarCount sys (xpos,mecount-1))
        sys ← return (if xcount  == 1 then delVarInfo sys xid else sys)
        sys ← return (if mecount == 1 then addPosAt0 sys xpos else sys)
        return sys
```

Como se pode ver, para além do decremento efectuado em ambos os sítios, ainda é necessário verificar se alguma das contagens (quer da própria variável, quer das variáveis da multiequação) ficaram a zero. Se a contagem da variável ficar a 0, a informação desta é removida da base de

dados. Se a contagem associada à multiequação ficar a 0, a sua posição é inserida na lista de multiequações candidatas para decomposição (e eliminação).

Unificação por Resolução de um Sistema

Para unificarmos dois termos basta então transformar cada um deles numa multiequação e resolver o sistema unitário por ela definido. A conversão de termos em multiequações, implementada pela função term2me :: T.Term →ME, é obtida por simples recursão estrutural. Uma variável resultará numa multiequação com um conjunto de variáveis singular e um multitermo vazio. Um termo não variável será convertido numa multiequação com um conjunto de variáveis vazio e um multitermo não vazio, que poderá ser um parâmetro ou uma função aplicada aos seus argumentos devidamente convertidos em multiequações. Se a unificação for concluída com sucesso, as multiequações calculadas são transformadas em substituições pela função me2substs :: ME →[Substs]. O processo é simples e consiste em associar cada uma das variáveis da multiequação ao seu multitermo (que como já vimos, representa um único termo).

3.3 Indexação de Termos

Nesta secção iremos apresentar algumas estruturas para indexação de termos. Estas estruturas permitem guardar conjuntos de termos, por forma a que se consiga, efectuar eficientemente uma recolha daqueles que satisfazem uma dada propriedade, ou pelo menos filtrar grande parte daqueles que não a satisfazem. No nosso caso estamos interessados em estruturas de indexação que facilitem a recolha de termos que unifiquem com um dado *query term*. As estruturas aqui descritas serão, *árvores de discriminação* (subsecção 3.3.1), *árvores de abstracção* (subsecção 3.3.2) e *árvores de substituição* (subsecção 3.3.3).

3.3.1 Indexação com Árvores de Discriminação

Árvores de discriminação [31] são construídas por forma a existir partilha de prefixos comuns dos termos indexados. Isto é, partilha de símbolos iniciais comuns, obtidos por travessia em pré-ordem. Os termos $f(b, g(a, b))$ e $f(b, g(h(x), y))$ têm em comum "$f(b, (g($", e portanto os símbolos f, b e g seriam partilhados na indexação de ambos numa árvore de discriminação. Já os termos $f(b, g(a, b))$ e $f(x, g(a, b))$ partilhariam apenas o símbolo f. Para aumentar a partilha de símbolos, variáveis não são distinguidas entre si, portanto os termos $f(x, g(a, b))$ e $f(y, g(z, b))$ partilhariam o prefixo "$f(*, g($", onde $*$ denota uma qualquer variável. Na figura 3.6 é apresentada uma árvore de discriminação com alguns termos indexados.

A recolha de termos para unificação com um dado *query term* t é efectuada precorrendo t em pré-ordem, e a árvore de discriminação segundo nós estruturalmente compatíveis com os nós do termo t. Quando se combinam uma variável $*$ da árvore e uma função em t, a descida estrutural de t é interrompida e prossegue-se para o próximo subtermo do mesmo. Da mesma forma, quando se combinam uma variável de t com um símbolo de função na árvore de indexação, é necessário passar para o nó descendente correspondente ao início do próximo subtermo na

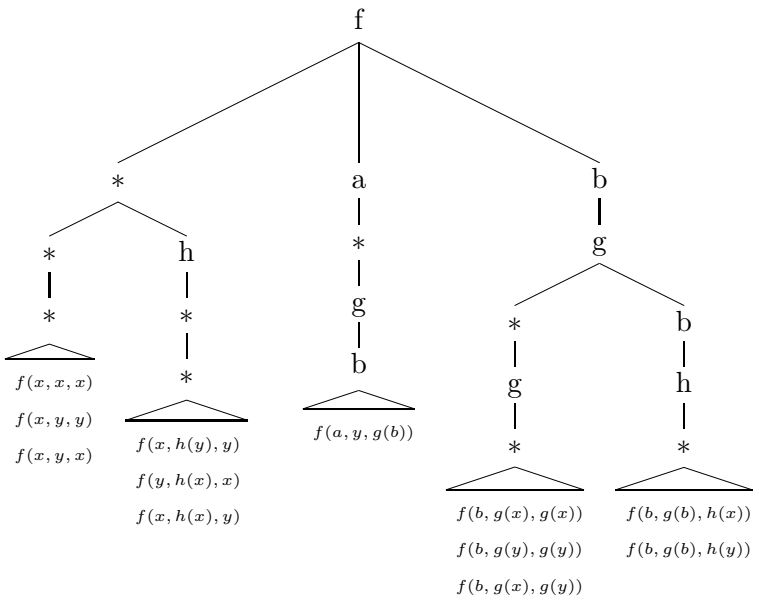

Figura 3.6: Árvore de discriminação (com filtro imperfeito).

árvore. A figura 3.7 ilustra uma recolha de termos na árvore de discriminação já apresentada, para unificação com o termo $f(b, z, h(b))$.

Associado a alguns nós da árvore de discriminação, aparece o *query term* com o símbolo a testar destacado. Os termos indexados recolhidos aparecem também em destaque. Como se pode ver, nem todos os termos recolhidos unificam com $f(b, z, h(z))$. Todos os termos do primeiro conjunto recolhido falham, $f(x, x, x)$ e $f(x, y, x)$ com erros estruturais e $f(x, y, y)$ com um erro de ocorrência, quanto ao segundo conjunto apenas o último termo unifica, ambos $f(x, h(y), y)$ e $f(x, h(y), y)$, falham com erros de ocorrência. Estes testes não podem ser efectuados durante a recolha devido à abstracção efectuada sobre as variáveis. Por este motivo, este tipo de indexação diz-se um *filtro imperfeito*. Para obter uma variante *perfeita* de *árvores de discriminação*, em vez de substituir todas as variáveis dos termos a inserir por um único símbolo '$*$', limitamo-nos a efectuar uma renomeação das variáveis, obtida através da sua enumeração. Percorrendo o termo a inserir em pré-ordem, a i-ésima nova variável é renomeada $*_i$. Como exemplo, o termo $f(x, h(z), y)$ passaria a $f(*_1, h(*_2), *_3)$, e o termo $f(x, y, x)$ a $f(*_1, *_2, *_1)$. Parte da versão *perfeita* da árvore de discriminação apresentada na figura 3.6 é apresentada na figura 3.8. Neste caso, seria facilmente detectada a incompatibilidade estrutural entre o *query term* $f(b, z, h(b))$, e os termos indexados $f(x, x, x)$ e $f(x, y, x)$, por comparação com os respectivos padrões, $f(*_1, *_1, *_1)$ e $f(*_1, *_2, *_1)$ durante a recolha. Da mesma forma, também os erros de ocorrência poderiam ser detectados. Repare-se ainda que estes erros seriam detectados para vários termos de uma só vez. Como desvantagem, este tipo de indexação efectua muito menos partilha de dados e consome muito mais memória.

Árvores de dicriminação, na sua variante perfeita, são usadas, por exemplo, pelo demonstrador automático de teoremas Waldmeister, [12, 24].

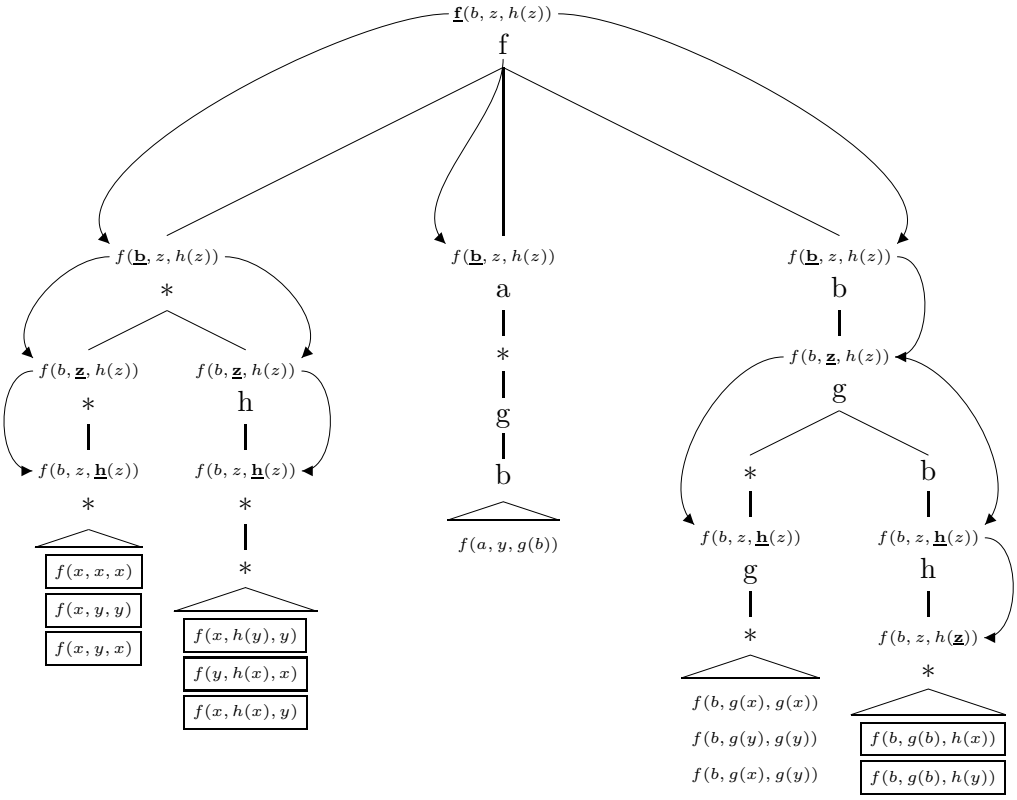

Figura 3.7: Recolha de termos indexados para unificação com $f(b, z, h(b))$.

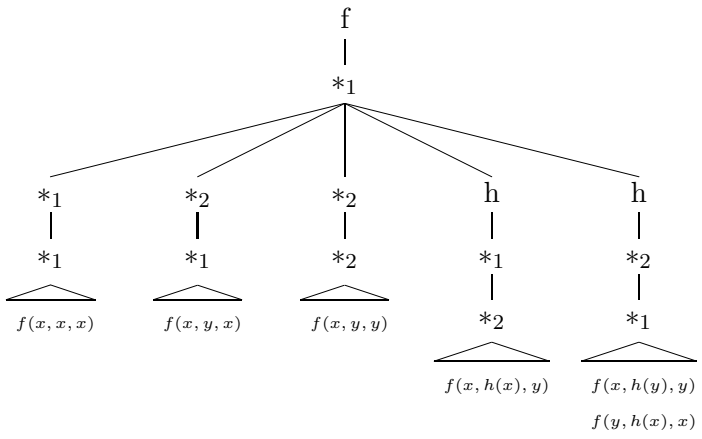

Figura 3.8: Árvore de discriminação (com filtro perfeito).

3.3.2 Indexação com Árvores de Abstracção

Árvores de abstracção [26], tiram partido da relação de pré-ordem parcial entre termos (e substituições) descrita na secção 1.5. Ou seja, é baseada na relação de instanciação entre termos. Cada nó de uma árvore de substituição possui uma lista de termos e uma lista de variáveis, que formam, respectivamente, o codomínio e o domínio de uma substituição. Os termos indexados são obtidos, começando com o termo na raiz da árvore, e aplicando sucessivamente as substituições indicadas no caminho que se percorre até à folha. Na figura 3.9 é apresentada a árvore de abstracção, para os termos que indexámos na figura 3.8 com uma árvore de discriminação perfeita.

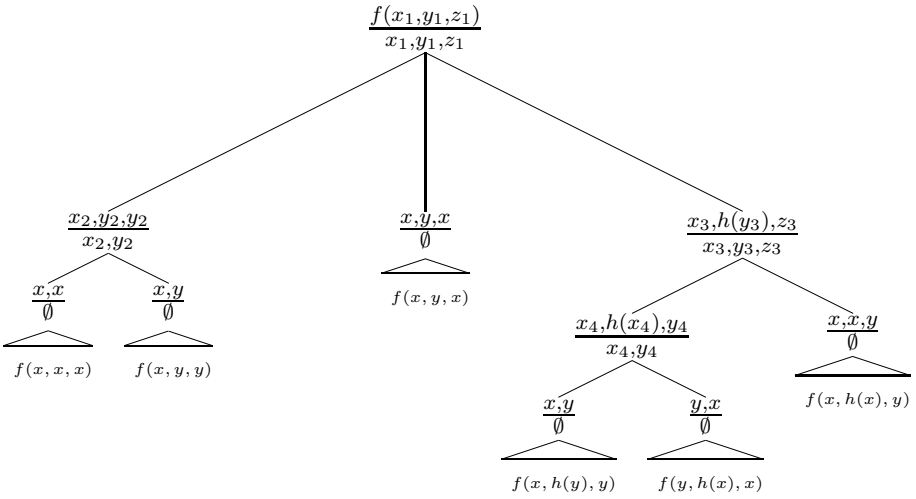

Figura 3.9: Árvore de abstracção.

Tomemos como exemplo o termo $f(x, h(y), y)$, indexado na folha mais à direita da árvore de abstracção. O caminho da raiz até à folha indicada começa com o termo $f(x_1, y_1, z_1)$ e define as substituições $\sigma_1 = \{x_1 \mapsto x_3, y_1 \mapsto h(y_3), z_1 \mapsto z_3\}$ e $\sigma_2 = \{x_3 \mapsto x, y_3 \mapsto x, z_1 \mapsto y\}$, de onde se retira que $f(x, h(y), y) = f(x_1, y_1, z_1)\sigma_1\sigma_2$. A recolha dos termos unificáveis com um dado *query term* t é bastante simples, começamos por unificar t com a raiz da árvore, ou seja, o termo mais geral de todos os indexados, se esta unificação falhar então nenhum dos filhos é unificável com t, caso contrário o unificador é aplicado às variáveis listadas nesse mesmo nó, e os termos resultantes serão unificados com as listas de termos nos nós filhos. Um percurso até às folhas significa uma unificação com sucesso.

Para além de ser um método com filtragem perfeita, este tipo de indexação permite maior partilha entre os termos do que o anteriormente apresentado. Tirar partido dessa possibilidade, no entanto, requer trabalho extra. Se no caso das árvores de discriminação, a indexação de cada termo era única, neste caso, para cada conjunto de termos existem várias árvores de abstracção possíveis (e por isso diz-se uma indexação *não deterministica*). A figura 3.10 apresenta duas possíveis configurações para uma indexação de três termos. Neste exemplo, ambas são igualmente boas, mas para um exemplo simples de uma má indexação, basta considerar a árvore em que cada termo está directamente relacionado com a raiz, ou seja, cada termo é

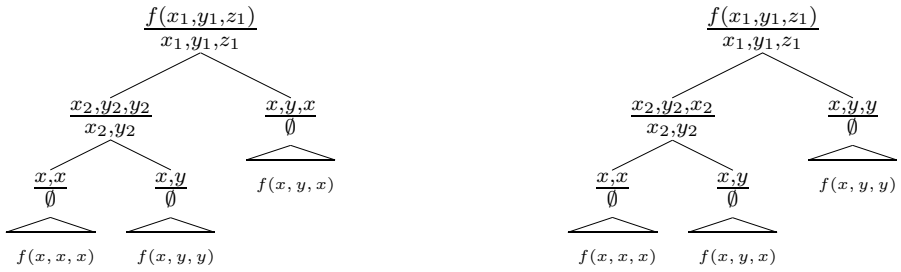

Figura 3.10: Duas árvores de abstracção distintas para um mesmo conjunto de termos.

obtido directamente a partir do mais geral, por aplicação de uma substituição (neste caso a partilha seria praticamente nula). Para obter uma boa indexação com árvores de abstracção são utilizadas heurísticas durante a sua construção, e portanto a rapidez e eficácia a nível da recolha é conseguida à custa de alguma perda de eficiência a nível das inserções de termos. Outra das desvantagens desta estrutura é o facto de serem utilizadas muitas substituições desnecessárias. O próximo método melhora esse esse aspecto, entre outros. Árvores de abstracção, são usadas por exemplo, em algumas versões do demonstrador FAUST [11, 29].

3.3.3 Indexação com Árvores de Substituição

O método de indexação com *árvores de substituição* [10] combina a indexação baseada na relação de instanciação, utilizada em árvores de abstracção, com a utilização de variáveis indicadoras $*_i$ para aumentar a partilha de termos, e diminuir o número de substituições necessárias. Como se pode ver na figura 3.11, ao contrário do que acontecia com o método anterior, cada folha está associada a mais do que um termo.

