Alípio Mário Guedes Jorge

Indução Iterativa de Programas Lógicos

Uma abordagem à síntese de programas lógicos a partir de especificações incompletas

Tese submetida para obtenção
do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Janeiro 1998

Aos meus Pais.

À minha Mulher.

Agradecimentos

O meu orientador, Pavel Brazdil, tem sido uma fonte ilimitada de entusiasmo, encorajamento e paciência. Obrigado pelos seus valiosos comentários, sugestões e indicações.

Os meus colegas do LIACC proporcionaram um excelente ambiente de trabalho, com relaxantes pausas sempre que necessário. Agradeço em especial aos colegas do NIAAD (o grupo de aprendizagem automática), em particular aos que comigo partilham o gabinete, o João Gama e o Luís Torgo.

Devo agradecer à JNICT (Programa Ciência, bolsa BD/1327/91/IA e PRAXIS XXI, bolsa BD/3285/94) pelo apoio financeiro sem o qual este trabalho não teria sido possível. Agradeço também a MLNet, ILPNet, Faculdade de Economia da U. Porto, e ao instituto IISF do Japão, que possibilitaram a minha participação em várias reuniões científicas internacionais.

Obrigado a todos os meus amigos (tenho felizmente uma boa colecção deles) por todos os bons momentos, jantares, festas, fins-de-semana, etc. Agradeço especialmente aos que estiveram mais directamente envolvidos com o meu trabalho, o Mário Florido e o Paulo Azevedo.

Agradeço à minha família, pelo apoio e carinho (tenho também uma grande família).

Um obrigado muito especial para a minha mulher, Xinha, pelo seu amor e companhia, e à minha filha Carolina, por me dar noites sossegadas desde os seus 3 meses.

E finalmente, agradeço aos meus pais, pelo seu amor, apoio e encorajamento desde sempre. Bem hajam.

Índice

1. Introdução

Nesta dissertação descrevemos uma metodologia de construção automática de programas em Prolog a partir de diversos fragmentos de informação disponíveis, tais como exemplos positivos e exemplos negativos da relação definida pelo programa pretendido, restrições de integridade, esboços de algoritmo, definições de predicados auxiliares e informação sobre a estrutura das cláusulas a construir. Esta metodologia é aqui apresentada sob a forma de um sistema implementado também em linguagem Prolog a que chamamos SKIL (Sketch-based Inductive Learner,), e de uma sua extensão iterativa denominada SKILit (SKIL, iterative version).

A informação dada ao sistema descreve a forma como o programa pretendido se deve comportar e pode ser vista como uma especificação desse mesmo programa. Uma vez que se tratam apenas de fragmentos de informação estamos perante uma *especificação incompleta* que não descreve totalmente o comportamento do programa. O comportamento não especificado é hipotetizado pela nossa metodologia através de técnicas de inferência indutiva. Por esse motivo, podemos ver o presente trabalho como uma abordagem à *síntese indutiva de programas lógicos a partir de especificações incompletas*. Neste caso estamos na área de *Programação Automática* ou de *Síntese (Automática) de Programas*.

Por outro lado a síntese indutiva de programas lógicos inclui-se naturalmente na área de *Programação Lógica por Indução* (*Inductive Logic Programming*[1], PLI). Por sua vez, a Programação Lógica por Indução, é simultaneamente uma sub-área da *Aprendizagem Automática* (*Machine Learning*), pois procura induzir teorias a partir de observações, e da *Programação Lógica* (*Logic Programming*), porque utiliza linguagens da programação lógica para descrever observações e teorias (Figura 1.1).

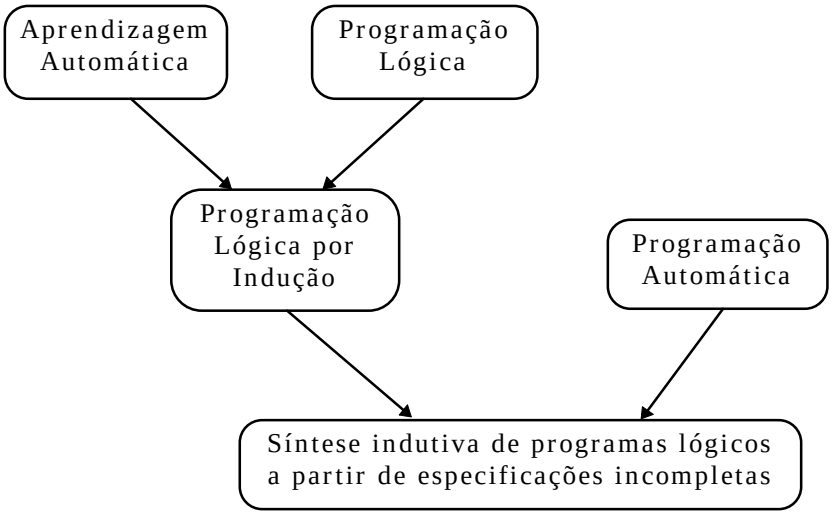

Figura 1.1: Localização do presente trabalho.

Vejamos um exemplo simples do que pode ser a síntese indutiva de programas lógicos a partir de especificações incompletas. Dado o conjunto E^+ de exemplos positivos da relação *descende/2*

descende(alipio,antonio).
descende(alipio,adriana).

Dado também o conjunto de exemplos negativos E^- da mesma relação

descende(antonio,alipio).

[1] Em Anexos incluimos uma lista de traduções de Português para Inglês dos termos técnicos mais específicos utilizados nesta tese.

descende(adriana,antonio).

e um programa auxiliar *B* (a que mais tarde chamaremos conhecimento de fundo)

filho(antonio,adriana).
filho(alipio,antonio).

é construído o programa lógico *P* que define o predicado *descende/2*.

descende(A,B)←filho(A,B).
descende(A,B)←filho(A,C),descende(C,B).

Em termos de síntese de programas, a especificação é constituída pelos conjuntos de exemplos E^+ e E^-. O programa *P* é um programa sintetizado que, juntamente com o programa auxiliar *B*, verifica essa especificação. Em termos de PLI, os exemplos E^+ e E^- constituem observações. Estas são explicadas pela teoria induzida *P* em conjunto com o conhecimento de fundo *B*, uma vez que E^+ são suas consequências lógicas, ao contrário de E^-.

As condições de que *P* verifica a especificação incompleta $\{E^+,E^-\}$, ou de que *P* explica as observações $\{E^+,E^-\}$ relativamente a *B* podem ser enunciadas assim:

$$P \cup B \models E^+ \qquad \text{e} \qquad P \cup B \not\models e^- \text{ para todo } e^- \in E^-$$

O objectivo, grosso modo, de um sistema de PLI (seja ou não encarado como um sistema de síntese de programas) é encontrar um programa *P* que satisfaça as condições acima enunciadas. A presente dissertação descreve a metodologia seguida por um desses sistemas: o SKILit.

1.1 Motivação

Uma das principais motivações do presente trabalho foi a fraca adequação de muitas técnicas e algoritmos actuais de PLI ao problema da síntese indutiva de programas, e

especificamente à síntese de programas recursivos. Os sistemas de PLI que representavam o estado da arte à data do início do desenvolvimento do presente trabalho, tais como FOIL e GOLEM, mostravam-se praticamente incapazes de lidar com especificações incompletas, exigindo quase uma definição extensiva dos predicados a induzir. O sistema SKILit que nós propomos consegue induzir definições recursivas a partir de conjuntos esparsos de exemplos positivos, mostrando grande robustez nos resultados alcançados em testes onde os exemplos positivos disponíveis são muito poucos e mesmo quando estes são gerados aleatoriamente. Para isso, o SKILit utiliza uma técnica de indução iterativa que constitui uma das principais contribuições deste trabalho.

Outros sistemas que surgiram entretanto exibem também esta capacidade de induzirem cláusulas recursivas em presença de conjuntos de exemplos positivos esparsos. Todavia esses sistemas são quase sempre bastante limitados em termos de circunscrição de linguagem (*language bias*) pois apenas conseguem induzir programas lógicos dentro de uma família relativamente restrita de programas. Seguindo a metodologia descrita nesta dissertação, o sistema SKILit consegue sintetizar potencialmente qualquer programa em Prolog puro pois permite a declaração de conhecimento de programação através de gramáticas de estrutura clausal. Estas gramáticas são escritas usando a notação de gramáticas de cláusulas definidas (*definite clause grammars, DCG*). Note-se todavia que o SKILit trabalha normalmente quando nenhuma gramática é dada tendo o sistema apenas que desenvolver mais esforço de procura.

Outro problema que abordamos nesta tese prende-se com o elevado número de exemplos negativos que muitos sistemas requerem para evitar a indução de programas sobre-gerais. A nossa metodologia permite a utilização de restrições de integridade para descrever os limites da relação definida pelo predicado a induzir. A utilização de restrições de integridade na PLI não é uma novidade por nós apresentada. Todavia, o seu processamento envolve habitualmente pesados mecanismos de demonstração de teoremas (*theorem provers*). A abordagem aqui adoptada para a verificação de restrições

de integridade é muito eficiente. Baseia-se numa técnica de Monte Carlo que permite, dada uma restrição de integridade *I* e um programa *P* induzido, verificar com algum grau de incerteza a consistência entre *P* e *I*.

1.2 Contribuições principais

A metodologia apresentada nesta dissertação combina algumas técnicas novas com métodos existentes. As nossas principais contribuições são o motor indutivo do sistema SKIL, a indução iterativa e um eficiente método de Monte Carlo para lidar com restrições de integridade. O motor indutivo base da nossa metodologia é adequado à síntese de programas a partir de poucos exemplos. Tira também partido de declarações de tipo e de modo, assim como de conhecimento de programação representado sob a forma de gramáticas de estrutura clausal e de esboços de algoritmo. Os esboços de algoritmo permitem ao utilizador representar conhecimento de programação específico e transmitir esse conhecimento ao sistema. A indução iterativa permite mais flexibilidade na escolha dos exemplos positivos dados a um sistema que sintetiza programas lógicos recursivos. O verificador de integridade de Monte Carlo torna prático o uso de restrições de integridade na síntese indutiva de programas. Iremos agora descrever sucintamente cada um destes aspectos.

1.2.1 O motor indutivo

A partir de uma espcificação que inclui os exemplos positivos

member(2,[2]).
member(2,[1,2]).

O motor indutivo do SKIL sintetiza as cáusulas

member(A,[A|B]). (C1)
member(A,[B|C])←member(A,C). (C2)

Na nossa metodologia, cada cláusula é construída procurando uma ligação relacional entre os argumentos de entrada e os de saída de um exemplo positivo. A ligação é feita utilizando os predicados auxiliares definidos como conhecimento de fundo e tendo em conta os modos de entrada/saída (*input/output modes*) declarados para cada predicado. Por exemplo, assumindo que o segundo argumento é de entrada e o primeiro de saída, os argumentos de *member(2,[1,2])* podem ser relacionalmente ligados da seguinte forma. Partindo de *[1,2]* obtemos os termos *1* e *[2]* por decomposição da lista *[1,2]* e a partir de *[2]* obtemos o termo *2*, utilizando a cláusula (C1). Estão assim ligados os termos. Esta ligação corresponde à seguinte instância da cláusula (C2).

member(2,[1,2])←member(2,[2]).

A instância é transformada numa cláusula substituindo termos por variáveis.

A procura de uma ligação relacional é guiada por um exemplo (indução guiada pelos dados), o que tem a vantagem de reduzir o número de cláusulas candidatas a considerar. A estratégia para construir cada cláusula depende apenas de um exemplo positivo de cada vez. A razão para isso é que o nosso motor indutivo não emprega qualquer tipo de heurística baseada na cobertura dos exemplos, tal como acontece com FOIL [96] ou CHILLIN [125]. Estas heurísticas tendem a ser menos fiáveis quando há poucos exemplos disponíveis.

O nosso motor de indução também explora conhecimento de programação representado como gramáticas de estrutura clausal. Este é um formalismo muito simples e poderoso que pode ser visto como uma circunscrição declarativa (*declarative bias*).

O motor indutivo permite também a síntese a partir de esboços de algoritmo, os quais podem ser vistos como exemplos positivos parcialmente explicados que aceleram o processo de síntese.

Por exemplo, o exemplo positivo *member(6,[3,1,6,5])* pode ser parcialmente explicado dizendo ao sistema que a partir da lista *[3,1,6,5]* obtém-se *[1,6,5]* e a partir desta lista

obtém-se *6*, o resultado desejado. Esta informação pode ser representada por um esboço de algoritmo e ser dada ao sistema. O esboço é representado como uma cláusula fechada com alguns literais especiais.

member(6,[3,1,6,5])←$P1([3,1,6,5], [1,6,5]), $P2([1,6,5],6).

Os predicados *$P1* e *$P2* representam sequências desconhecidas de literais que envolvem predicados operacionais. A tarefa de síntese consiste sobretudo em construir essas sequências de literais. Qualquer exemplo positivo como *member(2,[1,2])* pode ser representado como um esboço como *member(2,[1,2])←$P3(2,[1,2])*.

O motor indutivo aqui apresentado trata exemplos positivos e esboços de algoritmo de uma maneira uniforme. Cada cláusula é obtida a partir de um exemplo ou esboço usando um único operador de refinamento de esboços. Mostra-se que este operador de refinamento de esboços é completo sob os pressupostos adequados.

1.2.2 Indução iterativa

Para induzir uma cláusula recursiva a partir do exemplo *member(2,[1,2])*, o motor indutivo do SKIL precisa que o exemplo *member(2,[2])* lhe seja dado. Este facto dificulta a indução de programas recursivos pelo SKIL quando os exemplos positivos não são cuidadosamente escolhidos. A indução iterativa serve para facilitar a síntese de programas recursivos. O sistema SKILit utiliza a indução iterativa.

Suponhamos que a especificação contém agora os exemplos positivos

member(7,[7,9]).
member(2,[1,2]).

A partir desta especificação, o SKILit consegue sintetizar a mesma definição recursiva que vimos na Scção anterior.

member(A,[A|B]). (C1)

member(A,[B|C])←member(A,C). (C2)

Vejamos, em linhas gerais como. Numa primeira iteração são construídas duas cláusulas, uma para cada exemplo positivo.

member(A,[A|B]).
member(A,[B,A|C]). (C3)

Numa segunda iteração são de novo percorridos os exemplos positivos. A cláusula recursiva C2 é construída partindo do exemplo *member(2,[1,2])* e passando pelo facto *member(2,[2])*. Este facto, contudo, não se encontra na especificação. Em vez disso é coberto pela cláusula C1. Esta cláusula tem um papel muito importante no processo.

As cláusulas que surgem nas primeiras iterações são utilizadas pelo sistema para auxiliar a introdução de cláusulas recursivas, e correspondem a propriedades da relação a sintetizar. Estas propriedades podem fazer ou não parte do programa final. As propriedades tornadas redundantes por outras cláusulas são eliminadas pelo módulo do SKILit que faz compressão de programas, o CT.

1.2.3 Restrições de integridade e o método de Monte Carlo

O módulo MONIC do SKILit processa as restrições de integridade utilizando uma estratégia de Monte Carlo bastante eficiente, embora incompleta. Todo o programa *P* sintetizado pelo SKILit deve satisfazer as restrições de integridade. Essa verificação é feita gerando aleatoriamente *n* factos que são consequências lógicas do programa. Cada um desses factos serve para procurar uma instância violante (*violating instance*) de uma das restrições de integridade.

Por exemplo, a restrição de integridade para o predicado *union/3*

union(A,B,C),member(X,C)→member(X,A),member(X,B)

lê-se: "se *X* pertence à lista *C*, então ou pertence a *A* ou a *B*", e serve para descrever uma condição que se pretende impôr ao programa que defina o predicado *union/3*. Dado um

programa candidato *P* com *union([2],[],[3])* como consequência lógica, uma instância violante da restrição é

union([2],[],[3]),member(3,[3])→member(3,[]),member(3,[2])

pois o antecedente é verdadeiro e o consequente falso.

O nosso verificador MONIC não encontra necessariamente uma instância violante da restrição de integridade. Isso só acontece se uma das *n* consequências lógicas de *P* retiradas aleatoriamente resultar numa instância violante como exemplificado acima. A probabilidade de isso acontecer cresce com o *n*, o que pode ser controlado pelo utilizador.

1.3 Resumo dos capítulos

No Capítulo 2 situamos o presente trabalho num contexto mais geral de desenvolvimento de programas. Referimos as ferramentas CASE, os métodos formais, a síntese dedutiva e a síntese indutiva. No Capítulo 3 falámos de Programação Lógica por Indução (PLI). Começamos por uma introdução à Programação Lógica, e apresentamos os conceitos e técnicas da PLI que nos interessam para esta tese.

No Capítulo 4 apresentamos o motor indutivo no cerne da nossa metodologia. É apresentado como o sistema SKIL, que sintetiza programas lógicos a partir de exemplos e de esboços. Damos um operador de refinamento de esboços e mostramos um resultado de completude desse operador. No Capítulo 5 introduzimos a técnica da indução iterativa que resolve a principal limitação do sistema SKIL: a dificuldade de induzir definições recursivas a partir de conjuntos esparsos de exemplos. O sistema SKILit invoca iterativamente o (sub-)sistema SKIL. No Capítulo 6 fazemos uma avaliação empírica do método e do sistema SKILit. No Capítulo 7 descrevemos outro módulo do sistema, o verificador de integridade MONIC, que usa uma estratégia de Monte Carlo, e

que permite a inclusão de restrições de integridade nas especificações dadas ao sistema SKILit. No Capítulo 8 apresentamos conclusões, limitações e trabalho para o futuro.

2. Desenvolvimento de Programas

Neste capítulo fazemos uma breve descrição de várias metodologias do desenvolvimento de programas desde a engenharia de software à programação automática, passando pelas ferramentas CASE, e pelo desenvolvimento formal de programas. É dada maior atenção à síntese de programas a partir de especificações incompletas, em particular à síntese de programas lógicos a partir de exemplos.

2.1 Introdução

A engenharia de 'software' divide tradicionalmente o desenvolvimento dum programa em quatro fases distintas [113].

- *Elaboração da especificação*, onde as pretensões do utilizador final quanto ao programa a construir são recolhidas e registadas em linguagem corrente. A *especificação* deve conter informação sobre o que o programa deve fazer e não sobre como o deve fazer.

- *Análise e desenho*, onde primeiro se atacam os problemas postos pela especificação. Aqui se faz um delineamento alto-nível dos algoritmos envolvidos, das estruturas de dados escolhidas e do fluxo de dados.

- *Implementação*, onde os algoritmos alto-nível desenhados na fase anterior são passados a código executável[2].

- *Verificação*, onde o programa implementado é confrontado com a especificação. Caso se encontrem falhas no programa (i.e., o programa está incorrecto) refazem-se uma ou mais das fases anteriores.

O nosso trabalho pretende contribuir para a automatização da geração de código no âmbito da programação não extensiva[3], ao mesmo tempo que permite uma especificação incompleta, através de exemplos e não só. Por um lado pretende aliviar o trabalho de elaboração de uma especificação, por outro pretende-se automatizar total ou parcialmente a geração de código a partir de especificações incompletas.

2.2 Programação automática

E se o computador pudesse levar a cabo a difícil tarefa de programar? Este é um sonho tão velho quanto a própria programação. A busca da programação automática é movida por duas motivações principais:

---

[2] A expressão "código executável" é utilizada para nos referimos a linguagens para as quais exista um interpretador/compilador.

[3] Faz-se também a distinção entre a *programação extensiva* (programming in the large) e a *programação não extensiva* (programming in the small). A programação extensiva envolve uma equipa de analistas e programadores mais ou menos numerosa e constitui uma tarefa que leva largos meses ou mesmo anos a concretizar. A programação não extensiva diz respeito a sistemas que levam não mais do que poucos meses a ser desenvolvidos por pouco mais do que uma ou duas pessoas. A engenharia de 'software' dedica-se especialmente à programação extensiva.

- acelerar o processo de desenvolvimento de programas, principalmente a fase de implementação atrás referida, libertando tanto quanto possível o analista/programador de tarefas rotineiras;

- aumentar a fiabilidade dos programas, minimizando a intervenção humana, a qual está frequentemente na origem de erros.

As ferramentas informáticas de apoio ao desenvolvimento de 'software', as metodologias formais de desenvolvimento, e a síntese de programas têm perseguido estes objectivos.

2.3 Ferramentas CASE

A sigla *CASE* significa 'engenharia de software assistida por computador' (*Computer Aided Software Engineering*). *Ferramentas CASE* são programas de computador que auxiliam a tarefa de desenvolvimento de um sistema, desde a elaboração da especificação à produção da documentação [113].

Um sistema CASE pode compreender vários tipos de ferramentas. Existem editores de diagramas para gestão de vários tipos de informação: fluxo de dados, estrutura do sistema, relações entre entidades, etc. Estes editores são geralmente mais do que meras ferramentas de desenho. Devem ser capazes de capturar a informação contida nos diagramas e alertar o utilizador para inconsistências e outras anomalias. Para além dos editores podemos ter também ferramentas de interpelação à base de dados, dicionários que mantêm a informação relativa a entidades envolvidas, ferramentas que permitem a geração fácil de mapas e listagens (*reports*), do interface com o utilizador, etc.

As ferramentas CASE propiciam um aumento de produtividade no desenvolvimento de sistemas complexos, sendo hoje em dia comuns em ambientes profissionais. O seu principal papel é organizar a grande quantidade de informação envolvida num projecto de desenvolvimento de envergadura de maneira a que essa informação seja facilmente

acedida pelos elementos envolvidos no projecto. Além disso, os sistemas produzidos com o auxílio de ferramentas CASE têm tendência a ser mais fiáveis.

Alguns sistemas CASE incluem *geradores de código*. Estes são capazes de gerar segmentos preliminares de código (*skeleton code*) a partir de informação recolhida nos editores de diagramas e no dicionário de dados. No entanto, a oferta deste tipo de ferramentas CASE é muito limitada.

Em conclusão, as ferramentas CASE são principalmente úteis no apoio à gestão do projecto de desenvolvimento, deixando ainda muitas tarefas rotineiras para o programador. No entanto, sem ferramentas capazes de automatizar ou semi-automatizar a geração de código a tecnologia CASE está longe de atingir todo o seu potencial [35].

2.4 Métodos formais

Nas metodologias formais de desenvolvimento a programação é vista como uma actividade matemática, e os programas são considerados expressões matemáticas complexas [45]. Esta concepção da programação permite, por exemplo, provar que um programa está correcto de acordo com a sua especificação [30].

Em abordagens baseadas nos métodos formais a especificação é expressa numa linguagem rigorosa, como a lógica de primeira ordem [28], em vez da linguagem natural. O programa executável pode ser obtido a partir da especificação dada utilizando regras de inferência e/ou de reescrita. A aplicação dessas regras pode ser manual ou semi-automática. Em geral é difícil derivar mecanicamente um programa complexo dessa forma [28]. É por isso mais frequente encontrarmos metodologias de síntese de programas semi-automáticas guiadas pelo utilizador. Vejamos dois exemplos.

O sistema KIDS de Douglas Smith apoia o desenvolvimento de programas correctos e eficientes a partir de especificações formais (cf. Secção seguinte). O ambiente de desenvolvimento de KIDS é altamente automatizado, embora seja interactivo. O

utilizador toma decisões de alto nível em termos de planeamento do programa (*design*) e o sistema concretiza essas decisões gerando o código [111].

Jüllig [55] propõe um ambiente (REACTO) de desenvolvimento onde o espírito das ferramentas CASE é integrado com os métodos formais, e com a síntese de programas. Por um lado, são postas à disposição do utilizador ferramentas gráficas de apoio à análise, por outro lado o utilizador pode escrever especificações formais e obter o código. Um dos componentes de REACTO é o sistema KIDS referido anteriormente.

Em [55] faz-se também uma interessante distinção entre ferramentas CASE, métodos formais e síntese de programas. As ferramentas CASE providenciam apoio gráfico à análise mas dão pouco apoio na geração de código. Os métodos formais são mais usados para escrever as especificações do que para desenvolver código, e os sintetisadores de programas, discutidos na Secção seguinte, concentram-se na construção do código em desfavor da análise do sistema.

2.5 Síntese de programas

De uma maneira geral, a designação *síntese de programas* (*program synthesis*) aplica-se a processos sistemáticos de elaboração de programas a partir de uma especificação que descreve o que o programa deve fazer [27]. Dentro desta categoria de processos sistemáticos, cabem naturalmente os processos automáticos ou semi-automáticos de geração de código a partir de uma especificação do comportamento do programa pretendido. Neste caso a síntese de programas é também designada por *programação automática* [8,99].

Na síntese de programas uma especificação pode tomar diversas formas. Biermann organiza a investigação em programação automática em função da natureza das especificações [8]: síntese a partir de especificações formais (fórmulas de lógica de primeira ordem); síntese a partir de exemplos de resultados (*output*) do programa para

diferentes dados (*input*); síntese a partir de diálogos em linguagem natural entre o sistema de síntese e o utilizador. Uma especificação pode tomar outras formas como, por exemplo, a de uma máquina hierárquica de estados finitos [55] ou de uma lógica temporal [108].

Nas especificações expressas em linguagem natural os geradores de código deparam-se tipicamente com o problema da ambiguidade da linguagem natural, para além de terem de lidar com linguagens sintacticamente mais irregulares. É comum os sistemas de síntese a partir de linguagem natural serem interactivos permitindo que o utilizador faça a descrição do problema através de um diálogo com o sistema.

Nos anos 70 houve alguns projectos ambiciosos neste domínio [42]. O sucesso destes projectos foi limitado tendo-se mais tarde deslocado o foco de atenção para as especificações em *linguagens de muito alto nível* (*very high level languages*). Estas linguagens de especificação aproximam-se bastante das linguagens formais, embora por vezes permitam alguma informalidade típica das linguagens naturais [99,parte v].

2.5.1 Síntese de programas lógicos

Dentro do campo da síntese de programas estamos principalmente interessados na *síntese de programas lógicos*. Neste campo, Deville e Lau [27] dividem as especificações em *formais* e *informais*, podendo as especificações formais ser *completas* ou *incompletas*.

Uma especificação formal é expressa utilizando uma linguagem formal, como a lógica de primeira ordem ou um seu sub-conjunto. A especificação é um conjunto de fórmulas lógicas envolvendo um predicado *r* que se pretende definir através de um programa lógico a sintetizar. Esta noção de especificação no contexto da síntese de programas lógicos é suficientemente abrangente para incluir especificações completas e incompletas.

Uma *especificação completa* reúne todas as condições que o programa a sintetizar deve satisfazer. Uma *especificação incompleta* descreve parte dessas condições. Em geral

uma especificação a partir de exemplos de respostas de um programa lógico é incompleta, i.e., nem todo o comportamento do programa é especificado. Neste caso o gerador de código terá a tarefa de hipotetizar o comportamento não especificado. Uma especificação a partir de exemplos pode no entanto ser considerada uma especificação formal, desde que seja utilizada uma linguagem rigorosa para descrever os exemplos [27].

Exemplo 2.1: (de [27]) Duas especificações para o predicado *included(X,Y)* que define o conjunto de pares <*X*,*Y*>, tais que *X* e *Y* são listas e todo o elemento de *X* está contido em *Y*.

Especificação completa:

$\{included(X,Y) \leftrightarrow \forall A(\ member(A,X) \rightarrow member(A,Y)\ )\ \}$

Especificação incompleta (através de exemplos):

$\{\ included([],[2,1]),\ included([1,2],[1,3,2,1]),\ \neg included([2,1],[])\ \}$ ♦

A síntese de programas lógicos a partir de especificações formais é dividida em três tipos de abordagem:

- *síntese construtiva*, onde a partir de uma prova construtiva da existência de um programa que satisfaz a especificação se extrai o programa;
- *síntese dedutiva*, onde a partir de uma especificação se deriva um programa utilizando regras de dedução;
- *síntese indutiva*, onde se constrói um programa que generaliza a informação incompleta contida na especificação utilizando métodos indutivos.

