Um Estudo do Sistema *Oyster-Clam*, Implementação de Reescrita de Tipos e Uma Formalização Parcial da Teoria dos Grafos

Jerônimo Pellegrini

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Estudo do Sistema *Oyster-Clam*, Implementação de Reescrita de Tipos e Uma Formalização Parcial da Teoria dos Grafos

Jerônimo Pellegrini[1]

junho de 1997

Banca Examinadora:

- Jacques Wainer (Orientador)
- Flávio Soares Correa da Silva
  (IME - USP)
- Arnaldo Vieira Moura
  (Instituto de Computação - Unicamp)
- Cláudio L. Lucchesi
  (Instituto de Computação - Unicamp - Suplente)

[1]Este trabalho recebeu suporte financeiro da CAPES.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Pellegrini, Jerônimo

P364e Um estudo do sistema Oyster-Clam, implementação de reescrita de tipos e uma formalização parcial da teoria dos grafos / Jerônimo Pellegrini -- Campinas, [S.P. :s.n.], 1997.

Orientador : Jacques Wainer

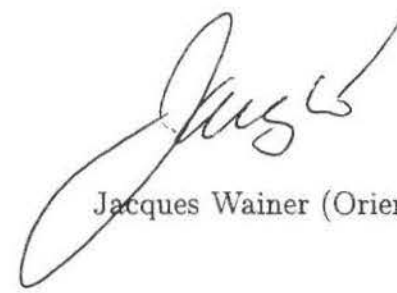
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Demonstração automática de teoremas. 2. Lógica simbólica e matemática. 3. Inteligência artificial. I. Wainer, Jacques. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Um Estudo do Sistema *Oyster-Clam*, Implementação de Reescrita de Tipos e Uma Formalização Parcial da Teoria dos Grafos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Jerônimo Pellegrini e aprovada pela Banca Examinadora.

Campinas, 19 de Maio de 1999.



Jacques Wainer (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 31 de julho de 1997 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. Flávio Soares Corrêa da Silva

Prof. Dr. Arnaldo Vieira Moura

Prof. Dr. Jacques Wainer

Agradecimentos

À minha família, da qual sempre tive todo o apoio;

Ao meu orientador Jacques, que me mostrou caminhos sem tolher a liberdade;

Ao professor Flávio Soares Correa da Silva, do IME-USP;

A Alan Bundy, Wamberto Wasconcelos, Alan Smaill, Ina Kraan e Santiago Negrete por sugestões e esclarecimentos sobre o sistema Oyster-Clam;

A todos os funcionários do IC, em especial ao Luiz (in memoriam), Roseli, Vera, Daniel e Niero;

A Cláudio Müller, por uma ótima estadia durante o Curso de Verão;

Aos amigos do IC, Aminadab, Ana Monteiro, Bruno, Célio, Chico Watzko, Chris Campos, Cláudia Nalon, Delano, Elder, Gisele, Guto, Karen, Marco Aurélio, Marcos André, Maria Emília, Nivando, Patrícia, Socorro, Solange, Vitor Hugo;

A todos os professores do Instituto de Computação;

Aos amigos de muito tempo, Ademir, Cereja, Flávio, Jhuli, Kátia, pelo incalculável apoio.

Ao professor Pedro Paulo Ayrosa, da Universidade Estadual de Londrina, responsável por meu primeiro interesse pela Inteligência Artificial;

À CAPES pelo suporte financeiro.

E a todos que de qualquer forma colaboraram para a realização desta tese.

Resumo

Nesta tese, mostramos uma implementação do processo de diagonalização de Cantor no sistema de prova de teoremas Oyster-Clam. Para isto, tivemos que estender o Oyster com comparação e indução em tipos, e desenvolvemos um método e algumas regras de reescrita. As regras de reescrita lidam com tipos, o que não era suportado ainda no sistema Oyster-Clam; algumas modificações foram feitas para que isto se tornasse possível.

Também desenvolvemos esquemas de indução para grafos neste sistema, e provamos alguns teoremas.

Abstract

In this thesis, we show an implementation of Cantor's diagonalization process in the Oyster-Clam theorem proving system. To achieve that, we have extended the Oyster logic with comparison and induction on types, and developed a method and some rewrite rules. The rewrite rules deal with types, what was not supported yet in the Oyster-Clam system, and some modifications were done to make that possible.

We have also developed induction schemes for graphs in that system, and some theorems were proven.

How happy is the little stone
That rambles in the road alone,
And doesn't care about carreers,
And exigences never fears;

Emily Dickinson (1830-1886)

Saber? Que sei eu?
Pensar é descrer.

Fernando Pessoa (1888-1935)

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

"Teu primeiro dever é discernir e aceitar, sem fúteis revoltas, os limites da mente humana para, dentro desses severos limites, labutar sem protestos nem desfalecimentos."
Nikos Kazantzákis (1883-1957) - *Ascese*

1.1 O Contexto

Desenvolvido em Edimburgo como reconstrução do sistema *NuPRL* [Con86] , o sistema *Oyster* é uma implementação da lógica de mesmo nome – uma variação da teoria de tipos de Martin-Löf. O *Oyster* é um ambiente de desenvolvimento de provas, exigindo um nível muito alto de interação com o usuário.

O *Oyster* é uma lógica construtiva, e a idéia algoritmica associada a cada prova é construída automaticamente à medida que regras de inferência são aplicadas. A cada regra aplicada, um λ-termo é expandido; assim, ao final da prova, terá sido construído um programa funcional.

Também foi desenvolvido pelo mesmo grupo o *Clam* – uma implementação do conceito de planos de prova, agindo sobre o ambiente *Oyster*. A técnica utilizada é semelhante ao paradigma de planejamento em Inteligência Artificial. O conjunto *Clam* rodando sobre *Oyster* é normalmente chamado de sistema *Oyster-Clam*. O *Oyster-Clam* é um provador automático de teoremas, não exigindo muita interação com o usuário.

O sistema *Clam* foi desenvolvido para realizar provas indutivas, fazendo uso da técnica de *rippling*[1], sendo particularmente eficiente nestas provas. Os métodos usados neste

[1]Explicada em detalhes no capítulo 4

sistema, no entanto, não constituem uma biblioteca fixa, e nada impede que o mesmo seja usado em diversos outros tipos de prova.

1.2 As Contribuições

Realizamos dois trabalhos diferentes com o sistema *Oyster-Clam*; o primeiro é a implementação da diagonalização de Cantor, e o segundo, a implementação de esquemas de indução para grafos.

1.2.1 Diagonalização

Em comunicação pessoal, Alan Bundy sugeriu a implementação de um método no *Clam* que realizasse provas por diagonalização, que o sistema não realizava ainda. O *Clam* faz uso pesado de reescrita de termos ao implementar o *rippling*. No entanto, apenas variáveis podem ser reescritas, não sendo a reescrita de tipos suportada na versão do sistema distribuída atualmente. Fizemos algumas modificações no *Oyster* e no *Clam* a fim de implementar a reescrita de tipos. No *Oyster*, foi necessário implementar a indução em tipos, e no *Clam*, algumas pequenas alterações nas táticas relacionadas à reescrita.

Fazendo uso desta nova possibilidade, desenvolvemos um método *Clam* para provas por diagonalização. Alguns teoremas foram provados usando este método, sobre a cardinalidade do conjunto potência, existência de função não-computável, e insolvabilidade do problema da parada.

1.2.2 Grafos

A teoria de grafos tem reconhecidamente uma ampla gama de aplicações. A prova de teoremas sobre grafos com a geração automática de um programa correspondente à prova é, então, um objetivo de grande interesse. Apresentamos a formalização de alguns conceitos e operações em grafos, que foram implementados no sistema *Oyster-Clam*, sendo de particular interesse a construção de esquemas de indução no *Clam* para grafos, uma vez que o *Clam* é um provador indutivo, e que a indução tem papel importantíssimo na prova de teoremas em grafos. Ao descrever estas implementações, mostramos como novos esquemas de indução são criados no *Clam*.

Não tentamos oferecer uma formalização total da teoria de grafos, é claro. A formalização feita é de alguns conceitos apenas; nossa intenção é estudar o processo de criação

de novos esquemas de indução no *Clam*, e mostrar a utilidade e relativa facilidade da formalização de conceitos da teoria de grafos.

1.3 Organização da Tese

Nos capítulos 2 e 3, apresentamos uma introdução ao sistema *Oyster-Clam*. No capítulo 4, temos a implementação de métodos e regras de reescrita que permitem ao sistema *Oyster-Clam* realizar provas por diagonalização. Utiliza-se, para isto, a reescrita de tipos. Por último, no capítulo 5, mostramos uma prova realizada no *Oyster* de alguns teoremas sobre coloração de arestas em grafos, e ilustramos o uso do λ-termo associado às provas. O capítulo 6 é dedicado às conclusões e considerações finais.

Capítulo 2

Oyster

> *"From a drop of water a logician could predict an Atlantic or a Niagara."*
> Sir Arthur Conan Doyle (1859-1930) - *A Study in Scarlet*

O *Oyster* [Hor88] é um ambiente para desenvolvimento de provas de teoremas, não se tratando de um provador automático. O sistema permite a edição de provas e as verifica na medida em que são desenvolvidas. *Oyster* é também o nome da lógica subjacente, que é uma variação da teoria intuicionista de tipos de Martin-Löf [ML82].

Os teoremas no *Oyster* são provados interativamente, através do uso de táticas. Táticas são sequências de regras frequentemente usadas, que realizam parte da prova de um teorema. O usuário do *Oyster* pode escolher entre táticas aplicáveis a cada sequente da prova; pode também desenvolver novas táticas.

2.1 A Teoria de Tipos de Martin-Löf

A Teoria de Tipos de Martin-Löf é uma teoria de tipos intuicionista, de alta ordem, cuja linguagem inclui a do λ-cálculo.

2.1.1 Proposições como Tipos

Nesta teoria, há equivalência entre as noções de proposição e tipo. Um tipo T será habitado por um elemento e quando e for uma prova da proposição T. Assim,

$$a \in A$$

significa tanto "a pertence ao tipo A" como "a é uma prova da proposição A". Na linguagem do *Oyster*, isto é "$a{:}A$". Os tipos são habitados por provas de sua existência (estas provas são λ-termos, como veremos adiante). O tipo dos números inteiros, *int*, por exemplo, é habitado por todos os números naturais - e qualquer número natural é a prova de que o tipo é habitado. Assim, para provar a proposição "*int*", basta apresentar um número inteiro.

Conectivos

Não apresentaremos a notação da teoria de Martin-Löf, por nos parecer desnecessária; apresentaremos a teoria usando a notação do *Oyster*.

O símbolo # representa produto cartesiano: $A\#B$ é o conjunto de todas as combinações de elementos de A e B (pares ordenados). Dizer que o tipo $A\#B$ é habitado, é dizer que tanto A como B são habitados (ou não seria possível formar um par ordenado que habitasse $A\#B$); assim, # é também interpretado como "e" lógico, quando lemos os tipos como proposições.

O símbolo \ representa união disjunta: $A\backslash B$ é o conjunto de todos os elementos de A e de B. Dizer que o tipo $A\backslash B$ é habitado é dizer que ao menos um dentre A e B é habitado (ou não seria possível obter um elemento que habitasse $A\backslash B$); assim, \ é também interpretado como "ou" lógico.

O símbolo => representa um conjunto de funções: $A => B$ é o conjunto de todas as funções de A em B. Dizer que o tipo $A => B$ é habitado é dizer que, se A é habitado, B também tem que ser, ou não haveria função habitando $A => B$ (pois não há elementos no contra-domínio); assim, => é também interpretado como implicação lógica.

O quantificador universal é obtido com o símbolo =>. Dizer "$a{:}A => B(a)$" é dizer que, sempre que tivermos um elemento $a \in A$, $B(a)$ vale, i.e., $\forall a \in A, B(a)$.

O quantificador existencial é obtido com o símbolo #. Dizer "$a{:}A\ \#B(a)$" é dizer que A é habitado pelo elemento a, e que $B(a)$ vale, i.e., $\exists a \in A, B(a)$.

A teoria de Martin-Löf foi desenvolvida para tipos arbitrários; estudaremos apenas os tipos existentes na implementação do *Oyster*.

2.1.2 Universos

No *Oyster*, há uma hierarquia cumulativa de universos (semelhante à hierarquia cumulativa de conjuntos em ZF [Pri96]).

O universo u_1 é construido indutivamenteda seguinte forma:

$$\begin{array}{rcl} int, pnat, atom, unit, void & \in & u_1; \\ x, y \in u_1 & \text{então} & x\#y \in u_1; \\ x, y \in u_1 & \text{então} & x\backslash y \in u_1; \\ x, y \in u_1 & \text{então} & x => y \in u_1; \\ x \in u_1 & \text{então} & x\ list\ \in u_1. \end{array}$$

E para todo $i > 1$, cada universo u_i é o menor conjunto tal que:

$$\begin{array}{rcl} u_{i-1} & \in & u_i; \\ u_{i-1} & \subset & u_i; \\ u_j, u_k \in u_i & \text{então} & u_j\#u_k \in u_i; \\ u_j, u_k \in u_i & \text{então} & u_j\backslash u_k \in u_i; \\ u_j, u_k \in u_i & \text{então} & u_j => u_k \in u_i; \\ u_j \in u_i & \text{então} & u_j\ list\ \in u_i. \end{array}$$

Note que os elementos de $int, pnat, atom$, etc. não pertencem a nenhum dos universos (a transitividade não vale para que $x \in int$ então $x \in u_1$, por exemplo). A notação do *Oyster* para u_i é `u(i)`.

