Pedro Henrique e Figueiredo Quaresma de Almeida
Departamento de Matemática
Universidade de Coimbra

Construção Modular de Sistemas de Dedução

Departamento de Informática
Universidade do Minho
1998

Dissertação apresentada à Escola de Engenharia da Universidade do Minho, para a obtenção do Grau de Doutor em Informática (especialidade de Fundamentos da Computação).

Gostaria de expressar os meus agradecimentos ao Departamento de Matemática da Universidade de Coimbra, o meu local de formação e trabalho, pelas condições que me foram dadas ao longo destes anos. Quero também agradecer ao Departamento de Informática da Universidade do Minho e ao Centro de Informática e Sistemas da Universidade de Coimbra pelos condições de trabalho que me proporcionaram durante a preparação desta dissertação.

Quero expressar o meu profundo agradecimento ao Professor Doutor José Manuel Esgalhado Valença pela sua preciosa orientação e pelo seu apoio amigo ao longo de todos estes anos.

A todos aqueles com que convivi, tanto no meio universitário como fora dele, a todos o meu agradecimento pois todos eles contribuiram para a realização deste trabalho. Nem só de trabalho se faz uma vida.

Conteúdo

Introdução

A teoria da dedução ("proof theory") tem como objecto de estudo a formalização das demonstrações matemáticas, assim como a análise da estrutura dessas mesmas demonstrações. A teoria da dedução teve a sua criação no princípio deste século, quando Hilbert tentou dar resposta aos problemas que entretanto tinham surgido quanto aos fundamentos da matemática. Hilbert propôs que se criasse uma axiomatização de todo o conhecimento matemático, e que se demonstrasse a coerência desse conjunto de axiomas (Oliveira, 1979; Schütte, 1977; Takeuti, 1975). A demonstração da coerência dos axiomas seria realizada no âmbito de um nova teoria matemática, que Hilbert designou por "Bweistheorie", que no presente trabalho se vai designar por, teoria da dedução. Segundo Hilbert a correcção dos métodos da teoria da dedução teria de estar fora de questão, para tal ele propôs que só fossem usados "finitistic methods", segundo a interpretação de Pohlers (1980) tratar-se-iam de métodos de inferência de carácter combinatorial, sobre domínios finitos.

Os teoremas da incompletude de Gödel demonstrados por este matemático nos anos 30 estabeleceram que o programa de Hilbert tal como tinha sido formulado, não era realizável. No entanto o desenvolvimento da teoria da dedução prosseguiu sobre duas direcções distintas, permitindo a obtenção de muitos resultados importantes. A primeira dessas direcções foi estabelecida pelo relaxar das condições impostas aos métodos a utilizar, permitindo-se então a utilização de certas formas de indução, veja-se a título de exemplo a demonstração da coerência da teoria elementar dos números por Gentzen (Szabo, 1969). A outra das direcções tomadas foi dada pelo considerar de sistemas lógicos nos quais os métodos estritamente finitos se aplicam. (Barwise, 1977; Feferman, 1994; Girard, 1987; Pohlers, 1980; Schütte, 1977; Takeuti, 1975).

Um primeiro resultado importante quanto à aplicabilidade dos métodos finitos para o desenvolvimento de demonstrações foi obtido por Herbrand em 1930 (Chang & Lee, 1973). Herbrand desenvolveu um algoritmo, o cálculo do Universo de Herbrand, que de uma forma sistemática procura uma interpretação capaz de tornar falsa uma dada fórmula, demonstrando por refutação

a validade de uma dada fórmula. No entanto a grande complexidade, deste método, levou a que a sua aplicação fosse muito difícil, mesmo com a sua implementação em computador feita em 1960 por Gilmore, e posteriormente melhorada por Davis e Putman (Chang & Lee, 1973).

Em 1965 Robinson desenvolveu o método de resolução (Chang & Lee, 1973). O método de resolução que consiste na aplicação repetitiva de uma só regra de inferência, tem uma implementação em computador fácil e eficiente. A sua utilização para a lógica proposicional ("ground resolution") e a sua extensão para a lógica de primeira ordem, através da incorporação do algoritmo de unificação deu origem ao primeiro conjunto de programas eficientes, capazes de desenvolver demonstrações de uma forma automática (Gallier, 1987; Chang & Lee, 1973).

A estes sistemas automáticos de demonstração, baseados num dado algoritmo de demonstração, e para uma dada linguagem lógica, vão-se contrapor os sistemas semi-automáticos de demonstração. Nestes sistemas pretende-se que os utilizadores possam interactuar com o sistema influenciando o desenvolver das demonstrações. Para tal o utilizador passa a dispor de uma linguagem de programação de estratégias de demonstração, cujos operadores foram designados por tácticas e tacticais[1] por M. Gordon, A. Milner, e C. Wadsworth (1979). As tácticas implementam as regras de inferência, os tacticais aceitam os primeiros como argumento e permitem a programação de estratégias de demonstração.

A evolução tecnológica entretanto verificada nos sistemas computacionais levou ao desenvolvimento de sistemas designados por meta-sistemas, nos quais é possível implementar sistemas de dedução específicos. Nestes sistemas é possível definir uma linguagem lógica, um conjunto de axiomas e as regras de inferência para essa mesma linguagem, assim como os mecanismos necessários para a automatização das demonstrações no sistema definido. Como exemplo de tais sistemas temos os sistemas: *Isabelle* (Kalvala, 1994; Paulson, 1993a; Paulson, 1993c; Paulson, 1993b), *ICLE* (Dawson, 1991; Dawson, 1992), *ELF* (Harper *et al.*, 1993; Pfenning, 1991), *2OBJ* (Stevens & Hobley, 1993), entre outros. No apêndice C apresenta-se uma listagem de sistemas de dedução que inclui os sistemas deste tipo.

A crescente complexidade dos sistemas lógicos em consideração fez surgir a necessidade da construção modular de sistemas de dedução. Embora muitos dos sistemas actuais tenham a capacidade de construção de sistemas de dedução por adição de sub-sistemas, sente-se a necessidade do desenvolvimento de mecanismos de construção mais poderosos do que os existentes.

A definição do conceito de Instituição (Goguen & Burstall, 1985; Goguen & Burstall, 1992) surge nesse contexto. Pretende-se descrever os sistemas lógicos de uma forma bastante abstracta para assim poder estudar as suas propriedades genéricas. Goguen e Burstal (1985; 1992) definem uma

[1]Por analogia com as funções e os funcionais

instituição como um quadruplo $\langle \mathcal{ASS}, \mathrm{Mod}, \mathrm{Sen}, \models_\Sigma \rangle$ constituído pela categoria (Herrlich & Strecker, 1973; MacLane, 1971) $\mathcal{ASS}$ das assinaturas e dos morfismos de assinaturas, esta componente quando instanciada, descreve a sintaxe de uma dada linguagem lógica; por um functor (Herrlich & Strecker, 1973; MacLane, 1971) $\mathrm{Mod} : \mathcal{ASS} \rightarrow \mathsf{Cat}^{\mathrm{op}}$, que para uma dada assinatura nos dá todos os seus modelos e os morfismos entre eles; por um functor $\mathrm{Sen} : \mathcal{ASS} \rightarrow \mathsf{Cat}$, que para uma dada Σ-assinatura nos dá o conjunto de todas as Σ-fórmulas, isto é, fórmulas formadas com símbolos da assinatura e todas as Σ-demonstrações; e por uma relação de satisfação $\models_\Sigma \subseteq \mathrm{Obj}(\mathrm{Mod}(\Sigma)) \times \mathrm{Obj}(\mathrm{Sen}(\Sigma))$, entre modelos, os objectos da categoria $\mathrm{Mod}(\Sigma)$, e fórmulas, os objectos da categoria $\mathrm{Sen}(\Sigma)$.

No presente trabalho propõe-se a construção de uma estrutura que, estando próxima das instituições, esquece a componente referentes aos modelos e realça a componente da dedução. Em primeiro lugar, e influenciado pelo trabalhos referentes às ligações entre a lógica e a teoria das categorias (Lambek, 1958; Lambek, 1969; Lambek, 1971; Lambek & Scott, 1988; Szabo, 1976; Szabo, 1978) define-se a categoria dos sistemas de deduções $\mathcal{SD}$. A categoria $\mathcal{SD}$ tem como objectos, conjuntos de sistemas de dedução sobre uma mesma assinatura Σ, definidos como sendo grafos com algumas setas específicas, e em que os nós são Σ-fórmulas e as setas são Σ-demonstrações. Os morfismos de $\mathcal{SD}$, são morfismos de sistemas de dedução, definidos de forma a preservar as demonstrações entre Σ-sistemas de dedução. De seguida e tendo como influência as instituições e as Π-instituições (Fiadeiro & Sernadas, 1988) define-se um sistema lógico, como sendo um triplo $\langle \mathcal{ASS}, \mathcal{SD}, \mathrm{Sd} \rangle$, com $\mathrm{Sd} : \mathcal{ASS} \rightarrow \mathcal{SD}$ um functor que transforma assinaturas em conjuntos de sistemas de dedução e morfismos de assinaturas em morfismo de sistemas de dedução. Pretende-se deste modo ter uma forma abstracta de poder descrever os sistemas de dedução, mantendo uma relação estreita entre a componente sintáctica, definidora da linguagem lógica, e a componente referente aos sistemas de dedução sobre essa mesma linguagem. Demonstra-se que a categoria $\mathcal{SD}$ é co-completa, estabelecendo deste modo um resultado que nos permite utilizar o cálculo de co-limites para a construção de sistemas de dedução, providenciando deste modo um conjunto de mecanismos para a construção de sistemas de dedução que vai muito além da soma de sistemas de dedução.

É estudada a possibilidade de aplicação destes conceitos, utilizando o sistema de demonstração automática *2OBJ*. Este sistema está baseado na linguagem de programação *OBJ3* (Goguen, 1988; Goguen & Winkler, 1988; Goguen *et al.*, 1992) o que lhe dá uma capacidade de construção modular e parametrizável de sistemas de dedução superior aos outros sistemas (Goguen *et al.*, 1992). Conclui-se no entanto que, para uma completa aplicação das construções estudadas, seria necessário estender a linguagem com os construtores necessários para o cálculo de co-limites.

É ainda analisada, brevemente, a possibilidade de estender o presente

trabalho, no que diz respeito à combinação de diferentes sistemas lógicos. São introduzidos os mecanismos de fibrilação de lógicas (Sernadas *et al.*, 1997a), e de sincronização de lógicas (Sernadas *et al.*, 1997b; Sernadas *et al.*, 1997c), sendo abordada a possibilidade de incorporar esses mecanismos no âmbito do presente trabalho.

Em apêndice apresenta-se a informação que embora importante para uma melhor compreensão do texto não foi aí incluída para o não sobrecarregar.

No capítulo 1 apresentam-se as bases teóricas em que assentam os sistemas de demonstração automática e semi-automática de teoremas.

Começa-se por descrever os vários formalismos mais usuais para a construção de sistemas de dedução, nomeadamente os sistemas Hilbertianos (Gallier, 1987; Hamilton, 1991; Szabo, 1969), os sistemas de dedução natural (Bibel & Eder, 1993; Szabo, 1969; A.S.Troelstra & Dalen, 1988), e os cálculos de sequências dedutivas (Gallier, 1987; Oliveira, 1979; Szabo, 1969; van Dalen, 1980).

De seguida descrevem-se vários procedimentos computacionais usados na automatização, total ou parcial, das demonstrações, nomeadamente a construção do Universo de Herbrand (Chang & Lee, 1973), o método de resolução (Chang & Lee, 1973), a definição do isomorfismo de Curry-Howard (Barendregt, 1981; Church, 1941; Curry & Feys, 1968; Girard *et al.*, 1988; Howard, 1980; Lambek & Scott, 1988), e as tácticas e os tacticais (Gordon *et al.*, 1979; Paulson, 1990; Paulson, 1992).

O capítulo acaba com uma proposta de classificação para os diferentes tipos de sistemas computacionais existentes, estabelecendo as diferenças entre eles pela sua aplicabilidade, genéricos ou específicos, e pelo seu modo de interacção com o utilizador, interactivos ou não interactivos. A descrição sumária, de muitos dos sistemas existentes actualmente é dada no apêndice C, isto dado que a listagem é demasiado extensa para ser incluída neste capítulo.

No capítulo 2 apresenta-se o problema que, de certa forma, motivou a realização deste trabalho, ou seja o problema da demonstração de correcção parcial de programas, utilizando sistemas de demonstração automática de teoremas.

Para tal definiu-se a linguagem $LPCG$, uma linguagem do tipo procedimental com comandos guardados (Alagič & Arbib, 1978; Gries, 1981; Meyer, 1990; Quaresma, 1995), definiu-se ainda o sistema de dedução $LPCG_{LK}$, um sistema de dedução para asserções de dedutibilidade, baseado numa definição axiomática da semântica da linguagem $LPCG$ (Clint & Hoare, 1972; Cousot, 1990; Hoare, 1971; Hoare, 1972; Hoare, 1973; Quaresma, 1995).

Para poder avaliar as potencialidades, assim como as dificuldades, no desenvolvimento de demonstrações automáticas, ou semi-automáticas, da correcção parcial de programas em $LPCG_{LK}$, implementou-se o sistema $LPCG_{LK}$ nos sistemas computacionais *Isabelle* e *2OBJ*.

O capítulo finaliza com a apresentação de alguns exemplos de demonstrações de correcção parcial de programas, demonstrações essas desenvolvidas nos sistemas já referidos, e com uma discussão dos problemas encontrados, assim como das suas possíveis soluções.

No capítulo 3 desenvolvem-se as bases teóricas necessárias a uma possível solução para o problema da construção modular de sistemas de dedução.

Pretende-se construir uma estrutura categorial que permita descrever os sistemas de dedução, assim como as construções entre sistemas de dedução.

Para poder descrever categorialmente os sistemas de dedução define-se o conceito de categoria dos Σ-sistemas de dedução como sendo uma categoria cujos objectos são os Σ-sistemas de dedução, isto é, sistemas de dedução, cujos objectos são Σ-fórmulas e cujas setas são Σ-deduções (Lambek & Scott, 1988; Szabo, 1976; Szabo, 1978), e cujos morfismo são morfismo de Σ-sistemas de dedução, definidos de forma a preservar as demonstrações entre Σ-sistemas de dedução.

Para poder descrever as transformações entre sistemas de dedução define-se o conceito de sistema lógico, como sendo um triplo $\langle \mathcal{ASS}, \mathcal{SD}, \mathrm{Sd} \rangle$, em que $\mathcal{ASS}$ é a categoria das assinaturas, $\mathcal{SD}$ é a categoria dos Σ-sistemas de dedução, e Sd é um functor que para uma dada assinatura Σ dá origem ao conjunto dos Σ-sistemas de dedução. A definição de sistema lógico está baseada na definição de instituição, mais concretamente na definição de Π-instituição, difere no entanto destas na medida em que pretende incorporar o trabalho de Lambek (1958; 1969; 1971), de Lambek e Scott (1988), assim como o trabalho de Szabo (1976; 1978), no que diz respeito ao tratamento categorial de sistemas de dedução.

Demonstra-se que a categoria $\mathcal{SD}$ é co-completa, como tal, pode-se efectuar a construção de sistemas de dedução através do cálculo de co-limites.

O capítulo termina com a apresentação de um exemplo, em que se procede à construção do sistema de dedução SD_5, para a demonstração de correcção parcial de programas que manipulam pilhas de naturais através da construção de um quadrado co-cartesiano ("push-out").

No capítulo 4 descreve-se a construção do sistema de dedução SD_5 no meta-sistema computacional de demonstração semi-automática de teoremas *2OBJ*. O sistema *2OBJ* está baseado na linguagem de programação genérica *OBJ3*, cujas capacidades de construção modular e parametrizável de programas podem ser usadas para a construção modular e parametrizável de sistemas de dedução em *2OBJ*.

É feita uma descrição das capacidades do *OBJ3*, na sua última versão, e apresenta-se a construção do módulo `SD5`, módulo que implementa o sistema de dedução SD_5. Embora se possa efectuar a construção de `SD5` de uma forma modular, as capacidade de construção modular e parametrizável do *OBJ3* não são suficientes para se efectuar a construção de `SD5` através de um quadrado co-cartesiano, como descrito no capítulo anterior.

No capítulo 5 são analisadas algumas das possíveis extensões ao presente

trabalho. É abordado a fibrilação de lógicas, e a sincronização de lógicas. No primeiro caso pretende-se combinar as conectivas, assim como as regras de inferência, de diferentes sistemas lógicos, num só sistema. No caso da sincronização de lógicas o sistema resultante vai estar contido no produto cartesiano dos sistemas de partida, sendo a sincronização estabelecida através da definição de aplicações de sincronização. Em ambos os casos é discutido a possibilidade de incorporar esses mecanismos no âmbito dos sistemas de dedução, tal como foram definidos no presente trabalho.

A extensão da linguagem de programação *OBJ3* de forma a esta incorporar as construções próprias do cálculo de co-limites é outra das possíveis extensões ao presente trabalho.

No apêndice A apresentam-se vários exemplos que servem de suporte ao texto dos capítulos anteriores, mas que não foram aí incluídos para não sobrecarregarem o texto.

No apêndice B apresentam-se as listagens dos vários sistemas referidos no capítulo 2, e que são aqui incluídas para complementar a informação dada nesse capítulo.

No apêndice C apresenta-se uma lista de sistemas de demonstração automática, e semi-automática de teoremas, na qual se tentou sistematizar a informação existente em várias fontes (Basin *et al.*, 1993; Dawson, 1991; Dawson, 1992; Goguen, 1988; Goguen, 1990; Goguen *et al.*, 1992; Harper *et al.*, 1993; HOL88a, 1991; Huang *et al.*, 1994; Heuerding *et al.*, 1996; Kalvala, 1994; Kohlhase & Talcott, 1998; Kolyang *et al.*, 1996; Michaylov & Pfenning, 1991; Nesmith, 1993; Paulson, 1993a; Paulson, 1993c; Paulson, 1993b; Pfenning, 1991; Schneider *et al.*, 1992b; Schneider *et al.*, 1992a; Schneider & Kropf, 1992; Schneider *et al.*, 1993; Schumann, 1994; Stevens & Hobley, 1993).

Capítulo 1

Demonstração Automática de Teoremas

> "Logic appears in a *sacred* and in a *profane* form. The sacred form is dominant in proof theory, the profane form in model theory"
>
> *D. van Dalen*, Logic and Structure (van Dalen, 1980)

Neste capítulo vão-se expor as bases teóricas em que assentam os sistemas de demonstração automática, e semi-automática, de teoremas.

1.1 Teoria da dedução

A teoria da dedução tem como objecto de estudo a formalização das demonstrações matemáticas, assim como a análise da estrutura dessas mesmas demonstrações (Oliveira, 1979; Schütte, 1977; Takeuti, 1975).

Conceitos fundamentais na teoria da dedução são os conceitos de linguagem formal, de sistema de dedução, e de demonstração.

Dado se pretender ter como objecto de estudo as demonstrações numa dada linguagem formal (lógica) torna-se necessário definir rigorosamente o que se entende por linguagem lógica, e como é que se efectuam as demonstrações nessa mesma linguagem lógica.

Ou seja pretende-se definir rigorosamente o que se entende por sistema de dedução para uma dada linguagem lógica.

Para definir um sistema de dedução é necessário (Hamilton, 1991):

- um alfabeto de símbolos, ou seja um conjunto, eventualmente infinito, enumerável de símbolos;
- um conjunto de palavras, isto é, sequências de símbolos constituídas por elementos do alfabeto, e designadas por fórmulas bem formadas;

- um conjunto, eventualmente infinito, enumerável de fórmulas bem formadas designadas por axiomas;
- um conjunto finito de regras de dedução, isto é, regras que nos permitem deduzir uma fórmula bem formada A, como consequência directa de um conjunto finito de fórmulas bem formadas $A_1, \ldots, A_n$.

O conceito de demonstração, central para qualquer sistema de dedução, vai ter algumas variantes; informalmente pode-se afirmar que é uma sequência de aplicações das regras de inferência, tendo como ponto de partida os axiomas da linguagem e como ponto de chegada a fórmula que se pretende validar. A fórmula bem formada conclusão de uma demonstração num dado sistema de dedução, é dita um teorema nesse sistema de dedução.

Para todo o sistema de dedução é necessário que se verifique que todo teorema é uma fórmula válida, e que todo a fórmula válida é deduzível, ou seja que o sistema de dedução é idóneo e completo, respectivamente (Gallier, 1987).

Uma questão importante quando se considera a possibilidade da demonstração automática é a existência de um método efectivo para decidir se uma dada fórmula é, ou não, um teorema. No caso de um tal método existir diz-se que o sistema lógico é decidível. Infelizmente muitos dos sistemas lógicos são não decidíveis, e como tal a utilização de um sistema de demonstração automática vai estar sempre limitada por esse facto.

Antes de descrever os vários formalismos para a apresentação de um sistema de dedução, torna-se necessário descrever a linguagem formal sobre a qual se quer basear o sistema de dedução, dando o alfabeto, i.e. o conjunto de símbolos disponíveis, e as regras para a construção das fórmulas bem formadas. Vamos usar, sem perda de generalidade, uma linguagem lógica proposicional, e um sistema de dedução nela baseado.

Definição 1.1 (Alfabeto) *O alfabeto de uma linguagem lógica proposicional L consiste em (Hamilton, 1991):*

- *um conjunto enumerável de símbolos proposicionais* $p_1, p_2, \ldots$;
- *conectivas proposicionais:* $\neg$ *(negação),* $\supset$ *(implicação),* $\wedge$ *(conjunção) e* $\vee$ *(disjunção);*
- *símbolos auxiliares "(" e ")".*

As conectivas lógicas têm de constituir um conjunto completo de conectivas para a linguagem lógica em vista. Desta forma será possível escrever toda e qualquer fórmula nessa mesma linguagem. Exemplos de conjuntos completos de conectivas são os conjuntos: $\{\vee, \neg\}$, $\{\wedge, \neg\}$, $\{\supset, \neg\}$, $\{\supset, \bot\}$, $\{|\}$ e $\{\downarrow\}$ (Andrews, 1986; Hamilton, 1991). De forma a simplificar a escrita é usual introduzir, por definição, outras conectivas, como sejam: $\equiv$ (equivalência), $\top$ (verdade) e $\bot$ (falso).

Por razões de exposição dos diferentes sistemas de dedução, e de apresentação de diferentes exemplos de deduções nesses sistemas, optou-se pelo seguinte conjunto completo de conectivas, $\{\supset, \wedge, \neg\}$.

Temos então as seguintes regras de formação de fórmulas bem formadas.

Definição 1.2 (Fórmula bem formada) *O conjunto das fórmulas bem formadas da linguagem L é o menor conjunto de palavras*[1] *em L tal que (Hamilton, 1991):*

1. *todo o símbolo proposicional é uma fórmula bem formada;*
2. *se A e B são fórmulas bem formadas, então* $(\neg A)$, $(A \supset B)$, *e* $(A \wedge B)$ *são fórmulas bem formadas;*
3. *uma expressão não é uma fórmula bem formada excepto se o for por virtude das regras anteriores.*

No que se segue vai-se designar as fórmulas bem formadas simplesmente por fórmulas, os símbolos proposicionais são também designados por fórmulas atómicas.

Posto isto vejamos as diferentes formalizações possíveis para o conceito de sistema dedutivo.

1.1.1 Sistemas Hilbertianos

O formalismo desenvolvido, entre outros, por Russel, Hilbert e Heyting (Szabo, 1969) é caracterizado por um conjunto finito de esquemas de axiomas que vão caracterizar cada uma das conectivas lógicas existentes na linguagem lógica, e de um número finito, usualmente pequeno, de esquemas de regras de inferência, isto é formas de argumentação válidas (Oliveira, 1979). Este formalismo está bastante afastado da usual prática dos matemáticos (Szabo, 1969) e não é bem adaptado a formas de automatização da demonstração.

Definição 1.3 (Sistema de Dedução H) *Dado uma linguagem lógica L, sejam A, B e C fórmulas em L, então as seguinte fórmulas são axiomas do sistema de dedução H (Gallier, 1987):*

L1 $(A \supset (B \supset A))$;

L2 $((A \supset B) \supset ((A \supset (B \supset C)) \supset (A \supset C)))$;

L3 $(A \supset (B \supset (A \wedge B)))$;

L4 $((A \wedge B) \supset A)$;

L5 $((A \wedge B) \supset B)$;

[1] Sucessões finitas de símbolos de L perfeitamente arbitrárias.

L6 $((A \supset B) \supset ((A \supset \neg B) \supset \neg A))$;

L7 $(\neg\neg A \supset A)$.

*Para quaisquer fórmulas A e B em L temos que: de A e $(A \supset B)$ tiramos como consequência directa B. Esta regra de inferência é usualmente designada por "*modus ponens*" (**MP**). As fórmulas A e $(A \supset B)$ são designadas por antecedentes e a fórmula B é designada por consequente. É usual apresentar esta regra de inferência na seguinte forma:*

$$\textbf{MP}\ \frac{A \quad (A \supset B)}{B}.$$

De seguida torna-se necessário formalizar o conceito de dedução e de demonstração. No que se segue vai-se designar por H, não só o conjunto de esquemas de axiomas e o esquema da regra de inferência, como também a linguagem lógica que lhe serve de base.

Definição 1.4 (Dedução) *Seja Γ um conjunto de fórmulas de H, uma sequência $A_1, \ldots, A_n$ de fórmulas de H é uma dedução a partir de Γ, se para todo o $i (1 \leq i \leq n)$ se verifica que; A_i é um dos elementos de Γ, ou A_i é um dos axiomas de H, ou A_i é consequência directa de dois elementos anteriores da sequência por aplicação da regra* modus ponens. *Diz-se então que A_n é deduzível de Γ, ou que é uma consequência de Γ. Denota-se tal facto por $\Gamma \vdash_H A$.*

Uma demonstração em H é uma dedução a partir de um conjunto vazio de fórmulas.

Definição 1.5 (Demonstração) *Uma demonstração em H é uma sequência $A_1, \ldots, A_n$, de fórmulas de H tais que para todo o $i (1 \leq i \leq n)$, ou A_i é um axioma de H, ou A_i é consequência directa de dois elementos anteriores da sequência por aplicação da regra* modus ponens. *Diz-se então que obtivemos uma demonstração de A_n, e que A_n é um teorema de H. Denota-se por $\vdash_H A$.*

Vejamos um exemplo de uma dedução em H. A dedução é apresentada numa forma usual para este tipo de sistema. Na coluna da esquerda enumeram-se os vários elementos que compõem a sequência de forma a puderem ser referenciados, na coluna do centro escrevem-se os elementos da sequência, e na coluna da direita escreve-se a justificação do passo efectuado por referência aos passos anteriores, e aos esquemas de axiomas e esquemas de regras de inferência do sistema.

Dedução de $\{A, (B \supset (A \supset C))\} \vdash_H (B \supset C)$.

(1)	A	assumpção
(2)	$B \supset (A \supset C)$	assumpção
(3)	$A \supset (B \supset A)$	L1
(4)	$B \supset A$	de (1) e (3) por MP
(5)	$(B \supset A) \supset ((B \supset (A \supset C)) \supset (B \supset C))$	L2
(6)	$(B \supset (A \supset C)) \supset (B \supset C)$	de (4) e (5) por MP
(7)	$B \supset C$	de (2) e (6) por MP

A dificuldade que se pode sentir de imediato numa tentativa de automatização da demonstração num cálculo deste tipo, é a dificuldade própria do encontrar de instâncias para os esquemas de axiomas que permitam a posterior aplicação da regra MP, de molde a obter a conclusão pretendida.

É de referir que, nesta dedução assim como no que se segue, se optou por omitir os parêntesis exteriores das fórmulas.

1.1.2 Dedução natural

Quando Gerard Gentzen formalizou o cálculo de dedução natural a sua intenção, aliás bem expressa no nome que lhe atribuiu, era a de formalizar um sistema que ficasse o mais perto possível da forma de raciocínio matemático usual (Szabo, 1969).

As demonstrações no cálculo de dedução natural começam, de uma forma geral, não com proposições lógicas básicas (axiomas), como no cálculo anterior, mas sim com assumpções (hipóteses) sobre as quais se aplicam um conjunto finito de regras de inferência. A conclusão é tornada independente das hipótese por cancelamento das mesmas através de certas regras de inferência.

O cálculo de dedução natural organiza-se em torno de regras de inferência para as várias conectivas lógicas da linguagem. Para cada conectiva existem, pelo menos, duas regras de inferência, uma de introdução da conectiva e outra de eliminação da conectiva, o que se entende por introdução ou eliminação de uma conectiva tornar-se-á claro aquando da definição de um dado sistema.

Definição 1.6 (Sistema de Dedução N) *Dado uma linguagem lógica L, sejam A, B e C fórmulas em L, então as regras de inferência para o cálculo de dedução natural N são (Szabo, 1969; van Dalen, 1980):*

Regras de Inferência

$$\frac{A \quad B}{A \wedge B}\ (I-\wedge) \qquad \frac{A \wedge B}{A}\ \frac{A \wedge B}{B}\ (E-\wedge)$$

$$\frac{\overset{[A]}{B}}{A \supset B}\ (I-\supset) \qquad \frac{A \quad A \supset B}{B}\ (E-\supset)$$

$$\frac{\overset{[A]}{\bot}}{\neg A}\ (I-\neg) \qquad \frac{A \quad \neg A}{\bot}\ \frac{\bot}{A}\ (E-\neg)$$

Os símbolos entre parêntesis rectos representam a possível introdução de um número arbitrário de assumpções, todas formalmente idênticas, nesse ponto. O símbolo $\perp$ é, por definição, equivalente a $A \wedge \neg A$ (Gallier, 1987).

A representação usual de deduções em N é sob a forma de árvores de dedução, ou seja, árvores em que os nós contêm fórmulas da linguagem, as folhas contêm assumpções, e a raiz a conclusão. A construção da árvore está dependente das regras de inferência do cálculo.

Exemplo de uma árvore de dedução em N; a dedução de $A \supset C$ tendo como assumpções não canceladas $A \supset B$ e $B \supset C$, e como assumpção cancelada A:

$$\dfrac{\dfrac{\dfrac{[A]^1 \quad A \supset B}{B}\,(E-\supset) \qquad B \supset C}{C}\,(E-\supset)}{A \supset C}\,(I-\supset)^1$$

A representação usada tem a vantagem de permitir acrescentar o nome da regra de dedução usada, realçando assim o facto de a árvore de dedução ser construída por aplicações sucessivas das regras de inferência.

Torna-se necessário definir formalmente o que se entende por árvore, dedução e cancelamento de assumpções.

Definição 1.7 (Árvore) *Uma árvore é um conjunto finito T de um ou mais nós tais que (Veloso* et al.*, 1983):*

1. *existe um nó denominado raiz da árvore;*
2. *os demais nós formam $n \geq 0$ conjuntos disjuntos $S_1, S_2, \ldots, S_n$, sendo que cada um desses conjuntos é uma árvore.*

Definição 1.8 (Dedução, Cancelamento de Assumpções) *O conjunto das deduções é o menor conjunto D tal que (Szabo, 1969; A.S.Troelstra & Dalen, 1988; van Dalen, 1980):*

1. *uma árvore contendo um só nó A, com A uma fórmula de N, é uma dedução. Não existem assumpções canceladas;*
2. *sejam $\overset{D_1}{\phi_1}$ e $\overset{D_2}{\phi_2}$ deduções com conclusão ϕ_1 e ϕ_2 respectivamente, então:*

$$\dfrac{\overset{D_1}{\phi_1}}{\psi}R_1 \qquad \dfrac{\overset{D_1}{\phi_1} \quad \overset{D_2}{\phi_2}}{\psi}R_2$$

com R_1 uma instância das regras $(E-\wedge), (I-\supset), (I-\neg)$ e $(E-\neg)$ (segunda variante), e R_2 uma instância das regras $(I-\wedge), (E-\supset)$ e $(E-\neg)$ (primeira variante), são deduções.

Cancelamento de assumpções:

(a) nas regras $(I-\supset)$ *e* $(I-\neg)$ *todas as assumpções da forma* A *em* D_1 *são canceladas.;*

(b) se D *é uma dedução do antecedente da última regra a aplicar numa dedução, então assumpções não canceladas em* D *permanecem não canceladas, excepto nos casos especificados em (2a).*

Exemplo, demonstração de $((A \wedge B) \supset C) \supset (A \supset (B \supset C))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[A]^2 \quad [B]^1}{A \wedge B}(I-\wedge) \qquad [(A \wedge B) \supset C]^3}{C}(E-\supset)}{B \supset C}(I-\supset)^1}{A \supset (B \supset C)}(I-\supset)^2}{((A \wedge B) \supset C) \supset (A \supset (B \supset C))}(I-\supset)^3$$

Os expoentes nas assumpções canceladas, e nos nomes das regras de inferência usadas, dão conta do ponto onde foi feito o cancelamento das assumpções.

A necessidade da introdução de assumpções, e o seu posterior cancelamento, dificulta a automatização de um cálculo de dedução natural.

1.1.3 Cálculo de sequências dedutivas

O cálculo de sequências dedutivas, ou de asserções de deductibilidade (Oliveira, 1979) foi introduzido por Gerard Gentzen tendo este tentado combinar as melhores características de ambos os formalismos anteriores. Por um lado o carácter "logicista" do formalismo de Hilbert, ou seja um cálculo no qual não é necessário fazer a introdução de assumpções e seu posterior cancelamento, e por outro lado o carácter sistemático do formalismo de dedução natural, com a divisão das regras de inferência em regras de introdução e de eliminação, para cada uma das conectivas lógicas (Szabo, 1969).

Para converter uma dedução dependente de assumpções numa dedução logicista procede-se da seguinte forma: converte-se a dedução de B dependente das assumpções $A_1, \ldots, A_m$ numa nova fórmula $A_1 \wedge \ldots \wedge A_m \supset B$, sendo que esta é independente de qualquer assumpção (Szabo, 1969).

De forma a não não perturbar o carácter sistemático do cálculo Gentzen introduziu o conceito de sequências dedutivas ("sequents") (Szabo, 1969). Em vez da fórmula $A_1 \wedge \ldots \wedge A_m \supset B$ introduzem-se os símbolos auxiliares "$\vdash$" e "," escrevendo-se,

$$A_1, \ldots, A_m \vdash B$$

cujo significado informal é o mesmo da fórmula anterior (Bibel & Eder, 1993; Gallier, 1987; Szabo, 1969).

Definição 1.9 (Sequência Dedutiva) *Uma sequência dedutiva é um par* (Γ, Δ) *de sequências finitas, eventualmente vazias,* $\Gamma = \langle A_1, \ldots, A_m \rangle$, $\Delta = \langle B_1, \ldots, B_n \rangle$ *de fórmulas(Gallier, 1987; Szabo, 1969).*

Usando a notação introduzida por Gentzen (Szabo, 1969) com recurso aos símbolos auxiliares já definidos temos $\Gamma \vdash \Delta$, designando-se Γ como o antecedente, e Δ como o consequente. Se Γ é vazio escreve-se $\vdash \Delta$ e o seu significado é o mesmo do que a fórmula $B_1 \vee \ldots \vee B_n$, se Δ é vazio escreve-se $\Gamma \vdash$ e o seu significado é o mesmo do que a fórmula $\neg(A_1, \wedge \ldots \wedge A_m)$, se tanto Γ como Δ são vazios, tem-se a sequência dedutiva incoerente (Gallier, 1987; Szabo, 1969).

Um determinado cálculo de sequências dedutivas organiza-se em torno de regras de inferência para as várias conectivas lógicas da linguagem. Analogamente ao cálculo de dedução natural tem-se a sistematização das regras de inferência de acordo com as conectivas lógicas, sendo que a divisão é feita consoante a posição da conectiva em relação ao símbolo auxiliar $\vdash$.

As deduções num cálculo de sequências dedutivas vão-se desenvolver em árvores de dedução em cuja raiz está a fórmula que se pretende demonstrar, i.e. a sequência dedutiva $\vdash A$, e em cujas folhas estão os axiomas, isto é, sequências dedutivas $\Gamma \vdash \Delta$ em que Γ e Δ tem uma mesma fórmula em comum.

Definição 1.10 (Sistema de Dedução G) *Dado uma linguagem lógica* L, *sejam* Γ, Δ *e* Λ *sequências arbitrárias de fórmulas de* L, *e* A *e* B *fórmulas em* L. *O sistema de dedução* G *define-se por (Gallier, 1987):*

Axiomas *Sequências dedutivas* $\Gamma \vdash \Delta$ *tal que* Γ *e* Δ *contêm uma fórmula em comum.*

Regras de inferência

$$\frac{\Gamma, A, B, \Delta \vdash \Lambda}{\Gamma, A \wedge B, \Delta \vdash \Lambda}(\wedge \vdash) \qquad \frac{\Gamma \vdash \Delta, A, \Lambda \quad \Gamma \vdash \Delta, B, \Lambda}{\Gamma \vdash \Delta, A \wedge B, \Lambda}(\vdash \wedge)$$

$$\frac{\Gamma, \Delta \vdash A, \Lambda \quad B, \Gamma, \Delta \vdash \Lambda}{\Gamma, A \supset B, \Delta \vdash \Lambda}(\supset \vdash) \qquad \frac{A, \Gamma \vdash B, \Delta, \Lambda}{\Gamma \vdash \Delta, A \supset B, \Lambda}(\vdash \supset)$$

$$\frac{\Gamma, \Delta \vdash A, \Lambda}{\Gamma, \neg A, \Delta \vdash \Lambda}(\neg \vdash) \qquad \frac{A, \Gamma \vdash \Delta, \Lambda}{\Gamma \vdash \Delta, \neg A, \Lambda}(\vdash \neg)$$

Definição 1.11 (Dedução) *O conjunto das árvores de dedução é o menor conjunto* D, *tal que (Gallier, 1987):*

1. *uma árvore contendo um só nó* $\Gamma \vdash \Delta$ *é um árvore de dedução, sendo que* $\Gamma \vdash \Delta$ *é uma sequência dedutiva da linguagem lógica que se está a considerar;*

2. *sejam* $\frac{T}{\Gamma \vdash \Delta}$, $\frac{T'}{\Gamma' \vdash \Delta'}$ *árvores de dedução com conclusões* $\Gamma \vdash \Delta$ *e* $\Gamma' \vdash \Delta'$ *respectivamente, então:*

$$\frac{\dfrac{T}{\Gamma \vdash \Delta}}{\Lambda \vdash \Theta}(R_1) \quad \text{e} \quad \frac{\dfrac{T}{\Gamma \vdash \Delta} \quad \dfrac{T'}{\Gamma' \vdash \Delta'}}{\Lambda \vdash \Theta}(R_2)$$

com R_1 *uma instância das regras* $(\wedge \vdash), (\vdash \supset), (\neg \vdash)$ *e* $(\vdash \neg)$, *e* R_2 *uma instância das regras* $(\vdash \wedge)$, *e* $(\supset \vdash)$, *são deduções.*

O conjunto das demonstrações é nos dado pelas deduções cujas folhas contêm axiomas do sistema lógico.

Definição 1.12 (Demonstração) *O conjunto das demonstrações é o menor conjunto* P, *tal que (Gallier, 1987):*

1. *uma árvore contendo um só nó* $\Gamma \vdash \Delta$, *sendo* $\Gamma \vdash \Delta$ *um axioma, é uma demonstração;*

2. *sejam* $\frac{T}{\Gamma \vdash \Delta}$, $\frac{T'}{\Gamma' \vdash \Delta'}$ *demonstrações com conclusões* $\Gamma \vdash \Delta$ *e* $\Gamma' \vdash \Delta'$ *respectivamente então:*

$$\frac{\dfrac{T}{\Gamma \vdash \Delta}}{\Lambda \vdash \Theta}(R_1) \quad \text{e} \quad \frac{\dfrac{T}{\Gamma \vdash \Delta} \quad \dfrac{T'}{\Gamma' \vdash \Delta'}}{\Lambda \vdash \Theta}(R_2)$$

com R_1 *uma instância das regras* $(\wedge \vdash), (\vdash \supset), (\neg \vdash)$ *e* $(\vdash \neg)$, *e* R_2 *uma instância das regras* $(\vdash \wedge)$, *e* $(\supset \vdash)$, *são demonstrações.*

Dado ser um cálculo logicista e sistematizado, o cálculo das sequências dedutivas é aquele que melhor se adapta à automatização das demonstrações.

Um exemplo, a demonstração de $((A \wedge B) \supset C) \supset (A \supset (B \supset C))$ no sistema G.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{B, A \vdash A \quad B, A \vdash B}{B, A \vdash A \wedge B}(\vdash \wedge) \qquad C, B, A \vdash C}{B, A, ((A \wedge B) \supset C) \vdash C}(\supset \vdash)}{A, ((A \wedge B) \supset C) \vdash (B \supset C)}(\vdash \supset)}{((A \wedge B) \supset C) \vdash (A \supset (B \supset C))}(\vdash \supset)}{\vdash ((A \wedge B) \supset C) \supset (A \supset (B \supset C))}(\vdash \supset)$$

A construção de uma demonstração desenvolve-se "de baixo para cima", ou seja da conclusão (raiz da árvore), para os axiomas (folhas da árvore), sendo que em determinados sistemas lógicos, por exemplo o cálculo proposicional, é fácil construir um algoritmo para automatizar a construção da árvore de demonstração (Gallier, 1987).

A formalização escolhida difere da formalização clássica na medida em que as regras estruturais e a regra do corte ("cut rule") não estão presentes de forma explícita (Gallier, 1987; Szabo, 1969; van Dalen, 1980). Tal escolha não representa uma perda de generalidade dado que se pode demonstrar que ambas as formalizações são equivalentes (Gallier, 1987). Subjacente à apresentação feita está a propriedade da sub-fórmula que nos diz, informalmente, que toda a informação necessária a uma demonstração se encontra na sua conclusão (Gallier, 1987; Szabo, 1969; van Dalen, 1980). É por este motivo que é usual desenvolver as demonstrações de baixo (conclusão) para cima (axiomas).

1.2 Automatização das demonstrações

Na secção anterior apresentaram-se os formalismos normalmente usados para o estudo dos sistemas lógicos, e para o desenvolvimento das demonstrações nesses sistemas. Trata-se agora de apresentar os métodos de automatização das demonstrações.

Pretende-se então, estabelecer a validade de uma fórmula através do uso de um sistema computacional. Uma fórmula proposicional G é válida, se e só se, para toda a interpretação I de G, esta tem o valor verdade, isto é, para toda a atribuição de valores de verdade aos símbolos proposicionais da fórmula G, o seu valor final é igual a, verdade (Chang & Lee, 1973). No caso da lógica proposicional, para uma fórmula com n símbolos atómicos diferentes existem 2^n interpretações diferentes, sendo assim é sempre possível decidir sobre a validade de uma fórmula. A introdução de símbolos de variáveis, e de símbolos de funções, na construção das fórmulas conduz à necessidade de atribuir valores, isto é, elementos e funções de um dado conjunto, a essas variáveis aquando da interpretação da fórmula. Tal facto torna a tarefa de verificação da validade de uma fórmula só possível em casos restritos, como por exemplo, em conjuntos de elementos finitos. Torna-se então necessário procurar algoritmos que não passem pela enumeração de todos os casos possíveis.

1.2.1 Universo de Herbrand

Em 1930 Herbrand[2] desenvolveu um algoritmo que permite verificar a validade de uma fórmula, neste método a demonstração desenvolve-se por refutação, isto é tenta-se demonstrar que a negação da fórmula original é incoerente, demonstrando deste modo a validade da fórmula original (Chang & Lee, 1973).

[2]Herbrand, J. ; Investigations in proof theory: the properties of true propositions ; "From Frege to Gödel: a Source Book in Mathematical Logic 1879–1931", (J. van Heijenoort ed.), Harvard Univ. Press., Cambridge, Massachusetts, 1967.

O algoritmo tem por base o resultado que se apresenta de seguida. Alguns dos conceitos apresentados serão explicitados posteriormente.

Teorema 1.1 *Um conjunto S de cláusulas é incoerente sse existe um conjunto finito S' de instâncias de base de S que é incoerente (Chang & Lee, 1973).*

É então necessário encontrar um dado conjunto finito S' de instâncias de base de S. Se a procura for bem sucedida, então S é incoerente, e a fórmula que se pretendia demonstrar é válida. Se S não é incoerente não existe um tal conjunto, e como tal o algoritmo tem de prever a possibilidade de parar após um número finito de passos. Neste último caso não se pode tirar nenhuma conclusão, pois não se conseguiu demonstrar que um dado conjunto finito S' não existe, mas tão somente que não foi possível encontrá-lo no número de passos escolhido.

Vejamos, muito brevemente, como se desenvolve essa procura do conjunto S', começando por explicitar os conceitos introduzidos no enunciado do teorema 1.1.