De entre estes três tipos de abordagem à síntese de programas a partir de especificações formais, o nosso trabalho insere-se na síntese indutiva. Em particular na *síntese indutiva a partir de exemplos* do comportamento do programa pretendido.

2.5.2 Síntese de programas a partir de exemplos

No início dos anos 80 o sistema MIS (*Model Inference System*) de Ehud Shapiro [109] sintetiza programas em linguagem Prolog a partir de exemplos fornecidos pelo utilizador.

O funcionamento do MIS é interactivo e segue uma filosofia de *depuração de programas* (*debugging*). O utilizador vai apresentando exemplos positivos e negativos ao sistema e este vai confrontando os exemplos conhecidos com os programas que vai gerando, começando pelo programa vazio. Quando um novo exemplo põe a descoberto um erro num programa, o sistema altera o programa de forma a que o erro seja eliminado.

A correcção de um programa pode consistir na eliminação, na criação ou na modificação de uma cláusula. Durante o processo de correcção o sistema pode interpelar o utilizador, fazendo perguntas sobre os predicados envolvidos na definição. Estas perguntas têm a forma "*p(a)* é verdadeiro ou falso?" (*membership queries* ou perguntas de pertença) e "para que valores *p(X)* é verdadeiro, sendo *X* uma variável?" (*existencial queries* ou perguntas existenciais).

O MIS consegue gerar programas em Prolog de pequenas dimensões, tais como *member/2*, que é verdadeiro se o primeiro argumento é membro do segundo, ou *append/3*, que faz a concatenação de duas listas numa terceira. O sistema também pode ser adaptado para sintetizar programas em notação DCG (*definite clause grammar*).

O trabalho de Ehud Shapiro contém uma metodologia de síntese de programas Prolog a partir de exemplos que ainda hoje é adoptada por alguns investigadores [41,95]. A influência do seu trabalho reflecte-se hoje em dia sobretudo no campo da *Programação Lógica por Indução* (PLI, cf. Capítulo 2).

Com a PLI, no início da década de noventa [77], surgem vários sistemas de geração de programas em Prolog a partir de exemplos. Embora o principal foco de interesse da investigação em PLI não tenha sido a programação automática muitos sistemas de PLI

apresentam como exemplos da sua aplicação a geração de programas Prolog do nível de um primeiro curso de programação lógica. Como exemplo de algumas abordagens mais orientadas para a programação automática, podemos citar os trabalhos de Quinlan [96,97], Bergadano et al.[5, 6], Flener [37,39] e Popelinsky et al. [94].

Embora recentemente a síntese indutiva tenha preferido de uma maneira geral as linguagens de programação lógica como o Prolog, nas décadas de 70 e 80 era mais estudada a síntese de programas em linguagem LISP, mais próxima do paradigma funcional. Trabalhos como os de Summers [118] e de Biermann [7] são bem exemplo disso.

A deslocação do paradigma ter-se-á devido à adequação da programação lógica à tarefa de induzir cláusulas a partir de exemplos particulares e à crescente popularidade do Prolog. Para essa mudança muito contribuiu o facto de as operações de generalização e especialização de um programa serem naturais numa linguagem baseada na lógica correspondendo a operações muito simples[4] [109]. Apesar do facto, continua-se a publicar sobre síntese indutiva de programas em linguagens funcionais. Um exemplo disso é o sistema ADATE de 1995 [89] que sintetiza programas na linguagem ML.

Em conclusão podemos dizer que tanto linguagens de programação lógica como linguagens funcionais têm características importantes que as tornam favoritas para a síntese de programas (e não só a partir de exemplos). Ambos os paradigmas apresentam facilidades ao nível da meta-programação, o que se revela importante para a programação automática. Tanto LISP como Prolog resultam em programas compactos que são relativamente fáceis de compreender. Por último, ambos os tipos de linguagens (funcionais e lógicas) têm fortes fundamentos teóricos, o que proporciona uma formalização das operações de indução e de transformação de programas [109, pp.162-163].

2.6 Outros tópicos relevantes

Outros tópicos de ciência dos computadores são relevantes para diminuir o esforço de desenvolvimento dos programadores e analistas de programas. Não os vamos referir de forma sistemática, deixando apenas alguns apontadores que podem ser seguidos.

- *Ambientes de desenvolvimento (*environments*)*. Em particular, na área de programação em lógica destacamos o trabalho de Mireille Ducassé [31]. Um exemplo interessante de como um ambiente de desenvolvimento pode ajudar um programador de FORTRAN a tirar partido de uma biblioteca de rotinas é o trabalho de Stickel et al. [117].

- *Depuração algorítmica (*algorithmic debugging*)*. É uma fonte de inspiração para a síntese a partir de exemplos, uma vez que se pode ver o processo de síntese como um processo de depuração que começa pelo programa vazio. Aqui referimos os trabalhos de Shapiro [109], Moniz Pereira e Miguel Calejo [17,91] e Paaki et al.[90].

- *Programação por demonstração (*programming by demonstration*)*. O objectivo da programação por demonstração é, segundo Cypher, o seguinte: se um utilizador sabe como desempenhar uma tarefa no computador, isso devia ser suficiente para criar um programa que automatizasse essa tarefa. Não devia ser necessário saber uma linguagem de programação como C ou BASIC. Tipicamente, o utilizador demonstra as suas intenções através de um interface gráfico. O sistema de programação por demonstração generaliza as acções do utilizador e infere um programa ou uma macro [19].

[4] Estas operações são simples quando o modelo de generalização utilizado é a subsunção-θ. Outros modelos de generalização podem ser mais complexos.

2.7 Sumário

As ferramentas CASE são uma boa ajuda para as tarefas de análise e de gestão do projecto, aumentando a produtividade global do desenvolvimento de *software* e a fiabilidade deste. No entanto, a metodologia CASE tem oferecido muito pouco a nível de geração de código, deixando ainda muitas tarefas rotineiras para o programador.

Os métodos formais de desenvolvimento vêem a programação como uma actividade matemática e os programas como expressões matemáticas complexas [45]. Propõem linguagens formais de especificação, a partir das quais se obtém o código seguindo uma metodologia formal de desenvolvimento. A correcção dos programas desenvolvidos em relação à especificação pode ser provada. Através dos processos rigorosos consegue-se eliminar grande parte dos erros. Este factor é de especial importância em aplicações críticas (como por exemplo o controlo de tráfego aéreo ou a manutenção de uma unidade industrial) onde um erro de programação pode ter custos dramáticos [30].

Ao desenvolvimento sistemático de programas a partir de especificações chamamos síntese de programas. As especificações podem ser formais ou informais, completas ou incompletas. A partir de especificações formais e completas os programas são derivados utilizando métodos dedutivos semelhantes aos usados para a demonstração de teoremas (*theorem proving*).

A maquinaria formal, no entanto, é muitas vezes muito pesada. A programação passa a estar apenas ao alcance de especialistas. Escrever uma especificação formal e completa não é fácil, e os métodos de derivação existentes não estão totalmente automatizados, exigindo ainda muito do programador/utilizador. A síntese de programas a partir de especificações formais e incompletas propõe algumas soluções para aliviar o programador do enorme esforço de abstracção exigido pelos métodos formais tradicionais.

Nesta área de síntese indutiva têm surgido principalmente trabalhos orientados para a geração de programas em linguagem LISP e em linguagem Prolog. Enquanto que a síntese de programas LISP a partir de exemplos tem evoluído pouco na década de 90, a síntese de programas Prolog multiplicou-se com o crescimento das áreas de Programação Lógica e de Programação Lógica Indutiva (PLI).

A PLI poderá vir a ser uma componente importante de futuros ambientes de programação. O objectivo de termos um dia uma ferramenta capaz de construir qualquer programa construído a partir de exemplos somente (o que Biermann designa por *programação auto-mágica* (*auto-magic programming*)) parece pouco realista. Todavia, a PLI pode dar um contributo importante para ajudar o programador a desempenhar a sua função, libertando-o de tarefas rotineiras. Por outro lado, a PLI pode permitir que utilizadores de computadores menos especializados possam desenvolver pequenos programas sem programar, aumentando enormemente a facilidade de construção de novas aplicações.

3. Programação Lógica por Indução

Neste capítulo apresentamos conceitos relevantes da Programação Lógica por Indução (PLI), área onde o nosso trabalho se insere. Começamos pela própria Programação Lógica, que se pode considerar um dos pilares da PLI. O Problema genérico da PLI é definido como a construção de um programa lógico que satisfaz determinadas condições. Descrevem-se as principais abordagens para a resolução deste problema terminando o capítulo com um resumo do estado da arte em PLI.

3.1 Introdução

Situada na intersecção entre a *Programação Lógica* (PL) e a *Aprendizagem Automática* (AA), a Programação Lógica por Indução (PLI) procura processos de geração de programas lógicos a partir de exemplos e por isso utiliza muito do enquadramento teórico da programação lógica. Nas secções seguintes, reunimos todo o vocabulário da programação lógica relevante para este trabalho, e fazemos uma descrição da Programação Lógica por Indução, focando o que nos é mais relevante.

3.2 Programas lógicos

Um programa lógico, no contexto deste trabalho, é um conjunto de fórmulas da *Lógica de Primeira Ordem* escritas em forma clausal e designadas por cláusulas. Os conceitos referidos nesta Secção seguem principalmente a notação e terminologia usada por Hogger [46] e Lloyd [64]. Este último contém uma abordagem detalhada da teoria da programação lógica.

3.2.1 Sintaxe

Uma *cláusula* é uma fórmula da lógica de primeira ordem em forma clausal:

$$\forall X_1, \ldots, X_s \, (L_1 \vee L_2 \vee \ldots \vee L_n)$$

em que cada L_i é um literal, e $X_1, \ldots, X_s$ são todas as variáveis que ocorrem na cláusula. Um *literal* é um átomo (literal positivo) ou um átomo negado (literal negativo). Um *átomo* é uma expressão da forma $p(t_1,t_2,...,t_k)$, com $k \geq 0$, onde p é o nome de um predicado de *aridade k*, também representado como *p/k*. Os t_i são os argumentos do átomo. Cada argumento t_i é um *termo*. Um átomo negado é da forma $\neg p(t_1,t_2,...,t_k)$.

Um termo pode ser uma *variável*, uma *constante*, ou um termo composto da forma $f(t_1,t_2,...,t_n)$, em que f é um *functor* de aridade $n > 0$ e os t_i são termos.

Por vezes referimo-nos a uma cláusula como o conjunto dos seus literais $\{L_1, L_2, \ldots, L_n\}$. Outra forma habitual de se representar uma cláusula é reescrevendo-a sob a forma de uma implicação

$$A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge ... \wedge B_n$$

em que cada A_i é um literal positivo, e cada B_i é o átomo de um literal negativo. A parte da cláusula correspondente a $A_1 \vee A_2 \vee \ldots \vee A_m$ é chamada *cabeça* da cláusula ou *consequente*. A $B_1 \wedge B_2 \wedge ... \wedge B_n$ chama-se *corpo* da cláusula ou *antecedente*.

As cláusulas podem ser classificadas de acordo com o número de literais positivos e negativos. Assim, uma cláusula como um e um só literal positivo (um e um só literal na cabeça) e zero ou mais literais negativos é uma *cláusula definida* (*definite clause*). Qualquer cláusula com mais do que um literal positivo é uma *cláusula indefinida* (*indefinite clause*). Uma cláusula sem literais designa-se por *cláusula vazia* (*empty clause*) e denota-se habitualmente por um quadradinho branco □. A cláusula vazia representa a contradição *falso* ← *verdadeiro*.

Uma *cláusula recursiva* é aquela onde pelo menos um dos literais do corpo tem o mesmo predicado que o literal da cabeça. Quando uma cláusula não contém variáveis diz-se uma *cláusula fechada* (*ground clause*). De forma idêntica usam-se as designações *literal fechado* e *termo fechado*. Uma cláusula fechada com um único literal positivo é um *facto*.

Um *programa lógico P* é um conjunto de cláusulas possivelmente vazio. O programa vazio denota-se por Ø. Para representarmos programas lógicos nós usaremos, por conveniência, uma notação idêntica à da linguagem de programação em lógica Prolog [116]. Uma cláusula é representada substituindo os símbolos de disjunção e de conjunção por vírgulas, terminando com um ponto final. A nossa notação difere da do Prolog num aspecto, utilizamos a seta (←) em vez do dois-pontos-traço (:-) do Prolog.

$$A_1, A_2, \ldots, A_m \leftarrow B_1, B_2, \ldots, B_n .$$

As variáveis são denotadas por palavras iniciadas com uma letra maiúscula (tal como *X*, *Y*, *A*, etc.), as constantes são denotadas por palavras iniciadas por uma letra minúscula (tal como *a*, *c*, *x*, etc.).

Um *programa lógico normal* contém cláusulas com pelo menos um literal positivo. Cada cláusula tem a forma

$$A \leftarrow L_1, \ldots, L_n.$$

em que A é um átomo e os L_i são literais. Uma cláusula de um programa normal define o predicado *p/k* do átomo *A*. A definição de um predicado *p/k* num programa *P* é o conjunto das cláusulas de *P* que definem *p/k*. Um programa lógico que contém apenas cláusulas definidas chama-se um *programa lógico definido*.

Exemplo 3.1: O seguinte conjunto de cláusulas definidas é um exemplo de um programa lógico:

progenitor(X,Y)←pai(X,Y).
progenitor(X,Y)←mãe(X,Y).
antepassado(X,Z)←progenitor(X,Y), antepassado(Y,Z).

Trata-se de um programa lógico definido (e normal) que define os predicados *progenitor/2* e *antepassado/2*.♦

3.2.2 Semântica

A teoria de modelos de Herbrand proporciona um ponto de partida para definirmos a semântica de um programa lógico. Um *modelo de Herbrand* de um programa lógico *P* é, informalmente, um conjunto de átomos fechados que validam logicamente cada cláusula de *P*. Os átomos fechados de que falo são elementos da *base de Herbrand* do programa *P*. A base de Herbrand, por sua vez, é o conjunto de átomos fechados que se podem construir utilizando os predicados contidos em *P* e quaisquer functores e constantes que pertençam à linguagem.

Um facto *q* diz-se uma *consequência lógica* de um programa *P* quando todos os modelos de *P* (de Herbrand ou não) também forem modelos de *q*. Escreve-se

$P \models q$

Um programa definido (o que exclui programas que contenham cláusulas com literais negativos no corpo) tem um conjunto de modelos de Herbrand que se organizam num reticulado (*lattice*) segundo a relação de ordem parcial $\subseteq$ entre conjuntos. Ao elemento

mínimo deste reticulado chama-se *modelo mínimo* de Herbrand. Usa-se a notação *MM*(*P*) para referir o modelo mínimo (de Herbrand) de *P*.

O modelo mínimo de um programa definido *P* corresponde exactamente ao conjunto dos factos fechados *q* que são suas consequências lógicas.

$P \models q$ se e só se $q \in MM(P)$

Em termos de semântica denotacional, o *significado de um programa lógico* definido *P* é *MM*(*P*). Por outras palavras, é o conjunto das consequências lógicas fechadas de *P*.

A semântica de um programa normal *P* com literais negados no corpo de alguma das suas cláusulas é definida a partir de do seu *completamento* (*completion*) Comp(*P*). O completamento de um programa consiste basicamente em transformar as cláusulas em equivalências, e em acrescentar cláusulas que definem uma teoria de igualdade [64].

Para um programa normal *P*, em vez de falarmos do modelo de *P*, falamos do modelo de Comp(*P*). Todavia, por uma questão de simplificação da exposição, iremos sempre referirmo-nos ao modelo de um programa *P*, ainda que não se trate de um programa definido. Devemos contudo chamar a atenção para o facto de que muitos resultados relativos a programas definidos não são válidos para a generalidade dos programas normais. Identificaremos essas diferenças sempre que tal for relevante.

Os programas sintetizados pela nossa metodologia são definidos. O sistema pode no entanto utilizar programas normais não definidos como conhecimento de fundo.

3.2.3 Derivação

Um programa lógico é executado pondo-lhe *perguntas* (*queries*). Uma pergunta é uma cláusula da forma $\leftarrow L_1,\ldots,L_n$ em que os L_i são literais. Basicamente uma pergunta $\leftarrow q$ a um programa *P* indaga se um facto fechado *q* é (ou não) uma consequência lógica de *P* ou se existe algum valor que se possa dar às variáveis contidas em *q* de modo a que *q*

seja uma consequência lógica fechada de *P* após substituirmos as variáveis pelos respectivos valores.

Uma resposta a uma pergunta pode ser *sucesso* (*success*) ou *falha* (*failure*). Em caso de sucesso e de a pergunta ter variáveis faz também parte da resposta uma *substituição* (chamada *substituição resposta*). Uma substituição é uma aplicação entre um conjunto de variáveis e um conjunto de termos e é representada como um conjunto de pares variável/termo. Ao processo de substituição de variáveis por termos chama-se *instanciação*. As substituições são habitualmente denotadas por letras gregas tais como θ e σ. Ao processo de substituição de variáveis por termos chama-se *instanciação*.

Um conceito fundamental é o de *unificação*. Dois átomos são unificáveis quando podem ser tornados idênticos substituindo as suas variáveis por termos de uma forma consistente. A uma substituição que torna dois átomos idênticos chama-se *unificador*. Entre todos os unificadores de dois átomos unificáveis existe um único *unificador mais geral*, que em termos simples corresponde à substituição que menos instancia os átomos em questão.

Exemplo 3.2: Os átomos *f(a)* e *f(X)* são unificáveis. A substituição θ = *{X/a}* é um unificador. Aplicando esta substituição ao segundo átomo obtemos o primeiro, *f(a)* = *f(X)*θ. A mesma substituição é também o unificador mais geral dos dois átomos.♦

A *resolução* é uma regra de inferência que permite derivar uma cláusula *R* a partir de outras duas cláusulas C_1 e C_2, chamadas cláusulas mães (*parent clauses*). A cláusula *R* é designada por *resolvente*. As cláusulas mães devem ser complementares, i.e., para algum literal A_1 de uma delas deve existir um literal $\neg A_2$ na outra de tal forma que A_1 e A_2 são unificáveis. Assim, se C_1 for a cláusula $A_1 \vee$ *Mais-Literais* e C_2 for $\neg A_2 \vee$ *Outros-Literais*, a resolvente *R* será (*Mais-Literais* $\vee$ *Outros-Literais*)θ, em que θ é o unificador mais geral de A_1 e A_2.

Uma resposta a uma pergunta Q é derivada a partir de P utilizando um *procedimento de prova*, o qual é um algoritmo que aplica um conjunto de regras de derivação a P e Q segundo uma determinada estratégia, construindo uma *prova*. O procedimento de prova que utilizamos neste trabalho é a *resolução-SLDNF*, a qual é usada nos interpretadores da linguagem Prolog. A resolução-SLDNF é, por sua vez, uma extensão da *resolução-SLD*.

A resolução-SLD processa-se da seguinte maneira. Dada uma pergunta Q da forma

$$\leftarrow Q_1,Q_2,...,Q_m$$

em que cada Q_i é um literal não negado, e dado um programa definido P, a resolução SLD começa por seleccionar um dos literais da pergunta. Aqui vamos assumir que a regra de selecção de literais escolhe sempre o literal mais à esquerda, pois essa é a regra de selecção mais frequentemente utilizada. Sendo assim, o primeiro literal a ser escolhido é Q_1.

A seguir escolhe-se uma cláusula C_1 do programa P, de tal forma que a cabeça de C seja unificável com Q_1. Supondo que C_1 é da forma

$$A\leftarrow B_1,B_2,...,B_n$$

então, $A\theta_1=Q_1\theta_1$, sendo θ_1 o unificador mais geral de A e Q_1.

Assim, obtemos a cláusula resolvente R_1

$$\leftarrow (B_1,B_2,...,B_n,Q_2,...,Q_m)\theta_1$$

Depois deste primeiro passo de resolução vamos proceder de igual forma com a resolvente R_1, seleccionando uma cláusula C_2 do programa P e obtendo um novo unificador θ_2 e uma nova resolvente R_2, até que se obtém como resolvente a cláusula vazia $\square$.

Uma *derivação-SLD* de um programa P a partir da pergunta $\leftarrow Q$ representa-se pela sequência $D = ((R_1,C_1,\theta_1), (R_2,C_2,\theta_2), \ldots , R_n)$, em que $R_1 = \leftarrow Q$, R_i é a resolvente entre R_{i-1} e C_{i-1}, e θ_i é o respectivo unificador, para $1 \leq i \leq n$. Uma *refutação* de $\leftarrow Q$ a partir de um programa P é uma derivação de P a partir de $\leftarrow Q$ que termina na cláusula vazia ($R_n = \square$). Refutando $\leftarrow Q$ a partir de P, prova-se $Q\theta$ a partir de P, sendo θ uma substiuição resposta. Diz-se também $Q\theta$ é derivável a partir de P.

A substituição resposta θ à pergunta inicial é obtida por composição da sequência de unificadores mais gerais θ_1, θ_2, etc. Caso a resolvente vazia não seja derivável a pergunta falha.

Quando um facto q é derivável por SLD a partir de um programa P escreveremos

$$P \vdash q$$

O conjunto de todas as substituições resposta dadas pela resolução SLD a uma pergunta é construído fazendo uma procura exaustiva no espaço das derivações segundo uma estratégia de procura. A *árvore-SLD* resultante desta procura tem como raíz a pergunta, sendo cada ramo uma derivação possível do programa para essa pergunta. Cada ramo pode terminar na cláusula vazia, o que corresponde a uma resposta, numa cláusula não vazia que não resolve com nenhuma cláusula do programa, ou pode tratar-se de um ramo infinito. Quando todos os ramos de uma árvore-SLD de uma pergunta $\leftarrow Q$ são finitos e nenhum termina em $\square$, diz-se que a pergunta falha finitamente.

Exemplo 3.3: Suponhamos que temos o seguinte programa P:

descende(X,Y)←filho(X,Y).
descende(X,Z)←filho(X,Y),descende(Y,Z).
filho(alipio,antonio).
filho(antonio,adriana).

e queremos responder à pergunta ←*descende(alipio,X)*. A resposta θ={*X/antonio*} é dada pela seguinte derivação:

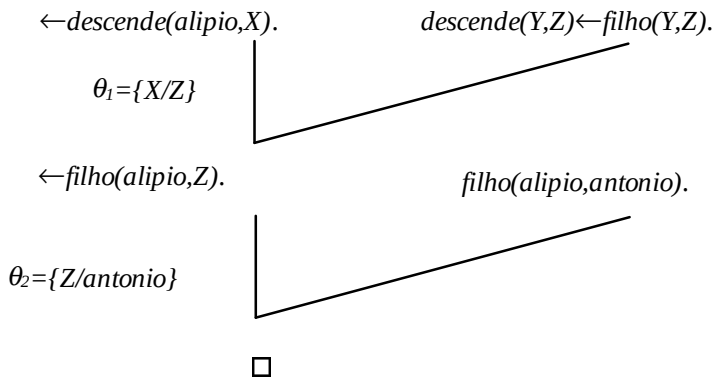

Figura 3.1: Grafo de derivação

Temos $P \vdash$ *descende(alipio,antonio)*.♦

É importante notar que o conjunto dos factos deriváveis por resolução-SLD a partir de um programa definido *P* corresponde exactamente ao modelo mínimo de Herbrand de *P*, i.e.,

$$P \vdash q \Leftrightarrow P \models q \Leftrightarrow q \in MM(P)$$

Por outras palavras, podemos usar a resolução-SLD para determinar quais são as consequências lógicas fechadas de um programa *P*.

Para podermos lidar com perguntas com literais negados e programas não definidos, necessitamos de estender a resolução SLD com a regra de *negação por falha* (*negation as failure*) de maneira a que possamos tratar a negação lógica. Assim se obtém a resolução-SLDNF. O operador de negação por falha é habitualmente denotado por *not* e é definido da seguinte forma: uma pergunta ←*not Q* sucede se e só se a pergunta ←*Q* falha finitamente.

Quando na sequência de uma derivação a partir de um programa *P* se encontra um literal *not Q* numa resolvente podem ocorrer duas situações. A primeira possibilidade é a de

que Q é derivável, e nesse caso *not* Q não pode ser eliminado da resolvente. A segunda possibilidade é a de que existe uma árvore-SLDNF T de Q a partir de P, finita e sem ramos a terminar em □ (Q é derivável a partir de P). Por isso representamos esse passo de derivação como $(\leftarrow Lits_1 \wedge not\ Q \wedge Lits_2, T), (\leftarrow Lits_1 \wedge Lits_2, C, \theta)$.

Impondo certas condições ao programa P podemos relacionar os factos fechados q deriváveis a partir de P por SLDNF com as consequências lógicas fechadas de $Comp(P)$.

$$P \vdash_{\text{SLDNF}} q \Leftrightarrow Comp(P) \models q$$

Resumidamente, as condições são as seguintes: nenhum predicado p de P deve ser expresso directa ou indirectamente em termos de *not* p; todas as variáveis de uma cláusula devem ocorrer pelo menos uma vez fora de um literal negado; P deve ser estrito (*strict*) em relação a qualquer pergunta q. Para uma definição de programa estrito ver [46]. Para além destas condições, a resolução-SLDNF nunca deve seleccionar um literal negado não totalmente instanciado.

A relação estabelecida entre programas lógicos (conjuntos de cláusulas de uma linguagem L) e factos (elementos da linguagem L), através de um conjunto R de regras de derivação, é o que se designa por *relação de derivabilidade*.

$$\vdash = \{<P,q> \mid P \subseteq L,\ q \in L,\ q \text{ é derivável a partir de } P \text{ usando } R\}$$

Um procedimento de prova permite construir a relação de derivabilidade. Por uma questão de legibilidade, utilizaremos o símbolo $\vdash$ para denotar tanto a derivação por resolução-SLD como por resolução-SLDNF.

Definição 3.1: Dada uma linguagem L, uma relação de derivabilidade $\vdash$, um programa $P \subseteq L$ e uma pergunta $\leftarrow q$ tal que $q \in L$, um *interpretador* da linguagem L é o operador,

$$Int(P, \leftarrow q, \vdash) = \{\ \theta \mid P \vdash q\theta\ \}$$

♦

Cada elemento de $Int(P,\leftarrow q, \vdash)$ é uma substituição resposta dada por $\vdash$ para uma pergunta $\leftarrow q$ colocada a P.

3.2.4 Tipos, modos de entrada/saída

Um *tipo* corresponde a um conjunto não vazio de termos fechados. Esse conjunto é designado por domínio do tipo (*type domain*) ou, abreviadamente por tipo. A cada um dos argumentos de um predicado podemos associar um tipo. No presente trabalho, esta associação é feita através de uma *declaração de tipos* $type(p(tipo_1,...,tipo_k))$ aquando da especificação do programa que define o predicado *p/k* (Secção 4.3.1).

O tipo dos argumentos serve como condição que deve ser satisfeita pelas perguntas postas ao programa e pelas substituições resposta [26]. Um n-tuplo de termos $(A_1,...,A_n)$ é *compatível* com um n-tuplo de tipos $(tipo_1,...,tipo_n)$ se existe uma substituição θ tal que $(A_1,...,A_n)\theta \in (tipo_1\times...\times tipo_n)$.

Exemplo 3.4: Especificamos o tipo dos argumentos de *member/2* como $(X,Y)\in(inteiro\times lista)$. Esta informação serve tanto como pré-condição como pós-condição. Como pré-condição é usada para filtrar as perguntas de *member/2*. Antes de uma pergunta $\leftarrow member(A,B)$ ser executada, é verificado se (A,B) é compatível com $(inteiro,lista)$. Como pós-condição serve para verificar se a substituição resposta θ dada é tal que $(A,B)\theta \in (inteiro\times lista)$. ♦

Uma vantegem das declarações de tipos ajudam o programador a estruturar os programas lógicos. Outra vantagem é a de que permitem que a execução desses programas seja mais eficiente [26]. As declarações de tipos interessam-nos principalmente como um factor de eficiência na programação lógica por indução (ver Secção 4.9 e [120]).