2.1.3 Regras de Inferência

O Oyster é uma lógica baseada em *sequentes*. Um sequente é um par $< A, B >$ de conjuntos finitos de fórmulas; Usualmente, sequentes são denotados $A \rightarrow B$, sugerindo implicação lógica, que realmente é o que o sequente representa. No Oyster, um sequente é denotado $A \vdash B$.

Não estudaremos as regras da Teoria de Martin-Löf; ao invés disto, estudaremos suas versões no *Oyster*. Agindo sobre um dado sequente $H \vdash C$, as regras de inferência no *Oyster* dividem-se em dois grupos: regras intro, agindo nas conclusões (C) do sequente, e regras elim, que agem nas hipóteses (H). As regras do *Oyster* são comentadas na seção 2.3.4.

2.2 Tipos no *Oyster*

No *Oyster* existe a mesma noção de proposições como tipos. Para cada tipo, poderemos ter operadores de decisão, uma relação de ordem (quando isto fizer sentido), e esquemas de indução.

A seguir, temos uma lista dos tipos do *Oyster*:

- void: este tipo não é habitado por nenhum elemento. Normalmente, proposições falsas são escritas como $P => void$ (se P for verdadeira, há uma prova para void, i.e., void é habitado – o que é absurdo);
- `atom`: tipo de strings, representadas como $atom('.....')$;
- `pnat`: naturais com aritmética de Peano;
- `int`: inteiros;
- `list`: listas (usando outros tipos). Exemplo: "*int list*" é o tipo de listas de inteiros. As listas são da forma `hd::tl`, onde *hd* é a cabeça e *tl* a cauda da lista;
- função: funções (que levam elementos de um tipo em outro), ou implicação (pois leva provas de uma proposição em provas de outra);
- união disjunta: união disjunta entre dois tipos, ou proposições do tipo "A ou B". pois tem prova de A ou de B;
- produto: pares ordenados (pares de elementos de dois tipos) "$A\#B$", ou proposição do tipo "A e B", pois contém prova para proposições A e B. Os pares ordenados $< A, B >$ são denotados por `A&B`;
- quociente: classes de equivalência dentro de um tipo – no tipo $A///[x, y, R]$, A é um tipo base, e R é uma relação entre x e y. O tipo $A///[x, y, R]$ contém os elementos de A onde a igualdade é dada pela relação R entre x e y Por exemplo, no tipo $int///[m, n, m + n < m * n]$ é composto de inteiros; dois destes elementos serão iguais (pertencerão à mesma classe) quando $m + n < m * n$.
- subconjunto: subtipos de um tipo – o tipo $\{x : T \backslash P\}$ é o conjunto de elementos x de T para os quais a propriedade P vale;
- definições recursivas: estruturas definidas recursivamente;

Além destes, há os tipos `less`, `pless` e `acc`, que não discutiremos aqui.

Os quantificadores são expressados da seguinte forma: $a{:}A{=>}B$ significa que para todo a do tipo A, B vale. E $a{:}A\#\ B$ significa que existe um elemento do tipo A, e que a propriedade B vale. Normalmente, B contém referência ao elemento a. Por exemplo, "$a{:}int\ \#\ a{>}5$" significa "$\exists a \in \mathbb{Z}, a > 5$".

O *Oyster* é uma lógica de alta ordem, baseado em sequentes, e a ordem de uma proposição (ou tipo) é o universo a que esta pertence. O universo u(0) só contém dois elementos: *void* e *unit*. O universo u(i) contém todos os tipos descritos acima, quando estes são proposições de i-ésima ordem.

As provas no *Oyster* são representadas por uma árvore, que é expandida à medida que táticas vão sendo aplicadas. Uma das propriedades de um nó da árvore é o universo a que ele pertence. Outra propriedade de um nó é seu status:

- *Incompleto* quando não foi expandido ainda;
- *Parcialmente completo* se já foi expandido, mas sua sub-árvore contém nós incompletos;
- *Completo* se sua sub-árvore só contém nós completos;
- *Completo com justificativa* se sua sub-árvore contém algum nó que foi fechado não com regra de inferência, mas "à força", usando a tática `because/0`.

Os teoremas são representados na forma `H==>G` ($H \models G$), onde H é um conjunto de hipóteses, representado no *Oyster* como lista. As hipóteses são sempre da forma "$e : T$", afirmando que certo elemento e habita um tipo T.

Teoremas são provados através da construção de um elemento do tipo G (provando assim que ele é habitado). Este elemento é um λ-termo, chamado *extract term*. Como exemplo, considere o teorema:

$$\vdash \forall i \in \mathbb{N}\ \ \exists j \in \mathbb{N}$$

Na linguagem do *Oyster*:

```
[]==>i:int=>int.
```

onde [] representa uma lista de hipóteses vazia. Para provarmos este teorema, temos que provar que o tipo $int{\rightarrow}int$ é habitado; os elementos que o habitam são funções de $\mathbb{Z}$ em $\mathbb{Z}$. Uma possível prova para este teorema então é:

$$\lambda x(x * x)$$

que é a função que leva x em x^2, habitando $int \rightarrow int$

2.3 Regras de Inferência

Apresentaremos de maneira geral as regras de inferência usadas no *Oyster*. É importante ressaltar que o raciocínio é para trás; assim, as regras são aplicadas, no sistema, de baixo para cima, embora logicamente devam ser lidas de cima para baixo. A idéia é que, como partimos do teorema a ser provado para os axiomas e postulados, a cada regra a aplicar devemos procurar "qual regra poderia ter dado origem a este sequente", e não "que regra aplicar neste sequente".

As regras no *Oyster* reflem a natureza construtiva da regra, gerando um *extract term* a partir de outro.

2.3.1 Regras Intro

As regras intro são aplicadas na conclusão do sequente, introduzindo conectivos nesta. Sendo o raciocínio para trás, estas regras eliminarão os conectivos à medida que as aplicamos e descemos na árvore de prova.

$\supset$-intro

$$\frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \supset B}$$

Uma forma do teorema de Herbrand, bastante óbvia. No *Oyster*, por exemplo, quando temos um sequente $H \vdash A => B$, podemos usar uma regra intro para obter $H, A \vdash B$, restando apenas provar que B é habitado.

Ao aplicar esta regra, obteremos na verdade dois novos goals: o que já mostramos e um outro, onde devemos provar que A pertence a um certo universo u_i.

#-intro

$$\frac{\Gamma \vdash A, \Gamma \vdash B}{\Gamma \vdash A\#B}$$

O tradicional "e" lógico. Sendo A e B habitados, A e B são verdadeiros. Logo, $A\#B$ é verdadeiro também, e habitado. No *Oyster*, dado o sequente $H \vdash A\#B$, utilizamos uma regra intro para obter $H \vdash A$ e $H \vdash B$. Assim, resta provar que A e B são habitados.

\-intro

$$\frac{\Gamma \vdash A}{\Gamma \vdash A\backslash B}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A\backslash B}$$

Tanto de A como de B podemos concluir $A\backslash B$. Tendo o sequente $A\backslash B$ no *Oyster*, utilizamos uma regra intro, dizendo também qual entre A e B escolheremos, para obter o sequente com apenas um dos dois.

Negação

$$\frac{\Gamma \cup \{A\} \vdash \text{void}}{\Gamma \vdash A \supset \text{void}}$$

Um uso particular do teorema de Herbrand; na verdade, esta regra é instância da regra de $\supset$-intro.

∀-intro

$$\frac{\Gamma \cup \{x : A\} \vdash B(x)}{\Gamma \vdash (\forall x : A), B(x)}$$

Se tivermos uma prova de que, para cada objeto x do tipo A, $B(x)$ vale, podemos concluir que $\forall x : A, B(x)$. Na linguagem do *Oyster*, $A => B$, i.e., existe uma função de A em B.

∃-intro

$$\frac{\Gamma \vdash B(x)}{\Gamma \vdash (\exists x : A), B(x)}$$

Se $B(x)$ vale, e x é um objeto do tipo A, podemos dizer que $\exists x : A, B(X)$. No *Oyster*, o sequente $x : A\#B(x)$ é transformado, por regra intro, em dois outros: $H \vdash x : A$ e $H \vdash B(X)$.

2.3.2 Regras Elim

As regras elim são aplicadas sobre as hipóteses do sequente, eliminando conectivos das conclusões. Aplica-se as regras em hipóteses específicas da lista. Como já dissemos para as regras intro, o raciocínio para trás fará que tenhamos a impressão de que estas regras, na verdade, introduzem conectivos.

#-elim

$$\frac{\Gamma \cup \{A, B\} \vdash \phi}{\Gamma \cup \{A\#B\} \vdash \phi}$$

A regra parece óbvia. Na prática, serve para, quando tivermos um sequente da forma $A\#B$, podermos separar as hipóteses A e B, a fim de fazer referência a cada uma delas.

\-elim

$$\frac{\Gamma \cup \{A\} \vdash \phi, \Gamma \cup \{B\} \vdash \phi}{\Gamma \cup \{A \backslash B\} \vdash \phi}$$

Encontrando um sequente $H, A\backslash B \vdash \phi$ no *Oyster*, usamos uma regra elim para obter $H, A, B \vdash \phi$.

⊃-elim

$$\frac{\Gamma \vdash A, \Gamma \cup \{B\} \vdash \phi}{\Gamma \cup \{A \supset B\} \vdash \phi}$$

Se tivermos provado que A é habitado, e que, se B é habitado, C o é, podemos dizer que $(A \supset B) \supset C$.

∃-elim

$$\frac{\Gamma \vdash A, \Gamma \cup \{(x : A), (B(x))\} \vdash \phi}{\Gamma \cup \{(\exists x : A, B(x))\} \vdash \phi}$$

Esta regra diz simplesmente que, se temos um elemento x habitando A, e se $B(x)$ vale, podemos dizre que $\exists x : A, B(x)$.

∀-elim

$$\frac{\Gamma \vdash A, \Gamma \cup \{(x : A), (B(x))\} \vdash \phi}{\Gamma \cup \{(\forall x : A, B(x))\} \vdash \phi}$$

Se, para cada elemento x habitando A, $B(x)$ vale, podemos dizre que $\exists x : A, B(x)$.

2.3.3 Indução e Recursão

O *Oyster* permite o uso de indução como regra de inferência; os *extract terms* de provas indutivas contêm definições recursivas.

Há três maneiras básicas de se aplicar indução no *Oyster*, a partir das quais outras podem ser criadas: indução forte, fraca e em listas. As regras que aplicam indução são:

$$\frac{P(0), \quad \forall n, (\forall m < n, P(m)) \supset P(n)}{\forall n \;\; P(n)}$$

$$\frac{P(0), \quad \forall n, P(n) \supset P(n+1)}{\forall n \;\; P(n)}$$

$$\frac{P(nil), \quad \forall v \in Tlist, \forall u \in T, \;\; P(v) \supset P(u :: v)}{\forall x \in Tlist, \quad P(x)}$$

Nas definições recursivas, há uma análise de casos, sendo que um deles é o caso "base". Assim, normalmente temos nestas definições: "caso 1: para o valor específico x, a função vale y; caso 2: o valor dado não é x, mas é $f(z)$. Então, a função vale $g(z)$".

Exemplo 2.1 A função *ocorrencias* é definida em listas, e retorna o número de ocorrências de um dado elemento. Podemos defini-la recursivamente da seguinte maneira:

- $l = \text{nil} \supset ocorrencias = 0$;
- $l = h :: t \supset ocorrencias = 1 + ocorrencias(t)$;

Ou seja, a lista l pode ser vazia ou não; fazemos uma análise de casos. No *Oyster*, esta definição é:

$$\text{ocorrencias}(l) = \text{list\_ind}(l, 0, [hd, tl, \text{valor-tl}, 1 + \text{valor-tl}])$$

Onde os parâmetros de *list_ind* são: a variável de recursão, o valor da função para *nil*, e uma lista. Nesta lista, temos a parte recursiva: *hd* e *tl* são cabeça e cauda de l; *valor-tl* é um suposto valor da função para *tl* (a hipótese de indução), e *1+valor-tl* é o valor da função para a lista l. ◇

Similarmente, *p_ind* permite fazer definições recursiva em *pnat*:

Exemplo 2.2 A função:

$$p\_ind(n, nil, [i, l, 1 :: l])$$

de *pnat* em *int list* vale *nil* no ponto 0. Em outros pontos, quando o valor da função no ponto anterior for l, no atual será *1:: l*. A função retornará, para cada $n \in \mathbb{N}$, uma lista de inteiros com n "1"s. ◇

E *ind*, para definições recursivas em *int*. Neste caso, há um valor definido para a função no ponto 0, e duas listas que implementam a recursão em duas direções: para os positivos e negativos.

Exemplo 2.3 A função:

$$ind(n, [\ ,\ , 0], 1, [\ , p, 2 * p])$$

tem como n como variável de recursão; quando $n = 0$, a função vale 1. Quando $n < 0$, a função vale 0. Quando $n > 0$, o valor é calculado a partir do valor p da função para $n - 1$: $2 * p$. Claramente, é a função 2^n em $\mathbb{Z}$. ◇

Além de provas indutivas, outras regras podem dar origem a *extract terms* recursivos. Havendo, por exemplo, uma variável l : *int list* nas hipóteses de um sequente, podemos usar a regra `elim(1)` para obter um λ-termo semelhante ao do exemplo 2.1, gerando dois subgoals; em cada um deles, deveremos provar a existência de uma lista. No segundo deles, teremos entre as hipóteses a cabeça e a cauda de l.