Definição 1.13 (Literal) *Um literal é uma fórmula atómica, ou a negação de uma fórmula atómica (Chang & Lee, 1973).*

Definição 1.14 (Cláusula) *Uma cláusula é uma disjunção finita de zero ou mais literais (Chang & Lee, 1973).*

Definição 1.15 (Universo de Herbrand) *Seja H_0 o conjunto das constantes pertencentes a um conjunto de cláusulas S. No caso em que S não contêm nenhuma constante considera-se H_0 como sendo formado por uma única constante arbitrária, seja então $H_0 = \{a\}$. Para $i = 0, 1, 2, \ldots$, H_{i+1} é a união de H_i com o conjunto de todos os termos da forma $f^n(t_1, \ldots, t_n)$ para toda a função de aridade n que ocorre em S, e onde $t_j, j = 1, \ldots, n$ são elementos do conjunto H_i, cada H_i é designado por conjunto constante de nível i de S, e $\lim_{i\to\infty} H_i$ ou H_∞ é designado por Universo de Herbrand de S (Chang & Lee, 1973).*

Definição 1.16 (Instância de Base) *Uma instância de base da cláusula C de um conjunto de cláusulas S, é uma cláusula obtida de C por substituição das suas variáveis por elementos do Universo de Herbrand de S (Chang & Lee, 1973).*

Temos então o seguinte algoritmo: dado um conjunto de cláusulas, que se supõe incoerente, geram-se os sucessivos conjuntos de cláusulas de base de S, $S'_1, S'_2, \ldots, S'_n, \ldots$ testando sucessivamente a incoerência de cada um deles. Então, pelo teorema 1.1, podemos afirmar que, no caso em que S é incoerente, existe um n finito tal que S'_n é incoerente.

Gilmore[3] em 1960, Davis e Putnam[4] também em 1960, entre outros, implementaram o algoritmo de Herbrand. Eles construiram programas capazes de gerar, sucessivamente, os conjuntos $S'_1, \ldots, S'_n, \ldots$ referidos atrás e consequente verificação da incoerência de cada um deles, sendo que Gilmore usou o método multiplicativo, enquanto Davis e Putnam usaram um método baseado em quatro regras, o qual se revelou mais eficiente do que o de Gilmore (Chang & Lee, 1973).

No entanto as implementações do algoritmo de Herbrand sofrem todas do constrangimento provocado pela necessidade de gerar a sequência $S_1, \ldots, S_n, \ldots$, esta sequência tem em geral um crescimento exponencial, o que faz com que os programas sejam muito ineficientes. Vejamos um exemplo (Chang & Lee, 1973):

Consideremos

$$S = \{P(x, g(x), y, h(x, y), z, k(x, y, z)), \neg P(u, v, e(v), w, f(v, w), x)\}$$

temos que:

$$\begin{aligned} H_0 &= \{a\} \\ H_1 &= \{a, g(a), h(a, a), k(a, a, a), e(a), f(a, a)\} \\ &\vdots \end{aligned}$$

O número de elementos em S'_0 e S'_1 é de 2 e 1512 respectivamente. O primeiro dos conjuntos S'_i que é incoerente é S'_5 cujo cardinal é da ordem de 10^{256} sendo o seu manuseamento computacional bastante difícil.

Em 1965 Robinson (Robinson, 1965) introduziu um método, que designou por princípio de resolução, o qual, podendo ser aplicado directamente a um qualquer conjunto S de cláusulas, evita a necessidade de gerar os conjuntos $S'_1, \ldots, S'_n, \ldots$ sendo muito mais eficiente que o algoritmo de Herbrand.

1.2.2 Método de resolução

A ideia base do princípio de resolução é a seguinte: dado um conjunto de cláusulas S, verificar se S contem a cláusula vazia; se S contem a cláusula vazia, então S é incoerente; se S não contem a cláusula vazia, então é necessário verificar se se pode derivar a cláusula vazia a partir de S. A cláusula vazia vai-se denotar por $\square$ (Chang & Lee, 1973)

O princípio de resolução pode ser visto como uma regra de inferência que se aplica repetidas vezes, a partir de S geram-se novas cláusulas até se obter a cláusula vazia.

[3]Gilmore, P. C. (1960): A proof method for quantification theory; its justification and realization, IBM J. Res. Develop. 28–35.

[4]Davis, M. and Putnam, H. (1960): A computing procedure for quantification theory, J. ACM, No3, 201–215.

Vejamos antes de mais a sua aplicação para a lógica proposicional, começando por definir o que se entende por complementar de um literal.

Definição 1.17 (Complementar) *Se A é uma fórmula atómica, então dois literais A e $\neg A$, são designados por complementares um do outro (Chang & Lee, 1973).*

Definição 1.18 (Regra de Resolução, Resolvente) *Para qualquer par de cláusulas C_1 e C_2, se existir um literal L_1 em C_1 que é complementar de um literal L_2 em C_2, então elimina-se L_1 e L_2 em C_1 e C_2 respectivamente, e constroi-se a disjunção das restantes cláusulas. A cláusula construída deste modo é um resolvente de C_1 e C_2 (Chang & Lee, 1973).*

Veja-se o seguinte exemplo. Dado as cláusulas:

$$\begin{aligned} C_1 &= ((\neg P) \vee Q) \vee R \\ C_2 &= (\neg Q) \vee S \end{aligned}$$

um seu resolvente, e único neste caso, é:

$$C_3 = ((\neg P) \vee R) \vee S$$

Torna-se necessário definir agora o que se entende por dedução, e por demonstração usando o princípio de resolução. Como passo necessário nesse sentido temos o seguinte resultado.

Teorema 1.2 *Dados duas cláusulas C_1 e C_2, um resolvente C de C_1 e C_2 é uma consequência lógica de C_1 e C_2 (Chang & Lee, 1973).*

Temos então as seguintes definições para dedução e demonstração usando o princípio de resolução. Dado se tratar de uma demonstração por refutação vai-se usar a designação de refutação como simplificação de demonstração por refutação.

Definição 1.19 (Dedução por Resolução) *Dado um conjunto S de cláusulas, uma dedução por resolução de C a partir de S, é uma sequência finita $C_1, C_2, \ldots, C_k$ de cláusulas, tais que cada C_i com $i = 1, \ldots, k$ ou é uma cláusula de S, ou é um resolvente de duas cláusulas que o precedem, e $C_k = C$ (Chang & Lee, 1973).*

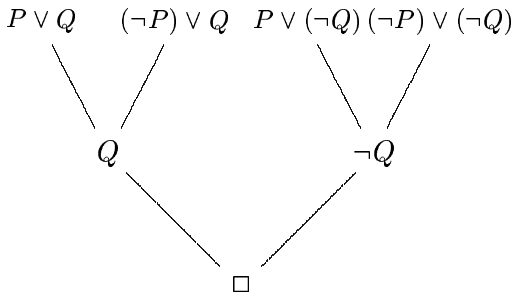
Definição 1.20 (Refutação por Resolução) *Uma dedução por resolução da cláusula vazia a partir de um conjunto S de cláusulas, é uma refutação de S por resolução (Chang & Lee, 1973).*

Vejamos um exemplo.
Seja $S = \{P \vee Q, (\neg P) \vee Q, P \vee (\neg Q), (\neg P) \vee (\neg Q)\}$. Temos então:

$$\begin{array}{lcll} C_1 & = & P \vee Q & \\ C_2 & = & (\neg P) \vee Q & \\ C_3 & = & P \vee (\neg Q) & \\ C_4 & = & (\neg P) \vee (\neg Q) & \\ C_5 & = & Q & \text{de } C_1 \text{ e } C_2 \\ C_6 & = & \neg Q & \text{de } C_3 \text{ e } C_4 \\ C_7 & = & \square & \text{de } C_5 \text{ e } C_6 \end{array}$$

Dado que a cláusula vazia foi deduzida a partir de S por resolução, e tendo por base o teorema 1.2, temos que a cláusula vazia é uma consequência lógica de S, logo S é incoerente.

A refutação pode-se representar gráficamente na forma de uma árvore de dedução em que as folhas representam o conjunto de cláusulas iniciais, os nós intermédios os vários passos do método de resolução, e a raíz a cláusula conclusão da dedução. No caso anterior teríamos:



Como vimos para o caso proposicional, o princípio de resolução baseia-se na procura de um par de literais complementares. No caso da lógica de predicados a introdução de variáveis torna este problema de muito difícil resolução.

Veja-se o seguinte exemplo:

$$\begin{array}{lcl} C_1 & = & P(x) \vee Q(x) \\ C_2 & = & (\neg P(f(x))) \vee R(x) \end{array}$$

não existe em C_1 e C_2 um par de literais complementares. No entanto se em C_1 substituirmos x por $f(x)$ obtemos:

$$C_1^* = P(f(x)) \vee Q(f(x))$$

que é uma instância de C_1. Em C_1^* e C_2 existe um par de literais complementares, sendo assim obtêm-se:

$$C_3 = Q(f(x)) \vee R(x)$$

sendo que C_3 é o resolvente mais geral de C_1 e C_2, num sentido que se vai explicitar mais adiante.

No caso da lógica de predicados a procura de um par de literais complementares passa então pela procura de uma substituição de variáveis que permita obter, sempre que possível, as instâncias mais gerais das cláusulas que se está a considerar. O algoritmo que permite determinar se é possível obter a substituição mais geral, e em caso afirmativo construí-la, designa-se por, algoritmo de unificação. Antes de enunciar o principal resultado sobre o algoritmo é melhor explicitar o que se entende por uma substituição assim como fixar algumas questões de notação.

Definição 1.21 (Substituição) *Uma substituição é um conjunto finito da forma* $\{t_1/v_1, \ldots, t_n/v_n\}$*, onde todo o* v_i *é uma variável, todo o* t_i *é um termo diferente de* v_i*, e os* v_i *são diferentes dois a dois. A substituição que consiste no conjunto vazio é designada por substituição vazia e é denotada por* ϵ *(Chang & Lee, 1973).*

Vai-se usar letras gregas minúsculas para denotar substituições.

Definição 1.22 (Instância) *Seja* $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ *uma substituição e* E *uma expressão. Então* $E\theta$ *é a expressão que se obtêm de* E *por uma substituição simultânea de cada ocorrência da variável* v_i *em* E*, pelo termo* t_i*, com* $1 \leq i \leq n$*.* $E\theta$ *é designado por uma instância de* E *(Chang & Lee, 1973).*

É possível compor duas substituições sendo que a composição de substituições é associativa e tem a substituição vazia ϵ como o elemento neutro (Chang & Lee, 1973).

Definição 1.23 (Unificador) *Uma substituição* θ *é designado um unificador para um conjunto* $\{E_1, \ldots, E_k\}$ *sse* $E_1\theta = E_2\theta = \cdots = E_k\theta$*. O conjunto* $\{E_1, \ldots, E_k\}$ *é dito unificável se existe um unificador para ele (Chang & Lee, 1973).*

Definição 1.24 (Unificador mais geral) *Um unificador* σ *para um conjunto* $\{E_1, \ldots, E_k\}$ *de expressões é um unificador mais geral sse para todo o unificador* θ *para o conjunto, existe uma substituição* λ *tal que* $\theta = \sigma\lambda$ *(Chang & Lee, 1973).*

Temos então o seguinte resultado.

Teorema 1.3 (Unificação) *Se* W *é um conjunto finito e não vazio de expressões, então o algoritmo de unificação determina se são ou não unificáveis, e em caso afirmativo acha a substituição que é o unificador mais geral para* W *(Chang & Lee, 1973).*

A demonstração destes resultados assim como a descrição do algoritmo está em (Chang & Lee, 1973).

Temos então que uma refutação pelo princípio de resolução para a lógica de predicados é, à semelhança do princípio de resolução para a lógica proposicional, uma sequência finita de cláusulas, partindo do conjunto de cláusulas que resultam da transformação da fórmula que se pretende refutar e continuando com a junção de resolventes que vão sendo obtidos das cláusulas anteriores até se atingir a cláusula vazia. No caso da lógica de predicados torna-se necessário incorporar o algoritmo de unificação de molde a obter os sucessivos resolventes.

Antes de apresentar o resultado principal torna-se necessário apresentar alguns resultados auxiliares.

Definição 1.25 (Factor) *Se dois ou mais literais (com o mesmo sinal) de uma cláusula C têm σ como o unificador mais geral, então $C\sigma$ é designado por factor de C (Chang & Lee, 1973).*

Definição 1.26 (Resolvente Binário, Cláusulas Ancestrais) *Sejam C_1 e C_2 duas cláusulas sem variáveis em comum. Sejam L_1 e L_2 dois literais em C_1 e C_2 respectivamente. Se L_1 e $\neg L_2$ tem um unificador mais geral σ então a cláusula*

$$(C_1\sigma - L_1\sigma) \bigcup (C_2\sigma - L_2\sigma)$$

é designada por resolvente binário de C_1 e C_2. Os literais L_1 e L_2 são designados por literais resolvidos, e C_1 e C_2 são designados por cláusulas ancestrais (Chang & Lee, 1973).

Definição 1.27 (Resolvente) *Um resolvente das cláusulas (ancestrais) C_1 e C_2 é um dos seguintes resolventes binários (Chang & Lee, 1973):*

- *um resolvente binário de C_1 e C_2;*
- *um resolvente binário de C_1, e um factor de C_2;*
- *um resolvente binário de um factor de C_1, e C_2;*
- *um resolvente binário de um factor de C_1, e de um factor de C_2.*

Uma refutacão para a lógica de predicados vai-se desenvolver então de maneira análoga à já vista para o caso proposicional. Vejamos um exemplo (Chang & Lee, 1973).

Seja $S = \{(\neg T(x,y,u,v)) \vee P(x,y,u,v),\ (\neg P(x,y,u,v)) \vee E(x,y,v,u,v,y),\ T(a,b,c,d),\ \neg E(a,b,d,c,d,b)\}$. Temos então:

C_1	=	$(\neg T(x,y,u,v)) \vee P(x,y,u,v)$	
C_2	=	$(\neg P(x,y,u,v)) \vee E(x,y,v,u,v,y)$	
C_3	=	$T(a,b,c,d)$	
C_4	=	$\neg E(a,b,d,c,d,b)$	
C_5	=	$(\neg T(x,y,u,v)) \vee E(x,y,v,u,v,y)$	de C_1 e C_2
C_6	=	$E(a,b,d,c,d,b)$	de C_3 e $C_5\sigma$ com $\sigma = (a/x, b/y, c/u, d/v)$
C_7	=	$\square$	de C_4 e C_6

É possível apresentar o principio de resolução de uma outra forma. Por exemplo Gallier (1987) apresenta um cálculo de sequências dedutivas constituído por uma só regra, a regra de resolução, que é então aplicada sucessivamente, não à fórmula que se quer demonstrar mas a uma sua transformação (logicamente equivalente) numa forma própria para o referido cálculo. Esta última apresentação é interessante pelo facto de frisar o aspecto mecânico, que a demonstração passa a possuir.

1.2.3 Isomorfismo de Curry-Howard

Em 1941 Alonzo Church (Church, 1941) estabeleceu uma relação entre um sistema lógico e um sistema do cálculo-λ. A relação foi estabelecida pela identificação dos valores lógicos de verdade e falsidade, com os inteiros positivos 2 e 1 respectivamente, baseando de seguida o sistema lógico no cálculo que ele designou por cálculo de conversão λ-δ. O sistema tinha um axioma, o termo 2, e sete regras de inferência. Os teoremas do sistema são os termos que podem ser derivados da fórmula 2 por uma sequência de aplicações das regras de inferência.

A introdução das conectivas lógicas usuais, negação, conjunção e disjunção, podia ser feita pela definição de termos apropriados (Church, 1941, pág.69).

É interessante notar que Church verificou que da definição da conectiva $\vee$ (disjunção) advinha que, se $A \vee B$ é um teorema, então A ou B são teoremas. Ou seja estava-se perante um sistema lógico diferente do sistema lógico clássico, e que Church classificou como tendo "*carácter finitista*".

A relação entre os sistemas da lógica intuicionista, e os sistemas do cálculo-λ com tipos foi posteriormente sendo posta em evidência pelos trabalhos de Lambek (1988), Curry (1968), e Howard (1980).

Temos então que a relação entre categorias cartesianamente fechadas e o cálculo-λ com tipo produto, e destes com os sistemas lógicos, sugere a observação de que tipos se identificam com fórmulas, e de que termos se identificam com demonstrações, ou melhor, com classes de equivalência de demonstrações (Lambek & Scott, 1988).

Vejamos então como se pode formalizar esta correspondência, começando por estabelecer um cálculo-λ com tipos produto e exponenciação, cálculo que vai corresponder a um cálculo proposicional intuicionista positivo (Lambek & Scott, 1988).

É necessário definir quais são os tipos admissíveis, quais são os símbolos primitivos do sistema, como se definem os termos dos diferentes tipos, além de estabelecer questões relacionadas com a definição de variáveis livres e mudas, assim como as questões relacionadas com a substituição de variáveis.

Definição 1.28 (Tipos) *Os tipos são (Barendregt, 1981; Girard* et al.*, 1988):*

- *os tipos atómicos, $T_1, T_2, \ldots, T_n$, são tipos;*
- *se U e V são tipos, então $U \times V$ e $U \to V$ são tipos;*
- *os únicos tipos são aqueles que se obtêm por aplicação das regras anteriores.*

Os tipos atómicos vão estar dependentes do sistema concreto com que se está a lidar. O tipo $U \times V$ é designado por produto de U por V, e a sua interpretação é o produto cartesiano de U por V; o tipo $U \to V$ (ou V^U) é designado por tipo exponenciação, e a sua interpretação é o conjunto das funções de U para V.

Definição 1.29 (Alfabeto) *O símbolos que constituem o alfabeto do cálculo-λ são (Girard* et al.*, 1988):*

- *para cada tipo atómico t um conjunto infinito mas enumerável de variáveis, $x_o^t, x_1^t, \ldots$;*
- *símbolos auxiliares: "λ", "$($", "$)$", "$\langle$", "$\rangle$", "π^1", "π^2", "$.$", "$\to$".*

Definição 1.30 (Termos, variáveis mudas e livres) *Os termos do cálculo-λ, e a caracterização das variáveis livres e mudas, é dada pelo seguinte conjunto de regras (Church, 1941; Girard* et al.*, 1988):*

- *as variáveis, $x_o^T, x_1^T, \ldots$ são termos do tipo T. Todas as variáveis de um dado tipo T são variáveis livres;*
- *se u e v são termos do tipo U e V respectivamente, então $\langle u, v\rangle$ é um termo do tipo $U \times V$. Uma ocorrência da variável x é livre ou muda em $\langle u, v\rangle$ conforme é livre ou muda em u, ou em v;*
- *se t é um termo do tipo $U \times V$, então $\pi^1 t$ e $\pi^2 t$ são termos do tipo U e V respectivamente. Toda a ocorrência de uma variável x em $\pi^1 t$ (respectivamente $\pi^2 t$) é livre ou muda consoante o seja em t;*

- *se v é um termo do tipo V e x^U é uma variável do tipo U, então $\lambda x^U.v$ é um termo do tipo $U \to V$. Uma ocorrência de um variável y^U, outra que não x^U, em $\lambda x^U.v$ é livre ou muda consoante o seja em v. Todas as ocorrências de x^U em $\lambda x^U.v$ são mudas;*

- *se t e u são termos do tipo $U \to V$ e U respectivamente, então tu é um termo do tipo V. Uma ocorrência de uma variável x em tu é livre ou muda consoante o seja em t, ou em u;*

- *não existem mais nenhuns termos além dos que se obtêm por aplicação das regras anteriores.*

Uma variável é livre (respectivamente muda) num termo se tem pelo menos uma ocorrência livre (respectivamente muda) no termo (Church, 1941). Um termo é dito fechado se não contém variáveis livres (Barendregt, 1981).

Definição 1.31 (Substituição) *O resultado de substituir as ocorrências livres de x, do tipo T, pelo termo t, do tipo T, $[x/t]$, é definido por (Barendregt, 1981):*

- $x[x/t] \doteq t$;
- $y[x/t] \doteq y$, *se* $x \not\equiv y$;
- $(\lambda y.M)[x/t] \doteq \lambda y.(M[x/t])$, *se* $x \not\equiv y$ *e* y *não é uma variável livre de* t;
- $(M_1 M_2)[x/t] \doteq (M_1[x/t])(M_2[x/t])$;
- $\langle M_1, M_2 \rangle [x/t] \doteq \langle M_1[x/t], M_2[x/t] \rangle$;
- $(\pi^1 M)[x/t] \doteq \pi^1(M[x/t])$;
- $(\pi^2 M)[x/t] \doteq \pi^2(M[x/t])$;

Temos então que, tomando para base o sistema de dedução natural N (definição 1.6), e o cálculo-λ anteriormente definido, a correspondência entre os dois sistemas é a seguinte:

Definição 1.32 (Isomorfismo de Curry-Howard) *A correspondência entre regras de inferência do sistema N, e termos do cálculo-λ é dado por (Girard* et al.*, 1988):*

- *à dedução A corresponde o termo (variável) x_i^A;*

- *à dedução $\dfrac{A \quad B}{A \wedge B}$ corresponde o termo $\langle u, v \rangle$, onde u e v correspondem às deduções A e B respectivamente;*

- *à dedução* $\frac{A \wedge B}{A}$ *(respectivamente* $\frac{A \wedge B}{B}$*) corresponde o termo* $\pi^1 t$ *(respectivamente* $\pi^2 t$*), onde* t *corresponde à dedução* $A \wedge B$*;*

- *à dedução* $\frac{\begin{array}{c}[A]\\ B\end{array}}{A \supset B}$ *corresponde o termo* $\lambda x_i^A.v$*, em que* x_i^A *corresponde à hipótese cancelada* A *e* v *corresponde à dedução de* B*;*

- *à dedução* $\frac{A \quad A \supset B}{B}$ *corresponde o termo* tu*, onde* t *e* u *correspondem às deduções* $A \supset B$ *e* A *respectivamente.*

É de notar que a conectiva lógica $\neg$ não está presente sendo introduzida por definição a partir da constante $\perp$ (falso).

Á noção de dedução vai corresponder a noção de redução. A demonstração vai ser um processo de redução com vista à obtenção de uma forma normal que se pretende de um determinado tipo (Barendregt, 1981; Girard *et al.*, 1988).

Por exemplo, a demonstração:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \quad [A \supset B]^3}{B}(E-\supset) \qquad [B \supset C]^2}{C}(E-\supset)}{A \supset C}(I-\supset)^1}{((B \supset C) \supset (A \supset C))}(I-\supset)^2}{(A \supset B) \supset ((B \supset C) \supset (A \supset C))}(I-\supset)^3$$

É transformada no seguinte termo

$$\lambda x_5^{A \to B}.\lambda x_4^{B \to C}.\lambda x_1^A(\lambda x_2^B.x_3^C((\lambda x_1^A.x_2^B)x_1^A))$$

que, aplicando as regras de redução próprias do cálculo-λ (Barendregt, 1981; Girard *et al.*, 1988) transforma-se no termo

$$(\lambda x_5^{A \to B}.\lambda x_4^{B \to C}.\lambda x_1^A)x_3^C$$

que é um termo normal (Girard *et al.*, 1988) e que tem o tipo $(A \to B) \to ((B \to C) \to (A \to C))$ que corresponde à fórmula $(A \supset B) \supset ((B \supset C) \supset (A \supset C))$.

O interesse do estabelecer do isomorfismo advém do facto de se poder usar a "maquinaria" do cálculo-λ para realizar demonstrações num dado sistema lógico.

1.2.4 Tácticas e Tacticais

Uma aproximação que partilha com a aproximação que se acabou de descrever, a utilização de "ferramentas" computacionais para construir um dado sistema de dedução, é nos dada pela linguagem das tácticas e dos operadores sobre as tácticas que usualmente se designam por tacticais (Gordon *et al.*, 1979).

Na década de 70, foi desenvolvido na Universidade de Edinburgh, o sistema computacional, *Edinburgh LCF* (Gordon *et al.*, 1979; Paulson, 1990; Paulson, 1992). O sistema inicial era um verificador de demonstrações para a lógica das funções computáveis ("Logic of Computable Functions"). A tentativa de desenvolver um sistema que estivesse a meio caminho entre um sistema de demonstração automática e um sistema de verificação, passo a passo, das demonstrações, levou ao desenvolvimento de um sistema programável.

O *Edinburgh LCF* possui uma linguagem do tipo funcional na qual o utilizador pode escrever funções para processar termos, fórmulas, e teoremas. Estes últimos são os elementos de um tipo abstracto de dados que possui como operadores funções que implementam as regras de inferência, são estes operadores que se vão designar por tácticas. A verificação de tipos assegura a idoneidade do sistema. Para distinguir esta linguagem de programação da linguagem lógica que vai ser manipulada, os autores designaram-na por *ML* ("meta-language"), tendo posteriormente evoluido para a linguagem de programação *Standard ML*.

A demonstração desenvolve-se em árvore, da conclusão (raiz) para as premissas (folhas). Cada táctica especifica um passo na demonstração, se o objectivo pertence ao domínio de definição da táctica o resultado é uma lista, eventualmente vazia, de sub-objectivos. No caso em que o objectivo não pertence ao domínio de definição da táctica, considera-se que a aplicação da táctica falhou.

O processo repete-se para todos os ramos da árvore até que a aplicação das tácticas conduzam a listas vazias de sub-objectivos, caso em que se obteve uma demonstração do objectivo proposto. No caso em que para um qualquer dos ramos, não é possível aplicar, com sucesso, nenhum das tácticas implementadas a tentativa de demonstração não foi bem sucedida.

As linguagens das tácticas e tacticais incluem tácticas básicas e tácticas compostas cuja construção é feita à custa dos tacticais. Embora haja muitas variantes a linguagem das tácticas e tacticais tem como base o seguinte:

- cada regra de inferência tem uma correspondente táctica;
- a táctica `tac1 THEN tac2` usa o tactical `THEN` para aplicar ambos as tácticas em sucessão. Ou seja aplica-se `tac1` ao objectivo e depois aplica-se `tac2` ao resultado. Esta táctica falha se qualquer uma das tácticas `tac1` ou `tac2` falha;

- a táctica `tac1 ORELSE tac2` usa o tactical `ORELSE` para aplicar uma das duas tácticas em alternativa. Começa-se por aplicar `tac1` ao objectivo, se esta falhar então aplica-se `tac2`. Se a aplicação de `tac2` também falha então a táctica falha;

- a táctica `REPEAT tac` usa o tactical `REPEAT` para iterar a aplicação da táctica `tac`. Vai-se aplicar a táctica `tac` repetitivamente até a aplicação da táctica `tac` falhar;

- a táctica `idtac` aplicada a um qualquer objectivo não o altera. Esta táctica é sempre bem sucedida, advindo daí o seu interesse para a construção de tácticas compostas.

Existem muitas variantes da noção de tácticas e tacticais, no entanto todas elas são coerentes com as ideias expostas acima. Ou seja, trata-se de ter uma meta-linguagem que permita ao utilizador construir "estratégias" de demonstração que permitam automatizar as demonstrações em sistemas de dedução previamente construídos (Gordon *et al.*, 1979; Paulson, 1992; Paulson, 1993c; Stevens & Hobley, 1993).

Por exemplo, usando a sintaxe do sistema *2OBJ* (Stevens & Hobley, 1993), uma possível implementação para o sistema G (definição 1.10) é:

```
ops AndR ImpR NotR AndL ImpL NotL  : -> Rule .
ops basic FalseL TrueR : -> Rule .

eq AndR (H |- X ^ Y) = (H |- X) , (H |- Y) .
eq ImpR (H |- X -> Y) = (H ; X |- Y) .
eq NotR (H |- ~ X) = (H ; X |- False) .
ceq AndL (H |- X) = (retira(H,acha(Z ^ Y,H)) ;
                     [Tsubterms(acha(Z ^ Y,H))] |- X)
   if existe(Z ^ Y,H) .
ceq ImpL (H |- X) =
        (retira(H,acha(Z -> Y,H)) |- acha(Z -> Y,H) : 1) ,
        (retira(H,acha(Z -> Y,H)) ; acha(Z -> Y,H) : 2 |- X)
   if existe(Z -> Y,H) .
ceq NotL (H |- X) =
        (retira(H,acha(~ Y,H)) |- acha(~ Y,H)  : 1)
   if existe(~ Y,H) .
eq basic (Y ; H |- X) = if  X == hyp(1,Y) then []
                        else basic (H |- X) fi .
eq TrueR (H |- True) = [] .
ceq FalseL (H |- X) = [] if existe(False,H) .
```

Com base neste conjunto de regras, pode-se construir uma táctica composta para o cálculo proposicional, que dado um objectivo tente aplicar

uma das tácticas básicas anteriores, e que repita o processo até obter uma demonstração ou até falhar.

```
op ProvaProp : -> Tactic .

ceq ProvaProp (H |- X) = basic ELSE idtac
   if existe(X,H) .
eq ProvaProp (H |- True) = TrueR ELSE idtac .
ceq ProvaProp (H |- X) = FalseL ELSE idtac
   if existe(False,H) .
ceq ProvaProp (H |- X) = (AndR THEN ProvaProp) ELSE idtac
   if  matches(Z ^ Y , X) .
ceq ProvaProp (H |- X) = (ImpR THEN ProvaProp) ELSE idtac
   if  matches( Z -> Y , X) .
ceq ProvaProp (H |- X) = (NotR THEN ProvaProp) ELSE idtac
   if  matches(~ Y , X) .
ceq ProvaProp (H |- X) = (AndL THEN ProvaProp) ELSE idtac
   if existe(Z ^ Y,H) .
ceq ProvaProp (H |- X) = (OrL THEN ProvaProp) ELSE idtac
   if existe(Z v Y,H) .
ceq ProvaProp (H |- X) = (ImpL THEN ProvaProp) ELSE idtac
   if existe(Z -> Y,H) .
ceq ProvaProp (H |- X) = (NotL THEN ProvaProp) ELSE idtac
   if existe(~ Y,H) and simples(X) .
eq ProvaProp DEFAULT Go = idtac .
```

É de notar que os operadores do tipo `-> Rule`, que correspondem à implementação das regras de inferência do cálculo de sequências deductivas G, são depois consideradas tácticas básicas aquando da construção da táctica `ProvaProp`, sendo este operador do tipo `-> Tactic`.

1.3 Demonstradores de teoremas

Por sistemas de demonstração automática de teoremas entende-se sistemas computacionais, ou sejam conjuntos de programas e ficheiros auxiliares, capazes de responder a questões que requeiram um raciocínio dedutivo, mais concretamente sistemas que implementem os mecanismos necessários para o desenvolvimento de deduções em determinado sistema lógico.

No que diz respeito aos sistemas existentes, uma primeira separação pode ser feita entre os meta-sistemas e os sistemas simples.

Por meta-sistemas entende-se sistemas computacionais capazes de implementar um determinado sistema dedutivo, isto é, uma dada linguagem lógica e correspondente conjunto de regras de dedução. Para isso estes sistemas têm

de possuir uma dada linguagem lógica, meta-lógica, na qual se podem expressar as lógicas objecto, e têm de implementar os mecanismos necessários para o desenvolvimento das demonstrações, assim como os mecanismos que permitem a construção de automatismos para o desenvolvimento das demonstrações. Exemplos de tais sistemas são os sistemas *ICLE* (Dawson, 1991; Dawson, 1992), *Isabelle* (Kalvala, 1994; Paulson, 1993a; Paulson, 1993c; Paulson, 1993b), *ELF* (Harper *et al.*, 1993; Michaylov & Pfenning, 1991; Pfenning, 1991), *2OBJ* (Stevens & Hobley, 1993), entre outros (ver apêndice C).

Por sistemas simples entende-se sistemas que estão baseados numa determinada linguagem lógica, e que possuem um mecanismo de inferência bem determinado. Existem muitos sistemas deste tipo cobrindo muitos sistemas de dedução diferentes (ver apêndice C).

Uma segunda classificação não necessariamente independente da anterior, é nos dada pelo tipo de interacção que cada um dos sistemas proporciona. Temos então que os sistemas dividem-se em sistemas interactivos, ou semi-automáticos, e sistemas automáticos.

Nos primeiros pode-se usar tais sistemas de uma forma interactiva; isto é, pode-se dividir a demonstração por uma sucessão de passos, sendo o utilizador chamado a intervir entre cada um deles de forma a que, baseado nas conclusões que o programa deu introduzir novos dados para a continuação da demonstração. Como caso extremo deste tipo de aproximação temos os programas que podem funcionar como verificadores de teoremas, ou seja, programas capazes de validar, passo a passo, uma determinada dedução a ser dada pelo utilizador.

Nos sistemas automáticos o utilizador dá o objectivo a atingir, os lemas auxiliares que entender necessários e os parâmetros de controlo da demonstração, e como resposta, espera obter o resultado final, ou seja, a confirmação de que a fórmula introduzida é um teorema da linguagem lógica que se está a considerar.

No apêndice C apresenta-se uma lista de demonstradores de teoremas. Tentou-se sistematizar a sua apresentação de forma a ser possível ter uma ideia para cada um dos elementos da lista, qual é o tipo do sistema, e quais são as suas principais características.

Capítulo 2

O Sistema $LPCG_{LK}$

2.1 Introdução

Pretende-se neste capítulo apresentar o problema que, de certa forma, motivou a realização do presente trabalho. Esse problema é o da demonstração da correcção parcial de programas, utilizando sistemas computacionais de demonstração automática de teoremas. Para tal vai-se apresentar a construção de um sistema de dedução para uma linguagem procedimental, e a sua implementação em dois sistemas genéricos de demonstração automática.

As metodologias de programação como sejam, a programação estruturada e descendente, e a programação modular, visam a construção de programas correctos, no entanto não providenciam nenhum mecanismo que permita efectuar a demonstração dessa mesma correcção (Alagič & Arbib, 1978; Gries, 1981; Meyer, 1990; Welsh & Elder, 1979; Wirth, 1973; Wirth, 1976). Por outro lado as técnicas de despistagem de erros, embora com ferramentas poderosas, nunca permitem obter uma demonstração da correcção dos programas. Temos então, que a utilização de sistemas automáticos de demonstração adaptados a sistemas lógicos capazes de descrever a semântica dos programas pode, e deve ser, encarado como um meio complementar aos já descritos para a obtenção de programas correctos.

De seguida vai-se descrever uma linguagem procedimental com comandos guardados, *LPCG* (Alagič & Arbib, 1978; Dijkstra, 1975; Gries, 1981; Meyer, 1990; Quaresma, 1995); um sistema de dedução de sequências deductivas para a referida linguagem, $LPCG_{\mathrm{LK}}$ (Chang & Lee, 1973; Clint & Hoare, 1972; Cousot, 1990; Gallier, 1987; Hoare, 1971; Hoare, 1972; Hoare, 1973; Quaresma, 1995; Szabo, 1969) e as respectivas implementações nos sistemas de demonstração automática de teoremas; *Isabelle* (Kalvala, 1994; Paulson, 1993a; Paulson, 1993b; Paulson, 1993c) e *2OBJ* (Stevens & Hobley, 1993). Finaliza-se com a apresentação de exemplos de demonstrações de correcção parcial de programas em ambos os sistemas, com a descrição das técnicas de demonstração usadas e os resultados obtidos.

2.2 A linguagem de programação

A linguagem *LPCG* é uma linguagem de programação procedimental com comandos guardados (Dijkstra, 1975; Gries, 1981), cujos programas são sequências ordenadas de instruções e cuja operacionalidade é nos dada pela manipulação directa dos dados, através das instruções de atribuição de valor, instruções essas que modificam o valor de variáveis (posições de memória) previamente definidas.

2.2.1 Sintaxe e semântica informal de *LPCG*

De seguida apresenta-se a sintaxe e semântica informal das diferentes instruções que compoẽm a linguagem *LPCG*. Para a descrição da sintaxe vai-se usar uma notação semelhante aquela que é usada na linguagem *OBJ3* para a definição dos operadores (Goguen *et al.*, 1992); nomeadamente é de referir o uso de "_" como uma variável sintáctica, isto é, pode ser substituída por uma qualquer variável de espécie apropriada. As espécies que vão ser consideradas são: `Elementos`, para as diferentes estruturas de dados; `Instruções` para as instruções da linguagem; `Proposições` para as proposições, não só referentes à lógica interna dos programas como à lógica na qual se vão desenvolver as demonstrações de correcção parcial.

Instrução nula

nula : → `Instruções`

A execução da instrução nula não produz qualquer efeito. O seu interesse reside no facto de se poder explicitar que, em determinadas condições, não se pretende alterar o estado das variáveis.

Instrução de Atribuição

`_:=_` : `Elementos Elementos` → `Instruções`

Procede-se ao cálculo da expressão (argumento do lado direito) e atribue-se esse valor à variável (argumento do lado esquerdo).

Instrução de Composição

`_;{_}_` : `Instruções Proposições Instruções` → `Instruções` [assoc]

Dado duas instruções executa-se a segunda após se ter executado a primeira. A operação de composição é associativa. A proposição é aqui considerada como um comentário.

Nas duas próximas instruções vai-se fazer uso daquilo que se vai designar por *comandos guardados*. É de notar que a sua funcionalidade difere, embora ligeiramente, daquela adoptada por Gries (1981).

A sua sintaxe é a seguinte:

$$\texttt{\_->\_} : \texttt{Proposições Instruções} \rightarrow \texttt{Comandos Guardados}$$

O seu significado informal é o seguinte: se a proposição designada por *guarda*, toma o valor de *verdade* executa-se a instrução; no caso contrário não se faz nada.

Tendo isso em conta vejamos então as instruções de iteração e condicional.

Instrução de Iteração

$$\texttt{faz \_ fimfaz} : (\texttt{Comandos Guardados})^* \rightarrow \texttt{Instruções}$$

Dado uma lista, eventualmente vazia, de comandos guardados, escolhe-se aleatoriamente um dos comandos guardados que possua um guarda com valor *verdade*, executando-se de seguida o comando correspondente. Repete-se o processo até que todos os guardas tenham o valor *falso*. No caso da lista de comandos guardados ser vazia, ou todos os guardas terem logo à partida o valor *falso*, a instrução de iteração é equivalente à instrução nula.

Instrução Condicional

$$\texttt{se \_ fimse} : (\texttt{Comandos Guardados})^* \rightarrow \texttt{Instruções}$$

Dado uma lista, eventualmente vazia, de comandos guardados, escolhe-se aleatoriamente um dos comandos guardados que possua um guarda com valor *verdade*, e executa-se o comando correspondente. Se todos os guardas tiverem o valor falso, ou no caso de uma lista vazia de comandos guardados, a instrução é equivalente à instrução nula.

Funções

Declaração de função

$$\begin{aligned}&\texttt{funcao \_(\_)var(\_)\{\_\}\_\{\_\}fimfuncao} : \texttt{Elementos } (\texttt{Elementos})^* \\ &\quad (\texttt{Elementos})^* \texttt{ Proposições Instruções Proposições} \\ &\quad \rightarrow \texttt{Proposições}\end{aligned}$$

Invocação de função

$\_(\_) : \texttt{Elementos}\ (\texttt{Elementos})^* \rightarrow \texttt{Elementos}$

O efeito da invocação da função é o da execução do corpo da função tal como está definido na declaração de função. Os argumentos substituem os parâmetros formais, sendo o valor calculado atribuído ao identificador de função.

Como se pode deduzir pelas espécies respectivas a declaração de função não contribui para a operacionalidade do programa directamente dado que o seu contradomínio é de espécie lógica. O contributo operacional está representado na invocação da função.

Assume-se que: os argumentos estão em correspondência com os parâmetros formais; que não é feita nenhuma atribuição a variáveis que não sejam locais ao corpo da função; que todos os valores dos argumentos são copiados para os parâmetros formais ("copia-por-valor"); e que o identificador de função é usado como uma variável (célula de memória) cujo conteúdo é o valor da função.

Sub-programas

Declaração de sub-programa

$\texttt{subprg\_}(\_,\_)\texttt{var}(\_)\{\_\}\_\{\_\}\texttt{fimsubprg} : \texttt{Elementos}\ (\texttt{Elementos})^*\ (\texttt{Elementos})^*\ (\texttt{Elementos})^*\ \texttt{Proposições}\ \texttt{Instruções}\ \texttt{Proposições} \rightarrow \texttt{Proposições}$

Invocação de sub-programa

$\_(\_,\_) : \texttt{Elementos}\ (\texttt{Elementos})^*\ (\texttt{Elementos})^* \rightarrow \texttt{Instruções}$

O efeito da invocação de um sub-programa é o da execução do corpo do procedimento tal como está na sua declaração. Os parâmetros formais são substituídos pelos correspondentes parâmetros reais. Os valores fornecidos ao sub-programa, valores de entrada, constituem a primeira lista de variáveis e os valores fornecidos pelo sub-programa, valores de saída, constituem a segunda lista de variáveis.

Assume-se que: não é feita nenhuma atribuição no corpo do sub-programa a variáveis que não as locais ao corpo do sub-programa; que todos os valores de entrada, são copiados dos parâmetros reais para os respectivos parâmetros formais no princípio da execução do sub-programa; e que todos os valores de saída são copiados dos parâmetros formais para os respectivos parâmetros reais no fim da execução do sub-programa.

Um programa em $LPCG$ consistirá numa zona de declaração de variáveis e constantes; noutra de declaração de funções e sub-programas e num corpo principal contendo as instruções a serem executadas.

Temos então

`prg _ var(_)const(_)_{_}_{_}fimprg` : `Elementos` (`Elementos`)* (`Elementos`)* (funções e sub-programas)* `Proposições` `Instruções` `Proposições` $\rightarrow$ $Mensagens$

O resultado da execução de um programa em $LPCG$ será dado por uma mudança no Estado, i.e. nos valores das variáveis (posições de memória) declaradas no começo do programa e nas mensagens enviadas pelo programa de entre um conjunto de mensagens que eventualmente o programa tenha definido.

Os tipos de dados em $LPCG$ são introduzidos pela espécie `Elementos`. Sempre que seja necessário ter um tipo de dados concreto, e.g. os inteiros, pode-se identificar a espécie `Elementos` com a espécie do tipo de dados que se quer introduzir, ou considerar que a espécie `Elementos` é a união disjunta da(s) espécie(s) que se pretenda(m) introduzir.

Vejamos um exemplo; um programa para o cálculo do máximo divisor de dois número naturais recorrendo a uma função que implementa o algoritmo de Euclides.

```
prg mdc
 var (x,y,res)

 const ()

 funcao euclides(x,y)
  var (a,b)

  {x > 0  ∧  y > 0  ∧  mdc(x,y) = mdc(x,y)}
  a := x;
  {a > 0  ∧  y > 0  ∧  mdc(a,y) = mdc(x,y)}
  b := y;
  {a > 0  ∧  b > 0  ∧  mdc(a,b) = mdc(x,y)}
  faz (~ (a = b) ->
     faz ((a > b) ->
         a := a - b)
     fimfaz;
     {a > 0  ∧  b > 0  ∧  mdc(a,b) = mdc(x,y)  ∧  ¬ (a > b)}
     faz((b > a) ->
         b := b - a)
     fimfaz)
```

```
  fimfaz;
  {a > 0 ∧ b > 0 ∧ mdc(a,b) = mdc(x,y) ∧ a = b}
  euclides := a
  {a > 0 ∧ b > 0 ∧ euclides(a,b) = mdc(x,y)}
 fimfuncao

{Verdade}
ler(x,y);
{x > 0 ∧ y > 0}
res := euclides(x,y);
{x > 0 ∧ y > 0 ∧ res = mdc(x,y)}
escrever(res)
{x > 0 ∧ y > 0 ∧ res = mdc(x,y)}
fimprg
```

Nota: `ler(_)` e `escrever(_)` vão ser consideradas como sub-programas pré-definidos e que têm o efeito de múltiplas instruções de atribuição, estabelecendo desse modo a comunicação entre o programa e o exterior.

2.3 Definição axiomática da linguagem

A definição axiomática da linguagens (Alagič & Arbib, 1978; Clint & Hoare, 1972; Cousot, 1990; Gries, 1981; Hoare, 1971; Hoare, 1972; Hoare, 1973; Meyer, 1990; Quaresma, 1995) vai-se basear na definição de pré-condições e pós-condições, para cada uma das instruções da linguagem de programação. Seguindo a notação usada por Alagič e Arbib (1978), e já usada na definição da sintaxe de $LPCG$ temos;

$$\{P\}S\{Q\}$$

O significado desta proposição é o seguinte; se a proposição P for verdadeira antes da execução da instrução S, então a proposição Q tem de ser verdadeira após o completar de S. Vai-se considerar somente a correcção parcial de programas assumindo-se que não existem situações de bloqueio.