O *modo de entrada/saída* de um predicado (ou simplesmente modo) determina os usos possíveis do programa que o define [26,64]. Para cada argumento do predicado é

definida uma condição de entrada ou de saída. As condições de entrada devem ser verificadas antes da execução do programa lógico e as de saída após ser obtida a substituição resposta. A mais simples condição de entrada é "o argumento deve ser um termo fechado". Outra condição pode ser, por exemplo, "o argumento deve ser uma variável". As condições de saída são semelhantes.

Os modos de entrada/saída que se usam mais frequentemente em PLI determinam quais os argumentos de um predicado que devem ser termos fechados antes da execução [82,96,109]. Por esse motivo são chamados os *argumentos de entrada*. Os restantes argumentos são chamados *argumentos de saída*. Aqui, a declaração de modo de entrada/saída de um predicado *p/k* é da forma

mode(p(M_1*, …,*M_k*)).*

em que M_i é um sinal mais '+' se o *i*-ésimo argumento for de entrada, e um sinal menos '-' caso contrário.

Exemplo 3.5: O modo de um predicado *p(X,Y)* pode especificar que este predicado deve ser invocado com a variável *X* instanciada. A variável *Y* pode estar instanciada ou não. Chamamos a *X* um argumento de entrada e a *Y* argumento de saída. O modo do predicado *p/2* é expresso como *p(+,-).*♦

Por conveniência usaremos por vezes a seguinte notação. Os sinais '+' ou '-' precedendo os argumentos de um literal numa cláusula significam que esses argumentos têm o modo 'entrada' ou 'saída', respectivamente. Assim, o literal *p(+a,+b,-c)* corresponde ao literal *p(a,b,c)* sendo o modo de entrada/saída do predicado *p/3 p(+,+,-).*

3.2.5 Restrições de integridade

As restrições de integridade são fórmulas da lógica de primeira ordem da forma $A_1 \wedge ... \wedge A_k \rightarrow B_1 \vee ... \vee B_n$, em que A_i e B_j representam literais. As restrições de integridade não são, em geral, representáveis por cláusulas definidas. São utilizadas em aplicações da

programação em lógica tais como bases de dados dedutivas [71,121] e programação lógica por indução. Tanto numa como noutra aplicação, estas cláusulas especiais servem para impedir que um programa lógico seja alterado de forma indesejável. No Capítulo 7 abordamos as restrições de integridade com mais detalhe, em particular no que se refere à sua aplicação à PLI.

3.3 O Problema da PLI

Se na Programação Lógica se parte dos programas para chegar às suas consequências lógicas, na *Programação Lógica por Indução* pretende-se partir das consequências lógicas para chegar aos programas. A descrição das consequências lógicas do programa pretendido é feita através de *exemplos positivos* e *negativos*. Estes exemplos são, na maior parte das vezes, átomos fechados[5]. Os exemplos positivos, sendo factos fechados, são como que amostras do modelo mínimo do programa pretendido. Os limites do modelo do programa pretendido são indicados pelos exemplos negativos: factos fechados que não devem ser consequência lógica do programa.

A tarefa indutiva consiste em procurar um programa *P* que seja uma hipótese compatível com os exemplos apresentados. Esta hipótese é procurada dentro de uma *linguagem de hipóteses L* (também chamada *linguagem de conceitos*). A linguagem *L* é um conjunto de programas lógicos. Diz-se então que o programa *P* é *induzido*, *sintetizado* ou *aprendido*. A tarefa é designada por *indução*, *síntese indutiva*, *síntese de programas* a partir de exemplos ou simplesmente *aprendizagem* através de exemplos. Esta multiplicidade de termos deve-se ao facto de este problema ser do interesse de diferentes comunidades dentro da ciência da computação e da inteligência artificial. Nós usaremos principalmente a designação *síntese indutiva de programas a partir de especificações incompletas*. Por comodidade, diremos por vezes que *P* é 'o programa pretendido' ou 'o

[5] Existem, no entanto, abordagens que utilizam cláusulas não fechadas para representarem exemplos positivos e negativos, como é o caso de [37,102], e do nosso próprio trabalho apresentado nesta dissertação.

programa a sintetizar', embora em geral haja mais do um programa aceitável para um problema de síntese.

Como acontece em muitas outras tarefas de Aprendizagem Automática, é de todo o interesse que a tarefa de síntese de um programa não comece do zero. Havendo outros predicados que possam ser usados como auxiliares nas cláusulas do programa a induzir, estes podem ser dados para a tarefa de síntese como *conhecimento de fundo* (*background knowledge*).

Os predicados do conhecimento de fundo podem ser definidos extensionalmente ou intensionalmente. Diz-se que o conhecimento de fundo é *extensional* quando é constituído por um conjunto de factos fechados sobre os predicados auxiliares. Se pelo contrário, os predicados auxiliares são definidos através de cláusulas não necessariamente fechadas, então o conhecimento de fundo é *intensional*.

O objectivo da PLI é geralmente enunciado da seguinte forma (De Raedt, Lavrac [24]*):*

Dados

- um conjunto de exemplos E (composto por exemplos positivos E^+, e exemplos negativos E^-),
- o conhecimento de fundo B,
- uma linguagem L de programas lógicos,
- e uma noção de explicação (uma semântica),

encontrar

- um programa $P \subseteq L$ que explique os exemplos E em relação a B.

Há várias noções de *explicação*. A mais frequentemente utilizada é a chamada semântica normal da PLI. Outra noção de explicação importante é dada pela semântica não monótona [22, 24, 36, 44]. Neste trabalho adoptaremos a semântica normal.

3.3.1 A semântica normal da PLI

Um programa P explica um conjunto de exemplos $E = E^+ \cup E^-$ relativamente ao programa B se

$P \cup B \models E^+$ (completude)

e

$P \cup B \not\models e^-$ para todo $e^- \in E^-$ (integridade)

Exemplo 3.6:

Exemplos positivos: *{descende(alipio,antonio)}*
Exemplos negativos: *{descende(antonio,alipio)}*
Conhecimento de fundo: *{filho(alipio,antonio), filho(antonio,adriana)}*
Hipótese: *{descende(X,Y)←filho(X,Y)}*

As condições de completude e de integridade podem ser verificadas usando a resolução-SLD (ou a resolução-SLDNF, no caso de as cláusulas não serem definidas). Relativamente à completude verificamos que os exemplos positivos são consequência lógica da hipótese P e do conhecimento de fundo B (neste caso temos apenas um exemplo positivo):

$P \cup B \vdash$ *descende(alipio,antonio)*

Quanto à integridade verificamos que o exemplo negativo não é consequência lógica de $P \cup B$.

$P \cup B \nvdash descende(antonio,alipio)$

♦

Quando um programa P explica um exemplo positivo $e^+ \in E^+$ diz-se que P cobre e^+. De forma idêntica se diz que um programa P cobre ou não um exemplo negativo e^-. Esta noção de cobertura é designada por *cobertura intensional*.

Definição 3.2: Um programa P cobre (intensionalmente) um facto e se $P \models e$. Um programa P cobre (intensionalmente) um facto e relativamente a um programa B se $P \cup B \models e$.

Algumas abordagens de PLI utilizam uma noção de cobertura diferente, computacionalmente emnos exigente, mas que dá, em geral, resultados diferentes. Trata-se da *cobertura extensional*.

Definição 3.3: Um programa P cobre (extensionalmente) um facto e relativamente a um modelo M se existe uma cláusula $C \in P$ ($C = H \leftarrow B$) e uma substituição θ tal que $C\theta$ é fechada, $H\theta = e$ e $B\theta \subseteq M$.

Exemplo 3.7: dado o programa P

descende(X,Z)←filho(X,Y),descende(Y,Z).
descende(X,Y)←filho(X,Y).

e o conhecimento de fundo

B = {*filho(alipio,antonio),filho(antonio,adriana)*}

temos que P cobre intensionalmente o exemplo

descende(alipio,adriana)

relativamente a B. No entanto, P não cobre extensionalmente o exemplo. ♦

Nesta dissertação usaremos sempre a noção de cobertura intensional, a menos que o contrário seja especificado.

As condições de completude e integridade enunciadas acima têm em conta apenas exemplos positivos e negativos. Este cenário pode ser alargado de forma a incluir restrições de integridade como uma fonte de informação mais expressiva, e particularmente de informação negativa. Uma restrição de integridade $Antec \rightarrow Cons$ é satisfeita por $P \cup B$ se é verdadeira no modelo mínimo do programa $P \cup B$. Isto pode ser verificado transformando a restrição de integridade numa pergunta.

$$P \cup B \nvdash Ant, not\ Cons$$

Um conjunto de restrições de integridade é satisfeito se cada restrição do conjunto é satisfeita. No Capítulo 7, formalizaremos estas noções e apresentaremos um método eficiente para verificação de restrições.

Exemplo 3.8: Seja I a restrição de integridade

$$descendant(X,Y) \wedge descendant(Y,X) \rightarrow false.$$

Esta restrição diz que ninguém é um descendente de um dos seus descendentes. Seja P o programa

$$descendant(X,Z) \leftarrow son(X,Y).$$
$$descendant(X,Y) \leftarrow son(Y,X).$$

e B o conhecimento de fundo.

$$B = \{son(antonio,adriana)\}$$

Para verificar se o programa P é satisfeito pela restrição I pomos ao programa a pergunta

$$\leftarrow descendant(X,Y), descendant(Y,X).$$

A pergunta sucede com *X*= *antonio* e *Y*= *adriana*. Logo *P* não satisfaz *I*.♦

Como veremos no Capítulo 7, tanto os exemplos positivos como os exemplos negativos podem ser reescritos como restrições de integridade. As condições de completude e integridade na definição do problema da PLI podem ser substituídas pela condição de satisfação das restrições.

No nosso trabalho, as três condições (completude, integridade e satisfação das restrições) serão testadas separadamente durante a indução. A integridade é verificada para cada cláusula candidata. A completude é obtida pela estratégia de síntese. A satisfação das restrições é verificada com algum grau de incerteza.

3.3.2 Direcções da PLI

O objectivo da PLI que foi enunciado acima serve apenas como um ponto de partida para um sistema que produza programas lógicos a partir de exemplos. A seguir referimos outros aspectos a considerar no desenvolvimento de um sistema de PLI.

- *Interactividade*. Um sistema de PLI pode ser ou não interactivo. Um sistema *interactivo* solicita informação a um *oráculo* (regra geral, o utilizador) durante o processo de indução. Os sistemas MIS [109], CLINT [20] e SYNAPSE [37] são interactivos. O sistema SKILit apresentado nos Capítulos 3 e 5 é não interactivo.

- *Ruído*. Os dados fornecidos ao sistema podem conter incorrecções de vários tipos (e.g. um exemplo dado como positivo pode não o ser de facto). Nesse caso dizemos que os dados contêm *ruído* (*noise*). Um sistema capaz de tratar ruído tem de relaxar as condições de completude e de integridade [61,62]. A nossa abordagem não lida com ruído.

- *Invenção de predicados*. Os predicados auxiliares definidos como conhecimento de fundo podem não ser suficientes para encontrar uma hipótese satisfatória. Alguns

sistemas de PLI contornam esta limitação inventando novos predicados [57,114]. Aqui não consideramos a invenção de predicados.

- *Aprendizagem/síntese mono ou multi-predicado*. Quando um sistema aceita exemplos relativos a vários predicados, e induz definições de vários predicados em simultâneo diz-se que faz *aprendizagem/síntese multi-predicado* (*multi-predicate learning/synthesis*) [25,100,109]. Caso contrário trata-se de *aprendizagem mono-predicado* (*single predicate learning*). Aqui concentramo-nos principalmente na aprendizagem mono-predicado. Contudo, mostramos que a nossa metodologia também se aplica à síntese multi-predicado.

- *Incrementalidade*. O sistema incremental tem a capacidade de modificar uma teoria inicial face ao aparecimento de novos exemplos. Na mesma situação, o sistema não incremental esquece a teoria inicial e recomeça a indução de uma nova teoria a partir do zero. Este sub-tema da PLI é chamado revisão de teorias (*theory revision*) [100,124]. O processo de síntese do sistema SKILit pode partir de um programa inicial, eliminar cláusulas (Secção 5.4.1) e inserir novas cláusulas.

3.4 Métodos e conceitos

Agora que temos a tarefa da PLI especificada, vamos ver de que tipo de abordagens dispomos para produzir um programa *P* a partir de exemplos e outros fragmentos de informação.

3.4.1 A procura num espaço de hipóteses

Como em quase todos os problemas de inteligência artificial, encontrar o programa pretendido pode ser reduzido a um problema de procura. Nesse caso, o *espaço de procura* (*search space*) é o conjunto dos programas da linguagem *L* escolhida como linguagem de hipóteses, e este espaço encontra-se estruturado pela relação de generalização entre hipóteses.

Definição 3.4: (Muggleton, De Raedt [83]) : Uma hipótese A é *mais geral* do que uma hipótese B se e só se $A \models B$. A hipótese B diz-se *mais específica* do que A. ♦

Partindo de um conjunto inicial de hipóteses, a procura é feita aplicando sucessivamente operadores de generalização e/ou de especialização às hipóteses existentes até que um critério de paragem seja satisfeito. Um *operador de generalização* produz um conjunto de hipóteses $G_1,G_2,...,G_n$ a partir de uma hipótese A, sendo cada G_i mais geral do que A. Um *operador de especialização* produz um conjunto de hipóteses mais específicas do que a inicial.

A estruturação do espaço de procura segundo uma relação de generalização permite eliminar muitas hipóteses. Por exemplo, dada uma hipótese H, um exemplo positivo e e o conhecimento de fundo B, se $H \cup B \not\models e$, então nenhuma especialização S de H vai ser tal que $S \cup B \models e$. Este facto poupa-nos o esforço de considerar hipóteses mais específicas do que H para cobrir o exemplo e. De forma idêntica, quando uma hipótese H viola a condição de integridade $H \cup B \models e^-$ para um exemplo negativo e^-, todas as hipóteses mais gerais do que H vão também ser não integras.

A relação de generalização baseada na implicação lógica corresponde à noção mais natural de generalização. Contudo, a implicação coloca alguns problemas conceptuais tais como:

- Dadas duas cláusulas C_1 e C_2 não é um problema decidível determinar se $C_1 \models C_2$.
- Duas cláusulas C_1 e C_2 não têm necessariamente generalização menos geral única sob a implicação [47].

Por esse motivo foram surgindo outros modelos de generalização computacionalmente mais atraentes e que se aproximam da implicação. Plotkin sugeriu a *subsunção-θ* (*θ-subsumption*) [92]. Buntine propôs a *inclusão generalizada* (*generalized subsumption*) [16] como um melhoramento em relação ao trabalho de Plotkin. Mais recentemente, Idestam-Almquist surgiu com a *implicação-T* (*T-implication*) [47], tentando resolver

alguns problemas deixados ainda pelos modelos de generalização anteriores. A subsunção-θ é mais frequentemente adoptada pelos algoritmos de PLI. É também o modelo de generalização que nós vamos adoptar e ao qual daremos mais atenção.

3.4.2 A relação de subsunção-θ entre cláusulas

A relação de generalização entre cláusulas é um caso particular bastante importante da generalização entre programas. Muitos dos algoritmos de PLI decompõem a procura no espaço geral das hipóteses em várias procuras no espaço das cláusulas.

Definição 3.5: (Plotkin [92]) uma cláusula *C1* θ*-subsume* uma cláusula *C2* se e só se existe uma substituição θ tal que $C1\theta \subseteq C2$. ♦

A relação de subsunção-θ é estritamente mais fraca do que a relação de implicação lógica [16]. Se uma cláusula *A* θ-subsume uma cláusula *B* então $A \models B$. O contrário não é verdadeiro.

Exemplo 3.9: Consideremos as duas seguintes cláusulas.

C1: p(X)←p(f(X)).
C2: p(X)←p(f(f(X)))

Temos $C1 \models C2$ sem termos *C1* θ-subsume *C2*.♦

Definição 3.6: Uma cláusula *C1* é θ*-equivalente* a uma cláusula *C2* se e só se *C1* θ-subsume *C2* e *C2* θ-subsume *C1*.♦

Definição 3.7: Uma cláusula *C* é *reduzida* se não for θ-equivalente a nenhum subconjunto próprio de si mesma.♦

A subsunção-θ entre cláusulas induz um reticulado (*lattice*) no conjunto das cláusulas reduzidas. Quaisquer duas cláusulas têm uma *generalização menos geral* única (*least upper bound*), e uma *especialização mais geral* única (*greatest lower bound*).

Dentro de um determinado conjunto de cláusulas vamos poder falar de *cláusulas maximamente específicas* e *cláusulas maximamente gerais*. A cláusula *member(X,Y)* é maximamente geral de entre as cláusulas que definem o predicado *member/2*. A cláusula □ (ou *false←true*) é maximamente geral em qualquer conjunto de cláusulas. Esta cláusula corresponde ao conjunto vazio pelo que θ-subsume qualquer cláusula. Uma cláusula maximamente específica que cubra um exemplo *e*, relativamente a uma teoria *T*, é $e \leftarrow b_1, b_2, \ldots$ onde os b_i são consequências fechadas de *T*. Neste caso, devem ser feitas restrições para que o conjunto de b_i seja finito. A cláusula maximamente específica é normalmente simbolizada por ⊥.

Dado este modelo de generalização, precisamos agora de operadores que nos permitam navegar no conjunto das cláusulas da linguagem de hipóteses. Veremos o operador de refinamento, que é um operador de especialização mais relevante para o nosso trabalho e o operador de generalização menos geral, que descreveremos com pouco detalhe. Os operadores de especialização permitem-nos deslocar no reticulado de cláusulas no sentido do mais geral para o mais específico (*abordagem descendente* (*top-down*)). Os operadores de generalização fazem-nos ir no sentido contrário (*abordagem ascendente* (*bottom-up*)).

3.4.3 O operador de refinamento (abordagem descendente)

Shapiro introduziu a noção de operador de refinamento de uma cláusula sob o modelo da subsunção-θ. Aqui damos uma definição mais genérica segundo De Raedt e Lavrac [24].

Definição 3.8: Um *operador de refinamento* ρ associa a uma cláusula *C* um conjunto de cláusulas $\rho(C)$ chamadas *refinamentos* de *C*. Este é um conjunto de especializações de *C* segundo a subsunção-θ. ♦

Um operador de refinamento típico aplica dois tipos de transformações a uma cláusula para a especializar:

1. instanciação de uma variável;
2. junção de um literal à cláusula.

Exemplo 3.10: A cláusula *member(A,[B|X])* pode ser especializada, por exemplo, substituindo *B* por *A*. Neste caso obtém-se o refinamento *member(A, [A|X])*. Obtemos outro refinamento se acrescentarmos um literal tal como em *member(A,[B|X])←member(A,X).*♦

Podemos procurar a cláusula procurada aplicando repetidamente um operador de refinamento. Começamos com a cláusula mais geral e aplicamos o operador de refinamento sucessivamente aos refinamentos. O processo de procura termina quando encontramos uma ou mais cláusulas que satisfaçam um determinado critério de paragem. Esta é a chamada abordagem descendente (*top-down*) à construção de uma hipótese, porque vai da cláusula mais geral para as mais específicas.

A *procura descendente* de uma cláusula utilizando operadores de refinamento corresponde a uma procura num grafo de refinamentos. Um grafo de refinamentos é um grafo dirigido acíclico cujos nós são cláusulas, sendo a raiz a cláusula de topo. Os ramos do grafo correspondem a operações de especialização.

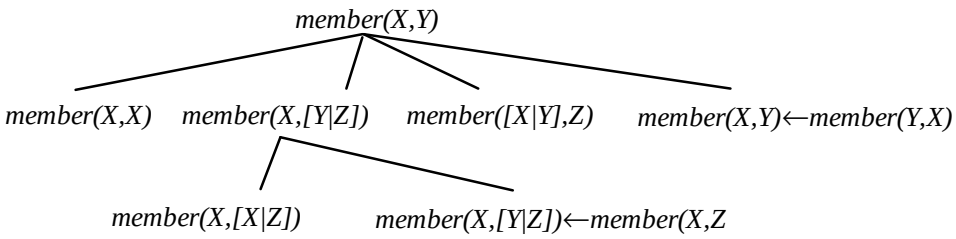

Figura 3.2: Parte de um grafo de refinamentos [109].

Vários factores afectam o tamanho e a forma da árvore de procura:

- *A cláusula de topo*. Se procuramos cláusulas para definir o predicado *p/n*, então a cláusula mais geral vai ser $p(X_1,...,X_n)$, em que cada X_i é uma variável [96,109]. Se não queremos determinar qual o predicado da cabeça da cláusula então podemos começar com a cláusula *true*←*false* [22].

- *O operador de refinamento*. Há três características principais de um operador de refinamento segundo Muggleton e De Raedt [83]. O operador é *globalmente completo* se é possível obter qualquer cláusula da linguagem aplicando o operador sucessivamente à cláusula de topo. O operador é *localmente completo* se, para qualquer cláusula *C*, $\rho(C)$ corresponde ao conjunto de todas as especializações mais gerais de *C*. Finalmente, o operador é *óptimo* se não gera cláusulas repetidas.

- *O critério de paragem*. O critério de paragem determina quando se pára a procura no grafo de refinamentos. Geralmente este critério é definido em função dos exemplos positivos e negativos. O sistema MIS de Shapiro [109] termina a construção de uma cláusula quando esta é suficientemente específica para não cobrir exemplos negativos. O critério de paragem pode também exigir que todas as variáveis da cláusula sejam ligadas [43]. Alguns sistemas utilizam heurísticas para definir o critério de paragem [96].

- *O método de procura*. A forma de percorrer o grafo de refinamentos também varia. Os métodos de procura mais usados (ver Secção 3.4.5) são a procura em largura (*breadth-first*) [109], a procura heurística (em particular os métodos heurísticos sem retrocesso (*greedy search*) [96,125]), e o método de aprofundamento iterativo (*iterative deepening*) [22].

3.4.4 O operador lgg (abordagem ascendente)

Sob a relação de subsunção-θ podemos definir a noção de generalizador menos geral de duas cláusulas.

Definição 3.9: A cláusula *G* é uma generalização de duas cláusulas *A* e *B* se e só se *G* θ-subsume *A* e *G* θ-subsume *B*. ♦

Definição 3.10: Uma cláusula *G* é a generalização menos geral (*least general generalization*) das cláusulas *A* e *B* se e só se, para toda a generalização *G*' de *A* e *B*, *G*' θ-subsume *G*. Escreve-*se lgg*(*A*,*B*)=*G*.♦

Exemplo 3.11: *lgg*(*p(a)*←*q(a), p(b)*←*q(b)*) = *p(X)*←*q(X)* ♦

Plotkin, no seu trabalho sobre a generalização segundo o modelo da subsunção-θ [92,93], mostra que, sob a subsunção-θ o *lgg* de duas cláusulas existe e é único (a menos da equivalência), e descreve um algoritmo para o construir. Plotkin define também um algoritmo para fazer a redução de cláusulas.

Mais tarde, Muggleton e Feng popularizaram o operador *lgg* aplicando-o no seu sistema GOLEM [82]. Neste sistema, os exemplos positivos são primeiro transformados em cláusulas iniciais maximamente específicas para o predicado dado. Cada uma destas cláusulas iniciais tem na cabeça o exemplo positivo e no corpo um conjunto finito de consequências lógicas do conhecimento de fundo. Aplicando o operador de generalização lgg de Plotkin, estas cláusulas são transformadas em cláusulas sucessivamente mais gerais. A mais importante contribuição deste trabalho de Muggleton e Feng foi o terem tornado eficientes as ideias de Plotkin. Isso foi conseguido principalmente devido às restrições feitas à linguagem de hipóteses. Outros sistemas usaram também o operador lgg [1,125].

3.4.5 Métodos de procura

Os métodos de procura empregues em PLI são basicamente os que se conhecem da inteligência artificial. O método de *procura em largura* [59] é um tipo de procura de força bruta. O espaço das cláusulas é percorrido exaustivamente. Trata-se de um método

de procura completo, i.e., se existir uma solução adissível no espaço de procura ela é encontrada.

Para efectuar a procura, o método de procura em largura mantém uma *fila ordenada* (*queue*) de refinamentos de cláusulas. Inicialmente a fila contém apenas a cláusula de topo. Em cada passo, o método retira a primeira cláusula da fila e expande-a resultando dessa expansão um conjunto de cláusulas. Todas os refinamentos de cláusulas resultantes dessa expansão são colocadas no fim da fila. A expansão de uma cláusula é feita aplicando-lhe o operador de refinamento.

Apesar de completo, o método tem a desvantagem de ser ineficiente (exige muita memória e tempo de computação). A sua utilização justifica-se quando o espaço de procura é suficientemente pequeno tendo em conta os recursos computacionais disponíveis, e quando outros métodos não obtêm sucesso.

O método de *procura heurística* é uma alternativa aos métodos de força bruta, e em particular à procura em largura. O métodos de procura heurística associam a cada cláusula um valor que reflecte o interesse dessa cláusula em função do objectivo da procura. Esse valor é calculado por uma função que se designa por heurística. A procura deixa de ser cega ao contrário do que acontecia com a procura em largura: as hipóteses mais prometedoras são consideradas em primeiro lugar.

O método de *subida mais rápida* (*hill-climbing*) escolhe entre todos os refinamentos de uma cláusula aquele que corresponde a um valor mais favorável da função heurística. Os restantes refinamentos são esquecidos, não havendo possibilidade de *retrocesso* (*backtracking*) na procura. Este é um método ganancioso (*greedy*).

A subida mais rápida é um método eficiente. Tem a desvantagem de não ser completo, podendo a procura enveredar por um caminho sem soluções. O sistema FOIL [96] de Quinlan utiliza este método de procura. Existem todavia outros métodos heurísticos mais

sofisticados que conseguem vencer as dificuldades postas pelo método de *hill-climbing* (cf. [59]).

3.4.6 A circunscrição de linguagem (*bias*)

Qualquer base para restringir o tamanho do espaço de procura ou para preferir uma solução a outras, que não seja a consistência com as observações é designado por *bias*, que nós aqui traduziremos por *circunscrição* [73,115]. Todos os algoritmos de aprendizagem, incluindo os de PLI, dependem de uma forma ou outra de algum tipo de circunscrição para levarem a cabo a procura de soluções com relativa eficiência. Os mecanismos que restringem a própria linguagem de conceitos são denominados por *circunscrição de linguagem* (*language bias*).

A linguagem de hipóteses pode ser restrita de muitas formas diferentes. Eis alguns exemplos de circunscrição de linguagem:

- *Vocabulário admissível*: As cláusulas induzidas só podem envolver predicados pertencentes a um conjunto pré-determinado. Este conjunto de predicados é designado por *vocabulário*. Em algumas abordagens, o conjunto de predicados admissíveis na construção da cláusula é determinado por outros predicados existentes, é o caso das *determinações* (*determinations*) de Russel[104].

- *Profundidade de termos*: A cada termo é associado um valor de nível (número natural). A restrição consiste em limitar a profundidade dos termos que ocorrem nas cláusulas Este conceito pretende capturar a noção de complexidade estrutural de um termo. A profundidade de variáveis e constantes é 0. A profundidade de um termo $f(t_1,...,t_n)$, é $1+max(profundidade(t_i))$, [21,82].

- *Cláusulas ligadas:* Uma cláusula é *ligada* (*linked clause*) se todas as suas variáveis forem ligadas. Uma variável é ligada se ocorrer na cabeça da cláusula ou se ocorrer num literal que contenha uma variável ligada (Helft [43]). Esta restrição tende a eliminar alguns literais potencialmente inúteis da cláusula.