2.3.4 Outras Regras, Táticas e Utilitários

Além das regras já descritas, há os seguintes recursos disponíveis:

- *hyp*: hyp(X) fechará o nó corrente se X estiver entre as hipóteses e for prova do *goal* atual (i.e., se o *goal* é habitado por X);

$$\frac{\emptyset}{\Gamma \cup \{\psi\} \vdash \psi}$$

- *seq*: permite concatenar um lema ao teorema corrente, constituindo alternativa ao uso explícito de um lema;

- *thin*: elimina um subconjunto da lista de hipóteses, que se acredita não serem mais úteis na prova (porque o raciocínio é para trás, e porque a lógica é monotônica);

$$\frac{\Gamma \vdash \psi}{\Gamma \cup \{H_1, H_2, ..., H_n\} \vdash \psi}$$

- *lemma*: permite o uso de lemas externos ao teorema. Os lemas devem ser provados como teoremas à parte;

- *def*: permite o acesso ao λ-termo de um teorema, desde que este esteja declarado na lista de hipóteses ("`h:term_of(T)`");

- *autotactic*: a cada regra aplicada, o *Oyster* tentará aplicar, em seguida e várias vezes, uma determinada tática definida pelo usuário.Algumas vezes é interessante usar "*intro*" como autotática, pois é comum que esta seja usada várias vezes após outras rergas. Normalmente, *autotactic* vale "idtac", que é uma tática vazia (sem efeito);

- *explicit-intro*: permite a construção explícita do λ-termo associado ao nó corrente da prova. Sempre prodizirá, no entanto, um novo nó, onde deverá ser feita uma prova de que o λ-termo escolhido perence à conclusão do sequente deste nó, como pode ser visto no exemplo 2.4;

Exemplo 2.4 O teorema "f" diz que há uma função de *int* em *int*.

```
f: [] incomplete autotactic(idtac)
==> f:int=>int
by _
```

A primeira linha nos diz que a lista de hipóteses é vazia ("[]"); que o nó atual está incompleto, e que a autotática sendo usada é "idtac". Na segunda linha, a conclusão do sequente é "existe uma função f, de $\mathbb{Z}$ em $\mathbb{Z}$.

Forneceremos a função $\lambda(x, x * 5)$ explicitamente:

```
yes
| ?- intro(explicit(lambda(x,x*5))),display.
 f: [] partial autotactic(idtac)
==> f:int=>int
by intro(explicit(lambda(x,x*5)))

  [1] incomplete
  ==> lambda(x,x*5)in(f:int=>int)

yes
| ?- extract.
lambda(x,x*5)
```

Um novo nó ("[1]") foi gerado, com o sequente que exige a prova de que nosso λ-termo realmente habita aquele tipo.

◇

Neste exemplo, foi introduzido o λ-termo $\lambda x \cdot (x * 5)$, que passou a ser o termo associado a este nó da prova; o nó gerado (que ainda tem que ser provado) diz que $\lambda x \cdot (x * 5) \in$ `f:int=>int`, ou seja, que este λ-termo é uma função de *int* em *int*.

- *subst*: permite a substituição de múltiplas ocorrências de um termo na conclusão do sequente por outro;
- *level*: permite verificar ou determinar o nível de universo atual. Se `I` é uma variável Prolog e `i` é um número inteiro, `level(I)` instanciará `I` com o número do universo do nó corrente da árvore; `level(i)` instanciará o número do universo do nó da árvore com `i`.

2.4 Geração de Programas

A cada regra de inferência utilizada, o λ-termo vai sendo refinado, e ao final de uma prova, este termo pode ser executado, construindo os objetos relacionados à prova (em uma teoria intuicionista, todas as provas são construtivas).

Após provarmos um teorema que diz que para cada natural existe um sucessor, teremos um λ-termo que será exatamente a função sucessor para os naturais. Uma prova do teorema fundamental da álgebra nos dará um termo que, dado um inteiro, nos dará a decomposição deste em primos. Uma vez que os programas gerados são derivados de provas, eles sempre estarão corretos.

É claro que o uso da tática `because/0` deve ser criterioso, para permitir a geração de programas; o *extract term* associado a ela é uma string: `atom('incomplete')`.

Os programas gerados podem ser interpretados através do predicado `eval/2`, como podemos ver no exemplo 2.5.

Exemplo 2.5 O teorema do exemplo 2.4 define uma função $f(x) = x*5$; podemos usá-la:

```
| ?- extract.
lambda(x,x*5)

yes
| ?- eval(f(22),V).

V = 110 ?

yes
| ?-
```

◇

Capítulo 3

Clam

> *"The Master said, 'I would not have him to act with me, who will unarmed attack a tiger, or cross a river without a boat, dying without any regret. My associate must be the man who proceeds to action full of solicitude, who is fond of adjusting his plans, and then carries them into execution.' "*
>
> Confúcio (551 - 479 a.C.) - *Analecta*

Em [Bun91], Bundy argumenta que os sistemas de prova automática de teoremas não trabalham da mesma forma que os matemáticos fazem, e propõe um novo paradigma: o *planejamento de provas*. Neste paradigma, o raciocínio consiste em escolher entre vários métodos de prova, assemelhando-se mais ao raciocínio humano.

Clam [BvHHS90] é a implementação do conceito de planos de prova. Um plano de prova para um teorema é uma sequência de métodos determinada para este teorema que, quando aplicada em um ambiente de provas como o *Oyster*, deve produzir uma prova para o teorema. Programas também são gerados automaticamente, através do λ-termo associado às provas do *Oyster*.

3.1 Métodos

O processo de planejamento de provas no *Clam* é feito usando métodos. Um método é uma tupla que especifica uma tática:

- `name` – nome do método;
- `input` – sequente de entrada para o método;

```
method(identity,                    % Nome
       _ ==> G,                     % Entrada
       [matrix(_,X=X in _,G)],      % Pre-condicoes
       [],                          % Pos-condicoes
       [],                          % Saida
       identity).                   % Tatica
```

Figura 3.1: Exemplo de Método: identity/0

- `output` – lista de sequentes de saída;
- `preconditions` – lista de condições a serem satisfeitas para que o método seja aplicável;
- `postconditions` – lista de condições a serem satisfeitas após a aplicação do método;
- `tactic` – tática correspondente (i.e., o programa Prolog que realiza os passos da prova descritos pelo método).

Veremos agora alguns exemplos.

O método da figura 3.1 é usado para checar se o goal é do tipo $X = X\ in\ T$, onde T é um tipo. Para que o método seja aplicável, o sequente atual deve unificar com o sequente do slot de entrada, e a lista de pré-condições deve ser satisfeita. Neste caso temos uma única pré-condição, que diz que a matriz do sequente (que é o sequente sem quantificadores) deve ser do tipo "$X = X\ in\ T$" (na verdade, a matriz deve unificar com "$X = X\ in\ \_$"). O método não gera mais subgoals, logo as listas de pós-condições e saída são vazias. O último slot do método tem o nome da tática correspondente.

O nome do método na figura 3.2 é "apply_lemma", e a aridade (do método e da tática) é 1 – como se vê no slot de nome.

O sequente de entrada é `_==>G`, o que significa que a lista de hipóteses não será usada no resto da especificação.

A lista de pré-condições

1. o *Clam* faz anotações nos goals, como parte do processo de indução (veja seção 3.4.1), chamadas anotações de meta-nível. Este método toma o sequente, retira estas anotações:

```
method(apply_lemma(Lemma),                                    % Nome
       _==>G,                                                 % Entrada
       [strip_meta_annotations(G,GG),                         % Pre-condicoes
        precon_matrix(_,AllPre=>Concl,GG),
        append( _, Pre, AllPre ),
        precon_matrix( [], Pre=>Concl, Matrix ),
        theorem(Lemma,LemmaGoal),
        (instantiate(LemmaGoal,Matrix,_)
         v
         (Matrix=(L=R in T),instantiate(LemmaGoal,R=L in T,_))
        )
       ],
       [],                                                    % Pos-condicoes
       [],                                                    % Saida
       apply_lemma(Lemma)                                     % Tatica
      ).
```

Figura 3.2: Exemplo de Método: apply_lemma/1

strip_meta_annotations(G,GG)

2. divide o mesmo em duas partes, condições e conclusões:

   precon_matrix(_,AllPre=>Concl,GG)

3. Seleciona uma a uma as condições:

   append(_, Pre, AllPre)

4. Monta uma matriz com as cada uma destas condições separadamente:

   precon_matrix([], Pre=>Concl, Matrix)

5. verifica se há algum lema *Lemma*,

   theorem(Lemma,LemmaGoal)

6. tal que a matriz montada seja uma instância do *goal* de *Lemma*:

   (instantiate(LemmaGoal,Matrix,_)
   v
   (Matrix=(L=R in T),instantiate(LemmaGoal,R=L in T,_)))

As listas de pós-condições e de sequentes de saída são vazias – portanto é um método *terminante*.

O nome da tática normalmente é igual ao nome do método, como se pode verificar no slot correspondente.

3.1.1 Linguagem de Especificação dos Métodos

Pode-se usar qualquer predicado Prolog na especificação das condições, mas é desejável que haja uma linguagem uniforme para especificação destas condições. Esta linguagem é descrita no manual do *Clam* [HtDg97].

3.2 Planejamento de Provas

Para encontrar um plano de prova, o *Clam* faz uma busca entre os métodos aplicáveis ao sequente atual. Cada método aplicável tem uma lista de subgoals, para os quais o sistema deverá encontrar novos métodos. Esta busca pode ser feita em largura, profundidade ou usando uma heurística (a única heurística implementada é a de escolher o método que gera menos subgoals). Há vários algoritmos de busca implementados, cada um tendo dado origem a um planejador:

- `dplan` - busca em profundidade;
- `idplan` - busca por aprofundamento iterativo;
- `bplan` - busca em largura;
- `gdplan` - usa heurística "menor número de nós": os métodos que geram o menor número de *subgoals* serão selecionados primeiro.

Para cada um destes há outro que usa o conceito de sugestões (os "hint planners"): `dhplan`, `bhplan`, `gdhplan` e `idhplan`.

Os slots de entrada e pré-condições restringem o espaço de busca, amenizando a explosão combinatória. Os slots de saída e pós-condições determinam quais serão os novos objetivos após a aplicação do atual método.

O *Clam* não realiza a prova do teorema enquanto determina o plano; isto deve ser pedido explicitamente pelo predicado `apply-plan` – e é possível que alguma das táticas

descritas pelos métodos do plano falhe, e neste caso o teorema não é provado. Bundy [Bun88] enfatiza que esta incerteza é uma característica desejável do sistema: ele deve ser usado também em áreas indecidíveis, e a certeza do sucesso ou falha faria do plano um procedimento de decisão:

> On the other hand, success cannot be guaranteed. We will want to use plans in undecidable areas. If our ability to predict its success or failure was always perfect then the plan would constitute a decision procedure – which is not possible.

O sistema *Clam* foi projetado para ser um provador essencialmente indutivo de teoremas, sendo este um aspecto bastante nítido na biblioteca de métodos e linguagem de especificação dos métodos.

3.3 Reescrita de Termos

A reescrita de termos apareceu como uma técnica para provar teoremas em teorias equacionais [KB70]. É utilizada hoje em diversas áreas da computação [DJ90, Rin96].

Regras de reescrita são equações, usadas para substituir termos equivalentes em uma expressão; estas regras, porém, têm direção, e a reescrita só se dá na direção indicada. Representamos a reescrita por uma seta ($\rightarrow$); assim, $f(x) \rightarrow x + y$ significa que o termo $f(x)$ pode ser reescrito da forma $x + y$, de acordo com esta regra. Um conjunto de regras de resescrita é chamado de sistema de reescrita de termos. Se não há cadeias infinitas de reescrita ($x_1 \rightarrow x_2 \cdots$), o sistema é *terminante*. Se, ao aplicarmos repetidamente regras de reescrita a um termo, chegarmos a um elemento para o qual não há mais regras de reescrita, diz-se que este é a forma normal do primeiro. Quando a reescrita de termos iguais leva sempre à mesma forma normal, o sistema é dito *convergente*.

O *Clam* suporta Sistemas de Reescrita Condicional de Termos (CRS). Novas regras podem ser adicionadas ao sistema, o que resultará na criação, para cada definição que utiliza reescrita, de:

- um objeto do tipo **synth**: este objeto é um teorema *Oyster* que diz simplesmente que $\exists f : Dom \mapsto Cdom$, onde *Dom* e *Cdom* são domínio e contra-somínio da função sendo definida. O que interessa neste objeto é seu *extract term*, que deve ser construído pelo usuário de forma a implementar a função sendo definida.

Exemplo 3.1 Para a função x^2, o objeto `synth` pode ser o teorema $\forall i \in \mathbb{Z}\ \exists j \in \mathbb{Z}$. O extract term do teorema poderá ser: $\lambda(x, x * x)$. ◇

- vários objetos do tipo `eqn`, que são equações de recorrência (as regras de reescrita em si). Estes objetos são teoremas *Oyster* que justifiquam as respectivas regras de reescrita.

  Exemplo 3.2 O teorema $\forall n \in pnat\ times(n, 0) = 0$ dá origem à regra de reescrita $times(n, 0) \rightarrow 0$; ◇

- um objeto do tipo `def`, que relaciona conjuntos de equações com objetos `synth`. Estes objetos são da forma `NewObj(`$x_1, ..., x_n$`) <==>` `term_of(NewObj) of` x_1 `of` x_2 ... `of` x_n. Aqui, "term_of X" significa "o extract term do objeto `synth` chamado X.

Os dois primeiros são teoremas do *Oyster*, necessitando de prova.