As regras de inferência vão-se basear num cálculo de sequências dedutivas para uma linguagem de primeira ordem, sendo essa linguagem extendida de molde a incluir todas as instruções da linguagem de programação. Os axiomas e regras de inferência respeitantes aos vários tipos de dados são definidos em sistemas de dedução distintos sendo "adicionados" ao sistema que se vai descrever "à medida das necessidades". No capítulo 4 explicita-se mais concretamente como se pretende efectuar a construção modular de sistemas de dedução.

2.3.1 O sistema de dedução $LPCG_{LK}$

O sistema de dedução para a demonstração da correcção parcial de programas escritos na linguagem $LPCG$ vai-se basear no sistema de dedução LK (Gallier, 1987; Szabo, 1969), e vai-se designar por $LPCG_{LK}$.

No que se segue usam-se as seguintes convenções: P, Q e R são fórmulas proposicionais; B, eventualmente com índices, é uma fórmula proposicional em $LPCG$; S, eventualmente com índices, é uma instrução de $LPCG$; H é uma lista de fórmulas de primeira ordem; os símbolos lógicos proposicionais ($\neg$, $\wedge$, $\vee$ e $\supset$) tem o seu significado usual (Gallier, 1987; Hamilton, 1991; Oliveira, 1979); x é uma variável; e é uma constante; a proposição P_e^x denota a proposição P após a substituição de todas as ocorrências livres da variável x pela expressão e (Gallier, 1987; Hamilton, 1991; Oliveira, 1979).

Sequências deductivas básicas:

$$\frac{}{H \vdash \{P\}\text{nula}\{P\}} \quad (\vdash \text{nula})$$

Se P é verdadeira antes da execução da instrução nula então tem de permanecer verdadeira após a sua execução.

$$\frac{}{H \vdash \{P_e^x\}x := e\{P\}} \quad (\vdash \text{atr})$$

Se P é verdadeira substituindo as ocorrências livres de x por e antes da atribuição então P tem de ser verdadeira se se atribui a x o valor de e.

Sequências deductivas não básicas:

$$\frac{H \vdash \{P\}S\{R\} \qquad H \vdash R \supset Q}{H \vdash \{P\}S\{Q\}} \quad (\vdash \text{consd})$$

$$\frac{H \vdash P \supset R \qquad H \vdash \{R\}S\{Q\}}{H \vdash \{P\}S\{Q\}} \quad (\vdash \text{conse})$$

Estas duas regras são designadas por regras do consequente, à direita e à esquerda respectivamente. A primeira destas duas regras afirma que; se a execução de S com pré-condição P, assegura a veracidade de R, então também assegura a veracidade de toda a proposição que seja consequência de R. A segunda regra de inferência afirma que; se R é uma pré-condição para a execução de S com pós-condição Q, então toda a proposição de que R seja consequência é pré-condição de S para assegurar a veracidade de Q.

$$\frac{H \vdash \{P\}S_1\{R\} \qquad H \vdash \{R\}S_2\{Q\}}{H \vdash \{P\}S_1;\{R\}S_2\{Q\}} \quad (\vdash \text{comp})$$

Se a instrução S_1 com pré-condição P assegura a veracidade de R, e se R é pré-condição para que S_2 assegure a veracidade de Q, então P é pré-condição para que a execução de S_2 após S_1 assegure a veracidade de Q.

$$\frac{H \vdash (P \wedge \neg(B_1 \vee \ldots \vee B_n)) \supset Q \;\; H \vdash \{P \wedge B_1\}S_1\{Q\} \ldots H \vdash \{P \wedge B_n\}S_n\{Q\}}{H \vdash \{P\}\texttt{se}(B_1 \rightarrow S_1, \ldots, B_n \rightarrow S_n)\texttt{fimse}\{Q\}} \quad (\vdash \text{se})$$

Se todos os guardas forem falsos, ou a lista de comandos guardados for vazia, a instrução condicional é equivalente à instrução nula. Temos então que, por uma das regras do consequente, a veracidade de $P \wedge \neg(B_1 \vee \ldots \vee B_n) \supset Q$, que neste caso é equivalente a $P \supset Q$, assegura a veracidade de Q como pós-condição da instrução condicional.

Se pelo menos um dos guardas for verdadeiro, e $P \wedge B_i$ for pré-condição para que a instrução S_i assegure a veracidade da pós-condição Q, com $1 \leq i \leq n$, então a instrução condicional assegura a veracidade de Q.

Em conclusão; se $(P \wedge \neg(B_1 \vee \ldots \vee B_n)) \supset Q$ é verdadeira e se $P \wedge B_i$ é pré-condição para que S_i assegure a veracidade da pós-condição Q, com $1 \leq i \leq n$, então P é a pré-condição e Q é a pós-condição da instrução condicional.

$$\frac{H \vdash \{P \wedge B_1\}S_1\{P\} \;\; \ldots \;\; H \vdash \{P \wedge B_n\}S_n\{P\}}{H \vdash \{P\}\texttt{faz}(B_1 \rightarrow S_1, \ldots, B_n \rightarrow S_n)\texttt{fimfaz}\{P \wedge \neg(B_1 \vee \ldots \vee B_n)\}} \quad (\vdash \text{faz})$$

Se todos os guardas forem falsos, ou a lista de comandos guardados for vazia, a instrução iterativa é equivalente à instrução nula. Como neste caso $P \wedge \neg(B_1 \vee \ldots \vee B_n)$ é equivalente a P temos então que $P \wedge \neg(B_1 \vee \ldots \vee B_n)$ é pós-condição da instrução iterativa.

Se pelo menos um dos guardas for verdadeiro, e se para todos os comandos guardados se tem que $P \wedge B_i$ é pré-condição para que S_i assegure a veracidade de P, com $1 \leq i \leq n$; então após a execução da instrução de iteração podemos assegura a veracidade de P, assim como a veracidade de $\neg(B_1 \vee \ldots \vee B_n)$, isto dado que a instrução de iteração termina quando todos os guardas forem falsos. Ou seja após esta instrução pode assegurar-se a veracidade de $P \wedge \neg(B_1 \vee \ldots \vee B_n)$.

$$\frac{}{H \vdash P_a^x \supset Q_{f(a),a}^{f,x}} \quad (\vdash \text{invocafun})$$

com

$$H_i = \texttt{funcao}\, f(x)\texttt{var}(aux)\{P\}S\{Q\}\texttt{fimfuncao}, \quad H_i \in H$$

No caso da regra de inferência (básica) referente à invocação de uma função, *invocafun* pretende-se verificar somente a invocação em si partindo do pressuposto que a veracidade da função já foi estabelecida.

Suponhamos que a execução de S com pré-condição P assegura a veracidade da pós-condição Q, o qual incluirá o nome da função, denotando o resultado da função; e tanto P como Q incluirão a lista de parâmetros

formais x. Temos então que a veracidade de P é suficiente para garantir a veracidade de Q para uma qualquer invocação da função; isto é, com todas as ocorrências de f substituídas por $f(a)$ e com todas as ocorrências de x substituídas por a, sendo a o argumento da função para a invocação em questão.

$$\frac{H \vdash \{P\}S\{Q\}}{H \vdash \texttt{funcao}\, f(x)\texttt{var}(aux)\{P\}S\{Q\}\texttt{fimfuncao}} \quad (\vdash \text{funcao})$$

Para se demonstrar a veracidade de uma função tem de se demonstrar a correcção parcial do corpo da função para uma lista de parâmetros formais genérica, para tal considera-se que as instruções de função quantificam universalmente os parâmetros formais assim como as suas variáveis internas.

$$\frac{}{H \vdash \{P^{x,y}_{e,a}\}p(e,a)\{Q^{x,y}_{e,a}\}} \quad (\vdash \text{invocasubprg})$$

com

$$H_i = \texttt{subprg}\, p(x,y)\texttt{var}(aux)\{P\}S\{Q\}\texttt{fimsubprg}, \quad H_i \in H$$

Novamente só se pretende verificar a correcção da invocação. Suponhamos que se demonstrou a veracidade de $\{P\}S\{Q\}$, onde tanto P como Q incluem os parâmetros formais do sub-programa $p(x,y)$, isto para toda e qualquer lista de parâmetros formais, então para uma invocação do sub-programa o que se quer verificar é a veracidade de $\{P^{x,y}_{e,a}\}p(e,a)\{Q^{x,y}_{e,a}\}$. Mas para que se possa estabelecer a validade da regra de inferência é necessário que se verifique o seguinte: todas as variáveis na lista de parâmetros reais têm de ser distintas e nenhuma delas deve estar contida na lista de parâmetros reais de entrada e (Hoare, 1971; Hoare, 1972).

$$\frac{H \vdash \{P\}S\{Q\}}{H \vdash \texttt{subprg}\, p(x,y)\texttt{var}(aux)\{P\}S\{Q\}\texttt{fimsubprg}} \quad (\vdash \text{subprg})$$

Para se demonstrar a veracidade de um sub-programa tem de se demonstrar a correcção parcial do corpo do sub-programa para uma lista de parâmetros formais genérica. Assim como para as funções vai-se considerar que as instruções de sub-programa quantificam universalmente os seus parâmetros formais assim como as variáveis internas.

$$\frac{H_1 \vdash fp_1 \ \ldots \ H_n \vdash fp_n \; H' \vdash \{P\}S\{Q\}}{H \vdash \texttt{prg}\, T\, \texttt{var}(v)\texttt{const}(c)fp_1,\ldots,fp_n\{P\}S\{Q\}\texttt{fimprg}} \quad (\vdash \text{prg})$$

com

$$H' = H, fp_1, \ldots, fp_n$$

$$H_i = H, fp_1, \ldots, fp_{i-1}, fp_{i+1}, \ldots, fp_n \quad 1 \leq i \leq n$$

com fp_i a declaração de uma função ou de um sub-programa.

Para se demonstrar a correcção parcial de um programa é necessário demonstrar a correcção de todas as suas funções e sub-programas assim como do corpo do programa. Ao fazer-se a demonstração de correcção parcial das diferentes partes do programa juntar-se-á à lista de premissas iniciais as declarações das funções e sub-programas que deste modo se consideram demonstrados, e que podem portanto ser usados para verificar a correcção da sua invocação nas diferentes partes do corpo do programa, assim como no corpo dos outros sub-programas e funções. Não se consideram as chamadas recursivas de sub-programas e funções.

2.4 Implementação de $LPCG_{LK}$

Procedeu-se à implementação do sistema de dedução $LPCG_{LK}$ em dois sistemas computacionais de demonstração automática de teoremas; os sistemas *Isabelle* e *2OBJ*. A escolha destes dois sistemas deveu-se a um certo número de factores, como se tenta explicitar de seguida.

Ambos os sistemas são demonstradores genéricos de teoremas. Ou seja ambos implementam um meta-sistema lógico que permite a definição pelo utilizador de uma dada linguagem lógica, e respectivo sistema de dedução, providenciando os mecanismos necessários para manipular as fórmulas e as regras de inferência do sistema de dedução, permitindo deste modo realizar demonstrações no novo sistema.

Em ambos os casos é possível adoptar um modo interactivo de condução das demonstrações recorrendo a *tácticas* e *tacticais*, o que permite uma grande flexibilidade na construção de algoritmos de demonstração, adaptando-os deste modo aos vários problemas que se querem tratar.

Ambos os sistema são baseados em linguagens de programação de alto nível as quais são acessíveis a partir dos respectivos sistemas de dedução; o *Isabelle* está baseado no *Standard ML* (Harper, 1986; Paulson, 1991; Wikström, 1989) e o *2OBJ* está baseado no *OBJ3* (Goguen *et al.*, 1992).

Finalmente, ambos os sistemas possuiam à partida um sistema de dedução para sequências deductivas, sistema no qual se pretendia basear o cálculo $LPCG$, mais concretamente o sistema LK.

Descrevem-se de seguida as duas implementações de $LPCG_{LK}$.

2.4.1 A implementação de $LPCG_{LK}$ no *Isabelle*

O *Isabelle* está baseado na linguagem de programação *Standard ML* (*New Jersey ML* ou *Polly ML*); representa a sintaxe dos sistemas lógicos que implementa num cálculo-λ com tipos e polimórfico; o seu meta-sistema lógico

é um sistema lógico de ordem superior. As meta-conectivas lógicas são: implicação, `_ ==> _`, que expressa consequência lógica; quantificação universal, $\bigwedge$ `_ . _`, que expressa generalização em esquemas de regras de inferência e axiomas; igualdade, `_ == _`, que expressa definição. Para relacionar o meta-nível com o nível objecto o *Isabelle* declara o operador `Trueprop` que expressa a veracidade ao nível objecto.

O cálculo de Sequências deductivas no qual se vai basear a implementação de $LPCG_{LK}$ é uma variante do sistema LK proposto por Gentzen (Paulson, 1993c).

A definição de um sistema de dedução em *Isabelle* passa pela escrita de uma `teoria` na qual se especificam: as `teorias` que vão servir de suporte; as novas espécies introduzidas, assim como as relações de ordem entre elas e entre estas e as anteriormente definidas; os vários símbolos de operadores, constantes, funções e predicados; e finalmente, os esquemas de axiomas e regras de inferência (Paulson, 1993a). A sintaxe usada para tal não é a do *Standard ML* mas uma meta-sintaxe em alguns aspectos semelhante à do *OBJ3*. Esta é depois transcrita, automaticamente, para uma *estrutura ML* (Harper, 1986; Paulson, 1991; Wikström, 1989).

Vai-se de seguida explicitar o que se disse usando a implementação de $LPCG_{LK}$ como exemplo.

Temos como primeira declaração:

```
LPCG_LK = LK +
```

ou seja vai-se estender (adicionar a) LK com novas espécies, novos símbolos de operadores e novos esquemas de regras de inferência.

De seguida explicitam-se as espécies que se pretendem introduzir bem como o seu ordenamento relativo e os seus construtores, caso seja necessário. A espécie pré-definida `term` dá-nos os *termos* da linguagem.

```
classes
 elems, inst < term
types
 elems, inst 0
arities
 elems, inst :: term
```

Definem-se de seguida as *constantes*, ou seja os símbolos de função e os símbolos de predicado. A espécie pré-definida "`o`" dá-nos os elementos do tipo lógico.

```
consts
 nula :: "inst""
 atr  :: "[elems, elems] => inst"  ("_:=_")
 comp :: "[inst,o,inst] => inst"  ("_;{_}_" [0,0,201] 200)
```

```
...
 prepos :: "[o,inst,o] => o"      ("{_}_{_}" [0,0,0] 100)
```

do lado direito tem-se a definição do nome do operador, que tem de obedecer a algumas restrições próprias do *Standard ML*, assim como a sua aridade (ao meio): do lado direito tem-se a definição de uma sintaxe alternativa assim como a informação referente à precedência e associatividade dos vários operadores. Temos assim que a expressão;

```
{P1}S1;{P2}S2;{P3}S3{P4}
```

é entendida do seguinte modo;

```
{P1}(S1;{P2}(S2;{P3}S3)){P4}
```

Seguem-se as definições respeitantes aos esquemas de regras de inferência e axiomas:

```
rules
 nulaD "$H |- $E,{P}nula{P},$F"
 atrD  "$H |- $E,{P(e)}x:=e{P(x)},$F"
 compD "[|$H |- $E,{P}S1{R},$F ;      \
\         $H |- $E,{R}S2{Q},$F|] ==>  \
\         $H |- $E,{P}S1;{R}S2{Q},$F"
 ...
```

em que `$H, $F, $E` representam listas de fórmulas. Os símbolos "\" são símbolos auxiliares do *Standard ML* para partir uma sequência de caracteres ("string") em várias linhas.

Os axiomas e regras de inferência são escritos como fórmulas ao meta-nível, "`[| ...|]`" dá-nos a possibilidade de explicitar uma lista de fórmulas. Todas as variáveis estão, implicitamente, quantificadas universamente. Como exemplo podemos ver que a leitura do último operador, `compD`, é o seguinte esquema de regra de inferência:

$$\frac{\Gamma \vdash \Delta, \{P\}S1\{R\}, \Psi \quad \Gamma \vdash \Delta, \{R\}S2\{Q\}, \Psi}{\Gamma \vdash \Delta, \{P\}S1;\{R\}S2\{Q\}, \Psi}$$

A substituição de variáveis é implementada através da redução β, temos assim que a leitura do operador `atrD` é:

$$\frac{}{\Gamma \vdash \Delta, \{P^x_e\}x := e\{P\}, \Psi}$$

Dado o modo como o *Isabelle* trata o problema da substituição de variáveis (ver (Paulson, 1993a, pag 8)) não é possível explicitar na regra `atrD`, que o que se pretende é $\{P^x_e\}x := e\{P\}$ e não $\{P\}x := e\{P^e_x\}$. Tal facto

leva a que as demonstrações em que se use a regra `consdD` precedendo a aplicação da regra `atrD` podem estar erradas. O mesmo facto levanta problemas aquando da implementação das regras de inferência para funções e sub-programas dado não ser possível assegurar as condições de aplicabilidade das mesmas.

Associado ao ficheiro que contém a informação que se acabou de descrever e que tem, obrigatoriamente, a extensão `thy`, está associado um outro, com mesmo nome, mas com extensão `ML`, no qual se escrevem comandos e estruturas *Standard ML* auxiliares.

No caso da definição de um sistema de dedução pode-se usar este ficheiro para definir tácticas e tacticais com vista à automatização das demonstrações; no caso de se estar a definir não um sistema de dedução completo mas tão somente um *teorema* que se pretenda demonstrar usa-se o ficheiro de extensão `thy` para definir eventuais operadores e axiomas necessários para o desenvolvimento da demonstração, e usa-se o ficheiro de extensão `ML` para a escrita do *teorema* que se pretende demonstrar.

2.4.2 Automatização da demonstração no *Isabelle*

O principal método de demonstração no sistema *Isabelle* é a regra de resolução (Chang & Lee, 1973; Gallier, 1987; Paulson, 1993a); para que num sistema de dedução previamente definido, se possa usar este método torna-se necessário construir *"packs"* respeitantes às regras de inferência do sistema que se definiu (Paulson, 1993a; Paulson, 1993c). No que se segue vai-se designar os *"packs"* de regras de inferência por *agrupamentos* de regras de inferência. Um agrupamento é um par de listas especificando quais as regras de inferência que podem ser usadas para tentar encontrar um resolvente. As regras de inferência são separadas em regras seguras e regras não seguras consoante respeitem ou não a propriedade da sub-fórmula.

Definiram-se assim os agrupamentos, `lpcg_pack` e `LPCG_pack`, sendo que o primeiro é construído a partir do agrupamento pré-definido `prop_pack` (*"propositional pack"*), adicionando a este as regras seguras definidas em `LPCG_LK`, a saber: `nulaD`, `compD`, `atrD`, `seD` e `fazD`. O segundo é construído a partir do primeiro por adição das regras não seguras, `conseD` e `consdD`.

Definiram-se ainda os seguintes tacticais, `consdD_tac`, `conseD_tac` e `lpcg_tac`. Os dois primeiros dizem respeito à aplicação das regras de inferência `consdD` e `conseD` respectivamente, aceitando um argumento, ou seja a fórmula a introduzir. O tactical `lpcg_tac` diz respeito à automatização da demonstração em $LPCG_{LK}$ por aplicação da regra da resolução às regras seguras e, no caso de tal não ser possível, às regras não seguras, repetindo todo o processo até se obter a demonstração pretendida. Devido à presença das regras não seguras, o tactical `lpcg_tac` pode entrar em ciclo, não conseguindo portanto terminar a demonstração como se pretendia.

2.4.3 A implementação de $LPCG_{LK}$ no *2OBJ*

O *2OBJ* está baseado no *OBJ3*, tem como meta-lógica a lógica equacional multi-espécies, com ordenamento entre estas (Goguen *et al.*, 1992). A definição de um sistema de dedução passa pela construção de um *objecto* em *OBJ3* especificando as espécies, operadores e equações. O sistema de dedução pré-definido, *FOPC* (*"First Order Predicate Calculus"*) é uma variante do sistema LJ com as sequências deductivas do tipo $\Gamma \vdash P$ com Γ uma sequência, eventualmente vazia, de fórmulas e P uma única fórmula. O adicionar da regra do terceiro excluído permite trabalhar na lógica clássica sendo o sistema uma variante do sistema LK (Stevens & Hobley, 1993).

No sistema *FOPC* todas as regras de eliminação da conectiva lógica à esquerda necessitam de explicitar qual a posição da fórmula para a qual se pretende aplicar a regra de inferência; para obstar a isso e permitir uma maior automatização da demonstração, construiu-se o sistema de dedução *CPPO* ("Cálculo de Predicados de Primeira Ordem"), em que se modificaram as regras de eliminação da conectiva lógica à esquerda de forma a não ser necessário explicitar a posição relativa da fórmula.

A falta de um mecanismo de unificação para as sequências deductivas leva a que, as regras de eliminação do quantificador existencial à esquerda, a regra de eliminação do quantificador existencial à direita, assim como as regras do consequente das funções e sub-programas, no cálculo $LPCG_{LK}$ tenham todas um argumento de entrada.

Como já foi referido a implementação de um sistema de dedução em *2OBJ* passa pela definição de um *objecto* em *OBJ3*, no qual se especifica; o sistema de dedução em que nos estamos a basear; as novas espécies e o seu ordenamento relativo; os novos operadores (constantes, funções e predicados); e os axiomas e regras de inferência. É de notar que, ao contrário do sistema *Isabelle*, o *2OBJ* não tem explícito um sistema *Pure* que contenha só a meta-lógica.

Temos assim que a definição de um novo sistema de dedução tem de ter como primeira declaração o *importar* de *CPPO* (Goguen *et al.*, 1992):

```
obj LPCG_LK is
 protecting CPPO .
```

De seguida tem-se a definição das espécies referentes aos elementos (`Elems`) e às instruções (`Inst`) e o seu ordenamento relativo.

```
sorts Elems , Inst .
sorts Var-Elems , Var-Inst .
subsorts Elems , Inst < Term .
subsorts Var-Elems , Var-Inst < Var-Term .
```

as sub-espécies `Var-Elems` e `Var-Inst` são necessárias para introduzir as variáveis de cada espécie de uma forma explicita (Stevens & Hobley, 1993).

A espécie `Term` (termos), `Var` (variáveis) e `Sent` (sentenças), são espécies pré-definidas em *CPPO*.

Temos de seguida a definição dos operadores para os símbolos de constantes, funções e predicados:

```
op nula : -> Inst .
op _:=_ : Elems Elems -> Inst [prec 0] .
op _;{_}_ : Inst Sent Inst -> Inst [prec 19 gather(e e E)] .
...
op (funcao_(_)vars(_){_}_{_}fimfuncao) :
   Elems Universal Universal Sent Instruc Sent -> Prop .
op _[_] :
   Elems Universal -> Elems .
...
op (prg_vars(_)consts(_)(_){_}_{_}fimprg) :
 Elems Universal Universal Universal Sent Instruc Sent -> Prop .
op {_}_{_} : Sent Inst Sent -> Prop [prec 20] .
```

além do nome do operador e da sua aridade pode especificar-se a precedência do operador, assim como a sua associatividade, de molde a poder simplificar a escrita das frases. Por exemplo:

```
{P1}S1;{P2}S2;{P3}S3{P4}
```

é interpretado como

```
{P1}(S1;{P2}(S2;{P3}S3)){P4}
```

Os operadores referentes às regras de inferência têm co-aridade `Rule`. Esta é uma espécie pré-definida no *2OBJ* que nos permite definir operadores cuja operacionalidade é nos dada pela escrita de equações para o operador pré-definido `_ _ : Rule Goal -> Goalist`, este operador permite-nos definir as regras de inferência que pretendemos usar.

Vejamos um exemplo; primeiro definem-se os símbolos dos operadores:

```
ops nulaD atrD seD fazD compD funD prgD : -> Rule .
ops conseD consdD : Sent -> Rule .
ops chamafunD : Universal NzInt -> Rule .
```

e de seguida as equações que definem os operadores:

```
eq nulaD (H |- {P}nula{P}) = [] .
eq conseD(R) (H |- {P}S{Q}) =
             (H |- (P -> R)),
             (H |- {R}S{Q}) .
ceq atrD (H |- {Q}X:=Y{P}) = []
```

```
         if Q == (P o Y / X) .
...
ceq chamafunD(V1,N) (H |-  (P -> Q)) = []
  if not(livrepara(V1,(hyp(N,H) : 3)))
  and P == (hyp(N,H) : 4) o V1 / (hyp(N,H) : 2)
  and Q == ((hyp(N,H) : 6) o (((hyp(N,H) : 1) [ V1 ]) ;; V1)
                        / ((hyp(N,H) : 1) ;; (hyp(N,H) : 2))) .
eq funD (H |- funcao nome(XS)vars(V1){P}S{Q}fimfuncao ) =
        (H |- {P}S{Q}) .
```

ao contrário do *Isabelle* a definição de um sistema de dedução em *2OBJ* é feito directamente em *OBJ3*, tal facto obriga à definição prévia de todas as variáveis com as aridades respectivas.

Nos exemplos anteriores é de notar que; `[]` dá-nos a lista de fórmulas vazia, sendo assim `nulaD` e `atrD` são sequências deductivas básicas, ou seja axiomas; ao contrário destas duas `conseD` necessita de um argumento, `R`, e tem dois sub-objectivos; as equações referente a `atrD` e a `chamafunD` são equações condicionais sendo que a primeira só é aplicável no caso em que `Q == P o Y / X`, ou seja sempre que $Q = P_y^x$, sendo que o operador `_ == _` dá-nos a meta-igualdade, e o operador `_ o _ / _` dá-nos a substituição de variáveis numa fórmula. As condições de aplicabilidade da segunda são um pouco mais complexas, o que é devido à complexidade das restrições que se tem de impor, para a aplicação da regra de inferência que faz a invocação de uma função.

2.4.4 Automatização da demonstração no *2OBJ*

Falta referir a implementação dos mecanismos necessários à automatização das demonstrações em *2OBJ*.

O *2OBJ* não possui implementado a regra de resolução, nem o algoritmo de unificação para sequências deductivas, tendo somente os mecanismos de demonstração interactiva, ou seja as tácticas e tacticais.

Temos então que, para automatizar a demonstração em $LPCG_{LK}$, é necessário definir operadores de co-aridade `Tactic`, e equações para o operador `_ _ : Tactic Goal -> Prooftactic`, em que se especifica a operacionalidade que se pretende para as tácticas e tacticais.

Para $LPCG_{LK}$ definiu-se o seguinte tactical.

```
op LpcgTac : -> Tactic .
eq LpcgTac (H |- {P}nula{Q}) =
             nulaD ELSE idtac .
eq LpcgTac (H |- {P}X:=Y{Q}) =
             atrD ELSE idtac .
eq LpcgTac (H |- {P}S1;{R}S2{Q}) =
```

```
              (compD THEN LpcgTac) ELSE idtac .
 ...
eq LpcgTac (H |- funcao nome(XS)vars(V1){P}S{Q}fimfuncao) =
        (funD THEN LpcgTac) ELSE idtac .
eq LpcgTac (H |- prg nome vars(XS)consts(CS)(SUBS){P}S{Q}
                  fimprg) =
        (prgD THEN LpcgTac) ELSE idtac .
eq LpcgTac DEFAULT G = idtac .
```

este tactical tenta aplicar recursivamente todas as regras seguras parando no caso em que tal já não é possível; `idtac` é o elemento neutro dos tacticais sendo o seu efeito sobre a demonstração nulo.

2.5 Apresentação de alguns exemplos

2.5.1 Algoritmo de divisão de números inteiros

Dado o algoritmo de divisão de dois inteiros usando somente adições e subtracções, pretende-se verificar a sua correcção parcial.

O programa é:

```
prg Divisao
 var (x,y,q,r)

```
$\{Verdade\}$
```
ler(x,y);
```
$\{x = 0 \times y + x\}$
```
r:=x;
```
$\{x = 0 \times y + r\}$
```
q:=0;
```
$\{x = q \times y + r\}$
```
faz (y<=r ->
     r:=r-y;
```
$\{x = q \times y + y + r\}$
```
     q:=q+1)
fimfaz;
```
$\{x = q \times y + r \ \wedge \ y > r\}$
```
escrever(q,r)
```
$\{x = q \times y + r \ \wedge \ y > r\}$
```
fimprg
```

Uma primeira observação que podemos fazer refere-se aos "elementos" usados no programa. Até aqui não se fez nenhuma referência aos tipos de dados sendo que no sistema $LPCG_{LK}$ a única referência que se tem é a de

que os programas manipulam entidades da espécie *elementos*. O objectivo é o de separar as duas entidades; por um lado os programas, e por outro os tipos de dados. Temos então que a espécie `Elems` deve ser vista como a união disjunta de todos os tipos de dados que possam ser manipulados pelos programas, mas que o considerar de um tipo de dados concreto só faz sentido aquando de uma demonstração de correcção concreta.

No caso presente ter-se-ia de considerar um sistema de dedução para uma linguagem de primeira ordem com igualdade, cuja assinatura mono-espécie contêm os símbolos de operadores, `+, -, *`, e os símbolos de predicado, `>` e `<=`. Ter-se-ia ainda de incluir os seguintes axiomas:

$$\forall_x \quad 0 \times x = 0 \tag{2.1}$$

$$\forall_x \quad 0 + x = x \tag{2.2}$$

$$\forall_{q,y,r} \quad q \times y + y + r - y = q \times y + r \tag{2.3}$$

$$\forall_{q,y,r} \quad q \times y + y + r = (q+1) \times y + r \tag{2.4}$$

$$\forall_{y,r} \quad \neg(y \leq r) \equiv y > r \tag{2.5}$$

De seguida apresenta-se a demonstração da correcção parcial do programa, não sob a forma de uma árvore de demonstração, dado a dificuldade de representação da mesma, mas sob a forma de uma tabela. À esquerda os vários passos da demonstração de correcção com as etiquetas numéricas a representar as várias "ramificações". À direita a justificação dos diferentes passos dados na demonstração de correcção. Assume-se que é feita uma correcta entrada de valores.

	$\{Verdade\}$	$\{Verdade\} \supset x = x \overset{2.2}{\supset} x = 0 + x \overset{2.1}{\supset} x = 0 \times y + x$
1	`ler(x,y)`	Não afecta a demonstração
	$\{x = 0 \times y + x\}$	
2	`r:=x;`	$(x = 0 \times y + r)^r_x \equiv x = 0 \times y + x$
	$\{x = 0 \times y + r\}$	
3	`q:=0;`	$(x = q \times y + r)^q_0 \equiv x = 0 \times y + r$
	$\{x = q \times y + r\}$	
4	`faz (y<= r ->`	$P \wedge \neg B \overset{2.5}{\supset} x = q \times y + r \wedge y > r$
	`r:=r-y`	
	$\{x = q \times y + y + r\}$	
	`q:=q+1)`	
	`fimfaz`	
	$\{x = q \times y + r \wedge y > r\}$	
demonstração do ciclo		
	$\{x = q \times y + r \wedge y <= r\}$	$P \wedge B \supset x = q \times y + r \overset{2.3}{\supset} x = q \times y + y + r - y$
4.1	`r:=r-y;`	$(x = q \times y + y + r)^r_{r-y} \equiv x = q \times y + y + r - y$
	$\{x = (q + 1) \times y + r\}$	$x = q \times y + y + r \overset{2.4}{\supset} x = (q + 1) \times y + r$
4.2	`q:=q+1`	$(x = q \times y + r)^q_{q+1} \equiv x = (q + 1) \times y + r$
	$\{x = q \times y + r\}$	P
	$\{x = q \times y + r \wedge y > r\}$	
5	`escrever(q,r)`	Não afecta a demonstração
	$\{x = q \times y + r \wedge y > r\}$	

O algoritmo de divisão no *Isabelle*

Para efectuar a demonstração da correcção parcial do programa que implementa o algoritmo de divisão, no sistema *Isabelle*, é necessário construir uma teoria que defina o sistema de dedução em que se vai desenvolver a demonstração. Temos assim um ficheiro "`divisao.thy`" onde se declara a teoria `Divisao` como sendo uma extensão de `LPCG`.

Optou-se por especificar somente o necessário para o desenvolvimento da demonstração, sendo assim, usou-se o mecanismo de definição e ordenamento de espécies para definir a espécie `int` como uma sub-espécie de `elems`. De seguida definiram-se os símbolos das constantes e dos operadores necessários:

```
Divisao = LPCG +

types
 int 0
```

```
arities
 int :: elems

consts
 "0"  :: "int" ("0")
 "1"  :: "int" ("1")
 "+"  :: "[int,int] => int" (infixl 65)
 "-"  :: "[int,int] => int" (infixl 65)
 "*"  :: "[int,int] => int" (infixl 70)
 leq  :: "[int,int] => o" ("_ <= _" [0,0] 50)
 deq  :: "[int,int] => o" ("_ .=. _ " [0,0] 50)

rules
 ax1 "$H, X .=. Q*Y+R, $K |- $E, X .=. Q*Y+Y+(R-Y), $F"
 ax2 "$H, X .=. Q*Y+Y+R, $K |- $E, X .=. (Q+1)*Y+R, $F"
end
```

Os axiomas `ax1` e `ax2` são casos particulares construídos de forma a permitir a conclusão da demonstração.

De seguida escreveu-se o ficheiro `Divisao.ML`, este ficheiro tem, obrigatoriamente, o mesmo nome do que o ficheiro anterior mas extensão `ML`, colocam-se aí as declarações dos operadores necessários para proceder à demonstração, neste caso a definição do agrupamento de tácticas `Divisao_pack`, o qual inclue os axiomas `ax1` e `ax2`, e o programa do qual se quer obter a demonstração de correcção parcial.

```
open Divisao;

val Divisao_pack = LPCG_pack add_safes [ax1,ax2];

goal Divisao.thy "|-                          \
\ {x .=. 0*y+x}                               \
\ (r:=x);{x .=. 0*y+r}                        \
\ (q:=0);{x .=. q*y+r}                        \
\ faz < (y <= r),(r:=r-y);{x .=. q*y+y+r} \
\              (q:=q+1) > : GCNil             \
\ fimfaz {x .=. q*y+r & ~ y<=r}";
```

A demonstração pode-se desenvolver iterativamente, ou automaticamente com recurso ao agrupamento `Divisao_pack`.

Sem automatização

$$\dfrac{\dfrac{\dfrac{}{-}\,\text{atrD}\;\;\dfrac{}{-}\,\text{atrD}}{1\qquad 2}\,\text{compD}}{1}\qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{-}\,\text{ax1}}{1}\,\text{conjL}}{1}\,\text{impD}\;\;\dfrac{}{-}\,\text{atrD}}{1\qquad 2}\,\text{conseD}}{1}\qquad \dfrac{\dfrac{\dfrac{\dfrac{}{-}\,\text{ax2}}{1}\,\text{impD}\;\;\dfrac{}{-}\,\text{atrD}}{1\qquad 2}\,\text{conseD}}{2}\,\text{compD}}{1}\,\text{compD}$$

$$\dfrac{}{\{P\}S\{Q\}}\;\text{by (lpcg\_tac Divisao\_pack 1);}$$

Com automatização

Foi possível proceder à demonstração de correcção parcial do programa de uma forma totalmente automática. No entanto é de notar que o facto de não existirem pré-definidas em LK regras para a igualdade obriga à inclusão dos axiomas `ax1` e `ax2`.

A apresentação do sistema de dedução em *2OBJ* para a demonstração de correcção parcial do algoritmo de divisão, assim como a discussão da sua demonstração de correcção parcial, é feita no apêndice A.

A comparação entre os dois sistemas de demonstração automática vai ser feita na secção que se segue, onde é apresentado um exemplo um pouco mais complexo e no qual se vão usar funções.

2.5.2 Algoritmo de Euclides

Pretende-se construir um programa que determine o máximo divisor comum de dois naturais (mdc(m,n)), usando para tal o algoritmo de Euclides.

```
ler(m,n);
a:=m;
b:=n;
faz (~(a = b) ->
     faz (a > b, a:=a-b) fimfaz;
     faz (b > a, b:=b-a) fimfaz
fimfaz;
escrever(a)
```

Para exemplificar a demonstração de correcção parcial de programas que contenham sub-programas, no caso presente funções, vai-se desenvolver a demonstração de correcção parcial do seguinte programa:

```
prg Divisao
 var (m,n,res)

 funcao euclides(m,n);
 var (a,b)

 {m > 0  ∧  n > 0}
 a:=m;
 {a > 0  ∧  n > 0  ∧  mdc(a,n) = mdc(m,n)}
 b:=n;
 {a > 0  ∧  b > 0  ∧  mdc(a,b) = mdc(m,n)}
 faz (~(a = b) ->
      faz (a > b ->
           a:=a-b)
      fimfaz;
      {a > 0  ∧  b > 0  ∧  mdc(a,b) = mdc(m,n)  ∧  ¬ a > b}
      faz (b > a ->
           b:=b-a)
      fimfaz
 fimfaz;
 {a > 0  ∧  b > 0  ∧  mdc(a,b) = mdc(m,n)  ∧  a = b}
 euclides:=a
 {euclides = mdc(m,n)}
 fimfuncao

{Verdade}
ler(x,y);
{x > 0  ∧  y > 0  ∧  mdc(x,y) = mdc(x,y)}
res:=euclides(x,y);
{x > 0  ∧  y > 0  ∧  res = mdc(x,y)}
escrever(res)
{x > 0  ∧  y > 0  ∧  res = mdc(x,y)}
fimprg
```

Para a demonstração de correcção parcial deste programa tem de se construir um sistema de dedução para uma linguagem de primeira ordem com igualdade cuja assinatura mono-espécie, contêm os símbolos de operadores, `-` e `mdc`, e o símbolo de predicado `>`. Além disso terá de incluir os seguintes axiomas:

$$\forall_{x,y} \quad x > 0 \wedge x > y \quad \supset \quad x - y > 0 \tag{2.6}$$

$$\forall_{x,y} \quad x > y \quad \supset \quad \mathrm{mdc}(x-y,y) = \mathrm{mdc}(x,y) \tag{2.7}$$

$$\forall_{x,y} \quad \mathrm{mdc}(x,y) \quad = \quad \mathrm{mdc}(y,x) \tag{2.8}$$

$$\forall_x \quad \text{mdc}(x,x) \quad = \quad x \tag{2.9}$$

Na demonstração de correcção parcial do programa assume-se que é feita uma correcta introdução de valores.

	$\{Verdade\}$	
1	`ler(x,y);`	Não afecta a demonstração
	$\{x > 0 \wedge y > 0\}$	$x > 0 \wedge y > 0 \overset{\text{euclides}}{\supset} \text{euclides}(x,y) = mdc(x,y)$
	$\{\text{euclides}(x,y) = \text{mdc}(x,y)\}$	
2	`res:=euclides(x,y);`	$(res = \text{mdc}(x,y))^{res}_{\text{euclides}} \equiv \text{euclides}(x,y) = \text{mdc}(x,y)$
	$\{res = \text{mdc}(x,y)\}$	
3	`escrever(res)`	Não afecta a demonstração
	$\{res = \text{mdc}(x,y)\}$	
demonstração da correcção parcial da função		
	$\{m > 0 \wedge n > 0\}$	$m > 0 \wedge n > 0 \supset m > 0 \wedge n > 0 \wedge \text{mdc}(m,n) = \text{mdc}(m,n)$
	$\{m > 0 \wedge n > 0 \wedge \text{mdc}(m,n) = \text{mdc}(m,n)\}$	
1	a:=m;	$(a > 0 \wedge n > 0 \wedge \text{mdc}(a,n) = \text{mdc}(m,n))^{a}_{m} \equiv m > 0 \wedge n > 0 \wedge \text{mdc}(m,n) = \text{mdc}(m,n)$
	$\{a > 0 \wedge n > 0 \wedge \text{mdc}(a,n) = \text{mdc}(m,n)\}$	
2	b:=n;	$(a > 0 \wedge b > 0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n))^{b}_{n} \equiv a > 0 \wedge n > 0 \wedge \text{mdc}(a,n) = \text{mdc}(m,n)$
	$\{a > 0 \wedge b > 0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)\}$	
3	`faz (~(a=b) ->`	
	`faz(a>b ->`	
	`a:=a-b)fimfaz;`	
	$\{a > 0 \wedge b > 0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge \neg a > b\}$	
	`faz(b>a ->`	
	`b:=b-a)fimfaz)`	
	`fimfaz;`	
	$\{a > 0 \wedge b > 0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge a = b\}$	$P \wedge \neg B$
demonstração do ciclo		
	$\{a > 0 \wedge b > 0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge \neg a = b\}$	$P \wedge B \supset a > 0 \wedge b > 0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)$

	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)\}$	
3.1	`faz (a>b -> a:=a-b) fimfaz;`	
	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge \neg a>b\}$	$P \wedge \neg B$

demonstração do primeiro ciclo interior

	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge a>b\}$	$P \wedge B \overset{2.6 \wedge 2.7}{\supset} a-b>0 \wedge b>0 \wedge \text{mdc}(a-b,b) = \text{mdc}(a,b)$
	$\{a-b>0 \wedge b>0 \wedge \text{mdc}(a-b,b) = \text{mdc}(m,n)\}$	
3.1.1	`a:=a-b`	$(a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n))^{b}_{a-b} \equiv a-b>0 \wedge b>0 \wedge \text{mdc}(a-b,b) = \text{mdc}(m,n)$
	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)\}$	P

	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge \neg a>b\}$	$a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge \neg a>b \supset a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)$
	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)\}$	
3.2	`faz (b>a -> b:=b-a) fimfaz`	
	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)\}$	$P \wedge \neg B \supset a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)$

demonstração do segundo ciclo interior

	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge b>a\}$	$P \wedge B \overset{2.6 \wedge 2.7 \wedge 2.8}{\supset} a>0 \wedge b-a>0 \wedge \text{mdc}(a,b-a) = \text{mdc}(m,n)$
	$\{a>0 \wedge b-a>0 \wedge \text{mdc}(a,b-a) = \text{mdc}(m,n)\}$	
3.2.1	`b:=b-a`	$(a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n))^{b}_{b-a} \equiv a>0 \wedge b-a>0 \wedge \text{mdc}(a,b-a) = \text{mdc}(m,n)$
	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n)\}$	P

	$\{a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge a=b\}$	$a>0 \wedge b>0 \wedge \text{mdc}(a,b) = \text{mdc}(m,n) \wedge a=b \overset{2.9}{\supset} a = \text{mdc}(m,n)$
	$\{a = \text{mdc}(m,n)\}$	
4	euclides:=a	$(\text{euclides} = \text{mdc}(m,n))^{\text{euclides}}_{a} \equiv a = \text{mdc}(m,n)$

$\{\text{euclides} = \text{mdc}(m,n)\}$ |

O algoritmo de Euclides no *Isabelle*

Para proceder à demonstração da correcção parcial do programa que implementa o algoritmo de Euclides, no sistema *Isabelle*, criaram-se os ficheiros "`Mdc.thy`" e "`Mdc.ML`". Pelas razões, já anteriormente expostas, optou-se por escrever um programa em que o cálculo do máximo divisor comum é feito directamente, sem recorrer a uma função.

Começa-se por especificar a teoria aonde se vai desenvolver a demonstração de correcção parcial do programa.

```
MDC = LPCG +

types
 int 0

arities
 int :: elems

consts
 "0"  :: "int" ("0")
 "-"  :: "[int,int] => int" (infixl 65)
 deq  :: "[int,int] => o"   ("_ .=. _ " [0,0] 50)
 mdc  :: "[int,int] => int"
 ">"  :: "[int,int] => o" ("_ > _" [0,0] 50)

rules
 abm0 "$H, A>0, $I, B>0, $J, A>B, $K |- $E, A-B>0,$F"
 bam0 "$H, A>0, $I, B>0, $J, B>A, $K |- $E, B-A>0,$F"
 mdc1 "$H, mdc(A,B) .=. mdc(X,Y), $I, (A>B), $K \
\      |- $E, mdc(A-B,B) .=. mdc(X,Y),$F"
 mdc2 "$H, mdc(A,B) .=. mdc(X,Y), $I, B>A, $K \
\      |- $E, mdc(A,B-A) .=. mdc(X,Y),$F"
end
```

Passando de seguida a declarar os agrupamentos e os tacticais que se definiram para ajudar na demonstração, assim como o programa propriamente dito.

```
open MDC;

val MDC_pack = LPCG_pack add_safes [abm0,bam0,mdc1,mdc2];
```

```
val MDC_tac = ((lpcg_tac lpcg_pack 1)
THEN
 ((conseD_tac "(a>0 & b>0 & mdc(a,b) .=. mdc(x,y) )" 1)
THEN
 ((conseD_tac "(a>0 & b>0 & mdc(a,b) .=. mdc(x,y) )" 3)
THEN
 ((consdD_tac "(a>0 &  b>0 & mdc(a,b) .=. mdc(x,y)) & ~ b>a" 4)
THEN
 (lpcg_tac LPCG_pack 1)))));

goal MDC.thy "|-                                                \
\ {x>0 & y>0 & mdc(x,y) .=. mdc(x,y)}                           \
\ (a:=x);{a>0 & y>0 & mdc (a,y) .=. mdc(x,y)}                   \
\ (b:=y);{(a>0 & b>0 & mdc(a,b) .=. mdc(x,y))}                  \
\ faz < ~ (a .=. b),                                            \
\   faz < a>b, (a:=a-b) > : GCNil fimfaz;                       \
\   {(a>0 & b>0 & mdc(a,b) .=. mdc(x,y)) & (~ (a>b))}           \
\   faz < b>a, (b:=b-a)> : GCNil fimfaz > : GCNil               \
\ fimfaz                                                        \
\ {(a>0 & b>0 & mdc(a,b) .=. mdc(x,y)) & ~(~(a .=. b))}";
```

A demonstração pode-se desenvolver iterativamente, ou automaticamente com recurso ao agrupamento `MDC_pack`.