- *Profundidade de uma variável:* A profundidade das variáveis que ocorrem nas cláusulas do programa pode também ser restringida. Seja $p(X_1,\ldots,X_n)\leftarrow L_1,L_2,\ldots,L_r,\ldots$ uma cláusula. Uma variável que ocorra na cabeça da cláusula, $X_1,\ldots,X_n$, tem uma profundidade de 0. Uma variável V cuja ocorrência mais à esquerda aconteça no literal L_r tem profundidade $1+d$, em que d é a profundidade máxima das variáveis em L_r que ocorrem em $p(X_1,\ldots,X_n)\leftarrow L_1,L_2,\ldots, L_{r-1}$ [62].

- *Recursividade:* As hipóteses podem-se limitar a conter cláusulas não recursivas. Esta é uma restrição muito forte, pouco adequada à síntese de programas Prolog.

- *Determinação*: Seja $A\leftarrow L_1,L_2,\ldots,L_r,\ldots$ uma cláusula. Uma variável que ocorra num literal L_r é *determinada* se tem uma substituição válida única determinada pelos valores das variáveis de L_r que ocorrem em $A\leftarrow L_1,L_2,\ldots, L_{r-1}$. O literal L_r é *determinado* se todas as suas variáveis que não aparecem em $A\leftarrow L_1,L_2,\ldots, L_{r-1}$ são determinados. Uma cláusula é *determinada* se todos os seus literais são determinados [62,82]. Impondo um limite j à aridade dos literais, e um limite i à profundidade de uma cláusula determinada, obtemos cláusulas *ij-determinadas*.

- *Tipos e modo entrada/saída:* As declarações dos tipos dos argumentos dos predicados, assim como as declarações de modo de entrada/saída também são úteis na PLI para limitar o espaço de procura. Do conjunto das cláusulas pertencentes à linguagem de hipóteses podemos eliminar as que não estão em conformidade com a declaração de tipos, ou com a declaração de modo [80,82,109].

A escolha da circunscrição da linguagem requer algum cuidado. Se a circunscrição torna a linguagem pouco expressiva, i.e. capaz de representar poucos conceitos, o sistema perde em generalidade, ganhando em eficiência. Se pelo contrário a circunscrição é fraca (pouco restritiva), o sistema torna-se mais geral a custo da eficiência.

3.4.7 Declarando a circunscrição de linguagem

É de todo o interesse que a circunscrição de linguagem seja um parâmetro do sistema e não uma sua propriedade estática. Ao utilizador interessa poder declarar, para cada problema, qual a circunscrição que escolhe. A este tipo de circunscrição definida pelo utilizador chama-se circunscrição declarativa.

A possibilidade de se definir a circunscrição de linguagem simbolicamente permite também que o sistema de PLI mude automaticamente a linguagem de hipóteses se necessário for. Assim, o sistema pode começar por procurar uma hipótese numa linguagem relativamente simples, e em caso de insucesso passar para linguagens de hipóteses sucessivamente mais complexas. Este é o conceito da *deslocação de linguagem* (*language shift* ou *shift of bias*) [20].

A forma mais simples de declarar a circunscrição de linguagem é através de parâmetros numéricos. Desta forma podemos limitar o número de cláusulas numa hipótese, o número de literais numa cláusula, o número de variáveis de um determinado tipo, a aridade dos predicados, a nível dos termos etc. Este é um tipo de circunscrição declarativa muito frequente nos sistemas de PLI [82].

Entretanto, surgiram outras formas mais sofisticadas de se descrever a linguagem de hipóteses. Wirth e O'Rorke [123] sugeriram os *grafos de dependências* que ilustram relações de dependência entre os literais. Os *modelos de regras* (*rule models*) de Kietz e Wrobel [56], assim como os *esquemas de cláusulas* (*clause schemata*) de Feng e Muggleton [34] são regras de ordem superior que representam o espaço de hipóteses. Um exemplo de uma regra de ordem superior é $P(X,Z) \leftarrow Q(X,Y),P(Y,Z)$. Os símbolos P e Q são variáveis que representam predicados. Substituindo essas variáveis por nomes de predicados obtemos cláusulas do espaço de hipóteses. Uma substituição possível daria

$$descende(X,Z) \leftarrow filho(X,Y),descende(Y,Z).$$

Outro formalismo utilizado para declarar a circunscrição de linguagem foi o das *gramáticas de cláusulas definidas* (*definite clause grammars*) ou DCG. Uma DCG é um programa Prolog escrito numa notação especial que facilita a definição de gramáticas [88]. William Cohen, no seu sistema Grendel [18] utilizou o formalismo das DCG para definir o corpo das cláusulas permitidas nas hipóteses. Klingspor [58] combina a abordagem das DCG com a das regras de ordem superior. As suas gramáticas já não descrevem directamente a linguagem das hipóteses mas antes definem um conjunto de regras de ordem superior que podem ser instanciadas, obtendo-se então as cláusulas da linguagem de hipóteses. A nossa própria metodologia de indução aqui descrita utiliza o formalismo das DCG para representar conhecimento de programação útil à síntese de programas. Um outro paradigma de descrição de circunscrição de linguagem foi apresentado por Bergadano [5].

Birgit Tausend juntou num só formalismo muitas destas diferentes formas de representação da circunscrição. A sua linguagem MILES-CTL [119] permite a descrição da forma geral das cláusulas através de estruturas chamadas *padrões de cláusulas* (*clause templates*). Dentro destas estruturas podem-se usar variáveis de predicados, definir tipos de predicados e de argumentos, restringir aridades, etc. Utilizando o MILES-CTL Tausend consegue comparar o efeito de diferentes circunscrições de linguagem em alguns casos teste [120].

Existe um outro tipo de conhecimento que é útil ao processo de síntese[14]. Quando o utilizador é capaz de descrever o funcionamento do algoritmo num exemplo particular, ainda que de forma vaga ou incompleta, essa informação pode ser explorada também para reduzir o esforço de procura. Na Secção 4.5.1 descrevemos como representar esse tipo de informação no que designamos por *esboços de algoritmo*.

3.5 Estado da Arte da PLI

3.5.1 Antecedentes

A PLI é hoje uma área de investigação bastante activa, com uma representação significativa dentro da área da aprendizagem simbólica [84]. A aprendizagem a partir de exemplos utilizou inicialmente linguagens de ordem zero (condições de pares atributo-valor, árvores de decisão) para representar as hipóteses ou formas muito restritas do cálculo de predicados [66].

Os trabalhos de Banerji [3], Plotkin [92,93], Michalski [67], Vere [122], Brazdil [13] e Sammut [107], entre outros, representam propostas para tornar a linguagem das hipóteses mais expressiva para que os algoritmos de aprendizagem tivessem uma aplicação mais vasta [106]. Mas se a linguagem de hipóteses aumentava a sua expressividade, os algoritmos de aprendizagem tinham de lidar com espaços de hipótese maiores e, consequentemente, a sua construção tornou-se um problemática. Faltava também uma teoria ou princípio unificador.

Uma tal teoria foi proposta por Shapiro [109], que utilizou para o seu sistema MIS a linguagem de cláusulas definidas para representar as hipóteses, e um pequeno conjunto de operadores para a geração dessas hipóteses. No final dos anos oitenta e princípios de noventa, a programação lógica foi adoptada como princípio subjacente a abordagens lógicas à aprendizagem simbólica. Muggleton cunhou o termo *Inductive Logic Programming* [77], e vários sistemas surgiram.

3.5.2 Alguns sistemas de PLI e afins

O sistema MIS de Shapiro é orientado para a depuração algorítmica (*algorithmic debugging*) de programas lógicos, assim como para a síntese de programas lógicos a partir de exemplos. Em cada sessão do MIS, parte dos exemplos positivos e negativos é fornecido no início, sendo os restantes solicitados pelo próprio sistema durante o

processo de indução. Para além dos exemplos, o sistema aceita declarações de tipo e de modo dos predicados envolvidos, assim como declarações de dependência entre predicados. O conhecimento de fundo é intensional.

Os sistemas GOLEM, de Muggleton e Feng [82] e FOIL de Quinlan [96,97,98], destacaram-se pela sua relativa eficiência e por algumas aplicações práticas. O motor de indução do sistema GOLEM é baseado no operador *lgg* de Plotkin [92], ao qual já nos referimos anteriormente (Secção 3.4.4). O sistema faz uma procura ascendente não completa: constrói cláusulas maximamente específicas a partir de exemplos escolhidos aleatoriamente, aplicando a estas o operador *lgg*, obtendo assim cláusulas cada vez mais gerais. As cláusulas que cobrirem mais exemplos positivos e menos exemplos negativos são escolhidas.

O sistema FOIL constrói cada cláusula segundo uma abordagem descendente. A cláusula de topo é uma cláusula maximamente geral (e.g. *member(X,Y)*). O sistema usa o método de procura da subida mais rápida (*hill-climbing*), e utiliza uma heurística calculada a partir de uma medida de informação baseada no número de exemplos positivos e negativos cobertos. As cláusulas vão sendo adicionadas segundo uma estratégia de cobertura tipo AQ [67,70].

Tanto o GOLEM como o FOIL aceitam exemplos positivos e negativos fechados fornecidos pelo utilizador. Para além disso são dadas declarações de modo de entrada/saída e de dependência para cada predicado. Ambos os sistemas são não interactivos e não incrementais. O conhecimento de fundo é extensional.

O sistema Progol de Muggleton [80] procura cada cláusula utilizando uma abordagem ascendente tal como GOLEM. Parte de uma cláusula maximamente específica e constrói como hipótese uma cláusula que seja uma sua generalização. A cláusula inicial tem como cabeça um exemplo positivo e no corpo um sub-conjunto do modelo do conhecimento de fundo. A procura da generalização é guiada por um algoritmo do tipo *A** [59]. O

Progol é um sistema relativamente eficiente, comparado com o GOLEM e o FOIL e permite a representação intensional do conhecimento de fundo.

CLINT [20] é um sistema interactivo e incremental que constrói uma teoria a partir de exemplos positivos e negativos fechados e de conhecimento de fundo. Dada uma linguagem clausal *L*, o sistema processa cada exemplo positivo *e* não coberto e constrói um conjunto *S* de cláusulas que cobrem *e* maximamente específicas em *L* (de acordo com a relação de subsunção-θ). Estas cláusulas não devem cobrir nenhum exemplo negativo. Depois, cada cláusula $C \in S$ é maximamente generalizada removendo-lhe literais do seu corpo. Antes de retirar um literal, o sistema pergunta ao utilizador qual o valor de verdade de um exemplo coberto pela cláusula resultante que não seja coberto por *C*. Se todos os exemplos são positivos, o passo de generalização é aceite, caso contrário é rejeitado. Os exemplos negativos obtidos no processo de generalização de uma cláusula são usados para detectar e remover cláusulas possivelmente incorrectas. Mais uma vez, o utilizador é inquirido neste processo.

O sistema SYNAPSE [38] de Pierre Flener pertence a uma classe diferente dos anteriores. É um sistema exclusivamente vocacionado para a programação automática. O sistema sintetiza programas a partir de exemplos fechados e de *propriedades* (cláusulas correctas mas incompletas), sendo um híbrido de diferentes abordagens à síntese de programas. Chamar-lhe um sistema de PLI é um pouco redutor. No SYNAPSE podemos encontrar síntese dedutiva, síntese baseada em conhecimento, e aprendizagem empírica [37].

A síntese é guiada por um *esquema* (*scheme*) que incorpora uma estratégia de programação particular (dividir-e-conquistar, geração-e-teste, produtor-consumidor, etc). O programa é construído transformando este esquema. O sistema permite também alguma interacção com o utilizador, de forma a evitar algumas procuras exponenciais. O SYNAPSE não usa programas auxiliares fornecidos pelo utilizador (conhecimento de fundo), mas pode inventar predicados caso seja necessário.

O sistema CRUSTACEAN [1] é um desenvolvimento do LOPSTER [60] e induz programas lógicos da forma

$p(Tb_1,\ldots,Tb_n)$.
$p(Th_1,\ldots,Th_n) \leftarrow p(Tr_1,\ldots Tr_n)$.

em que cada Tx_i é um termo. A cláusula base e a cláusula recursiva são construídas por decomposição estrutural dos exemplos positivos fechados dados. São também dados exemplos negativos fechados que são usados para eliminar programas candidatos sobregerais. Decompor um exemplo consiste em encontrar todos os *subtermos* possíveis dos seus argumentos. Por exemplo, suponhamos que temos o exemplo positivo *last_of(a,[c,a])*. O primeiro argumento só pode ser decomposto no subtermo *a*. O segundo argumento *[c,a]* pode ser decomposto em *[c,a]*, *c*, *[a]*, *a* e *[]*. Cada subtermo é obtido aplicando uma sequência de operadores de decomposição ao termos inicial. Esta sequência é designada por *termo gerador*. O número de vezes que o termos gerador é aplicado chama-se *profundidade*. O termo *[a]*, por exemplo, é obtido a partir de *[c,a]* pelo termo gerador *pair(2)*, i.e., a função que obtém o resto de uma lista. A profundidade é 1. Quando o subtermo é obtido aplicando decomposição nenhuma, o termo gerador é *none*. O CRUSTACEAN obtém todas as decomposições possíveis do exemplo combinando todas as decomposições possíveis dos seus argumentos. Uma decomposição possível do exemplo *last_of(a,[c,a])* é *last_of(a,[a])*. É obtida pela combinação dos termos geradores (*none*, *pair(2)*) à profundidade 1.

Suponhamos agora que temos um outro exemplo positivo *last_of(b,[x,y,b])*. Uma das decomposições deste exemplo é *last_of(b,[b])*. Os termos geradores correspondentes são *none* para o primeiro argumento, e *pair(2)* para o segundo. Contudo, *pair(2)* tem de ser aplicado duas vezes (profundidade 2).

CRUSTACEAN pode agora combinar as duas decomposições dos exemplos, uma vez que têm os mesmos termos geradores (*none,pair(2)*). O resultado da combinação é um

programa. A cláusula base é o *lgg* dos átomos que resultam da aplicação dos termos geradores aos exemplos *lgg*(*last_of(a,[a])*, *last_of(b,[b])*).

last_of(A,[A]).

Para obter a cabeça da cláusula recursiva aplicamos os termos geradores dos exemplos 0, 1, *n*-1 vezes em que *n* é a profundidade respectiva. Os átomos resultantes são *last_of(a,[c,a])* para o primeiro exemplo, e *last_of(b,[x,y,b])*, *last_of(b,[y,b])* para o segundo. A cabeça da cláusula é o *lgg* destes três átomos. O literal recursivo é obtido aplicando o termo gerador à cabeça da cláusula.

last_of(A,[B,C|D])←last_of(A,[C|D]).

è óbvio que o CRUSTACEAN não encontra a combinação certa de termos geradores directamente. Todos os diferentes termos geradores de todos os subtermos de todos os argumentos têm de ser encontrados. Depois disso o sistema constrói todas as combinações possíveis de termos geradores dos argumentos de cada exemplo. As combinações de exemplos diferentes são então emparelhadas de todas as formas possíveis.

Cada par é então ou descartado devido à incompatibilidade dos termos geradores ou resulta num programa. Os programas são filtrados. Programas redundantes, programas infinitamente recursivos e programas cobrindo exemplos negativos não são considerados. Os programas restantes são a resposta do CRUSTACEAN.

Devido à circunscrição de linguagem demasiado restritiva, o sistema não é capaz de explorar qualquer tipo de conhecimento de fundo.

Os sistemas acima referidos representam apenas uma selecção do estado da arte em PLI. Outros porém são também de interesse. É o caso de CHILLIN [125], CLAUDIEN [22], FORCE2 [12], FOCL [110], FORTE [100], ITOU [102], MOBAL [76], SMART [74],

TIM [49], WiM [95], etc. Todavia, não pretendemos aqui ser exaustivos na descrição dos sistemas existentes.

Sistema	***Recolha de exemplos***	***Estratégia***	***Conhecimento de fundo***	***Exemplos de aplicações***
MIS	interactivo, incremental	procura completa	intensional	síntese de programas
GOLEM		selecção heurística de hipóteses com geração aleatória de sementes. Utiliza *lgg*.	extensional	biologia, desenho técnico, modelos quantitativos
FOIL		estratégia de cobertura tipo AQ. Construção descendente de cláusulas com subida mais rápida.	extensional	síntese de programas
Progol		estratégia de cobertura tipo AQ.	intensional	bioquímica
CLINT	interactivo, incremental	construção ascendente de cláusulas	intensional	síntese de programas actualização de bases de conhecimento agentes autónomos
SYNAPSE	interactivo,	transformação de esquemas.	nenhum inventa predicados	síntese de programas
CRUSTACEAN		decomposição de termos	nenhum	síntese de programas

Tabela 3.1: Características principais de alguns sistemas de PLI importantes.

3.5.3 Aplicações

A maior parte das aplicações de PLI são no campo da extracção de conhecimento ou na síntese de programas. Quanto a aplicações na área da extracção de conhecimento o sistema GOLEM, por exemplo, foi aplicado a problemas de construção de modelos qualitativos para fenómenos físicos [11], construção de modelos temporais para operações de manutenção de um satélite [33], predição de estrutura de proteínas [77], e ao desenho técnico de componentes [29]. O Progol foi já aplicado para extracção de conhecimento na área da bioquímica [85]. Os resultados gerados pelo Progol foram inclusive divulgados em publicações científicas da especialidade [86].

Outros sistemas têm tido também aplicação prática. É o caso do MOBAL [75] , CLAUDIEN [22], FORTE [100] e FOCL [32].

3.5.4 Síntese indutiva de programas

Se no campo da extracção de conhecimento e da descoberta científica a PLI já se mostra uma ferramenta útil, o mesmo não se pode dizer em relação à síntese de programas a partir de exemplos (síntese indutiva). Neste campo, há ainda grandes problemas para resolver antes de termos uma verdadeira aplicação prática. O objectivo deste trabalho é dar um passo no sentido da utilização de ferramentas indutivas para auxiliar o desenvolvimento de pequenos programas.

Nesta secção mostramos informalmente resultados ilustrativos de sistemas representativos do que tem sido conseguido na área de síntese indutiva de programas. Os sistemas referidos são MIS, GOLEM, FOIL, CRUSTACEAN e SYNAPSE.

Vejamos em primeiro lugar um exemplo de uma sessão do MIS que é dado por Shapiro [109] e que mostra a síntese do predicado *isort/2* que ordena uma lista utilizando uma estratégia de inserção. A definição de *isort/2* é a seguinte:

isort([X|Y],Z)←isort(Y,V),insert(X,V,Z).
isort([],[]).

O predicado auxiliar *insert/3* é também sintetizado.

insert(X,[],[X]).
insert(X,[Y|Z],[X,Y|Z])←X≤Y.
insert(X,[Y|Z],[Y|V])← insert(X,Z,V),Y≤X.

A sessão é relatada em oito (!) páginas da tese de Shapiro, as quais são preenchidas principalmente por informação dada pelo sistema sobre o ponto da situação (e que o utilizador tem de ler para verificação), e pelas perguntas feitas ao utilizador e respectivas respostas. Um resumo da sessão indica que foram necessários 30 factos sobre *isort/2* e *insert/3* para a síntese. Foram consumidos 36 segundos de cpu (que não incluem os tempos de espera pelas respostas do utilizador).

No campo da síntese de programas a partir de exemplos, o sistema GOLEM obteve sucesso na indução de predicados como *member/2*, *reverse/2*, *multiply/2* e *qsort/2*, mas apenas quando os exemplos foram cuidadosamente escolhidos.

A cláusula recursiva da definição do predicado *qsort/2* (*quick sort*) é um teste clássico da síntese a partir de exemplos:

```
qsort([],[]).
qsort([A|B],[C|D])←
        partition(A,B,E,F),
        qsort(F,G),
        qsort(E,H),
        append(H,[A|G],[C|D]).
```

Trata-se de uma cláusula com dois literais recursivos, o que a torna problemática para algumas estratégias de síntese. Para além disso, tem 4 literais no corpo (6 se não são usados functores), um número relativamente grande de variáveis (8), tendo algumas delas profundidade 3 (ver pág. 50). GOLEM gerou a definição de '*quick sort*' a partir de 15 exemplos bem escolhidos em cerca de um centésimo de segundo. O conhecimento de fundo continha 84 factos sobre *partition/4* e *append/3*. Como é óbvio, estes resultados não são garantidos se forem usados outros exemplos.

O sistema FOIL foi avaliado em [97] pelos seus autores. A tarefa de teste consistia em sintetizar uma série de predicados tirados do livro de Bratko "*Prolog for Artificial Intelligence*" [10]. Como exemplo, podemos referir que o FOIL gerou a seguinte definição para *reverse/2*:

```
reverse(A,B)←A=B, conc(A,C,D), sublist(A,C).
reverse(A,B)←
        dest(A,C,D),reverse(D,E),
        append(F,D,A),append(E,F,B).
```

Esta definição foi sintetizada a partir de 40 exemplos positivos e 1561 exemplos negativos (ver Apêndice A com definições de predicados auxiliares). Estes constituem o conjunto de todos os exemplos que envolvem listas de tamanho igual ou inferior a 3.

Apesar de necessitar de um grande número de exemplos para gerar um programa, o FOIL é robusto na presença de redundância no conhecimento de fundo.

O Progol sintetiza uma definição do '*quick sort*' em menos de 1 segundo a partir de 11 exemplos positivos e 12 negativos bem escolhidos. Em [80] podemos encontrar um resumo de resultados obtidos pelo Progol na síntese indutiva.

O sistema SYNAPSE consegue sintetizar programas da dificuldade do '*insertion sort*', embora dando uma definição diferente da obtida pelo sistema MIS. A partir de 10 exemplos positivos, as 3 propriedades

isort([X],[X]).
isort([X,Y],[X,Y])←X≤Y.
isort([X,Y],[Y,X])←Y>X.

e algum conhecimento específico de programação relativo a este problema, consegue gerar uma definição de *isort/2*, inventando uma definição para o predicado *insert/3* [37, pp.209]. Neste caso o SYNAPSE dispensa o utilizador de saber quais os predicados auxiliares necessários à síntese.

O sistema CRUSTACEAN consegue sintetizar programas recursivos com functores sem predicados auxiliares. Cada programa é constituído por uma cláusula base e uma cláusula recursiva. Eis um exemplo:

split([],[],[]).
split([A,B|C],[A|D],[B|E])←
split(C,D,E).

CRUSTACEAN consegue gerar este programa a partir de 2 exemplos positivos e 4 negativos, sem mais nenhum tipo de informação. A linguagem de hipóteses é todavia muito limitada. A estratégia usada pelo sistema é bastante robusta à escolha dos exemplos. Por outras palavras, os dois exemplos não têm necessáriamente que ser bem escolhidos.

3.5.5 Problemas e limitações

- *Conhecimento de fundo intensional*. O GOLEM e o FOIL aceitam apenas conhecimento de fundo extensional. Se por uma lado, a representação extensional proporciona eficiência, por outro lado torna difícil a construção e a manutenção de teorias de conhecimento de fundo de grandes dimensões [79]. Alguns sistemas (CRUSTACEAN, SYNAPSE) não permitem sequer o emprego de de conhecimento de fundo.

- *Síntese de programas recursivos a partir de conjuntos esparsos de exemplos*. Tanto o Progol, como o GOLEM e FOIL têm dificuldade em sintetizar programas lógicos recursivos a partir de conjuntos de exemplos positivos relativamente pequenos. Quinlan refere que a síntese de *member/2* é afectada ligeiramente se for dado menos um exemplo positivo [97]. Numa experiência, registou-se que quando 25% dos exemplos positivos são apagados aleatoriamente, o programa ainda é correcto, mas contém três cláusulas supérfluas.

- *Utilização de conhecimento genérico de programação*. Os actuais sistemas de PLI, com poucas excepções, fazem uma procura cega do programa a sintetizar. Poucos tiram partido de uma base de conhecimentos sobre programação. Uma excepção é o sistema SYNAPSE, que constrói as cláusulas segundo uma (sempre a mesma) estratégia de dividir-e-conquistar. Mesmo o SYNAPSE não permite a definição de novas estratégias sem alteração do código do próprio sistema. Na Secção 4.5.1.1 descrevemos as gramáticas de estrutura clausal: um formalismo para representar conhecimento genérico de programação dentro da nossa metodologia de síntese.

- *Utilização de conhecimento específico de programação*. Se o utilizador tem alguma ideia, ainda que incompleta, sobre a estratégia do programa particular que quer sintetizar, deve-lhe ser dada a oportunidade de transmitir essa informação ao sistema de síntese. Os esboços de algoritmo que apresentamos na Secção 4.5.1 permitem que esse tipo de informação seja transmitido ao sistema SKILit.

- *Sobre-generalização*. O problema do número excessivo de exemplos negativos que é necessário fornecer a alguns sistemas de PLI para que induzam o programa pretendido atraiu já alguma atenção dos investigadores. As estratégias que têm sido propostas são, a nosso ver, insatisfatórias. O utilizador precisa de uma forma compacta de representar a informação negativa. As restrições de integridade permitem essa representação compacta mas colocam grandes problemas de eficiência. No Capítulo 7 propomos um algoritmo eficiente que permite a utilização de restrições de integridade no contexto da PLI.

3.6 Sumário

A nível da síntese indutiva de programas lógicos a PLI é uma área de investigação promissora, mas será necessário mais algum trabalho para que esteja pronta a ser útil em aplicações práticas. Para tal é necessário resolver problemas como a síntese de programas recursivos a partir de conjuntos esparsos de exemplos positivos, a eficaz utilização e representação de conhecimento genérico de programação e conhecimento específico do domínio, assim como a possibilidade de utilização de restrições de integridade para representação de informação habitualmente transmitida ao sistema através de exemplos negativos.

Nos Capítulos 3 , 5 e 7 abordamos estes problemas e propomos uma metodologia de síntese de programas que tenta resolver algumas das actuais limitações das abordagens indutias à síntese de programas a partir de exemplos.

4. Uma Abordagem à Síntese Indutiva

Este capítulo apresenta uma abordagem à síntese de programas lógicos a partir de especificações incompletas. O sistema SKIL é apresentado. Descrevemos a informação que é dada ao sistema, e definimos a classe de programas sintetizáveis. Descrevemos o processo de síntese através dos seus algoritmos principais.

4.1 Introdução

Neste Capítulo descrevemos a metodologia subjacente ao sistema SKIL. Trata-se de um sistema de programação lógica por indução (PLI) orientado para a síntese de programas lógicos (simplesmente programas, doravante) a partir de exemplos do seu comportamento. Em termos de síntese de programas podemos ver o SKIL como um sistema de síntese a partir de *especificações incompletas*.

O ponto de partida é uma descrição incompleta de um predicado *p/k*. O objectivo é sintetizar um programa *P* que define *p/k*. Essa descrição designa-se por especificação e é feita através de *exemplos positivos* e *negativos* desse predicado, *restrições de*

integridade, *declarações de modos de entrada/saída* e *declarações de tipo*. A partir destes dados o sistema constrói um programa que generaliza os exemplos positivos, e que é consistente com exemplos negativos e restrições de integridade (ver Capítulo 7). O programa produzido pelo SKIL é constituído por cláusulas definidas (*definite*) sem functores.

Outro elemento importante do processo é o *conhecimento de fundo (CF)*. Este é um programa lógico que define predicados auxiliares que poderão ser utilizados na definição do predicado a sintetizar. Embora não seja directamente uma circunscrição à linguagem, o conhecimento de fundo actua também como uma restrição ao conjunto de cláusulas sintetizáveis, já que tem um papel determinante na escolha de literais devido à estratégia de construção de cláusulas que o sistema SKIL utiliza.

Para além da especificação e do conhecimento de fundo, o sistema SKIL tira partido de outras fontes de informação que condicionam o processo de síntese e que são aqui genericamente designadas por *conhecimento de programação*. É o caso dos *esboços de algoritmo* e da *gramática de estrutura clausal (GEC)*. As gramáticas de estrutura clausal constituem a circunscrição de linguagem (*language bias*) pois definem o conjunto de cláusulas sintetizáveis pelo sistema.