Exemplo 3.3 A adição em $\mathbb{N}$ é definida por regras de reescrita no *Clam*:

- `synth(nat_plus)` é o teorema "$\exists f : \mathbb{N} \rightarrow \mathbb{N}$". O enunciado do teorema não diz nada sobre a função sendo implementada, mas a prova deste deve refletir a idéia da adição, produzindo um λ-termo que seja exatamente esta função (apresentando este termo, mostramos que o tipo `pnat => pnat` é habitado, provando o teorema).
- As equações são:
  - `nat_plus(0,x)=x`
  - `nat_plus(s(x),y)=s(nat_plus(x,y))`

  Estas equações dão origem a duas regras de reescrita.
- A definição é: `nat_plus(x,y) <==> term_of(synth(nat_plus)) of x of y`

◇

3.4 Rippling

Rippling é uma técnica de reescrita introduzida por Alan Bundy [BvHIS93], tendo sido aplicada com sucesso pelo sistema *Oyster-Clam* em provas indutivas envolvendo números naturais e listas.

Exemplo 3.4 Consideremos a seguinte regra de reescrita:

$$a + s(b) \rightarrow s(a + b)$$

onde s representa a função sucessor nos naturais. Ao encontrarmos, durante a prova de um teorema, a expressão $x + s(y - z)$, a regra é aplicável (com $a = x$ e $b = y - z$); após a aplicação da regra, teremos: $s(x + y - z)$. O efeito da aplicação é o de aumentar o escopo do símbolo de função s, como se ele se movesse para fora da expressão. ◇

Definição 3.1 Frentes de onda *são termos com subtermos próprios deletados. A parte da expressão que se move durante o processo é chamada* frente de onda. *No exemplo 3.4, "$s(\ )$".*

Definição 3.2 *Os subtermos deletados das frentes de onda são chamados* buracos de onda, *ou simplesmente buracos. No exemplo 3.4, "$y - z$".*

Definição 3.3 *Chama-se* esqueleto *da expressão tudo aquilo que não for frente de onda (inclusive buracos). No exemplo 3.4, o esqueleto é $x + y - z$.*

A notação gráfica usada para representar frentes de onda é um retângulo ao redor da mesma, com o buraco de onda sublinhado [Rei96]. Assim, a expressão do exemplo é: $x + \boxed{\text{s(y-z)}}$, transformada em $\boxed{\text{s(x+y-z)}}$. Estes símbolos são chamados *anotações*, e diz-se que a equação está *anotada*. A linguagem do *Clam* possui predicados específicos para anotar ou remover as anotações de uma expressão.

Para o exemplo a seguir, utilizamos a definição recursiva da adição em $\mathbb{N}$, mostrada no exemplo 3.3; as equações são:

$$\begin{aligned} 0 + y &= y; \\ s(x) + y &= s(x + y). \end{aligned}$$

Exemplo 3.5 Consideremos a associatividade sa soma:

$$\forall x, y, z \in \mathbb{N} x + (y + z) = (x + y) + z$$

Faremos uma prova será por indução em x. Hipótese de indução:

$$\forall x, \{y, z \in \mathbb{N}, x + (y + z) = (x + y) + z\}$$

Caso base:

$$(0 + y) + z = 0 + (y + z)$$

Por definição, $0 + y = y$, então a base é válida.

Passo de indução:

$$\begin{aligned} \forall x \{\forall y, z \in \mathbb{N}, x + (y + z) &= (x + y) + z \\ \supset s(x) + (y + z) &= (s(x) + y) + z\} \end{aligned}$$

Agora, aplicando a técnica de rippling, usaremos a hipótese várias vezes para provar que o teorema vale para $s(x)$. O sinal $\frown$ indica qual parte da equação foi alterada por rippling no passo anterior.

$$x + (y + z) = (x + y) + z \supset \boxed{\mathrm{s}(\underline{\mathrm{x}})} + (y + z) = (\boxed{\mathrm{s}(\underline{\mathrm{x}})} + y) + z$$

$$x + (y + z) = (x + y) + z \supset \overbrace{s(x + (y + z))} = \overbrace{\boxed{\mathrm{s}(\underline{\mathrm{x+y}})}} + z$$

$$x + (y + z) = (x + y) + z \supset s(x + (y + z)) = \overbrace{s((x + y) + z)}$$

Que é claramente verdadeiro. ◇

3.4.1 Variações de Rippling

Há diversas categorias de rippling [BvHIS93], caracterizadas pela maneira como a frente de onda se move no esqueleto da expressão. Listamos aqui algumas delas:

- Rippling-out: quando a frente de onda move-se para fora da expressão. Usa-se uma seta para cima para indicar que a frente de onda deverá mover-se desta maneira: $f(\boxed{\mathrm{g}(\underline{\mathrm{x+y}})}^{\uparrow})$

- Rippling-in: quando a frente de onda move-se para dentro da expressão, e seu escopo diminui. Neste caso, o subtermo para onde a frente de onda converge é chamado *ralo*[1]. Os ralos são delimitados por $\lfloor\ \rfloor$: $\boxed{\mathrm{f}(\underline{\mathrm{g}(\lfloor x+y \rfloor)})}^{\downarrow}$
- Rippling-sideways: quando a frente de onda move-se lateralmente na equação;
- Rippling-existential: quando a frente de onda atravessa um quantificador existencial;
- Outros: rippling-across, coloured rippling, rippling funcional.

A escolha da variável de indução e as anotações são feitas através de *difference matching*, um algoritmo que unifica hipótese e conclusão da indução.

O uso da hipótese de indução para chegar à tautologia na conclusão é chamado de *fertilização*.

3.5 Críticos de Divergência

O planejamento de provas usando rippling pode divergir, com a aplicação de regras sem que se chegue a um fim. Muitas vezes, a sugestão de um lema ou a generalização podem resolver este problema. É isto que fazem os críticos de divergência apresentados por Walsh [Wal96]. Um crítico de divergência pode propor um lema que permita a convergência do processo de planejamento; quando o crítico detecta este tipo de problema, o planejamento pára, um lema é sugerido e (possivelmente) provado; só então o planejamento volta ao curso normal.

Exemplo 3.6 As regras de reescrita abaixo são derivadas da definição da função *dbl* [DP90], que calcula o dobro de um número natural:

$$\begin{aligned} x+0 &\rightarrow x; \\ x+s(y) &\rightarrow s(x+y); \\ dbl(0) &\rightarrow 0; \\ dbl(s(x)) &\rightarrow s(s(dbl(x))). \end{aligned}$$

Agora, tentaremos provar o teorema

$$\forall n \in I\!N, \mathrm{dbl}(n) = n + n.$$

[1] "sink", no original

Ao tentar provar este teorema por indução, o passo será:

$$\text{dbl}(n) = n + n \supset \text{dbl}(s(n)) = s(n) + s(n)$$

Após o rippling, teremos:

$$\text{dbl}(n) = n + n \supset s(s(\text{dbl}(n))) = s(s(n) + n)$$

Cancela-se o s externo nos dois lados da conclusão:

$$\text{dbl}(n) = n + n \supset s(\text{dbl}(n)) = s(n) + n$$

E o sistema fertiliza o lado esquerdo da conclusão com a hipótese:

$$\text{dbl}(n) = n + n \supset s(n + n) = s(n) + n$$

E não há mais como simplificar. O sistema tentará uma nova indução, que levará à divergência:

$$\begin{aligned} s(n+n) &= s(n) + n \\ s(s(n+n)) &= s(s(n)) + n \\ s(s(s(n+n))) &= s(s(s(n))) + n \\ &\vdots \end{aligned}$$

A existência de um lema $s(x) + y = s(x + y)$ poderia resolver o problema, levando à identidade $s(x + x) = s(x + x)$. Este lema é sugerido por um crítico de divergência. ◇

O *Clam* tem um crítico de divergência; os lemas são propostos através de *difference matching.*

3.6 Analogias e Meta-métodos

Descrita por Huang e Kerber [HK95], a técnica de planejamento analógico de provas foi inicialmente implementada no sistema $\Omega - MKRP$ [HKK+94], e é essencialmente a aplicação de raciocínio baseado em casos ao planejamento de provas. Usa-se rascunhos de provas antigas bem-sucedidas para guiar a elaboração de um novo plano de prova.

No $\Omega - MKRP$, o slot de tática dos métodos não contém apenas predicados prolog (um programa), como no *Clam*; contém um rascunho da prova a ser realizada pela tática,

e um interpretador para este rascunho. Ao planejar uma nova prova, o sistema verifica se o teorema não se parece com algum já provado; se for o caso, toma o métoto usado no outro teorema e o adapta para o novo teorema (isto é facilitado pela natureza declarativa das táticas). Assim, um novo método será criado; esta adaptação é feita por um método chamado *meta-método*. A versão 3 do *Clam* possui mecanismo de analogia [Mel95], e tem os slots dos métodos diferentes dos slots das versões anteriores.

Capítulo 4

Reescrita de Tipos e Diagonalização

"The whole is more than the sum of its parts."
Aristóteles (384 - 322 a.C.)

4.1 O processo de Diagonalização

Diagonalização é uma técnica usada para provar teoremas do tipo:

$$\neg surj(f, \alpha, \beta) \tag{4.1}$$

$$greater\_than(\alpha, \beta) \tag{4.2}$$

Ou seja, (4.1) "a função $f : \alpha \to \beta$ não é sobrejetora, e (4.2) "a cardinalidade de α é maior que a de β".

Alguns teoremas como estes são:

- Indecidibilidade do problema da parada;
- $\forall x |\wp(x)| > |x|$;
- Existência de uma função não-computável;
- $|\mathbb{N}| < |\mathbb{R}|$;

e muitos outros em teoria da computabilidade [Cut92].

Normalmente, a função f que conseguiremos provar não ser sobrejetora é de um conjunto qualquer α em um conjunto de funções β, onde β é $\alpha \mapsto \gamma$[1]. A diagonalização consiste em construir uma função δ que não esteja na imagem de f, mas esteja em seu contradomínio.

Na figura 4.1, a sequência de todos os elementos de α formam a coluna à esquerda, e a linha acima da figura. Dentro da figura, temos elementos de γ.

Seja f uma função de α em um contra-domínio β. Sabemos expressar β como um conjunto de funções de α em γ. A função $f : \alpha \mapsto \beta$ leva elementos da coluna à esquerda a funções do contra-domínio; estas são, por sua vez, funções que levam elementos da linha superior à matriz $(\alpha \mapsto \gamma)$. Logo, a função é $f : \alpha \mapsto (\alpha \mapsto \gamma)$. Pode parecer que estamos misturando o uso de $\mapsto$, usando o símbolo tanto para especificar domínio e contra-domínio, como para indicar que β é um conjunto de funções. Mas a notação usada pelo *Oyster* é esta mesma. Tal função seria: `f:(alpha=>alpha=>gamma)`

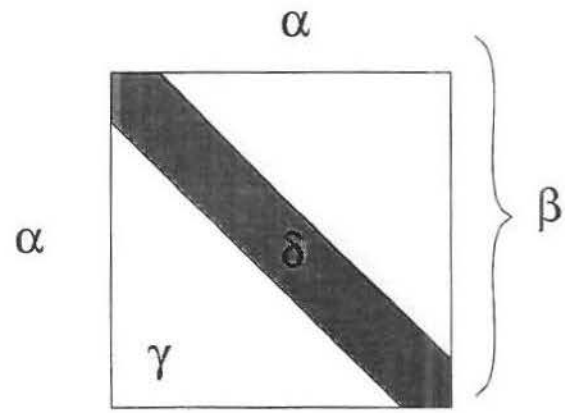


Figura 4.1: Diagonalização

Para provar (4.1) por diagonalização: se $|\gamma| > 1$, podemos construir uma função *diff* que, para cada elemento de γ, retorne outro diferente. Agora, seja δ igual a $\lambda x.diff(f(x)(x))$. O elemento δ é uma função de α em γ (pertence ao contra-domínio de f), mas não pertence à imagem de f. Se δ estivesse na imagem de f, estaria em uma das linhas da figura 4.1 – mas δ não se encaixa em nenhuma linha da figura porque difere em um elemento em cada linha (difere no i-ésimo elemento na i-ésima linha).

Exemplo 4.1 Ao provar o teorema "$\forall x, \forall f : x \mapsto \mathcal{P}(x)$, f não é sobrejetora" ($\mathcal{P}(s)$ é o conjunto das partes de s), teríamos $\alpha = x, \beta = \mathcal{P}(x), \gamma = \{0, 1\}$. ◇

[1]Usaremos a notação $A \mapsto B$ para o conjunto de todas as funções de A em B, normalmente denotado por B^A. A notação que usaremos está mais próxima da usada pelo *Oyster*

4.1.1 Generalização da Diagonalização

Para que a diagonalização seja possível, o contradomínio de f (o conjunto β) não precisa ser exatamente $\alpha \mapsto \gamma$, mas deve ser da forma $\theta \mapsto \gamma$; deve haver um mapeamento τ: $\alpha \mapsto \theta$ (τ deve ser uma injeção), para podermos encontrar a diagonal, e deve haver uma função de modificação *diff*: $\gamma \mapsto \gamma$ tal que $\forall g \in \gamma, diff(g) \neq g$.

O elemento δ será:

$$\delta = \lambda x.diff(f(x)(\tau(x)))$$

Precisamos, portanto, das funções τ e *diff*. A função *diff* pode ser facilmente determinada para conjuntos bem-ordenados (em especial para conjuntos infinitos), sendo $diff(x)$ o sucessor imediato de x.

Quanto à possibilidade da diagonal ser encontrada, há 3 casos:

- $|\alpha| = |\theta|$. Neste caso (figura 4.2), encontraremos δ sem problemas; O mapeamento τ será uma bijeção entre α e θ.