Sem automatização. Vai-se usar somente o tactical `safe_goal_tac` sobre o agrupamento `prop_pack` de molde a automatizar as demonstrações proposicionais. Vão-se apresentar os diferentes passos necessários para efectuar a demonstração.

```
br compD 1;
br compD 1;
br atrD 1;
br atrD 1;
br fazD 1;
br conseD 1;
br compD 2;
br fazD 2;
br conseD 2;
br atrD 3;
by (safe_goal_tac prop_pack 1);
by (safe_goal_tac prop_pack 1);
br abm0 1;
br mdc1 1;
by (conseD_tac "(a>0 & b>0 & mdc(a,b) .=. mdc(x,y) )" 1);
```

```
by (safe_goal_tac prop_pack 1);
br consdD 1;
br fazD 1;
br conseD 1;
br atrD 2;
by (safe_goal_tac prop_pack 1);
by (safe_goal_tac prop_pack 3);
br bam0 1;
br mdc2 1;
Level 24
No subgoals!
```

Com automatização (sem axiomas suplementares). Isto é, usando o tactical `MDC_tac`.

$$\frac{\underset{1}{\overline{}}\ \text{abm0} \quad \underset{2}{\overline{}}\ \text{mdc1} \quad \underset{3}{\overline{}}\ \text{bam0} \quad \underset{4}{\overline{}}\ \text{mdc2}}{\{P\}S\{Q\}}\ \text{by MDC\_tac;}$$

Com:

$$\begin{aligned}
1 &\equiv a > 0, b > 0, \text{mdc}(a,b) = \text{mdc}(x,y), a > b \vdash a - b > 0 \\
2 &\equiv a > 0, b > 0, \text{mdc}(a,b) = \text{mdc}(x,y), a > b \vdash \text{mdc}(a-b,b) \\
&\quad = \text{mdc}(x,y) \\
3 &\equiv a > 0, b > 0, \text{mdc}(a,b) = \text{mdc}(x,y), b > a \vdash b - a > 0 \\
4 &\equiv a > 0, b > 0, \text{mdc}(a,b) = \text{mdc}(x,y), b > a \vdash \text{mdc}(a,b-a) \\
&\quad = \text{mdc}(x,y)
\end{aligned}$$

Com automatização (com axiomas suplementares) Modificou-se o tactical `MDC_tac` de molde a usar o agrupamento `MDC_pack` no último passo.

$$\frac{\underset{}{\overline{}}}{\{P\}S\{Q\}}\ \text{by MDC\_tac;}$$

O algoritmo de Euclides no *2OBJ*

Como já foi referido a implementação de um sistema de dedução em *2OBJ* passa pela definição de um *objecto* em *OBJ3*, objecto esse que tem de conter: as novas espécies e o seu ordenamento relativo; os novos operadores (constantes, funções e predicados); e os axiomas e regras de inferência.

Temos então o seguinte ficheiro, `Euclides.obj`:

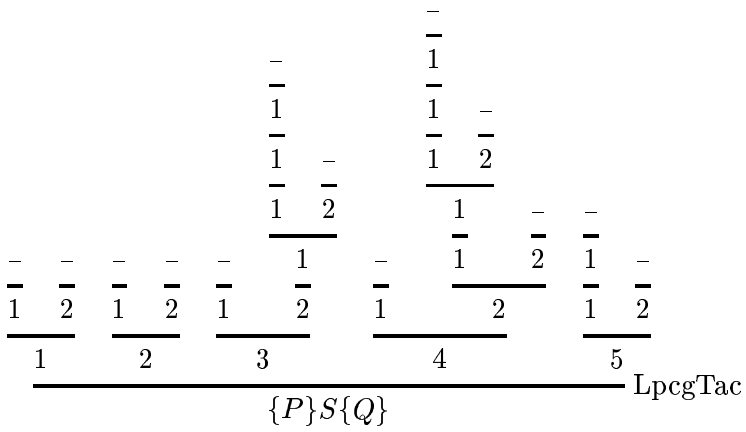
```
obj MDC is
```

```
 pr LPCG .

op _ > _ : Expr Expr -> Sent [prec 51] .
op _ = _ : Expr Expr -> Sent [prec 51] .
ops x y a b euclides Mdc res : -> Var-Elems .
ops mdc : Elems Elems -> Elems .

let Prova = (* |-
prg Mdc vars(x ;; y ;; res ;; euclides) consts (*nil*)
(funcao euclides(x ;; y) vars (a ;; b)
 {(x>0) ^ (y>0)}
 a := x ;
 {(a>0) ^ (y>0) ^ (mdc(a,y)=mdc(x,y))}
 b := y ;
 {(a>0) ^ (b>0) ^ (mdc(a,b)=mdc(x,y))}
 faz << ~(a=b) ;
      faz << (a>b) ; a := (a-b) >> fimfaz ;
      {((a>0) ^ (b>0) ^ (mdc(a,b)=mdc(x,y))) ^ (~(a>b))}
      faz << (b>a) ; b := (b-a) >> fimfaz
 >> fimfaz ;
 {((a>0) ^ (b>0) ^ (mdc(a,b) = mdc(x,y))) ^ (~(~(a=b)))}
 euclides := a
 {(mdc(x,y) = euclides)}
 fimfuncao)
{(x>0) ^ (y>0)}
res := (euclides[(x ;; y)])
{(mdc(x,y)=res)}
fimprg) .

*** Lemas.
ops mdc1 mdc2 mdc3 abm0 bam0 : -> Lemma .
eq lemma mdc1 =
 (* |- |A| a . |A| b . (a>b) -> mdc(a,b) = mdc((a-b),b)) .
eq lemma mdc2 =
 (* |- |A| a . |A| b . mdc(a,b) = mdc(b,a)) .
eq lemma mdc3 =
 (* |- |A| a . |A| b . (a = b) -> (mdc(a,b) = a)) .
eq lemma abm0 =
 (* |- |A| a . |A| b . ((a>b) ^ (a>0)) -> ((a-b) > 0)) .
eq lemma bam0 =
 (* |- |A| a . |A| b . ((b>a) ^ (b>0)) -> ((b-a) > 0)) .
endo
```

A demonstração pode-se desenvolver não automaticamente, ou semi-au-

tomaticamente com recurso ao tactical `Lpcg_tac`. De seguida apresenta-se a demonstração desenvolvida de uma forma semi-automática com uma descrição dos passos e a correspondente árvore de demonstração.

```
LpcgTac
1 conseD(mdc(x,y) = (euclides[x ;; y]))
1 1 chamafunD((x ;; y),1)
1 2 atrD
2 conseD((x>0) ^ (y>0) ^ (mdc(x,y)=mdc(x,y)))
2 1 ProvaProp
2 2 atrD
3 conseD((a>0) ^ (b>0) ^ (mdc(a,b)=mdc(x,y)))
3 1 ProvaProp
3 2 fazD
3 2 1 conseD(((a-b)>0) ^ (b>0) ^ (mdc((a-b),b)=mdc(x,y)))
3 2 1 1 reduze(none,mdc((a-b),b) =' mdc(a,b) [a ;; b | 'mdc1])
3 2 1 1 1 reduze(none,((a-b) > 0) =' True [a ;; b | 'abm0])
3 2 1 1 1 1 ProvaProp
3 2 1 2 atrD
4 conseD((a>0) ^ (b>0) ^ (mdc(a,b)=mdc(x,y)))
4 1 ProvaProp
4 2 consdD(((a>0) ^ (b>0) ^ (mdc(a,b)=mdc(x,y))) ^ ~(b>a))
4 2 1 fazD
4 2 1 1 conseD((a>0) ^ ((b-a)>0) ^ (mdc(a,(b-a))=mdc(x,y)))
4 2 1 1 1 reduze(none,mdc(a,b-a) =' mdc(a,b) [a ;; b | 'mdc2])
4 2 1 1 1 1 reduze(none,((b-a)>0) =' True [a ;; b | 'bam0])
4 2 1 1 1 1 1 ProvaProp
4 2 1 1 2 atrD
4 2 2 ProvaProp
5 conseD(mdc(x,y) = a)
5 1 reduze(none,(mdc(x,y)=a) =' True [x ;; y ;; a | 'mdc3])
5 1 1 ProvaProp
5 2 atrD
```



2.6 Conclusões

A implementação do sistema de dedução $LPCG_{LK}$ nos sistema genéricos de demonstração automática *Isabelle* e *2OBJ* levanta algumas questões. No caso do *Isabelle* o mecanismo de substituição de variáveis não permite validar a aplicação das regras referentes aos sub-programas e às funções, o mesmo acontece com a regra referente à instrução de atribuição embora neste caso o problema só surja se antecedermos a sua aplicação por uma aplicação da regra do consequente à direita.

Os mecanismos de automatização da demonstração no *2OBJ* são insuficientes. Embora se tenha implementado o sistema *CPPO* que permite uma maior automatização das demonstrações proposicionais, isto em contraposição ao sistema pré-definido *FOPC*, a falta de um mecanismo de unificação das cláusulas que permita a implementação da regra de resolução, limita muito as possibilidades do sistema *2OBJ*. Como nota positiva é de referir que o sistema *2OBJ* permite utilizar os mecanismo de inferência equacional, próprios do *OBJ3*, o que possibilita a demonstração de proposições referentes aos tipos de dados, nomeadamente as que necessitam de uma prévia simplificação das expressões.

No caso do *Isabelle* os mecanismos de demonstração pré-definidos permitem um grau elevado de automatização da demonstração sendo que a utilização dos *agrupamentos* permite uma fácil aplicação da regra da resolução a um conjunto de regras de inferência previamente estabelecido. Faltou explorar o mecanismo dos "simp-set" conjuntos de simplificação para a utilização de mecanismos de reescrita na demonstração de igualdade de expressões referentes aos tipos de dados.

A implementação de $LPCG_{LK}$ em ambos os sistema mostrou haver a necessidade de combinar duas entidades diferentes numa só demonstração. Por um lado o sistema de dedução referente às instruções da linguagem, por outro lado um sistema de dedução referente aos tipos de dados utilizados. A criação de um sistema de dedução que possa abarcar todos os casos não parece ser exequível. A resposta a este problema é dada pela construção de um sistema de dedução "à medida" do programa cuja demonstração da correcção parcial se pretende obter. Ou seja a resposta está na utilização das capacidades de construção modular e parametrizável de programas que ambas as linguagens *ML* e *OBJ3* possuem, para se poder construir um sistema de dedução de uma forma modular e parametrizável.

Temos assim, pelo menos, três possibilidades:

- Construção modular e parametrizável sob o mesmo sistema lógico e sob o mesmo sistema de dedução. Por exemplo: construir um sistema de sequências deductivas para a lógica de primeira que permita o desenvolvimento de demonstrações de correcção parcial de programas que manipulem listas de inteiros, a partir de sistemas de dedução para

programas, para inteiros e para listas.

- Construção modular e parametrizável sob diferentes sistemas lógicos mas sob o mesmo sistema de dedução. Por exemplo: combinar um sistema de sequências deductivas de primeira ordem para o desenvolvimento de demonstrações de correcção parcial de programas e um sistema de ordem superior para o tipo de dados dos inteiros.

- Construção modular e parametrizável sob diferentes sistemas de dedução (e diferentes sistemas lógicos). Neste caso as árvores de demonstração seriam compostas de ramos de diferentes tipos. Esta seria a generalização daquilo que já se fez com a utilização do sistema de inferência equacional do *OBJ3* nas demonstrações desenvolvidas no sistema *LK* do *2OBJ*.

Nos capítulos que se seguem vai-se tentar responder a estas questões.

Capítulo 3

Sistemas Lógicos

A crescente complexidade dos sistemas de dedução passíveis de serem implementados pelos sistemas computacionais de demonstração automática e semi-automática de teoremas, fez surgir a necessidade da construção modular de sistemas de dedução. Pretende-se então ter acesso aos mecanismos necessários à construção modular de sistemas de dedução, ou seja, pretende-se combinar sistemas de dedução obtendo um sistema de dedução, pretende-se re-utilizar sistemas de dedução previamente definidos na definição de novos sistemas de dedução, e isto através de mecanismos que têm de ir além da simples soma de sistemas de dedução, mecanismo já implementado em muitos sistemas computacionais.

Uma possível solução é dada pelo cálculo de co-limites. Tendo escolhido essa abordagem tornou-se necessário definir uma categoria, a categoria $\mathcal{SD}$, cujos objectos são conjuntos de sistemas de dedução, e cujos morfismo são aplicações entre conjuntos de sistemas de dedução. Demonstrou-se que a categoria $\mathcal{SD}$ é fechada para co-limites, como tal é possível usar as construções próprias do cálculo de co-limites para combinar, e re-utilizar, sistemas de dedução, de forma a obter outros sistemas de dedução.

O capítulo termina com a apresentação de um exemplo de aplicação dos mecanismos atrás referidos à construção modular de um sistema de dedução, mais concretamente, construiu-se um sistema de dedução para a demonstração da correcção parcial de programas que manipulam pilhas de naturais, através da construção de um quadrado co-cartesiano ("push-out").

3.1 Sistemas de dedução

Intuitivamente um sistema de dedução é uma entidade que agrega, para uma dada linguagem lógica, as fórmulas, e as deduções que se podem efectuar sobre essas fórmulas.

Vejamos então como podemos definir formalmente um sistema de dedução, começando por definir alguns conceitos auxiliares.

Definição 3.1 (Grafo) *Um grafo consiste em duas classes: a classe O dos objectos, e a classe S das setas, e de duas funções, ambas da classe das setas para a classe dos objectos, uma das funções é designada por* domínio *e a outra por* co-domínio. *Escreve-se usualmente $f : A \to B$ ou $A \overset{f}{\to} B$ sempre que se verifique que,* domínio$(f) = A$ *e* co-domínio$(f) = B$ *(Lambek & Scott, 1988; MacLane, 1971).*

Se considerarmos que um sistema de dedução se caracteriza pelo seu conjunto de fórmulas, e pelo seu conjunto de deduções, podemos então pensar num sistema de dedução como um grafo com algumas setas específicas.

Definição 3.2 (Sistema de Dedução) *Um sistema de dedução é um grafo no qual para cada objecto A existe uma seta $1_A : A \to A$, e para cada par de setas $f : A \to B$ e $g : B \to C$ existe uma seta $gf : A \to C$ designada por composição de f com g (Lambek & Scott, 1988).*

A visão usual de um sistema de dedução obtem-se facilmente se pensarmos nos objectos do sistema de dedução como fórmulas, nas setas como deduções, e nas operações com setas como sendo as regras de inferência. Por exemplo a regra da composição de setas pode ser escrita do seguinte modo

$$\frac{f : A \to B \qquad g : B \to C}{gf : A \to C}$$

Dado que se pretende estudar os mecanismos de combinação de sistemas de dedução é necessário definir uma entidade que os agregue, e na qual se possam efectuar as operações pretendidas. Vai-se então definir a categoria $\mathcal{SD}$, a categoria dos sistema de dedução, na qual se vão considerar conjuntos de sistemas de dedução como objectos da categoria, e as aplicações entre conjuntos de sistemas de dedução como setas da categoria.

Para poder explicitar a componente da linguagem lógica na qual se baseia um dado sistema de dedução, e para poder relacionar as construções com os sistemas de dedução, com construções nas linguagens lógicas, vai-se definir a categoria $\mathcal{ASS}$, a categoria das assinaturas e das aplicações entre elas.

No que se segue vão-se adoptar as seguintes convenções de escrita; vão-se usar letras caligráficas maiúsculas para designar as categorias, letras maiúsculas para designar os objectos das categorias e, letras minúsculas para designar os morfismos (setas) das categorias. Dado um categoria $\mathcal{A}$ a classe dos objectos de $\mathcal{A}$ é designada por $\mathrm{Obj}(\mathcal{A})$, a classe dos morfismos por $\mathrm{Mor}(\mathcal{A})$, e o conjunto dos morfismos de A para B é designado por $\mathcal{A}(A, B)$. Para designar os functores usam-se letras maiúsculas, eventualmente seguidas de minúsculas. Excepção ao que foi dito é a utilização de um tipo de letra sem serifas para designar a categoria dos conjuntos, Set, a categoria das categorias pequenas, Cat, e a categoria dos co-cones sobre um dado diagrama D numa categoria $\mathcal{C}$, $\mathsf{Co-Cone}(D, \mathcal{C})$ (Borceaux, 1994).

Antes de mais alguns conceitos auxiliares.

Definição 3.3 (Categoria) *Uma categoria é um grafo, no qual para cada objecto A existe uma seta, $1_A : A \to A$ designada por identidade, e para cada par de setas $f : A \to B$ e $g : B \to C$ existe uma seta, $gf : A \to C$ designada por composição, tais que para todo o $h : C \to D$ tem-se:*

$$\begin{array}{ll} f1_A = f = 1_B f & \text{unidade} \\ (hg)f = h(gf) & \text{associatividade} \end{array}$$

(Lambek & Scott, 1988; MacLane, 1971)

Um exemplo de uma categoria usualmente dado, não só pela sua fácil "visualização", como pela sua utilidade é a categoria dos conjuntos Set. A classe dos objectos em Set é a classe de todos os conjuntos, e a classe das setas é a classe de todas as funções entre conjuntos. Outro exemplo normalmente dado é o da categoria Cat, cujos objectos são as categorias pequenas e cujas setas são functores entre categorias pequenas, as categorias pequenas são as categorias para as quais a classe dos objectos é um conjunto (Borceaux, 1994; Herrlich & Strecker, 1973; MacLane, 1971).

Definição 3.4 (Functor) *Um functor $F : \mathcal{A} \to \mathcal{B}$, com domínio a categoria $\mathcal{A}$ e co-domínio a categoria $\mathcal{B}$, é um par de funções: a função nos objectos F que atribui a cada objecto de $\mathcal{A}$ um objecto de $\mathcal{B}$, e uma função nas setas, também designada por F, que atribui a cada seta $f : A \to A'$ uma seta $F(f) : F(A) \to F(A')$ de tal forma que as identidades e a composição são preservadas, isto é:*

$$F(1_A) = 1_{F(A)} \qquad F(gf) = F(g)F(f)$$

sempre que gf estiver definido (Lambek & Scott, 1988; MacLane, 1971).

Definição 3.5 (Categorial Dual) *Seja $\mathcal{C} = (O, S, \mathrm{dom}, \mathrm{cod}, \cdot)$ uma categoria com lei de composição "$\cdot$". A categoria dual é a categoria $\mathcal{C}^{op} = (O, S, \mathrm{cod}, \mathrm{dom}, *)$ com a lei de composição "$*$" definida por $f * g = g \cdot f$ (Herrlich & Strecker, 1973).*

Vejamos então como definir o conjunto de símbolos da linguagem lógica de um sistema de dedução, isto é a sua assinatura.

Definição 3.6 (Assinatura) *Uma assinatura (mono-espécie) Σ é um conjunto Σ e uma função $a : \Sigma \to \mathbb{N}$. Todo o símbolo $\sigma \in \Sigma$ tem uma aridade $a(\sigma)$ que indica o número de argumentos de σ. Símbolos de aridade 0 são designados por constantes (Gallier, 1987).*

Importante para a construção da categoria das assinaturas é a definição de morfismo de assinaturas.

Definição 3.7 (Morfismo de Assinaturas) *Dado duas assinaturas Σ e Σ' um morfismo de assinaturas é uma função entre os conjuntos de símbolos de forma a que a aridade dos símbolos seja preservada (Gallier, 1987).*

Temos então que a categoria cujos objectos são assinaturas e cujos morfismo são os morfismos de assinaturas é a entidade que nos permite descrever, de uma forma genérica, as assinaturas e as relações entre elas. É necessário demonstrar que estamos perante uma construção correcta de uma categoria.

Teorema 3.1 (Categoria de Assinaturas) *A categoria das assinaturas $\mathcal{ASS}$ tem como objectos as assinaturas e como setas os morfismo de assinaturas.*

Demonstração. Para cada $\Sigma \in \mathrm{Obj}(\mathcal{ASS})$ existe a função identidade id_Σ que é um morfismo de assinaturas e que verifica $\mathrm{id}_\Sigma f = f$ e $g\,\mathrm{id}_\Sigma = g$ para $f : \Sigma' \to \Sigma$ e $g : \Sigma \to \Sigma'$, morfismos de assinaturas.

A composição de morfismos $f : \Sigma \to \Sigma'$ e $g' : \Sigma' \to \Sigma''$ é a composição usual de funções, gf é um morfismo de assinaturas dado que, para qualquer $\sigma \in \Sigma$ tem-se, $a(\sigma) = a'(f(\sigma)) = a''(g(f(\sigma))) = a''(gf(\sigma))$. A composição é associativa.

c.q.d.

3.2 Σ-Sistemas de dedução

Trata-se agora de definir um sistema de dedução cujas fórmulas bem formadas sejam construídas, indutivamente, com símbolos de uma dada assinatura (Gallier, 1987). Vão-se designar por Σ-fórmulas as fórmulas construídas, indutivamente, com símbolos de Σ, e por Σ-deduções as deduções entre Σ-fórmulas.

Definição 3.8 (Σ-Sistemas de Dedução) *Dado uma assinatura Σ, o conjunto dos Σ-sistemas de dedução C_{SD_Σ}, tem como elementos todos os Σ-sistemas de dedução, isto é, todos os sistemas de dedução cujos objectos são Σ-fórmulas e cujas setas são Σ-deduções.*

Por conveniência de linguagem, e sempre que tal não provocar ambiguidades, vai-se designar por Σ-sistema de dedução tanto o conjunto C_{SD_Σ} como cada um dos seus elementos.

Definição 3.9 (Morfismos de Σ-Sistemas de Dedução) *Dado um morfismo de assinaturas $\sigma : \Sigma \to \Sigma'$ um morfismo de Σ-sistemas de dedução é uma função $f : C_{SD_\Sigma} \to C_{SD_{\Sigma'}}$ tal que:*

$$\Sigma \xrightarrow{\sigma} \Sigma'$$

$$C_{SD_\Sigma} \xrightarrow{f} C_{SD_{\Sigma'}}$$

$$SD_\Sigma \xmapsto{\tilde{\sigma}} SD_{\Sigma'}$$

$$\begin{array}{ccc} s_1 & & \tilde{\sigma}(s_1) \\ p\downarrow & \mapsto & \downarrow\tilde{\sigma}(p) \\ s_2 & & \tilde{\sigma}(s_2) \end{array}$$

com $\tilde{\sigma}$ a extensão única de σ às Σ-fórmulas (Gallier, 1987; Goguen & Burstall, 1992).

A categoria cujos objectos são os conjuntos de Σ-sistemas de dedução e cujos morfismos são os morfismos de Σ-sistemas de dedução é a entidade que nos permite descrever, de uma forma genérica, os Σ-sistemas de dedução e as relações entre eles. É necessário demonstrar que estamos perante uma construção correcta de uma categoria.

Teorema 3.2 (Categoria dos Σ-Sistemas de Dedução) *A categoria dos Σ-sistemas de dedução $\mathcal{SD}$ tem como objectos os conjuntos de Σ-sistemas de dedução C_{SD_Σ}, e como morfismos os morfismos de Σ-sistemas de dedução.*

Demonstração. Para cada objecto de $\mathcal{SD}$ existe uma função id_{SD_Σ} que corresponde à função id_Σ em $\mathcal{ASS}$, id_{SD_Σ} é um morfismo em $\mathcal{SD}$ e verifica $\mathrm{id}_{SD_\Sigma} f = f$ e $g\,\mathrm{id}_{SD_\Sigma} = g$ para f e g morfismos em $\mathcal{SD}$.

A composição de morfismos é dada pela composição das funções $\tilde{\sigma}$ extensões das funções σ às Σ-fórmulas.

$$\Sigma \xrightarrow{\sigma} \Sigma' \xrightarrow{\sigma'} \Sigma''$$

$$C_{SD_\Sigma} \xrightarrow{f} C_{SD_{\Sigma'}} \xrightarrow{f'} C_{SD_{\Sigma''}}$$

$$SD_\Sigma \xmapsto{\tilde{\sigma}} SD_{\Sigma'} \xmapsto{\widetilde{\sigma'}} SD_{\Sigma''}$$

$$\begin{array}{ccccc} s_1 & & \tilde{\sigma}(s_1) & & \widetilde{\sigma'}(\tilde{\sigma}(s_1)) \\ p\downarrow & \mapsto & \downarrow\tilde{\sigma}(p) & \mapsto & \downarrow\widetilde{\sigma'}(\tilde{\sigma}(p)) \\ s_2 & & \tilde{\sigma}(s_2) & & \widetilde{\sigma'}(\tilde{\sigma}(s_2)) \end{array}$$

A composição de morfismos é associativa.

c.q.d.

3.3 Sistemas lógicos

Pretende-se agora descrever as transformações que ocorrem num sistema de dedução sempre que há transformações na linguagem lógica que serve de suporte. Vamos para tal definir um sistema lógico como sendo uma entidade que agrupa a categoria das assinaturas, a categoria dos sistemas de dedução, e um functor que para uma dada assinatura nos dá o correspondente sistema de dedução.

Definição 3.10 (Sistemas Lógicos) *Um sistema lógico* $\langle \mathcal{ASS}, \mathcal{SD}, \mathrm{Sd} \rangle$ *é um terno em que:*

- $\mathcal{ASS}$ *é a categoria das assinaturas;*
- $\mathcal{SD}$ *é a categoria dos* Σ*-sistemas de dedução;*
- $\mathrm{Sd} : \mathcal{ASS} \to \mathcal{SD}$ *é um functor que para uma dada assinatura* Σ *dá origem ao conjunto dos* Σ*-sistemas de dedução, e para um dado morfismo de assinaturas dá origem a um morfismo de* Σ*-sistemas de dedução.*

As categorias $\mathcal{ASS}$ e $\mathcal{SD}$ já foram anteriormente caracterizadas. Falta caracterizar Sd e verificar que é um functor.

Teorema 3.3 (Functor Sd) *A aplicação* $\mathrm{Sd} : \mathcal{ASS} \to \mathcal{SD}$ *que faz corresponder a cada assinatura* $\Sigma \in \mathrm{Obj}(\mathcal{ASS})$ *um* Σ*-sistema de dedução* $C_{SD_\Sigma} \in \mathrm{Obj}(\mathcal{SD})$*, e que a cada morfismo de assinaturas* $\sigma : \Sigma \to \Sigma' \in \mathrm{Mor}(\mathcal{ASS})$ *faz corresponder um morfismo de* Σ*-sistemas de dedução* $f : C_{SD_\Sigma} \to C_{SD_{\Sigma'}}$*, define um functor da categoria das assinaturas para a categoria dos* Σ*-sistemas de dedução.*

Demonstração. Para cada $\Sigma \in \mathrm{Obj}(\mathcal{ASS})$, $\mathrm{Sd}(\Sigma) = C_{SD_\Sigma} \in \mathrm{Obj}(\mathcal{SD})$ pela definição 3.8. Para cada $\sigma : \Sigma \to \Sigma' \in \mathrm{Mor}(\mathcal{ASS})$, $\mathrm{Sd}(\sigma) = f : C_{SD_\Sigma} \to C_{SD'_\Sigma} \in \mathrm{Mor}(\mathcal{SD})$ pela definição 3.9.

Verifica-se que:

- $\mathrm{Sd}(\mathrm{id}_\Sigma) = f : C_{SD_\Sigma} \to C_{SD_\Sigma} = \mathrm{id}_{C_{SD_\Sigma}} = \mathrm{id}_{\mathrm{Sd}(\Sigma)}$.
- $\mathrm{Sd}(\sigma'\sigma) = f'f = \mathrm{Sd}(\sigma')\mathrm{Sd}(\sigma)$.

c.q.d.

A definição de sistema lógico está perto da definição de instituição (Goguen & Burstall, 1985; Goguen & Burstall, 1992), está no entanto mais próxima da definição de Π-instituição (Fiadeiro & Sernadas, 1988).

Em relação ao conceito de instituição a principal diferença consiste na diferença de aproximação. Enquanto a definição de sistema lógico visa uma aproximação dentro da teoria da dedução a definição de instituição está ligada à teoria dos modelos.

A definição de instituição apresentada por Goguen e Burstall (1985) define uma instituição como sendo um quádruplo $\langle \mathcal{ASS}, \mathrm{Mod}, \mathrm{Sen}, \models_\Sigma \rangle$, em que $\mathrm{Mod} : \mathcal{ASS} \to \mathsf{Cat}^{\mathrm{op}}$ é um functor dando origem aos Σ-modelos e aos morfismos entre Σ-modelos, $\mathrm{Sen} : \mathcal{ASS} \to \mathsf{Cat}$ é um functor dando origem às Σ-fórmulas e às Σ-demonstrações, e finalmente $\models_\Sigma$ é uma relação de satisfação entre Σ-modelos e Σ-fórmulas.

Analisando as duas definições verifica-se que a componente referente aos modelos não está presente na definição de sistema lógico, a razão para essa omissão prende-se com a já referida aproximação à teoria da dedução. O functor $\mathrm{Sen} : \mathcal{ASS} \to \mathsf{Cat}$ é, de certa forma, substituindo pelo functor $\mathrm{Sd} : \mathcal{ASS} \to \mathcal{SD}$, neste caso o objectivo é o mesmo sendo que a definição de sistema de dedução é menos exigente. No entanto é de notar que para qualquer sistema de dedução pode-se obter uma categoria impondo uma relação de equivalência apropriada entre demonstrações (Lambek & Scott, 1988); a relação de satisfação não está presente sendo substituída pela noção de consequência presente na definição de sistema de dedução, embora não de forma explícita, este ponto ficará mais claro quando analisarmos as diferenças entre a presente definição e a de Π-instituição.

A definição de instituição apresentada por Goguen e Burstall (1992) difere da anteriormente apresentada pelos mesmos autores pelo facto de o functor Sen ter agora como co-domínio a categoria Set. Esta nova definição, aliás já apresentada no artigo anterior (Goguen & Burstall, 1985) como um dos casos particulares da definição geral, não faz mais do que afastar a noção de instituição da teoria da dedução, e como tal da noção de sistema lógico.

Em contraponto com a noção de instituição está a noção de Π-instituição apresentada por Fiadeiro e Sernadas (1988). Neste caso está-se perante uma aproximação dentro da teoria da dedução. Uma Π-instituição é um triplo $\langle \mathcal{ASS}, \Phi, \{Cn_\Sigma\}_{\Sigma \in \mathrm{Obj}(\mathcal{ASS})} \rangle$, com $\Phi : \mathcal{ASS} \to \mathsf{Set}$ um functor que dá origem às Σ-fórmulas, e com Cn_Σ um operador de consequência definido em $\mathscr{P}(\Phi(\Sigma))$ (Fiadeiro & Sernadas, 1988).

O operador de consequência Cn_Σ verifica para cada $A, B \subseteq \Phi(\Sigma)$ e $\sigma : \Sigma \to \Sigma'$ as seguintes propriedades:

- $A \subseteq Cn_\Sigma(A)$ (extensividade);
- $Cn_\Sigma(Cn_\Sigma(A)) = Cn_\Sigma(A)$ (idempotência);
- $Cn_\Sigma(A) = \bigcup_{\substack{B \subseteq A \\ B \text{ finito}}} Cn_\Sigma(B)$ (Compacidade);
- $\Phi(\sigma)(Cn_\Sigma(A)) \subseteq Cn_{\Sigma'}(\Phi(\sigma)(A))$ (estruturalidade)

Analisemos as diferenças entre os conceitos de Π-instituição e sistema lógico. Na definição de Π-instituição o functor Φ tem como co-domínio a categoria Set, dando origem às Σ-fórmulas. Em contraponto o functor Sd tem como co-domínio a categoria $\mathcal{SD}$ dando origem às Σ-fórmulas e às

Σ-deduções. A riqueza acrescida de $\mathcal{SD}$ tem como vantagem que nesta categoria é possível definir internamente uma qualquer relação de consequência impondo para tal restrições nos morfismos, veja-se por exemplo o trabalho de Lambek (1958; 1969; 1971), de Lambek e Scott (1988) e o de Szabo (1976; 1978). No entanto é importante verificar que o modo como se definiu um sistema de dedução tem como consequência que functor Sd verifica a propriedade designada por estruturalidade. Vejamos que assim é:

Teorema 3.4 *Para um qualquer operador de relação de consequência Cn_Σ definido em $\mathscr{P}(\mathrm{Obj}(SD_\Sigma))$ o functor Sd verifica a condição de estruturalidade, com $SD_\Sigma \in \mathrm{Obj}(\mathcal{SD})$.*

Demonstração. Por $\mathrm{Obj}(SD_\Sigma)$ entende-se o conjunto das Σ-fórmulas de $\mathcal{SD}_\Sigma$, e por $\mathrm{Mor}(SD_\Sigma)$ o conjunto dos morfismos (setas) de SD_Σ. A definição de um dado operador de relação de consequência num sistema de dedução é dada pela definição dos morfismos entre objectos (Avron, 1991; Lambek & Scott, 1988).

Temos então:

$$Cn_\Sigma(A) \quad = \quad \{s' \in \mathrm{Obj}(SD_\Sigma) \mid p : s \to s' \quad \text{com } s \in A, p \in \mathrm{Mor}(SD_\Sigma)\}$$

por aplicação do functor Sd a $\sigma : \Sigma \to \Sigma'$

$$\begin{aligned}\mathrm{Sd}(\sigma)(Cn_\Sigma(A)) \quad = \quad & \{\mathrm{Sd}(\sigma)(s') \in \mathrm{Obj}(SD_{\Sigma'}) \mid \mathrm{Sd}(\sigma)(p) : \mathrm{Sd}(\sigma)(s) \to \\ & \mathrm{Sd}(\sigma)(s') \quad \text{com } \mathrm{Sd}(\sigma)(s) \in \mathrm{Sd}(\sigma)(A),\ \mathrm{Sd}(\sigma)(p) \in \\ & \mathrm{Mor}(SD_{\Sigma'})\}\end{aligned}$$

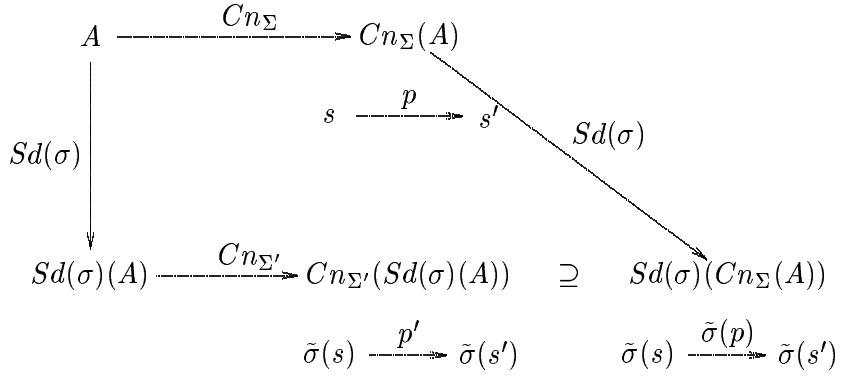
por outro lado, tem-se:

$$\begin{aligned}Cn_{\Sigma'}(\mathrm{Sd}(\sigma)(A)) \quad = \quad & \{s' \in \mathrm{Obj}(SD_{\Sigma'}) \mid p' : Sd(\sigma)(s) \to s' \text{ com} \\ & Sd(\sigma)(s) \in \mathrm{Sd}(\sigma)(A), p' \in \mathrm{Mor}(SD_{\Sigma'})\}\end{aligned}$$

Pela definição do functor Sd temos que dado um $p : s \to s' \in \mathrm{Mor}(SD_\Sigma)$ então $\tilde{\sigma}(p) : \tilde{\sigma}(s) \to \tilde{\sigma}(s') \in \mathrm{Mor}(SD_{\Sigma'})$ temos então que:

$$\mathrm{Sd}(\sigma)(Cn_\Sigma(A)) \quad \subseteq \quad Cn_{\Sigma'}(\mathrm{Sd}(\sigma)(A))$$

Graficamente temos



c.q.d.

Com a definição de sistema lógico obtem-se uma entidade que relaciona as assinaturas, definidoras das linguagens lógicas, com os sistemas de dedução. Podemos desta forma relacionar as transformações nos sistemas de dedução com as transformações nas assinaturas, no entanto para poder construir sistemas de dedução de uma forma modular é ainda necessário demonstrar que $\mathcal{SD}$ é fechada para co-limites. Para tal vai-se demonstrar que a categoria $\mathcal{ASS}$ é co-completa e que o functor $Ass : \mathcal{SD} \to \mathcal{ASS}$ reflecte co-limites.

3.4 A categoria $\mathcal{SD}$ é co-completa

Para poder construir modularmente sistemas de dedução em $\mathcal{SD}$ através do cálculo de co-limites, é necessário demonstrar que $\mathcal{SD}$ é fechada para co-limites. Para demonstrar tal facto começa-se por verificar que a categoria $\mathcal{ASS}$ é co-completa, de seguida define-se o functor Ass : $\mathcal{SD} \to \mathcal{ASS}$ e demonstra-se que ele reflecte co-limites, da conjugação destes dois factos conclui-se que $\mathcal{SD}$ é co-completa, como pretendíamos.

3.4.1 A categoria $\mathcal{ASS}$ é co-completa

Pretende-se estabelecer que a categoria $\mathcal{ASS}$ é co-completa. Para tal vai-se começar por constatar que uma assinatura pode ser vista como uma família indexada por $I \subseteq \mathbb{N}$ de símbolos de operadores, ou seja $\mathcal{ASS}$ é um caso particular de $\mathcal{FC}_S$, a categoria das famílias de conjuntos disjuntos indexadas por S, para o caso em que $S = I$. De seguida demonstra-se que a categoria $\mathcal{FC}_S$ e a categoria fibrada (Set/S) são isomorfas dado ser possível definir um isomorfismo entre elas. Dado que a categoria (Set/S) é co-completa conclui-se que $\mathcal{ASS}$ é co-completa como se pretendia.

Vejamos então como definir uma assinatura como uma família de conjuntos disjuntos indexada por $I \subseteq \mathbb{N}$.

Proposição 3.1 (Assinatura) *Uma assinatura (mono-espécie)* Σ*, é uma família de conjuntos disjuntos de símbolos de operadores, indexada por* $I \subseteq \mathbb{N}$*,* $\Sigma = \langle \sigma_i | i \in I \rangle$*.*

Demonstração. É necessário demonstrar que esta definição é equivalente à anteriormente dada (definição 3.6).

Uma família I-indexada de conjuntos disjuntos é uma função (Herrlich & Strecker, 1973),

$$\begin{array}{rcl} f : I & \longrightarrow & \bigcup_{i \in I} \sigma_i \\ i & \longmapsto & \sigma_i \end{array}$$

temos então que:

$$\Sigma = \bigcup_{i \in I} \{\sigma | \sigma \in \sigma_i\}$$

$$\begin{array}{rcl} a : \Sigma & \longrightarrow & I \\ \sigma & \longmapsto & i \quad \text{para todo o } \sigma \in \sigma_i \end{array}$$

a função a está bem definida dado que os conjuntos σ_i, para $i \in I$, são disjuntos dois a dois.

Por outro lado, se $\Sigma = \langle \Sigma, a \rangle$ é uma assinatura (mono-espécie) de símbolos de operadores com:

$$\begin{array}{rcl} a : \Sigma & \longrightarrow & \mathbb{N} \\ \sigma & \longmapsto & a(\sigma) = i \end{array}$$

temos então:

$$\begin{array}{rcl} f : a(\Sigma) & \longrightarrow & \bigcup_{i \in a(\Sigma)} a^{-1}(\{i\}) \\ i & \longmapsto & a^{-1}(\{i\}) = \sigma_i = \{\sigma | i \in a(\Sigma), i = a(\sigma)\} \end{array}$$

dado que todo o operador em Σ tem uma aridade bem definida, a função f está bem definida e os conjuntos $a^{-1}(\{i\})$ são disjuntos dois a dois.

c.q.d.

Temos então que a categoria $\mathcal{ASS}$ é um caso particular de $\mathcal{FC}_S$. É necessário estabelecer que $\mathcal{FC}_S$ é de facto uma categoria estabelecendo quais são os seus objectos e os seus morfismos.

Lema 3.1 *$\mathcal{FC}_S$ é uma categoria.*

Demonstração. Os objectos de $\mathcal{FC}_S$ são funções x com domínio em S, tais que para cada $s \in S, x(s) = X_s$, com $X_s \cap X_{s'} = \emptyset$, sempre que $s \neq s'$. Vão-se denotar tais objectos por $(X_s)_{s \in S}$. Os morfismo em $\mathcal{FC}_S$ são famílias de funções $(f_s : X_s \to Y_s)_{s \in S}$ com $f_s(x(s)) = y(s)$, para todo o $s \in S$ (Herrlich & Strecker, 1973).

A identidade é dada por $(id_s : X_s \to X_s)_{s \in S}$. A composição de morfismo é a composição usual de funções, componente a componente, $(f_s : X_s \to Y_s)_{s \in S}(g_s : Y_s \to Z_s)_{s \in S} = (g_s f_s : X_s \to Z_s)_{s \in S}$, a composição de morfismos é associativa.

c.q.d.

Antes de definir um functor entre $\mathcal{FC}_S$ e (Set/S) é necessário caracterizar a categoria fibrada (Set/S), estabelecendo quais são os seus objectos e os seus morfismos. A categoria fibrada (Set/S) é um caso particular de uma categoria dois-functores ("Comma Category") (MacLane, 1971). Temos então de começar por caracterizar o que se entende por uma categoria dois-functores.

Definição 3.11 (Categoria dois-functores) *Dado um functor $F : \mathcal{A} \to \mathcal{C}$ e um functor $G : \mathcal{B} \to \mathcal{C}$, define-se a categoria dois-functores (F/G) como sendo a categoria cujos objectos são triplos $\langle A, B, f \rangle$, com $A \in \mathrm{Obj}(\mathcal{A})$, $B \in \mathrm{Obj}(\mathcal{B})$ e $f : F(A) \to G(B) \in \mathrm{Mor}(\mathcal{C})$. Os morfismos de (F/G) são pares de morfismos, $\langle a, b \rangle : \langle A, B, f \rangle \to \langle A', B', f' \rangle$, com $a : A \to A'$ e $b : B \to B'$ tais que $f'F(a) = G(b)f$. Ou seja, o seguinte diagrama é comutativo:*

$$\begin{array}{ccccc}
A & \xrightarrow{\quad F \quad} & C & \xleftarrow{\quad G \quad} & B \\
A & \longmapsto & F(A) \xrightarrow{f} G(B) & \longmapsto & B \\
a \downarrow & \longmapsto & F(a) \downarrow & & G(b) \longmapsto \downarrow b \\
A' & \longmapsto & F(A') \xrightarrow[f']{} G(B') & \longmapsto & B'
\end{array}$$

A composição de morfismos em (F/G), $\langle a', b' \rangle \langle a, b \rangle$ é dada por $\langle a'a, b'b \rangle$, sempre que tal esteja definido (Borceaux, 1994; MacLane, 1971).

Nos casos em que o functor F é o functor identidade na categoria $\mathcal{A}$, e em que o functor G define um dado objecto O nessa mesma categoria, é usual denotar a categoria dois-functores por $(\mathcal{A}/O)$, e designá-la por categoria fibrada.

Para (Set/S) temos que $\mathcal{A} = \mathcal{C} = \mathsf{Set}$, $\mathcal{B} = \mathbf{1}$, $F = \mathrm{Id}_{\mathsf{Set}}$ e $G : \mathbf{1} \to \mathsf{Set}$ com $G(*) = S$.