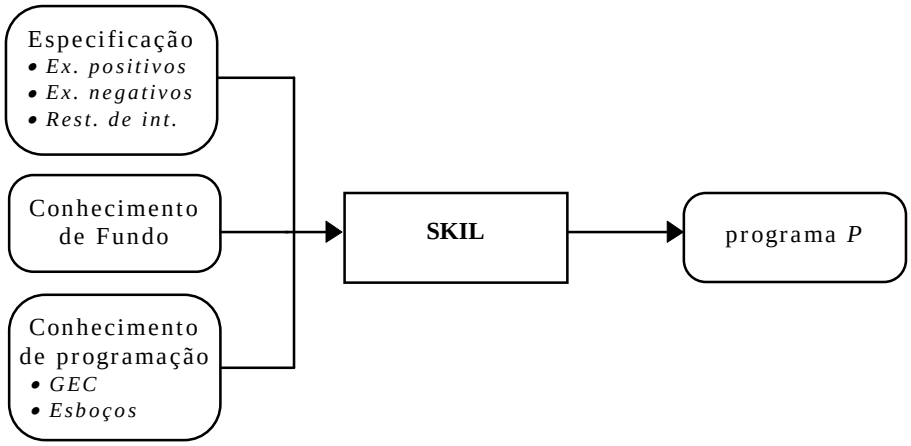

Figura 4.1: Enquadramento do sistema SKIL

4.2 Resumo

Começamos por descrever os dados do SKIL. O que é uma especificação (Secção 4.3), o que pode ser dado como conhecimento de fundo (Secção 4.4) e como o conhecimento de programação é representado (Secção 4.5). Dentro da Secção sobre o conhecimento de programação definimos esboços de algoritmo e explicamos o papel das gramáticas de estrutura clausal. Na Secção 4.6 caracterizamos os programas que o SKIL pode sintetizar.

O processo de síntese de um programa lógico é descrito em detalhe na Secção 4.7. Aí descrevemos os algoritmos para a construção de programas (SKIL) e construção de cláusulas. Apresentamos o operador de refinamento de esboços e a noção de sub-modelo relevante. Descrevemos também o interpretador de profundidade limitada utilizado na interpretação do conhecimento de fundo e dos programas construídos pelo sistema, e como as gramáticas de estrutura clausal são usadas pelo operador de refinamento. A Secção termina com uma descrição da verificação de tipos feita pelo SKIL.

Na Secção 4.9 mostramos uma sessão de síntese de programas com o SKIL e nas restantes três secções discutimos as limitações da metodologia, o trabalho relacionado e apresentamos um sumário breve dos resultados deste Capítulo.

4.3 Especificação

A *especificação* fornecida ao sistema SKIL é *incompleta*. O comportamento do programa que não é descrito directamente na especificação é inferido. A especificação descreve um único predicado *p/k* a ser definido como o programa *P*.

Dado um predicado *p/k* uma especificação é definida como (T,M,E^+,E^-,RI), em que:

- T é uma declaração do tipo do predicado *p/k*;
- M é uma declaração de modo de entrada/saída de *p/k*;

- E^+ é um conjunto de exemplos positivos de *p/k*;
- E^- é um conjunto de exemplos negativos de *p/k*;
- *RI* é um conjunto de restrições de integridade sobre *p/k*.

A Figura 4.2 abaixo mostra-nos o formato típico de uma especificação do SKIL. A notação tem uma sintaxe idêntica à da linguagem Prolog,: declarações de modo e de tipo, exemplos e restrições de integridade são representados como cláusulas.

```
mode( p(m1, ...,mk) ).
type( p(t1,...,tk) ).

% exemplos positivos
p(...).
...
p(...).

% exemplos negativos
–p(...).
...
–p(...).

% restrições de integridade
p(...),...,q(...)-->r(...),...,s(...).
...
```

Figura 4.2: Formato geral de uma especificação de um predicado *p/k*.

4.3.1 Objectivo da metodologia de síntese

Dado o conhecimento de fundo *BK* e uma especificação (T,M,E^+,E^-,*RI*) descrevendo o predicado *p/k*, o SKIL constrói um programa *P* definindo *p/k*. O programa tem, idealmente, as seguintes características:

- Todos os exemplos positivos são cobertos.

  $P \cup BK \vdash E^+$

- Nenhum exemplo negativo é coberto.

$P \cup BK \nvdash e^-$ para todo $e^- \in E^-$

- O programa é consistente com as *RI*. Esta condição é verificada com algum grau de incerteza, devido à estratégia de Monte Carlo utilizada (ver Capítulo 7).

  $P \cup BK \nvdash$ (*Antec, not Cons*) para toda $I \in IC$, *I* tem a forma *Antec*→*Cons.*

4.3.2 Exemplos, modos, tipos, restrições de integridade

Os *exemplos positivos* dados ao SKIL são átomos fechados, i.e., átomos que não contêm variáveis. Os *exemplos negativos* são factos fechados precedidos do sinal '–'. A *declaração de modo* de um predicado *p/k* atribui a cada um dos *k* argumentos uma direcção de entrada ou de saída. Os argumentos de entrada são assinalados com o sinal '+', e os de saída com o sinal '–'.

Um exemplo positivo do predicado *reverse/2*, que faz a inversão de uma lista, poderá ser *reverse([2,1],[1,2])*. Este exemplo positivo determina que o programa a sintetizar deverá responder que a inversa da lista *[2,1]* é a lista *[1,2]*. Um exemplo negativo da mesma relação será *–reverse([0,3],[0,3])*.

A declaração de modo de entrada saída é

mode(reverse(+,–)).

Esta declaração de entrada/saída significa que uma pergunta ao programa que contém a definição do predicado *reverse/2* terá o primeiro argumento obrigatoriamente instanciado, tal como em *?-reverse([2,4,3],X)*.

A *declaração de tipo* associa a cada argumento um identificador que representa o tipo atribuído. Os tipos aqui cosiderados incluem listas (identificador *list)*, inteiros (identificador *int*), etc. (Apêndice B). No caso do predicado *reverse/2*, a declaração de tipo é *type(reverse(list,list))*. A declaração do tipo de um predicado facilita o processo de indução mas é facultativa. Na Figura 4.3 vemos o exemplo de uma especificação.

```
mode( reverse(+,–) ).
type( reverse(list,list) ).

% exemplos positivos
reverse([],[]).
reverse([1],[1]).
reverse([1,2],[2,1]).

% exemplos negativos
–reverse([],[1]).
–reverse([1,2],[1,2]).
–reverse([1,2,3],[2,1,3]).

%restrições de integridade
reverse([A,B],[C,D])-->A=D.
reverse([A,B],[C,D])-->B=C.
```

Figura 4.3: Exemplo de uma especificação para o predicado *reverse/2*.

As *restrições de integridade* são cláusulas não fechadas que podem representar um conjunto de exemplos negativos. Cada exemplo negativo pode ser transformado numa restrição de integridade. Por uma questão de clareza distinguiremos a descrição do tratamento de exemplos negativos e de restrições de integridade. Este último ponto será descrito no Capítulo 7.

4.4 Conhecimento de fundo

O conhecimento de fundo fornecido ao sistema SKIL é um programa Prolog que define predicados auxiliares que podem ser invocados pelo programa a sintetizar. As cláusulas do conhecimento de fundo podem conter functores e negação. A Figura 4.4 mostra o tipo de programas auxiliares que se podem encontrar no conhecimento de fundo.

```
addlast([],X,[X]).
addlast([A|B],X,[A|C])←
    addlast(B,X,C).

null([]).

dest([A|B],A,B).

const([A|B],A,B).
```

Figura 4.4: Um exemplo de conhecimento de fundo

De entre os predicados definidos como conhecimento de fundo, o utilizador pode indicar quais são os *predicados admissíveis* para uma determinada tarefa de síntese. Isto é feito através de uma declaração que é dada ao sistema juntamente com a especificação. Vejamos um exemplo.

```
adm_predicates( reverse/2, [const/3,dest/3,null/1,addlast/3,reverse/2] ).
```

A declaração acima indica que o sistema pode induzir uma definição para o predicado *reverse/2* com cláusulas envolvendo os predicados *const/3*, *dest/3*, *null/1*, *addlast/3* e *reverse/2*, e apenas estes predicados. A declaração de predicados admissíveis delimita o *vocabulário* da tarefa de síntese.

4.5 Conhecimento de programação

Para além da informação contida na especificação e do conhecimento de fundo, o SKIL emprega outras fontes de conhecimento auxiliar tais como *esboços*, que contêm conhecimento específico a cada tarefa de síntese e a *gramática de estrutura clausal*, que contém conhecimento genérico de programação. A esse corpo de informação chamamos *conhecimento de programação*.

Estes elementos não fazem, obviamente, parte da especificação, sendo antes utilizados para levar a cabo a tarefa de síntese. Isto é, enquanto exemplos e restrições de integridade indicam o *que* (*what*) se pretende sintetizar, esboços e gramáticas indicam *como* (*how*) a síntese deve ou pode ser feita. Esta distinção entre o 'que' e o 'como' do processo de síntese foi já assinalada no Capítulo 2.

4.5.1 Esboços de algoritmo

Por vezes, o utilizador de um sistema de síntese pode conhecer alguns dos predicados envolvidos e de que forma é que esses predicados contribuem para a derivação de um determinado exemplo positivo. Existindo esse conhecimento, é de interesse que possa ser aproveitado pelo sistema de síntese. Esse conhecimento é comunicado ao sistema através de um *esboço de algoritmo*. O sistema SKIL é capaz de explorar esboços de algoritmo fornecidos pelo utilizador [14]. Devemos no entanto chamar a atenção para o facto de que os esboços de algoritmo não são dados obrigatórios para o SKIL.

4.5.1.1 O que é um esboço de algoritmo

Informalmente um esboço de algoritmo representa a explicação de um exemplo positivo feita em termos de uma ligação relacional entre os argumentos de entrada e os argumentos de saída do exemplo. Formalmente, um esboço de algoritmo relativo a um programa *P* é uma cláusula fechada cuja cabeça é um exemplo positivo de um predicado *p/k* definido em *P*, e o corpo contém literais que explicam os argumentos de saída do exemplo a partir dos argumentos de entrada. Quando alguma parte da explicação não é conhecida, os argumentos são ligados por literais especiais designados por *literais de esboço*. Os restantes literais, que envolvem predicados admissíveis são designados por *literais operacionais*. Os predicados usados nos literais de esboço (*predicados de esboço*) começam pelo caracter $. Para todos os predicado envolvidos existe uma declaração de entrada/saída. Podemos assim falar dos argumentos de entrada e de saída dos literais envolvidos.

Definição 4.1: Dado um conjunto de literais α, um termo t é *direccionalmente ligado* em α relativamente a um conjunto de termos T se e só se $t \in T$ ou t é um argumento de saída de algum literal $L \in \alpha$ e todos os argumentos de entrada de L são direccionalmente ligados em α relativamente a T.♦

Note-se que iremos aqui utilizar uma notação semelhante à da utilizada em Prolog para representarmos conjuntos de literais. Assim, a sequência de literais $L_1,L_{2,\ldots},L_n$ representa o conjunto $\{L_1,L_{2,\ldots},L_n\}$.

Exemplo 4.1: O termo *e* é direccionalmente ligado com respeito a {*a*,*b*} no conjunto de literais *p(+a,–c),q(+b,–d),r(+c,+d,–e)*. A ligação está graficamente representada na Figura 4.5 .

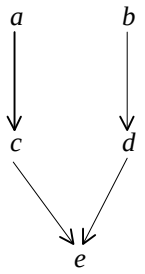

Figura 4.5: Ligação dos termos {*a*,*b*} ao termo *e*.

No mesmo conjunto de literais podemos encontrar outras ligações. Por exemplo, o termo *b* está direccionalmente ligado em relação a {*b*}.♦

Definição 4.2: Um conjunto de literais α diz-se uma *ligação relacional* entre um conjunto de termos T_1 e um conjunto de termos T_2 se e só se todo o termo $t \in T_2$ é direccionalmente ligado em α relativamente a T_1.♦

Exemplo 4.2: Uma ligação relacional liga os termos T_1 aos termos T_2 e não contém literais com termos que não estejam ligados em relação a T_1. O conjunto α = *p(+a,–b),q(+c,–d)* não é uma ligação relacional de {*a*} para {*d*} porque *c* não está

direccionalmente ligado em α. Todavia, é uma ligação relacional de {*a*,*c*} para qualquer subconjunto de {*a*,*b*,*c*,*d*}.

O conjunto de literais *p(+a,–c),q(+b,–d),r(+c,+d,–e)* é uma ligação relacional de {*a*,*b*} para qualquer subconjunto de {*a*,*b*,*c*,*d*,*e*}.♦

Definição 4.3: Um termo *t* é *direccionalmente ligado* num esboço $H \leftarrow \beta$, em que β é um conjunto de literais, se e só se existe uma ligação relacional $\alpha \subseteq \beta$ a partir dos argumentos de entrada de *H* para *t*. ♦

Definição 4.4: Uma cláusula $H \leftarrow \alpha$ é uma *cláusula direccionalmente ligada* se todos os argumentos de saída de *H* são termos direccionalmente ligados em α relativamente ao conjunto de argumentos de entrada de *H*.♦

Definição 4.5: Um *esboço de algoritmo* é uma cláusula direccionalmente ligada e fechada da forma $H \leftarrow L_1,L_2,..,L_n$ com $n \geq 1$. *H* é um exemplo positivo do conceito que se pretende definir. Os literais $L_1,L_2,..,L_n$ são literais operacionais ou *literais de esboço*.♦

Os literais de esboço têm a função de ligar argumentos que de outra forma estariam desligados e distinguem-se sintaticamente pelos símbolos de predicado da forma *$Px*, em que *x* é um inteiro positivo.

Exemplo 4.3: Seja *rv([3,2,1],[1,2,3])* um exemplo positivo do predicado *rv(+,–)*. A seguinte cláusula é um esboço.

r v(+[3,2,1],–[1,2,3])←
$P1(+[3,2,1],–3,–[2,1]),rv(+[2,1],–[1,2]),$P2(+3,+[1,2],-[1,2,3]).

Este esboço envolve dois predicados de esboço *$P1* and *$P2*, e um predicado operacional *rv/2*. Pode ser visto como uma explicação de como inverter a lista *[3,2,1]*; "A partir da lista *[3,2,1]* obtêm-se os termos *3* e *[2,1]* (não é descrito como), inverte-se

[2,1], e combina-se o resultado com *3* para obter *[1,2,3]* (mais uma vez, de alguma maneira)".

No esboço mostrado acima, a lista *[3,2,1]* está ligada a *[1,2,3]*. A Figura 4.6 mostra uma representação gráfica de um esboço.

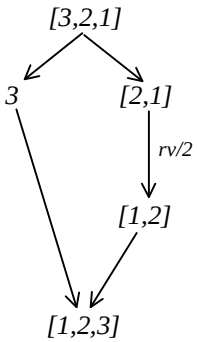

Figura 4.6: Representação gráfica de um esboço.

♦

4.5.1.2 Exemplos positivos são esboços mínimos

Qualquer exemplo positivo pode ser visto como um esboço de algoritmo que não tem qualquer informação sobre como chegar dos argumentos de entrada aos argumentos de saída do exemplo. A ligação entre os argumentos de entrada e os argumentos de saída é feita por um único literal de esboço que serve apenas para tornar explícitas as ligações que faltam.

Definição 4.6: Quando o corpo do esboço contém apenas um literal de esboço, trata-se de um *esboço mínimo*. O esboço mínimo associado a um exemplo positivo $p(t_1,\ldots,t_k)$ é da forma

$$p(t_1,\ldots,t_k) \leftarrow \$P(t_1,\ldots t_k).$$

em que *$P($t_1$,…$t_k$)* é um literal de esboço com os mesmos argumentos do exemplo positivo e *$P/k* é um predicado com o mesmo modo de entrada saída de *p/k*.♦

4.5.1.3 Os esboços como refinamentos

Um esboço de algoritmo pode também ser visto como uma representação interna de uma cláusula que está a ser construída segundo uma estratégia de ligação dos argumentos do exemplo. A procura de um esboço operacional adequado é feita num espaço de esboços de algoritmo partindo de um esboço inicial usando um operador de refinamento próprio. Nessa perspectiva, cada cláusula é obtida a partir de um esboço operacional que explica um determinado exemplo positivo.

Definição 4.7: Um esboço de algoritmo é um *esboço operacional* se não contém literais de esboço.♦

Definição 4.8: Ao processo de substituição dos literais de esboço (de um esboço de algoritmo) por literais operacionais de forma a obter-se um esboço operacional chama-se *consolidação de um esboço*.♦

A metodologia de síntese de programas descrita neste Capítulo segue uma estratégia de consolidação de esboços. Quando um esboço está completamente consolidado, cada termo e cada literal do esboço estão operacionalmente ligados.

Definição 4.9: Um termo *t* diz-se operacionalmente ligado num esboço $H \leftarrow \beta$ se e só se existir uma ligação relacional $\alpha \subseteq \beta$ a partir dos argumentos de entrada de *H* para *t* e α contém apenas literais operacionais.♦

Definição 4.10: Um literal *L* está direccionalmente ligado num esboço *Esb* se e só se todos os argumentos de entrada de *L* estiverem operacionalmente ligados em *Esb*.♦

Embora um esboço seja representado como uma cláusula, e por conseguinte possa ser visto como um conjunto de literais, nós definiremos os algoritmos de consolidação de

esboços assumindo uma dada ordenação dos literais no corpo do esboço, por uma questão de clareza de exposição.

Definição 4.11: Um esboço $H \leftarrow \alpha$ diz-se um *esboço sintacticamente ordenado* se e só se as duas seguintes condições se verificarem.

1) Todo literal operacionalmente ligado fica à esquerda de qualquer literal noutras condições.

2) Todo literal operacional que seja operacionalmente ligado fica à esquerda de qualquer literal de esboço. ♦

Exemplo 4.4: O esboço

r v(+[3,2,1],–[1,2,3])←
$P1(+[3,2,1],–3,–[2,1]),rv(+[2,1],–[1,2]),$P2(+3,+[1,2],-[1,2,3]).

está sintaticamente ordenado. O esboço

r v(+[3,2,1],–[1,2,3])←
rv(+[2,1],–[1,2]),$P1(+[3,2,1],–3,–[2,1]), $P2(+3,+[1,2],-[1,2,3]).

não está ordenado. O literal *$P1(+[3,2,1],–3,–[2,1])* é operacionalmente ligado, embora não seja um literal operational. Por isso deveria estar à esquerda do literal *rv(+[2,1],–[1,2])* que não está operacionalmente ligado.

O esboço

r v(+[3,2,1],–[1,2,3])←
$P2(+3,+[1,2],-[1,2,3]),$P1(+[3,2,1],–3,–[2,1]),rv(+[2,1],–[1,2]).

também não está ordenado. O literal *$P2(+3,+[1,2],-[1,2,3])* não está operacionalmente ligado (nenhum dos termos de entrada está direccionalmente ligado) e aparece à esquerda de *$P1(+[3,2,1],–3,–[2,1])*. ♦

4.5.2 Gramáticas de estrutura clausal

Outra fonte de informação importante para a nossa metodologia de síntese de programas, e que não faz parte da especificação, é a *gramática de estrutura clausal* (*clause structure grammar*). A gramática de estrutura clausal contém algum conhecimento de programação, e por esse motivo não é específica à tarefa de sintetizar nenhum predicado em particular, mas genérica para uma certa classe de programas. Uma determinada gramática de estrutura clausal pode servir para sintetizar programas do tipo dividir-e-conquistar enquanto outra pode descrever programas, por exemplo, de geração e teste. Na nossa metodologia, as gramáticas de estrutura clausal são descritas utilizando uma gramática de cláusulas definidas (*definite clause grammar*, DCG) [88]. As gramáticas de estrutura clausal aqui utilizadas são descritas na Secção 4.7.5.

Tanto esboços de algoritmo como gramáticas de estrutura clausal servem para facilitar a tarefa de síntese. Obviamente, o utilizador terá de despender algum esforço a fornecer ao sistema esta informação.Todavia, as gramáticas clausais são potencialmente reutilizáveis e não particulares de um dado programa.

4.6 Classe de programas sintetizados

Os programas sintetizados pelo nosso sistema são constituídos por cláusulas com um e um só literal na cabeça e sem literais negados no corpo. Noutras palavras, são programas constituídos por cláusulas definidas (*definite clauses*).

Outra característica dos programas lógicos produzidos é não terem functores nem constantes. Os argumentos dos literais presentes nas cláusulas são sempre variáveis não instanciadas. A necessidade de functores é eliminada pela utilização de predicados apropriados. Ao processo de transformar um programa com functores num programa equivalente sem functores chama-se *achatamento* (*flattening*) [102]. Por exemplo, a sequência de literais *p([A|B]),q(B)* que contém um termo estruturado (a lista *[A|B]*) pode ser substituída por *p(X),decomp(X,Y,Z),q(Z)*. O predicado *decomp/3* decompõe

uma lista *X* em cabeça *Y* e cauda *Z*. Como iremos ver em alguns exemplos, vários predicados auxiliares semelhantes a *decomp/3* irão ser utilizados para auxiliar a tarefa de síntese. As definições destes predicados auxiliares são acrescentadas ao conhecimento de fundo fornecido ao sistema.

As constantes são tratadas de forma semelhante. Predicados como *null/1* e *zero/1* servem para introduzir nas clásusulas as constantes *[]* (lista vazia) e *0* (número zero), respectivamente.

A escolha de uma linguagem livre de functores não é fundamental no sentido em que o sistema poderia ser adaptado de forma a trabalhar com functores. No entanto, ao substituirmos functores e constantes por predicados, simplificamos a operação de refinamento de uma cláusula e consequentemente os algoritmos de síntese.

No entanto, as cláusulas achatadas (*flattened*) produzidas pelo SKILit podem ser automaticamente desachatadas pelo sistema para apresentação. Alguns dos programas aqui mostrados são apresentados na sua forma descachatada.

4.7 Síntese de um programa lógico

A metodologia de síntese do sistema SKIL parte de um conjunto de exemplos positivos E^+, exemplos negativos E^-, restrições de integridade *RI* sobre um predicado *p/k*, um programa inicial P_0 (possivelmente vazio) e um programa auxiliar de conhecimento de fundo *BK*. O resultado é um programa lógico *P* que define um predicado *p/k*. o sistema usa uma estratégia de cobertura, a qual funciona da seguinte forma.

Para cada exemplo positivo $e \in E^+$ ainda não coberto, tenta-se construir uma nova cláusula que, acrescentada a *P*, ajude a cobrir *e* (ver Algoritmo 1). A construção da cláusula é feita pelo procedimento *ConstróiCláusula* (Algoritmo 2). No caso deste não conseguir construir uma nova cláusula retorna o conjunto vazio (∅). O programa *P* permanece então inalterado, e o Algoritmo 1 passa a tratar o exemplo positivo seguinte.

O programa P pode ser inicialmente vazio ou conter algumas cláusulas que definam o predicado *p/k*. Estas cláusulas iniciais podem ser fornecidas pelo utilizador, ou por outros procediementos que invocam o SKIL, como é o caso do SKILit apresentado no Capítulo 5. O programa inicial é denominado P_0. O Algoritmo 1 mostra os detalhes do procedimento de cobertura.

Procedimento SKIL
entrada: E^+,E^-,*RI*, P_0, *BK*
saída: P
$P := P_0$
para cada $e \in E^+$ tal que $P \cup BK \cup E^+\text{-}\{e\} \nvdash e$
    NovaCláusula := *ConstróiCláusula*(e,$E^+\text{-}\{e\}$,E^-,*RI*,P,*BK*)
    $P := P \cup$ *NovaCláusula*
seguinte
retorna P

Algoritmo 1: Construção de um programa pelo SKIL

4.7.1 O construtor de cláusulas

Cada cláusula é construída para cobrir um determinado exemplo positivo do predicado a sintetizar. Esse exemplo funciona como uma *semente* no processo de construção, pois os seus argumentos guiam a selecção dos literais do corpo da cláusula.

A estratégia de construção da cláusula baseia-se na procura de uma *ligação relacional* (Definição 4.2) entre os argumentos de entrada do exemplo e os argumentos de saída. Esta ligação é feita através dos predicados auxiliares admissíveis. No caso dos programas recursivos, o próprio predicado a sintetizar é um predicado admissível. O predicado a ser sintetizado é definido parcialmente pelos exemplos positivos E^+ e, possivelmente, por cláusulas existentes (por exemplo em P_0). Quando o procedimento *ConstróiCláusula* é invocado pelo Algoritmo 1 o exemplo e a ser coberto e os exemplos postivos restantes em $E^+\text{-}\{e\}$ são passados como argumentos separados. Os exemplos em $E^+\text{-}\{e\}$ permitem ao SKIL a construção de cláusulas recursivas.

A cláusula procurada é extraída a partir da ligação relacional, i.e., da sequência de literais que ligam os argumentos de entrada do exemplo positivo com os seus argumentos de saída. Este último passo consiste basicamente na transformação de constantes em variáveis.

Exemplo 4.5: Vamos supor que temos a seguinte situação:

Exemplo positivo (com declaração de modo):

mode(grandfather(+,-)).
grandfather(tom,bob).

Conhecimento de fundo (com declarações de modo):

mode(father(+,-)).	*mode(mother(+,-)).*
father(tom,anne).	*mother(anne,bob).*
father(tom,jack).	*mother(anne,chris).*

Para construir uma cláusula que cubra o exemplo positivo vamos procurar ligar o argumento de entrada (*tom*) com o de saída (*bob*). A sequência de literais *father(tom,anne),mother(anne,bob)* é uma ligação relacional entre os termos *tom* e *bob* e pode ser vista como uma espécie de explicação do exemplo positivo.

Agora, com o exemplo positivo e a sequência de literais encontrada construímos a clásula candidata instanciada (esboço)

grandfather(tom,bob)←father(tom,anne),mother(anne,bob)

Do qual extraímos a cláusula

grandfather(X,Z)←father(X,Y),mother(Y,Z)

substituindo constantes por variáveis. ♦

Grande parte do processo de construção de uma cláusula que, juntamente com o conhecimento de fundo, cubra um exemplo positivo dado, consiste na consolidação de um esboço associado a esse exemplo. O esboço associado a um exemplo *e* é fornecido

pelo utilizador ou é o esboço mínimo associado ao exemplo da forma *e←$P(...)*, que é automaticamente gerado pelo sistema (ver Secção 4.5.1).

A consolidação de um esboço consiste numa *procura em largura* de um *esboço operacional* no espaço dos esboços. Esse espaço é percorrido usando um *operador de refinamento* ρ, que retorna o conjunto de refinamentos de cada esboço. Um refinamento de um esboço também é um esboço. O ponto de partida da procura é o esboço associado ao exemplo positivo.

Procedimento ConstróiCláusula
entrada: e, E^+, E^-, RI, P, BK
saída: *Cl* (a nova cláusula)
EsbIn := *EsboçoAssociado(e)*
Q := [*EsbIn*]
repete
  se *Q* = ∅ **então retorna** ∅
  Esb := primeiro esboço em *Q*
  se *Esb* é um esboço operacional **então**
    Cl := *Variabiliza(Esb)*
    se $\{Cl\} \cup P \cup BK \cup E^+$ cobre e
      e $\{Cl\} \cup P \cup BK \cup E^+$ não cobre nenhum $e \in E^-$
      e $\{Cl\} \cup P \cup BK \cup E^+$ não viola *RI*
        então retorna {*Cl*}
    fim de se
  fim de se
  Q := *Q* - *Esb*
  EsbNovos := $\rho(Esb, P, BK, E^+)$ (Algoritmo 3)
  Q := *Q* juntando-lhe *EsbNovos* ao final
sempre

Algoritmo 2: Geração de uma cláusula por refinamento de um esboço

A procura pára quando se encontra um esboço operacional e dele é possível extrair uma cláusula que satisfaz o critério de paragem. A cláusula é obtida substituindo os termos do esboço por variáveis (processo a que chamamos *variabilização*). A variabilização é feita pela função *Variabiliza* descrita na Secção 4.7.1.1.