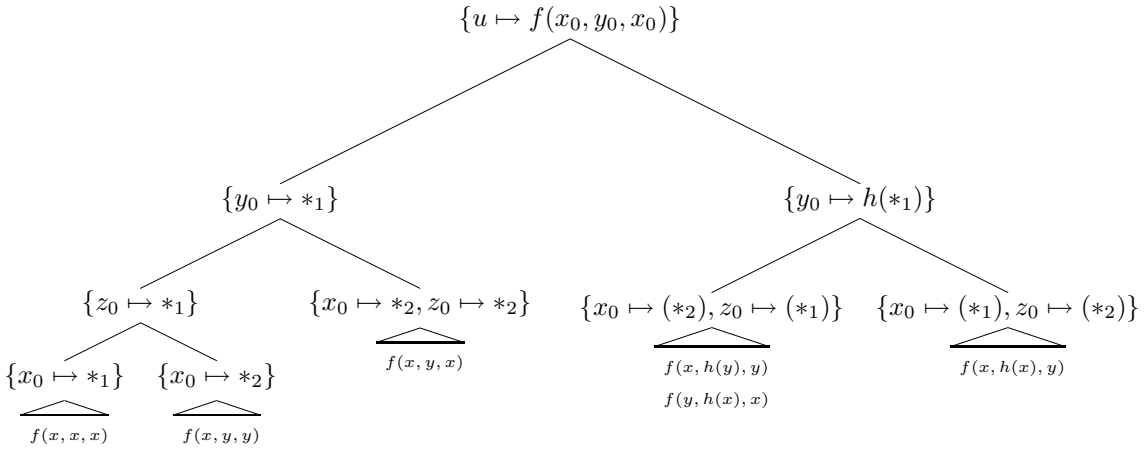

Figura 3.11: Árvore de substituição.

Consideremos o caminho até à segunda folha que contém o termo, $f(x, y, y)$. A aplicação sucessiva das substituições listadas resulta em $u\{u \mapsto f(x_0, y_0, z_0)\}\{y_0 \mapsto *_1\}\{z_0 \mapsto *_1\}\{x_0 \mapsto *_2\}$, ou seja, $f(*_2, *_1, *_1)$, portanto essa folha poderá indexar quaisquer termos que satisfaçam esse padrão. Árvores de substituição permitem recolhas extremamente rápidas, mas, mais uma vez, construir uma boa indexação exige uma inserção mais complexa, e este método, de entre

os três apresentados, é o menos eficiente a esse nível [10]. Uma implementação de árvores de substituição desenvolvida pelo próprio autor deste método de indexação [9] é utilizada no demonstrador automático SPASS [38, 31].

Capítulo 4

Processo de Inferência

Iremos debruçar-nos neste capítulo sobre a mecanização de demonstrações, ou seja, sobre o processo de inferência. No capítulo 1 (secção 1.2) escolhemos trabalhar com sistemas de Gentzen por apresentarem propriedades que simplificam a mecanização pretendida (subsecção 1.2.3) segundo um processo de aplicação inversa das regras de inferência. No capítulo 2, (secção 2.3) iniciámos o estudo da automação deste processo, que nos levou à introdução de metavariáveis para permitir considerar termos ainda não definidos na árvore de demonstração, e parâmetros para auxiliar a instanciação das metavariáveis, ajudando a garantir que a aplicação das regras de inferência resultantes são válidas. Neste capítulo focamos essencialmente aspectos relacionados com a definição das regras de inferência e a sua aplicação, e a forma como é construída a árvore de demonstração, tendo em atenção a eficiência e completude de todo o processo.

4.1 Regras de Inferência

No caso proposicional, a construção de demonstrações é bastante simples. A aplicação inversa das regras de inferência, independentemente da ordem por que é efectuada resulta sempre numa árvore de demonstração, caso exista (ver tabela 4.1), ou numa árvore de contra-exemplo caso não exista uma demonstração, (ver tabela 4.2). Contúdo, uma aplicação cuidada das regras de inferência pode tornar o processo mais ou menos eficiente. No caso de lógica de primeira ordem existe o problema da não decidibilidade do processo de demonstração, por este motivo a escolha das regras requer cuidados adicionais. Começamos por estudar aquela que é a origem da não decidibilidade no método utilizado, a necessidade de duplicação de algumas das fórmulas principais.

4.1.1 Duplicação de Fórmulas

Suponhamos que queremos demostrar que, da hipótese $\forall x.\, P(x) \Rightarrow P(f(x))$ podemos deduzir $P(b) \Rightarrow P(f(f(b)))$, para um dado b constante. Intuitivamente é fácil perceber que será necessário usar a hipótese duas vezes, uma para deduzir $P(b) \Rightarrow P(f(b))$ e outra para deduzir $P(f(b)) \Rightarrow P(f(f(b)))$, para assim conseguirmos chegar a $P(b) \Rightarrow P(f(f(b)))$. A árvore de demonstração correspondente encontra-se na tabela 4.3 (note-se que foi implicitamente utilizada a regra da monotonia antes da aplicação de cada regra $\Rightarrow_l$, para simplificar a árvore).

$$\dfrac{\dfrac{\mathbf{p},\ q \Rightarrow r \vdash\ \mathbf{p},\ r \qquad \dfrac{p,\ q,\ \mathbf{r} \vdash\ \mathbf{r} \qquad p,\ \mathbf{q} \vdash\ \mathbf{q},\ r}{p,\ q,\ q \Rightarrow r \vdash\ r}\Rightarrow_l}{p,\ p \Rightarrow q,\ q \Rightarrow r \vdash\ r}\Rightarrow_l}{p \Rightarrow q,\ q \Rightarrow r\ \vdash\ p \Rightarrow r}\Rightarrow_r$$

Tabela 4.1: Árvore de demonstração para $p \Rightarrow q,\ q \Rightarrow r\ \vdash\ p \Rightarrow r$.

$$\dfrac{\dfrac{\mathbf{p},\ r \Rightarrow q \vdash\ \mathbf{p},\ r \qquad \dfrac{p,\ q\ \vdash\ r \qquad p,\ q\ \vdash\ r}{p,\ q,\ r \Rightarrow q\ \vdash\ r}\Rightarrow_l}{p,\ p \Rightarrow q,\ r \Rightarrow q \vdash\ r}\Rightarrow_l}{p \Rightarrow q,\ r \Rightarrow q\ \vdash\ p \Rightarrow r}\Rightarrow_r$$

Tabela 4.2: Árvore de contra-exemplo para $p \Rightarrow q,\ r \Rightarrow q\ \vdash\ p \Rightarrow r$.

$$\dfrac{\dfrac{\dfrac{\mathbf{P}(\mathbf{b})\ \vdash\ \mathbf{P}(\mathbf{b}) \qquad \dfrac{\dfrac{\mathbf{P}(\mathbf{f}(\mathbf{b}))\ \vdash\ \mathbf{P}(\mathbf{f}(\mathbf{b})) \qquad \mathbf{P}(\mathbf{f}(\mathbf{f}(\mathbf{b})))\ \vdash\ \mathbf{P}(\mathbf{f}(\mathbf{f}(\mathbf{b})))}{P(f(b)) \Rightarrow P(f(f(b))), P(f(b))\ \vdash\ P(f(f(b)))}\Rightarrow_l}{\forall x.\, P(x) \Rightarrow P(f(x)), P(f(b))\ \vdash\ P(f(f(b)))}\forall_l}{P(b) \Rightarrow P(f(b)), \forall x.\, P(x) \Rightarrow P(f(x)), P(b)\ \vdash\ P(f(f(b)))}\Rightarrow_l}{\forall x.\, P(x) \Rightarrow P(f(x)), P(b)\ \vdash\ P(f(f(b)))}\forall_l}{\forall x.\, P(x) \Rightarrow P(f(x))\ \vdash\ P(b) \Rightarrow P(f(f(b)))}\Rightarrow_r$$

Tabela 4.3: Árvore de demonstração para $\forall x.\, P(x) \Rightarrow P(f(x))\ \vdash\ P(b) \Rightarrow P(f(f(b)))$.

Como se pode ver, a regra $\forall_l$ é aplicada duas vezes para a mesma fórmula $\forall x.\, P(x) \Rightarrow P(f(x))$. Tentemos construir esta demonstração segundo o método de aplicação inversa das regras de inferência, que introduzimos na secção 2.3. A tabela 4.4 apresenta uma tentativa falhada. Na folha esquerda poderíamos unificar $?x$ com b para obter um axioma, mas da instanciação na folha direita obteríamos $P(f(b)), P(b) \vdash P(f(f(b)))$, que não pode ser demonstrado. De forma análoga, se optássemos por unificar $?x$ com $f(b)$ para obter um axioma na folha direita, obteríamos um não teorema na folha esquerda.

$$\dfrac{\dfrac{\dfrac{P(b) \vdash P(?x), P(f(f(b))) \quad P(f(?x)), P(b) \vdash P(f(f(b)))}{P(?x) \Rightarrow P(f(?x)), P(b) \vdash P(f(f(b)))}\Rightarrow_l}{\forall x.\, P(x) \Rightarrow P(f(x)), P(b) \vdash P(f(f(b)))}\forall_l}{\forall x.\, P(x) \Rightarrow P(f(x)) \vdash P(b) \Rightarrow P(f(f(b)))}\Rightarrow_r$$

Tabela 4.4: Tentativa de demonstração de $\forall x.\, P(x) \Rightarrow P(f(x)) \vdash P(b) \Rightarrow P(f(f(b)))$.

O problema resultou do facto de descartarmos, na aplicação inversa da regra $\forall_l$, a fórmula principal que decompusemos (e instanciámos). Durante a aplicação inversa deveríamos ter duplicado a fórmula principal, ou seja, esta deveria continuar a aparecer na premissa para que pudesse ser novamente decomposta, com outra metavariável, para uma eventual instanciação com um novo termo. Esta observação é válida também para a regra $\exists_r$ e, de um modo geral (considerando outros sistemas lógicos de Gentzen), para quaisquer regras com quantificadores que envolvam a 'escolha de um termo' (ou seja, a utilização de metavariáveis). Passamos então a escrever as regras $\forall_l$ e $\exists_r$, para aplicação inversa, por forma a explicitar a duplicação das fórmulas principais.

$$\dfrac{\Gamma, A(t), \forall x.\, A(x) \vdash \Delta}{\Gamma, \forall x.\, A(x) \vdash \Delta}\forall_l \qquad \dfrac{\Gamma \vdash A(y), \Delta}{\Gamma \vdash \forall x.\, A(x), \Delta}\forall_r$$

$$\dfrac{\Gamma, A(y) \vdash \Delta}{\Gamma, \exists x.\, A(x) \vdash \Delta}\exists_l \qquad \dfrac{\Gamma \vdash A(t), \exists x.\, A(x), \Delta}{\Gamma \vdash \exists x.\, A(x), \Delta}\exists_r$$

Após estas observações, apresentamos uma nova tentativa de criação da árvore de demonstração de $\forall x.\, P(x) \Rightarrow P(f(x)) \vdash P(b) \Rightarrow P(f(f(b)))$. Numa primeira fase, após a aplicação de $\forall_l$, tal como na tentativa anterior podemos escolher uma de duas unificações possíveis (uma em cada folha). Escolhemos a unificação $?x \mapsto b$ que resolve a folha da esquerda (tabela 4.5).

Após a instanciação de $?x$ com b podemos verificar que a aplicação de $\forall_l$ resultou no termo $P(b) \Rightarrow P(f(b))$ como premissa. A demonstração resume-se agora à folha direita, onde tentamos deduzir $P(f(f(b)))$ não só de $\forall x.\, P(x) \Rightarrow P(f(x))$ e $P(b)$, mas também de $P(f(b))$ (tabela 4.6).

O resto do processo de demonstração é semelhante, desta vez, ambos os ramos podem ser

$$\dfrac{\dfrac{\dfrac{\forall x.\, P(x) \Rightarrow P(f(x)), \mathbf{P(b)} \;\vdash\; \mathbf{P(?x)}, P(f(f(b)))}{P(?x) \Rightarrow P(f(?x)), \forall x.\, P(x) \Rightarrow P(f(x)), P(b) \;\vdash\; P(f(f(b)))}\,\forall_l}{\forall x.\, P(x) \Rightarrow P(f(x)), P(b) \;\vdash\; P(f(f(b)))}\,\Rightarrow_r}{\forall x.\, P(x) \Rightarrow P(f(x)) \;\vdash\; P(b) \Rightarrow P(f(f(b)))}$$

Tabela 4.5: Demonstração de $\forall x.\, P(x) \Rightarrow P(f(x)) \vdash P(b) \Rightarrow P(f(f(b)))$ (primeira fase).

$$\dfrac{\dfrac{\dfrac{\forall x.\, P(x) \Rightarrow P(f(x)), \mathbf{P(b)} \;\vdash\; \mathbf{P(b)}, P(f(f(b)))}{P(b) \Rightarrow P(f(b)), \forall x.\, P(x) \Rightarrow P(f(x)), P(b) \;\vdash\; P(f(f(b)))}\,\forall_l}{\forall x.\, P(x) \Rightarrow P(f(x)), P(b) \;\vdash\; P(f(f(b)))}\,\Rightarrow_r}{\forall x.\, P(x) \Rightarrow P(f(x)) \;\vdash\; P(b) \Rightarrow P(f(f(b)))}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{P(f(b)), \forall x.\, P(x) \Rightarrow P(f(x)), P(b) \;\vdash\; P(f(f(b)))}}{\forall x.\, P(x) \Rightarrow P(f(x)), \mathbf{P(b)}, P(b) \;\vdash\; \mathbf{P(b)}, P(f(f(b)))}\,\Rightarrow_l$$

Tabela 4.6: Demonstração de $\forall x.\, P(x) \Rightarrow P(f(x)) \vdash P(b) \Rightarrow P(f(f(b)))$ (primeira instanciação).

terminados, unificando $?y$ com $f(b)$ (tabela 4.7). Da instanciação correspondente resulta a subárvore de demonstração que faltava (tabela 4.8). Repare-se como, após a instanciação, aparece na premissa da aplicação de $\forall_l$ a hipótese $P(f(b)) \Rightarrow P(f(f(b)))$, tal como esperávamos.