Os objectos de (Set/S) são triplos $\langle A, *, c \rangle$, com $c : A \to S$. Os morfismos de (Set/S) são funções $f : \langle A, *, c \rangle \to \langle B, *, d \rangle$, com $f : A \to B$ tal que $df = c$. Graficamente temos:

$$\mathsf{Set} \xrightarrow{\mathrm{id}_{\mathsf{Set}}} \mathsf{Set} \xleftarrow{G} \mathbf{1}$$

$$\begin{array}{ccccc} A & \longmapsto & A & & * \\ & & & \searrow^{c} & \\ f \downarrow & \mathrm{id}_{\mathsf{Set}}(f) = f & \downarrow & & S \\ & & & \nearrow_{d} & \\ B & \longmapsto & B & & \end{array}$$

Pretende-se agora demonstrar que $\mathcal{FC}_S$ e (Set/S) são isomorfas, $\mathcal{FC}_S \cong (\mathsf{Set}/S)$, isto é, existe um functor $F : (\mathsf{Set}/S) \to \mathcal{FC}_S$ que é um isomorfismo de (Set/S) para $\mathcal{FC}_S$.

Teorema 3.5 *A categoria das famílias de conjuntos disjuntos S-indexados $\mathcal{FC}_S$ é isomorfa à categoria fibrada* (Set/S).

Demonstração. Vai-se definir o functor $F : (\mathsf{Set}/S) \to \mathcal{FC}_S$, do seguinte modo:

$$\begin{array}{lll} F : (\mathsf{Set}/S) \longrightarrow \mathcal{FC}_S & & \\ \langle A, *, c\rangle \longmapsto (c^{-1}(s))_{s\in S} & & \\ \langle A, *, c\rangle & (c^{-1}(s))_{s\in S} & \\ f \downarrow & \longmapsto \downarrow (f_s)_{s\in S} & \text{com } f_s = f|_{c^{-1}(s)} \\ \langle B, *, d\rangle & (d^{-1}(s))_{s\in S} & \end{array}$$

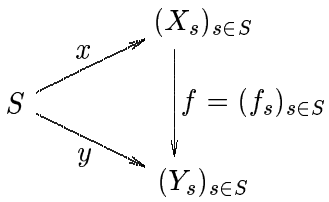
interpretando deste modo a função $c : A \to S$ como uma função que, para um dado $a \in A$, lhe atribui o seu índice $c(a) = s, s \in S$ (Goguen & Burstall, 1984).

Dado que f é, por hipótese, um morfismo em (Set/S) temos então que c e d são funções bem definidas e como tal $(c^{-1}(s))_{s\in S}$ e $(d^{-1}(s))_{s\in S}$ definem famílias de conjuntos disjuntos indexadas por S, com elementos de A e B respectivamente. A família de funções $(f_s)_{s\in S}$, está bem definida dado que se trata de definir a restrição de f às imagens inversas de c, para todo o $s \in S$. Dado que $df = c$, então $f(c^{-1}(s)) = d^{-1}(s)$.

Para demonstrar que F é um isomorfismo é necessário e suficiente demonstrar que a função $F : \mathrm{Mor}((\mathsf{Set}/S)) \to \mathrm{Mor}(\mathcal{FC}_S)$ é sobrejectiva e injectiva (Herrlich & Strecker, 1973).

Sobrejectividade. Vejamos que F é sobrejectivo quando aplicada aos morfismos.

Dado um morfismo em $\mathcal{FC}_S$, $(f_s : X_s \to Y_s)_{s\in S}$



podemos definir as funções $c = (c_s)_{s\in S}$, com $c_s(a) = s$, para todo o $a \in X_s$, e $d = (d_s)_{s\in S}$, com $d_s(b) = s$, para todo o $b \in Y_s$. Dado que os conjuntos X_s, assim como os conjuntos Y_s, são disjuntos dois a dois, então as funções c e d estão bem definidas.

Temos então que $\langle \cup_{s\in S} \{a \mid a \in X_s\}, *, c\rangle$ e $\langle \cup_{s\in S} \{b \mid b \in Y_s\}, *, d\rangle$, são dois objectos de (Set/S). Podemos agora definir uma função

$$k : \cup_{s\in S} \{a \mid a \in X_s\} \to \cup_{s\in S} \{b \mid b \in Y_s\}$$

tal que, para todo o $a \in X_s$, se tem que $k(a) = f_s(a) = b$, com $b \in Y_s$, para todo o $s \in S$.

Por definição $d(k(a)) = d(b) = s = c(a)$, ou seja k é um morfismo em (Set/S), além disso temos que $F(k) = f$, como queríamos demonstrar.

Injectividade. Para demonstrar a injectividade de F quando aplicado aos morfismos, vai-se demonstrar que F é injectivo em relação aos objectos, e posteriormente vai-se usar esse facto para simplificar a demonstração de que F é injectivo nos morfismos.

Suponhamos que se tem $\langle A, *, c\rangle \neq \langle B, *, d\rangle$. Estes dois objectos em (Set/S) podem ser distintos porque $A \neq B$ ou porque $A = B$ mas $c \neq d$. Se A$\neq$ B então $F(\langle A, *, c\rangle) = (c^{-1}(s))_{s\in S} \neq (d^{-1}(s))_{s\in S} = F(\langle B, *, d\rangle)$ trivialmente.

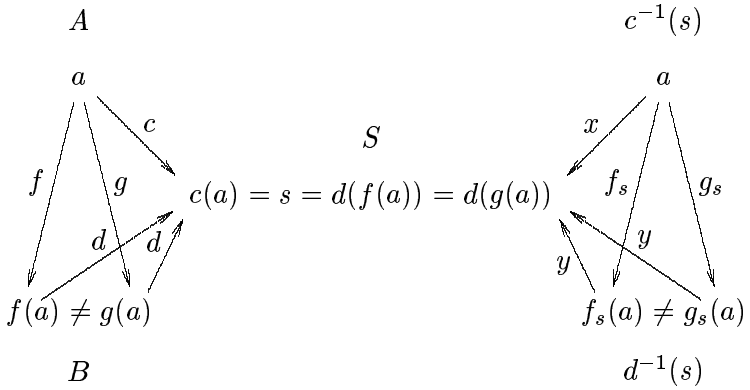
No caso em que $A = B$ mas $c \neq d$ temos então que, existe um $a \in A$ tal que $s = c(a) \neq d(a)$, então tem-se que $a \in c^{-1}(s)$ e $a \notin d^{-1}(s)$ ou seja $(c^{-1}(s))_{s\in S} \neq (d^{-1}(s))_{s\in S}$ como queríamos demonstrar.

Temos então que F é injectivo nos objectos. Sendo assim para demonstrar que F é injectivo nos morfismos basta-nos demonstrar que, para quaisquer duas funções $f, g : \langle A, *, c\rangle \to \langle B, *, d\rangle$, com $f \neq g$, se tem que $F(f) \neq F(g)$.

Os outros casos são triviais se atendermos a que F é injectivo nos objectos.

Consideremos então duas funções f e g tais que $f, g \in \mathrm{Mor}((\mathsf{Set}/S))$ e $f \neq g$. Então existe um $a \in A$ tal que $f(a) \neq g(a)$, como $df = c = dg$ temos que $c(a) = s = d(f(a)) = d(g(a))$, sendo assim o conjunto $d^{-1}(s)$ vai conter $f(a)$ e $g(a)$, com $a \in c^{-1}(s)$. Por definição do functor F as imagens de f e g pelo functor são tais que $f_s = f|_{c^{-1}(s)}$ e $g_s = g|_{c^{-1}(s)}$, sendo assim temos que $f_s(a) \neq g_s(a)$, ou seja $F(f) \neq F(g)$ como queríamos demonstrar.

Graficamente temos:



Demonstrámos que F é bijectiva quando aplicada aos morfismos, temos então que F é um isomorfismo.

c.q.d.

Para demonstrar que $\mathcal{ASS}$ é co-completa, como pretendemos, é então suficiente utilizar o resultado anterior assim como o conhecimento de que a categoria (Set/S) é co-completa.

Teorema 3.6 ($\mathcal{ASS}$ é co-completa) *A categoria $\mathcal{ASS}$ é co-completa.*

Demonstração. A categoria Set é co-completa (Herrlich & Strecker, 1973; MacLane, 1971), então a categoria (Set/S) é co-completa (Borceaux, 1994).

Como, pelo teorema 3.5, a categoria $\mathcal{ASS}$ é isomorfa à categoria (Set/S) então $\mathcal{ASS}$ é co-completa.

c.q.d.

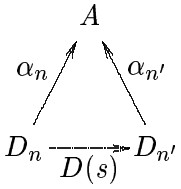
Tendo demonstrado que a categoria $\mathcal{ASS}$ é co-completa torna-se agora necessário demonstrar o resultado respeitante ao functor Ass.

3.4.2 O functor Ass reflecte co-limites

Pretende-se demonstrar que o functor $\mathrm{Ass} : \mathcal{SD} \to \mathcal{ASS}$ reflecte co-limites. Antes de enunciar e demonstrar o teorema respeitante a esse resultado é necessário introduzir alguns conceitos auxiliares, respeitantes à noção de co-limite, e de functor que reflecte co-limites. Vamos começar por definir a categoria $\mathsf{Co-Cone}(D, \mathcal{C})$ de todos os co-cones sobre um dado diagrama D numa categoria $\mathcal{C}$.

Definição 3.12 (Diagrama) *Um diagrama D numa categoria $\mathcal{C}$, consiste num grafo G e de um par de funções de etiquetagem. A cada nó n de G vai corresponder um objecto D_n de $\mathcal{C}$, e a cada seta $s : n \to n'$ vai corresponder um morfismo $D(s)$ de $\mathcal{C}$, de tal forma que $D(s) : D_n \to D_{n'}$ (Goguen & Burstall, 1992; Rydheard & Burstall, 1988).*

Definição 3.13 (Co-cone) *Um co-cone α numa categoria $\mathcal{C}$ para um diagrama D, consiste num objecto A de $\mathcal{C}$, juntamente com um morfismo $\alpha_n : D_n \to A$ para cada nó n de G, tal que para cada seta $s : n \to n'$ em G o diagrama*

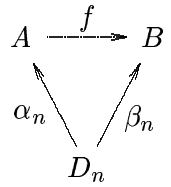


é comutativo (Goguen & Burstall, 1992; Herrlich & Strecker, 1973; Rydheard & Burstall, 1988).

Dado um co-cone α, designa-se D por *base*, A por *ápice*, e G por *forma*. Vai-se denotar o co-cone por $\{\alpha_i : D_i \to A\}$.

É de notar que Herrlich (1973) designa esta construção por "natural sink" e que Goguen e Burstall (1992) a designam por *cone*. Neste último caso julgo que a designação não é, de certo modo, a mais correcta na medida em que a construção que se acabou de definir está ligada à construção de co-limites. Rydeheard e Burstall (1988) usam a designação de *co-cone* para a construção que se acabou de descrever, reservando a designação de *cone* para a construção dual.

Definição 3.14 (Morfismos de co-cones) *Sejam α e β co-cones com base D e ápices A e B respectivamente, então um morfismo de co-cones é um morfismo $f : A \to B$ em $\mathcal{C}$ tal que para cada nó n em G o seguinte diagrama*



é comutativo (Rydheard & Burstall, 1988).

Podemos então definir a categoria de todos os co-cones sobre D em $\mathcal{C}$ e dos morfismos de co-cones, vai-se designar essa categoria por $\mathsf{Co-Cone}(D, \mathcal{C})$ (Herrlich & Strecker, 1973).

Definição 3.15 (Objecto Inicial) *um objecto A de uma categoria $\mathcal{C}$ diz-se um* objecto inicial *se para todo o objecto $C \in \mathrm{Obj}(\mathcal{C})$ existe exactamente um único morfismo de A para C.*

Definição 3.16 (Co-limite) *Um co-limite de um diagrama D numa categoria $\mathcal{C}$ é um objecto inicial em $\mathsf{Co-Cone}(D, \mathcal{C})$ (Goguen & Burstall, 1992; Herrlich & Strecker, 1973).*

Definição 3.17 (Functor reflecte co-limites) *Dado um functor $F : \mathcal{C} \to \mathcal{C}'$, diz-se que ele reflecte (cria) co-limites se, sempre que D é um diagrama em $\mathcal{C}$ tal que o diagrama $F(D)$ em $\mathcal{C}'$ tem um co-cone co-limite $\{\alpha'_i : D'_i \to A'\}$ em $\mathcal{C}'$, então existe um único co-cone co-limite $\{\alpha_i : D_i \to A\}$ em $\mathcal{C}$ tal que $\alpha' = \alpha F$, isto é, $\alpha'_n = F(\alpha_n)$ para todos os nós n na base de D (Goguen & Burstall, 1992; Herrlich & Strecker, 1973).*

Vejamos agora como caracterizar a transformação dos objectos e morfismos de $\mathcal{SD}$ em objectos e morfismos de $\mathcal{ASS}$, demonstrando de seguida que tal transformação define um functor entre as duas categorias.

Teorema 3.7 (Functor Ass**)** *A aplicação* Ass $: \mathcal{SD} \to \mathcal{ASS}$ *que faz corresponder a cada conjunto de Σ-sistemas de dedução a respectiva assinatura Σ, e que transforma morfismos entre conjuntos de Σ-sistemas de dedução nos correspondentes morfismos de assinaturas,*

$$\begin{array}{ccc} \text{Ass} : & \mathcal{SD} & \longrightarrow \mathcal{ASS} \\ & C_{SD_\Sigma} & \longmapsto \Sigma \end{array}$$

$$\begin{array}{ccc} C_{SD_\Sigma} & & \Sigma \\ \downarrow f & \longmapsto & \downarrow \sigma \\ C_{SD_{\Sigma'}} & & \Sigma' \end{array}$$

é um functor.

Demonstração. Atendendo à forma como foram definidos os objectos e os morfismos de $\mathcal{SD}$ (definições 3.8 e 3.9) temos:

$$\text{Ass}(\text{id}_{C_{SD_\Sigma}}) = id_\Sigma = \text{id}_{\mathcal{ASS}(C_{SD_\Sigma})}$$

e

$$\text{Ass}(f'f) = \sigma'\sigma = \text{Ass}(f')\text{Ass}(f)$$

c.q.d.

Teorema 3.8 *O functor* Ass $: \mathcal{SD} \to \mathcal{ASS}$ *reflecte co-limites.*

Demonstração.

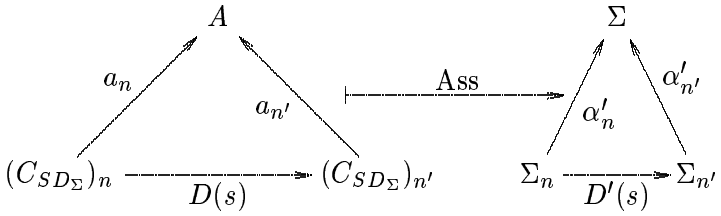
Suponhamos que $D : G \to \mathcal{SD}$ é um diagrama em $\mathcal{SD}$, com $D_n = (C_{SD_\Sigma})_n$ para todo o objecto n de G, e $D(s) : (C_{SD_\Sigma})_n \to (C_{SD_\Sigma})_{n'}$ para toda a seta $s : n \to n'$ de G.

Seja $D' = \text{Ass}(D)$ o diagrama correspondente em $\mathcal{ASS}$ com $D'_n = \Sigma_n$.

Dado que $\mathcal{ASS}$ é co-completa podemos considerar $\{\alpha'_i : D'_i \to \Sigma\}$ um co-cone co-limite para D' em $\mathcal{ASS}$.

Trata-se então de demonstrar que existe um co-cone co-limite α para D em $\mathcal{SD}$ tal que $\mathrm{Ass}(\alpha) = \alpha'$.

Temos então o seguinte:



Vamos definir $A = C_{SD_\Sigma}$ da seguinte forma:

$$C_{SD_\Sigma} = \left(\bigcup_{n \in \mathrm{Obj}(G)} a_n((C_{SD_\Sigma})_n) \right)$$

com $a_n = \alpha_n$ definido do seguinte modo:

$$\alpha_n : (C_{SD_\Sigma})_n \dashrightarrow C_{SD_\Sigma}$$

$$(SD_\Sigma)_n \dashmapsto SD_\Sigma$$

$$s_n \overset{\widetilde{\alpha'_n}}{\dashmapsto} s$$

$$\begin{array}{ccc} s_{1n} & & \widetilde{\alpha'_n}(s_{1n}) \\ \downarrow p_n & \dashmapsto & \downarrow \widetilde{\alpha'_n}(p_n) \\ s_{2n} & & \widetilde{\alpha'_n}(s_{2n}) \end{array}$$

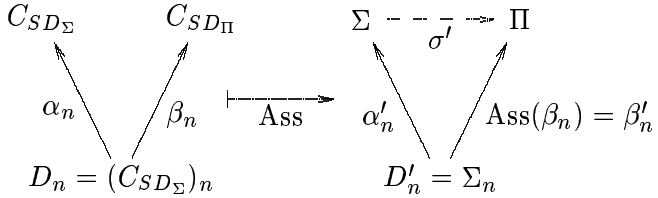
com $\widetilde{\alpha'_n}$ a extensão única de α'_n às Σ-fórmulas.

Os α_n assim definidos são morfismos de $\mathcal{SD}$ e temos então que α é um co-cone para D em $\mathcal{SD}$.

É necessário verificar de seguida que se trata de um co-cone co-limite.

Suponhamos que $\beta = \{\beta_n : (C_{SD_\Sigma})_n \to C_{SD_\Pi}\}$ é um outro co-cone para D em $\mathcal{SD}$, nesse caso tem de existir um morfismo f em $\mathcal{SD}$, tal que $\alpha f = \beta$.

Aplicando o functor Ass aos co-cones α e β obtemos



Dado que $\{\alpha'_i : D'_i \to \Sigma\}$ é um co-cone co-limite para D' em $\mathcal{ASS}$, existe um único σ' tal que $\alpha'_n\sigma' = \beta'_n$ para todo o objecto n de G. Então existe um único $f : C_{SD_\Sigma} \to C_{SD_\Pi}$ tal que $\alpha_n f = \beta_n$ para todo o objecto n de G, nomeadamente a extensão única de σ' aos sistemas de dedução.

Pela forma como foi definido f é um morfismo de Σ-sistemas de dedução,

$$\begin{array}{rcl} f : C_{SD_\Sigma} & \dashrightarrow & C_{SD_\Pi} \\ SD_\Sigma & \mapsto & SD_\Pi \\ s & \mapsto & \widetilde{\sigma'}(s) \\ s_1 & & \widetilde{\sigma'}(s_1) \\ p \downarrow & \mapsto & \downarrow \widetilde{\sigma'}(p) \\ s_2 & & \widetilde{\sigma'}(s_2) \end{array}$$

Temos então que $\{\alpha_i : D_i \to C_{SD_\Sigma}\}$ é um co-cone co-limite para D em $\mathcal{SD}$ com $\mathrm{Ass}(\alpha) = \alpha'$, sendo assim e atendendo à definição 3.17 o functor Ass reflecte co-limites.

c.q.d.

Definição 3.18 (Categoria co-completa) *Uma categoria $\mathcal{C}$ é dita co-completa se tem co-limites para todos os diagramas e é dita finitamente co-completa se tem co-limites para todos os diagramas finitos (Goguen & Burstall, 1992; Herrlich & Strecker, 1973).*

Temos então que, pelos teoremas 3.6 e 3.8, concluímos que a categoria $\mathcal{SD}$ é co-completa.

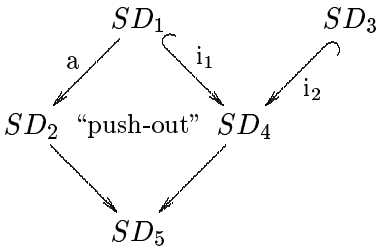
Teorema 3.9 *A categoria $\mathcal{SD}$ é co-completa.*

Demonstração. Dado que, pelo teorema 3.6, a categoria $\mathcal{ASS}$ é co-completa, e dado que, pelo teorema 3.8, o functor $\mathrm{Ass} : \mathcal{SD} \to \mathcal{ASS}$ reflecte co-limites, então, pela definição 3.18, a categoria $\mathcal{SD}$ é co-completa.

c.q.d.

3.5 Um exemplo

Pretende-se construir um sistema de dedução para a demonstração da correcção parcial de programas que manipulam pilhas de naturais, não por construção directa mas sim através do seguinte quadrado co-cartesiano ("push-out").



Em que, SD_1 é um sistema de dedução para pilhas genéricas, SD_2 para pilhas de naturais, SD_3 para programas, SD_4 para programas que podem manipular pilhas genéricas, e SD_5 para programas que podem manipular pilhas de naturais. Na descrição dos vários sistemas vai-se assumir que todas as variáveis estão quantificadas universalmente nas suas espécies respectivas.

Todos os sistemas referidos pressupõem a existência de um sistema de dedução para uma linguagem lógica de primeira ordem. Os sistemas são construídos por extensão da linguagem lógica, caso seja necessário, e por extensão do conjuntos de axiomas e de regras de inferência.

3.5.1 Sistema SD_1, Pilhas genéricas

Pretende-se com o sistema SD_1 ter um sistema de dedução para raciocinar sobre pilhas genéricas. Temos então um sistema com uma assinatura multi-espécie e cujo conjunto de símbolos de operadores inclui os operadores habituais das pilhas.

Os problemas decorrentes da aplicação dos operadores *top* e *pop* à pilha vazia foram resolvidos atribuindo valores concretos aos operadores para esses casos particulares.

Os objectos de SD_1 são definidos tendo como base uma assinatura multi-espécies que define as espécies, os símbolos de operadores para cada espécie, assim como as respectivas aridades. Temos então a seguinte assinatura $\Sigma_1 = \langle E_1, O_1 \rangle$, em que o conjunto das espécies E_1, e o conjunto dos símbolos de operadores O_1 são definidos do seguinte modo:

Espécies:
$E_1 = \{\text{pilhas}, \text{elementos}\}$

Símbolos de Operadores:
$O_1 = \{a_0, \ldots, a_n, \sqcup, \text{push}, \text{pop}, \text{top}\}$

Aridades:

a_i	:	$\mathbf{1} \rightarrow$ elementos, $\quad 0 \leq \text{i} \leq \text{n}$
$\sqcup$	:	$\mathbf{1} \rightarrow$ pilhas
push	:	pilhas elementos $\rightarrow$ pilhas
pop	:	pilhas $\rightarrow$ pilhas
top	:	pilhas $\rightarrow$ elementos

A esta assinatura é necessário acrescentar as espécies e operadores referentes a uma lógica de primeira ordem intuicionista, assim como o símbolo de igualdade (Lambek & Scott, 1988; Szabo, 1976; Szabo, 1978).

Esta assinatura determina um conjunto de sistemas de dedução cujos objectos são fórmulas geradas indutivamente a partir dos símbolos definidos na assinatura. Dentro desses sistemas interessa-nos destacar aqueles cujo conjunto de morfismos inclui, além dos morfismos definidos para todo o sistema de dedução, um conjunto de morfismos e regras de dedução que definem a operacionalidade dos objectos pilhas.

Temos então os morfismos que definem os objectos pilhas sobre os quais queremos raciocinar, aos quais se vão adicionar os morfismos, e regras de inferência próprios da lógica de primeira ordem intuicionista e com símbolo de igualdade, como já referido.

Morfismos:

$$\begin{array}{lcl} \top & \rightarrow & \mathrm{pop}(\mathrm{push}(p,x)) \doteq p \\ \top & \rightarrow & \mathrm{top}(\mathrm{push}(p,x)) \doteq x \\ \top & \rightarrow & \mathrm{top}(\sqcup) \doteq \sqcup \\ \top & \rightarrow & \mathrm{pop}(\sqcup) \doteq a_0 \end{array}$$

3.5.2 Sistema SD_2, Pilhas de Naturais

Este sistema resulta do ajuste do sistema SD_1 para o caso em que os elementos das pilhas são números naturais.

Considerando nat como um subconjunto finito de $\mathbb{N}$, temos:

$$\begin{array}{lccll} \sigma: & \Sigma_1 & \longrightarrow & \Sigma_2 & \\ & \text{elementos} & \longmapsto & \text{nat} & \\ & a_i & \longmapsto & i, & 0 \leq i \leq n \\ & * & \longmapsto & *, & \text{quaisquer outros símbolos.} \end{array}$$

Esta função extende-se de forma única às fórmulas definindo deste modo o morfismo entre sistemas de dedução.

3.5.3 Sistema SD_3, Programas (Instruções + Elems)

O sistema SD_3 é um sistema de dedução para raciocinar sobre programas que manipulam elementos. A linguagem de programação descrita é uma simplificação da linguagem descrita no capítulo 2.

As regras de inferência são também uma versão simplificada das regras de inferência descritas no capítulo 2.

Assim como para o sistema SD_1 vai-se enriquecer a assinatura que vai ser descrita com as espécies e símbolos de operadores de uma lógica de primeira ordem intuicionista com símbolo de igualdade. A espécie bool usada aquando da descrição das aridades de alguns dos símbolos de operadores, deve ser entendida como a espécie co-domínio dos símbolos desse cálculo.

Na definição da assinatura $\Sigma_3 = \langle E_3, O_3 \rangle$ vai-se usar uma notação, a exemplo do que foi feito no capítulo 2, semelhante à sintaxe da linguagem de programação *OBJ3*. Ou seja o símbolo auxiliar "_" serve como variável sintáctica, definindo a posição para os vários argumentos dos diferentes operadores.

Espécies:
$E_3 = \{\text{elementos}, \text{insts}\}$

Símbolos de operadores.
$O_1 = \{a_1, \ldots, a_n, \text{nula}, \_:=\_, \text{se\_ então\_ senão\_ fimse}, \text{enquanto\_ faz\_ fimenquanto}, \_;\{\_\}\_, \{\_\}\_\{\_\}\}$

Aridades:

$$\begin{array}{rcl} a_i & : & \mathbf{1} \to \text{elementos}, \qquad 0 \leq \text{i} \leq \text{n} \\ \text{nula} & : & \mathbf{1} \to \text{insts} \\ \_:=\_ & : & \text{elementos elementos} \to \text{insts} \\ \text{se\_ então\_ senão\_ fimse} & : & \text{bool insts insts} \to \text{insts} \\ \text{enquanto\_ faz\_ fimenquanto} & : & \text{bool insts} \to \text{insts} \\ \_;\{\_\}\_ & : & \text{insts bool insts} \to \text{insts} \\ \{\_\}\_\{\_\} & : & \text{bool insts bool} \to \text{bool} \end{array}$$

Regras de Inferência:

$$\frac{A{\to}\{P\}S\{R\} \quad A{\to}R{\supset}Q}{A{\to}\{P\}S\{Q\}}\ \text{consd} \qquad \frac{A{\to}P{\supset}R \quad A{\to}\{R\}S\{Q\}}{A{\to}\{P\}S\{Q\}}\ \text{conse}$$

$$\frac{}{A{\to}\{P\}\text{nula}\{P\}}\ \text{nula} \qquad \frac{A{\to}\{P\}S_1\{R\} \quad A{\to}\{R\}S_2\{Q\}}{A{\to}\{P\}S_1;\{R\}S_2\{Q\}}\ \text{comp}$$

$$\frac{}{A{\to}\{P^x_e\}x{:=}e\{P\}}\ \text{atr} \qquad \frac{A{\to}\{P{\wedge}B\}S_1\{Q\} \quad A{\to}\{P{\wedge}{\neg}B\}S_2\{Q\}}{A{\to}\{P\}\text{se } B \text{ então } S_1 \text{ senão } S_2 \text{ fimse}\{Q\}}\ \text{se}$$

$$\frac{A{\to}\{P{\wedge}B\}S\{P\}}{A{\to}\{P\}\text{enquanto } B \text{ faz } S \text{ fimenquanto}\{P{\wedge}{\neg}B\}}\ \text{enquanto}$$

3.5.4 Sistema SD_4, Programas (Instruções + Elems + Pilhas)

O sistema SD_4 é um sistema para raciocinar sobre programas que manipulam elementos e pilhas de elementos. Assim como nos sistemas anteriores considera-se integrado neste sistema uma linguagem lógica de primeira ordem intuicionista contendo o símbolo de igualdade.

Espécies:
$E_4 = \{\text{elementos}, \text{pilhas}, \text{insts}\}$

Símbolos de operadores:
$O_4 = \{a_1, \ldots, a_n, \sqcup, \text{nula}, \_:=_{\text{elementos}}, \_:=_{\text{pilhas}}, \text{se\_ então\_ senão\_ fimse}, \text{enquanto\_ faz\_ fimenquanto}, \_;\{\_\}\_, \{\_\}\_\{\_\}, \text{top}, \text{pop}, \text{push}\}$

Aridades:

$$\begin{array}{rcl} a_i & : & \mathbf{1} \to \text{elementos}, \qquad 0 \leq \text{i} \leq \text{n} \\ \sqcup & : & \mathbf{1} \to \text{pilhas} \\ \text{pop} & : & \text{pilhas} \to \text{pilhas} \\ \text{top} & : & \text{pilhas} \to \text{elementos} \\ \text{push} & : & \text{pilhas elementos} \to \text{pilhas} \\ \text{nula} & : & \mathbf{1} \to \text{insts} \\ \_:=\_{\text{elementos}} & : & \text{elementos elementos} \to \text{insts} \\ \_:=\_{\text{pilhas}} & : & \text{pilhas pilhas} \to \text{insts} \\ \text{se\_ então\_ senão\_ fimse} & : & \text{bool insts insts} \to \text{insts} \\ \text{enquanto\_ faz\_ fimenquanto} & : & \text{bool insts} \to \text{insts} \\ \_;\{\_\}\_ & : & \text{insts bool insts} \to \text{insts} \\ \{\_\}\_\{\_\} & : & \text{bool insts bool} \to \text{bool} \end{array}$$

Morfismos:

$\top \to \text{pop}(\text{push}(p, x)) \doteq p$

$\top \to \text{top}(\text{push}(p, x)) \doteq x$

$\top \to \text{top}(\sqcup) \doteq \sqcup$

$\top \to \text{pop}(\sqcup) \doteq a_0$

Regras de Inferência:

$$\frac{A \to \{P\}S\{R\} \quad A \to R \supset Q}{A \to \{P\}S\{Q\}}\ \text{consd} \qquad \frac{A \to P \supset R \quad A \to \{R\}S\{Q\}}{A \to \{P\}S\{Q\}}\ \text{conse}$$

$$\frac{}{A \to \{P^x_e\}x{:=}e_{\text{elementos}}\{P\}}\ \text{atr}_{\text{elementos}} \qquad \frac{A \to \{P\}S_1\{R\} \ \ A \to \{R\}S_2\{Q\}}{A \to \{P\}S_1;\{R\}S_2\{Q\}}\ \text{comp}$$

$$\frac{}{A \to \{P^x_e\}x{:=}e_{\text{pilhas}}\{P\}}\ \text{atr}_{\text{pilhas}} \qquad \frac{A \to \{P \wedge B\}S_1\{Q\} \ \ A \to \{P \wedge \neg B\}S_2\{Q\}}{A \to \{P\}\text{se } B \text{ então } S_1 \text{ senão } S_2 \text{ fimse}\{Q\}}\ \text{se}$$

$$\frac{}{A \to \{P\}\text{nula}\{P\}}\ \text{nula} \qquad \frac{A \to \{P \wedge B\}S\{P\}}{A \to \{P\}\text{enquanto } B \text{ faz } S \text{ fimenquanto}\{P \wedge \neg B\}}\ \text{enquanto}$$

3.5.5 Definição dos morfismos

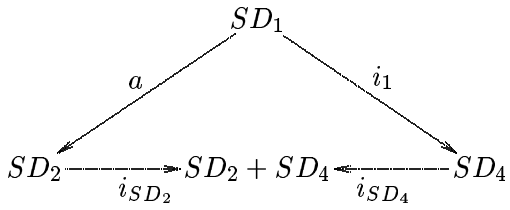
O morfismo $a : C_{SD_1} \to C_{SD_2}$ já foi definido na secção 3.5.2. A definição dos morfismos de inclusão $i_1 : C_{SD_1} \to C_{SD_4}$ e $i_2 : C_{SD_3} \to C_{SD_4}$ é trivial, estabelecendo-se a correspondência entre os objectos e morfismos dos sistema de partida, e os mesmos objectos e morfismos no sistema de chegada.

3.5.6 Construção do sistema SD_5

O sistema SD_5 pode então ser construído por aplicação de construção de um quadrado co-cartesiano aplicado aos morfismos a e i_1.

Vejamos que o sistema que assim se obtem é o que pretendemos. Para tal vai-se fazer a construção do quadrado co-cartesiano através de um co-igualizador de um co-produto.

Construção do co-produto de SD_2 e SD_4. O co-produto que pretendemos construir é dado pelo seguinte diagrama:



A assinatura $\Sigma_2 + \Sigma_4$, é caracterizado do seguinte modo:

Espécies:
$E_2 + E_4 = \{\text{pilhas}_2, \text{nat}, \text{elementos}, \text{pilhas}_4, \text{insts}\}$

Operadores:
$O_2 + O_4 = \{0, \ldots, n, \sqcup_2, \text{push}_2, \text{pop}_2, \text{top}_2, a_0, \ldots, a_n, \sqcup_4, \text{push}_4, \text{pop}_4,$
$\text{top}_4, \text{nula}, \_:=_{\text{elementos}}, \_:=_{\text{pilhas}}, \text{se\_ então\_ senão\_ fimse},$
$\text{enquanto\_ faz\_ fimenquanto}, \_;\{\_\}\_, \{\_\}\_\{\_\}\}$

As aridades dos operadores são aquelas já descritas aquando da caracterização dos sistemas SD_2 e SD_4.

Os objectos do sistema de dedução pertencentes a $C_{SD_2+SD_4}$ são definidos indutivamente a partir dos símbolos definidos na assinatura.

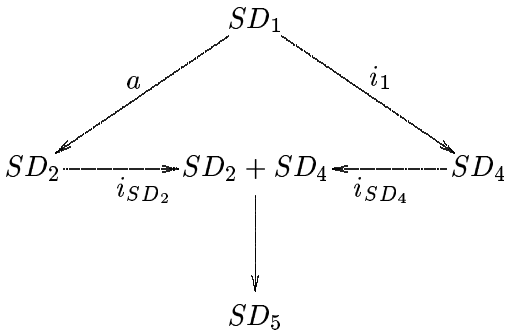
Os morfismos de $SD_2 + SD_4$ vão ser dados pelo co-produto dos dois conjuntos de morfismos de SD_2 e de SD_4.

Temos assim

Morfismos:
$\top_2 \to \text{pop}_2(\text{push}_2(p, x)) \doteq_2 p$
$\top_2 \to \text{top}_2(\text{push}_2(p, x)) \doteq_2 x$
$\top_2 \to \text{top}_2(\sqcup_2) \doteq_2 \sqcup_2$
$\top_2 \to \text{pop}_2(\sqcup_2) \doteq_2 0$
$\top_4 \to \text{pop}_4(\text{push}_4(p, x)) \doteq_4 p$
$\top_4 \to \text{top}_4(\text{push}_4(p, x)) \doteq_4 x$
$\top_4 \to \text{top}_4(\sqcup_4) \doteq_4 \sqcup_4$
$\top_4 \to \text{pop}_4(\sqcup_4) \doteq_4 a_0$

As regras de inferência são as regras de inferência do sistema SD_4. É de notar que o sistema $SD_2 + SD_4$ vai conter duas cópias disjuntas da já referida linguagem lógica de primeira ordem, isto dado que ambos os sistemas incorporam essa mesma linguagem.

Construção do co-igualizador de $i_{SD_2}a$ e de $i_{SD_4}i_1$. O co-igualizador que se pretende construir é dado pelo seguinte diagrama:



O co-equalizador vai tornar equivalentes os elementos que são imagens por $i_{SD_2}a$ e $i_{SD_4}i_1$ do mesmo elemento de SD_1.

Em termos das assinatura isto vai querer dizer que:

Espécies:
$\text{pilhas}_2 \equiv \text{pilhas}_4$
$\text{nat} \equiv \text{elementos}$

Operadores:
$i \equiv a_i, \qquad 0 \leq i \leq n$
$\sqcup_2 \equiv \sqcup_4$
$\text{push}_2 \equiv \text{push}_4$
$\text{pop}_2 \equiv \text{pop}_4$
$\text{top}_2 \equiv \text{top}_4$
$\_:=\_{\text{nat}} \equiv \_:=\_{\text{elementos}}$

Esta equivalência extende-se, de modo único, para os objectos do sistema de dedução resultante do co-igualizador.

De igual modo estabelecem-se a seguintes equivalências referentes aos morfismos:

Morfismos:
$\top_2 \to \text{pop}_2(\text{push}_2(p, x)) \doteq_2 p \equiv \top_4 \to \text{pop}_4(\text{push}_4(p, x)) \doteq_4 p$
$\top_2 \to \text{top}_2(\text{push}_2(p, x)) \doteq_2 x \equiv \top_4 \to \text{top}_4(\text{push}_4(p, x)) \doteq_4 x$
$\top_2 \to \text{top}_2(\sqcup_2) \doteq_2 \sqcup_2 \equiv \top_4 \to \text{top}_4(\sqcup_4) \doteq_4 \sqcup_4$
$\top_2 \to \text{pop}_2(\sqcup_2) \doteq_2 0 \equiv \top_4 \to \text{pop}_4(\sqcup_4) \doteq_4 a_0$

Os dois cálculos proposicionais intuicionistas fundem-se num só cálculo por efeito do co-igualizador.

Em conclusão, SD_5 é um sistema de dedução para programas que manipulam naturais e pilhas de naturais como pretendíamos.

Exemplo de demonstração. Como exemplo de uma demonstração desenvolvida em SD_5 vejamos a demonstração de $f : \top \to \{5 = 5\}x := \text{top}(\text{push}(\sqcup, 5))\{x = 5\}$. Considerando as regras de inferência de SD_5 temos, $atr : \top \to \{5 = 5\}x := 5\{x = 5\}$, temos também o axioma, $ax1 : \top \to \forall_p \forall_e \text{top}(\text{push}(p, e)) = e$. Considerando os seguinte axiomas próprias da linguagem lógica de primeira ordem intuicionista: $\Phi_\sqcup : \forall_p \forall_e \text{top}(\text{push}(p, e)) = e \to \forall_e \text{top}(\text{push}(\sqcup, e)) = e$, $\Phi_5 : \forall_e \text{top}(\text{push}(\sqcup, e)) = e \to \text{top}(\text{push}(\sqcup, 5)) = 5$ (Szabo, 1976; Szabo, 1978), e a regra de inferência para a igualdade (Lambek & Scott, 1988) temos então a seguinte demonstração,

$$f = eq(\Phi_5 \Phi_\sqcup ax1, atr)$$

No apêndice A apresenta-se uma representação desta demonstração em forma de árvore de demonstração.

Capítulo 4

Construção do Sistema SD_5 em *2OBJ*

4.1 Introdução

A construção modular de um sistema de dedução, por aplicação dos conceitos descritos no capítulo anterior, passa pela tentativa de escolha de um sistema de demonstração automática que possua os mecanismos necessários a uma tal construção.

O sistema de demonstração automática *2OBJ* surge como um potencial candidato para esta tarefa. O *2OBJ* possui mecanismos próprios para a construção modular de sistemas de dedução, mais concretamente, o *2OBJ* está baseado na linguagem de programação genérica *OBJ3*, linguagem essa que possui alguns dos mecanismos necessários para a construção modular e parametrizável de programas.

A implementação modular do sistema de dedução SD_5 passa então, pela construção de um conjunto de módulos, e pela caracterização da estrutura que os combina num determinado módulo global.

Temos então os seguintes passos:

- construção do módulo `SD1[X]`;
- construção do módulo `SD3`;
- construção do módulo `SD2`, por instanciação do módulo `SD1[X]` com o módulo `NAT`;
- construção do módulo `SD4[X]`, por importação dos módulos `SD1[X]` e `SD3`;
- construção do módulo `SD5`, por importação dos módulos `SD2` e `SD3`, ou por instanciação do módulo `SD4[X]` com o módulo `NAT`.

A construção do sistema SD_5, tal como foi descrito no capítulo anterior, não é possível com o sistema *2OBJ*. Isto verifica-se porque os mecanismos que o *OBJ3* possue para a construção modular e parametrizável de programas, embora bastante poderosos, não têm ainda a capacidade de construção expressa pelo cálculo de co-limites (Goguen *et al.*, 1992).

Antes de se passar à descrição concreta da implementação dos diferentes sistemas, vai-se descrever as capacidades de programação modular e parametrizável do *OBJ3*, capacidades essas que foram usadas na construção dos vários sistemas lógicos.

4.2 A modularidade em *OBJ3*

A descrição que de seguida se faz da linguagem *OBJ3*, vai centrar-se no estudo das capacidades de construção modular e parametrizável de programas, esquecendo muitos dos pormenores básicos da linguagem. Dado se pretender introduzir os mecanismos que permitiram a implementação modular do sistema SD_5, a apresentação exaustiva da linguagem não é, a nosso ver, necessária. Para o eventual interessado numa apresentação detalhada da linguagem de programação *OBJ3*, recomenda-se a leitura de "Introducing OBJ" (Goguen *et al.*, 1992).

Todos os resultados aqui apresentados, assim como alguns dos exemplos, estão no texto atrás referido.

4.2.1 A construção de módulos

O *OBJ3* é uma linguagem de programação funcional de primeira ordem, baseada na lógica equacional, com ordenamento de espécies.

Um programa em *OBJ3* tem a estrutura de um grafo genérico. Os nós dos grafos são módulos, e as arestas ou são devidas às relações de inclusão de sub-módulos, ou são explicitamente definidos à custa de interpretações de um módulo num outro.

Vejamos com um pouco mais de pormenor cada uma das entidades referidas acima.

Um módulo em *OBJ3* designa, ou um "*object*" (objecto) ou uma "*theory*" (teoria). Os objectos contêm o código executável, e são compostos por uma zona de declaração de "*sorts*" (espécies); outra de "*operations*" (operadores); e outra de "*equations*" (equações). As declarações das espécies e dos operadores definem a assinatura do objecto, isto é, a sua componente sintáctica. A semântica dos objectos é definida pela sua zona de declaração de equações, sendo estas interpretadas como regras de re-escrita, que substituem instâncias do lado esquerdo pelas correspondentes instâncias do lado direito. A semântica operacional do *OBJ3* está baseada na re-escrita de termos com ordenamento de espécies.

Vejamos como exemplo, um objecto que implementa sequências de "bits".

```
obj BITS is
 sorts Bit Bits .
 subsorts Bit < Bits .
 ops 0 1 : -> Bit .
 op __ : Bit Bits -> Bits .
endo
```

Dado que não contém equações, este objecto define o conjunto de todas as sequências de "bits" criadas por justaposição dos símbolos `0` e `1`.

Uma teoria em *OBJ3* vai definir o interface de um módulo parametrizado, isto é, vai definir qual é a estrutura, e quais são as propriedades requeridas para se ter uma instanciação correcta do módulo em questão. Semanticamente uma teoria é uma "variedade" de modelos, contendo todas as álgebras com ordenamento de espécies que a satisfazem, em contrapartida um objecto define só um modelo, a sua álgebra inicial. É de notar que uma teoria em *OBJ3* não pretende denotar uma teoria lógica, isto é, um conjunto de fórmulas fechado para uma dada relação de demonstrabilidade (Gallier, 1987).

Embora a linguagem *OBJ3* permita que se possam efectuar reduções nas teorias, é necessário ter em conta que as equações aí escritas não são, necessariamente, para ser interpretadas como regras de re-escritas. Como tal as reduções podem ser infinitas.

Exemplo de uma teoria para especificar uma relação de equivalência:

```
theory EQUIV is
 sort Elt .
 op _eq_ : Elt Elt -> Bool .
 vars E1 E2 E3 : Elt .
 eq (E1 eq E1) = true .
 eq (E1 eq E2) = (E2 eq E1) .
 cq (E1 eq E3) = true if (E1 eq E2) and (E2 eq E3) .
endth
```

A equação referente à reflexividade não pode ser interpretada como regra de re-escrita, pois daria azo a reduções infinitas. No caso de estarmos a definir um objecto esta equação teria de ser retirada, em sua substituição adicionar-se-ia à operação `_eq_` o atributo `comm`, especificando deste modo que a operação é comutativa.

4.2.2 Importação de módulos

Os módulos definem um primeiro nível na construção de programas em *OBJ3*. Um segundo nível é dado pela importação de módulos por um outro módulo, definindo deste modo a estrutura de um grafo acíclico de módulos.

Um arco orientado num tal grafo indica que o módulo superior (nó de destino) importa o módulo inferior (nó de partida). O contexto de um dado módulo é o sub-grafo do qual ele é o topo.

Uma importação do módulo M' pelo módulo M é do tipo:

protecting (pr) se e só se M não adiciona novos operadores de espécies de M', nem adiciona novas equações sobre os operadores de M';

extending (ex) se e só se M não adiciona novas equações sobre os operadores de M';

including (inc) ou using (us) se não há garantias de que nenhuma das situações anteriormente descritas se verifica.