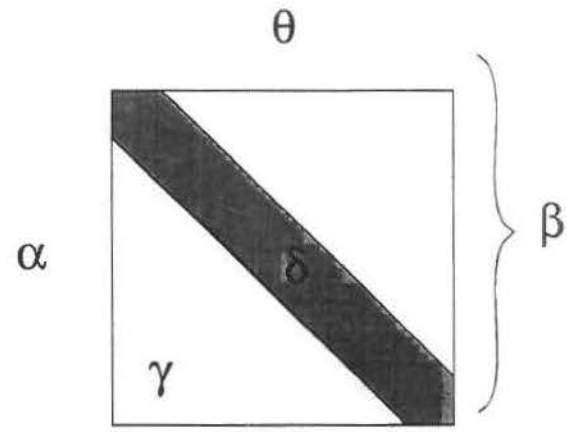


Figura 4.2: Diagonalização: caso $|\alpha| = |\theta|$

- $|\alpha| < |\theta|$. Neste caso (figura 4.3), teremos apenas uma parte de δ, que será suficiente para a prova. τ pode ser qualquer função injetora de α em θ.

- $|\alpha| > |\theta|$. Neste caso (figura 4.4), τ não pode ser injeção, e a prova não é possível.

Se quisermos diagonalizar em funções onde o contra-domínio é um subconjunto de um conjunto do tipo $\theta \mapsto \gamma$, precisaremos, além de todos os passos descritos acima, provar que o elemento encontrado pertence ao contra-domínio da função.

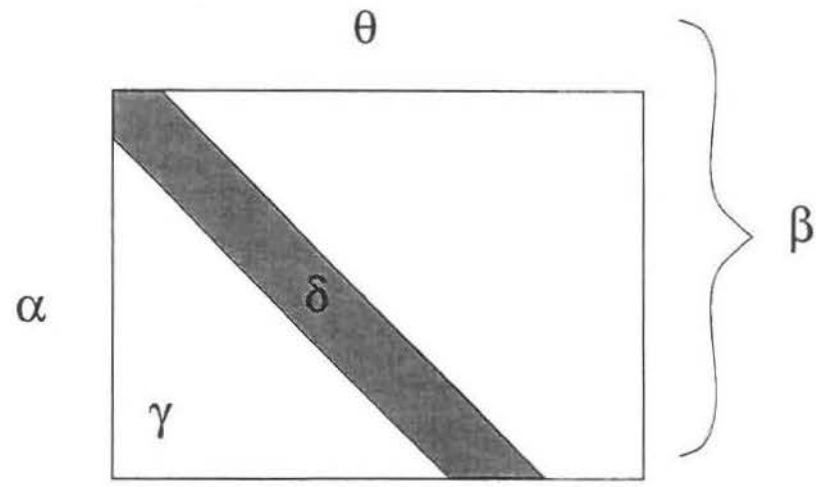


Figura 4.3: Diagonalização: caso $|\alpha| < |\theta|$

Exemplo 4.2 Não há função sobrejetora de $\{0,1,2,3\}$ em $\{<0,0,0,0>,<0,0,0,1>,$ $<0,0,1,0>,<0,0,1,1>,<0,1,0,0>\}$, pois a cardinalidade do primeiro conjunto é menor que a do segundo. Temos na figura 4.5 uma tentativa de provar por diagonalização. Consideraremos os vetores como funções de $\{0,1,2,3\}$ em $\{0,\ 1\}$. Note que no contra-domínio de f não temos todas as funções de um conjunto em outro, mas apenas um subconjunto delas. Mas o elemento encontrado invertendo a diagonal é $<1,1,0,0>$, que não pertence ao contra-domínio de f. Um elemento que serviria de contra-exemplo é: $<0,1,0,0>$, mas este não pode ser encontrado por diagonalização, nem mesmo permutando linhas ou colunas (no contra-exemplo há três zeros, e não há linha com três uns na matriz). ◇

Nas próximas seções, descrevemos como o processo de diagonalização foi implementado no sistema *Oyster-Clam*, usando reescrita de tipos.

4.2 Nossa Abordagem

Por ser este sistema um provador essencialmente indutivo, a implementação da diagonalização será mais difícil que incluir novos esquemas de indução, por exemplo.

Mostramos dois tipos de conjectura no início deste capítulo (4.1 e 4.2). Desenvolvemos um método *Clam* que reduz conjecturas como estas a uma afirmação do tipo $\exists diff : \gamma \mapsto \gamma, \forall c \in \gamma, diff(c) \neq c$.

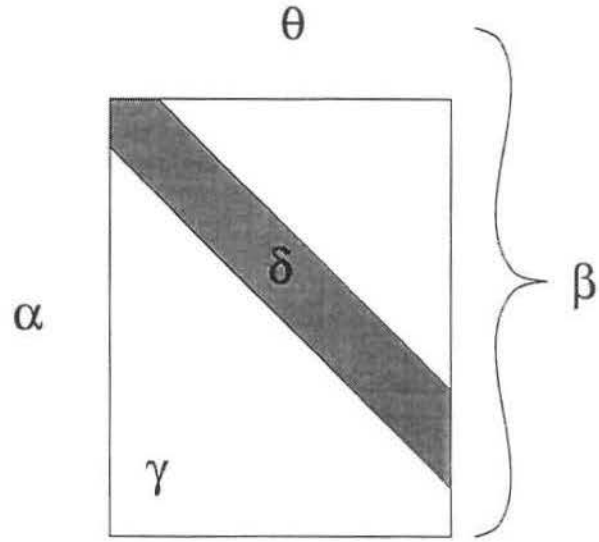


Figura 4.4: Diagonalização: caso $|\alpha| > |\theta|$

	0	1	2	3
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1

$\delta =< 1, 1, 0, 0 >$

Figura 4.5: Tentativa de diagonalização

Nosso método lida com a forma expandida de goals do tipo (4.1), e outros, como $\theta \sim (\alpha \mapsto \gamma) \rightarrow \forall f : \alpha \mapsto \theta, \exists \delta : \alpha \mapsto \gamma, f(a) \neq \delta$, onde $\sim$ denota equipotência.

Para goals da forma (4.2), uma única regra de reescrita resolve o problema:

$$greater\_than(\alpha, \theta) \rightarrow \forall f : \alpha \mapsto \theta, \neg surj(f, \alpha, \theta)$$

Esta regra os porá na forma (4.1).

Outras formas de provar que um conjunto (tipo) é maior que outro podem ser utilizadas, bastando incluir uma nova regra de reescrita para *greater_than*.

Tendo desenvolvido este método, precisamos cuidar dos *goals* da forma $\exists diff : \gamma \mapsto \gamma, \forall c \in \gamma, diff(c) \neq c$. Para isto, desenvolvemos regras de reescrita, que tentarão resuzir a um *goal* elementar - normalmente $s(x) \neq x$ ou $x + 1 \neq x$, que o *Clam* pode provar. Se γ

for um tipo unário (e neste caso a conjectura não é verdadeira), o *Clam* não achará regras de redução para a função *diff*. Estas regras serão mostradas mais adiante.

4.2.1 Modificações no *Oyster*

Uma versão do *Oyster* [HS94] suportava indução em tipos. Não se sabe, no entanto, se chegou a ser utilizada com algum planejador de provas. A versão atualmente distribuída (Release 1) não suporta este princípio.

As regras de inferência do *Oyster* são implementadas através do predicado `rule`; para adicionar uma nova regra, basta escrever uma nova cláusula para este predicado.

Nós incluímos algumas regras de inferência no *Oyster*, para permitir que uma regra *elim* seja aplicada a qualquer variável do tipo $u(i)$ na lista de hipóteses de um sequente, gerando diversos subgoals, um para cada tipo, e um termo de indução em tipos como *extract term*.

$$\frac{\overbrace{\Gamma \cup \{x_1 : x = int\} \vdash \phi}^{A}, \Gamma \cup \{x_2 : x = pnat\} \vdash \phi, \cdots, \overbrace{\Gamma \cup \{T : u(1), x_n : x = T\ list\} \vdash \phi}^{B}}{\Gamma \cup \{x : u(1)\} \vdash \phi}$$

Note que a regra é semelhante à de #-eliminação. Na parte A, a regra diz que se x for o tipo *int*, ϕ pode ser deduzido. Isto é feito para todos os tipos, inclusive os tipos compostos, como na parte B, onde temos "Dado um tipo T em $u(1)$, se x for o tipo T *list*, devemos poder deduzir ϕ. Se isto valer para todos os tipos, podemos dizer que "para qualquer tipo em $u(1)$, ϕ vale" - que é exatamente o que diz a conclusão da regra.

O λ-termo resultante da aplicação desta regra é:

$$it\_ind(V, Tvoid, Tunary, Tint, Tpnat, Tatom, L)$$

V é a variável de recursão; se V for um dos tipos básicos, o valor será o das variáveis listadas (quando V=unary, o λ-termo acima vale *Tunary*, por exemplo), e se V for composto, a parte recursiva da definição está na lista L:

```
L = [
      Type1,Type2,

      List,
      Tlist,
```

```
    Disj1,
    Disj2,
    Tdisj,

    Fun1,
    Fun2,
    Tfun,

    Prod1,
    Prod2,
    Tprod

:

]
```

Onde *Tlist*, *Tdisj*, *Tfun* e *Tprod* são o valor definido para o termo, dependendo dos valores das outras variáveis (ver definições recursivas e indução no *Oyster*). Para as listas, por exemplo, a regra de inferência diz que temos que ter um elemento do tipo *T list*; sendo o raciocínio para trás, deveremos produzir este elemento. Assim, dado o elemento *List*, o termo valerá *Tlist*.

Como é possível executar o λ-termo derivado das provas no *Oyster*, foi necessário escrever também cláusulas para o predicado `eval`, responsável pela interpretação de λ-termos.

Também incluímos regras e operadores de decisão (como mostramos no exemplo 4.3). Estas modificações permitem a implementação das regras de reescrita para tipos, descritas na próxima seção.

Exemplo 4.3 Se tivermos um sequente: $\{v0 : u(1)\} \vdash P$, podemos aplicar a regra `decide(v0=int in u(1))`; quatro subgoals serão gerados: $vo = int$ e $vo \neq int$, mais dois de *well-formedness*. O *extract term* gerado será `type_eq(v0,int,_,_)`, um procedimento de decisão que compara $v0$ com *int* e executa um de dois outros *extract terms*:

```
| ?- display.
u1: [1] incomplete autotactic(idtac)
1. v0:u(1)
==> u(1)
```

```
by _

yes
| ?- decide(v0=int in u(1)),display,extract.
u1: [1] partial autotactic(idtac)
1. v0:u(1)
==> u(1)
by decide(v0=int in u(1))

  [1] incomplete
  2. v1:v0=int in u(1)
  ==> u(1)

  [2] incomplete
  2. v1:v0=int in u(1)=>void
  ==> u(1)

  [3] incomplete
  ==> v0 in u(1)

  [4] incomplete
  ==> int in u(1)

type_eq(v0,int,_,_)

yes
| ?-
```

O que fizemos foi simplesmente iniciar uma prova por casos: sabemos que há um elemento que habita $u(1)$. Separamos dois casos - o deste elemento ser *int*, e o de não o ser. Para cada um dos dois casos, o *goal* continua o mesmo. ◇

4.2.2 A Função *diff*

É claro que a função *diff* não pode ser definida para um tipo específico; o conjunto γ pode ser um tipo razoavelmente complexo. A função *diff* é dependente do tipo γ. Sendo necessário o raciocínio sobre tipos, optamos por fazê-lo com regras de reescrita, que farão a análise de casos necessária. Entre as regras usadas na definição de *diff* temos, por

exemplo:

$$\begin{aligned} \mathit{diff}(x, T\ \mathit{list}) &\rightarrow \mathit{element}(T) :: x \\ \mathit{diff}(x, \mathit{int}) &\rightarrow x + 1 \end{aligned}$$

ou seja, para o caso da função ser aplicada a uma lista, devemos simplesmente concatenar um novo elemento à lista dada - o que é feito pela função *element*, definida também com regras de reescrita para diversos tipos. Já no caso de um número inteiro, somamos 1 ao mesmo.

O sistema de reescrita de termos do *Clam* foi implementado originalmente de forma a lidar apenas com termos dos tipos *int*, *pnat*, e listas destes. O método `sym_eval` usa as regras de reescrita disponíveis. Com tipos especificamente, este método funciona corretamente, mas sua tática falha, não executando a reescrita. Contudo, tipos são termos sobre universos, e tendo extendido o *Oyster* para trabalhar com tipos como termos, melhoramos também as táticas do *Clam*, especificamente as táticas de *well-formedness* usadas pela tática `sym_eval`. Tendo feito isto, pudemos adicionar regras de reescrita para tipos no *Clam*.