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, P(x) \Rightarrow P(f(x)), \mathbf{P(f(b))}, P(b) \;\vdash\; \mathbf{P(?y)}, P(f(f(b)))}{P(?y) \Rightarrow P(f(?y)), \forall x.\, P(x) \Rightarrow P(f(x)), P(f(b)), P(b) \;\vdash\; P(f(f(b)))}\,\forall_l}{P(f(b)), \forall x.\, P(x) \Rightarrow P(f(x)), P(b) \;\vdash\; P(f(f(b)))}}{\vdots}\,\Rightarrow_l$$

Tabela 4.7: Demonstração de $\forall x.\, P(x) \Rightarrow P(f(x)) \vdash P(b) \Rightarrow P(f(f(b)))$ (segunda fase).

É importante notar que desta duplicação de fórmulas podem resultar subárvores de profundidade infinita. Este facto é reflexo da não decidibilidade da lógica de primeira ordem (secção 1.3). Na prática, o número de duplicação das fórmulas quantificadas é limitado a um valor, fixo ou calculado segundo uma certa heurística.

$$\dfrac{\dfrac{\dfrac{}{\forall x.\, P(x) \Rightarrow P(f(x)), \mathbf{P(f(b))}, P(b) \vdash \mathbf{P(f(b))}, P(f(f(b)))} \quad \dfrac{}{\mathbf{P(f(f(b)))}, \forall x.\, P(x) \Rightarrow P(f(x)), \mathbf{P(f(b))}, P(b) \vdash P(f(f(b)))}\ \Rightarrow_l}{P(f(b)) \Rightarrow P(f(f(b))), \forall x.\, P(x) \Rightarrow P(f(x)), P(f(b)), P(b) \ \vdash\ P(f(f(b)))}\ \forall_l}{P(f(b)), \forall x.\, P(x) \Rightarrow P(f(x)), P(b) \ \vdash\ P(f(f(b)))}$$

$$\vdots$$

Tabela 4.8: Demonstração de $\forall x.\, P(x) \Rightarrow P(f(x)) \vdash P(b) \Rightarrow P(f(f(b)))$ (segunda instanciação).

4.1.2 Regras do Sistema de Dedução

Como referimos no primeiro capítulo (secção 1.1), existem outras conectivas de utilização comuns em sistemas de lógica de primeira ordem clássica, para além das que considerámos na nossa definição. Convém notar que, se por um lado, em teoria, poderiamos trabalhar com um conjunto bastante reduzido de conectivas, na prática a introdução explícita de regras de inferência para algumas conectivas extra, pode tornar as demonstrações mais simples. No contexto de inferência automática essa característica é particularmente importante, pois demonstrações mais complicadas serão, em princípio, mais difícieis de calcular. Consideremos o caso da conectiva $\Leftrightarrow$. Se não introduzirmos regras explícitas para esta conectiva, para expressarmos equivalência traduzimos fórmulas do tipo $A \Leftrightarrow B$ como $A \Rightarrow B \wedge B \Rightarrow A$. As tabelas 4.9 e 4.10 apresentam árvores de inferência para sequentes que contenham equivalências à esquerda e direita, respectivamente.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \ \vdash\ A, B, \Delta \qquad \mathbf{A}\ , \Gamma \ \vdash\ \mathbf{A}, \Delta}{B \Rightarrow A\ , \Gamma \ \vdash\ A, \Delta}\ \Rightarrow_l \qquad \dfrac{\mathbf{B}, \Gamma \ \vdash\ \mathbf{B}, \Delta \qquad A, B\ , \Gamma \ \vdash\ \Delta}{B, B \Rightarrow A\ , \Gamma \ \vdash\ \Delta}\ \Rightarrow_l}{A \Rightarrow B, B \Rightarrow A\ , \Gamma \ \vdash\ \Delta}\ \Rightarrow_l}{A \Rightarrow B \wedge B \Rightarrow A\ , \Gamma \ \vdash\ \Delta}\ \wedge_l}{A \Leftrightarrow B\ , \Gamma \ \vdash\ \Delta}\ def \Leftrightarrow$$

Tabela 4.9: Árvore de inferência para $A \Leftrightarrow B\ , \Gamma \ \vdash\ \Delta$.

$$\dfrac{\dfrac{\dfrac{A,\ \Gamma \ \vdash\ B, \Delta}{\Gamma \ \vdash\ A \Rightarrow B, \Delta}\ \Rightarrow_r \qquad \dfrac{B,\ \Gamma \ \vdash\ A, \Delta}{\Gamma \ \vdash\ B \Rightarrow A, \Delta}\ \Rightarrow_r}{\Gamma \ \vdash\ A \Rightarrow B \wedge B \Rightarrow A, \Delta}\ \wedge_r}{\Gamma \ \vdash\ A \Leftrightarrow B, \Delta}\ def \Leftrightarrow$$

Tabela 4.10: Árvore de inferência para $\Gamma \ \vdash\ A \Leftrightarrow B\ , \Delta$.

Repare-se ainda que se, após a expansão de $\Leftrightarrow$ segundo a sua definição, decompuséssemos uma das fórmulas em Γ ou Δ segundo uma regra que criasse dois ramos, duplicaríamos esta árvore de inferência, portanto a utilização da definição e não de regras próprias pode complicar bastante a demonstração. Assim sendo, é conveniente introduzir regras de inferência próprias para a equivalência. São elas:

$$\frac{\Gamma \vdash A, B, \Delta \qquad A, B, \Gamma \vdash \Delta}{\Gamma \vdash A \Leftrightarrow B, \Delta} \Leftrightarrow_l \qquad\qquad \frac{A, \Gamma \vdash B, \Delta \qquad B, \Gamma \vdash A, \Delta}{A \Leftrightarrow B, \Gamma \vdash \Delta} \Leftrightarrow_r$$

Já o operador referente à disjunção exclusiva ($\dot{\vee}$), por exemplo, pode perfeitamente ser definido à custa de $\Leftrightarrow$, expandido $A \dot{\vee} B$ para $\neg(A \Leftrightarrow B)$, uma vez que considerar regras próprias significaria apenas evitar a aplicação de uma regra do tipo $\neg$, que é trivial.

4.1.3 Sequentes Restritos

Como vimos na secção 2.3, para garantir que a instanciação de metavariáveis não viola as condições das regras de inferência de $\forall_r$ e $\exists_l$, associamos às variáveis livres (parâmetros) resultantes da aplicação destas regras, as metavariáveis presentes no sequente nessa altura. Na prática, esta associação requer a recolha das metavariáveis do sequente, o que implica a travessia de todas as fórmulas. Para evitar este trabalho podemos manter associado a cada sequente um conjunto de metavariáveis, inicialmente vazio, ao qual irá ser adicionada cada metavariável resultante da aplicação das fórmulas $\forall_l$ e $\exists_r$. Desta forma, quando um parâmetro é criado, basta associar ao mesmo o conjunto de metavariáveis já calculado. De seguida são apresentadas as regras de inferência para quantificadores, alteradas por forma a explicitar a alternativa de implementação que acabámos de descrever.

$$\frac{\Gamma, A(?y), \forall x.\, A(x) \vdash \Delta \parallel \{?y\} \cup \mathcal{R}}{\Gamma, \forall x.\, A(x) \vdash \Delta \parallel \mathcal{R}} \forall_l \qquad \frac{\Gamma \vdash A(y\mathcal{R}), \Delta \parallel \mathcal{R}}{\Gamma \vdash \forall x.\, A(x), \Delta \parallel \mathcal{R}} \forall_r$$

$$\frac{\Gamma, A(y\mathcal{R}) \vdash \Delta \parallel \mathcal{R}}{\Gamma, \exists x.\, A(x) \vdash \Delta \parallel \mathcal{R}} \exists_l \qquad \frac{\Gamma \vdash A(?y), \exists x.\, A(x), \Delta \parallel \{?y\} \cup \mathcal{R}}{\Gamma \vdash \exists x.\, A(x), \Delta \parallel \mathcal{R}} \exists_r$$

Outra possibilidade consiste em associar a cada metavariável os respectivos parâmetros proíbidos, em vez de se associarem a cada parâmetro as metavariáveis correspondentes. Para isso basta, de cada vez que se cria um novo parâmetro, associá-lo como restrição a todas as metavariáveis presentes no sequente. Neste caso, para evitar percorrer sequentes em busca de metavariáveis podemos manter, associado a cada sequente, uma tabela com as metavariáveis criadas e os parâmetros proibidos correspondentes. Assim de cada vez que é criada uma nova metavariável, ela é adicionada à tabela, sem nenhum parâmetro associado, e de cada vez que é criado um parâmetro, este é adicionado aos conjuntos de restrições de todas as metavariáveis presente na tabela. As regras para quantificadores alteradas por forma a explicitar o método que descrevemos são apresentadas de seguida:

$$\frac{\Gamma, A(?y), \forall x.\, A(x) \vdash \Delta \parallel \{(?y, \emptyset)\} \cup \mathcal{T}}{\Gamma, \forall x.\, A(x) \vdash \Delta \parallel \mathcal{T}}\ \forall_l \qquad \frac{\Gamma \vdash A(y), \Delta \parallel \{(?x, \{y\} \cup \mathcal{R}) \mid (?x, \mathcal{R}) \in \mathcal{T}\}}{\Gamma \vdash \forall x.\, A(x), \Delta \parallel \mathcal{T}}\ \forall_r$$

$$\frac{\Gamma, A(y) \vdash \Delta \parallel \{(?x, \{y\} \cup \mathcal{R}) \mid (?x, \mathcal{R}) \in \mathcal{T}\}}{\Gamma, \exists x.\, A(x) \vdash \Delta \parallel \mathcal{T}}\ \exists_l \qquad \frac{\Gamma \vdash A(?y), \exists x.\, A(x), \Delta \parallel \{(?y, \emptyset)\} \cup \mathcal{T}}{\Gamma \vdash \exists x.\, A(x), \Delta \parallel \mathcal{T}}\ \exists_r$$

Qualquer destas alternativas, baseadas em metavariáveis, parâmetros, e restrições associadas, são formas equivalentes de implementar a modificação de sistemas de Gentzen conhecida como *Cálculo de Sequentes Restritos*, descrita em [18, 29].

4.1.4 Implementação das Regras de Inferência

Representação das Regras de Inferência

A representação das regras de inferência pode ser simplificada por omissão das fórmulas secundárias dos sequentes (tabela 4.11). Este tipo de apresentação é utilizado por exemplo em [33]. A forma como implementámos as tabelas de dedução, tenta reproduzir uma apresentação

$$\frac{A, B \vdash}{A \wedge B \vdash}\ \wedge_l \qquad \frac{\vdash A \quad \vdash B}{\vdash A \wedge B}\ \wedge_r$$

$$\frac{A \vdash \quad B \vdash}{A \vee B \vdash}\ \vee_l \qquad \frac{\vdash A, B}{\vdash A \vee B}\ \vee_r$$

$$\frac{\vdash A \quad B \vdash}{A \Rightarrow B \vdash}\ \Rightarrow_l \qquad \frac{A \vdash B}{\vdash A \Rightarrow B}\ \Rightarrow_r$$

$$\frac{\vdash A, B \quad A, B \vdash}{A \Leftrightarrow B \vdash}\ \Leftrightarrow_l \qquad \frac{A \vdash B \quad B \vdash A}{\vdash A \Leftrightarrow B}\ \Leftrightarrow_r$$

$$\frac{\vdash A}{\neg A \vdash}\ \neg_l \qquad \frac{A \vdash}{\vdash \neg A}\ \neg_r$$

$$\frac{A(?x), \forall x.\, A(x) \vdash}{\forall x.\, A(x) \vdash}\ \forall_l \qquad \frac{\vdash A(y)}{\vdash \forall x.\, A(x)}\ \forall_r$$

$$\frac{A(y) \vdash}{\exists x.\, A(x) \vdash}\ \exists_l \qquad \frac{\vdash A(?x), \exists x.\, A(x)}{\vdash \exists x.\, A(x)}\ \exists_r$$

Tabela 4.11: Regras de dedução simplificadas por omissão das fórmulas secundárias

um pouco semelhante. A função redCon define a tabela de dedução para as conectivas:

```
redCon :: DeductionTableCon

redCon L NEG [a]   = [ [ ] |- [a]                    ]
redCon R NEG [a]   = [ [a] |- [ ]                    ]

redCon L AND [a,b] = [ [a,b] |- [ ]                  ]
redCon R AND [a,b] = [ [   ] |- [a]   ,  [ ] |- [b]  ]

redCon L OR  [a,b] = [ [a] |- [   ]   ,  [b] |- [ ]  ]
redCon R OR  [a,b] = [ [ ] |- [a,b]                  ]

redCon L IMP [a,b] = [ [ ] |- [a]     ,  [b] |- [ ]  ]
redCon R IMP [a,b] = [ [a] |- [b]                    ]

redCon L EQV [a,b] = [ [a,b] |- [ ]   ,  [ ] |-[a,b] ]
redCon R EQV [a,b] = [ [a  ] |- [b]   ,  [b] |-[a  ] ]
```

Analisemos, por exemplo, a definição da regra de inferência para $\wedge_r$.

```
redCon R AND [a,b] = [ [ ] |- [a]   ,  [ ] |- [b]   ]
```

Os argumentos da função indicam o tipo de regra a aplicar, trata-se de uma regra para o membro direito (R) ou seja, antecedente, e para a conectiva $\wedge$ (AND). Como já sabemos que a conectiva principal é uma conjunção, para definirmos a fórmula principal necessitamos apenas dos argumentos dessa conectiva, ou seja, das fórmulas laterais, que são o último argumento da função ([a,b]). As premissas correspondentes são apresentadas no resultado da função redCon, a primeira é dada pelo sequente $\vdash A$ e segunda pelo sequente $\vdash B$ (omitindo fórmulas secundárias). A apresentação da secção da tabela de dedução referente aos quantificadores é semelhante:

```
redQnt :: DeductionTableQnt

redQnt L FA x a  = [ [dup , a|?|x] |- [      ] ]
redQnt R FA x a  = [ [             ] |- [a|!|x] ]

redQnt L EX x a  = [ [a|!|x] |- [            ] ]
redQnt R EX x a  = [ [      ] |- [dup , a|?|x] ]
```

Neste caso, existe um argumento extra, a variável quantificada, e três símbolos (|?|, |!|, dup). Como exemplo, a premissa da regra $\forall_l$, a primeira das regras para quantificadores, é [dup , a|?|x] |- [ ] que representa um sequente com consequente vazio, e que no antecedente duplica a fórmula principal (dup) e contém a fórmula quantificada (a) mas com as ocorrências da variável quantificada substituídas por uma metavariável (|?|). A expressãoa |!| x, que aparece nas regras $\forall_r$ e $\exists_l$, representa a fórmula a com a variável ligada da fórmula principal substituída por um parâmetro.

Na verdade as funções redCon e redQnt não recebem nem devolvem sequentes, limitam-se a definir regras de inferência através de *acções*. Acções são operações a efectuar sobre as fórmulas

principal ou laterais, e sobre o sequente resultante da remoção dessa mesma fórmula, para obter os novos sequentes. Definimos quatro tipos de acções, *duplicar* a fórmula original, *fixar* uma fórmula lateral, substituir uma *variável ligada* por uma *metavariável* numa fórmula lateral, e substituir uma *variável ligada* por um *parâmetro* numa fórmula lateral. A cada acção deve ser associado o lado do sequente ao qual irá ser aplicada.