A relação de importação é uma relação transitiva.

Vejamos, como exemplo, a escrita de um objecto para calcular o comprimento de uma lista de inteiros. Dado que os objectos da lista são inteiros, e dado que a função que procede ao cálculo do comprimento tem co-domínio nos inteiros, e requer a adição de inteiros para proceder ao cálculo, é necessário fazer a importação de `INT`, o módulo dos inteiros. Uma vez que não se vão acrescentar novos operadores de tipo inteiro, nem acrescentar equações sobre os operadores já definidos em `INT` a importação do módulo dos inteiros é do tipo `protecting`.

```
obj LISTA-DE-INT is
 protecting INT .
 sort Lista .
 subsort Int < Lista .
 op __ : Int Lista -> Lista .
 op comprimento_ : Lista -> Int .
 var I : Int .
 var L : Lista .
 eq comprimento I = 1 .
 eq comprimento(I L) = 1 + comprimento L .
endo
```

A instrução `protecting INT` define a importação do módulo pré-definido `INT` que contêm a implementação dos inteiros em *OBJ3*. A instrução `subsort Int < Lista` tem como efeito que todos os inteiros passam a poder ser considerados como listas unitárias, isto é, o conjunto dos inteiros passa a estar contido no conjunto das listas.

4.2.3 Módulos parametrizados

Um terceiro nível na construção de programas em *OBJ3* é nos dado pela construção de módulos parametrizados, a estrutura de um programa passa

a ser então a de um grafo genérico, como já foi referido no começo desta secção.

Retomando o exemplo do módulo `LISTA-DE-INT` vejamos como o poderiamos alterar para que ele nos dê a possibilidade de trabalhar com listas de elementos genéricos. Pretende-se então, criar um módulo que implemente listas de elementos, sem especificar de que espécie concreta é que são esses elementos, isso é possível através da construção do seguinte módulo parametrizado:

```
obj LISTA[X :: TRIV] is
 protecting INT .
 sort Lista .
 subsort Elt < Lista .
 op __ : Elt Lista -> Lista .
 op comprimento_ : Lista -> Int .
 var E : Elt .
 var L : Lista .
 eq comprimento E = 1 .
 eq comprimento(E L) = 1 + comprimento L .
endo
```

O módulo `LISTA` tem como parâmetro um módulo `X` que satisfaz a teoria `TRIV`, esta teoria está contida no módulo de arranque do *OBJ3*, tendo a seguinte definição:

```
theory TRIV is
 sort Elt .
endth
```

Podemos agora criar instâncias deste módulo dando como argumentos módulos que verifiquem a teoria `TRIV`. Por exemplo:

```
make LISTA-DE-INT is LISTA[INT] endm
```

Cria o módulo `LISTA-DE-INT` que é equivalente ao anteriormente criado sem recorrer à parametrização.

Este comando é equivalente ao comando:

```
obj LISTA-DE-INT is protecting LISTA[INT] . endo
```

A questão que fica em aberto é a forma como o módulo `INT` verifica a teoria `TRIV`.

O *OBJ3* especifica a forma como um dado módulo satisfaz uma dada teoria à custa da declaração de interpretações ("views").

Uma interpretação Φ, de uma teoria T, para um módulo M, $\Phi : T \rightarrow M$, consiste numa aplicação das espécies de T nas espécies de M, preservando

a relação de ordem entre espécies, e de uma aplicação das operações de T nas operações de M, preservando aridades assim como o significado dos atributos `assoc`, `comm`, `idem`, `id:` e `idr:`, e de tal forma que toda a equação em T é verdadeira para todo o modelo de M.

No exemplo descrito acima usou-se uma interpretação por omissão ("default view"), mecanismo pré-definido no *OBJ3* e que nos diz que:

i) todo a declaração da forma `sort S to S` pode ser omitida, excepto nos casos em que a espécie destino é uma espécie numa teoria parâmetro;

ii) uma declaração da forma `sort S to S'` pode ser omitida, se tanto `S` como `S'` são espécies principais nos respectivos módulos (as primeiras espécies a serem declaradas);

iii) todo a declaração da forma `op o to o` pode ser omitida;

iv) se `op o to o'` é uma das declarações da interpretação e se `o` e `o'` têm atributos `ide:e` e `ide:e'` respectivamente, então `op e to e'` pode ser omitido.

Temos então que a interpretação por omissão define a declaração `Elt to Int`, instanciando deste modo os elementos da lista como sendo inteiros.

Explicitando a interpretação teríamos:

```
view TRIV-PARA-INT from TRIV to INT is
 sort Elt to Int .
endv
```

A instrução que permite criar o módulo seria então:

```
make LISTA-DE-INT is LISTA[TRIV-PARA-INT] endm
```

A parametrização de módulos não está limitada a um só parâmetro. O módulo anterior pode ainda ser generalizado retirando a sua dependência da adição de inteiros. Temos então que construir uma teoria que defina uma espécie, e uma operação interna a essa espécie, operação essa que tem de possuir as propriedades que a adição possue. Podemos então definir a seguinte teoria:

```
theory ADICAO is
 sort Elt .
 ops zero um : -> Elt .
 op _mais_ : Elt Elt -> Elt [assoc comm id: zero] .
endth
```

A definição do objecto seria então:

```
obj LISTA[X :: TRIV, Y :: ADICAO] is
 sort Lista .
 subsort Elt.X < Lista .
 op __ : Elt.X Lista -> List .
 op comprimento_ : Lista -> Elt.Y .
 var E : Elt.X .
 var L : Lista .
 eq comprimento E = um .
 eq comprimento(E L) = um mais comprimento(L) .
endo
```

Para criar uma dada instância deste módulo temos que especificar duas interpretações. Para o módulo referente ao parâmetro `X` poder-se-à optar, na maior parte do casos, pela interpretação por omissão. No caso do módulo referente ao parâmetro `Y`, já será necessário explicitar qual é a operação `mais`, assim como as constantes `um` e `zero`, este módulo vai ser instânciado pelo módulo `FLOAT`, a implementação dos reais em *OBJ3*. Adoptando a escrita das interpretações integrada na construção das instâncias, temos a seguinte instância de listas de inteiros:

```
make LISTA-DE-INT-ADFLOAT is
 LISTA[NAT, view to FLOAT is op _mais_ to _+_ .
                             op um to 1.0 . endv]
endm
```

É de notar que na escrita da interpretação se adoptou as convenções próprias do *OBJ3* para as interpretações por omissão. Desta forma não foi necessário explicitar as declarações `sort Elt to Float` e `op zero to 0.0`, esta última devido ao facto das constantes `zero` e `0.0` serem os atributos de identidade das operações de adição nos respectivos módulos.

4.2.4 Expressões com módulos

Um último nível na construção de programas em *OBJ3* é dado pela escrita de expressões com módulos. As expressões com módulos incluem a instanciação de módulos, como já foi descrito atrás, a re-designação (*renaming*), e a soma.

As entidades atómicas das expressões com módulos são os módulos simples, sejam eles os módulos que definem os diferentes tipos de dados pré-definidos, como sejam os `NAT`, `INT`, e `FLOAT`, entre outros, ou os diferentes módulos não parametrizados definidos pelo programador.

A re-designação cria um novo módulo através de uma transformação nas espécies, e de uma transformação nos operadores. Como exemplo veja-se a construção de uma teoria para especificar uma relação de equivalência por re-designação de uma teoria referente a uma pré-ordem.

```
th EQUIV is
 using PREORD * (op _<=_ to _eq_) .
 vars E1 E2 : Elt .
 eq (E1 eq E2) = (E2 eq E1) .
endth
```

A outra outra possibilidade que se tem para a construção de expressões com módulos é nos dado pela soma de módulos. Por exemplo, a soma dos módulos `A` e `B` é dada por `A + B`. É de notar que a soma de módulos não implementa o co-produto de módulos como poderia deixar a entender. A expressão `A + B` é equivalente à instrução:

```
obj AB is pr A . pr B . endo
```

as relações de inclusão de módulos determinam, que a inclusão de sub-módulos comuns aos módulos que se está a importar, se faz por partilha, e não por criação de múltiplas cópias diferenciadas pela origem. Sendo assim os sub-módulos comuns a `A` e a `B` serão partilhados, não dando azo à criação de duas cópias diferentes referentes aos dois módulos em questão. O módulo que resulta da adição de módulos é considerado uma extensão das suas parcelas.

A possibilidade de as aplicações entre módulos poderem ser elementos activos na construção de expressões com módulos não está implementada em *OBJ3*, sendo assim a construção de expressões com módulos através do cálculo de co-limites, não está disponível na actual versão do *OBJ3* ("OBJ3 version 2.03").

4.3 Construção do sistema SD_5 em *2OBJ*

O módulo `SD5` que pretende implementar o sistema SD_5, vai ser construído modularmente a partir dos módulos `SD1`, `SD2`, `SD3`, e `SD4`. Vai-se então descrever cada uma desses sistemas, começando pelos módulos `SD1` e `SD3` que constituem os pontos de partida para toda a construção que de seguida se efectua.

Os módulos `SD1` e `SD3` incorporam o módulo `CPPO` que define um cálculo de sequências dedutivas para a lógica de primeira ordem, a listagem deste módulo está no apêndice B.

4.3.1 O sistema SD_1 em *2OBJ*

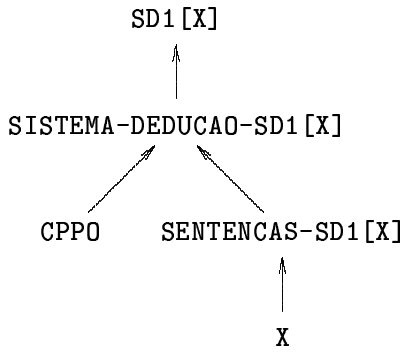
O módulo `SD1` implementa o sistema de dedução SD_1. Temos então um objecto que lida com pilhas de elementos genéricos.

Seguindo a metodologia de construção de sistemas de dedução usada na construção do módulo `FOPC` ("First Order Predicate Logic") (Stevens

& Hobley, 1993), construiu-se um módulo `SENTENCAS-SD1` que define as fórmulas da linguagem lógica de SD_1. De seguida este módulo é incorporado no módulo `SISTEMA-DEDUCAO-SD1` que define os axiomas de SD_1, por sua vez este módulo incorpora o módulo `CPPO`. O módulo `SD1` define o topo do grafo ao incorporar o módulo `SISTEMA-DEDUCAO-SD1`.

O módulo `SD1` é um módulo parametrizado, tendo como teoria interface a teoria `TRIV+`, este módulo define uma espécie `Elt` e um elemento `elt` genéricos, permitindo deste modo que o módulo `SD1` seja posteriormente instanciado, dando origem a lista de elementos de um dado tipo concreto. A espécie `Var-Elt` é necessária por razões internas ao *2OBJ*.

Temos então que `SD1` define o seguinte grafo:



Listagem de `SD1`

```
in cppo.obj

theory TRIV+ is
 sort Elt .
 op elt : -> Elt .
endth

obj SENTENCAS-SD1 [ X :: TRIV+ ] is
*** especies
 sorts Pilha Var-Pilha Var-Elt .
 subsort Var-Elt < Elt .
 subsort Var-Pilha < Pilha .
*** operadores
 op Vazia : -> Pilha .
 op push : Pilha Elt -> Pilha .
 op pop : Pilha -> Pilha .
 op top : Pilha -> Elt .
endo
```

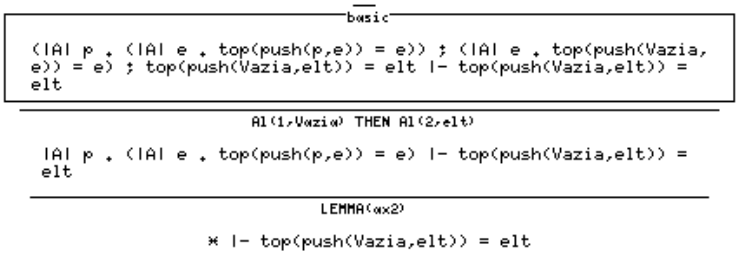
```
obj SISTEMA-DEDUCAO-SD1[X :: TRIV+] is
***
 pr SENTENCAS-SD1[X] .
 pr CPPO .
***
 subsorts Pilha Elt < Term .
 subsorts Var-Pilha Var-Elt < Var .
***
 ops ax1 ax2 ax3 ax4 : -> Axiom .
 op p : -> Var-Pilha .
 op e : -> Var-Elt .
*** axiomas
 eq axiom ax1 = ( * |- |A| p . |A| e . pop(push(p,e)) = p) .
 eq axiom ax2 = ( * |- |A| p . |A| e . top(push(p,e)) = e) .
 eq axiom ax3 = ( * |- pop(Vazia) = Vazia) .
 eq axiom ax4 = ( * |- top(Vazia) = elt) .
endo

obj SD1[X :: TRIV+] is
 pr SISTEMA-DEDUCAO-SD1[X] .
endo
```

Um exemplo

Apresenta-se de seguida a representação gráfica dada pelo *X2OBJ*, o interface gráfico do *2OBJ* para o sistema *X-Windows* (Stevens & Hobley, 1993), da árvore de demonstração para a seguinte sequência dedutiva em `SD1`.

```
* |- top(push(Vazia,elt)) = elt
```

em que `*` representa uma lista de sentenças vazia.



Partindo da raiz, isto é, a sequência dedutiva que se pretende demonstrar, aplica-se o tactical `LEMMA`, que introduz o axioma `ax2`, de seguida aplica-se, por duas vezes, a regra de inferência `Al`, eliminação do quantificador universal no lado esquerdo, o tactical `THEN` permite aplicar, `Al(1,Vazia)`, e depois

da aplicação bem sucedida desta regra de inferência, aplicar `Al(2,elt)`, eliminando deste modo os quantificadores na terceira sentença do lado esquerdo da sequência dedutiva. A árvore de demonstração fica completa com a aplicação da táctica `basic` que verifica que existe uma mesma sentença de ambos os lados da sequência dedutiva.

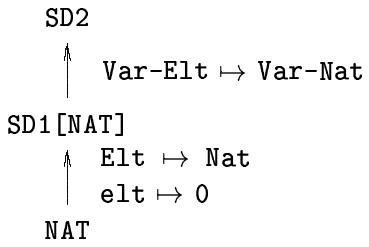
4.3.2 O sistema SD_2 em *2OBJ*

O módulo `SD2` é construído através de uma instanciação do módulo `SD1` tendo como parâmetro o módulo `NAT`, módulo pré-definido no *OBJ3*, que implementa o tipo de dados dos números naturais. A interpretação `TRIV+NAT` de `X` em `NAT` define os pares `Elt to Nat` e `elt to 0`, os elementos genéricos definidos em `SD1` são, deste modo, instanciados para os naturais.

Foi também necessário fazer um re-designação do módulo `SD1` definindo o par `Var-Elt to Var-Nat`, dado que o módulo `NAT` não possui essa espécie auxiliar, necessária para a definição de sistema de dedução em *2OBJ*.

A interpretação definida implementa o ajuste descrito no capítulo anterior.

O módulo `SD2` define o seguinte grafo:



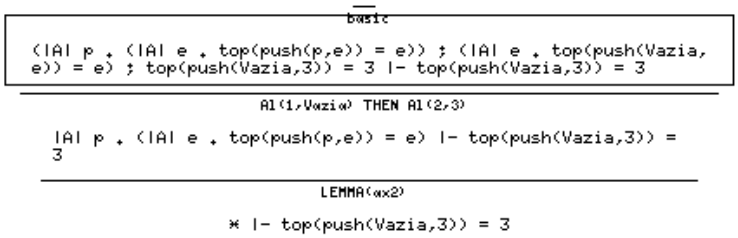
Listagem de `SD2`

```
in SD1.obj

view TRIV+NAT from TRIV+ to NAT is
 sort Elt to Nat .
 op elt to 0 .
endv

obj SD2 is
 pr (SD1 * (sort Var-Elt to Var-Nat))[TRIV+NAT] .
endo
```

Um exemplo

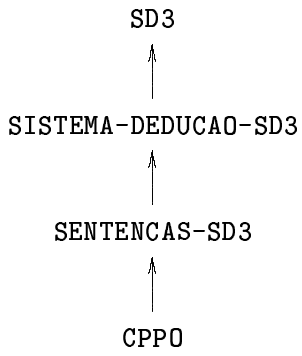
Apresenta-se de seguida a representação gráfica dada pelo *X2OBJ*, da árvore de demonstração para a seguinte sequência dedutiva em `SD2`.

```
* |- top(push(Vazia,3)) = 3
```



4.3.3 O sistema SD_3 em *2OBJ*

O módulo SD3 implementa o sistema SD_3, a sua estrutura segue de perto a metodologia de construção de sistemas de dedução em *2OBJ*. Temos então que SD3 define o seguinte grafo:



Listagem de SD3

```
in cppo.obj

obj SENTENCAS-SD3 is
***
 pr CPPO .
*** especies
 sorts Elms Inst Var-Elms Var-Inst .
 subsort Var-Elms < Elms .
 subsort Var-Inst < Inst .
 subsorts Elms Inst < Term .
 subsorts Var-Elms Var-Inst < Var .
*** operadores
 op nula : -> Inst .
 op _ := _ : Elms Elms -> Inst [prec 0] .
 op _;{_}_ : Inst Sent Inst -> Inst [prec 19 gather (e e E)] .
```

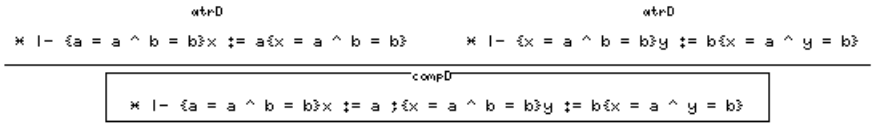
```
 op se_entao_senao_fimse : Prop Inst Inst -> Inst .
 op enquanto_faz_fimfaz : Prop Inst -> Inst .
 op {_}_{_} : Sent Inst Sent -> Sent [prec 20] .
endo


obj SISTEMA-DEDUCAO-SD3 is
***
 pr SENTENCAS-SD3 .
***
 vars X Y : Elms .
 vars P Q R : Sent .
 vars B : Prop .
 vars S S1 S2 : Inst .
 vars H : SentenceList .

 ops consdD conseD : Sent -> Rule .
 ops nulaD atrD seD enquantoD compD : -> Rule .
*** regras de inferencia
 eq nulaD ( H |- {P}nula{P} ) = [] .
 eq consdD(R) ( H |- {P}S{Q} ) =
              ( H |- ( {P}S{R} ),
              ( H |- ( R -> Q ) ) ) .
 eq conseD(R) ( H |- {P}S{Q} ) =
              ( H |- ( P -> R ) ),
              ( H |- {R}S{Q} ) .
 cq atrD ( H |- {Q}X := Y{P} ) = []
         if Q == ( P o Y / X ) .
 eq compD ( H |- {P}S1 ; {R}S2 {Q} ) =
  ( H |- {P}S1{R}),
  ( H |- {R}S2{Q}) .
 eq seD ( H |- {P}se B entao S1 senao S2 fimse{Q} ) =
        ( H |- {P ^ B}S1{Q}),
        ( H |- {P ^ (~ B)}S2{Q}).
 ceq enquantoD ( H |- {P}enquanto B faz S fimfaz{Q}) =
               ( H |- {P ^ B}S{P})
               if Q == (P ^ (~ B)) .

*** Tactical
 var G : Goal .

 op ProgsTac : -> Tactic .

 eq ProgsTac ( H |- {P}nula{Q}) =
```

```
        nulaD ELSE idtac .
 eq ProgsTac ( H |- {P}(X := Y){Q} ) =
        atrD ELSE idtac .
 eq ProgsTac ( H |- {P}se B entao S1 senao S2 fimse{Q} ) =
        (seD THEN ProgsTac) ELSE idtac .
 eq ProgsTac ( H |- {P}enquanto B faz S fimfaz{Q} ) =
        (enquantoD THEN ProgsTac) ELSE idtac .
 eq ProgsTac ( H |- {P}S1 ; {R}S2{Q} ) =
        (compD THEN ProgsTac) ELSE idtac .
 eq ProgsTac DEFAULT G = idtac .
endo

obj SD3 is
 pr SISTEMA-DEDUCAO-SD3 .
endo
```

Um exemplo

Apresenta-se de seguida a representação gráfica dada pelo *X2OBJ*, da árvore de demonstração da seguinte sequência dedutiva em `SD2`.

```
* |- {(a = a) ^ (b = b)}
     x := a ;
     {(x = a) ^ (b = b)}
     y := b
     {(x = a) ^ (y = b)}) .
```



A árvore de demonstração fica completa por aplicação das regras de inferência `compD` e `atrD` definidas em `SD3`. A completa automatização desta demonstração é possível através da aplicação do tactical `ProgsTac` definido em `SD3`.

4.3.4 O sistema SD_4 em *2OBJ*

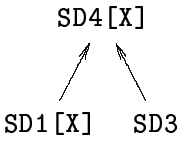
A construção do módulo `SD4` afasta-se de uma implementação directa do sistema de dedução SD_4. Enquanto que na construção de SD_4 se optou por construir o sistema, e após isso, definir as injecções de SD_1 e SD_3 para SD_4, na construção de `SD4` optou-se por construir o módulo por importação dos

módulos SD1 e SD3, sem explicitar as relações de importação dos módulos SD1 e SD3 em SD4.

A razão para tal escolha advém do facto, já acima referido, de que as expressões com módulos só terem como entidades activas os módulos, e não as aplicações entre eles. As aplicações entre módulos (interpretações, re-designações, e relações de importação) não possuem outros operadores entre elas que não seja o operador de composição.

É de notar que o efeito de sobre-designação ("overloading") do operador _ := _ em SD_4 é dado, em SD4, pela declaração `subsorts Elt Pilha < Elems`. Esta declaração tem como efeito que o operador `_:=_ : Elems Elems -> Inst` pode ter como argumentos elementos de Elt e de Pilha, dando assim a operacionalidade pretendida.

O grafo definido por SD4 é o seguinte:



Listagem de SD4

```
in SD1
in SD3

***
obj SD4[X :: TRIV+] is
 pr SD1[X] .
 pr SD3 .
 subsorts Elt Pilha < Elms .
 subsorts Var-Elt Var-Pilha < Var-Elms .
endo
```

Um exemplo

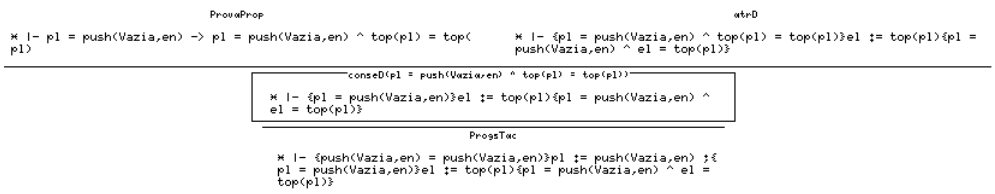
Apresentam-se de seguida uma representação gráfica simplificada da árvore de demonstração da seguinte sequência dedutiva em SD4.

```
* |- {(push(Vazia,en) = push(Vazia,en))}
     pl := push(Vazia,en) ;
     {(pl = push(Vazia,en))}
     el := top(pl)
     {(pl = push(Vazia,en)) ^ (el = top(pl))}) .
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1.2.1.1.1}\,\text{basic}\quad\dfrac{}{1.2.1.1.2}\,\text{eq}}{1.2.1.1}\,\text{AndR}}{1.2.1}\,\text{ImpR}\qquad\dfrac{}{1.2.2}\,\text{atrD}}{1.2}\,\text{conseD}\begin{pmatrix}\text{pl=push(Vazia,en)}\\ \wedge\\ \text{top(pl)=top(pl)}\end{pmatrix}\qquad\dfrac{}{1.1}\,\text{atrD}}{1}\,\text{compD}$$

A representação gráfica obtida através do *X2OBJ* é demasiado grande para ser aqui incluída, a sua apresentação é feita no apêndice A.

A necessidade de aplicar a regra de inferência do consequente `conseD`, impede a total automatização da demonstração. No entanto se se importar o módulo `TACTICAS`, módulo que define um conjunto de tácticas e tacticais para o cálculo de sequências dedutivas para a lógica de primeira ordem, pode-se realizar uma demonstração muito mais compacta, pela aplicação dos tacticais `ProgsTac` definido em `SD3`, e `ProvaProp` definido em `TACTICAS`. De seguida apresenta-se a representação gráfica dada pelo *X2OBJ* da árvore de demonstração obtida por aplicação destes dois tacticais.

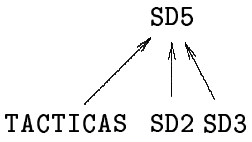


4.3.5 O sistema SD_5 em *2OBJ*

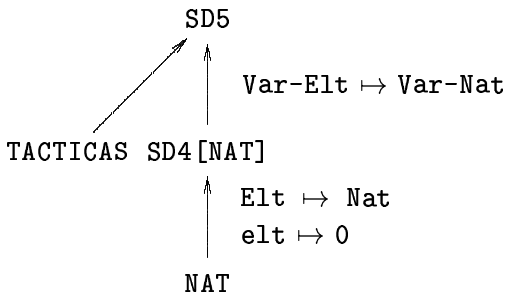
A implementação de SD_5 como quadrado co-cartesiano dos morfismos $a : SD_1 \to SD_2$ e $i_1 : SD_1 \to SD_4$, não é possível em *2OBJ*. Como já foi referido as capacidades da linguagem *OBJ3* estão limitadas à construção de expressões com módulos, ficando as aplicações entre módulos com um papel auxiliar em tais construções. A possibilidade de dotar a linguagem *OBJ3* com a capacidade de construções baseadas no cálculo de co-limites, é referida em (Goguen *et al.*, 1992), no entanto na actual versão do *OBJ3* ("OBJ3 version 2.03") isso não foi considerado.

Temos então que o módulo `SD5` foi construído por importação dos módulos `SD2` e `SD3`, acrescentando de seguida a declaração `subsorts Nat Pilha < Elems` de forma a que os programas (`SD3`) possam lidar com os naturais e as pilhas (`SD2`)

O grafo definido por `SD5` é então:



Outra forma, alternativa, de definir SD5 seria por instanciação de `SD4[X]` com o módulo dos naturais através de uma interpretação, e de uma redesignação, a exemplo do que já foi feito para `SD2`. Teríamos então:



O módulos `SD5` importa também o módulo `TACTICAS`, o qual vai ser necessário para o completar da demonstração num exemplo que iremos apresentar.

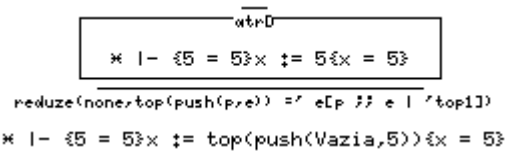
Listagem de `SD5`

```
in SD2.obj
in SD3.obj
in tacs.obj

obj SD5 is
 pr SD2 .
 pr SD3 .
 pr TACTICAS .
 subsorts Nat Pilha < Elms .
 subsorts Var-Nat Var-Pilha < Var-Elms .
endo
```

Um exemplo

Apresenta-se de seguida a representação gráfica dada pelo *X2OBJ*, da árvore de dedução para a seguinte sequência dedutiva em `SD5`.

```
* |- {5 = 5}x := top(push(Vazia,5)){x = 5}
```



O tactical `reduze` definido no módulo `TACTICAS` faz a ligação entre o *2OBJ* e o mecanismo de re-escrita do *OBJ3*, permitindo deste modo, que se possa aplicar uma determinada equação como regra de re-escrita, no caso apresentado foi a equação `top(push(p,e))=e`, reduz-se desta forma, um dado termo à sua forma normal, no caso do exemplo `top(push(Vazia,5))` reduz para `5`.

4.4 Conclusões

A escolha do sistema de demonstração automática *2OBJ* foi baseada no conhecimento que se tinha da linguagem *OBJ3*, e no verificar que as suas capacidades de construção modular e parametrizável de programas eram superiores às de outras linguagens de programação, nas quais se baseiam os outros sistemas de demonstração automática, isso mesmo é referido em (Goguen *et al.*, 1992).

A possibilidade de modificar a linguagem de programação *OBJ3* de molde a esta poder incluir as capacidades próprias ao cálculo de co-limites, parece ser possível, no entanto não há indicações de que tal esteja a ser considerado pelos seus autores (Goguen *et al.*, 1992).

Dado que o programa fonte, referente à linguagem de programação *OBJ3*, está disponível, essa modificação pode ser encarada como um projecto de trabalho futuro, com vista à aplicação dos resultados estabelecidos no capítulo anterior.

Capítulo 5

Possíveis Extensões

Os exemplos apresentados neste trabalho quanto à combinação de sistemas lógicos, têm subjacente a ideia de se estar a trabalhar dentro de uma mesma linguagem lógica. No exemplo apresentado aquando da construção de SD_5, todos os sistemas lógicos estão baseados numa linguagem lógica de primeira ordem, sobre um cálculo de sequências dedutivas, sendo o sistema resultante também do mesmo tipo. No entanto na demonstração de {5=5}x:=top(push(Vazia,5)){x=5} no sistema SD5 (ver secção 4.3.5), foi usado o mecanismo de re-escrita do *OBJ3*, através de uma regra de inferência em CPPO, ou seja usou-se a inferência equacional num dos passos de uma demonstração feita num cálculo de asserções de dedutibilidade para a lógica de primeira ordem.

A necessidade de dar um suporte teórico à combinação de sistemas lógicos abre uma área de estudo muito interessante, veja-se a este propósito a extensa bibliografia citada por A. Sernadas, C. Sernadas e C. Caleiro (1997a; 1997b; 1997c).

De seguida apresentamos alguns dos resultados contidos nos artigos atrás referidos os quais afigura-se-nos possível incorporar, de alguma forma, no trabalho desenvolvido sobre os sistemas de dedução. A notação difere um pouco daquela que é usada pelos autores, tendo as alterações sido feitas com vista a uniformizar com as convenções usadas no restante texto, ou seja a notação usada para as categorias e functores, e para as classes dos objectos e dos morfismos foi alterada em relação à usada pelos autores, sendo adoptada a notação que foi definida no capítulo 3.

5.1 Fibrilação de Lógicas

Em (Sernadas *et al.*, 1997a) é estudado o mecanismo de combinação de lógicas designado pelos autores de fibrilação de lógicas ("fibring of logics"). Pretende-se com a fibrilação de lógicas ter a possibilidade de misturar as conectivas assim como as regras de inferência das diferentes lógicas envolvi-

das na fibrilação. Interessa-nos aqui destacar a fibrilação de lógicas numa aproximação própria à teoria da dedução, esse estudo passa pela definição de uma categoria cujos objectos são sistemas de dedução e na qual a fibrilação de sistemas de dedução possa ser considerada como uma operação interna.

Os autores definem um *Cálculo de Hibert*, como sendo uma entidade que agrega os construtores da linguagem, assim como os mecanismos de inferência.

Definição 5.1 (Cálculo de Hilbert) *Um cálculo de Hilbert é um triplo* $\langle C, P, D\rangle$ *onde:*

- C *é uma assinatura;*
- $P \subseteq \mathscr{P}_{fin}(L(C,\Theta)) \times L(C,\Theta))$*;*
- $D \subseteq (\mathscr{P}_{fin}(L(C,\Theta))\backslash\emptyset) \times L(C,\Theta))$.

tal que

- $D \subseteq P$.

(Sernadas et al.*, 1997a).*

Com Θ um conjunto fixo de variáveis (esquemáticas), $L(C,\Theta)$, um conjunto de (esquemas de) fórmulas geradas com símbolos da assinatura C, e pelas variáveis (esquemáticas). Cada elemento $r = \langle Prem(r), Conc(r)\rangle$ de P é um esquema de regra de inferência ("proof rule"), e cada elemento de D é um esquema de regra de derivação ("derivation rule").

De seguida os autores definem morfismo entre dois cálculos de Hilbert como sendo um morfismo de assinaturas que preserva as regras de inferência, definindo então a categoria $\mathcal{HIL}$, a categoria dos cálculos de Hilbert, e dos morfismos entre cálculos de Hilbert.

5.1.1 Fibrilação de lógicas sem restrições

Ao considerar a fibrilação de lógicas sem restrições pretende-se obter uma lógica que combine os construtores e as regras de inferência de ambas as lógicas, temos então que a construção que se pretende obter é dada pelo co-produto em $\mathcal{HIL}$ dos correspondentes cálculos de Hilbert.

Proposição 5.1 *A fibrilação de lógicas sem restrições é dada pelo co-produto dos cálculos de Hilbert correspondentes em* $\mathcal{HIL}$ *(Sernadas* et al.*, 1997a).*

5.1.2 Fibrilação de lógicas com restrições

Na fibrilação com restrições pretende-se que os dois cálculos em consideração interactuem, seja por partilha de construtores, seja por adição de novas regras.

A definição das restrições, no caso da partilha de construtores, é dada ao nível das assinaturas. Essas restrições são depois elevadas ao nível do cálculo de Hilbert correspondente através do mecanismo de elevação co-cartesiana. Tem-se então a seguinte definição:

Definição 5.2 (Fibrilação de lógicas com restrições) *Sejam* $\langle C', P', D' \rangle$ *e* $\langle C'', P'', D'' \rangle$ *dois cálculos de Hilbert,* $f' : C \to C'$ *e* $f'' : C \to C''$ *dois morfismos de assinaturas injectivos. A fibrilação, com restrições por partilha de construtores, dos dois sistemas é dada por:*

$$\langle C', P', D' \rangle \overset{f'Cf''}{\oplus} \langle C'', P'', D'' \rangle = q(\langle C', P', D' \rangle \oplus \langle C'', P'', D'' \rangle)$$

onde $q : C' \oplus C'' \to C' \overset{f'Cf''}{\oplus} C''$ *é o co-igualizador de* $i'f' : C \to C' \oplus C''$ *e* $i''f'' : C \to C' \oplus C''$ *(Sernadas* et al.*, 1997a).*

5.1.3 Fibrilação de lógicas em $\mathcal{SD}$

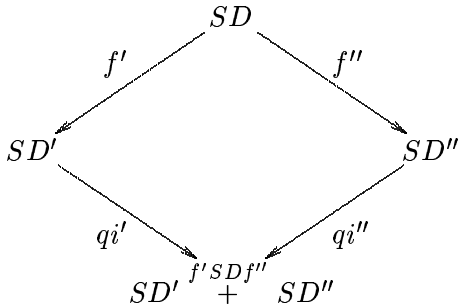
Os sistemas de dedução, tal como foram definidos neste texto (ver definição 3.2), agregam numa mesma entidade as fórmulas (objectos) e as deduções (setas). Para se obter um cálculo de Hilbert a partir de um Σ-sistema de dedução (ver definição 3.8) basta definir as deduções (setas) e as regras de inferência (operações com setas) de forma apropriada.

Tendo isto em conta julgamos ser possível demonstrar que:

- A fibrilação de lógicas sem restrições é dada, em $\mathcal{SD}$, pelo co-produto dos sistemas de dedução intervenientes.

$$SD' \xrightarrow{i'} SD' + SD'' \xleftarrow{i''} SD''$$

- A fibrilação de lógicas com restrições definidas pelos morfismos de assinaturas f' e f'', é dada pela construção do seguinte quadrado co-cartesiano em $\mathcal{SD}$:



5.2 Sincronização de lógicas

Em (Sernadas *et al.*, 1997b; Sernadas *et al.*, 1997c) são estudados os mecanismos de sincronização de lógicas, dos quais nos interessa destacar os mecanismos para a sincronização de sistemas de consequência ("consequence systems"). Assim como para o caso da fibrilação de lógicas, pretende-se construir uma categoria cujos objectos são sistemas de dedução, e na qual se possa internalizar os mecanismos de sincronização de lógicas.

O sistema que se obtém da sincronização de lógicas vai estar contido no produto cartesiano dos sistemas de partida. No sistema resultante da sincronização, um par $\langle \varphi_1, \varphi_2 \rangle$ indica que φ_1, numa das lógica deve ser interpretada na outra lógica como φ_2, ou vice-versa. No caso em que uma determinada fórmula não tem contrapartida, o par da sincronização é $\langle \varphi_1, \mathscr{V} \rangle$ ou $\langle \mathscr{V}, \varphi_2 \rangle$ conforme os casos, com $\mathscr{V}$ a denotar a fórmula "verdade". Quando se quer impor uma dada relação de sincronização entre as duas lógicas inclui-se um par $\langle \varphi_1, \varphi_2 \rangle$.

Temos então a seguinte definição:

Definição 5.3 (Sistemas de consequência) *Um sistema de consequência é um par $C = \langle L, \vdash \rangle$ onde $L \in \mathrm{Obj}(\mathsf{Set}_*)$ é um conjunto com um ponto seleccionado $\mathscr{V}$, e $\vdash\colon \mathscr{P}(L) \to \mathscr{P}(L)$ é uma aplicação tal que:*

- $\{\mathscr{V}\}^{\vdash} \subseteq \emptyset^{\vdash}$;
- $\Gamma \subseteq \Gamma^{\vdash}$ *(reflexividade);*
- $(\Gamma^{\vdash})^{\vdash} \subseteq \Gamma^{\vdash}$ *(idempotência);*
- *se* $\Gamma \subseteq \Psi$ *então* $\Gamma^{\vdash} \subseteq \Psi^{\vdash}$ *(monotonia).*

(Sernadas et al.*, 1997b).*

Com L a linguagem lógica, $\vdash$ o operador de consequência, e Set_* a categoria dos conjuntos pontuados (conjuntos não vazios com ponto seleccionado), e das funções entre conjuntos que preservam os pontos seleccionados ("pointed sets category") (MacLane, 1971).

Definindo os morfismos entre sistemas de consequência de forma apropriada os autores constroem a categoria $\mathcal{CSY}$, a categoria dos sistemas de consequência e dos morfismos entre sistemas de consequência.

5.2.1 Sincronização nas fórmulas

A definição do sistema de consequência resultante da sincronização nas fórmulas de outros sistemas de consequência, é dada pela definição dos pares $\langle \varphi_1, \varphi_2 \rangle$, temos então:

Proposição 5.2 (Sincronização nas fórmulas) *Sejam* $C_1 = \langle L_1, \vdash_1 \rangle$ *e* $C_2 = \langle L_2, \vdash_2 \rangle$ *sistemas de consequência. Sejam* $\alpha_1 : L_1 \to L_1 \times L_2$ *e* $\alpha_2 : L_2 \to L_1 \times L_2$ *morfismos em* Set_* *tais que,* $p_1(\alpha_1(\varphi_1)) = \varphi_1$ *e* $p_2(\alpha_2(\varphi_2)) = \varphi_2$. *Então a sincronização, nas fórmulas, de* C_1 *e* C_2, *por* α_1 *e* α_2, *é o sistema de consequência.*

$$C_1 \ {}^{\alpha_1}\!\bullet^{\alpha 2} \ C_2 = \langle L, \vdash \rangle$$

onde:

- $L = \alpha_1(L_1) \cup \alpha_2(L_2)$;
- $\Gamma^{\vdash} = \mathrm{lfp}(\Delta, \Gamma)$, *onde* $\Delta(\Gamma) = \Gamma \cup \alpha_1(p_1(\Gamma)^{\vdash_1}) \cup \alpha_2(p_2(\Gamma)^{\vdash_2})$.

(Sernadas et al., *1997b).*

Com $\mathrm{lfp}(\Delta, \Gamma)$, o menor ponto fixo para a aplicação de consequência Δ de um sistema de derivação num passo. Os sistemas de derivação num passo e os morfismos de sistemas de derivação num passo formam a categoria $\mathcal{DSY}$ (Sernadas *et al.*, 1997b).

Nesta construção α_1 e α_2 são designadas por aplicações de sincronização e a sua definição é crucial para a construção do sistema resultante da sincronização. No caso extremo em que não há sincronização entre os dois sistemas tem-se que $\alpha_1(\varphi_1) = \langle \varphi_1, \mathscr{V}_2 \rangle$ e $\alpha_2(\varphi_2) = \langle \mathscr{V}_1, \varphi_2 \rangle$ para quaisquer φ_1 e φ_2.

Os autores demonstram que as aplicações de sincronização são morfismos em $\mathcal{CSY}$ (Sernadas *et al.*, 1997b).

5.2.2 Sincronização de sistemas de consequência em $\mathcal{SD}$

A possibilidade de desenvolver o estudo sobre a sincronização de sistemas de consequência no âmbito da categoria $\mathcal{SD}$ parece-nos possível.

Os factos a demonstrar são, a nosso ver; se $\mathcal{CSY}$ e $\mathcal{DSY}$ podem ser vistas como sub-categorias de $\mathcal{SD}$, e se tal se verificar, se é possível internalizar em $\mathcal{SD}$ a construção de sistemas de consequência por sincronização nas fórmulas. Como questão que se torna importante verificar de seguida, temos o analisar dos pontos positivos (ou negativos) que tal aproximação leva ao estudo da sincronização dos sistemas de consequência.

5.3 Aplicação das construções categoriais

A possibilidade de estender um dado sistema computacional de demonstração automática, de forma a poder abarcar o cálculo de co-limites, dá-nos um outro campo de trabalho futuro.

Como já foi referido no capítulo anterior, julgamos ser possível efectuar essa extensão no sistema *2OBJ*, através da dotação do sistema *OBJ3* com os

construtores necessários ao cálculo de co-limites (Goguen & Winkler, 1988). Dado que os programas fontes em *AKCLisp/GCL* ("Austin-Kyoto Common Lisp/Gnu Common Lisp"), referentes ao sistema *OBJ3* estão disponíveis, o desenvolvimento da extensão referida passaria por; em primeiro lugar, adaptar o sistema *OBJ3* a vários sistemas computacionais, nomeadamente os sistemas *Linux*; em segundo lugar tratar-se-ia de desenvolver os programas necessários à extensão do sistema de módulos do *OBJ3* de forma a este incorporar os operadores necessários ao cálculo de co-limites.

Capítulo 6

Conclusões

A grande diversidade, assim como a crescente complexidade, dos sistemas de dedução que se querem considerar aquando do uso de sistemas computacionais para a demonstração automática, ou semi-automática de teoremas, levanta a questão da construção modular de sistemas de dedução.

Os actuais sistemas computacionais permitem a construção de sistemas de dedução, seja por por adição de componentes, a uma base fixa, seja por adição de componentes a uma base previamente implementada sobre um dado meta-sistema lógico. Esta última aproximações tenta resolver o problema da implementação de sistemas de dedução para um conjunto muito diversificado de sistemas lógicos, com estes sistemas computacionais é possível construir um sistema de dedução para uma lógica ainda não considerada, bastando para tal construir uma nova base sobre o meta-sistema lógico. A questão da complexidade dos sistemas de dedução a implementar fica só parcialmente resolvido dado que a construção de sistemas de dedução por adição de componentes não reduz muito significativamente o trabalho necessário para a construção de um dado sistema.

A definição de um sistema lógico como um triplo $\langle \mathcal{ASS}, \mathcal{SD}, \mathrm{Sd} \rangle$, tal como foi feito no presente trabalho, define-nos uma estrutura na qual se pode estudar o relacionamento entre as transformações nos sistemas de deduções dadas as transformações nas assinaturas das linguagens lógicas que lhes servem de base. Por outro lado ao demonstrar-se que $\mathcal{SD}$ é fechada para co-limites estabeleceu-se que em $\mathcal{SD}$ se podem construir sistemas de dedução recorrendo aos mecanismos próprios do cálculo de co-limites, providenciando deste modo os meios para a construção modular de sistemas de dedução, com mecanismos que vão muito além da simples adição de componentes.

A aplicação destes conceitos requere o uso de um sistema computacional que possua os mecanismos necessários para o cálculo de co-limites, o sistema *2OBJ* foi uma possibilidade estudada dado estar baseado na linguagem de programação *OBJ3*, sendo que esta possuí mecanismos de construção modular e parametrizável de programas bastante poderosos. No entanto

constatou-se que os mecanismos do *OBJ3* ainda não são suficientes para o fim em vista, a extensão do *OBJ3* com vista à inclusão das capacidades necessárias para o cálculo de co-limites afigura-se-nos possível, sendo um campo de trabalho futuro.

A construção de sistemas de dedução por combinação de diferentes lógicas é outra área que a nosso entender pode ser considerada tendo por base a presente abordagem. Neste contexto foram analisados os mecanismos de fibrilação de lógicas e de sincronização de lógicas, em ambos os casos afigura-se-nos possível incorporar o estudo desses mecanismos no âmbito dos sistemas lógicos.

Todas estas questões abrem áreas de trabalho futuro para as quais a definição de sistema lógico, tal como foi feito no presente trabalho, dá um contributo positivo.

Apêndice A

Exemplos

Nesta apêndice apresentam-se os exemplos que, por várias razões, não foram incluídos no corpo principal do presente texto. Além dos exemplos explicitamente referidos no restante texto inclue-se também, um exemplo de demonstração da correcção parcial, de um programa para a determinação do maior número natural contido na raiz quadrada de um dado número inteiro.

A.1 O Algoritmo de Divisão em *2OBJ*

A demonstração da correcção parcial do programa que implementa o algoritmo de divisão, desenvolve-se no sistema de dedução $LPCG_{LK}$, extendido com os axiomas necessários para o completar da demonstração.

Divisao.obj