No *Clam*, as definições são implementadas com regras de reescrita. As regras condicionais são da forma $\mathit{Cond} \supset \{A \rightarrow B\}$, significando que *Cond* deve valer para que a regra seja aplicada. Assim, definimos a função *diff* com as regras mostradas a seguir. O uso de reescrita de tipos é bastante clara: $\mathit{element}(x\mathit{list}) \rightarrow \mathit{nil}$ significa "$\forall x \in u(1)$", `element(x list) :=>nil`, i.e., para qualquer tipo em $u(1)$, uma lista de elementos daquele tipo pode ser a lista vazia.

$$\begin{aligned} \mathit{element}(\mathit{int}) &\rightarrow 0 \\ \mathit{element}(\mathit{pnat}) &\rightarrow 0 \\ \mathit{element}(\mathit{atom}) &\rightarrow \mathit{atom}(a) \\ \mathit{element}(x\ \mathit{list}) &\rightarrow \mathit{nil} \\ \mathit{element}(\mathit{unary}) &\rightarrow \mathit{unit} \\ \mathit{element}(t\#t2) &\rightarrow (\mathit{element}(t) \& \mathit{element}(t2)) \\ \mathit{element}(t\backslash t2) &\rightarrow \mathit{inr}(\mathit{element}(t2)) \\ \mathit{element}(t\backslash t2) &\rightarrow \mathit{inl}(\mathit{element}(t)) \end{aligned}$$

$$\mathit{diff}(x, \mathit{int}) \rightarrow x + 1$$

$$
\begin{aligned}
diff(x, pnat) &\rightarrow s(x) \\
diff(atom(a), atom) &\rightarrow atom(b) \\
(x = atom(a) \mapsto void) \supset \{diff(x, atom) &\rightarrow atom(a)\} \\
diff(x, t\ list) &\rightarrow element(t) :: x \\
diff(e\&e2, t\#t2) &\rightarrow (diff(e, t)\&diff(e2, t2)) \\
diff(inl(e), t\backslash t2) &\rightarrow inr(element(t2)) \\
diff(inr(e), t\backslash t2) &\rightarrow inl(element(t))
\end{aligned}
$$

Nas duas últimas regras, *inl* e *inr* são usados para indicar a qual dos dois tipos de $A\backslash B$ o elemento pertence. Assim, quando $inl(x)$ é um elemento habitando $A\backslash B$, x é necessariamente do tipo A. Se $inr(y) \in A\backslash B$, y é do tipo B.

A função *element* permite a prova de goals da forma $\exists x, x \in T$, que o *Clam* só prova em poucos casos.

Note que *diff* não está definida para todos os tipos possíveis. No caso de *void* e *unary* isto é intencional, pois então não deve haver regras que permitam reescrever *diff*.

No caso de outros tipos, decidimos não implementar regras (apesar de acreditarmos que seriam úteis). É provável que não seja fácil, por exemplo, achar regras para os tipos *subconjunto*.

Além da diagonalização, a reescrita de tipos é claramente útil em outras situações; a função element é útil para provar teoremas quantificados existencialmente, e em base de indução.

4.3 Alguns Teoremas Provados

Os teoremas a seguir foram provados pelo *Clam*, usando o método e as regras de reescrita mostrados:

- *Powerset*: O método foi suficientemente genérico para provar tanto este teorema ($\forall x, |\mathcal{P}(x)| > |x|$) como algumas instâncias ($|\mathcal{P}(int)| > |int|$ e outras). Aqui, $\mathcal{P}(x) = x \mapsto \mathcal{B}$, onde $\mathcal{B}$ é um conjunto (tipo) binário. Usamos *unary* \ *unary* como $\mathcal{B}$; a escolha de outros tipos para $\mathcal{B}$ dificilmente influiria no sucesso da prova (há apenas o problema dos casos em que *diff* não está definida).

- *Existência de Função não Computável*: "Dada uma enumeração $\mathcal{E}$ de todas as funções computáveis, existe uma função δ não computável". Este teorema tem como *extract term* uma função que, dada a enumeração $\mathcal{E}$, retorna a função não computável δ.

- $|\mathbb{N}| < |\mathbb{R}|$: Este teorema e uma variante, $\forall f : \mathbb{Z} \mapsto \mathbb{R}, \neg surj(f, \mathbb{Z}, \mathbb{R})$, foram facilmente provados pelo *Oyster-Clam*.

4.4 Trabalhos Relacionados

Outros sistemas, como *TPS* [ABI+94] e *NQTHM* [BM79b] realizam provas por diagonalização. O *Coq* [HKPM97] e o *Isabelle* [PN90, Pau93] suportam raciocínio sobre tipos; nenhum destes, no entanto, é baseado em planejamento de provas.

Até onde sabemos, há apenas um sistema baseado em planejamento de provas que realiza provas por diagonalização: o Ω-*MKRP* [HKK+94, HKC95]. O método usado neste sistema para realizar estas provas [HK95] é o do planejamento com analogias. Este método implica no uso de meta-métodos, e na construção de novos métodos. Isto pode ser dispendioso. Apesar de apontarmos para esta vantagem da nossa abordagem, não é nossa intenção invalidar o uso de analogias em planejamento de provas: trata-se de uma técnica poderosa, que foi recentemente incorporada ao *Clam* [Mel95], e esperamos que outros sistemas passem a utilizá-la.

Capítulo 5

Provas Indutivas em Novas Estruturas

"Por que cometer erros antigos se há tantos erros novos a escolher?"

Bertrand Russell (1872-1970)

O *Clam* possui esquemas de indução para números inteiros, naturais e em listas. Para permitir o uso de novos esquemas de indução, basta:

- formalizar o esquema de indução na linguagem do *Oyster*;
- provar um teorema justificando o esquema (o *Clam* pode usar esquemas de indução que não sejam provados corretos, mas isto não é aconselhável, e produzirá um aviso sempre que o esquema for carregado);
- escrever uma nova cláusula para a tática que aplica indução, para poder fornecer o λ-termo associado às provas indutivas que usarem o novo esquema;

5.1 Definições Indutivas e Regras de Reescrita

Os esquemas de indução estão relacionados com definições indutivas, e sempre que as tivermos poderemos escrever um esquema equivalente.

O exemplo mais simples de definição indutiva no *Clam* é o de soma:

$$\begin{aligned} plus(0, x) &\rightarrow 0 \\ plus(x, s(y)) &\rightarrow s(plus(x, y)) \end{aligned}$$

A soma ("plus") tem duas regras de reescrita. Uma definição indutiva pode ter quantas equações forem necessárias, dando origem a muitas regras de reescrita. A definição de número de Fibonacci tem três equações, que dão origem a três regras de reescrita.

Quando as regras de reescrita são condicionais formando um conjunto complementar, o *Clam* usará este conjunto para realizar provas por casos (se necessário).

5.2 Grafos

A teoria de grafos é reconhecidamente de grande importância [BM79a], tendo ampla gama de aplicações. A possibilidade de provas automáticas de teoremas em grafos, ainda que simples, é muito interessante. A geração automática de programas corretos que resolvam problemas em grafos faz desta possibilidade algo ainda mais almejável.

A formalização de alguns conceitos da teoria de grafos é apresentada aqui. Trataremos grafos como pares $< V, E >$, onde V é um conjunto de vértices e E é o conjunto de arestas. Em grafos orientados, teremos uma aresta como um par ordenado $< u, v >$, e em grafos não orientados, como conjuntos binários $\{u, v\}$. No *Oyster*, usamos a seguinte definição para o tipo *graph*:

$$graph \stackrel{def}{\Longleftrightarrow} int\ list\#(int\#int)list$$

ou seja, uma lista de inteiros (os vértices) e uma lista de pares de inteiros (as arestas).

Apresentamos a seguir algumas definições indutivas para grafos orientados; a definição equivalente para grafos não orientados é óbvia.

Primeira definição:

- Um vértice é um grafo ($Gr[< v, \emptyset >]$);
- Um conjunto de vértices com um novo vértice em V é um grafo ($Gr[< V, \emptyset >] \supset Gr[< V \cup \{v\}, \emptyset >]$);
- Um grafo com uma nova aresta ligando dois de seus vértices é um grafo ($Gr[< V, E >] \supset \forall u, v \in V\ Gr[< V, E \cup \{< u, v >\} >]$).

Segunda definição:

- Um conjunto não vazio de vértices é um grafo ($Gr[< V, \emptyset >]$);
- Dois conjuntos de vértices, quando postos um ao lado do outro formam um grafo ($Gr[< V_1, \emptyset >] \wedge Gr[< V_2, \emptyset >] \supset Gr[< V_1 \cup V_2, \emptyset >]$);
- Um grafo com uma nova aresta ligando dois de seus vértices é um grafo ($Gr[< V, E >] \supset \forall u, v \in V Gr[< V, E \cup \{< u, v >\} >]$).

Traduziremos as definições para um esquema de indução:

Primeiro esquema de indução:

- Base: $\phi[< v, \emptyset >]$;
- Passo: $\phi[< V, \emptyset >] \supset \phi[< V \cup \{v\}, \emptyset >]$;
- Passo: $\phi[< V, E >] \supset \forall u, v \in V \phi[< V, E \cup \{< u, v >\} >]$.

Segundo esquema de indução:

- Base: $\phi[< v, \emptyset >]$;
- Passo: $\phi[< V_1, \emptyset >] \wedge \phi[< V_2, \emptyset >] \supset \phi[< V_1 \cup V_2, E_1 \cup E_2 >]$;
- Passo: $\phi[< V, E >] \supset \forall u, v \in V \phi[< V, E \cup \{< u, v >\} >]$.

Ao provar a base e os dois passos para uma proposição ϕ qualquer, provamos que ϕ vale para qualquer grafo.

Para fins de implementação no sistema *Oyster-Clam*, representaremos conjuntos como listas de átomos (strings).

Primeiro esquema de indução:

```
scheme([_::_,_::_],indgraf,
[
 [U:int]                                ==>phi(U::nil,nil),    % Base: grafo trivial
 [U:int,V:int list,phi(V,nil)]          ==>phi(U::V,nil),      % Passo: vertice
 [V:int list,E:(int#int list),
  phi(V,E),Ed:int#int,member(Ed,V)] ==>phi(V,Ed::E)        % Passo: aresta
]
==>phi( Vertices:(int list) , Edges:(int#int) list ).
```

Segundo esquema de indução:

```
scheme([_::_,_::_],indgraf_2,
[
 [U:int]                                ==>phi(U::nil,nil),      % Base: grafo trivial
 [Va:int list,Vb:int list,
  phi(V,nil)]                            ==>phi(app(Va,Vb),nil), % Passo: uniao
 [V:int list,E:(int#int list),
  phi(V,E),Ed:int#int,member(Ed,V)] ==>phi(V,ED::E)          % Passo: aresta
]
==>phi( Vertices:(int list) , Edges:(int#int) list ).
```

Este esquema usa duas definições, a de concatenação de listas (app) e a de membro de lista (member).

Os dois esquemas apresentados diferem apenas no passo dos vértices; nos teoremas mostrados, ambos puderam ser aplicados, apenas apresentando provas diferentes. É importante notar que provas diferentes dão origem a *extract terms* diferentes, que realizam a mesma tarefa. A eficiência dos λ-termos derivados de diferentes esquemas de indução foi estudada por Madden [Mad91].

Além dos esquemas, são necessárias algumas regras de reescrita para que a definição seja útil durante o planejamento de provas. As definições que implementamos são:

- *Member* - pertinência para listas de inteiros - $member(el, l)$ verifica se el é membro de l.
- *App* - $app(l1, l2)$ concatena de duas listas de inteiros $l1el2$.
- *Ins-Vtx* - $ins\_vtx(v, e)$ innsere um vértice v em uma lista e.
- *New-V* - $new\_v(v, g)$ insere novo vértive v em um grafo g.
- *Union* - $union(a, b)$ é a união de 2 conjuntos A e B (implementados como listas).
- *Ed-Member* - $ed\_member(e, u, v)$ verifica se $< u, v > \in e$.
- *Ins-Ed* - $ins\_ed(e, u, v)$ insere aresta $< u, v >$ na lista e.
- *New-E* - $new\_e(g, u, v)$ insere aresta $< u, v >$ em um grafo g.
- *Rm-Edge* - $rm\_edge(u, v, e)$ remove aresta $< u, v >$de uma lista e.

- *In-Degree* - $in\_degree(e, v)$ é o grau entrando do vértice v em uma lista e de arestas.
- *Out-Degree* - $out\_degree(e, v)$ é o grau saindo do vértice v em uma lista e de arestas.
- *Degree* - $degree(e, v)$ é o grau do vértice v em uma lista e de arestas.
- *Path* - $path(u, v, V, E)$ acha um caminho de u a v, se possível; as regras são mostradas abaixo. A quarta regra não é necessária, por ser caso especial da segunda, mas pode acelerar o processo.

$$\begin{aligned} path(B, B, C, D) &\rightarrow nil \\ out\_degree(B, C) = 0 \; in \; int \supset \{path(C, D, E, B) &\rightarrow nil\} \\ ed\_member(B, C, D) \supset \{path(C, D, E, B) &\rightarrow (C\&D) :: nil\} \\ path(B, C, D, nil) &\rightarrow nil \\ path(B, C, D, E) \neq nil \supset & \\ \{path(F, C, D, (F\&B) :: E) &\rightarrow (F\&B) :: path(B, C, D, E)\} \end{aligned}$$

Há muitas outras possibilidades de indução em grafos, sendo interessante capturar as mais comuns.

5.2.1 Graus de Vértices

Alguns teoremas sobre graus dos vértices de grafos foram provados usando os conceitos anteriores.

Exemplo 5.1

$$\forall\, G(V, E), \quad \sum_{v \in V} d^+(v) - d^-(v) = 0$$

onde $d^+(v)$ é o grau saindo de v (número de arestas que saem de v), e $d^-(v)$ é o grau entrando em v.

O *Clam* prova o teorema por indução. A base é o grafo trivial (um vértice isolado), e os passos consistem em colocar vértices (que não influem na soma dos graus), e arestas (mas para cada aresta tanto d^+ como d^- serão incrementados de 1. ◇

Exemplo 5.2 Em grafos não dirigidos, temos outro teorema envolvendo somatório dos graus:

$$\forall\, G(V,E),\quad \sum_{v \in V} d(v) \text{ é par}$$

A prova feita pelo *Clam* é análoga à anterior. ◇

5.2.2 Coloração de Arestas

O conceito de grau, definido na seção anterior, é um passo na direção da prova de teoremas mais interessantes, como o teorema de Vizing, por exemplo:

$$\forall G(V,E),\ \mathit{simples}(G) \supset \chi'(G) = \Delta \quad \mathit{ou} \quad \chi'(G) = \Delta + 1$$

Onde Δ é o maior grau de um vértive do grafo, e χ' é o menor número de cores necessário para colorir as arestas (sem que duas arestas adjacentes tenham a mesma cor).