```
data Action = Dup | Fixed Form | Variab VarId Form | Param ParId Form
```

Cada regra da tabela é definida por pares (lado, acção), que são por sua vez definidos com base nos operadores (|−), (|?|), (|!|) e (dup).

```
(|−)  :: [Action] → [Action] → [(Side, Action)]
(|?|) :: Action   → VarId    → Action
(|!|) :: Action   → ParId    → Action
dup   :: Action
```

Comecemos por analisar uma das regras de inferência para conectivas. A regra para $\wedge_r$ é definida por:

```
redCon R AND [a,b] = [ [ ] |− [a]  ,  [ ] |− [b] ]
```

Os argumentos da função redCon são, o lado e a conectiva que estamos a tratar, neste caso lado direito (R) e conectiva $\wedge$ (AND), e acções que definem as fórmulas laterais obtidas por decomposição da fórmula orginal, na forma de acções do tipo Fixed. O operador (|−) recebe uma lista de acções a aplicar no lado esquerdo e uma outra para o lado direito, e transforma-as numa lista de pares (lado, acção). Neste caso é efectuada apenas a operação [ ] |− [a], que define a lista de pares (lado, acção) [(L,a)] cujo único elemento deve ser interpretado como *"adicionar a lado esquerdo uma fórmula (fixa) a"*, e [b] |− [ ] que, de forma análoga, deve ser interpretado como *"adicionar a lado direito uma fórmula (fixa) b"*. Repare-se que é definida uma lista de pares (lado, acção) para cada novo sequente a construir. A regra para $\wedge_l$, por exemplo, define uma única lista.

```
redCon L AND [a,b] = [ [a,b] |− [ ] ]
```

O resultado de [a,b] |− [ ] é [(L,a), (L,b)] e será interpretado como *"adicionar ao lado esquerdo do sequente a fórmula (fixa) a e em seguida b"*.

As regras para quantificadores são um pouco mais complexas. Para além da inserção de fórmulas num dos lados do sequente, poderão envolver substituições de variáveis ligadas por parâmetros ou metavariáveis, e duplicação da fórmula original. As regras para o quantificador $\forall$ são:

```
redQnt L FA x a  = [ [dup , a|?|x] |- [      ] ]
redQnt R FA x a  = [ [            ] |- [a|!|x] ]
```

Ambas resultam em apenas uma lista de pares (lado, acção). Para a regra $\forall_l$ temos como resultado a expressão `[dup , a|?|x] |- [ ]`, ou seja, `[(L,Dup), (L,Variab x a')]`, onde `a'` representa a formula fixada em `a`. Esta lista de pares deverá ser interpretada como, *"duplicar a fórmula principal no antecedente do novo sequente, e adicionar também ao antecedente a fórmula quantificada, mas com a variável ligada substituída por uma nova metavariável"*. A regra para o quantificador à direita é semelhante, mas não envolve duplicação da forma original e a substituição envolve a criação de um novo parâmetro, em vez de uma metavariável.

A interpretação das acções para construção dos novos sequentes envolve a transformação dos pares (lado, acção) em pares (lado,fórmula), e posterior introdução de cada uma das fórmulas no membro correspondente do sequente considerado. A tradução das acções para fórmulas é efectuada pela função `actionToForm`. Esta função recebe um conjunto de identificadores de metavariáveis (as restrições para os parâmetros introduzidos), a fórmula original (principal), e a acção em questão. O resultado é uma operação no *monad* `ActionM`. Este *monad* é utilizado para controlar os geradores de identificadores de metavariáveis e parâmetros e, em versões mais recentes do demonstrador, para recolha desses mesmos identificadores para associar aos sequentes criados.

```
actionToForm                    :: Set VarId→Form→Action→ActionM Form
actionToForm _ orig Dup         = return orig
actionToForm _   _ (Fixed    a) = return a
actionToForm _   _ (Variab _ a) = fmap (\xid→substBndForm(mkVar xid) a)
                                       newVarId
actionToForm rst _ (Param  _ a) = fmap (\pid→substBndForm(mkPar pid rst) a)
                                       newParId
```

A tradução das acções de duplicação e de fórmulas fixas é bastante simples, consiste apenas em devolver, respectivamente, a fórmula principal e a fórmula fixada. No restantes casos é necessário efectuar uma substituição da variável quantificada por uma metavariável ou um parâmetro, estes gerados (e acumulados no monad, em versões mais recentes) pelas funções monádicas `newVarId` e `newParId`.

A aplicação de regras de inferência envolve ainda, numa primeira fase, a identificação da regra correspondente à fórmula principal em causa, e a decomposição da mesma em fórmulas laterais. A função `applyDedTable` implementa esse procedimento.

```
applyDedTable :: Side→Form → [[(Side,Action)]]
applyDedTable s (Con{conSym=con,conForms=as}) = redCon s con (map Fixed as)
applyDedTable s (Qnt{qntSym=qnt,qntForm=a,qntVarId=x})
                                              = redQnt s qnt x (Fixed a)
```

Repare-se como as fórmulas laterais (argumentos das conectivas ou dos quantificadores) são traduzidas para acções (fórmulas fixas) para que possam ser manipuladas pelas funções redCon e redQnt.

A implementação das tabelas de dedução sob a forma de acções tem um objectivo duplo. Por um lado pretende simplificar a implementação das regras do sistema utilizado através da criação de uma espécie de meta-linguagem. Por outro, tornar possível a análise destas regras por outras funções para, por exemplo, as classificar de acordo com determinadas medidas de custo.

Representação dos Sequentes

Cada sequente irá conter as fórmulas do antecedente, as fórmulas do consequente, e eventualmente restrições associadas. A representação mais óbvia seria baseada em duas listas de fórmulas (membros esquerdo e direito do sequente) e um conjunto de restrições. Esta estrutura seria contudo muito pouco adequada ao tipo de operações que desejamos efectuar, que são selecção e remoção de fórmulas não atómicas para decomposição, inserção de novas fórmulas, e unificação de predicados do antecedente com predicados do consequente.

As fórmulas não atómicas devem ser indexadas separadamente das atómicas, pois os acessos efectuados são bastante diferentes. A indexação de fórmulas não atómicas (FormIx) irá depender da forma como é efectuada a selecção das fórmulas principais e a inserção das novas fórmulas calculadas. Por exemplo, se seguirmos uma estratégia do tipo *last in first out* poderemos utilizar uma pilha, ou, se a selecção for efectuada de acordo com um determinado custo inteiro, poderemos optar por um mapeamento finito utilizando esse mesmo custo como chave poderia ser uma opção. A selecção de fórmulas para decomposição será tratada com mais detalhe na subsecção 4.2.1 . Note-se que de um modo geral não existem vantagens em separar as fórmulas não atómicas do antecedente das fórmulas não atómicas do consequente (contudo, neste caso, dever-se-á associar a cada uma das fórmulas o lado a que pertencem, como é óbvio). Quanto à indexação dos predicados, deve ser efectuada por forma a facilitar a sua unificação, ou seja de preferência usando as técnicas descritas no capítulo 3[1]. São mantidas quatro colecções de predicados distintas, duas de predicados indexados (PrdIx) e uma de predicados por indexar (Form), para cada membro do sequente.