```
obj DIVISAO is
 pr LPCG .

op _ <= _ : Expr Expr -> Sent [prec 51] .
op _ = _ : Expr Expr -> Sent [prec 51] .

ops x y r q divisao : -> Var-Elems .

let Prova = ( * |-
prg divisao vars(x ;; y ;; r ;; q) consts (*nil*) (*nil*)
{(x = ((y * 0) + x))}
r := x ;
{(x = (y * 0 + r))}
q := 0 ;
{(x = (y * q + r))}
faz << y <= r ;
```

```
       r := (r - y) ;
       {(x = (y * q + y + r))}
       q := (q + 1)
>> fimfaz
{(x = ((y * q) + r) ^ (~ (y <= r)))}
fimprg) .

vars z1 z2 : Var-Elems .
[axiom,div1] eq z1 + (- z1) = 0 .
[axiom,div2] eq - z1 + z1 = 0 .
[axiom,div3] eq z1 + (z1 * z2) = z1 * (z2 + 1) .

endo
```

O mecanismo de re-escrita do *OBJ3* vai ser usado na demonstração, para permitir a redução de expressões aritméticas à sua forma normal. Sem o recurso ao *OBJ3*, e implicitamente aos seus tipos de dados pré-definidos, ter-se-ia de implementar um sistema de dedução próprio para os tipos de dados que o programa usa, neste caso os inteiros.

A demonstração vai-se desenvolver de uma forma quase automática. A falta de um mecanismo de unificação para sequências dedutivas no *2OBJ*, impede a automatização das regras do consequente, e como tal um maior grau de automatização da demonstração.

$$\dfrac{\dfrac{\dfrac{\dfrac{-}{1.1.1.1}\ \text{ProvaProp}}{1.1.1}\ \text{reduzf('div3)} \quad \dfrac{-}{1.1.2}\ \text{atrD}}{1.1}\ \text{conseD}(I) \quad \dfrac{\dfrac{\dfrac{-}{1.2.1.1}\ \text{ProvaProp}}{1.2.1}\ \text{reduz} \quad \dfrac{-}{1.2.2}\ \text{atrD}}{1.2}\ \text{conseD}(II)}{\texttt{* |- prg divisao vars(x;;y;;r;;q) consts (*nil*) (*nil*) \{x=((y*0)+x)\} r:=x; \{x=(y*0+r)\} q:=0; \{x=(y*q+r)\} faz << y<=r ; r:=(r-y); \{x=(y*q+y+r)\} q:=(q+1) >> fimfaz \{(x=((y*q)+r) ^ (~ (y<=r)))\} fimprg}}\ \text{Lpcg\_Tac}$$

com

I	`x = y * q + r + y - y`
II	`x = r+y * (q + 1)`
1.1	`* |- {x=y*q+r ^ y<=r} r:=(r-y); {x=y*q+y+r}`
1.1.1	`* |- x=y*q+r ^ y<=r -> x=y*q+r+y-y`
1.1.1.1	`* |- x=y*q+r ^ y<=r -> x=y*q+r`
1.1.2	`* |- {x=y*q+r+y-y} r:=(r-y); {x=y*q+y+r}`
1.2	`* |- {x=y*q+y+r} q:=(q+1) {x=y*q+r}`
1.2.1	`* |- x=y*q+y+r -> x=r+y*(q+1)`
1.2.1.1	`* |- x=r+y*(q+1) -> x=r+y*(q+1)`
1.2.2	`* |- {x=r+y*(q+1)} q:=(q+1) {x=y*q+r}`

A.2 O problema *RaizNatural*

Dado o algoritmo para a determinação do maior número natural contido na raiz de um dado número natural n (Alagič & Arbib, 1978), a demonstração da correcção parcial do programa que o implementa, desenvolve-se no sistema de dedução $LPCG_{LK}$, extendido com os lemas necessários para o completar da demonstração.

De seguida apresentam-se as implementações em *Isabelle* e *2OBJ*, do sistema de dedução, onde se desenvolveu a demonstração de correcção parcial do programa, assim como um exemplo de uma possível demonstração em cada um dos sistemas referidos.

A.2.1 O problema *RaizNatural* no *Isabelle*

No sistema *Isabelle* criaram-se dois ficheiros de nome `RaizNatural`, mas com extensão `.thy` e `.ML`, respectivamente. O primeiro diz respeito à definição do sistema de dedução aonde se vai desenvolver a demonstração, e que é uma extensão do sistema $LPCG_{LK}$. No segundo define-se a sequência dedutiva que define o programa cuja demonstração parcial se pretende obter, e define-se também um "pack", por junção dos lemas próprios do problema às regras seguras de $LPCG_{LK}$.

RaizNatural.thy

```
(* Algoritmo para o cálculo do maior número natural      *)
(* contido na raiz de um dado número natural n.          *)
(*                                                       *)
(* Fazer use_thy "LPCG"; e depois use_thy "RaizNatural"; *)

RaizNatural = LPCG +

types
 nat 0

arities
 nat :: elems

consts
(* Operadores auxiliares para a escrita de expressões *)
 "0" :: "nat" ("0")
 um  :: "nat" ("1")
 dois  :: "nat" ("2")
 "+" :: "[nat,nat] => nat" (infixl 65)
 "-" :: "[nat,nat] => nat" (infixl 65)
```

```
 "*" :: "[nat,nat] => nat" (infixl 70)
 le  :: "[nat,nat] => o" ("_ <= _" [0,0] 50)
 deq  :: "[nat,nat] => o" ("_ .=. _" [0,0] 50)
 sq  :: "nat => nat"  ("_ ^2" 80)

rules
 ax1 "$H, A .=. (B+1)^2, $I, A <= C, $J |- $E, (B+1)^2 <= C,$F"
 ax2 "$H, A .=. 2*B+1, $I |- $E, A+2 .=. 2*(B+1)+1 ,$F"
 ax3 "$H, A .=. B^2 , $I, C .=. 2*B+1 , $K |- $E, \
\ A+C .=. (B+1)^2 ,$F"
end
```

RaizNatural.ML

```
open RaizNatural;

val RaizNatural_pack = lpcg_pack add_safes [ax1,ax2,ax3];

goal RaizNatural.thy "|- \
\ {0^2 <= a & 1 .=. (0+1)^2 & 1 .=. 2*0+1} \
\ (x := 0);{x^2 <= a & 1 .=. (x+1)^2 & 1 .=. 2*x+1}\
\ (y := 1);{x^2 <= a & y .=. (x+1)^2 & 1.=. 2*x+1}\
\ (z := 1);{x^2 <= a & y .=. (x + 1)^2 & z .=. 2*x+1}\
\ faz < y <= a ,\
\        (x := x+1);{x^2 <= a & y .=. x^2 & z+2 .=. 2*x+1}\
\        (z := z+2);{x^2 <= a & y .=. x^2 & z .=. 2*x+1}\
\        (y := y+z) > : GCnil \
\ fimfaz;{(x^2 <= a & y .=. (x+1)^2 & z .=. 2*x+1) & \
\         (~ (y <= a))}\
\ (root := x)\
\ {(root^2 <= a & y .=. (root+1)^2 & z .=. 2*root+1) & \
\  (~ (y <= a))}";
```

Demonstração de *RaizNatural* no *Isabelle*

Usando os mecanismos próprios do *Isabelle* para a automatização da demonstração, obtem-se uma demonstração bastante compacta.

Temos então:

```
by (lpcg_tac lpcg_pack 1);
by (conseD_tac "(x+1)^2<=a & y.=.(x+1)^2 & z+2.=.2*(x+1)+1" 1);
by (conseD_tac "x^2<=a & y+z.=.(x+1)^2 & z.=.2*x+1" 3);
by (lpcg_tac RaizNatural_pack 1);

No subgoals!
```

A.2.2 O problema *RaizNatural* no *2OBJ*

No sistema *2OBJ* criou-se o ficheiro `RaizNatural.obj`. Este ficheiro contém o objecto que define o sistema de dedução, extensão do sistema $LPCG_{LK}$, define também a sequência de dedutiva que define o programa cuja demonstração parcial se pretende obter, assim como um conjunto de lemas que vão ser necessários para o completar da demonstração.

RaizNatural.obj

```
*** Algoritmo para o cálculo do maior número natural
*** contido na raiz de um dado número natural n.
***
obj RAIZNATURAL is

 pr LPCG .

*** Operadores auxiliares para a escrita de expressões
***
op _ <= _ : Expr Expr -> Sent [prec 51] .
op _ = _ : Expr Expr -> Sent [prec 51] .
op _ ^2 : Expr -> Expr .

*** Algoritmo.
***
ops x y z a raiz RaizNat : -> Var-Elems .

let Prova = ( * |-
prg RaizNat vars(x;;y;;z;;a;;raiz) consts (*nil*) (*nil*)
((0^2)<=a)^(1=((0+1)^2))^(1=(2*0+1))
x := 0 ;
((x^2)<=a)^(1=((x+1)^2))^(1=(2*x+1))
y := 1 ;
((x^2)<= a)^(y=((x+1)^2))^(1=(2*x+1))
z := 1 ;
((x^2)<=a)^(y=((x+1)^2))^(z=(2*x+1))
faz << y <= a ;
    x := (x+1) ;
    ((x^2)<=a)^(y=(x^2))^((z+2)=(2*x+1))
    z := (z+2) ;
    ((x^2)<=a)^(y=(x^2))^(z=(2*x+1))
    y := (y+z)>>
fimfaz ;
(((x^2)<=a)^(y=((x+1)^2))^(z=(2*x+1)))^(~ (y<=a))
raiz := x
```

```
(((raiz^2)<=a)^(y=((raiz+1)^2))^(z=(2*raiz+1)))^(~ (y<=a))
fimprg) .

*** Lemas.
***
ops lema1 lema2 lema3 : -> Lemma .
ops b c : -> Var-Elems .

eq lemma lema1 =
( * |- |A|a. |A|b. |A|c. ((a<=b)^(a=((c+1)^2)))->
                                        (((c+1)^2)<=b)) .

eq lemma lema2 =
( * |- |A|a. |A|b. (a=((2*b)+1))->
                                  ((a+2)=((2*(b+1))+1))) .

eq lemma lema3 =
( * |- |A|a. |A|b. |A|c. ((a=(b^2))^(c=((2*b)+1)))->
                                     ((a+c)=((b+1)^2))) .
endo
```

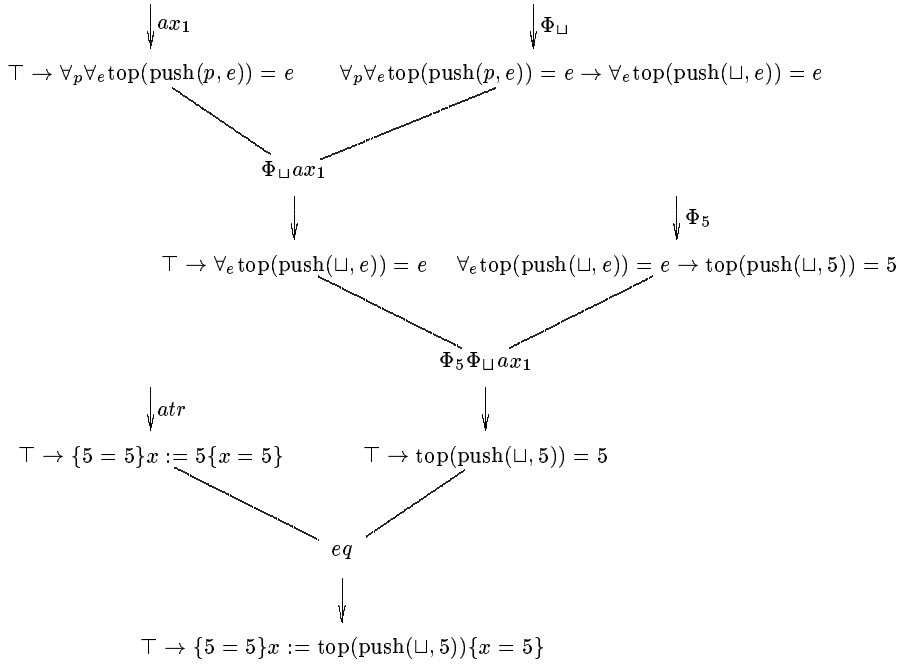
Demonstração de *RaizNatural* no *2OBJ*

A falta de um mecanismo de unificação para sequências dedutivas faz com que a demonstração no *2OBJ* seja um pouco longa. A introdução dos lemas e posterior instanciação própria para o problema em questão, tem de ser feito com a intervenção do utilizador, o resto da demonstração é feita de uma forma automática, recorrendo aos tacticais `LpcgTac`, e `ProvaProp`.

```
LpcgTac
1 conseD((((x+1)^2)<=a)^(y=((x+1)^2))^((z+2)=(2*(x+1)+1)))
1 1 ProvaProp
1 1 1 LEMMA(lema1)
1 1 1 1 Al(5,y) THEN Al(6,a) THEN Al(7,x) THEN ProvaProp
1 1 2 LEMMA(lema2)
1 1 2 1 Al(5,z) THEN Al(6,x) THEN ProvaProp
1 2 LpcgTac
2 conseD(((x^2)<=a)^((y+z)=((x+1)^2))^(z=(2*x+1)))
2 1 ProvaProp
2 1 1 LEMMA(lema3)
2 1 1 1 Al(4,y) THEN Al(5,x) THEN Al(6,z) THEN ProvaProp
2 2 atrD

Proof is complete
```

A.3 Demonstração em SD_5

Como referido no capítulo 2, apresenta-se de seguida a representação em forma de árvore da demonstração de $\top \rightarrow \{5 = 5\}x := \text{top}(\text{push}(\sqcup, 5))\{x = 5\}$



A.4 Demonstração em SD4

Como referido no capítulo 4, apresenta-se de seguida a representação gráfica, dada pelo *X2OBJ*, da árvore de demonstração da seguinte sequência dedutiva em SD4.

```
* |- {(push(Vazia,en) = push(Vazia,en))}
     pl := push(Vazia,en) ;
     {(pl = push(Vazia,en))}
     el := top(pl)
     {(pl = push(Vazia,en)) ^ (el = top(pl))}) .
```

basic
eq
p1 = push(Vazia,en) |- p1 = push(Vazia,en)
p1 = push(Vazia,en) |- top(p1) = top(p1)
AndR
p1 = push(Vazia,en) |- p1 = push(Vazia,en) ^ top(p1) = top(p1)
ImpR
* |- p1 = push(Vazia,en) -> p1 = push(Vazia,en) ^ top(p1) = top(p1)
wkrD
* |- {p1 = push(Vazia,en) ^ top(p1) = top(p1)}e1 := top(p1){p1 = push(Vazia,en) ^ e1 = top(p1)}
conseD(p1 = push(Vazia,en) ^ top(p1) = top(p1))
wkrD
* |- {push(Vazia,en) = push(Vazia,en)}p1 := push(Vazia,en){p1 = push(Vazia,en)}
* |- {p1 = push(Vazia,en)}e1 := top(p1){p1 = push(Vazia,en) ^ e1 = top(p1)}
compD
* |- {push(Vazia,en) = push(Vazia,en)}p1 := push(Vazia,en); {p1 = push(Vazia,en)}e1 := top(p1){p1 = push(Vazia,en) ^ e1 = top(p1)}

Apêndice B

Listagens

Neste apêndice apresentam-se as listagens dos vários sistemas referidos ao longo do presente texto.

Começa-se por apresentar o sistema *CPPO*, cálculo de predicados de primeira ordem, este sistema foi construído para substituir o sistema *FOPC*, que está pré-definido no *2OBJ*. Algumas das regras de inferência em *FOPC* necessitam de um ou mais argumentos, tal facto impede o atingir de um grau elevado de automatização das demonstrações, assim sendo algumas dessas regras de inferência foram re-escritas de forma a não necessitarem de nenhum argumento, e como tal permitirem um maior grau de automatização das demonstrações.

As tácticas e os tacticais pré-definidos no objecto `tacs.obj` foram modificados de forma a poderem usar as nova regras de inferência. Foi também acrescentado uma táctica que permite automatizar as demonstrações proposicionais.

Após o sistema *CPPO* apresenta-se o sistema $LPCG_{LK}$, sistema de dedução para uma linguagem procedimental com comandos guardados, nas duas implementações propostas, ficheiros `LPCG.thy` e `LPCG.ML` para a implementação do sistema $LPCG_{LK}$ no *Isabelle*, e o ficheiro `LPCG.obj`, referente à implementação dos sistema $LPCG_{LK}$ no *2OBJ*.

B.1 CPPO, Cálculo de Predicados de Primeira Ordem

O sistema *CPPO* é baseado no sistema pré-definido *FOPC*, diferindo deste pelo facto de que a aplicação de certas regras de inferência não necessitarem de interacção com o utilizar.

cppo.obj

```
*** Example logic definition module for 2OBJ
```

```
*** L\'ogica Proposicional - Modifica\c{c}\~ao feita a partir
*** do ficheiro fopc.obj
*** Pedro Quaresma de Almeida

*** Module defining frame for CPPO
*** First import modules for syntax of simple sequents: SEQUENTS

 *** The sorts and operators for encoded sentences and sentence
 *** fragments are introduced in a seperate module because
 *** it is going to be used in several places other than the
 *** module defining the logic's inference rules.  E.g. modules
 *** where user-defined
 *** predicates are introduced.  This is a bit more complicated
 *** than as described in the manual because we're using a
 *** seperate module for the syntax of sequents to keep things
 *** compact.

obj CPPO-SENTENCES is

sorts Sent Term Var .
sorts Prop .
subsort Prop < Sent .
subsorts Var < Term .
 *** Define the operators for the chosen logic as OBJ3
 *** operators.  (Neat eh!)

op |A| _ . _   : Var Sent  -> Sent [ prec 28 ] .
op |E| _ . _   : Var Sent  -> Sent [ prec 28 ] .
op _ ^ _  : Sent Sent -> Sent [ prec 20 gather ( e & ) ] .
op _ v _  : Sent Sent -> Sent [ prec 22 gather ( e & ) ] .
op _ -> _ : Sent Sent -> Sent [ prec 25 gather ( e & ) ] .
op ~ _    : Sent -> Sent [ prec 18 ] .
op _ = _  : Term Term -> Sent [ prec 17 ] .
op False  : -> Prop .
op True   : -> Prop .

endo

obj CPPO-LANGUAGE is
pr SEQUENTS-LANGUAGE[ CPPO-SENTENCES ] .

 *** The logic language definition is always an extension of
 *** ``GOAL'' - the logic mechanisation expects a sort of
 *** Goals and axioms.  In this case these are going
 *** to be Seq's (as defined in the SEQUENT-LANGUAGE module).

pr GOAL .
subsorts Seq < Goal .
```

```
pr NAT .
 *** make all natural numbers and Bool's terms
subsorts Bool Nat < Term .
 *** Allow for variables of type Nat
sort Var-Nat .
subsort Var-Nat < Var Nat .

endo

obj ENCODED-CPPO is

pr CPPO-LANGUAGE .
 *** DO NOT WORRY ABOUT THE DETAILS OF THIS SECTION UNLESS YOU
 *** HAVE GOT TO THE SECTION ON DEFINING FRAMES IN THE TUTORIAL
 *** INTRODUCTION SIMPLY SKIP ONTO THE BIT GIVING EQUATIONS FOR
 *** INFERENCE RULES...

pr OTERMX . *** This imports the meta-programming interface

 *** The binding characteristics of the operators are defined by
 *** providing equations for the bound operator.  The
 *** meta-programming interface uses these equations to correctly
 *** implement substitution etc.

vars V V' : Var .
vars X Y Z : Sent .
eq variable( V ) = true .
eq bound( ( |A| V . X ) ) = V .
eq bound( ( |E| V . X ) ) = V .

 *** The tedious stuff for manipulating simple sequents is kept in
 *** a seperate module as it is likely that we might want to
 *** re-use it for other similar logics.

pr SEQUENTS[ CPPO-SENTENCES ] .

 *** Logic definition is always a specialisation of the
 *** theory ``RULE-SET''.  The logic mechanisation expects
 *** the inference rules of the logic to be defined as
 *** equations for the rule application operator.

pr RULE-SET .

 *** Define the inference rules of the logic as OBJ operators
 *** with coarity Rule.   When linked into the logic mechanisation
 *** these wil provide us with the base terms of the tactic algebra
 *** that we will use to define proof procedures. We also
 *** define the condition under which formulas are well-formed
```

```
vars A : Universal .
vars L : Universal .
vars N : NzInt .
vars I : Int .
vars S : Seq .
vars T AT BT : Term .
vars IL : IdList .

eq wff( S ) = ( freevars( S ) == *nil* ) .

op AndR  : -> Rule .
op OrR1  : -> Rule .
op OrR2  : -> Rule .
op OrR   : -> Rule .
op ImpR  : -> Rule .
op NotR  : -> Rule .
op Er    : Term -> Rule .
op Ar    : -> Rule .

op andL  : NzInt -> Rule .
op orL   : NzInt -> Rule .
op impL  : NzInt -> Rule .
op notL  : NzInt -> Rule .
op AndL  : -> Rule .
op OrL   : -> Rule .
op ImpL  : -> Rule .
op NotL  : -> Rule .
op Al    : NzInt Term -> Rule .
op El    : NzInt -> Rule .

op basic  : -> Rule .
op FalseL : -> Rule .

op TrueR  : -> Rule .
op eq     : -> Rule .
op thin   : -> Rule .
op falseL : NzInt -> Rule .
op hyp    : NzInt -> Rule .
op cut    : Sent -> Rule .

op red : Id Selection   -> Rule .
op red : Id Selection IdList -> Rule .
op app : NzInt Direction Search Substitution Selection -> Rule .
op app : Id Id Direction Substitution Search Selection -> Rule .

op ind    : Term -> Rule .

vars H H1 H2 : SentenceList .
vars M R : Id .
```

```
vars PL : Selection .
vars D : Direction .
vars E : Search .
vars Subst : Substitution .

op indsubgoals : Universal Seq Universal -> GoalList .
op subgoals : SentenceList Universal -> GoalList .
op eqnhyps  : Universal -> EqnList .

***
*** operadores auxiliares
***
 op retira : SentenceList Sent -> SentenceList .
 eq retira(*,X) = X .
 eq retira(Y ; H,X) = if Y == X then H else Y ; retira(H,X) fi .

 op acha : Sent SentenceList -> Sent .
 eq acha(Z,X ; H) = if matches( Z , X ) then X else acha(Z,H) fi .

 op existe : Sent SentenceList -> Bool .
 eq existe(Z,*) = false .
 eq existe(Z,X ; H) = if matches( Z , X ) then true
                      else existe(Z,H) fi .

 op simples : Sent -> Bool .
 eq simples(X) = Tsubterms(X) == *nil* .

 *** Define the inference rules - i.e.
 *** their form, the subgoals they produce when applied to a
 *** suitable goal - as equations for the rule application operator
 *** _ _ : Rule Goal -> GoalList .  It is these equations that
 *** will be used by the logic mechanisation module to realise
 *** an abstract data-type of proofs in the defined logic.

 eq AndR  ( H |- X ^ Y ) = ( H |- X ) , ( H |- Y ) .
 eq OrR1  ( H |- X v Y ) = ( H |- X ) .
 eq OrR2  ( H |- X v Y ) = ( H |- Y ) .
 eq OrR   ( H |- X v Y ) = if existe(X,H) then ( H |- X ) else
              (if existe(Y,H) then ( H |- Y ) else  ( H |- X ) fi)
            fi .

 eq ImpR  ( H |- X -> Y ) = ( H ; X |- Y ) .
 eq NotR  ( H |- ~ X  )   = ( H ; X |- False ) .

 eq Er( T ) ( H |- |E| V . Y ) = ( H |- ( Y o T / V ) ) .

ceq Ar      ( H |- |A| V . Y ) = ( H |- Y )
               if not ( V in freevars( H ) ) .
ceq Ar      ( H |- |A| V . Y ) = ( H |- new( Y, V ) )
```

```
               if V in freevars( H ) .

ceq andL( N ) ( H |- X ) = ( retira(H,hyp(N, H)) ;
               [ Tsubterms(hyp(N, H))] |- X )
    if matches( Z ^ Y, hyp(N, H) )  .

ceq AndL ( H |- X ) = ( retira(H,acha(Z ^ Y,H)) ;
               [ Tsubterms(acha(Z ^ Y,H))] |- X )
   if existe(Z ^ Y,H) .

ceq orL( N )    ( H |- X ) =
    ( retira(H,hyp(N, H)) ; hyp(N,H) : 1 |- X ),
    ( retira(H,hyp(N, H))  ; hyp(N,H) : 2 |- X )
      if matches( Z v Y, hyp(N, H) )  .

ceq OrL    ( H |- X ) =
    ( retira(H,acha(Z v Y,H)) ; acha(Z v Y,H) : 1 |- X ),
    ( retira(H,acha(Z v Y,H)) ; acha(Z v Y,H) : 2 |- X )
      if existe(Z v Y,H) .

ceq impL( N )   ( H |- X ) =
    ( retira(H,hyp(N,H)) |- hyp(N,H) : 1 ) ,
    ( retira(H,hyp(N,H)) ; hyp(N,H) : 2 |- X )
      if matches( Z -> Y, hyp(N, H) ) .

ceq ImpL ( H |- X ) =
    ( retira(H,acha(Z -> Y,H)) |- acha(Z -> Y,H) : 1 ) ,
    ( retira(H,acha(Z -> Y,H)) ; acha(Z -> Y,H) : 2 |- X )
      if existe(Z -> Y,H) .

ceq notL( N )  ( H |- X ) =
    ( retira(H,hyp(N, H)) |- hyp(N,H) : 1 )
      if matches( ~ Y, hyp(N, H) )  .

ceq NotL ( H |- X ) =
    ( retira(H,acha(~ Y,H)) |- acha(~ Y,H) : 1 )
      if existe(~ Y,H) .

ceq Al( N, T ) ( H |- X )  =
        ( H ; ( hyp(N,H) : 2 o T / ( hyp(N,H) : 1) ) |- X )
        if matches( |A| V . Y, hyp(N, H) ) .


ceq El(N) ( H |- X  )  =
          ( if not ( ( hyp(N, H) : 1 ) in freevars( H |- X )  )
          then ( H ; hyp(N, H) : 2 ) |- X
          else ( H ; new( hyp(N,H) : 2 , hyp(N,H) : 1) ) |- X
            fi )
        if matches( |E| V . Y, hyp(N, H) ) .
```

```
*** Axiomas do c\'alculo de sequentes ( H,P,G |- P) .

 eq basic ( Y ; H |- X) = if X == hyp(1,Y) then []
                          else basic (H |- X) fi .

 eq TrueR       ( H |- True )  = [] .
 eq thin        ( H |- X ) = ( * |- X ) .
ceq eq          ( H |- (AT = BT) ) = [] if AT == BT .
ceq falseL( N ) ( H |- X )  = []
    if hyp(N, H ) == False .

ceq FalseL ( H |- X ) = []
      if existe(False,H) .

ceq hyp( N )  ( H |- X )  = [] if X == hyp(N,H) .
 eq cut( Y )  ( H |- X )  =
        if  freevars( Y ) diff freevars( H |- X ) == *nil*
        then ( H |- Y) , ( H ; Y |- X )
        else ! 'BadCutTerm
        fi .

 *** Rule for using reduction in named modules
 *** PL is a position list. I.e. ``top'' or a list of numbers
 *** indicating a position in the term to be reduced

 eq red( M, PL ) ( H |- X ) = ( H |- reduce( M, PL, X ) ) .
 eq red( M, PL, IL ) ( H |- X ) =
        ( H |-  extendred( M, none, PL,
                           eqnhyps( hyps( IL, H )), X  )) .
 *** Should check hypotheses being used as equations!

 *** Rule for applying a named equation from a named module
 *** or equation from hypothesis list
 eq app( M, R, D, Subst, E, PL ) ( H |- X )   =
   subgoals( H, namedred( M, R, PL, D, E, Subst, false, X ) ) .

ceq app( N, fwd, E, Subst, PL ) ( H |- X ) =
  ( H |- givenred( hyp(N, H) : 1, hyp(N, H ) : 2, PL, E, Subst, X ) )
  if matches( AT = BT, hyp(N, H )) .
ceq app( N, bwd, E, Subst, PL ) ( H |- X ) =
  ( H |- givenred( hyp(N, H ) : 2, hyp(N, H) : 1, PL, E, Subst, X ) )
  if matches( AT = BT, hyp(N, H )) .

*** Rule for induction over data-types
*** nothing clever - purely intended for familiarisation
*** purposes. You have to provide an equation giving constructors
*** for each sort. Constructors for Nat are built in.
```

```
vars V : Var .

ceq ind( V ) ( H |- X ) =
        indsubgoals( V, H |- X, constructors( V ) )
    if  V in freevars(X) .

pr CONSTRUCTORS .
vars Nn : Nat .
 eq constr( Nn ) = ( 1 + Nn ) ;; 0 .

 *** Definitions for auxiliary functions

ops objcond objcondl : Bool -> Sent [ strat ( 0 ) ] .
op  defobjcond       : Sent -> Sent .
op  isobjcondl       : Sent -> Bool .

var C1 C2 : Bool .
 eq subgoals( H, *nil* ) = [] .
 eq subgoals( H, ( X ;; L ) ) = ( H |- X ) , subgoals( H, L ) .
 eq subgoals( H, ( C1 ;; L ) ) =
      ( H |- objcond( C1 ) ), subgoals( H, L ) .

 eq objcond( C1 ) = defobjcond( objcondl( C1 ) ) .
 eq objcondl( AT == BT ) = ( AT = BT ) .
 eq objcondl( AT =/= BT ) = ( ~ AT = BT ) .
 eq objcondl( C1 and C2 ) = objcond( C1 ) ^ objcond( C2 ) .
 eq objcondl( C1 or C2 ) = objcond( C1 ) v objcond( C2 ) .
 eq objcondl( not C1 ) = ~ objcond( C1 ) .

 eq defobjcond( objcondl( C1 ) ) = ( C1 = true ) .
ceq defobjcond( X ) = X if isobjcondl( X ) =/= true .
 eq isobjcondl( objcondl( C1 ) ) = true .

 eq indsubgoals( V, S, *nil* ) = [] .
 eq indsubgoals( V, S, ( T ;; L) )   = indseq( S, V, T ) ,
                   indsubgoals( V, S, L ) .

 op eqnhyp : Sent Universal -> EqnList .
 eq eqnhyps( X ;; L ) = eqnhyp( X, *nil* ) , eqnhyps( L ) .
 eq eqnhyps( *nil* ) = noeqns .

var Frees : Universal .
 eq eqnhyp( AT = BT, Frees ) = ( AT =' BT [ Frees | 'hyp ] ) .

endo

*** Introduces a tactic for applying lemma's and axiom's.  This
*** is very useful as applying these directly is rather fiddly...
```

```
obj CPPO is
pr 2OBJ[ ENCODED-CPPO ] .

vars P <: Proof .
vars AN : Axiom .
vars LN : Lemma .
vars S : Seq .
vars Se : Sent .
op LEMMA : ProofTactic -> Tactic .
eq LEMMA(LN) S = cut( concl( lemma LN ) )
              THENL ( ( thin THEN LN ) , idtac ) .
eq LEMMA(AN) S = cut( concl( axiom AN ) )
              THENL ( ( thin THEN AN ) , idtac ) .
endo
```

B.2 Tacticais para CPPO

Conjunto de tacticas implementadas para aproveitar a capacidade acrescida do sistema *CPPO*, feito por modificação das tácticas definidas no objecto `TACS` pré-definido no *2OBJ*. Além das modificações referidas foi acrescentado uma táctica para a automatização das demonstrações proposicionais.

tacs.obj

```
*** Introduce mechanised logic - Modifica\c{c}\~ao feita a partir do
*** ficheiro tacs.obj pr\'e-definido no sistema 2OBJ.
*** Pedro Quaresma de Almeida, Universidade de Coimbra

obj TACTICAS is

pr CPPO .

vars N : Nat .
vars P : Proof .
vars S : Seq .

vars PT : ProofTactic .
op REPEAT : Nat ProofTactic -> Tactic .
eq REPEAT( N, PT ) S = if N == 0 then idtac
                       else PT THEN REPEAT( p( N ), PT ) fi .

op TRY : ProofTactic -> Tactic .
eq TRY( PT ) S = PT ELSE idtac .

op EXHAUST : ProofTactic -> Tactic .
eq EXHAUST( PT ) S = (PT THEN EXHAUST(PT)) ELSE idtac .

op reduce : -> Tactic .
```

```
eq reduce S = red( '% , top ) .

op apply _ _ _ : Id Search Selection -> Tactic .
op apply _ _ _ : NzInt Search Selection -> Tactic .
vars Rule : Id .
vars Hyp : NzInt .
vars Sel  : Selection .
vars Exh  : Search .
eq ( apply Rule Exh Sel ) S =
       app( '% , Rule, fwd, nosub , Exh, Sel ) .
eq ( apply Hyp Exh Sel ) S =
       app( Hyp, fwd, Exh, nosub, Sel ) .

var Sl : SubTmSel .
vars X Y Z : Sent .
var V : Var .
var G : Sent .
var H : SentenceList .
var Hyp : NzInt .
vars A B : Term .
op concltac : Selection -> Tactic .
op hyptac   : Sent NzInt -> Tactic .

eq clicktac( 2 )    (H |- G) = concltac( top ) .
eq clicktac( 2 Sl ) (H |- G) = concltac( Sl ) .
eq clicktac( 1 )   (H |- G ) = hyptac( H,  1 ) .
eq clicktac( 1 Sl ) (H |- G) = hyptac( Tselect( Sl, H ),
                                       Tocseltoacsel( Sl, H ) ) .

eq concltac( Sel ) (H |- X ^ Y ) = AndR .
eq concltac( Sel ) (H |- |A| V . X ) = Ar .
eq concltac( Sel ) (H |- X -> Y ) = ImpR .
eq concltac( Sel ) (H |- A = A) = eq .
ceq concltac( Sel ) ( H |- X ) = ind(Tselect( Sel, X ))
    if variable(Tselect(Sel, X )) .
eq concltac( Sel ) DEFAULT S = red( '% , Sel ) .

eq hyptac( X ^ Y, Hyp ) S = andL( Hyp ) .
eq hyptac( |E| V . X, Hyp ) S = El( Hyp ) .
eq hyptac( X v Y , Hyp ) S = orL( Hyp ) .
ceq hyptac( G, Hyp ) DEFAULT S = red( '%, top, Hyp )
    if eqnhyp( G, *nil* ) =/= *nil* .

var Go : Goal .

op ProvaProp : -> Tactic .

ceq ProvaProp (H |- X) =
        basic ELSE idtac
```

```
        if existe(X,H) .
eq ProvaProp ( H |- True ) =
        TrueR ELSE idtac .
eq ProvaProp ( H |- X = X) =
        eq ELSE idtac .
ceq ProvaProp ( H |- X) =
        FalseL ELSE idtac
        if existe(False,H) .
ceq ProvaProp (H |- X) =
        (AndR THEN ProvaProp) ELSE idtac
        if  matches(  Z ^ Y , X ) .
ceq ProvaProp (H |- X) =
        (ImpR THEN ProvaProp) ELSE idtac
        if  matches(  Z -> Y , X ) .
ceq ProvaProp (H |- X) =
        (NotR THEN ProvaProp) ELSE idtac
        if  matches(  ~ Y , X ) .
ceq ProvaProp (H |- X) =
        (AndL THEN ProvaProp) ELSE idtac
        if existe(Z ^ Y,H) .
ceq ProvaProp (H |- X) =
        (OrL THEN ProvaProp) ELSE idtac
        if existe(Z v Y,H) .
ceq ProvaProp (H |- X) =
        (ImpL THEN ProvaProp) ELSE idtac
        if existe(Z -> Y,H) .
ceq ProvaProp (H |- X) =
        (NotL THEN ProvaProp) ELSE idtac
        if existe(~ Y,H) and simples(X) .
eq ProvaProp DEFAULT Go =
        idtac .

op reduz : -> Rule .
eq reduz ( H |- G) = ( H |- reduce('%,top,G)) .

var Ids : IdList .
op reduzf : IdList -> Rule .
eq reduzf(Ids) ( H |- G) = ( H |- filtdred('%,Ids,top,G)) .

var Eqs : EqnList .
op reduze : IdList EqnList -> Rule .
eq reduze(Ids,Eqs) (H |- G) =
          ( H |- extendred('%,Ids,top,Eqs,G)) .

var Id : Id .
var Bb : Bool .
var Dir : Direction .
op reduzn : Id Direction Bool -> Rule .
eq reduzn(Id,Dir,Bb) (H |- G) =
```

```
          (H |- namedred('%,Id,top,Dir,within,nosub,Bb,G)) .

endo
```

B.3 $LPCG_{LK}$, Linguagem de Programação com Comandos Guardados

O sistema de dedução para uma linguagem procedimental com comandos guardados foi implementado em dois sistema de demonstração automática de teoremas, o sistema *Isabelle* e o sistema *2OBJ*. Apresenta-se em primeiro lugar a implementação no *Isabelle*, e de seguida a implementação no *2OBJ*.

B.3.1 O sistema $LPCG_{LK}$ em Isabelle

No sistema *Isabelle* a implementação do sistema $LPCG_{LK}$ passa pela construção de uma *teoria*, onde se especificam; as espécies, os operadores, os axiomas, e as regras de inferência do sistema que se quer construir.

A este ficheiro de extensão `.thy` acrescenta-se outro com o mesmo nome mas extensão `.ML`, neste último definem-se os mecanismos apropriados à automatização das demonstrações em $LPCG_{LK}$.

LPCG.thy

```
(*
  T\'{\i}tulo: LPCG.thy
  Autor: Pedro Quaresma de Almeida, Universidade de Coimbra
  Axiom\'atica de Hoare para uma linguagem procedimental com
  comandos guardados
*)
(* Para carregar este programa fazer: use_thy "LPCG"; open LPCG; *)
LPCG = LK +

classes
 elems < term
 inst  < term

types
 elems,inst   0
 list         1
 "*"          2 (infixl 6)

arities
 elems,inst   :: term
 list         :: (term)term
 "*"          :: (term,term)term

consts
```

```
(** LISTAS de Pares (Proposi\c{c}\~ao x Instru\c{c}\~ao) **)
 GCnil :: "(o*inst) list"
 ":" :: "[(o*inst) ,(o*inst) list] => (o*inst) list" (infixr 60)
 GCPair :: "[o,inst] => o*inst"("(1<_,/_>)" [0,0] 100)
(** OPERADORES AUXILIARES **)
 BB   ::  "(o*inst) list => o"
(** INSTRU\c{C}\~OES **)
 nula  :: "inst"
 atr   :: "['a::elems,'a] => inst" ("_ := _")
 comp  :: "[inst,o,inst] => inst" ("_;{_}_" [0,0,201] 200)
 if    :: "(o*inst) list => inst"("if _ fi")
 se    :: "(o*inst) list => inst"("se _ fimse")
 do    :: "(o*inst) list => inst" ("do _ od")
 faz   :: "(o*inst) list => inst" ("faz _ fimfaz")
 fundec  :: "['a::elems,'b::elems,'b,o,inst,o] => o" \
                      ("funcao _[_][_] {_}_{_} fimfuncao")
 funcha :: "['a::elems,'b::elems] => 'a" ("_[_]")
 prepos :: "[o,inst,o] => o" ("{_}_{_}"  [0,0,0] 100)

rules
(** Operadores AUXILIARES **)
 BBnilD "$H |- $E,$F ==>$H |- $E,BB(GCnil),$F"
 BBconsD "$H |- $E,B | BB(L),$F ==> \
\ $H |- $E,BB(<B,S>:L),$F"
 BBnilE "$H,$E |- $F ==> $H,BB(GCnil),$E |- $F"
 BBconsE "$H,B | BB(L),$E |- $F ==> \
\ $H,BB(<B,S>:L),$E |- $F"
(** INSTRU\c{C}\~OES **)
 conseD "[| $H |- $E,P --> R,$F ; $H |- $E,{R}S{Q},$F|] ==> \
\ $H |- $E,{P}S{Q},$F"
 consdD "[| $H |- $E,{P}S{R},$F ; $H |- $E,R --> Q,$F|] ==> \
\ $H |- $E,{P}S{Q},$F"
 nulaD "$H |- $E,{P}nula{P},$F"
 atrD "$H |- $E,{P(e)}x:=e{P(x)},$F"
 compD "[| $H |- $E,{P}S1{R},$F ; $H |- $E,{R}S2{Q},$F|] ==> \
\ $H |- $E,{P}S1;{R}S2{Q},$F"
 ifD "[|$H |- $E,(P & ~BB(<B,S>:L)) -->Q,$F ; \
\ $H |- $E,{P}se <B,S>:L fimse{Q},$F|] ==> \
\ $H |- $E,{P}if <B,S>:L fi{Q},$F"
 seOD "$H |- $E,{P}se GCnil fimse{Q},$F"
 seND "[|$H |- $E,{P & B}S{Q},$F ; $H |- $E,{P}se L fimse{Q},$F|] \
\ ==> $H |- $E,{P}se <B,S>:L fimse{Q},$F"
 faz1D "$H |- $E,{P & B}S{P},$F  ==> \
\ $H |- $E,{P}faz <B,S>:GCnil fimfaz{P & ~ B},$F"
(*
 A regra doD \'e n\~ao segura dado que n\~ao \'e feita a
 verifica\c{c}\~ao da condi\c{c}\~ao de aplicabilidade,
 i.e. Q == P ^ ~BB
*)
```

```
 fazD "[|$H |- $E,{P}do <B,S>:L od{Q},$F ; \
\ $H |- $E,(P & ~BB(<B,S>:L)) --> Q,$F |] ==> \
\ $H |- $E,{P}faz <B,S>:L fimfaz{Q},$F"
 doOD "$H |- $E,{P}do GCnil od{Q},$F"
 doND "[|$H |- $E,{P & B}S{P},$F ; \
\ $H |- $E,{P}do L od{Q},$F|] ==> \
\ $H |- $E,{P}do <B,S>:L od{Q},$F"
(*
Regra para funcoes: A verificacao que e' feita so assegura que a
variavel x e' livre para a
*)
funD "$H,funcao f[x][v]{P}S{Q(f)} fimfuncao,$I |- $E,{P}S{Q(f)},$F \
\ ==> $H,funcao f[x][v]{P}S{Q(f)} fimfuncao,$I |- $E,P --> Q(f[x]),$F"
end

ML
```

LPCG.ML

```
(*
  T\'{\i}tulo: LPCG.ML
  Autor: Pedro Quaresma de Almeida, Universidade de Coimbra
  Tacticas para aplicar as regras do consequente
*)

open LPCG;

signature LPCG_RESOLVE =
sig
 val conseD_tac: string  -> int -> tactic
 val consdD_tac: string  -> int -> tactic
 val lpcg_pack: pack
 val LPCG_pack: pack
 val lpcg_tac:  pack -> int -> tactic
end;

structure LPCG_Resolve :  LPCG_RESOLVE =
struct
 fun consdD_tac (sP: string) i =
     res_inst_tac [ ("R",sP) ] consdD i;
 fun conseD_tac (sP: string) i =
     res_inst_tac [ ("R",sP) ] conseD i;

(*
  Definicao de um conjunto de regras seguras
*)
 val lpcg_pack = prop_pack add_safes
      [nulaD,atrD,compD,ifD,seND,seOD,faz1D,
       BBconsD,BBnilD,BBconsE,BBnilE];
```

```
(*
  Acrescentar as regras inseguras (n\~ao respeitam a propriedade
  da sub-formula)
*)
 val LPCG_pack = lpcg_pack add_unsafes [fazD,doND,do0D,
                                        conseD,consdD];

(*
 Usa a resolu\c{c}\~ao aplicada \'as regras seguras repetidamente
 de seguida usa a resolu\c{c}\~ao aplicada \'as regra n\~ao segura
 e depois volta ao princ\'{\i}pio. Usei a regra "repeat_goal_tac"
 como base
*)

fun lpcg_tac (Pack(safes,unsafes)) =
 let val restac  =    RESOLVE_THEN safes
     and lastrestac = RESOLVE_THEN unsafes;
     fun gtac i = REPEAT(restac gtac i ORELSE lastrestac gtac i)
 in  gtac  end;

end;

open LPCG_Resolve;
```

B.3.2 O sistema $LPCG_{LK}$ em *2OBJ*

No sistema *2OBJ* a implementação do sistema $LPCG_{LK}$ passa pela construção de um objecto, onde se especificam; as espécies, os operadores, os axiomas, e as regras de inferência do sistema que se quer construir.

LPCG.obj

```
*** Pedro Quaresma de Almeida, Universidade de Coimbra
***
*** Depende dos sistemas CPPO (cppo.obj) e TACTICAS (tacs.obj)
***

obj LPCG-ELEMENTS is

 pr CPPO .

 sorts Elems Expr .
 sorts Var-Elems Var-Expr .
 subsorts Var-Elems Var-Expr < Var .
 subsorts Var-Elems < Var-Nat .
 subsort Var-Elems < Elems .
 subsort Var-Expr < Expr .
 subsort Elems < Expr .
 subsort Int Int Bool < Elems .
```

```
endo

obj ENCODED-LPCG is

 pr LPCG-ELEMENTS .
 pr TACTICAS .