O procedimentos *ConstróiCláusula* (Algoritmo 2) inicializa uma fila *Q* de refinamentos com o esboço associado ao exemplo dado como argumento de entrada. Em cada iteração do ciclo 'repete', é retirado o primeiro esboço em *Q* sendo construído o conjunto dos seus refinamentos. Os esboços pertencentes ao conjunto de refinamentos são colocados no final da fila *Q*.

Como podemos observar, o ciclo repete pode terminar por várias razões. Idealmente pára porque foi encontrado um esboço operacional que deu origem a uma cláusula que cobre o exemplo positivo e que não viola as restrições de integridade nem cobre nenhum exemplo negativo. Para não violar os exemplos negativos $\{Cl\}\cup P\cup BK\cup E^+$ não pode cobrir intensionalmente nenhum deles. Quanto às restrições de integridade, elas são verificadas pelo módulo MONIC que será descrito no Capítulo 7. O Algoritmo 2 pára também quando a fila *Q* de refinamentos se esgota. Nesta situação é devolvido o conjunto vazio.

Na implementação actual do sistema SKIL existe ainda um controlo do número de refinamentos construídos durante a geração de uma cláusula. Para tal é imposto um limite ao número de refinamentos construídos, a que chamamos *limite de esforço*. Este limite tem um valor por defeito (300 refinamentos), mas pode ser definido pelo utilizador para cada tarefa de síntese através de uma declaração específica. Quando o limite de esforço é atingido a construção da cláusula termina retornando o conjunto vazio.

4.7.1.1 Variabilização

A *variabilização* de um esboço consiste na substituição dos termos que ocorrem no esboço por variáveis. Esta substituição pode ser feita usando diferentes estratégias de variabilização. Aqui descrevemos duas delas: a *estratégia de variabilização simples* e a *estratégia de variabilização completa*.

Para variabilizar um esboço seguindo a estratégia de variabilização simples substitui-se cada termo por uma variável, correspondendo a mesma variável a diferentes ocorrências do mesmo termo. Assim, a cláusula a extrair do esboço $p(a,z)\leftarrow q(a,c),t(a,c,z)$ é

p(A,Z)←q(A,C),t(A,C,Z). Este método de variabilização pressupõe que a duas variáveis diferentes correspondem dois termos diferentes no esboço e tem a vantagem de fazer corresponder a cada esboço uma única cláusula.

O método de variabilização completa retorna, para cada esboço, o conjunto de cláusulas que têm esse esboço como instância. A variabilização completa de um esboço *p(a,z)←q(a,c),t(a,c,z)* é um conjunto de 20 cláusulas incluindo

p(A,Z)←q(A,C),t(A,C,Z),
p(A,Z)←q(B,C),t(A,C,Z),
p(A,Z)←q(A,C),t(B,C,Z),
p(A,Z)←q(A,C),t(A,D,Z),

etc.

Se a função *Variabiliza* usa o processo de variabilização completa então retorna um conjunto de cláusulas em vez de apenas uma. Nesse caso as condições de paragem do Algoritmo 2 têm de ser verificadas para cada cláusula resultante da variabilização. O algoritmo pára se alguma das cláusulas satisfizer as condições. O resultado de *ConstróiCláusula* é então o conjunto de variabilizações (cláusulas) que satisfazem o critério de paragem.

Na presente versão do SKILit apenas está disponível o processo de variabilização simples. A estratégia de variabilização pode ser uma opção do utilizador. Podem também definir-se outras estratégias de variabilização intermédias.

4.7.2 O operador de refinamento

O conjunto de refinamentos de um esboço *Esb* é dado pelo operador de refinamento ρ (Algoritmo 3). Este operador aplica-se a um esboço *Esb* e identifica um literal de esboço *$P(X,Y)* para consolidar (*X* representa o conjunto de argumentos de entrada e *Y* os de saída). A tarefa do operador de refinamento é a de encontrar todas as substituições

possíveis para para este literal de esboço. Cada substituição é composta por um literal operacional e um literal de esboço. Em último caso, o literal *$P(X,Y)* também pode ser removido.

O operador de refinamento consolida o esboço a partir dos argumentos de entrada para os argumentos de saída, i.e., apenas introduz literais operacionais cujos argumentos de entrada estejam ligados aos argumentos de entrada da cabeça do esboço através de literais operacionais apenas. Por esse motivo, o literal *$P(X,Y)* seleccionado deve ser um literal cujos argumentos de entrada *X* sejam termos operacionalmente ligados dentro do esboço. Se existir mais do que um literal de esboço nessas condições, é escolhido o que se encontrar mais à esquerda. Para facilitar a descrição do Algoritmo 3 considera-se que o esboço a refinar está *sintacticamente ordenado* (Secção 4.5.1). Isto significa que o literal *$P(X,Y)* seleccionado será sempre o literal mais à esquerda.

Identificado o literal de esboço *$P(X,Y)* a refinar, é construído um conjunto de átomos que, pertencendo ao modelo de $P \cup BK \cup E^+$, têm como argumentos de entrada termos em *X*. A este conjunto de átomos chamamos o *sub-modelo relevante* (ver Algoritmo 4).

Cada elemento $Pred(X_M,Y_M)$ do sub-modelo relevante *ModRel* vai dar origem a um refinamento. Para tal substitui-se o literal de esboço por uma conjunção $Pred(X_M,Y_M)$,*$Pnovo*$(X_{Pnovo},Y_{Pnovo})$, onde *$Pnovo* representa o novo literal de esboço. O novo literal de esboço representa novas oportunidades de consolidação em passos de refinamento subsequentes. O conjunto de termos de entrada X_{Pnovo} inclui os termos em *X* e em Y_M e corresponde a todos os termos do esboço que são argumentos de entrada da cabeça ou argumentos de saída de um literal operacional do corpo. O conjunto de termos de saída Y_{Pnovo} inclui os termos em *Y* que ainda não foram operacionalmente ligados.

Se o conjunto de termos de saída *Y* de *$P(X,Y)* fôr vazio, o algoritmo também deve retornar o refinamento obtido pela supressão deste literal de esboço. Fazer um literal de esboço desaparecer permite que que o SKIL passe ao literal de esboço seguinte e que eventualmente consolide todo o esboço.

Procedimento ρ
entrada: esboço *Esb*
P,BK,E^+
saída: um conjunto de esboços refinamentos de *Esb*
Esb := $e \leftarrow \alpha, \$P(X,Y), \beta$ em que
α e β são sequências de literais,
$P(X,Y) é o literal de esboço mais à esquerda cujos
argumentos de entrada são termos ligados.
X representa o conjunto dos seus argumentos de entrada,
Y o conjunto de argumentos de saída,
se não existe um *$P(X,Y)* nessas condições **retorna** $\varnothing$
ModRel := *SubModeloRelevante(X,P,BK,*E^+*,*$e \leftarrow \alpha$*)*
NovosLiterais := { (*Pred(*X_M*,*Y_M*),$Pnovo(*$X \cup Y_M$*, Y–*$Y_M$*)*) |
Pred(X_M*,*Y_M*)* $\in$ *ModRel* e *$Pnovo* é um novo literal de esboço }
Refin := { $e \leftarrow \alpha, \gamma, \beta$ | $\gamma \in$ *NovosLiterais* }
se *Y*=$\varnothing$ **então** *Refin* := *Refin* $\cup$ { $e \leftarrow \alpha, \beta$ }
retorna *Refin*

Algoritmo 3: Operador de Refinamento

Exemplo 4.6: Seja *Esb* o esboço

grandfather(+tom,–bob)$\leftarrow$*father(+tom,–anne),$P1(+tom,+anne,–bob).*

Esb tem um literal de esboço (*$P1(+tom,+anne,–bob)*). Cada elemento do conjunto dos seus refinamentos é construído substituindo este literal de esboço por uma conjunção de um literal operacional e de um novo literal de esboço. Eis o conjunto de refinamentos, usando os predicados definidos no Exemplo 4.5:

Refin = { (*grandfather(+tom,–bob)*$\leftarrow$
father(+tom,–anne),
mother(+anne,–bob),
$P2(+tom,+anne,+bob).),
(*grandfather(+tom,–bob)*$\leftarrow$
father(+tom,–anne),
mother(+anne,–chris),
$P3(+tom,+anne,+chris,–bob).) }

♦

4.7.3 O sub-modelo relevante

Os literais operacionais que vão substituir o literal de esboço correspondem a um conjunto *ModRel* de factos fechados derivados do programa $P \cup BK \cup E^+$. Este conjunto *ModRel* é um sub-conjunto relevante do modelo de $P \cup BK \cup E^+$ (adiante designado por sub-modelo relevante) e é construído da seguinte forma (ver Algoritmo 4). Para cada predicado admissível são construídas perguntas (*queries*) usando os argumentos de entrada do literal de esboço. As perguntas são colocadas ao programa $P \cup BK \cup E^+$ usando um interpretador de programas de profundidade limitada (Secção 4.7.4). O conjunto de respostas dadas pelo interpretador é o sub-modelo *ModRel* pretendido.

Procedimento *SubModeloRelevante*
entrada: $X, P, BK, E^+, e \leftarrow \alpha$
saída: *ModRel* um sub-modelo relevante de $P \cup BK \cup E^+$
ModRel := $\varnothing$
Predicados := *PredicadosSeguintes*$(e \leftarrow \alpha)$
para cada $Pred \in Predicados$
    $Perguntas := \{ Pred(X_p, Y_p) \mid X_p \subseteq X, Y_p$ são variáveis $\}$
    $Átomos := \{ Q\theta \mid Q \in Perguntas$ e $\theta \in Int(P \cup BK \cup E^+, Q, \vdash) \}$
    $ModRel := ModRel \cup Átomos$
seguinte
ModRel := *ModRel*$-\alpha$ (elimina repetições de literais)
ModRel := *Poda*(*ModRel*)
retorna *ModRel*

Algoritmo 4: Construção do sub-modelo relevante

Exemplo 4.7: Os argumentos de entrada {*tom, anne*} do literal de esboço *$P1(+tom, +anne, –bob)* no esboço seguinte:

```
grandfather(+tom,–bob)←                                    (1)
    father(+tom,–anne),
    $P1(+tom,+anne,–bob).
```

vão ser usados para formular perguntas com os predicados admissíveis *father/2* e *mother/2* (vamos supor que estes são predicados admissíveis). Considerando as

definições dos predicados tais como apresentadas no Exemplo 4.5 as perguntas possíveis seriam

Perguntas = { *father(tom,X), father(anne,X), mother(tom,X), mother(anne,X)* }

A primeira e a quarta pergunta obtêm duas substituições resposta cada, dando assim origem a dois factos cada uma. À segunda e à terceira pergunta não corresponde nenhum facto. O conjunto de factos construído a partir das respostas é então

Factos = { *father(tom,anne), father(tom,jack),*
mother(anne,bob), mother(anne,chris) }

O sub-modelo relevante é

RelMod = {*father(tom,jack), mother(anne,bob), mother(anne,chris)*}

Note-se que *father(tom,anne)* foi excluído do sub-modelo relevante uma vez que já pertence ao esboço em (1).

♦

Porque estamos nós interessados num sub-modelo de $P \cup BK \cup E^+$? O conhecimento de fundo *BK* permite a introdução de predicados auxiliares. Os exemplos positivos E^+ permitem a introdução de literais recursivos. As cláusulas em *P* previamente induzidas aceleram a indução de novas cláusulas recursivas. Embora seja possível aprender cláusulas recursivas a partir de sub-modelos relevantes de $BK \cup E^+$ apenas (sem *P*), isso faria com que o sucesso do sistema fosse demasiado dependente da escolha dos exemplos positivos. Este assunto será desenvolvido no Capítulo seguinte.

O Algoritmo 4 retira ao sub-modelo relevante átomos que já existam como literais no esboço que está a ser refinado. Este controlo evita a repetição desnecessária de literais na cláusula final.

4.7.3.1 Poda

A função *Poda* consiste nos dois passos heurísticos abaixo descritos. Uma versão não-heurística do Algoritmo 4 pode ser obtida retirando a chamada à função *Poda*.

Primeiro passo heurístico:

> *ModRel* := *ModRel*–{ *e*' | *e*' tem o mesmo predicado que *e*
> e os seus argumentos de entrada são
> um subconjunto dos argumentos de entrada de *e*}

Segundo passo heurístico:

> *ModRel* := *ModRel*–{ *L* | *L* introduz termos produzidos por $e \leftarrow \alpha$ }

O primeiro passo heurístico retira ao sub-modelo relevante alguns átomos que originam literais recursivos potencialmente causadores de não-terminação no programa sintetizado. O critério é retirar todos os átomos cujos argumentos de entrada sejam um subconjunto dos argumentos de entrada da cabeça do esboço. Assim não teremos cláusulas como $p(X) \leftarrow p(X)$ nem como $p(X,Y) \leftarrow p(Y,X)$. Este é um controlo elementar de não-terminação, que não evita todas as situações indesejáveis. Mas não esqueçamos que o interpretador de programas usado pelo SKIL tem ele próprio um mecanismo de prevenção de não terminação: o controlo de profundidade das computações.

O segundo passo heurístico remove do sub-modelo relevante os átomos que tentam reintroduzir termos já existentes em $e \leftarrow \alpha$. O conjunto de termos de saída de um átomo *L* do sub-modelo relevante deve ser disjunto do *conjunto de termos produzidos* por $e \leftarrow \alpha$.

Definição 4.12: Dada a cláusula $e \leftarrow \alpha$, e as declarações de modo de entrada/saída dos predicados envolvidos, o conjunto de termos *produzidos* pela cláusula é

> $in(e) \cup$ { termos direccionalmente ligados de α relativamente a $in(e)$ }

em que $in(e)$ é o conjunto de termos de entrada de *e*. O conjunto de termos produzidos por $e \leftarrow \alpha$ denota-se por $produzidos(e \leftarrow \alpha)$ ♦

Assim, cada átomo *L* do sub-modelo relevante gerado pelo Algoritmo 4 deve satisfazer a seguinte condição:

$$out(L) \cap produzidos(e \leftarrow \alpha) = \emptyset$$

em que *out(L)* denota o conjunto de termos de saída do átomo *L*.

Os átomos que não satisfazem esta restrição são eliminados uma vez que, após variabilização do esboço, iriam corresponder a literais potencialmente inúteis. Esta é uma heurística razoável uma vez que o objectivo do processo de refinamento é o de produzir os termos de saída do exemplo dado, sendo tipicamente desnecessário produzir cada termo mais do que uma vez. Todavia, sob esta heurística e dado um exemplo, algumas cláusulas que o cobrem podem não ser sintetizáveis.

Exemplo 4.8: Seja $e \leftarrow \alpha$ no Algoritmo 4 *rv(+[3,2],–[2,3])←dest(+[3,2],–3,–[2])*. Neste caso o átomo *rv(+[2],–[2])* não estará em *ModRel* porque

$$out(\ rv(+[2],–[2])\ ) = \{[2]\}$$
$$produzidos(\ rv(+[3,2],–[2,3]) \leftarrow dest(+[3,2],–3,–[2])\ ) = \{\ [3,2],\ 3,\ [2]\ \}$$
$$\{[2]\} \cap \{\ [3,2],\ 3,\ [2]\ \} = \{[2]\} \neq \emptyset$$

Logo a cláusula *rv(A,B)←dest(A,C,D), rv(D,D), const(B,C,D)* nunca será sintetizada. ♦

O uso deste filtro reduz o número de refinamentos de esboço possíveis em cada passo de refinamento, assim como reduz o factor de ramificação (*branching factor*) do processo de procura aumentando consequentemente a eficiência.

Todavia, este filtro tem a desvantagem de causar incompletude na construção de cláusulas.

Exemplo 4.9: Suponhamos que o exemplo e_1 é *rv([1,2],[2,1])*. A cláusula recursiva é

rv(A,B)←dest(A,C,D),rv(D,E),addlast(E,C,B).

O esboço a encontrar pelo SKIL é

rv([1,2],[2,1])←dest([1,2],1,[2]),rv([2],[2]),addlast([2],1,[2,1]).

Este esboço nunca é produzido pelo SKIL a partir do exemplo *rv([1,2],[2,1])* se fôr usado o filtro que elimina repetições. Quando o SKIL refina *rv([1,2],[2,1])←dest([1,2],1,[2]),$Px(…),* o átomo *rv([2],[2])* não é incluído no sub-modelo relevante porque tenta reintroduzir o termo *[2]*.♦

4.7.4 O interpretador de profundidade limitada

O método de síntese seguido pelo SKIL envolve a utilização da resolução SLD/SLDNF nas seguintes situações:

- Testes de cobertura de exemplos positivos e negativos;
- Construção do sub-modelo relevante.

A resolução SLD pode ser problemática na prática devido à possibilidade de surgirem computações infinitas ou mesmo demasiado longas. Para garantir a terminação do processo de síntese, o interpretador de programas usado pelo SKIL dispõe de um mecanismo que controla a profundidade de cada refutação.

Definição 4.13: Seja D uma derivação de um programa P. O *nível de invocação* de uma ocorrência C_i de uma cláusula $C \in P$ numa derivação D é definido como $ni(C_i,D)$:

$ni(C_i,D) = 0$ se C_i está no primeiro passo de D, i.e. $D = ((Q,C_i,\theta),\ldots)$.

$ni(C_i,D) = k+1$ se C_i resolve com um literal cuja primeira ocorrência é na resolvente R_{j+1} em D, sendo R_{j+1} obtido resolvendo R_j e C_j, e $ni(C_j,D)=k$. ♦

Exemplo 4.10: Considere o seguinte programa de ordem zero:

a←b,a. C_1

$a \leftarrow c.$ C_2
$b.$ C_3
$c.$ C_4

A Figura 4.7 mostra uma possível derivação do programa.

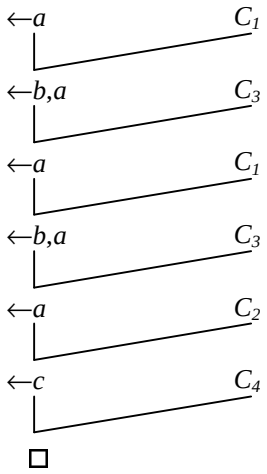

Figura 4.7: Uma derivação do programa..

Simbolicamente, a derivação é representada por

$$D = ((\leftarrow a, C_{1,1}), (\leftarrow b,a, C_{3,1}), (\leftarrow a, C_{1,2}), (\leftarrow b,a, C_{3,2}), (\leftarrow a, C_{2,1}), (\leftarrow c, C_{4,1}), \square)$$

(as substituições não são consideradas uma vez que aqui não são necessárias):em que $C_{k,i}$ representa a i-ésima ocurrência da cláusula C_k.

O nível de invocação de $C_{1,1}$ é 0 uma vez que se encontr no primeiro passo da derivação. O nível de invocação de $C_{3,1}$ é 1

$$ni(C_{3,1},D) = 1+ni(C_{1,1}, D)$$

uma vez que $C_{3,1}$ resolve com o literal b introduzido por $C_{1,1}$. O nível de invocação de $C_{1,2}$ também é 1. Quanto ao resto da derivação,

$$ni(C_{3,2},D) = 2 = 1+ni(C_{1,2}, D) = 1+1$$

$ni(C_{2,1},D) = 2 = 1+ni(C_{1,2}, D) = 1+1$

$ni(C_{4,1},D) = 3 = 1+ni(C_{2,1}, D) = 1+2$

♦

Podemos agora definir a profundidade de uma refutação em termos do nível de invocação máximo de uma cláusula em todas as derivações da árvore SLD.

Definição 4.14: Seja P um programa lógico definido e $\leftarrow Q$ uma pergunta. A *profundidade da refutação*, $prof(\leftarrow Q,P)$, de $\leftarrow Q$ a partir de P é o nível de invocação máximo de todas as ocorrências de cláusulas na árvore-SLD de derivação T de $\leftarrow Q$:

$prof(\leftarrow Q,P) = max(\{\ ni(C_i,D) \mid D$ é um ramo de T e C_i ocorre em $D\ \})$

♦

A extensão à resolução SLDNF das duas noções acima definidas é natural.

O interpretador de profundidade limitada responde apenas às perguntas que admitem uma refutação com profundidade inferior a um certo limite h. Quando, no decurso de uma demonstração, o limite é ultrapassado, o interpretador falha em dar uma resposta.

Definição 4.15: Seja P um programa, e $\leftarrow Q$ uma pergunta, um *interpretador com limite de profundidade h* pode ser definido como,

$Int(P,\leftarrow Q, \vdash_h) = \{\ \theta \mid P \vdash_h Q\theta\ \}$

em que $\vdash_h$ representa a relação de derivabilidade

$P \vdash_h Q$ se e só se $P \vdash Q$ e $prof(\leftarrow Q,P) \leq h$.

♦

É comum encontrarmos em sistemas de PLI algum tipo de controlo de profundidade das computações. O interpretador utilizado no SKIL faz um controlo de profundidade semelhante ao sugerido por Shapiro no seu sistema MIS [109] para diagnosticar programas cíclicos. Muggleton e Feng utilizam a noção de modelo *h-easy* para construirem subconjuntos dos modelos de um programa [82].

Definição 4.16: Dado um programa lógico *P*, um átomo *q* é *h-easy* relativamente a *P* se e só se existe uma derivação de *q* a partir de *P* envolvendo no máximo *h* passos de resolução. O *modelo h-easy* de Herbrand de *P* é o conjunto de todas as instanciações de átomos *h-easy* relativamente a *P*.♦

O modelo *h-easy* de um programa *P* corresponde, grosso modo, ao conjunto de factos que se podem derivar de *P* utilizando um interpretador de profundidade limitada *h*. Para que o modelo *h-easy* seja finito as cláusulas de *P* devem ser 'range restricted'.

De Raedt criticou a utilização de um limite baseado na profundidade da computação alegando que é mais intuitivo impôr um limite à complexidade dos termos envolvidos na computação [21].

Definição 4.17: Um átomo $f(t_1,\ldots,t_n)$ é *h-complexo* se e só se $\forall i$: profundidade do termo $t_i \leq h$ (pág. 49).♦

Um *modelo h-complexo* de um programa *P* corresponde ao conjunto de átomos para os quais existe uma derivação a partir de *P* envolvendo apenas termos h-complexos. Um programa *P* diz-se *h-conforme* se para todos os átomos *q* h-complexos a árvore SLD de *q* a partir de *P* contém apenas átomos h-complexos.

Embora fosse simples adoptar a abordagem h-complexa para controlar a terminação no SKIL, acreditamos que os resultados práticos obtidos pelo método não seriam muito diferentes. Por outro lado, um interpretador com controlo de complexidade de termos seria, em geral, computacionalmente mais pesado. Para programas h-comformes o controlo de complexidade pode ser feito estaticamente. Infelizmente, para garantirmos

que um programa é h-conforme temos de impôr várias condições sintáticas às cláusulas entre as quais a de que todas as variáveis do corpo aparecem na cabeça da cláusula. Esta condição é, no entanto, demasiado forte.

4.7.5 Vocabulário e gramática de estrutura clausal (GEC)

Os predicados admissíveis que podem ser utilizados na construção do sub-modelo são dados pela função *PredicadosSeguintes* invocada pelo Algoritmo 4. Esses predicados são determinados em primeiro lugar pela declaração dos predicados admissíveis, que constituem o *vocabulário* que o sistema de síntese dispõe para a construção de cláusulas. A função *PredicadosSeguintes* pode ser definida de uma forma simples retornando o conjunto dos predicados do vocabulário. Esta é a solução adoptada habitualmente pelos sistemas de PLI.

No entanto, o desenvolvimento semi-automático de programas devia explorar o *conhecimento de programação* [38,111]. O conhecimento respeitante ao processamento de objectos estruturados, tais como listas, poderia incluir, por exemplo, o seguinte. Se queremos processar um objecto usando um procedimento *P*, decompomos esse objecto em partes, chamamos o mesmo procedimento recursivamente e juntamos as soluções parciais. O sistema SKIL permite que este tipo de conhecimento seja expresso sob a forma de uma *gramática de estrutura clausal* (GEC).

Uma gramática de estrutura clausal define as sequências de nomes de predicados admissíveis no corpo das cláusulas a sintetizar. As GEC são definidas usando a notação de *gramáticas de cláusulas definidas* (*definite clause grammars*, DCG).

As regras do topo das GEC's utilizadas têm a forma

$$body(P) \text{--}> L_1(<O_1>,<N_1>),\ldots,recurs(<O_r>,<N_r>,P),\ldots,L_n(<O_n>,<N_n>).$$

em que, para cada $L_i(<O_i>,<N_i>)$

- L_i é o nome de um grupo de literais (e.g. literais de teste, literais de decomposição, etc.),

- $<O_i>$ ou é * ou +. O símbolo * significa que a sequência de literais pode ser vazia. O símbolo + significa que deve haver pelo menos um literal na sequência.

- $<N_i>$ é um inteiro maior do que 0, o qual limita o número máximo de literais admissíveis no grupo.

- *P* é uma variável da DCG.

O grupo *recurs* é um grupo especial para literais recursivos. O único nome de predicado admitido neste grupo é o do predicado a ser sintetizado. O seu nome é transmitido pela variável *P*.

Para cada L_i a GEC contém um conjunto de regras da forma

L_i(_,N)-->lit_L_i,{N>0}.
L_i(_,N)-->lit_L_i,{N2 is N-1},L_i(+,N2).
L_i(,N)-->[].*

lit_L_i-->[<P_1>];...;[<P_k>].

em que $<P_i>$ é um predicado do grupo L_i, *lit_*L_i é um nome de predicado da DCG, e *N*, *N2* são variáveis da DCG.

O grupo especial *recurs* é definido pelo conjunto de regras

recurs(_,N,P)-->lit_recurs(P),{N>0}.
recurs(_,N,P)-->lit_recurs(P),{N2 is N-1},recurs(+,N2,P).
recurs(,N,P)-->[].*

lit_recurs(P)-->[P].

Exemplo 4.11: A GEC aqui mostrada descreve um conjunto de cláusulas recursivas. Começa por definir diferentes grupos de literais. O primeiro grupo decompõe certos

argumentos da cabeça da cláusula em sub-termos (usando predicados como *dest/3* que separa uma lista em cabeça e corpo). O segundo grupo contém literais de teste. O terceiro grupo permite a introdução de literais recursivos. Por último, o quarto grupo consiste em literais de composição cujo objectivo é o de construir os argumentos de saída a partir dos termos obtidos pelos literais anteriores (usando predicados como *append/3*). A estrutura geral da cláusula recursiva é descrita da seguinte forma:

body(P)-->decomp(+,2),test(,2),recurs(*,2,P),comp(*,2).*

onde o argumento *P* transporta o nome do predicado da cabeça (por exemplo *member/2* se estamos a sintetizar *member*). O número máximo de qualquer grupo de literais é 2. Todos os grupos de literais podem ser vazios excepto o grupo *decomp*. O grupo de decomposição é definido seguindo o modelo acima descrito:

decomp(_,N)-->lit_decomp,{N>0}.
decomp(_,N)-->lit_decomp,{N2 is N-1},decomp(+,N2).
decomp(,N)-->[].*

lit_decomp-->[dest/3];[pred/2];[partb/4].

O grupo de literais recursivos é também definido como foi indicado. Os grupos de teste e de composição são definidos de forma idêntica ao grupo de decomposição. Aqui mostramos apenas as regras de *lit_comp* e *lit_test*:

lit_test-->[null/1];[memberb/2].
lit_comp-->[appendb/3]; [addlast/3];[const/3].

Algumas cláusulas admitidas por esta GEC (assumindo que neste exemplo estamos a sintetizar o predicado *rv/2*) teriam a forma:

rv(_,_)←dest(_,_,_),rv(_,_).
rv(_,_)←pred(_,_,_),rv(_,_).
rv(_,_)←dest(_,_,_),rv(_,_),addlast(_,_,_).