```
data Sequent = Sq{ sqPrdIxL   :: PrdIx
                 , sqPrdIxR   :: PrdIx
                 , sqPrdNewL  :: [Form]
                 , sqPrdNewR  :: [Form]
                 , sqFIx      :: FormIx
                 , sqRest     :: Restrictions
                 }
```

Assim, sempre que são obtidos novos predicados por decomposição de fórmulas, estes são colocados na lista de predicados não indexados correspondente, e só serão indexados após servirem

[1]Neste momento a indexação de predicados utilizada é trivial, e consiste apenas em listas de listas de termos (ou seja, uma listas com os argumentos do predicado) indexadas pelo símbolo e aridade desse mesmo predicado.

de *query term* para unificação com predicados indexados do membro oposto do sequente.

No que diz respeito à implementação das restrições associadas aos sequentes optou-se pelo primeiro método descrito na subsecção 4.1.3. Ao contrário do que acontece na abordagem escolhida, no segundo método descrito não existe qualquer referência às restrições ao nível dos termos das fórmulas de um sequente, o que complica o processo de unificação pois cria a necessidade de um argumento extra (tabela de restrições) para se poder efectuar verificações de ocorrência. Em termos de gestão de memória, uma vez que em Haskell ocorrências múltiplas de um mesmo parâmetro são partilhadas em memória, não irá acontecer duplicação desnecessária dos conjuntos de restrições associados. De facto, poderá mesmo acontecer que parâmetros distintos partilhem um mesmo conjunto de restrições (basta que o conjunto de metavariáveis associado ao sequente seja o mesmo na altura da criação destes parâmetros).

4.2 Construção da Árvore de Inferência

A construção de uma árvore de demonstração implica escolhas sucessivas da folha da árvore de inferência que iremos expandir e da fórmula sobre a qual irá ser efectuada a expansão. Nesta secção começamos por estudar este segundo aspecto, como escolher aquela que será a fórmula principal de um dado sequente, para de seguida nos debruçarmos sobre a ordem pela qual serão expandidas as folhas da árvore de inferência.

4.2.1 Escolha da Fórmula Principal

Um critério natural para a escolha da fórmula principal num sequente é uma eventual *ordem de custo* da aplicação inversa das regras de inferência. Comecemos por analisar as regras de inferência para as conectivas. A única diferença relevante entre estas regras é o número de premissas que têm. Enquanto que a decomposição de fórmulas segundo as regras $\wedge_l$, $\vee_r$ e $\Rightarrow_r$ (com uma premissa apenas) resulta apenas num novo sequente, da aplicação das regras $\wedge_r$, $\vee_l$ $\Rightarrow_l$ (com duas premissas) resultam dois sequentes, ambos contendo todas as restantes fórmulas da conclusão (fórmulas secundárias) que poderão ter que vir a ser decompostas em ambos os ramos. Por este motivo devemos dar prioridade ao primeiro grupo de regras de inferência. Veja-se, como exemplo, as tabelas 4.12 e 4.13, onde se apresentam duas demonstrações cuja única diferença é a aplicação da regra $\Rightarrow_l$, que é deixada para último lugar na primeira demonstração, e é aplicada em primeiro lugar na segunda. No caso geral, deve ser dada prioridade à aplicação de regras de inferência para conectivas com menor número de premissas, por forma a reduzir a largura da árvore de demonstração, e consequentemente o número de decomposições (potencialmente) efectuadas.

A análise das regras de inferência para quantificadores tem que ter em conta também outros factores. Um dos principais cuidados a ter na forma como se escolhe a fórmula principal de um sequente, é o de garantir que todas as fórmulas serão eventualmente escolhidas. Estratégias que satisfazem este requisito dizem-se *justas* [29]. Se nos restringirmos à lógica proposicional, qualquer estratégia utilizada na escolha da fórmula principal é justa, uma vez que qualquer fórmula é reduzida apenas às suas componentes atómicas num número finito de aplicações inversas das

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{A} \vdash \mathbf{A},\ B \qquad \mathbf{B},\ A \vdash \mathbf{B}}{A \Rightarrow B,\ A \vdash B}\Rightarrow_l}{A \Rightarrow B,\ \vdash B,\ \neg A}\neg_r}{A \Rightarrow B,\ \neg B \vdash \neg A}\neg_l}{A \Rightarrow B \vdash \neg B \Rightarrow \neg A}\Rightarrow_r$$

Tabela 4.12: Demonstração (curta) de $A \Rightarrow B \vdash \neg B \Rightarrow \neg A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{A} \vdash B,\ \mathbf{A}}{\vdash B,\ A,\ \neg A}\neg_r}{\neg B \vdash A,\ \neg A}\neg_l}{\vdash A,\ \neg B \Rightarrow \neg A}\Rightarrow_r \qquad \dfrac{\dfrac{\mathbf{B} \vdash \mathbf{B},\ \neg A}{\neg B,\ B \vdash \neg A}\neg_l}{B \vdash \neg B \Rightarrow \neg A}\Rightarrow_r}{A \Rightarrow B \vdash \neg B \Rightarrow \neg A}\Rightarrow_l$$

Tabela 4.13: Demonstração (longa) de $A \Rightarrow B \vdash \neg B \Rightarrow \neg A$.

regras de inferência. Já no caso da lógica de predicados a possibilidade de duplicação da fórmula principal faz com que seja possível expandir ramos da árvore de inferência infinitamente, e nesse caso pode acontecer que determinadas fórmulas do sequente nunca sejam escolhidas para decomposição (ver tabela 4.14).

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\forall x.\, P(x),\ P(?z),\ P(?y),\ P(?x),\ Q(a) \wedge Q(b)\ \vdash\ Q(a)}}{\forall x.\, P(x),\ P(?y),\ P(?x),\ Q(a) \wedge Q(b)\ \vdash\ Q(a)}\ \forall_l}{\forall x.\, P(x),\ P(?x),\ Q(a) \wedge Q(b)\ \vdash\ Q(a)}\ \forall_l}{\forall x.\, P(x),\ Q(a) \wedge Q(b)\ \vdash\ Q(a)}\ \forall_l$$

Tabela 4.14: Escolha injusta da fórmula principal (é ignorada a fórmula $Q(a) \wedge Q(b)$).

Por este motivo é conveniente que as regras que envolvem duplicação (como é o caso de $\forall_l$ e $\exists_r$) tenham prioridade mais baixa que todas as outras. Ainda assim é necessário garantir que existe justiça nas escolhas efectuadas também entre conjuntos de fórmulas cujas regras de inferência correspondentes envolvam duplicação. Uma solução consiste em atribuir a cada fórmula principal acabada de duplicar uma penalização sobre a sua prioridade anterior. No que diz respeito a regras de inferência que não envolvam duplicação podemos restringir-nos a avaliar o número de premissas tal como no caso das conectivas, o que significa que as regras $\forall_r$ e $\exists_l$ deverão ter prioridade máxima (poderá eventualmente ser vantajoso preferir as regras para conectivas com igual número de premissas, uma vez que não têm o trabalho de substituir a variável ligada por um parâmetro).

Alguns demonstradores, como FAUST e Folderol [29, 28], utilizam apenas estes critérios na selecção das fórmulas principais. Avaliar apenas o tipo de regras a utilizar, contudo, é uma heurística que deixa um pouco a desejar. A análise estrutural das fórmulas é bastante superficial, do que resulta uma classificação muito pouco refinada. Um possível melhoramento consiste em efectuar a escolha da fórmula principal, por forma a tentar obter rapidamente novos predicados, uma vez que tal significa uma nova oportunidade de terminar um nó com sucesso. Para isso podemos utilizar como critério de comparação a profundidade mínima de um predicado nas fórmulas. Assim a fórmula $(P \wedge Q) \wedge R$, que necessita de apenas uma decomposição para devolver um predicado (R), teria prioridade sobre $((P \wedge Q) \wedge (R \wedge S)) \wedge ((T \wedge U) \wedge (V \wedge X))$, que necessitaria de três decomposições para devolver um predicado. Este critério é utilizado apenas para desempatar entre fórmulas cujas regras correspondentes tenham igual prioridade.

4.2.2 Escolha do Sequente a Expandir

A ordem pela qual são expandidos os nós de uma árvore de inferência deve ter em conta critérios de eficiência e completude. Apresentamos de seguida algumas das vantagens e desvantagens dos métodos clássicos de construção em largura, construção em profundidade, e algumas variantes.

Construção em largura: Como o próprio nome indica, é efectuada uma construção nível a nível, para isso o sequente escolhido é sempre uma das folhas da árvore de inferência a menor profundidade. Na prática esta estratégia é habitualmente implementada com uma *fila*. A escolha do sequente a expandir é feita à cabeça da fila, e os sequentes resultantes da expansão adicionados à cauda da mesma. Este método tem a vantagem de ser completo, isto é, mesmo que haja ramos que possam ser expandidos infinitamente (o que pode acontecer em lógica de primeira ordem), é possível chegar a qualquer nó da árvore num número finito de passos. Como desvantagem, esta estratégia obriga a um consumo de memória muito grande uma vez que o número de folhas poderá crescer exponencialmente em relação à altura da árvore (no pior dos casos h^k onde h é a altura da árvore e k o maior número de permissas de uma regra de inferência).

Construção em profundidade: Segundo esta estratégia escolhemos sempre um dos sequentes a um nível mais profundo da árvore criada. A política seguida é *last in, first out*, ou seja, o próximo sequente a ser escolhido é sempre um dos últimos criados, e portanto a implementação pode ser feita à base de uma estrutura de dados *pilha*. Como cada nó (folha) da árvore pode ser descartado após ter sido terminado com sucesso, uma construção em profundidade necessita de manter em memória, a cada instante, uma quantidade linear de nós relativamente à profundidade máxima da árvore de demonstração. A grande desvantagem desta estratégia é o facto de não ser completa devido à possível existência de ramos de profundidade infinita. Assim um ramo pode ser expandido indefinidamente quando a análise de outros ramos terminaria o processo.

Construção heurística: Esta estratégia de construção não segue uma ordem pré-determinada para a expansão dos nós da árvore de inferência, em vez disso a escolha é efectuada segundo uma determinada heurística. A escolha desta heurística deverá ser feita com cautela pois existe o risco de se combinar o pior de cada uma das estratégias anteriores, o consumo de memória de uma construção em largura e a incompletude de uma construção em profundidade. O problema da incompletude é fácil de contornar, basta garantir uma escolha justa da fórmula principal, não apenas ao nível de cada sequente, mas a nível dos vários sequentes da árvore de demonstração. Quanto ao consumo de memória, no pior dos casos continuará a crescer exponencialmente com a altura da árvore [17].

Apesar da incompletude da construção em profundidade, esta é, de um modo geral, preferível às construções em largura e heurística, pois a elevada quantidade de memória necessária por estas duas últimas estratégias faz com que não sirvam senão para demonstrações bastante simples [29]. Existem, no entanto, alternativas que visam melhorar construções em profundidade.

Construção heurística em profundidade: Durante uma construção em profundidade, de cada vez que um sequente é expandido, os sequentes resultantes poderão ser colocados na pilha por forma a ficarem ordenados de acordo com uma determinada avaliação heurística (que poderá ser a comparação das fórmulas principais escolhidas em cada um). As melhorias conseguidas por este processo são normalmente pouco relevantes, uma vez que o número de sequentes alcançados pela heurística é muito limitado [32]. Esse alcance pode ser aumentado tomando para ordenação, não apenas os sequentes obtidos na última expansão, mas os k sequentes no topo da pilha (incluindo já os novos sequentes calculados), para um determinado k constante. Esta constante deverá ser escolhida por forma a manter um equilíbrio entre o proveito retirado da heurística, e o consumo extra de memória que esta implica. Se tomarmos $k = 0$ o processo resultante é a variante inicialmente descrita neste ponto, em que a heurística actua apenas sobre os filhos do nó expandido (e portanto o consumo de memória será idêntico ao da construção em profundidade), para $k = \infty$ o processo resultante é a *construção heurística* anteriormente descrita (e portanto o consumo de memória será idêntico ao de uma construção em largura).

Construção em profundidade iterada: Esta estratégia consiste simplesmente em sucessivas construções em profundidade com limites máximos de profundidade progressivamente relaxados. Esta estratégia é completa e, como é óbvio, apresenta o mesmo consumo de memória que construção em profundidade. Como contrapartida, em cada iteração serão recalculadas, árvores de inferência já determinadas nas iterações anteriores. Por outras palavras, a eficiêncial espacial desta estratégia de construção completa é conseguida à custa de perda de eficiência temporal [16, 32].

Relativamente aos limites de profundidade impostos durante construções em profundidade (e profundidade iteradas), é importante notar que a partir do momento em que um sequente é terminado por atingir esse limite, a árvore não será mais terminada com sucesso, ou seja, não iremos conseguir nessa iteração uma árvore de demonstração, isto porque para tal acontecer teriamos que terminar com sucesso todos os sequentes da mesma. A expansão dos restantes ramos pode, no entanto, terminar o processo de inferência com insucesso, ou seja podemos descobrir que o sequente inicial não é um teorema. Relembramos que para um nó terminar com insucesso é necessário que este não seja um axioma e todas as fórmulas estejam completamente decompostas. Por este motivo sequentes com fórmulas quantificadas por $\forall$ no antecedente ou por $\exists$ no consequente (que são duplicadas quando escolhidas como fórmulas principais) nunca terminarão com insucesso. Ramos que contenham estas fórmulas ou terminam com sucesso ou então são expandidos indefinidamente. Daqui retiramos, por exemplo, que se tentarmos demonstrar um sequente com $\forall$ à esquerda ou $\exists$ à direita, não existe qualquer vantagem em utilizar construção por profundidade iterada. Sabemos à priori que nenhum ramo irá terminar com insucesso portanto, quando um ramo atinge o limite de profundidade, nada se poderá concluir da expansão dos outros ramos.

4.2.3 Controlo da Unificação

Um aspecto ainda não mencionado acerca do cálculo de unificadores para terminar sequentes com sucesso, foi como proceder caso existam múltiplas soluções. Como exemplo, dado sequente $Q(?x, ?y) \vdash Q(a, b), Q(?y, f(b)), Q(?x, b)$ o termo no antecedente pode ser unificado com qualquer termo no consequente, do que resultam três *mgu*s distintos. Pode acontecer que apenas alguns ou algum destes unificadores permita determinar uma árvore de demonstração. A solução mais óbvia para este problema passa por escolher um deles, prosseguir com o processo de inferência e, caso este não termine com sucesso, efectuar *backtracking* e tentar uma nova escolha. Este processo levanta dois problemas, um deles a necessidade de memória extra para guardar o contexto necessário para o *backtracking*, o outro, o poder ter que refazer vários passos do processo de inferência já efectuados após instanciação com o unificador anterior. Evitar ou minorar estes problemas poderá passar por de alguma forma manter múltiplas soluções associadas a metavariáveis, em simultâneo, durante o processo de inferência, e será posteriormente alvo de estudo. Uma pequena optimização consiste em, dos *mgu*s resultantes de cada unificação, considerar apenas os mais gerais entre si. No exemplo anterior os *mgu*s resultantes de cada uma das unificações calculadas seriam $\sigma_1 = \{?x \mapsto a, ?y \mapsto b\}$, $\sigma_2 = \{?x \mapsto f(b), ?y \mapsto f(b)\}$ e $\sigma_3 = \{?y \mapsto b\}$, e portanto poderiamos desprezar o primeiro, uma vez que $\sigma_1 \precsim \sigma_3$.

É importante notar que não existe o risco de concluirmos erradamente que um sequente (teorema) é um não teorema, por não utilizarmos um unificador que o permita demonstrar. Poderiamos pensar que tal poderia acontecer se, após uma instanciação de metavariáveis com um unificador que não permita efectuar a demonstração, um dos sequentes terminasse com insucesso, mas isso é impossível. Se o sequente possui metavariáveis para serem instanciadas, então possui também a fórmula quantificada que pode ser infinitamente duplicada. Assim sendo, podemos por questões de eficiência, restringirmo-nos ao uso um subconjunto limitado de unificadores, com a garantia de que não serão, por esse motivo, obtidos resultados (conclusivos) incorrectos.

Outro pormenor importante diz respeito à forma como são instanciadas as metavariáveis após o cálculo de um unificador. Se forem mantidas, associadas a cada sequente, as metavariáveis geradas (como descrito na secção 4.1.4), podemos utilizar essa informação para ignorar sequentes que não contenham nenhuma das variáveis a substituir. Se utilizarmos uma estratégia em profundidade o processo de substituição pode ser tornado ainda mais eficiente. Como a estrutura utilizada para manter os sequentes é neste caso uma pilha, quando criamos um sequente com uma nova metavariável, sabemos que ela nunca irá ocorrer abaixo desse nível da pilha.

4.2.4 Implementação da Construção da Árvore de Inferência

A implementação estável do processo de inferência é efectuado quase na sua totalidade dentro do *monad* IO. A implementação é toda bastante directa, sem a utilização de quaisquer técnicas de programação mais interessantes, dignas de registo, seguindo mesmo um estilo de implementação quase *imperativo*. O objectivo dessa implementação foi o de simplificar ao máximo a obtenção de *feedback* por parte do demonstrador, por forma a simplificar a inspecção do processo de construção das árvores de inferência, mesmo enquanto o código não estivesse estabilizado.

Neste momento, está a ser reimplementada esta componente do demonstrador, de forma mais elegante, tendo como um dos objectivos uma maior separação do cálculo da árvore de inferência propriamente dito, e do tratamento do *output* para o utilizador.

Nesta versão de desenvolvimento são considerados dois tipos de ambientes (*monads*) para construção de demonstrações. Os mais simples, são *monads* do tipo Ded, que têm como função encapsular pormenores específicos de cada estratégia de dedução, como por exemplo, a forma como são tratados unificadores múltiplos ou as heurísticas utilizadas na escolha da fórmula principal, e também de construirem um *trace* durante o processo de demonstração. Neste momento o *monad* Ded é definido como um *monad* RWS, isto é, um *monad* que combina as características dos *monads Reader*, *Writer* e *State*. A componente *Writer* é o que lhe permite criar *streams* de *feedback* que serão eventualmente consumidas fora do mesmo, as componentes *Reader* e *State* permitem guardar opções (definição de estratégias, heurísticas, e outros parâmetros), constantes no primeiro caso e mutáveis no segundo, e ainda, no caso da componente *State*, recolher alguma informação sobre o desenrolar do processo (número de iterações efectuadas, profundidade máxima atingida, tamanho da pilha de demonstração etc.).