***
*** Especies
***
 sorts Instruc Var-Instruc  .
 subsort Var-Instruc < Instruc .
 subsorts Elems Expr Instruc < Term .

***
*** Listas de Pares "Comandos Guardados" (Sentence,Instrucao)
***
 pr SIMPLELIST[2TUPLE[Sent , Instruc]] * (sort List to CGList) .

***
*** Listas de Pares (Instrucoes,Condicoes)
***
 pr SIMPLELIST[2TUPLE[Instruc , Sent]] * (sort List to ICList) .

***
*** Operadores Auxiliares
***
 op BB : CGList -> Sent .
 op dis : Universal Universal -> Bool .
 op sdups : Universal Universal -> Universal .
 op (|Alist| _ . _) : Universal Sent -> Sent .
 op livrepara : Universal Universal -> Bool .

 vars Z W A E : Universal .
 var S : Instruc .
 var B  : Sent .
 var BSlist : CGList .
 var P : Sent .

 eq BB( [] ) =  False .
 eq BB( (<< B ; S >>) ; BSlist) = if BSlist == []
                                  then B
                                  else B v ( BB(BSlist)) fi  .

 eq sdups(*nil*,W) = *nil* .
 eq sdups(Z,W) = if Umember(Uhd(Z),W)
                 then sdups(Utl(Z),W)
                 else Uhd(Z) ;; sdups(Utl(Z),Uhd(Z) ;; W)
```

```
               fi .

 eq dis(Z,W) = (Udiff(Z,W) == Z) and (Z == sdups(Z,*nil*)) .

*** O operador Alist fica sem equacoes associadas dado que
*** nao se pretende o seu desdobramento.
***
*** eq (|Alist| *nil* . P) = P .
*** eq (|Alist| Z . P) = (|A| Uhd(Z) . (|Alist| Utl(Z) . P) ) .

*** Mas pretende-se o desdobramento para aplicar ao
*** operador "freevars".
***
 eq livrepara( *nil* ,W) = true .
 eq livrepara(Z,W) = (Uhd(Z) in freevars(W)) and livrepara(Utl(Z),W) .

***
*** Operadores (Instrucoes da linguagem de comandos guardados)
***
 op nula : -> Instruc .
 op _ := _ : Elems Expr -> Instruc [prec 0] .
 op _;{_}_ : Instruc Sent Instruc -> Instruc [prec 19 gather (e e E)] .
 op alg_fimalg : ICList -> Instruc .
 op se_fimse : CGList -> Instruc .
 op faz_fimfaz : CGList -> Instruc .
*** Declaracao de procedimento
 op (subprg_(_):(_)vars(_){_}_{_}fimsubprg) :
    Elems Universal Universal Universal Sent Instruc Sent -> Sent .
*** Chamada de procedimento
 op (invoca_(_):(_)fiminvoca) :
    Elems Universal Universal  -> Instruc .
*** Declaracao de funcao
 op (funcao_(_)vars(_){_}_{_}fimfuncao) :
    Elems Universal Universal Sent Instruc Sent -> Prop .
*** Chamada de funcao
 op _[_] :
    Elems Universal -> Elems .
*** Programa
 op (prg_vars(_)consts(_)(_){_}_{_}fimprg) :
  Elems Universal Universal Universal Sent Instruc Sent -> Prop .

*** Operador "semantico"
 op {_}_{_} : Sent Instruc Sent -> Prop [prec 20] .

***
*** Regras de Inferencia
***
 vars X : Elems .
 vars Y : Expr .
```

```
vars Q R P0 P1 Pn B1 : Sent .
vars S1 S2 : Instruc .
vars H Plist : SentenceList .
var Slist : ICList .
var N : NzInt .
var nome : Elems .
vars U1 U2 : Universal .

ops consdD conseD : Sent -> Rule .
ops nulaD atrD seD seind fazD fazind compD funD prgD : -> Rule .
ops procD invD : NzInt -> Rule .
ops chamafunD : Universal NzInt -> Rule .

eq nulaD ( H |- {P}nula{P} ) = [] .

eq consdD(R) ( H |- {P}S{Q} ) =
            ( H |- ( {P}S{R} ),
            ( H |- ( R -> Q ) ) ) .

eq conseD(R) ( H |- {P}S{Q} ) =
            ( H |- ( P -> R ) ),
            ( H |- {R}S{Q} ) .

cq atrD ( H |- {Q}(X := Y){P} ) = []
        if Q == ( P o Y / X ) .

eq compD ( H |- {P}S1 ; {R}S2 {Q} ) =
 ( H |- {P}S1{R}),
 ( H |- {R}S2{Q}) .

eq seD ( H |- {P}se (<< B1 ; S1 >> ; BSlist) fimse{Q} ) =
       ( H |- ( ~ BB((<< B1 ; S1 >> ; BSlist)) ^ P) -> Q),
       ( seind ( H |- {P}se (<< B1 ; S1 >> ; BSlist) fimse{Q})).
eq seind ( H |- {P}se ([]) fimse{Q}) = [] .
eq seind ( H |- {P}se (<< B1 ; S1 >> ; BSlist) fimse{Q}) =
         ( H |- {(P ^ B1)}S1{Q}),
         ( seind ( H |- {P}se BSlist fimse{Q})).

ceq fazD ( H |- {P}faz BSlist fimfaz{Q}) =
         ( fazind ( H |- {P}faz BSlist fimfaz{Q}))
         if Q == (P ^ ~ BB(BSlist)) .
eq fazind ( H |- {P}faz ([]) fimfaz{Q}) = [] .
eq fazind ( H |- {P}faz (<< B1 ; S1 >> ; BSlist) fimfaz{Q}) =
          ( H |- {P ^ B1}S1{P}),
          ( fazind ( H |- {P}faz BSlist fimfaz{Q})) .

ceq procD(N) ( H |- {P}(invoca nome ( U1 ) : ( U2 ) fiminvoca){Q}) =
   ( H |- (|Alist| (hyp(N,H) : 4) . {P}(hyp(N,H) : 6){Q}))
   if dis(U1,U2)
```

```
    and hyp(N,H) == (subprg nome ( U1 ) : ( U2 )
                     vars (hyp(N,H) : 4)
                     {P} (hyp(N,H) : 6) {Q} fimsubprg) .

 ceq invD(N) ( H |- {P}(invoca nome ( U1) : ( U2 ) fiminvoca){Q}) =
  ( H |- {(hyp(N,H) : 5)}
         (invoca nome (hyp(N,H) : 2) : (hyp(N,H) : 3) fiminvoca)
         {(hyp(N,H) : 7)})
  if P == ((hyp(N,H) : 5) o (U1 ;; U2) /
           ((hyp(N,H) : 2) ;; (hyp(N,H) : 3)))
  and Q == ((hyp(N,H) : 7) o (U1 ;; U2) /
            ((hyp(N,H) : 2) ;; (hyp(N,H) : 3)))
  and dis(U1,U2)
  and dis((hyp(N,H) : 2),(hyp(N,H) : 3)) .

 vars V1 XS CS SUBS : Universal .

 ceq chamafunD(V1,N) ( H |-  (P -> Q)) = []
    if not(livrepara(V1,(hyp(N,H) : 3)))
    and P == (hyp(N,H) : 4) o V1 / (hyp(N,H) : 2)
    and Q == ((hyp(N,H) : 6) o (((hyp(N,H) : 1) [ V1 ]) ;; V1)
                             / ((hyp(N,H) : 1) ;; (hyp(N,H) : 2) )) .

 eq funD ( H |- funcao nome(XS)vars(V1){P}S{Q}fimfuncao ) =
         ( H |- {P}S{Q}) .

 eq prgD ( H |- prg nome vars(XS)consts(CS)(*nil*){P}S{Q}fimprg ) =
 ( H |- {P}S{Q}) .
 eq prgD ( H |- prg nome vars(XS)consts(CS)(SUBS){P}S{Q}fimprg ) =
 ( H ; SUBS |- {P}S{Q}),
         ( H |- SUBS ) .
***
*** Procedimentos Recursivos
***
 ops procrecD : NzInt NzInt -> Rule .
 ops N1 N2 : -> NzInt .

 ceq procrecD(N1,N2) ( H |-
{P}(invoca nome ( U1 ) : ( U2 ) fiminvoca){Q}) =
    ( H |- (|Alist| (hyp(N1,H) : 4) . {P}(hyp(N1,H) : 6){Q}))
    if dis(U1,U2)
    and hyp(N1,H) == (subprg nome ( U1 ) : ( U2 )
                     vars (hyp(N1,H) : 4)
                     {P} (hyp(N1,H) : 6) {Q} fimsubprg)
    and hyp(N2,H) == ({P}(invoca nome ( U1 ) : ( U2 )
                          fiminvoca){Q}) .
***
*** Especificacao do que se entende por variaveis ligadas
***
```

```
 var K1 K2 K3 : Universal .
 var K4 : Elems .
 var K5 : Instruc .
 var K6 K7 : Sent .
***
*** Foi necessario alterar a definicao original (K3 ;; K2 ...) pois
*** que no caso de K1, por exemplo, ser uma lista de variaveis
*** na definicao anterior esta contava como (so um) elemento .
***
 eq bound(subprg K4 ( K1 ) : ( K2 ) vars ( K3 ) { K6 } K5 { K7 }
          fimsubprg) =  Uappend(Uappend(Uappend(K1,K2),K3),K4) .

 eq bound( funcao K4 (K1) vars (K3) {K6} K5 {K7} fimfuncao )
     =  Uappend(Uappend(K1,K3),K4) .

 eq bound(prg K4 vars( K1 ) consts ( K2 ) ( K3 ) { K6 } K5 { K7 }
          fimprg) =  Uappend(Uappend(K1,K2),K4) .
***
*** Tacticais
***
 var G : Goal .
 var V : Var .

 op LpcgTac : -> Tactic .

 eq LpcgTac ( H |- {P}nula{Q}) =
        nulaD ELSE idtac .
 eq LpcgTac ( H |- {P}(X := Y){Q} ) =
        atrD ELSE idtac .
 eq LpcgTac ( H |- {P}se BSlist fimse{Q} ) =
        (seD THEN LpcgTac) ELSE idtac .
 eq LpcgTac ( H |- {P}faz BSlist fimfaz{Q} ) =
        (fazD THEN LpcgTac) ELSE idtac .
 eq LpcgTac ( H |- {P}S1 ; {R}S2{Q} ) =
        (compD THEN LpcgTac) ELSE idtac .
 eq LpcgTac ( H |- funcao nome(XS)vars(V1){P}S{Q}fimfuncao ) =
        (funD THEN LpcgTac) ELSE idtac .
 eq LpcgTac ( H |- prg nome vars(XS)consts(CS)(SUBS){P}S{Q}fimprg ) =
(prgD THEN LpcgTac) ELSE idtac .
 eq LpcgTac DEFAULT G =
idtac .
endo

obj LPCG is

 pr ENCODED-LPCG .

endo
```

Apêndice C

Lista de Sistemas Computacionais

Neste apêndice apresenta-se uma lista de sistemas de demonstração automática e semi-automática de teoremas. A lista foi construída tendo por base a informação contida em duas listas deste tipo já existentes (Kohlhase & Talcott, 1998; Schumann, 1994), assim como outra informação existente na literatura da especialidade. Na apresentação da informação foi escolhido um formato próximo do formato das referências bibliográficas, ou seja: nome do sistema, (muito) breve descrição, autor(es) ou pessoa(s) a contactar, referências.

Na descrição dos diferentes sistemas, tentou-se sistematizar a informação de forma a responder às questões que se põem quanto; ao tipo de sistema, qual é o seu mecanismo de demonstração, e para que tipos de sistemas lógicos está vocacionado. Para permitir uma fácil comparação dos diferentes sistemas, a descrição está estruturada da seguinte forma:

descrição ::= sistema | meta-sistema

meta-sistema ::= meta-sistema que permite construir sistemas ⟨*sistema*⟩ ⟨*particularidades*⟩.

sistema ::= ⟨*tipo de sistema*⟩ ⟨*método de demonstração*⟩ ⟨*tipo de sistema lógico*⟩ ⟨*particularidades*⟩.

tipo de sistema ::= interactivo | automático | misto | ϵ

método de demonstração ::= resolução | *tacticais* | *tableau* | regras de re-escrita | ϵ

tipo de sistema lógico ::= primeira ordem | ordem superior | ... | ϵ

particularidades ::= ... | ϵ

o símbolo ϵ denota a opção vazia.

C.1 Lista de Sistemas Computacionais

- 2OBJ, meta-sistema que permite construir sistemas interactivos cujo mecanismo de inferência são as tácticas e tacticais, para sistemas lógicos formalizados em termos de um cálculo de sequências dedutivas. Possui já implementado um sistema para a lógica de primeira ordem, Andrew Stevens e Keith Hobley, (Stevens & Hobley, 1993).

- ${}_3T^AP$, sistema do tipo *tableau* para as lógicas de primeira ordem multivalentes, Reiner Hähnle e Bernhard Beckert, (Kohlhase & Talcott, 1998; Schumann, 1994).

- ACL2, sistema misto para um sub-conjunto do *Common Lisp* aplicativo, J. Strother Moore e Matt Kaufmann, (Kohlhase & Talcott, 1998).

- AF2, sistema interactivo para a lógica de segunda ordem, Christophe Raffalli, (Kohlhase & Talcott, 1998).

- Bluegum, sistema do tipo *tableau* para a lógica de primeira ordem na forma clausal, K. Wallace, (Schumann, 1994).

- Cassandra, meta-sistema que permite a construção de sistemas do tipo *tableau* para várias linguagens lógicas, M. Grundy,(Schumann, 1994).

- CLAM, sistema automático tendo por base o sistema *Oyster*, Alan Bundy, (Kohlhase & Talcott, 1998).

- Concurrency Workbench, sistema automático de manipulação e análise de sistemas concorrentes, Perdita Stevens, (Kohlhase & Talcott, 1998).

- Coq, sistema orientado por objectivos e usando *tácticas* para o cálculo das construções indutivas, Gerard Huet, (Kohlhase & Talcott, 1998).

- Deep Thought, sistema do tipo *tableau* analítico com variáveis livres para a lógica de primeira ordem sem igualdade, S. Gerberding, (Kohlhase & Talcott, 1998; Schumann, 1994).

- ECLiPSe, meta-sistema que permite o desenvolvimento de sistemas de resolução de restrições, Micha Meier, (Kohlhase & Talcott, 1998).

- Enhanced Hierarchical Development Methodology, sistema baseado em procedimentos de decisão para cláusulas básicas (*ground decision procedures*), para a lógica de ordem superior, John Rushby e San Owre, (Kohlhase & Talcott, 1998).

- ELAN, um ambiente para o prototipar de combinações de sistemas computacionais diferentes, sejam eles sistemas de resolução de restrições, ou demonstradores de teoremas, ou linguagens de programação lógicas, Mirian Vittek, (Kohlhase & Talcott, 1998).

- Elf, meta-sistema que permite construir sistemas de dedução para diferentes linguagens lógicas, Frank Pfenning, (Harper *et al.*, 1993; Kohlhase & Talcott, 1998; Michaylov & Pfenning, 1991; Pfenning, 1991).

- EVES/NEVER, sistema misto usa regras de re-escrita para desenvolver demonstrações na lógica de primeira ordem sem tipos, Dan Craigen, (Kohlhase & Talcott, 1998).

- Expander, ambiente de teste e demonstração automática para a especificação algébrica de tipos de dados e programas lógicos e funcionais, Peter Padawitz, (Kohlhase & Talcott, 1998).

- Faust, sistema do tipo *tableau* para a lógica de primeira ordem, K. Schneider, T. Kropf, (Schneider *et al.*, 1992b; Schneider *et al.*, 1992a; Schneider & Kropf, 1992; Schneider *et al.*, 1993; Schumann, 1994).

- Forest Theorem Prover, sistema do tipo *tableau* para a lógica modal de acção de primeira ordem, J. Cunningham, (Schumann, 1994).

- ft, sistema para a lógica de primeira ordem intuicionista, Torkel Franzen, (Kohlhase & Talcott, 1998).

- Gandalf, sistema automático para a lógica clássica de primeira ordem,Tanel Tammet, (Kohlhase & Talcott, 1998).

- GETFOL, sistema para a lógica de primeira ordem, baseado no sistema *FOL*, Fausto Giunchiglia, (Kohlhase & Talcott, 1998).

- HimMLKreuz, sistema do tipo *tableau* para a lógica de primeira ordem sem igualdade, J. Goubault, (Schumann, 1994).

- HOL, ambiente interactivo para a demonstração de teoremas na lógica de ordem superior, John Harrison e Phillip J. Windley, (HOL88a, 1991; HOL88b, 1991; HOL88c, 1991b; HOL88c, 1991a; Kohlhase & Talcott, 1998).

- HOL-Z, sistema baseado no sistema Isabelle, sistema lógico HOL, para a demonstração de correcção de especificações em *Z*, Thomas Santen, Kolyang e Burkhart Wolff, (Kolyang *et al.*, 1996).

- ICLE, meta-sistema que permite construir sistemas interactivos para sistemas lógicos formalizados em termos de um cálculo de sequências

dedutivas. Possui já implementados vários sistemas lógicos, nomeadamente a lógica de primeira ordem clássica e intuicionista, Mark Dawson, (Dawson, 1991; Dawson, 1992; Kohlhase & Talcott, 1998).

- IMPS, sistema baseado numa versão não construtiva da versão simples da teoria dos tipos com funções parciais e sub-tipos, William M. Farmer, Joshua D. Guttman e F. Javier Thayer, (Kohlhase & Talcott, 1998).

- INKA, sistema para a lógica de primeira ordem com indução, Diter Hutter, (Kohlhase & Talcott, 1998).

- Isabelle, meta-sistema que permite construir sistemas interactivos cujo mecanismo de inferência são as tácticas e tacticais. Possui já implementado vários sistemas lógicos nomeadamente lógica de primeira ordem clássica e intuicionista, lógica de ordem superior, cálculo de sequências dedutivas LK, Lawrence C. Paulson, Tobias Nipkow, (Kalvala, 1994; Kohlhase & Talcott, 1998; Paulson, 1993a; Paulson, 1993c; Paulson, 1993b).

- iTAB, sistema baseado na biblioteca de programas *ILFA*, A. Wolf, (Schumann, 1994).

- Jape, meta-sistema interactivo de edição de demonstrações, Bernard Sufrin, (Kohlhase & Talcott, 1998).

- Linseq, sistema do tipo *tableau* para a lógica linear de primeira ordem, T. Tammet, (Schumann, 1994).

- KEIM, conjunto de módulos em *Common Lisp* com *CLOS* e que constituem uma "caixa de ferramentas electrónica" extensível para o desenvolvimento de sistemas de dedução, Dan Nesmith, (Huang *et al.*, 1994; Kohlhase & Talcott, 1998; Nesmith, 1993).

- KIV, sistema interactivo concebido para o raciocínio formal sobre programas imperativos, Andreas Wolpers, (Kohlhase & Talcott, 1998).

- Kripke prover, sistema para lógica LR ("relevant logic LR"), Gustav Meglicki, (Kohlhase & Talcott, 1998).

- LAMBDA, conjunto de programas para a verificação formal de sistemas electrónicos e programas, (Kohlhase & Talcott, 1998).

- LA, sistema interactivo para a lógica de primeira ordem, John Garland e John Guttag, (Kohlhase & Talcott, 1998).

- LeanTAP, demonstrador em *micro Prolog* para a lógica de primeira ordem, Bernard Becket e Joachim Possega, (Kohlhase & Talcott, 1998).

- LEGO, sistema interactivo baseado na teoria dos tipos, Randy Pollack, (Kohlhase & Talcott, 1998).
- Linseq, sistema do tipo *tableau* para a lógica linear de primeira ordem, T. Tammet, (Schumann, 1994).
- Logics Workbench, sistema interactivo e modular que permite trabalhar em vários sistemas lógicos proposicionais, nomeadamente a lógica clássica e intuicionista, as lógicas modais e multi-modais, as lógicas temporais, as lógicas não monotónicas, o lambda cálculo, e a lógica combinatória, Gerard Jaeger, Alain Heuerding e Stefan Schwendimann, (Heuerding *et al.*, 1996; Kohlhase & Talcott, 1998).
- LPTP, sistema interactivo para a verificação formal de programas em Prolog, Robert Staerk, (Kohlhase & Talcott, 1998).
- MARK2, sistema interactivo para a lógica equacional, Randall Holmes, (Kohlhase & Talcott, 1998).
- Merill, sistema de re-escrita para a lógica equacional com ordenamento das espécies, Brain Matheus, (Mattews, 1996; Kohlhase & Talcott, 1998).
- Meteor, sistema que compila cláusulas numa estrutura de dados que é depois interpretada por mecanismos de inferência em máquinas sequenciais ou paralelas, O. L. Astrachan, (Schumann, 1994).
- Metamath, meta-sistema que providência uma linguagem capaz de expressar teoremas e demonstrações, Norman Megill, (Kohlhase & Talcott, 1998).
- Mizar, sistema para a escrita e verificação de matemática formal baseado na teoria de conjuntos e na dedução natural, Piotr Rudnicki, (Kohlhase & Talcott, 1998).
- MuRal, sistema para o desenvolvimento de inferências sobre especificações em VDM, Brian Ritchie, (Kohlhase & Talcott, 1998).
- Neitz[1], sistema do tipo *tableau* para a lógica de primeira ordem, baseado no *Prolog Technology Theorem Prover*, W. Neitz, (Schumann, 1994).
- NQTHM, sistema para a lógica sem quantificadores, e para as funções recursivas sobre os inteiros ou sobre outras estruturas geradas finitamente, Robert S. Boyer e J. Strother Moore, (Kohlhase & Talcott, 1998).

[1]Nome do autor, o sistema não tem nome definido.

- NUPRL, sistema para a teoria dos tipos intuicionista baseado na teoria dos tipos de Martin-Löf, Robert Constable, (Kohlhase & Talcott, 1998).

- OBJ3, linguagem de programação baseada na lógica equacional com ordenamento das espécies, Joseph Goguen e José Meseguer, (Goguen, 1988; Goguen, 1990; Goguen *et al.*, 1992; Kohlhase & Talcott, 1998).

- Ophelders[2], sistema do tipo *tableau* para a lógica de primeira ordem com símbolos de função mas sem igualdade, W. Ophelders, (Schumann, 1994).

- OSHL, sistema automático baseado na estratégia de ordenação semântica de hiper-ligação ("ordered semantic hyper linking"), para a lógica de primeira ordem, David Plaisted, (Kohlhase & Talcott, 1998).

- Otter, sistema cujo mecanismo de inferência é o método de resolução para a lógica de primeira ordem, William McCune, (Kohlhase & Talcott, 1998).

- Oyster, implementação do sistema NUPRL, Alan Bundy, (Kohlhase & Talcott, 1998).

- PartabX, sistema do tipo *tableau* implementado em *Strand* de forma a puder ser executado em máquinas de arquitectura distribuída ou paralela, R. Johnson, (Schumann, 1994).

- Parthenon, sistema do tipo *tableau* para a lógica de primeira ordem e com uma implementação usando o paralelismo OR, E. M. Clarke, (Schumann, 1994).

- Pc-Nqthm, aperfeiçoamento do sistema Nqthm, Matt Kaufmann, (Kohlhase & Talcott, 1998).

- Pegasys, ferramenta para a concepção de sistemas, usa o método de inferência *tableau* para a verificação de restrições em especificações de sistemas (e.g. restrições de tipos na lógica de predicados), R. Riemenschneider, (Schumann, 1994).

- Porgi, gerador de demonstrações ou refutações para a lógica de primeira ordem intuicionista, Allen Stoughton, (Kohlhase & Talcott, 1998).

- ProCom, sistema baseado no *Prolog Technology Theorem Prover*, Gerd Neugebauer, (Kohlhase & Talcott, 1998).

[2]Nome do autor, o sistema não tem nome definido.

- ProofPower, ferramenta de especificação e demonstração baseada numa implementação de lógica de ordem superior, com suporte para especificações e demonstrações em Z, (Kohlhase & Talcott, 1998).
- PROTEIN, sistema automático para a lógica de primeira ordem na forma clausal, Peter Baumgartner, (Schumann, 1994).
- Prover/NP-Tools, sistema para a lógica de primeira ordem, no caso de não encontrar nenhuma demonstração para uma dada fórmula este sistema produz um contra-modelo, Graeme I. Parkin e Gunnar Stalmarck, (Kohlhase & Talcott, 1998).
- PTTP (Prolog Technology Theorem Prover), uma implementação do procedimento de eliminação de modelos que faz a extensão do *Prolog* para a lógica de primeira ordem completa, M. Stickel, (Schumann, 1994).
- pTAB, sistema do tipo *tableau* para a lógica de primeira ordem, A. Wolf, (Schumann, 1994).
- PVS, sistema de especificação/verificação baseado na lógica de primeira ordem, John Rushby, N. Shankar e Sam Owre, (Kohlhase & Talcott, 1998).
- ReDuX, sistema para programar e experimentar sistemas de re-escrita, Reinhard Buendgen, (Kohlhase & Talcott, 1998).
- RRL, sistema automático para a lógica de primeira ordem com igualdade, Deepak Kapur e Hantao Zhang, (Kohlhase & Talcott, 1998).
- Saturate, sistema para a lógica de primeira ordem baseado no método de saturação, Harald Ganzinger e Robert Nieuwenhuis, (Kohlhase & Talcott, 1998).
- Schwind[3], sistema do tipo *tableau* para as lógicas de primeira ordem, modal e temporal, C. B. Schwind, (Schumann, 1994).
- SDVS, sistema para a especificação e demonstração de programas baseados numa lógica temporal, Ranwa Haddad e Leo Marcus, (Kohlhase & Talcott, 1998).
- SETHEO, sistema do tipo *tableau* baseado num cálculo de eliminação de modelos para a lógica de primeira ordem, J. Schumann, (Schumann, 1994).
- SHARE, sistema do tipo *tableau* para a lógica de primeira ordem, J. Posegga, (Schumann, 1994).

[3]Nome do autor, o sistema não tem nome definido.

- SPIN, sistema de verificação de modelos baseado no CSP, Gerard Holzmann, (Kohlhase & Talcott, 1998).
- SPASS & FLOTTER, protótico de um sistema para a lógica de primeira ordem, e tradutor-CNF, Cristoph Weidenbach, (Kohlhase & Talcott, 1998).
- Tableau, sistema do tipo *tableau* que implementa o algoritmo de inferência de Smullyan, para a lógica de primeira ordem na forma clausal, J. Crawford e L. D. Auton, (Schumann, 1994).
- Tableaux, sistema do tipo *tableau* para a lógica de primeira ordem, Ron Burback, (Kohlhase & Talcott, 1998).
- TempEst, conjunto de programas para a verificação formal de propriedades de segurança ("safety properties") de programas em Esterel, Carlos Puchol, (Kohlhase & Talcott, 1998).
- Tools for TLA, especificações baseadas na lógica temporal de acções, Heiko Krumm, (Kohlhase & Talcott, 1998).
- TPS, sistema para a lógica de primeira ordem e para a lógica de ordem superior, Peter Andrews, (Kohlhase & Talcott, 1998).
- UV, sistema interactivo para verificação de modelos para *UNITY*, Markus Kaltenbach, (Kohlhase & Talcott, 1998).
- Vallstroem[4], sistema do tipo *tableau* para a lógica modal de Solovay G e G^*, D. Vallstroem, (Schumann, 1994).
- VERSA, sistema para analisar limites de recursos em sistemas de tempo real usando a álgebra de processos ACSR ("Algebra of Communicating Shared Resources"), D.Clarke, (Kohlhase & Talcott, 1998).
- VSE, sistema formal de suporte ao desenvolvimento de programas, Dieter Hutter e Werner Stephan, (Kohlhase & Talcott, 1998).
- Yarrow, sistema interactivo para a teoria dos tipos, Jan Zwanenburg, (Kohlhase & Talcott, 1998).

[4]Nome do autor, o sistema não tem nome definido.

Referências

Alagič, Suad, & Arbib, Michael A. 1978. *The Design of Well-Structured and Correct Programs.* New York: Springer-Verlag.

Andrews, Peter B. 1986. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, Inc.

A.S.Troelstra, & Dalen, D.van. 1988. *Constructivism in Mathematics — An Introduction.* Studies in Logic, and The Foundations of Mathematics, vol. 1. Amsterdam: North-Holland Publishing Co.

Avron, Arnon. 1991. Simple Consequence Relations. *Information and Computation*, **92**(1), 105–139.

Barendregt, Hendrick Pieter. 1981. *The Lambda Calculus, Its Syntax and Semantics.* Studies in Logic and The Foundations of Mathematics, vol. 103. Amsterdam: North-Holland Publishing Company.

Barwise, John (ed). 1977. *Handbook of Mathematical Logic.* Studies in Logic and the Foundations of Mathematics, vol. 90. Amsterdam: North-Holland Publishing Company.

Basin, David A., Bundy, Alan, Kraan, Ina, & Matthews, Sean. 1993. A Framework for Program Development Based on Schematic Proof. *Pages 162–171 of: Proceedings of the 7th International Workshop on Software Specification and Design.* IEEE Computer Society Press.

Bibel, W., & Eder, E. 1993. Methods and Calculi for Deduction. *Pages 67–182 of:* Gabbay, Dov M ., Hogger, C. J., & Robinson, J. A. (eds), *Handbook of Logic in Artificial and Logic Programming*, vol. 1. Oxford: Oxford Science Publications.

Borceaux, Francis. 1994. *Handbook of categorical Algebra 1, Basic Category Theory.* Encyclopedia of Matehmatics and its applications. Cambridge: Cambridge University Press.

Chang, Chin-Liang, & Lee, Richard Char-Tung. 1973. *Symbolic Logic and Mechanical Theorem Proving.* Orlando: Academic Press, Inc.

CHURCH, ALONZO. 1941. *The Calculi of Lambda-Converstion.* Princeton: Princeton University Press.

CLINT, M., & HOARE, C. A. R. 1972. Program Proving: Jumps and Functions. *Acta Informatica*, 214–224.

COUSOT, PATRICK. 1990. *Methods and Logics for Proving Programs.* Second edn. Handbook of Theoretical Computer Science, vol. B. Amesterdam: Elsevier. Chap. 15, pages 841–993.

CURRY, HASKELL B., & FEYS, ROBERT. 1968. *Combinatory Logic.* 2nd edn. Studies in Logic and the Foundations of Mathematics, vol. I. Amsterdam: North-Holland Pub. Comp.

DAWSON, MARK. 1991 (May). *Deriving Rules with the Imperial Logic Environment.* Technical Report DOC-91-39. Departement of Computing of the Imperial College of Science Technology and Medecine, London.

DAWSON, MARK. 1992. *ICLE - The Imperial College Logic Environment, Reference Manual.* Imperial College of Science, Technology & Medicine, London.

DIJKSTRA, EDSGER W. 1975. Guarded Commands, Nondeterminacy and Formal Derivations of Programs. *Communications of the ACM*, **18**(8), 453–457.

FEFERMAN, SOLOMON. 1994 (May 13). *Deciding the Undecidable: Wrestling with Hilbert's Problems.* Text of inaugural lecture as the Patrick Suppes Family Professor in Humanities and Sciences, Stanford University.

FIADEIRO, JOSÉ, & SERNADAS, AMÍLCAR. 1988. Structuring Theories on Consequence. *Pages 44–72 of:* SANNELLA, D., & TARLECKI, A. (eds), *Recent Trends in Data Type Specification.* Lecture Notes in Computer Science, no. 332. Springer-Verlag.

GALLIER, JEAN H. 1987. *Logic For Computer Science: Foundations of Automatic Theorem Proving.* Computer Science and Technology Series. New York: John Wiley & Sons.

GIRARD, JEAN-YVES. 1987. *Proof Theory and Logical Complexity.* Studies in Proof Theory, vol. 1. Napoli: Bibliopolis.

GIRARD, JEAN-YVES, TAYLOR, P., & LAFONT, Y. 1988. *Proofs and Types.* Cambridge University Press.

GOGUEN, JOSEPH. 1988 (August). *OBJ as Theorem Prover with Applications to Hardware Verification.* Tech. rept. SRI-CSL-88-4R2. SRI International.

GOGUEN, JOSEPH. 1990 (October). Proving and Rewriting. *In: Proceedings 2nd International Conference on Algebraic and Logic Programming.*

GOGUEN, JOSEPH, & BURSTALL, ROD. 1984. Some Fundamental Algebraic Tools for the Semantics of Computation, Part 1: Comma Categories, Colimits, Signatures and Theories. *Theoretical Computer Science*, **31**(2), 175–209.

GOGUEN, JOSEPH, & BURSTALL, ROD. 1985. *A Study in the Foundations of Programming Methodology: Specifications, Institutions, Charters and Parchments.* Lecture Notes in Computer Science, vol. 240. Springer-Verlag. Pages 313–333.

GOGUEN, JOSEPH, & BURSTALL, ROD. 1992. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the Association for Computing Machinery*, **39**(1), 95–146.

GOGUEN, JOSEPH, & WINKLER, TIMOTHY. 1988 (August). *Introducing OBJ.* Technical Report SRI-CSL-88-9. SRI International, Computer Science Lab.

GOGUEN, JOSEPH, WINKLER, TIMOTHY, MESEGUER, JOSÉ, FUTATSUGI, KOKICHI, & JOUANNAUD, JEAN-PIERRE. 1992 (March). *Introducing OBJ.* Technical Report SRI-CSL-92-03. SRI International. Draft.

GORDON, MICHAEL J., MILNER, ARTHUR J., & WADSWORTH, CRISTOPHER P. 1979. *Edinburgh LCF.* Lecture Notes in Computer Science, vol. 78. Berlin: Springer-Verlag.

GRIES, DAVID. 1981. *The Science of Programming.* New York: Springer-Verlag.

HAMILTON, A. G. 1991. *Logic for Mathematicians.* Cambridge University Press.

HARPER, ROBERT. 1986. *Introduction to Standard ML.* University of Edinburgh.

HARPER, ROBERT, HONSELL, FURIO, & PLOTKIN, GORDON. 1993. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, **40**(1), 143–184.

HERRLICH, HORST, & STRECKER, GEORGE. 1973. *Category Theory.* Allyn and Bacon Inc.

HEUERDING, ALAIN, JÄGER, GERHARD, SCHWENDIMANN, STEFAN, & SEYFRIED, MICHAEL. 1996. The Logics Workbench LWB: A Snapshot. *Euromath Bulletin*, **1**(2), 177–186.

HOARE, C. A. R. 1971. Procedures and Parameters: An Axiomatic Approach. *Lecture Notes in Mathematics*, **188**, 102–106.

HOARE, C. A. R. 1972. Proof of Correctness of Data Representations. *Acta Informatica*, **1**, 271–281. Springer-Verlag.

HOARE, C. A. R. 1973. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica*, **2**, 335–355.

HOL88A. 1991. *The HOL System TUTORIAL.* 2nd edn. DSTO, Cambridge University, CRC-SRI.

HOL88B. 1991. *The HOL System DESCRIPTION.* DSTO, Cambridge University, CRC-SRI.

HOL88C. 1991a. *The HOL System LIBRARIES.* DSTO, Cambridge University, CRC-SRI.

HOL88C. 1991b. *The HOL System REFERENCE.* DSTO, Cambridge University, CRC-SRI.

HOWARD, W. A. 1980. The formulae-as-types notion of construction. *In:* HINDLEY, J. R., & SELDIN, J. P. (eds), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press.

HUANG, X., KERBER, M., KOHLHASE, M., MELIS, E., NESMITH, D., RICHTS, J., & SIEKMANN, J. 1994. KEIM: A Toolkit for Automated Deduction. *In: Proceedings of CADE12.*

KALVALA, SARA. 1994. *A Gentle Introduction to Isabelle.* Tech. rept. University of Cambridge, Computer Laboratoty.

KOHLHASE, MICHAEL, & TALCOTT, CAROLYN. 1998. *Mechanized Reasoning Systems.* http://www-formal.stanford.edu/clt/ARS/ars-db.html.

KOLYANG, SANTEN, T., & WOLFF, B. 1996. A Structure Preserving Encoding of Z in Isabelle/HOL. *Pages 283–298 of:* VON WRIGHT, J., GRUNDY, J., & HARRISON, J. (eds), *Theorem Proving in Higher Order Logics — 9th International Conference.* LNCS, no. 1125. Springer Verlag.

LAMBEK, JOACHIM. 1958. Deductive Systems and Categories I. *Mathematical Systems Theory*, **2**(4), 287 – 318.

LAMBEK, JOACHIM. 1969. Deductive Systems and Categories II. *Lecture Notes in Mathematics*, 76–122.

LAMBEK, JOACHIM. 1971. Deductive System and Categories III. *Lectures Notes in Mathematics*, 57 – 82.

LAMBEK, JOACHIM, & SCOTT, P. J. 1988. *Introduction to Higher Order Categorical Logic.* Cambridge Studies in Advance Mathematics. Cambridge: Cambridge University Press.

MACLANE, S. 1971. *Categories for the Working Mathematician.* New York: Springer-Verlag.

MATTEWS, BRIAN. 1996 (October). *MERILL, An Equational Reasoning System in Standard ML, A User Guide.* Software Engineering Group, Informatics Department Rutherford Appleton Laboratory.

MEYER, B. 1990. *Introduction to the Theory of Programming Languages.* New York: Prentice Hall International.

MICHAYLOV, SPIRO, & PFENNING, FRANK. 1991. Natural Semantics and Some of its Meta-Theory in Elf. *Pages 299–344 of:* ERIKSSON, L.-H., HALLNÄS, L., & SCHROEDER-HEISTER, P. (eds), *Proceedings of the Second International Workshop on Extensions of Logic Programming.* Stockholm, Sweden: Springer-Verlag LNAI 596.

NESMITH, DAN. 1993. *Using Keim.*

OLIVEIRA, A. J. FRANCO. 1979. *Lógica Elementar (Introdução à Lógica Matemática).* Textos de Lógica Matemática, Teoria de Conjuntos e Fundamentos, vol. 1. Faculdade de Ciências de Lisboa: Associação de Estudantes da Faculdade de Ciências de Lisboa.

PAULSON, LAWRENCE C. 1990. Isabelle: The Next 700 Theorem Provers. *Pages 361–386 of:* ODIFREDDI, P. (ed), *Logic and Computer Science.* Academic Press.

PAULSON, LAWRENCE C. 1991. *ML for the Working Programmer.* Cambridge: Cambridge University Press.

PAULSON, LAWRENCE C. 1992. Designing a Theorem Prover. *In:* ABRANSKY (ed), *Handbook of Logic in Computer Science*, vol. II. Oxford: Oxford Science Publications.

PAULSON, LAWRENCE C. 1993a. *Introduction to Isabelle.* Tech. rept. 280. University of Cambridge, Computer Laboratory.

PAULSON, LAWRENCE C. 1993b. *The Isabelle Reference Manual.* Tech. rept. 283. University of Cambridge, Computer Laboratory.

PAULSON, LAWRENCE C. 1993c. *Isabelle's Object-Logics.* Tech. rept. 286. University of Cambridge, Computer Laboratory.

PFENNING, FRANK. 1991. Logic Programming in the LF Logical Framework. *In:* HUET, GÉRARD, & PLOTKIN, GORDON D. (eds), *Logical Frameworks*. Cambridge University Press.

POHLERS, WOLFRAM. 1980. *Proof Theory, An Introduction*. Lecture Notes in Mathematics, vol. 1407. Berlin: Springer-Verlag.

QUARESMA, PEDRO. 1995 (January). Implementing a Deduction System for a Guarded Commands Procedural Language using a Generic Theorem Prover. *Pages 14 – 26 of:* MARKOVITCH, SHAUL (ed), *Proceedings of the 11th Israeli Symposium on Artificial Intelligence*. CIS Report, no. 9502. Technion - Israel Institute of Technology, Haifa 32000 Israel.

ROBINSON, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, **12**(1), 23–41.

RYDHEARD, D.E., & BURSTALL, R.M. 1988. *Computacional Category Theory*. Computer Science. Prentice-Hall International.

SCHNEIDER, KLAUS, & KROPF, RAMAYYA KUMAR THOMAS. 1992. Efficient Representation and Computation of Tableaux Proofs. *In: Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its applications.*

SCHNEIDER, KLAUS, KUMAR, RAMAYYA, & KROPF, THOMAS. 1992a. The Faust-Prover. *Lectures Notes in Computer Science*, **607**.

SCHNEIDER, KLAUS, KUMAR, RAMAYYA, & KROPF, THOMAS. 1992b. Structuring Hardware Proofs: First steps towards Automation in a Higher-Order Environment, CAV-91. *In: Proceedings of the Conference on Computer Aided Verification*. Lecture Notes in Computer Science, no. 575. Berlin: Springer-Verlag.

SCHNEIDER, KLAUS, KUMAR, RAMAYYA, & KROPF, THOMAS. 1993. Hardware-Verification using First Order BDDs. *Pages 45–62 of: Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications*. IFIP Transactions A: Computer Science and Technology, vol. 32. Amsterdam, The Netherlands: North-Holland.

SCHUMANN, J. 1994. Tableau-Based Theorem Provers: Systems and Implementations. *Journal of Automated Reasoning*, **13**, 409–421.

SCHÜTTE, KURT. 1977. *Proof Theory*. Berlin: Springer-Verlag.

SERNADAS, A., SERNADAS, C., & CALEIRO, C. 1997a. *Fibring of Logics as a Categorial Construction*. Research Report. Section of Computer

Science, Department of Mathematics, Instituto Superior Tecnico, 1096 Lisboa. Submitted for publication.

SERNADAS, A., SERNADAS, C., & CALEIRO, C. 1997b. Synchronization of logics. *Studia Logica*. In print.

SERNADAS, A., SERNADAS, C., & CALEIRO, C. 1997c (Setembro). *Synchronization of Logics with Mixed Rules: Completeness Preservation*. Escola "Fundamentos teóricos da Computação".

STEVENS, ANDREW, & HOBLEY, KEITH. 1993 (February). *Mechanised Theorem Proving with 2OBJ: A Tutorial Introduction*. Draft edn.

SZABO, M. E. (ed). 1969. *The Collected Papers of Gerhard Gentzen*. Studies in Logic and The Foundations of Mathematics. Amesterdam: North-Holland Publishing Company.

SZABO, M. E. 1976. Quantifier-Complet Categories. *Journal of Pure and Applied Algebra*, **7**, 97–114. North-Holland Publishing Company.

SZABO, M. E. 1978. *Algebra of Proofs*. Studies in Logic and The foundations of Mathematics, vol. 88. Amsterdam: North-Holland Publishing Company.

TAKEUTI, GAISI. 1975. *Proof Theory*. Studies in Logic and The Foundations of Mathematics, vol. 81. Amsterdam: North-Holland Publishing Company.

VAN DALEN, DIRK. 1980. *Logic and Structure*. Universitext. Springer-Verlag.

VELOSO, PAULO, DOS SANTOS, CLÉSIO, AZEREDO, PAULO, & FURTADO, ANTÓNIO. 1983. *Estruturas de Dados*. Rio de Janeiro: Editora Campus Ltda.

WELSH, JIM, & ELDER, JOHN. 1979. *Introduction to Pascal*. 2nd edition edn. Computer Science. London: Prentice-Hall International, Inc.

WIKSTRÖM, ÅKE. 1989. *Functional Programming Using Standard ML*. Computer Science. Prentice-Hall International.

WIRTH, NIKLAUS. 1973. *Systematic Programming*. Englewood Clifs, New Jersey: Prentice-Hall, Inc.

WIRTH, NIKLAUS. 1976. *Algorithms + Data Structures = Programs*. Automatic Computation. Englewood Clifs, New Jersey: Prentice-Hall, Inc.