Algumas cláusulas não admitidas pela GEC:

rv(_,_)←rv(_,_).

As cláusulas devem ter pelo menos um literal de decomposição.

rv(_,_)←rv(_,_),dest(_,_,_),rv(_,_).

Nenhuma cláusula pode ter um literal de decomposição entre dois literais recursivos.

rv(_,_)←dest(_,_,_),dest(_,_,_),dest(_,_,_).

O número máximo de literais de decomposição é 2.♦

Quando o Algoritmo 4 invoca a função *PredicadosSeguintes*, que tem como argumento a parte do esboço $e \leftarrow \alpha$ à esquerda do literal *$P(...)*, é construído o conjunto de nomes de predicados admissíveis que, segundo a GEC, podem seguir α. A GEC não restringe os argumentos dos literais. Simplesmente define sequências aceitáveis de predicados que podem aparecer numa cláusula.

Seria relativamente simples estender as GEC de forma a que os argumentos dos literais fossem restringidos também. Preferimos no entanto adoptar esta solução simples uma vez que isso torna as GEC mais fáceis de escrever e de modificar. Em qualquer dos casos, a escolha dos argumentos dos literais é restringida pelo mecanismo de construção de cláusulas, o qual segue sempre uma ligação relacional e tem em conta os tipos dos predicados.

A função *PredicadosSeguintes* invoca o predicado *body/3* definido pela GEC da seguinte forma: o primeiro argumento vem instanciado com o nome do predicado a definir, o segundo argumento é uma lista cujos primeiros elementos representam a sequência de nomes dos predicados em α, o elemento seguinte dessa lista é uma variável que irá ser instanciada com o nome do predicado que se pode seguir na sequência, o resto da lista é uma variável não instanciada. O terceiro argumento é a lista vazia.

Exemplo 4.12: $e \leftarrow \alpha$ é a cláusula

sort([2,1],[1,2])←dest([2,1],2,[1]).

Então, dada a GEC do Exemplo 4.11, o conjunto de predicados que se podem seguir é {*dest/3, partition/4, sort/2*}. Para obter este conjunto basta coleccionar as respostas dadas à pergunta

?-body(sort/2,[dest/3,PRED|_],[]).

A variável *PRED* irá ser sucessivamente unificada com *dest/3, partition/4* e *sort/2*. Se se considerassem todos os predicados do vocabulário, independentemente da GEC, o conjunto de *PredicadosSeguintes* seria

{*dest/3, partition/4, null/1, member/2, const/3, append/3*}

♦

A gramática de estrutura clausal permite-nos descrever uma circunscrição de linguagem (*language bias*) adequada. O método é bastante poderoso pois a mesma gramática pode cobrir uma grande classe de definições de predicados, sendo por isso potencialmente reutilizável.

4.7.6 Verificação de tipos

Os tipos declarados na especificação são também verificados durante a construção do sub-modelo relevante. Esse passo não foi explicitamente incluído no Algoritmo 4 por uma questão de clareza da descrição desse algoritmo. Na realidade, o conjunto de perguntas construído pela instrução

$$Perguntas := \{\ Pred(X_p,Y_p) \mid X_p \subseteq X,\ Y_p \text{ são variáveis}\ \}$$

do Algoritmo 4 excluí aquelas cujos argumentos de entrada não estão conformes com a declaração de tipos. Para tal, o SKIL verifica se cada termo de entrada pertence ao domínio do respectivo tipo declarado, ou, por outras palavras, se o n-tuplo de argumentos da pergunta é *compatível* com a declaração de tipos (Secção 3.2.4). Esta

verificação é feita utilizando as definições de tipos (ver Apêndice B). Para os predicados cujo tipo não é declarado aceitam-se quaisquer termos de entrada.

4.8 Propriedades do operador de refinamento

Nesta Secção discutimos algumas das propriedades teóricas do operador de refinamento do SKIL. Estamos principalmente interessados em determinar se o operador de refinamento consegue sempre encontrar uma cláusula que cubra um exemplo dado caso a cláusula esteja no espaço de procura.

Dado um programa P e um exemplo $e(X,Y)$ tal que $in(e(X,Y))=X$ e $out(e(X,Y))=Y$, se há uma ligação relacional α de X para Y tal que $P \vdash \alpha$, então o operador de refinamento do SKIL (ρ) consegue encontrá-la.

Se temos um exemplo positivo sem um esboço associado, o operador de refinamento ρ começa pelo esboço mínimo $e(X,Y) \leftarrow \$P1(X,Y)$ e encontra todos os refinamentos $e(X,Y) \leftarrow p(X2,Y2),\$P2(X3,Y3)$ tais que $P \vdash p(X2,Y2)$ e $X2 \subseteq X$, em que $\$P2(X3,Y3)$ é um novo literal de esboço cujos argumentos $(X3,Y3)$ são uma conbinação de (X,Y) e de $(X2,Y2)$. A aplicação repetida de ρ produz todas as ligações relacionais de X para Y. Se há um esboço associado o operador de refinamento trata cada literal de esboço de uma forma semelhante. Dado um programa P e um esboço Sk para o qual há uma cláusula C que é uma variabilização de uma consolidaçãpo de Sk então o SKIL encontra essa cláusula.

Seguidamente formalizaremos o que foi afirmado acima. Mostramos que o operador de refinamento do SKIL consegue encontrar todos os refinamentos operacionais interessantes de um esboço dado. Em consequência, o SKIL consegue encontrar todas as variabilizações desses refinamentos. Começamos por definir o conceito de consolidação. Os refinamentos interessantes de um esboço são as suas consolidações.

Note-se que utilizamos aqui uma notação do tipo cláusula para representar conjuntos de literais. A sequência α_1,α_2 representa o conjunto de literais $\alpha_1 \cup \alpha_2$, em que α_1 e α_2 são conjuntos de literais. A sequência L,α representa o conjunto $\{L\} \cup \alpha$, em que L é um único literal e α é um conjunto de literais.

Definição 4.18: Um conjunto de literais α é uma *consolidação* de um conjunto β de literais operacionais ou de esboço, denotado $\alpha \angle \beta$ sse:

a) $\alpha = \beta$;
b) β é da forma *$P(X,Y)* e α é uma ligação relacional de um conjunto de termos $SX \subseteq X$ para um conjunto de termos $SY \supseteq Y$;
c) β é da forma (L,β_2), em que L é um literal de esboço ou operacional, α é da forma (α_1,α_2), $\alpha_1 \angle L$ e $\alpha_2 \angle \beta_2$. ♦

Intuitivamente, um conjunto de literais β é uma consolidação de um literal de esboço *$P(X,Y)* se β produz todos os termos de saída *Y* de *$P(X,Y)* a partir de um subconjunto dos seus termos de entrada *X*. Note-se que o conjunto vazio é uma consolidação aceitável para qualquer literal de esboço que não tenha termos de saída. A noção de consolidação é recursivamente estendida a conjuntos arbitrários de literais.

Exemplo 4.13: Suponhamos que temos dois predicados *p(+,–)* e *q(+,–,–)*. O conjunto de literais *p(a,b),q(b,c,d)* é uma consolidação possível do literal de esboço *$P1(+a,–c,–d)* uma vez que há uma ligação relacional de {*a*} para {*c,d*}. Também é verdade que *p(+a,–b),q(+b,–c,–d)* é uma consolidação de *$P2(+a,–c)* uma vez que, em particular, há uma ligação relacional de {*a*} para {*c*}. O conjunto vazio é uma consolidação de *$P3(+a,+b)*. Outra consolidação deste literal de esboço é *p(+a,–b),p(+b,–c)*.

Uma consolidação de *p(+a,–b),$P4(+b,-d),p(+d,–f)* é *p(+a,–b),p(+b,–c),$P5(+b,+c,–d),p(+d,–f)*. ♦

Um esboço é uma consolidação de outro se ambos têm a mesma cabeça e há uma relação de consolidação entre os seus corpos.

Definição 4.19: Sejam S_1 e S_2 dois esboços. S_2 é uma *consolidação* de S_1, denotado $S_2 \angle S_1$, sse $S_1=(H \leftarrow \alpha_1)$, $S_2=(H \leftarrow \alpha_2)$ e $\alpha_2 \angle \alpha_1$. ♦

Um operador de refinamento de esboços produz consolidações de um esboço.

Definição 4.20: Um *operador de refinamento de esboços* (ORE) ρ é um operador que, dado um esboço S, retorna um conjunto de esboços, denotado por $\rho(S)$, em que para cada $S' \in \rho(S)$ temos $S' \angle S$. ♦

O operador de refinamento do SKIL tem quatro argumentos: $\rho(S,P_0,BK,E^+)$. O primeiro argumento é o esboço a refinar. Os outros são o programa inicial P_0, o conhecimento de fundo BK e os exemplos positivos E^+. Nesta Secção consideramos estes últimos três argumentos como um só programa $P = P_0 \cup BK \cup E^+$. Pela mesma razão invocamos *SubModeloRelevante* com o conjunto vazio nos terceiro e quarto argumentos. Como forma abreviada de $\rho(S,P_0,BK,E^+)$ escrevemos $\rho(S)$.

Definição 4.21: O conjunto de refinamentos de um esboço S obtido por aplicação repetida de um ORE ρ é denotado como $\rho^*(S) = \{S\} \cup \rho(S) \cup \rho^2(S) \cup \rho^3(S) \cup \ldots$. ♦

Definimos agora a noção de completude de um operador de refinamentos de esboços em termos da noção de consolidação.

Definição 4.22: Seja ρ um ORE, SS um conjunto de esboços, S_1 um esboço sintaticamente ordenado em SS, e S_2 um esboço operacional de SS tal que $S_2 \angle S_1$. O ORE ρ é *completo* em SS sse $S_2 \in \rho^*(S_1)$. ♦

Teorema 4.1: Dado um programa P, o operador de refinamento ρ do SKIL é completo no conjunto de esboços $SS=\{S \mid$ para cada literal operacional L no corpo de S, $P \vdash L\}$.

Demonstração: Seja S um esboço operacional em SS e S_1 um esboço arbitrário em SS tal que $S \angle S_1$. Temos de provar que $S \in \rho^*(S_1)$,.

Se S_1 não tem literais de esboço então, por definição de consolidação $S=S_1$. Pela definição de ρ^*, temos que $S \in \rho^*(S_1)$.

Se S_1 tem pelo menos um literal de esboço, então S_1 é da forma $H \leftarrow \alpha_1, \$P(X,Y), \beta_3$, em que α_1 é uma sequência de literais operacionais. Por definição de consolidação S é da forma $H \leftarrow \alpha_1, \alpha_2, \alpha_3$, em que $\alpha_2 \angle \$P(X,Y)$ e $\alpha_3 \angle \beta_3$.

Se $\alpha_2 = \varnothing$ então o conjunto de termos de saída Y tem de ser vazio, doutra forma não teríamos $\alpha_2 \angle \$P(X,Y)$. Neste caso $(H \leftarrow \alpha_1, \beta_3) \in \rho(H \leftarrow \alpha_1, \$P(X,Y), \beta_3)$ uma vez que, se Y é vazio, um dos refinamentos é obtido por eliminação do literal de esboço $\$P(X,Y)$.

Se $\alpha_2 \angle \$P(X,Y)$ e $\alpha_2 \neq \varnothing$ deve haver um literal operacional $L \in \alpha_2$ tal que $in(L) \subseteq X$. Suponhamos que tal literal não existe. Então nenhum termos em Y está direccionalmente ligado em α_2 relativamente a X o que contradiz $\alpha_2 \angle \$P(X,Y)$.

Uma vez que $P \vdash L$ temos que $L \in SubModeloRelevante(X,P,\varnothing,\varnothing, H \leftarrow \alpha_1)$. Isso é justificado pelo facto de que o sub-modelo relevante é obtido construindo todas as perguntas (*queries*) com todos os predicados permitidos para $H \leftarrow \alpha_1$ com todas as combinações possíveis de argumentos de entrada tirados de X. Logo $(H \leftarrow \alpha_1, L, \$P2(X \cup in(L), Y\text{-}out(L)), \beta_3) \in \rho(S_1)$.

Agora seja α_2' igual a α_2 menos L. Temos que $\alpha_2' \angle \$P2(X \cup in(L), Y\text{-}out(L))$ porque α_2 liga $SX_2 \cup out(L) \subseteq X \cup out(L)$ a $SY_2\text{-}out(L) \supseteq Y\text{-}out(L)$. Por isso podemos aplicar a α_2' o mesmo raciocínio que aplicamos a α_2 e concluír que $H \leftarrow \alpha_1, \alpha_2, \beta_3 \in \rho^{n+1}(S_1)$ assumindo que α tem n literais.

Aplicando o mesmo raciocínio aos restantes literais de esboço de S_1 como foi feito para $\$P(X,Y)$ podemos concluir que $H \leftarrow \alpha_1, \alpha_2, \alpha_3 \in \rho^k(H \leftarrow \alpha_1, \$P(X,Y), \beta_3)$, para algum inteiro k, i.e., $S \in \rho^*(S_1)$. ♦

Se considerarmos uma gramática de estrutura clausal *G*, o conjunto de esboços *SS* restringe-se aos esboços admitidos por *G*.

Teorema 4.2: Dado um programa *P*, um esboço *S* e uma cláusula $C = H_C \leftarrow B_C$, se há uma substituição θ tal que $C\theta \angle S$ e $P \vdash B_C\theta$, então o SKIL consegue encontrar a cláusula *C*, assumindo que a variabilização completa (Secção 4.7.1.1) é utilizada.

Demonstração: Pela completude de ρ e o pressuposto de que $P \vdash B_C\theta$ temos que $C\theta \in \rho^*(S)$. Logo o SKIL consegue encontrar o esboço $C\theta$ e em consequência consegue encontrar todas as variabilizações de $C\theta$ incluindo a cláusula *C*.♦

4.9 Uma sessão com o SKIL

Começamos por empregar o sistema SKIL para sintetizar o predicado *rv/2*. Este exemplo serve para ilustrar o funcionamento do sistema quando lhe são dados exemplos positivos e negativos bem escolhidos, um programa auxiliar como conhecimento de fundo e uma gramática de estrutura clausal. O resultado é um programa recursivo. No final do rasto (*trace*) do sistema são indicados tempo de processador gasto (em segundos) e o número total de refinamentos de esboço construídos pelo SKIL.

Especificação

```
mode( rv(+,-) ).
type( rv( list,list ) ).

rv([],[]).
rv([1,2,3],[3,2,1]).
rv([2,3],[3,2]).

-rv([1,2],[1,2]).
-rv([1,2,3],[2,1,3]).
-rv([1,2,3],[2,3,1]).
-rv([1,2,3,4],[3,4,2,1]).
```

Conhecimento de programação

```
background_knowledge( list ).                          % Apêndice A
clause_structure( decomp_test_rec_comp_2 ).            % Apêndice C
adm_predicates( rv/2, [const/3,dest/3,null/1,addlast/3,rv/2] ).
```

Resultado do SKIL:

```
?- skil(rv/2).

exemplo a cobrir: rv([],[])
cláusula c(12) gerada após 2 refinamentos:
rv(A,A)←
   null(A).

exemplo a cobrir: rv([1,2,3],[3,2,1])
cláusula c(13) gerada após 32 refinamentos:
rv(A,B)←
   dest(A,C,D),
   rv(D,E),
   addlast(E,C,B).

exemplo a cobrir: rv([2,3],[3,2])
exemplo coberto pela cláusula c(13)
Programa gerado (prv):

c(12):rv(A,A)←
   null(A).
c(13):rv(A,B)←
   dest(A,C,D),
   rv(D,E),
   addlast(E,C,B).

34 refinamentos (total)
2.200 secs
```

O conhecimento de fundo (*list*) contém as definições, declarações de tipo e de modo dos predicados admissíveis, entre outros (Apêndice A). A gramática de estrutura clausal usa uma estratégia de dividir e conquistar semelhante à do Exemplo 2.1 (Apêndice C). Para tal, cada cláusula tem no corpo uma sequência de literais de divisão (ou decomposição), uma sequência de teste, uma de literais recursivos, e outra de recomposição (ou composição).

Correndo o SKIL com os mesmos dados, mas sem que este utilize uma GEC, obtemos o mesmo resultado. Todavia o número de refinamentos sobe para 60 (quase o dobro) num problema que é relativamente simples. O tempo de processador gasto é também superior (cerca de 2.7 segundos).

As declarações de tipo também afectam o desempenho do sistema. Experimentamos retirar apenas a declaração de tipo do predicado auxiliar *addlast/3*. O número de refinamentos foi de 84 (em vez de 34) e o tempo gasto foi de 3.5 segundos.

Os predicados declarados como admissíveis também influenciam o esforço de procura. Esta influência tanto pode ser positiva, no sentido de diminuir o número de refinamentos considerados, como negativa, no sentido de aumentar esse número. Acrescentando, por exemplo, o predicado *append/3* à declaração de predicados admissíveis, obtemos o mesmo resultado ao fim de 86 refinamentos em 3.6 segundos.

Se, em vez dos três exemplos positivos, dermos ao SKIL o primeiro exemplo e um esboço, como indicado abaixo, o mesmo programa é sintetizado após 9 refinamentos e em 2/3 do tempo.

```
rv([],[]).    % positive example
sketch( rv([1,2,3],[3,2,1])←
        $P1([1,2,3],1,[2,3]), rv([2,3],[3,2]), $P2([3,2],1,[3,2,1]) ).
```

4.10 Limitações

O sistema SKIL conseguiu sintetizar uma definição recursiva a partir de três exemplos positivos bem escolhidos. Qualquer que seja a ordem de apresentação destes três exemplos, o resultado final do SKIL inclui sempre as duas cláusulas *c(12)* e *c(13)*. Apenas se registam variações nos tempos de síntese, surgindo também para algumas sequências de exemplos uma terceira cláusula redundante em relação às duas anteriores. Em qualquer dos casos, este conjunto de exemplos positivos parece suficiente para a indução das duas cláusulas relevantes.

Vamos agora experimentar um conjunto de exemplos positivos ligeiramente diferente.

rv([],[]).
rv([1,2,3],[3,2,1]).
rv([4,5],[5,4]).

Neste caso, o programa sintetizado pelo SKIL é

c(12):rv(A,A)←
null(A).

c(14):rv(A,B)←
dest(A,C,D),
dest(D,E,F),
addlast(F,E,D),
addlast(D,C,B).

Este programa não cobre o exemplo *rv([1,2,3],[3,2,1])* dado. O processo de procura de uma cláusula que cobrisse este exemplo terminou após ter construído todos os refinamentos de esboços dentro da circunscrição de linguagem. Em particular o SKIL não conseguiu induzir a cláusula recursiva *c(13)* gerada na experiência anterior.

Como iremos ver no Capítulo seguinte, a cláusula recursiva não surgiu devido à falta do exemplo *rv([2,3],[3,2])*. De facto, o SKIL tem dificuldades em gerar definições recursivas a partir de conjuntos de exemplos positivos que não sejam bem escolhidos, devido à estratégia de procurar ligações relacionais. Por esse motivo propomos uma estratégia de indução iterativa que consegue sintetizar cláusulas recursivas a partir de conjuntos de exemplos positivos como o apresentado no exemplo acima. Essa abordagem é descrita no Capítulo seguinte

4.11 Trabalho relacionado

4.11.1 Termos ligados

Em 1977 Steven Vere [122] abordou o problema da indução de produções relacionais a partir de exemplos em presença de um corpo de factos relevantes (conhecimento de fundo) usando uma técnica de ligação de termos de diferentes literais. Para Vere, uma *produção relacional* (*relational production*) tem a forma $\alpha \Leftarrow \beta$, em que α e β são conjunções de literais. De forma a poder incorporar literais do conhecimento de fundo em conjunções de literais dadas como exemplos, Vere propôs a noção de *cadeia de associações*. Depois de combinar estas cadeias de associação com os literais do exemplo, procurava generalizá-las. Dois literais L_1, L_2 têm uma associação $A_{i,j}(L_1,L_2)$ se o i-ésimo termo de L_1 é igual ao j-ésimo termo de L_2. Uma *cadeia de associações* é uma sequência de associações $A_{i_1,i_2}(L_1,L_2)$, $A_{i_3,i_4}(L_2,L_3),\ldots,$ $A_{i_n,i_{n+1}}(L_{n-1},L_n)$, em que para r ímpar $i_r \neq i_{r+1}$.

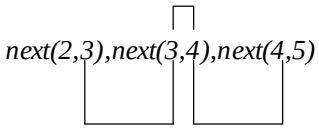

Figura 4.8: Exemplo de uma cadeia de associações de Vere.

A Figura 4.8 mostra o exemplo de uma cadeia de associações de Vere. Por uma questão de clareza, usamos uma notação semelhante à do Prolog em vez da notação usada por Vere. Um contraexemplo de uma cadeia de associações de Vere é a sequência de literais *next(2,3),next(3,4),odd(3)*. Para uma abordagem recente no âmbito da PLI a produções ver [23].

Embora as cadeias de associações e as ligações relacionais aqui descritas tenham um espírito semelhante, representam conceitos diferentes. Uma ligação relacional é definida em termos de argumentos de entrada e de saída e a sua intenção é estabelecer uma ligação entre dois conjuntos de tremos: o conjunto de argumentos de entrada de um

exemplo e o conjunto dos seus argumentos de saída. Uma cadeia de associação liga dois literais. Numa cadeia de associações há no máximo uma ligação entre quaisquer dois literais. As ligações relacionais são mais complexas uma vez que um literal pode estar ligado a muitos outros.

Richards e Mooney usam a *procura de caminhos relacionais* (*relational pathfinding*) no sistema FORTE [100] como parte de um método de especialização de cláusulas. A ideia desta técnica é ver o domínio de termos da base de Herbrand de um programa lógico como um hipergrafo de termos ligados pelas relações definidas no programa. Por exemplo, dado o exemplo positivo *uncle(arthur,charlotte)*, a procura de uma cláusula faz-se expandindo cada um dos termos do exemplo. Para tal consideram-se os dados conhecidos sobre a relação *parent/2*.

parent(cristopher,arthur).
parent(penelope,arthur).
parent(cristopher,victoria).
parent(penelope, victoria).
parent(victoria, charlotte).
parent(james, charlotte).
parent(victoria, colin).
parent(james, colin).

A expansão do termo *arthur* leva aos novos termos {*christopher, penelope*} (factos *parent(cristopher,arthur)* e *parent(penelope,arthur)*). A expansão do termo *charlotte* leva a {*victoria, james*} (factos *parent(victoria, charlotte)* e *parent(james, charlotte)*). Não há intersecção entre os dois conjuntos de termos obtidos na expansão. Expandindo os termos que resultaram da expansão de *arthur* vamos obter (finalmente) o termo *victoria* (ou o facto *parent(cristopher,victoria)* ou *parent(penelope, victoria)*). Temos assim uma intersecção entre o conjunto de termos obtidos a partir de *arthur*, {*chistopher, penelope*} $\cup$ {*victoria*}, e a partir de *charlotte*, {*victoria, james*}, que corresponde a um caminho relacional. Esse caminho relacional pode ser reescrito como

uncle(arthur,charlotte)$\leftarrow$*parent(christopher,arthur),*
parent(cristopher,victoria), parent(victoria, charlotte).

o que corresponde à cláusula

uncle(X,Y)←parent(Z,X),parent(Z,W),parent(W,Y).

A técnica da procura de caminhos relacionais (PCR) diferencia-se da que propomos no sistema SKIL em vários aspectos. Em primeiro lugar, o SKIL explora fortemente o modo de entrada/saída dos predicados envolvidos na definição. No SKIL faz-se, de certa maneira, a procura de um caminho relacional, mas essa procura é direccionada. Em segundo lugar, no sistema FORTE, quando o método de PCR produz uma cláusula que é sobre-geral, a especialização dessa cláusula é feita usando um método de subida mais rápida idêntico ao do FOIL [96]. No SKIL, a construção de uma cláusula faz-se usando um único operador de especialização (Algoritmo 3: Operador de Refinamento) que procura de ligações relacionais tendo em conta exemplos negativos e restrições de integridade . Temos assim um algoritmo de construção de cláusulas mais simples e que evita as desvantagens do método de subida mais rápida (cf. Secção 3.4.5).

4.11.2 Conhecimento genérico de programação

Como já foi referido no Capítulo 1, vários formalismos de representação do conhecimento genérico de programação têm sido propostos no contexto da construção indutiva de programas lógicos, nomeadamente os *grafos de dependências* de Wirth e O'Rorke [123], os *modelos de regras* de Kietz e Wrobel [56], e os *esquemas de cláusulas* de Feng e Muggleton [34]. Cohen [18] e Klingspor [58] usaram também a notação DCG para representarem a circunscrição de linguagem dos seus sistemas.

As gramáticas de estrutura clausal usadas no SKIL são menos expressivas do que outros formalismos como por exemplo as DCG's de Cohen, pois não permitem restringir os argumentos dos literais das cláusulas induzidas. A simplicidade das GEC é, no entanto, vantajosa no que diz respeito à sua concepção e manutenção.

4.12 Sumário

O sistema SKIL sintetiza programas lógicos definidos sem functores ou constantes a partir de uma especificação dada, de conhecimento de fundo e de conhecimento de programação dados. A especificação contém exemplos positivos, exemplos negativos, restrições de integridade e declaração de modo entrada/saída e de tipo do predicado a sintetizar. O conhecimento de programação contém gramáticas de estrutura clausal e esboços de algoritmo.

A síntese de um programa lógico no SKIL é feita construindo uma cláusula de cada vez. Cada cláusula é construída com base num esboço de algoritmo associado a um exemplo positivo. A estratégia de construção passa por consolidar o esboço procurando uma ligação relacional entre os argumentos do literal na cabeça do esboço. A cláusula candidata será extraída do esboço consolidado através de uma operação de variabilização e não deverá cobrir nenhum exemplo negativo. Para encontrar o esboço pretendido percorre-se um espaço de refinamentos usando um operador de refinamento apropriado. A gramática de estrutura clausal permite definir a estrutura das cláusulas a sintetizar. O operador de refinamento tem em conta esta informação.

A noção de consolidação de um esboço é definida formalmente e é relacionada com a noção de refinamento de esboço. Mostra-se que o operador de refinamento de esboços é completo relativamente às consolidações operacionais de um esboço, assumindo que não se utilizam as heurísticas de poda no sub-modelo relevante. Assumindo que a variabilização completa é imposta, caracterizamos o conjunto de cláusulas que podem ser encontrados pelo SKIL.

A principal limitação do SKIL, comum a muitos outros sistemas de PLI, é o facto de exigir uma boa escolha dos exemplos para sintetizar uma definição recursiva. Este problema vai ser abordado no Capítulo seguinte com a introdução da indução iterativa.

5. Indução Iterativa

Descrevemos o problema da indução de cláusulas recursivas e várias abordagens a este problema. Apresentamos o método da indução iterativa e o sistema SKILit que utiliza o método para sintetizar definições recursivas a partir de conjuntos esparsos de exemplos positivos. A indução iterativa aplicada ao sistema SKIL apresentado no Capítulo anterior consegue resolver assim a sua principal limitação.

5.1 Introdução

A indução de definições recursivas a partir de exemplos positivos de uma relação continua a ser uma tarefa difícil para um sistema de PLI. Por um lado encontramos os sistemas que esperam que os exemplos fornecidos sejam favoráveis à indução de uma definição recursiva (os chamados bons exemplos [63]). Por outro lado estão os sistemas que sintetizam uma pequena classe de programas lógicos, o que lhes permite o emprego de estratégias específicas na procura de definições recursivas ([1, 12, 49]. O sistema SKILit, que apresentámos neste Capítulo é capaz de sintetizar definições recursivas a partir de exemplos que colocam dificuldades a outros sistemas.

O sistema SKILit é uma extensão do sistema SKIL e utiliza a estratégia de indução iterativa para sintetizar definições recursivas a partir de conjuntos de exemplos escolhidos sem conhecimento prévio dos resultados pretendidos.