```
type Ded = RWS DedReadOnly DedFeedback DedState
```

O resultado da construção de uma árvore de dedução pode ser *sucesso* caso se demonstre o sequente pretendido, *insucesso* caso se mostre que o sequente é um não teorema, ou então indefinido se não conseguirmos chegar a nenhuma conclusão.

```
type DedResult = Success | NoSuccess | Indetermination
```

É importante notar que o terceiro caso pode acontecer por vários motivos, e está dependente das restrições que impomos no processo de dedução. Durante este processo poderão ser levantadas e capturadas excepções. Como vimos na implementação de métodos de unificação (capítulo 3), estas são normalmente modeladas pelo *monad Error*. Para combinarmos um *monad* de *Error* com o nosso *monad* para deduções utilizamos um *monad transformer*, neste caso, mais concretamente o *monad transformer ErrorT*.

```
type DedCtrl = ErrorT DedError Ded
```

O resultado é um ambiente para construção de deduções onde, para além de tudo o que referimos anteriormente, podemos ainda levantar e capturar excepções.

Outro aspecto importante relacionado com a nova implementação do processo de inferência, tem a ver com adaptações efectuadas à representação das fórmulas para optimizar (principalmente) o processo de avaliação das mesmas. Uma das heurísticas referidas para escolha da formula principal, foi, por exemplo, a profundidade mínima de um predicado. Na versão estável do demonstrador cada vez que são obtidas novas fórmulas por decomposição, este valor tem que ser calculado e associado à fórmula em causa. Seria mais eficiente associar à partida a todas as

subfórmulas de cada uma das fórmulas, uma etiqueta com estes valores. Assim a definição do tipo de dados *Fórmula* deverá ser alterada para:

```
data Form l =
   Prd{prdId    :: PrdId,  prdArity :: Arity, prdTerms :: [Term], label :: l}
 | Con{conSym   :: ConSym, conArity :: Arity, conForms :: [Form], label :: l}
 | Qnt{qntSym   :: QntSym, qntVarId :: String, qntForm  :: Form,  label :: l}
```

Notas Finais

Estudámos ao longo deste trabalho, tal como nos haviamos proposto, sistemas lógicos de dedução para lógica proposicional e de predicados de primeira ordem, e automação de demonstrações. O método de automação focado consiste na construção de demonstrações segundo um processo de *aplicação inversa das regras de inferência* na variante de *sistemas de dedução de Gentzen* conhecida por *cálculo de sequentes restritos*. Foram explorados vários métodos de unificação, uma das tarefas fundamentais no processo demonstração automática em lógica de predicados, entre eles, algoritmo de Robinson, algoritmo de Martelli e Montanari e indexação de termos. Foram ainda apresentadas várias técnicas e heurísticas para construção de árvores demonstração. Este estudo foi acompanhado da implementação em Haskell de um sistema de dedução automática cujos os detalhes de implementação mais relevantes foram já descritos. De momento o demonstrador está preparado para funcionar apenas com lógicas proposicional e de predicados de primeira ordem clássicas, no entanto está construído por forma a que seja simples a adaptação a outras lógicas, definidas em sistemas de Gentzen, que satisfaçam as propriedade de subfórmula e antecessor (secção 1.2). De seguida apresentamos como definir uma lógica para o demonstrador. Começamos pela parte sintáctica dessa definição, a *linguagem lógica*.

Definição da Linguagem Lógica

A sintaxe da lógica do demonstrador é definida no módulo Definitions.Logic (ver tabela 4.15). Em primeiro lugar deverão ser identificadas as conectivas e quantificadores utilizados (quaisquer identificadores servem deste que sejam válidos como construtores de valores em Haskell). Para lógica de predicados de primeira ordem definimos as conectivas `NEG`, `AND`, `OR` `IMP` e `EQV` (negação, conjunção, disjunção, implicação e equivalência, respectivamente), e os quantificadores `FA` e `EX` (universal e existencial, respectivamente). De seguida deverá ser definida uma tabela para as conectivas, indicando em cada linha uma conectiva, a respectiva representação da conectiva, a forma como é utilizada (em prefixo, posfixo, ou infixo) e a sua prioridade. Esta tabela será utilizada por exemplo, na definição do *parser* e do *printer* para ler e escrever fórmulas. De forma análoga, deverá ser definida uma tabela para os quantificadores, mas esta apenas necessita de ter em cada linha o quantificador e a respectiva representação.

```
-- DEFINICAO LOGICA
module Definitions.Logica where

-- Indicar Conectivas e Quantificadores (comecar com maiúsculas):
data ConSym = NEG | AND | OR | IMP | EQV
data QntSym = FA  | EX

-- Tabela de Conectivas:
-- Cada linha da tabela deverá conter os seguintes campos,
-- 1. Conectiva  (definida anteriormente)
-- 2. Sintaxe    (string correspondente),
-- 3. Aplicação  (prefix, infix ou postfix -> PRECON, INCON, POSCON),
-- 4. Prioridade (0-inf, 0 prioridade mais alta)

conTable =
 [(NEG ,  "~"   , PRECON , 2),
  (AND ,  "&"   , INCON  , 4),
  (OR  ,  "|"   , INCON  , 6),
  (IMP ,  "=>"  , INCON  , 8),
  (EQV ,  "<=>" , INCON  , 10)]

-- Tabela de Quantificadores:
-- Cada linha da tabela deverá conter os seguintes campos
-- 1. Quantificador  (definido anteriormente)
-- 2. Sintaxe        (string correspondente),