Ao colorirmos uma aresta $< u, v >$, devemos escolher uma cor diferente das usadas nas arestas incidindo em u e v. Dadas uma lista de cores l e uma lista de cores já usadas lu, a seguinte definição permite escolher uma cor não usada ainda:

$$\begin{aligned} \mathit{pick\_color}(\mathit{nil}, lu) &\rightarrow 0 \\ \mathit{member(e,lu)} \supset \{\mathit{pick\_color}(e :: l, lu) &\rightarrow \mathit{pick\_color(l,lu)}\} \\ \mathit{pick\_color}(e :: l, lu) &\rightarrow e \end{aligned}$$

Assim, podemos usar a próxima definição para colorir as arestas; a definição a seguir, *ed_coloured*, diz que uma aresta já foi colorida - e diz como fazê-lo.

$$\begin{aligned} \mathit{pick\_color}(c, \mathit{union}(\mathit{incident}(u), \mathit{incident}(v))) &= 0 \supset \\ \{\mathit{ed\_coloured}(c,u,v) &\rightarrow \mathit{new\_color(c,u,v)}\} \\ \mathit{ed\_coloured}(c,u,v) &\rightarrow \mathit{pick\_color}(c, \mathit{union}(\mathit{incident}(u), \mathit{incident}(v))) \end{aligned}$$

Usando *ed_coloured*, implementamos "*colourable_with*", que diz quantas cores são usadas numa coloração qualquer do grafo:

$$\forall G(V,E), \{\forall u, v, < u, v > \in E, ed\_coloured(c,u,v) \supset colourable\_with(G) \rightarrow lenght(c)\}$$

Já temos o suficiente para usar indução. Note que as definições anteriores, se seguidas, determinarão uma certa ordem de coloração de arestas. Isto não fará diferença na prova do teorema a seguirm pois no passo de indução tratamos do "número máximo de novas coras usadas a cada nova aresta do grafo".

Exemplo 5.3 Provamos um teorema que diz que as arestas de um grafo podem ser coloridas com $|E|$ cores:

$$\forall G(V,E), colourable\_with(G) \leq lenght(E)$$

A prova é simples: a base se dá sem arestas (e o número de cores é 0); o passo onde se adicionam vértices não altera c; o passo que acrescenta uma aresta pode, no máximo, usar uma nova cor. ◇

Apesar de muito simples, esta prova por indução nos abre caminho para provas mais interessantes relacionadas a coloração de arestas.

O teorema de Vizing ainda não foi provado, mas acreditamos não oferecer grandes dificuldades.

Capítulo 6

Conclusões

> *"Não estou certo de nada. Não tenho mesmo, para dizer a verdade, nenhuma convicção definitiva - a respeito do que quer que seja. Sei apenas que nasci e que existo; experimento o sentimento de ser levado pelas coisas. Existo à base de algo que não conheço. Apesar de toda a incerteza, sinto a solidez do que existe e a continuidade do meu ser, tal como sou."*
> Carl Jung (1875-1961)

Apresentamos um estudo do sistema *Oyster-Clam* e o paradigma de planejamento de provas. O estudo deste sistema foi difícil devido a problemas diversos: no sistema - alguns predicados não se comportavam como descritos; na documentação - o manual do *Oyster* não é mais que o código fonte comentado e formatado em *LaTeX*, e o manual do *Clam* deixa muito a desejar, não trazendo muito sobre a implementação. A implementação de novos esquemas de indução mostrou-se mais difícil que descrito no manual.

Apesar disso, esta tarefa pode tornar-se mais fácil com a implementação de utilitários que ajudem na tarefa. Um utilitário que facilite a geração de táticas para novas estruturas facilitaria muito o processo, e não parece ser muito complexo, já que as provas indutivas costumam seguir um certo padrão.

As iniciativas na direção de uma formalização da teoria dos grafos em provadores de teoremas ainda são tímidas: até onde sabemos, há uma formalização feita por Yamamoto [YNHT95], uma de grafos não dirigidos por Chou [Cho94], e uma feita especificamente para modelar sinalização em malhas ferroviárias [Won92]. Todas estas formalizações feitas em ambientes interativos de prova de teoremas, mas nenhuma delas foi desenvolvida ou usada com um provador automático como o *Oyster-Clam*.

Há uma biblioteca de teoremas para teste de sistemas automáticos de prova, desenvolvida por Sutcliffe e Suttner [SS97]; a biblioteca é dividida em classes de problemas. Há apenas um problema em teoria de grafos. Acreditamos que isto é devido à característica intuitiva das técnicas usadas em provas em grafos, e à existência de inúmeros esquemas possíveis - e razoavelmente diferentes entre si - para indução. Um estudo na direção do raciocínio sobre estes esquemas e sua generalização parece promissor. A formalização parcial da teoria de grafos que apresentamos é bastante simples, mas constitui um passo na direção de algo maior.

A implementação da diagonalização no sistema *Oyster-Clam* foi feita por sugestão de Alan Bundy em comunicação pessoal, que classificou o problema como "moderadamente difícil". Esta parte de nosso trabalho foi aceita para publicação no 8º Congresso Português de Inteligência Artificial [PW97].

Apêndice A

Diagonalização

A.1 O Método

Este é o método desenvolvido para diagonalizar.

```
% metodo: diag/2
method(diag(Alpha,Gamma),                                % Name
       AnnH==>AnnG,                                      % Input
       [
        % Once we have chosen to diagonalize, these annotations will
        % not be useful.
        strip_meta_annotations(AnnG,G),
        unannotated_hyps(AnnH,H),

        matrix(U_M,E,M,G),
        append(H,U_M,U),
       (
         (
           unify(M,void),
             (
               thereis{Hyp\H}:
                 (
                   unify(Hyp,P:Prop),
                   matrix(Un,Ex,F of A=FmEx in (Alpha=>Gamma),Prop)
                 )
             )
              ,
```

```
            (
              unify(Hyp,P:Prop),
              matrix(Un,Ex,F of A=FmEx in (Alpha=>Gamma),Prop)
            )
          )
           v
          (
            unify(M,F of A=FmEx in TT=>void),
            (
              thereis{Exist\H}:
                unify(Exist,Delta:(Alpha=>Gamma))
            ),
            unify(Exist,Delta:(Alpha=>Gamma))
          )
         ),
         unify(Var,_:Alpha=>Alpha=>Gamma)          %
        ],                                         % Preconditions
        [
         hfree([X],H),
         delete(H,Hyp,NewH)                        %
        ],                                         % Postconditions
        [
         % If, for all x in Gamma, diff(x) != x, then
         % the proof will be possible.
         NewH ==>
          X:Gamma=>                                % Output
          diff(X,Gamma)=X in Gamma=>void
        ],
        diag(Alpha,Gamma)                          % Tactic
).
```

A.2 Regras de Reescrita

As regras de reescrita a seguir são usadas pelo método diagonalizante.

A.2.1 Diff

```
x:int  =>diff(x,int)                    = x+1 in int
```

```
x:pnat =>diff(x,pnat)                    = s(x)in pnat
diff(atom(a),atom)                       = atom(b)in atom
x:atom => diff(x,atom)                   = atom(a)in atom
t:u(1) => x:t list => diff(x,t list)     = element(t)::x in t list
t:u(1) => t2:u(1) => e:t => e2:t2
       => diff(e&e2,t#t2)                = diff(e,t)&diff(e2,t2)in(t#t2)
t:u(1) => t2:u(1) => e:t
       => diff(inl(e),t\t2)              = inr(element(t2))in(t\t2)
t:u(1) => t2:u(1) => e:t
       => diff(inr(e),t\t2)              = inl(element(t))in(t\t2)
```

A.2.2 Element

```
element(int)                             = 0 in int
element(pnat)                            = 0 in pnat
element(atom)                            = atom(a)in atom
x:u(1)=>element(x list)                  = nil in x list
element(unary)                           = unit in unary
t:u(1) => t2:u(1) => element(t#t2)       = (element(t)#element(t2))in(t#t2)
t:u(1) => t2:u(1) => element(t=>t2)      = lambda(~,element(t2))in(t=>t2)
t:u(1) => t2:u(1) => element(t\t2)       = inr(element(t2))in(t\t2)
```

A.3 Indução em Tipos

As mudanças realizadas no *Oyster* a fim de implementar indução em tipos são listadas a seguir:

A.3.1 Regras

Transitivity of membership: every object inhabits some universe. The universes are not "universes of types" anymore; they are "universes of types and objects"...

```
%
% Atom
%
rule(intro(atom(A)),
```

```
     _==>u(1) ext atom(A),
     []
    ):- atom(A).
% Atoms are realizations for u(1).

rule(intro,
     _==>atom(A) in u(1),
     []
    ):- atom(A).
% Atoms are members of u(1).


%
% Int
%
rule(intro(X),
     _==>u(1) ext X,
     []
    ):- var(X),X='<integer>';integer(X).
% Atoms are realizations for u(1).

rule(intro,
     _==>I in u(1),
     []
    ):- integer(I).
% Integers are members of u(1).

rule(intro(- ~),
     _==>u(1) ext -E,
     [==>int ext E]
    ).
% -X are members of u(1), only if X is integer.

rule(intro,
     _==> -I in u(1),
     [==> I in int]
    ).
% -X is member of u(1) if X is in int.

rule(intro(- ~),
     _==> T=TT in u(1),
```

```
     [==> -T = -TT in int]
    ).
% Equality rule.

rule(intro(~ + ~),
     _==>u(1) ext A + B,
     [
      ==>int ext A,
      ==>int ext B
     ]
    ).

rule(intro(~ - ~),
     _==>u(1) ext A - B,
     [
      ==>int ext A,
      ==>int ext B
     ]
    ).

rule(intro(~ * ~),
     _==>u(1) ext A * B,
     [
      ==>int ext A,
      ==>int ext B
     ]
    ).

rule(intro(~ / ~),
     _==>u(1) ext A / B,
     [
      ==>int ext A,
      ==>int ext B
     ]
    ).

rule(intro(~ mod ~),
     _==>u(1) ext A mod B,
     [
      ==>int ext A,
      ==>int ext B
```

```
    ]
   ).
% Refinement rules...

rule(intro,
     _==>A + B in u(1),
     [
      ==>A in int,
      ==>B in int
     ]
    ).

rule(intro,
     _==>A - B in u(1),
     [
      ==>A in int,
      ==>B in int
     ]
    ).

rule(intro,
     _==>A / B in u(1),
     [
      ==>A in int,
      ==>B in int
     ]
    ).

rule(intro,
     _==>A * B in u(1),
     [
      ==>A in int,
      ==>B in int
     ]
    ).

rule(intro,
     _==>A mod B in u(1),
     [
      ==>A in int,
      ==>B in int
```

```
     ]
    ).


%
% Pnat
%

rule(intro(pnat(0)),
     _==>u(1) ext 0,
     []
    ).
% 0 is a realization for u(1).

rule(intro,
     _==>0 in u(1),
     []
    ).
% 0 is member of u(1).

rule(intro(s),
     _==>u(1) ext s(X),
     [
      ==>pnat ext X
     ]
    ).
% s(x) is a realization for u(1), provided that x is in pnat.

rule(intro,
     _==> s(X) in u(1),
     [
      ==>X in pnat
     ]
    ).
% s(x) is member of u(1), provided that x is in pnat.


%
% Unary
%
```

```
rule(intro(unit),
     _==>u(1) ext unit,
     []
    ).
% unit is a realization for u(1).

rule(intro,
     _==>unit in u(1),
     []
    ).
% atoms are members of u(1).
```

A.3.2 Eval

```
eval(type_eq(A,B,S,T),R):-
   eval(A,X),
   eval(B,Y),
   (X=Y,eval(S,R);
        eval(T,R)),
   !.

eval(it_ind(V,Tvoid,Tunary,Tint,Tpnat,Tatom,
     [
      Type1,Type2,
      List,
      Tlist,

      Disj1,
      Disj2,
      Tdisj,

      Fun1,
      Fun2,
      Tfun,

      Prod1,
      Prod2,
      Tprod]),C):-
```

```
    eval(V,VV),
   (
     VV=void,eval(Tvoid,C);
     VV=unary,eval(Tunary,C);
     VV=int,eval(Tint,C);
     VV=pnat,eval(Tpnat,C);
     VV=atom,eval(Tatom,C);
     VV=T1 list,
     eval(Tlist,[T1],[Type1],C);

     VV=(T1 \ T2),write(Type1),nl,write(Tdisj),nl,
     eval(Tdisj,[T1,T2],[Type1,Type2],C);

     VV=(T1 => T2),
     eval(Tfun,[T1,T2],[Type1,Type2],C);

     VV=(T1 # T2),
     eval(Tprod,[T1,T2],[Type1,Type2],C)
   ),!.
```

A.3.3 Regras para eliminar v:u(1)

Now, these are the necessary rules to eliminate v:u(1)

```
% Here we have constructors for the types in u(1).
% Decision operator for types in u(1), and related rules.

rule(decide(T1=T2 in u(1),new[U]),
     H==>T ext type_eq(T1,T2,su(E1,[axiom],[U]),su(E2,[lambda(~,axiom)],[U])),
     [
      [U:T1=T2 in u(1)]==>T ext E1,
      [U:T1=T2 in u(1)=>void]==>T ext E2,
      ==>T1 in u(1),
      ==>T2 in u(1)
     ]
    ):-
    \+ var(T1), \+ var(T2), syntax(H,T1), syntax(H,T2), free([U],H).
% This rule generates the decision operator for two elements in u(1).
```

```
rule(intro(new[Z]),
     H==>type_eq(T1,T2,X,Y) in T,
     [
      ==>T1 in u(1),
      ==>T2 in u(1),
      [Z:T1=T2 in u(1)]==>X in T,
      [Z:T1=T2 in u(1)==>void]==>Y in T
     ]
    ):-free([Z],H).
% The decision operator is in T if X and Y are both in T.

rule(reduce(true),
     _==>type_eq(T1,T2,X,Y)=X in T,
     [
      ==> X in T,
      ==> Y in T,
      ==> T1=T2 in u(1)
     ]
    ).
% Allows the reduction of the current goal.

rule(reduce(false),
     _==>type_eq(T1,T2,X,Y)=Y in T,
     [
      ==> X in T,
      ==> Y in T,
      ==> T1=T2 in u(1)=>void
     ]
    ).
% Allows the reduction of the current goal.

rule(elim(V,new([V0,V1,Type1,Type2,List,Disj1,Disj2,Fun1,Fun2,Prod1,Prod2])),
     H==>G ext it_ind(V,Tvoid,Tunary,Tint,Tpnat,Tatom,
     [
      Type1,Type2,
      List,
      Tlist,

      Disj1,
      Disj2,
      Tdisj,
```

```
 Fun1,
 Fun2,
 Tfun,

 Prod1,
 Prod2,
 Tprod]),
[
  [
   V0:V=void in u(1)
  ]
  ==>G ext Tvoid,
  [
   V0:V=unary in u(1)
  ]
  ==>G ext Tunary,
  [
   V0:V=int in u(1)
  ]
  ==>G ext Tint,
  [
   V0:V=pnat in u(1)
  ]
  ==>G ext Tpnat,
  [
   V0:V=atom in u(1)
  ]
  ==>G ext Tatom,
  [
   Type1:u(1),
   List:G,
   V1:V=Type1 list in u(1)
  ]
  ==>G ext Tlist,
  [
   Type1:u(1),
   Type2:u(1),
   Disj1:G,
   Disj2:G,
   V1:V=(Type1 \ Type2) in u(1)
```

```
        ]
        ==>G ext Tdisj,
        [
         Type1:u(1),
         Type2:u(1),
         Fun1:G,
         Fun2:G,
         V1:V=(Type1 => Type2) in u(1)
        ]
        ==>G ext Tfun,
        [
         Type1:u(1),
         Type2:u(1),
         Prod1:Type1,
         Prod2:Type2,
         V1:V=(Type1 # Type2) in u(1)
        ]
        ==>G ext Tprod
    ]) :-
     decl(V:u(1),H),
     free([V0,V1,Type1,Type2,List,Disj1,Disj2,Fun1,Fun2,Prod1,Prod2],H).
% Eliminating the universe u(i) will generate a type induction term

eval(type_eq(A,B,S,T),R):-
   eval(A,X),
   eval(B,Y),
   (X=Y,eval(S,R);
        eval(T,R)),
   !.
```

Apêndice B

Esquemas de Indução

B.1 Os esquemas

```
scheme([_::_,_::_],indgraf,
[
 [U:int]                              ==>phi(U::nil,nil),     % Base: grafo trivial
 [U:int,V:int list,phi(V,nil)]        ==>phi(U::V,nil),       % Passo: vertice
 [V:int list,E:(int#int list),
  phi(V,E),Ed:int#int,member(Ed,V)] ==>phi(V,Ed::E)          % Passo: aresta
]
==>phi( Vertices:(int list) , Edges:(int#int) list )).

scheme([_::_,_::_],indgraf_2,
[
 [U:int]                              ==>phi(U::nil,nil),       % Base: grafo trivial
 [Va:int list,Vb:int list,
  phi(V,nil)]                         ==>phi(app(Va,Vb),nil),   % Passo: uniao
 [V:int list,E:(int#int list),
  phi(V,E),Ed:int#int,member(Ed,V)] ==>phi(V,ED::E)           % Passo: aresta
]
==>phi( Vertices:(int list) , Edges:(int#int) list )).
```

B.2 Regras de reescrita usadas

```
graph  =  int list#(int#int)list
```

```
/* Pertinencia para listas de inteiros - member(el,l) verifica se
   el e membro da lista l. */

member(B,nil)                              = void
B=C in int    => member(B,C::D)            = {true}
(B=C in int) => void => member(B,C::D)     = member(B,D)


/* Concatenacao de duas listas de inteiros. */

app(nil,B)  = B
app(B::C,D) = B::app(C,D)


/* Insere um vertice em uma lista, sem repetir vertices. */

member(B,C)            => ins_vtx(B,C) = C
(member(B,C) => void) => ins_vtx(B,C) = app(B::nil,C)


/* new_v(v,G) insere um novo vertice v em um grafo G */

new_v(B,C) = ins_vtx(C,fst(B))&snd(B)


/* Uniao de dois conjuntos (implementados como listas). */

union(nil,B)                               = B
union(B,nil)                               = B
member(B,C)            => union(B::D,C) = union(D,C)
(member(B,C) => void) => union(B::D,C) = B::union(D,C)


/* ed_member(E,u,v) verifica se <u,v> pertence a E. */

ed_member(nil,B,C)                                      = void
B=C&D in (int#int)            => ed_member(B::E,C,D) = {true}
(B=C&D in (int#int)) => void => ed_member(B::E,C,D) = ed_member(E,C,D)
```

```
/* ins_ed(E,u,v) insere uma aresta <u,v> na lista E, sem repetir
   arestas. */

ed_member(B,C,D)          => ins_ed(B,C,D) = B
ed_member(B,C,D) => void => ins_ed(B,C,D) = (C&D)::B


/* new_e(G,u,v) insere uma aresta <u,v> em um grafo G. */

new_e(B,C,D) = ins_vtx(C,ins_vtx(D,fst(B)))&ins_ed(snd(B),C,D)


/* Remove uma aresta de uma lista E. */

rm_edge(B,C,nil)                                    = nil
B=C in int # D=E in int => rm_edge(B,D,(C&E)::F) = rm_edge(B,D,F)
(B=C in int) => void    => rm_edge(B,D,(C&E)::F) = (C&E)::rm_edge(B,D,F)
(B=C in int) => void    => rm_edge(D,B,(E&C)::F) = (E&C)::rm_edge(D,B,F)


/* Dadas a lista de arestas E e um vertice  v, in_degree(E,v)
   calcula o in-degree de v. */

in_degree(nil,B)                                = 0
B=C in int           => in_degree((D&B)::E,C) = 1+in_degree(E,C)
(B=C in int) => void => in_degree((D&B)::E,C) = in_degree(E,C)


/* Dadas a lista de arestas E e um vertice v, out_degree(E,v)
   calcula o out-degree de v. */

out_degree(nil,B)                                = 0
D=C in int           => out_degree((D&B)::E,C) = 1+out_degree(E,C)
(D=C in int) => void => out_degree((D&B)::E,C) = out_degree(E,C)


/* Dadas a lista de arestas E e um vertice v, degree(E,v)
   calcula o grau de v. */
```

```
degree(nil,B)                                  = 0
B=C in int            => degree((D&B)::E,C) = 1+degree(E,C)
D=C in int            => degree((D&B)::E,C) = 1+degree(E,C)
(D=C in int) => void => degree((D&B)::E,C) = degree(E,C)


/* Dadas uma lista de cores C e uma lista de cores ja usadas U,
   pick_color(C,U) escolhe uma cor ainda nao usada. */

pick_color(nil,lu)                    = 0
member(e,lu) => pick_color(e::l,lu) = pick_color(l,lu)
pick_color(e::l,lu)                   = e


pick_color(c,union(incident(u),incident(v)))=0
       => ed_coloured}(c,u,v)  = new_color(c,u,v)
ed_coloured(c,u,v)             = pick_color(c,union(incident(u),incident(v)))


ed_coloured}(c,u,v) => colourable_with(g) = lenght(c)


/* path(u,v,V,E) acha um caminho de u a v, se existir algum. */

path(B,B,C,D)                                        = nil
out_degree(B,C)=0 in int => path(C,D,E,B)        = nil
ed_member(B,C,D)         => path(C,D,E,B)        = (C&D)::nil
path(B,C,D,nil)                                      = nil
(path(B,C,D,E)=nil in (int#int)list) => void
                         => path(F,C,D,(F&B)::E) = (F&B)::path(B,C,D,E)
```

Bibliografia

[ABI+94] Peter Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. Tps: An interactive and automatic tool for proving theorems of type theory. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications, 6th International Workshop, HUG '93*, number 780 in Lecture Notes in Computer Science, pages 366–370, Vancouver, B.C., Canada, 1993, August 1994. Springer-Verlag.

[BM79a] J. A. Bondy and U. S. R. Murty. *Graph theory with applications.* North Holland, 1979.

[BM79b] Robert Boyer and J. S. Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, 1979.

[Bun88] Alan Bundy. Explicit plans to guide inductive proofs. Technical Report 349, Dept. of Artificial Intelligence, University of Edinburgh, 1988.

[Bun91] Alan Bundy. A science of reasoning. In J-L Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research paper 445.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *Proceedings of the 10th CADE*, number 449 in LNAI, pages 647–648. Springer-Verlag, 1990.

[BvHIS93] Alan Bundy, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: a heuristic to guide inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[Cho94] Ching–Tsun Chou. A fomal theory of undirected graphs in higher-order logic. In *7th International Workshop on Higher-Order Logic Theorem Proving System and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 144–157. Springer-Verlag, 1994.

[Con86] R. L. Constable. *Implementing Mathematics with the Nuprl Proof Development system.* Prentice Hall, Englewood Cliffs, 1986.

[Cut92] Nigel Cutland. *Computability: an introduction to recursive function theory.* Cambrifge University Press, 1992.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannod. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. North-Holland, 1990.

[DP90] Nachum Dershowitz and E. Pinchover. Inductive systems of equational programs. In *Proceedings of the 8th National Conference on AI*, pages 234–239. American Association for Artificial Intelligence, 1990.

[HK95] Xiaorong Huang and Manfred Kerber. Reuse of proofs by meta-methods. In Jana Köhler, editor, *Proceedings of the IJCAI-Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, Montreal, Canada, 1995.

[HKC95] Xiaorong Huang, Manfred Kerber, and Lassaad Cheikhrouhou. Adapting the diagonalization method by reformulations. In Alon Levy and Pandu Nayak, editors, *Proceedings of the Symposium on Abstraction, Reformulation, and Approximation, SARA-95*, Ville d'Esterel, Canada, 1995.

[HKK+94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. Ω-MKRP: A proof development environment. In Alan Bundy, editor, *Automated Deduction — CADE-12*, Proceedings of the 12th International Conference on Automated Deduction, pages 788–792, Nancy, France, 1994. Springer-Verlag, Berlin, Germany. LNAI 814.

[HKPM97] Gerard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant : A tutorial : Version 6.1. Technical Report 0204, INRIA, 1997.

[Hor88] Christian Horn. *The Oyster Proof Development System.* Dept. of Artificial Intelligence, Univ. of Edinburgh, 1988.

[HS94] Christian Horn and Alan Smaill. *Artificial Intelligence in Mathematics*, chapter From Meta-level Tactics to Object-level Programs, pages 135–146. Institute of Mathematics and its Applications, Claredon Press, 1994.

[HtDg97] Frank Van Harmelen and the Dream group. *The Clam Proof Planner.* Dept. of Artificial Intelligence, Univ. of Edinburgh, 1997. Clam is available with its manual at ftp://dream.dai.ed.ac.uk.

[KB70] Donald Knuth and P. B. Bendix. *Computational Problems in Abstract Algebra*, chapter Simple Words Problems in Universal Algebras, pages 263–297. Pergamon Press, Oxford, U.K., 1970.

[Mad91] Peter Madden. *Automated Program Transformation Through Proof Transformation.* PhD thesis, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1991.

[Mel95] Erica Melis. Analogy in Clam. Technical Report 766, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1995.

[ML82] Per Martin-Löf. Constructive mathematics and computer programing. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover, August 1982. Published by North Holland, Amsterdam.

[Pau93] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[PN90] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, Computer Laboratory, January 1990.

[Pri96] Carlos Augusto Di Prisco. Una introduccion a la Teoria de Conjuntos y los fundamentos de las matematicas. 1996.

[PW97] Jerônimo Pellegrini and Jacques Wainer. Diagonalization and type rewriting in Clam. Aceito para publicação nos anais da 8.a Conferência Portuguesa em Inteligência Artificial, 1997.

[Rei96] Gordon Reid. The representation of the rippling heuristic in proof planning using coloured annotations. Available from DAI, Edinburgh, 1996.

[Rin96] Maurício Ayala Rincón. Sistemas de reescrita de termos - o modelo de computação funcional e suas aplicações. XV Jornada de Atualização em Informática - XVI Congresso da SBC, Recife, Agosto de 1996.

[SS97] Geoff Sutcliffe and Christian Suttner. Thousands of problems for theorem provers. http://wwwjessen.informatik.tu-muenchen.de:80/~tptp/, 1997.

[Wal96] Toby Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[Won92] Wai Wong. A simple graph theory and its application in railway signaling. In M. Archer et al, editor, *Proc. of 1991 Workshop on the HOL Theorem Proving System and Its Applications*, pages 395–409. IEEE Computer Society Press, 1992.

[YNHT95] Mitsuharu Yamamoto, Shin–ya Nishizaki, Masami Hagiya, and Yozo Toda. Formalization of planar graphs. In *Higher-Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 369–384. Springer-Verlag, 1995.