qntTable =
 [(FA , "\\FA"),
  (EX , "\\EX")]
```

Tabela 4.15: Definição de uma linguagem lógica para o demonstrador.

Analisador Sintático

O analisador sintático (*parser*) foi implementado um usando a biblioteca de combinadores *Parsec* [21]. A sintaxe para representação de fórmulas no demonstrador é apresentada na tabela 4.16 em notação Extended Backus-Naur Form (EBNF):

⟨formula⟩ ::= ⟨formulaQnt⟩ | ⟨formulaCon⟩ | ⟨formulaPrd⟩

⟨formulaQnt⟩ ::= ⟨simboloQnt⟩ ⟨identVar⟩ '.' ⟨formula⟩
⟨formulaCon⟩ ::= ⟨formulaConPre⟩ | ⟨formulaConPos⟩ | ⟨formulaConIn⟩
⟨formulaConPre⟩ ::= ⟨simboloCon⟩ ⟨formula⟩
⟨formulaConPos⟩ ::= ⟨formula⟩ ⟨simboloCon⟩
⟨formulaConIn⟩ ::= ⟨formula⟩ ⟨simboloCon⟩ ⟨formula⟩
⟨formulaPrd⟩ ::= ⟨identPrd⟩ [⟨listaTermos⟩]

⟨termo⟩ ::= ⟨metavar⟩ | ⟨parametro⟩ | ⟨varligada⟩ | ⟨função⟩

⟨metavar⟩ ::= '?'⟨identVar⟩
⟨parametro⟩ ::= '!'⟨identPar⟩⟨listaVar⟩
⟨varligada⟩ ::= ⟨indentVar⟩
⟨função⟩ ::= ⟨identFun⟩⟨listaTermos⟩

⟨listaTermos⟩ ::= '['⟨termo⟩ [',' ⟨termo⟩] ']'
⟨listaVar⟩ ::= '['⟨identVar⟩ [',' ⟨identVar⟩] ']'

⟨identPrd⟩ ::= ⟨letraMaiuscula⟩[⟨letra⟩]
⟨identVar⟩ ::= ⟨letraMinuscula⟩[⟨letra⟩]
⟨identPar⟩ ::= ⟨letraMinuscula⟩[⟨letra⟩]
⟨identFun⟩ ::= ⟨letraMinuscula⟩[⟨letra⟩]

Tabela 4.16: Definição da sintaxe para fórmulas no demonstrador implementado.

A definição de ⟨simboloCon⟩ e ⟨simboloQnt⟩, ou seja dos possíveis símbolos para conectivas e quantificadores, é gerada automaticamente a partir do módulo que descreve a linguagem lógica a utilizar. Ao contrário do que é habitual são utilizados parêntesis rectos e não curvos para a lista de argumentos das funções e predicados. O objectivo é simplificar a leitura das fórmulas, diferenciando entre listas de argumentos e os parêntesis curvos inseridos para agrupar subfórmulas. Também com o objectivo de simplificar a leitura das fórmulas, é exigido que os identificadores de predicados comecem por letras maiúsculas, enquanto que os identificadores de termos (funções, metavariáveis, parâmetros e variáveis ligadas) começam obrigatoriamente por letras minúsculas. A possibilidade de introdução de metavariáveis e parâmetros existe apenas para efeitos de teste, devendo as fórmulas dos sequentes a demonstrar ser fechadas.

São exemplos de termos sintacticamente válidos:

`?x`	– metavariável x
`!a`	– parâmetro a
`!a[x,y]`	– parâmetro a, com dependências x e y
`b`	– constante b (função zero-ária)
`f[?x, g[a]]`	– função aplicada à metavariável x e à função g[a]

São exemplos de fórmulas válidas, assumindo definidas as conectivas e quantificadores para lógica clássica anteriormente descritas:

`P`	– fórmula atómica proposicional
`P => Q`	– fórmula não atómica proposicional
`R | Q => ~S & Q`	– idem
`~(P | Q) <=> R`	– idem
`Q[?x]`	– fórmula atómica de 1ª ordem (aberta)
`\FA x . Q[x]`	– fórmula atómica de 1ª ordem (fechada)
`\FA x . \EX y . Q[x] => R(x,y)`	– idem
`(\FA x . Q[x]) => (\EX x . P[f[x]])`	– idem

Definição das regras de Inferência

O método utilizado para a descrição das regras de inferência consiste na definição de tabelas de dedução que serão interpretadas como listas de acções, tal como descrito no capítulo 4, subsecção 4.1.4. Na tabela 4.17 é apresentada a definição das regras de inferência para lógica de primeira ordem clássica.

Definição de Teorias

A definição da linguagem lógica, do *parser* para essa linguagem e do sistema de dedução correspondente, são todos efectuados durante a compilação. O principal problema de uma definição em tempo de execução seria a necessidade de parameterização do programa em função destes. As soluções até agora consideradas (que passariam pela utilização de *parâmetros implícitos*, uma extensão do compilador GHC, ou encapsulamento de todo o processo em *monads*) estão longe de ser elegantes. Teorias, por outro lado, podem facilmente ser definidas em tempo de execução. Na prática são apenas conjuntos de fórmulas que deverão ser acrescentados ao antecedente dos sequentes a demonstrar. Neste momento a definição de uma teoria consiste apenas no seu nome e no conjunto de fórmulas que a define (ver tabela 4.18).

```
-- DEFINICAO DAS REGRAS DE INFERÊNCIA
module Definitions.Rules where
import ...

-- Regras de Inferencia para Conectivas:
-- Definir para regra (lado, conectiva) a lista de premissas correspondentes.
-- Omitir fórmulas secundárias.

redCon L NEG [a]   = [ [ ]|-[a]               ]
redCon R NEG [a]   = [ [a]|-[ ]               ]

redCon L AND [a,b] = [ [a,b]|-[ ]             ]
redCon R AND [a,b] = [ [   ]|-[a] , [ ]|-[b]  ]

redCon L OR  [a,b] = [ [a]|-[   ] , [b]|-[ ]  ]
redCon R OR  [a,b] = [ [ ]|-[a,b]             ]

redCon L IMP [a,b] = [ [ ]|-[a]   , [b]|-[ ]  ]
redCon R IMP [a,b] = [ [a]|-[b]               ]

redCon L EQV [a,b] = [ [a,b]|-[ ] , [ ]|-[a,b] ]
redCon R EQV [a,b] = [ [a  ]|-[b] , [b]|-[a  ] ]


-- Regras de Inferencia para Quantificadores:
-- Acções possíveis,
-- ``dup''    <- duplicar fórmula original
-- ``|a!x|''  <- substituir variável ligada de a por um parâmetro x
-- ``|a?x|''  <- substituir variável ligada de a por uma metavariável x

redQnt L FA x a  = [ [dup , a|?|x] |- []      ]
redQnt R FA x a  = [ []            |- [a|!|x] ]

redQnt L EX x a  = [ [a|!|x] |- []            ]
redQnt R EX x a  = [ []      |- [dup , a|?|x] ]
```

Tabela 4.17: Definição de uma linguagem lógica para o demonstrador.

```
# Indicar Nome da Teoria

theoryName = "Teoria de  Grupos"


# Indicar Axiomatização da Teoria

\FA x . P[e,x,x] & P[x,e,x]                              # elemento neutro

\FA x . \EX y . P[x,y,e] & P[y,x,e]                      # elemento inverso

\FA x . \FA y . \FA z . \FA v . \FA u . \FA w .
      (P[x,y,u] & P[y,z,v]=>P[u,z,v]) <=> P[x,v,w])  # associatividade


\FA x . \EX y . P[x,y,f[x,y]]                            # operação de grupo

\FA x . \EX y . \FA u . \EX v .                          # garantir que operação
     (P[x,y,u] & P[x,y,v]) => EQ[u,v]                    # tem resultado único

\FA x . \EX y . \FA u . \EX v .                          # garantir resultados
     (P[x,y,u] & EQ[u,v]) => P[x,y,v]                    # iguais com termos iguais
```

Tabela 4.18: Definição de uma teoria de grupos.

Trabalho Futuro

Neste momento as prioridades a nível de implementação são completar uma reimplementação mais elegante da construção da árvore de demonstração bem como alteração das estruturas de dados base (*fórmulas* e eventualmente *termos*) para suporte de *labels* que permitam a introdução de informação extra que possa ser utilizada na sua avaliação, tal como descrito no capítulo 4. Concluídas estas alterações o código deverá ser estabilizado para se proceder à análise de gestão de memória (*profiling*) e a uma avaliação experimental cuidada. Para o efeito está a ser implementado um *parser* que irá permitir a utilização dos casos de teste definidos na biblioteca *"Thousands of Problems for Theorem Provers"* (TPTP) [36]. Outros objectivos de curto prazo são a implementação e teste de métodos de indexação de termos e adaptação dos processos de unificação para funcionarem com classes de teorias de unificação, e não apenas termos.

Bibliografia

[1] Stephen Adams. Efficient sets - a balancing act. *J. Funct. Program.*, 3(4):553–561, 1993.

[2] Lex Augusteijn. Sorting morphisms. In *Advanced Functional Programming*, páginas 1–27, 1998.

[3] F. Baader e W. Snyder. Unification theory. In A. Robinson e A. Voronkov, editores, *Handbook of Automated Reasoning*, volume I, chapter 8, páginas 445–532. Elsevier Science, 2001.

[4] Arthur Baars, S. Doaitse Swierstra e Andres Löh. UU AG system user manual, 2003.

[5] Richard L. Call. Constructing sequent rules for generalized propositional logics. *Notre Dame Journal of Formal Logic*, 1984.

[6] Oege de Moor, Kevin Backhouse e S. Doaitse Swierstra. First class attribute grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. Special Issue: Attribute grammars and Their Applications.

[7] D. M. Gabbay. Elements of algorithmic proof. In S. Abramsky, D. M. Gabbay e T. S. E. Maibaum, editores, *Handbook of Logic in Computer Science: Background - Computational Structures (Volume 2)*, páginas 311–413. Clarendon Press, Oxford, 1992.

[8] Jean H. Gallier. *Logic for computer science: foundations of automatic theorem proving.* Harper & Row Publishers, Inc., 1985.

[9] Harald Ganzinger, Robert Nieuwenhuis e Pilar Nivela. Context trees. *Lecture Notes in Computer Science*, 2083, 2001.

[10] Peter Graf. Substitution tree indexing. Relatório Interno MPI-I-94-251, Max Planck Institut für Informatik, Saarbruecken, 1994.

[11] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[12] Thomas Hillenbrand. Citius altius fortius: Lessons learned from the theorem prover waldmeister, 2003.

[13] Graham Hutton. Fold and unfold for program semantics. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34(1), páginas 280–288. ACM Press, New York, 1998.

[14] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[15] Stephen C. Kleene. *Logique Mathématique*. Librarie Armand Colin, 1971.

[16] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[17] Richard E. Korf. Artificial intelligence search algorithms. In *Algorithms and Theory of Computation Handbook*. CRC press, 1999.

[18] Ramayya Kumar, Thomas Kropf e Klaus Schneider. First steps towards automating hardware proofs in hol. In *TPHOLs*, páginas 190–193, 1991.

[19] R. Lämmel e J. Visser. A Strafunski Application Letter. In V. Dahl e P. Wadler, editores, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, páginas 357–375. Springer-Verlag, Janeiro 2003.

[20] Ralf Lämmel e Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Março 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[21] Daan Leijen e Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Relatório Interno UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[22] Andres Löh. *Exploring Generic Haskell*. Tese de Doutoramento, University of Utrecht, 2004.

[23] Alberto Martelli e Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[24] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov e Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. *Lecture Notes in Computer Science*, 2083, 2001.

[25] U. Norell e P. Jansson. Polytypic programming in haskell, 2004.

[26] Hans Jurgen Ohlbach. Abstraction tree indexing for terms. In *European Conference on Artificial Intelligence*, páginas 479–484, 1990.

[27] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[28] L. C. Paulson. Designing a theorem prover. In S. Abramsky, D. M. Gabbay e T. S. E. Maibaum, editores, *Handbook of Logic in Computer Science: Background - Computational Structures (Volume 2)*, páginas 415–475. Clarendon Press, Oxford, 1992.

[29] R. Kumar, T. Kropf e K. Schneider. Integrating a First-Order Automatic Prover in the HOL Environment. In M. Archer, J.J. Joyce, K.N. Levitt e P.J. Windley, editores, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, páginas 170–176, Davis, California, 1991. IEEE Computer Society Press.

[30] Fethi Rabhi e Guy Lapalme. *Algorithms; A Functional Programming Approach*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[31] I.V. Ramakrishnan, R. Sekar e A. Voronkov. Term indexing. In A. Robinson e A. Voronkov, editores, *Handbook of Automated Reasoning*, volume II, chapter 26, páginas 1853–1964. Elsevier Science, 2001.

[32] A. Reinefeld e T. A. Marsland. Enhanced iterative-deepening search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(7):701–710, 1994.

[33] M. Ryan e M. Sadler. Valuation systems and consequence relations. In S. Abramsky, D. M. Gabbay e T. S. E. Maibaum, editores, *Handbook of Logic in Computer Science: Background - Mathematical Structures (Volume 1)*, páginas 1–78. Clarendon Press, Oxford, 1992.

[34] Jonathan Seldin. Manipulating proofs. Department of Mathematics, Concordia University, Montreal, Quebec, Canada.

[35] Manuela Sobral. *Álgebra*. Universidade Aberta, 1996.

[36] G. Sutcliffe, C. Suttner e T. Yemenis. The tptp problem library. In A. Bundy, editor, *Automated Deduction-CADE-12*, páginas 252–266. Springer, Berlin, Heidelberg, 1994.

[37] Andrei Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, páginas 13–28. Springer-Verlag, 2001.

[38] Christoph Weidenbach. System description: Spass version 1.0.0. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, páginas 378–382, London, UK, 1999. Springer-Verlag.

[39] Larry Wos e Gail W. Pieper. *A Fascinating Country in the World of Computing*. World Scientific Publishing, 2000.