5.2 Indução de cláusulas recursivas

A possibilidade de definir conceitos recursivamente de forma concisa e elegante é uma das características mais atraentes da programação lógica. A recursividade, no entanto, é também fonte de muitos problemas práticos e teóricos. Na programação lógica por indução é bem conhecido o problema da indução de definições recursivas a partir de conjuntos de exemplos que não sejam construídos a pensar precisamente nesse fim. Neste Capítulo analisámos este problema detalhadamente, e descrevemos a nossa contribuição para o abordar, por via da *indução iterativa*.

Os sistemas existentes que fazem indução de cláusulas recursivas a partir de exemplos de forma não interactiva (sem oráculo) podem-se dividir em dois grupos de acordo com a abordagem adoptada. O primeiro grupo engloba abordagens em que os exemplos positivos não têm influência no espaço de procura das cláusulas o qual é exaustivamente explorado, mas apenas no critério de paragem da procura (WiM [95], FORCE2 [12]). Estes métodos podem ser vistos como variações sobre o conhecido método da força bruta. São também designados por métodos guiados pelo modelo. Esta abordagem tem a vantagem de ser mais robusta em relação a variações no conjunto de exemplos inicial, mas a desvantagem de não tirar partido dessas variações para acelerar a procura.

O segundo grupo inclui sistemas que geram as cláusulas pretendidas a partir dos exemplos positivos e, em alguns casos, do conhecimento de fundo (SKIL e [1, 80, 82, 96]). Nestes sistemas os exemplos servem para tomar decisões com base em heurísticas reduzindo-se o espaço de procura inicial. Por essa razão, são sistemas menos robustos em relação a variações no conjunto de exemplos positivos quando comparados com os

métodos de força bruta. A principal vantagem é a eficiência. Estes métodos são por vezes chamados *métodos guiados pelos dados* (*data-driven* [1]).

Para todos estes métodos guiados pelos dados é importante considerar um modelo *M* do conjunto dos exemplos E^+ e do conhecimento de fundo *Bf* (isto é, o conjunto de factos que se podem inferir a partir de $E^+ \cup Bf$). Dado um método de síntese de programas lógicos, e em particular de síntese de programas recursivos, que exemplos positivos lhe devem ser dados para que consiga sintetizar um determinado programa *P*? Iremos ver a seguir algumas características de conjuntos de exemplos positivos.

5.2.1 Conjuntos de exemplos completos/esparsos

O sistema FOIL [12] consegue sintetizar a definição de *member/2* se lhe forem dados como exemplos todos os factos relativos a este predicado envolvendo a uma lista (por exemplo *[1,2,3]*) e todas as suas sub-estruturas. Todos estes exemplos facilitam a tarefa de seleccionar os literais mais indicados. Os resultados do FOIL degradam-se quando o conjunto de exemplos não é completo [97]. O FOIL precisa de todos estes exemplos porque utiliza uma heurística baseada no número de exemplos cobertos para seleccionar o melhor literal a acrescentar pelo operador de refinamento e o teste de cobertura é extensional. O sistema GOLEM tem as mesmas limitações.

Exemplo 5.1: A cláusula *member(A,[B|Y])←member(A,Y)* só cobre o exemplo positivo *member(2,[1,2,3])* se o exemplo *member(2,[2,3])* for também dado.♦

Informalmente, e seguindo o vocabulário de Quinlan, dizemos que um *conjunto de exemplos* positivos é *completo* em relação a um conjunto de cláusulas se todos os exemplos forem extensionalmente cobertos por alguma das cláusulas. Quando um conjunto de exemplos não é completo diz-se um *conjunto de exemplos esparso*.

Exemplo 5.2: Dado o programa que define o predicado *member/2*,

member(A,[A|B]).

member(A,[B|C])←
member(A,C).

Um conjunto de exemplos que inclua *member(2,[1,3,2])*, para ser completo, deve ter também *member(2,[3,2])* e *member(2,[2])*. ♦

O facto de FOIL e GOLEM necessitarem de um conjunto de exemplos completo para sintetizarem um determinado conjunto de cláusulas tornam difícil a síntese indutiva de programas recursivos uma vez que é pouco provável que numa situação realista o utilizador forneça conjuntos de exemplos desnecessariamente grandes. Os sistemas de PLI devem também ser capazes de lidar com conjuntos esparsos de exemplos.

5.2.2 Conjunto representativo básico (CRB)

C. Ling [63] usou a noção de *conjunto representativo básico* (CRB, *basic representative set*) para definir o que são *bons exemplos* para a indução de um programa lógico. Para alguns sistemas de PLI, um CRB é uma condição necessária para poderem sintetizar um programa. É o caso de todos os sistemas que fazem testes de cobertura extensional às cláusulas induzidas. Um programa *P* nunca poderá cobrir extensionalmente um conjunto de exemplos que não inclua um CRB de *P*. Esta limitação inclui sistemas como FOIL, GOLEM, Progol, entre outros. O sistema SKILit não necessita de um conjunto representativo básico para sintetizar um programa.

Um conjunto de exemplos positivos que é completo em relação a um conjunto de cláusulas contém pelo menos um conjunto representativo básico dessas cláusulas.

Definição 5.1: Um conjunto representativo básico de um programa *P* é qualquer conjunto *S* de átomos fechados obtidos a partir de uma instância fechada verdadeira (no modelo mínimo de *P*) de cada cláusula *C* de *P*.♦

Uma vez que cada cláusula de um programa lógico pode ter várias instâncias verdadeiras, pode também ter vários conjuntos representativos básicos.

Exemplo 5.3: Dado o programa que define o predicado *member/2*, (ver Exemplo 5.2) um conjunto representativo básico desse programa é

{*member(1,[1,2]), member(4,[2,3,4]), member(4,[3,4])*}

Que correspondem à seguinte instanciação verdadeira do programa:

member(1,[1,2]).
member(4,[2,3,4])←member(4,[3,4]).

Outro CRB do mesmo programa é

{*member(3,[2,3,4]), member(3,[3,4])*}

Este conjunto tem apenas dois exemplos pois *member(3,[3,4])* pertence à instanciação das duas cláusulas.

member(3,[3,4]).
member(3,[2,3,4])←member(3,[3,4]).

Se retirarmos qualquer exemplo a qualquer um dos dois CRB acima indicados, este deixa de ser um conjunto representativo básico do programa.♦

Definição 5.2: Seja *C* uma cláusula de um programa *P*, um conjunto representativo básico de *C* relativamente a *P*, denota-se *CRBC(C,P),* é um conjunto de átomos fechados obtidos a partir de uma instância fechada de *C* verdadeira no modelo mínimo de *P*. O exemplo que corresponde à instanciação da cabeça da cláusula *C* é um exemplo representativo de *C* relativamente a *P*. ♦

Um CRB de um programa pode incluir, por definição, exemplos de predicados diferentes. No entanto, por conveniência, sempre que nos referirmos a um CRB de um programa *P* que define um predicado *p/k* a sintetizar, iremos apenas considerar os exemplos do CRB relativos a *p/k*. Assumimos que os elementos do CRB relativos a outros predicaos são dados extensionalmente ou intensionalmente.

5.2.3 Caminho de resolução

A síntese indutiva pode também tirar partido do facto de os exemplos positivos estarem envolvidos numa mesma derivação de um ou vários exemplos representativos de cláusulas de um programa a sintetizar. Ao conjunto de átomos envolvidos na derivação de uma consequência lógica chama-se caminho (ou cadeia) de resolução.

Definição 5.3: Seja *e* um exemplo, *P* um programa, e $D = ((R_1,C_1,\theta_1), (R_2,C_2,\theta_2), \ldots (R_n,C_n,\theta_n), \square)$, onde $R_1 = \leftarrow e$ e $C_i \in P$, a derivação de *e* a partir de *P*, o *caminho de resolução* de *e* em relação a *P*, *CR(e,P)* é o conjunto de átomos

$$CR(e,P) = \bigcup_{i=1}^{n} átomos(R_i)\theta_1\theta_2\ldots\theta_n$$

em que *átomos*(R_i) representa o conjunto dos átomos de R_i.♦

O caminho de resolução de um exemplo *e* relativamente a um programa *P* corresponde ao conjunto de factos utilizados para provar *e* a partir de *P*. Os elementos de um conjunto representativo básico de uma cláusula *C* pertencem a um mesmo caminho de resolução. Sendo *e* um exemplo representativo da cláusula $C \in P$, e $D = ((\leftarrow e,C,\theta_1), \ldots, (R_n,C_n,\theta_n), \square)$, uma derivação de *e* a partir de *P*, o conjunto de literais em $C\theta_1\theta_2\ldots\theta_n$ é um *CBRC(C,P)*.

Exemplo 5.4: Consideremos o programa para *member/2* definido no Exemplo 5.2 e o exemplo *member(4,[3,2,4])*. Para provar este facto construimos uma derivação (Figura 5.1).

Esta derivação é representada simbolicamente, omitindo as substituições, por

D = ((←*member(4,[3,2,4])*,C2), (←*member(4,[2,4])*,C2), (←*member(4,[4])*,C1))

O caminho de resolução é agora obtido reunindo os átomos das resolventes da derivação..

CR(member(4,[3,2,4]),P) =
{*member(4,[3,2,4])*} ∪ {*member(4,[2,4])*} ∪ {*member(4,[4])*}

Assim, os exemplos no caminho de resolução de *member(4,[3,2,4])* são {*member(4,[3,2,4])*, *member(4,[2,4])*, *member(4,[4])*}.♦

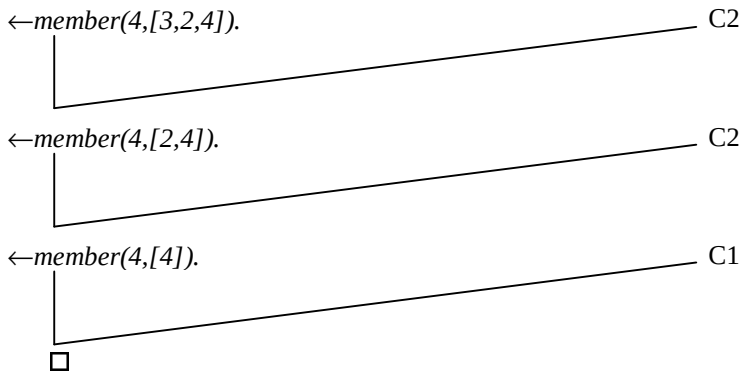

Figura 5.1: Derivação de um exemplo positivo.

Alguns algoritmos, como a *inversão de implicação* de Muggleton [77] ou o do sistema LOPSTER [60] (que emprega uma técnica chamada *sub-unificação*) não necessitam de um CRB para induzir uma cláusula recursiva. Basta-lhes um exemplo representativo dessa cláusula e um outro exemplo pertencente ao caminho de resolução relativo ao literal recursivo. Referimo-nos apenas a um literal recursivo pois o sistema LOPSTER só sintetiza cláusulas com um máximo de um literal recursivo. Na descrição do algoritmo de inversão de implicação essa limitação não é mencionada mas parece implícita.

Exemplo 5.5: Para induzir o programa no Exemplo 5.3 bastaria a um programa como o LOPSTER os exemplos *member(4,[3,2,4])* e *member(4,[4])*. Note-se que estes dois exemplos não constituem um CRB do programa.♦

Para o sistema CRUSTACEAN (um desenvolvimento do LOPSTER) os exemplos representativos de uma cláusula recursiva não precisam de pertencer ao mesmo caminho de resoluçao [1]. A técnica usada para descobrir a recursividade dos conceitos representados consiste na análise da estrutura dos termos que são argumentos dos exemplos.

Exemplo 5.6: Para induzir o programa no Exemplo 5.2 bastaria a um programa como o CRUSTACEAN os exemplos *member(4,[3,2,4])* e *member(1,[2,1])*. Note-se que o segundo exemplo não está no caminho de resolução do primeiro.♦

5.3 A indução iterativa

Como se comporta o SKIL em relação à indução de uma cláusula recursiva? Se lhe forem dados os exemplos positivos

member(2,[3,2]).
member(2,[2]).

o SKIL induz o programa

member(A,B)←dest(B,C,D),member(A,D).
member(A,B)←dest(B,A,C).

Este é um bom resultado, pois temos um programa recursivo que, juntamente com a definição de *dest/3* (Apêndice A), cobre os dois exemplos positivos dados. O segundo exemplo é representativo da cláusula base, e os dois são representativos da cláusula recursiva. Estes dois exemplos são um CRB do programa induzido e por consequência estão no mesmo caminho de resolução.

Se dermos ao SKIL um dos exemplos positivos acima, e um outro novo

member(2,[3,2]).
member(7,[7,1]).

o programa induzido agora será

member(A,B)←dest(B,C,D),dest(D,A,E).
member(A,B)←dest(B,A,C).

Perdemos a recursividade. No entanto, este programa induzido pelo SKIL não deixa de ser interessante. Embora o programa não seja recursivo, cada uma das suas cláusulas é uma *propriedade* do conceito *member/2*. A primeira propriedade, por exemplo, diz que todo o segundo elemento de uma lista é membro dessa lista. Estas propriedades que generalizam os exemplos inicialmente fornecidos ao sistema podem agora ser usadas para procurarmos cláusulas recursivas. Vamos ver como.

A razão porque o SKIL não encontra a cláusula recursiva a partir do segundo conjunto de exemplos é a seguinte. Para gerar a cláusula recursiva a partir do exemplo *member(2,[3,2])* o SKIL precisa de encontrar o esboço

member(2,[3,2])←dest([3,2],3,[2]),member(2,[2]).

Para tal é necessário que cada um dos átomos pertencentes ao esboço esteja no modelo de {*member(7,[7,1])*}∪*Bf*. Ora tal não é o caso. O átomo *member(2,[2])* não está no modelo e por essa razão a cláusula recursiva não aparece.

A única razão porque aquele átomo não está no modelo é porque ele não é um dos exemplos positivos inicialmente dados ao sistema. No entanto, após uma passagem do SKIL pelos exemplos positivos surgem as duas propriedades (chamemos-lhe *Props*), uma das quais cobre o próprio exemplo em falta (*member(2,[2])* ∈ *M*(*Bf*∪*Props*)). Por outras palavras, o exemplo crucial que não estava nos dados iniciais pode ser abduzido pelo próprio SKIL. Então agora o SKIL tem informação para gerar a cláusula recursiva. Efectivmente, a cláusula recursiva é gerada durante a segunda passagem pelos exemplos graças às propriedades geradas anteriormente.

Para tal deve acontecer uma segunda passagem pelos exemplos positivos, desta vez usando as propriedades para construir os refinamentos dos esboços.

Generalizando este processo obtemos um algoritmo iterativo que invoca o SKIL em cada iteração. A este método chamamos *indução iterativa*.

5.4 O algoritmo SKILit

O algoritmo SKILit (SKIL iterativo) constrói programas lógicos utilizando o método da indução iterativa. O SKIL é invocado pelo SKILit como um sub-algoritmo que percorre os exemplos positivos tentando construir novas cláusulas. O Algoritmo 5 descreve em detalhe este procedimento.

Procedimento SKILit
entrada: E^+, E^- (exemplos positivos e negativos)
BK (conhecimento de fundo).
saída: *P* (programa lógico)
$i := 0$
$P_0 := \varnothing$
repete
$P_{i+1} := SKIL(E^+, E^-, P_i, BK)$
$i := i+1$
até P_{i+1} não conter novas cláusulas em relação a P_i
$P := CT(P_{i+1}, BK, E^+, E^-)$
retorna *P*

Algoritmo 5: Indução iterativa

O SKILit começa com o programa P_0, o qual é inicialmente vazio. Na primeira iteração, SKILit cria o programa P_1. As cláusulas em P_1 generalizam alguns dos exemplos positivos e são tipicamente não recursivas. Em geral, é difícil introduzir a recursividade neste nível, devido à falta de exemplos positivos cruciais entre os exemplos dados. Assim, é provável que as cláusulas em P_1 sejam definidas apenas com predicados auxiliares (i.e., sem literais recursivos).

Numa segunda iteração, o programa P_2 é induzido. Aqui, é mais provável que a recursividade apareça, uma vez que P_1 cobre alguns exemplos cruciais que faltavam na

primeira iteração. De igual modo, como P_2 cobre mais factos, outras cláusulas recursivas interessantes podem aparecer nas iterações seguintes. O processo pára quando uma das iterações não introduz cláusulas novas. Após a última iteração é invocado o algoritmo CT (comprime teoria) que elimina cláusulas redundantes, tipicamente propriedades que foram induzidas em iterações iniciais e que foram posteriormente tornadas redundantes por cláusulas recursivas.

5.4.1 Bons exemplos

O método de indução iterativa sintetiza um programa P construindo uma sequência de programas P_0, P_1, …,P_n em que $P_0 = \varnothing$ e $P_n = P$. Cada P_i é obtido juntando a P_{i-1} uma ou mais cláusulas (com a excepção de P_n que é igual a P_{n-1}). Assim, e tratando-se de programas definidos, temos que

$$M(P_i \cup BK) \supseteq M(P_{i-1} \cup BK), \;\; 1 \leq i \leq n$$

Uma vez que o modelo de $P_i \cup BK$ cresce com i e, em cada iteração i a construção de cláusulas é condicionada pelo modelo de $P_{i-1} \cup BK \cup E^+$, a probabilidade de se sintetizar a cláusula recursiva pretendida numa iteração é maior ou igual do que nas iterações anteriores. Mas qual o conjunto de exemplos que deve ser dado inicialmente para que o nosso método de indução iterativa induza a cláusula pretendida? Como se caracteriza um conjunto de "bons exemplos"?

Como vimos na Secção 5.3, a indução iterativa não necessita de um conjunto representativo básico de exemplos para sintetizar uma cláusula recursiva. No entanto, o método precisa de todos os factos de um CRBC para sintetizar a cláusula, o que não é o mesmo que dizer que o conjunto de exemplos dado deve conter um CRBC. Vejamos então que exemplos devemos dar.

Comecemos por analisar a situação em que temos uma cláusula recursiva $C = (l_1 \leftarrow \ldots, l_2, \ldots)$ com um só literal recursivo l_2. Seja $\{e_1, e_2\}$ a parte de um CRBC relativa ao predicado *p/k* definido em C. Para sintetizar C, a indução iterativa precisa do exemplo

e_1 e de um exemplo $\hat{e}_2$ que serve de substituto a e_2. O exemplo $\hat{e}_2$ é representativo de uma cláusula não recursiva C_p que (juntamente com *BK*) cobre e_2 (o índice p é sugerido por C_p ser encarada como uma propriedade). Assim, um conjunto de bons exemplos para sintetizar C será $\{e_1, \hat{e}_2\}$. A indução iterativa sintetiza C_2 a partir de $\hat{e}_2$ numa iteração i e na iteração $i+1$ sintetiza C a partir de e_1 e C_p.

Exemplo 5.7: Consideremos o programa *P*

member(A,B)←dest(B,A,C). (C1)
member(A,B)←dest(B,C,D),member(A,D). (C2)
dest([A|B],A,B). (C3)

Um possível CRBC de C2 é { e_1 = *member(3,[1,2,3,4])*, e_2 = *member(3,[2,3,4])*}. Uma cláusula C_p não recursiva que cobre e_2 é

member(A,B)←dest(B,C,D),dest(D,A,E).

Vários exemplos cobertos por C_p podem servir como exemplo $\hat{e}_2$. Um deles é *member(5,[2,5])*. Temos então um conjunto de bons exemplos para a indução iterativa $\{e_1 = member(3,[1,2,3,4]), \hat{e}_2 = member(3,[2,5])\}$. ♦

Para cada exemplo e_2 existem várias cláusulas não recursivas que o cobrem. O próprio exemplo pode ser transformado numa cláusula unitária fechada. Para podermos caracterizar quais são os exemplos $\hat{e}_2$ aceitáveis dado um exemplo e_2 de um CRBC vamos indicar como construir a cláusula não recursiva C_p.

Dado um programa *P* e um exemplo e_2 coberto por esse programa podemos, aplicando a resolução às cláusulas de *P*, obter uma cláusula não recursiva C_p que cubra e_2. Seja *D* uma refutação $((\leftarrow e_2, C_1, \theta_1), (R_2, C_2, \theta_2), \ldots, (R_n, C_n, \theta_n), \square)$ de e_2 a partir de *P*. A cláusula C_p obtém-se transformando a cláusula C_1 segundo a sequência de passos de derivação de *D*, saltando aqueles que resolvem literais recursivos. Passamos a explicar.

Retiramos a *D* todos os passos de derivação que envolvem cláusulas que não definem o predicado *p/k*. Retiramos também o primeiro passo de derivação a *D*. No agora primeiro passo de derivação (R_j,C_j,θ_j), substituimos R_j por C_1. Resolvemos um literal negativo de C_1 com o literal positivo de C_j obtendo assim o passo de derivação (C_1,C_j,σ_j). Aplicando os restantes passos de *D* obtemos como resultado a cláusula C_p. Esta é uma cláusula não recursiva e que cobre e_2.

Exemplo 5.8: Continuando o Exemplo anterior vamos mostrar como se constrói a cláusula C_p a partir de *P*. A derivação *D*, omitindo as substituições, de e_2 é ((←*member(3,[2,3,4])*, C2), (←*dest([2,3,4],A,B),member(3,B)*, C3), (←*member(3,[3,4])*, C1), □) (ver Figura 5.2). Retirando à sequência de cláusulas na derivação a primeira cláusula (C2) e a cláusula C3 que não define *member/2*, resta-nos a cláusula C1. Resolvendo C2 com C1 obtemos a cláusula C_p não recursiva que cobre e_2.

member(A,B)←*dest(B,C,D),dest(D,A,E).*

♦

Agora coloca-se uma questão importante:

- Dado um qualquer exemplo $\hat{e}_2$ coberto por essa cláusula C_p a indução iterativa irá sempre obter uma cláusula que cubra e_2?

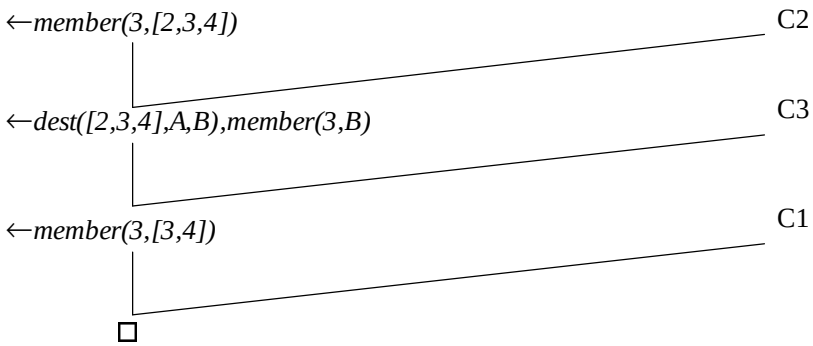

Figura 5.2: Derivação *D* do exemplo e_2.

Em geral, dado um exemplo $\hat{e}_2$ coberto por uma cláusula que cubra um outro exemplo e_2, o método de indução iterativa pode sintetizar uma cláusula $\hat{C}_p$ que não cobre e_2. (embora a prática nos diga que na maior parte dos casos a cláusula $\hat{C}_p$ cobre também e_2). Isto acontece porque o algoritmo que constrói os programas em cada iteração (SKIL, Algoritmo 1) usa uma *estratégia de cobertura*. Se um exemplo já está coberto o SKIL não procura construir outra cláusula alternativa que o cubra. Assim, a primeira cláusula encontrada é aquela que fica. O problema causado pela estratégia de cobertura sugere uma *estratégia iterativa pura* mais poderosa (embora mais pesada). Esta alternativa será descrita na Secção 5.4.2.

A análise que acabamos de fazer aplica-se a uma cláusula com um só literal recursivo. Caso a cláusula C tenha mais do que um literal recursivo será necessário um exemplo análogo a $\hat{e}_2$ para cada um desses literais. Uma vez que o CRBC de uma cláusula com k literais recursivos $C = (\ l_1 \leftarrow \ldots, l_2, \ldots, l_{k+1}, \ldots)$ contém $k+1$ exemplos $\{e_1, e_2, \ldots, e_{k+1}\}$, a indução iterativa precisa de um conjunto de exemplos $\{e_1, \hat{e}_2, \ldots, \hat{e}_{k+1}\}$. Cada um dos exemplos $\hat{e}_i$ representa uma cláusula C_i que cobre e_i , $2 \leq i \leq k+1$.

Para cada CRBC $\{e_1, e_2, \ldots, e_{k+1}\}$, de uma cláusula C de um programa P temos uma família de conjuntos de bons exemplos. Chamaremos a cada um desses conjuntos CRBCI (conjunto representativo básico de uma cláusula para a indução iterativa). Cada CRBCI $\{e_1, \hat{e}_2, \ldots, \hat{e}_{k+1}\}$ obtém-se do CRBC substituindo um ou vários exemplos e_i $2 \leq i \leq k+1$, por um exemplo $\hat{e}_i$ que seja coberto por uma cláusula não recursiva obtida por resolução a partir de P da forma descrita acima.

Note-se contudo que uma vez que o SKILit é iterativo, a propriedade C_p pode ela própria ser uma cláusula recursiva. Nesse caso o conjunto de bons exemplos para gerar a cláusula recursiva C deve incluir um conjunto de bons exemplos para gerar C_p.

Exemplo 5.9: Podemos sintetizar a definição recursiva de *member/2* a partir dos seguintes exemplos:

member(2,[1,3,2,4]).	*-member(2,[]).*
member(5,[5,6]).	*-member(2,[3]).*
member(6,[1,2,3,4,5,6]).	*-member(2,[1,4,3]).*
	-member(2,[1,4]).

Usando a GEC ´decomp_test_rec_comp_2' é possível ter na primeira iteração a cláusula não recursiva

member(A,[A|B]).

O seu exemplo representativo é *member(5,[5,6])*. Na segunda iteração o SKILit obtém a cláusula

member(A,[B,C|D])←member(A,D).

Esta é uma propriedade recursiva de *member/2* gerada a partir do exemplo *member(2,[1,3,2,4])* e da primeira cláusula. As duas cláusulas ainda não cobrem o exemplo *member(6,[1,2,3,4,5,6])*. A partir deste exemplo e da propriedade recursiva surge outra cláusula recursiva na terceira iteração:

member(A,[B|C])←member(A,C).

Os três exemplos positivos constituem um conjunto de bons exemplos para sintetizar esta cláusula.♦

Uma vez que o programa *P* não é conhecido antes de ser sintetizado, como podemos construir um CRBCI? Uma boa estratégia é fornecer uma série de exemplos positivos cujos termos de entrada variem em complexidade (caso sejam termos estruturados, tais como listas) ou em valor (no caso de estarmos a lidar com um domínio ordenado tal como inteiros) desde o termo mais simples (lista *[]*, ou inteiro *0*) terminando com termos razoavelmente complexos (listas de tamanho 4 ou menor, inteiros até *4*). Para cada nível de complexidade devemos dar exemplos que representem diferentes casos. Por exemplo *sort([1,2],[1,2])* e *sort([2,1],[1,2])* representam os dois casos possíveis para listas de

tamanho 2. No primeiro exemplo os dois elementos da lista de entrada mantêm as suas posições na lista de saída, e no segundo as suas posições são trocadas.

5.4.2 Estratégia iterativa pura

Como vimos acima, quando a estratégia de cobertura é utilizada não podemos garantir que o SKILit encontra uma dada cláusula C_p dado um exemplo $\hat{e}_2$ coberto por essa cláusula. Por essa razão apresentamos aqui uma nova estratégia iterativa.

Em cada iteração, o SKILit tenta construir uma vova cláusula para cada exemplo positivo quer este esteja coberto ou por cobrir. Note-se que com a estratégia de cobertura o SKILit não utiliza os exemplos cobertos para gerar novas cláusulas. O processo termina quando nenhuma cláusula nova é encontrada numa das iterações. A terminação é garantida se a linguagem clausal é finita, como normalmente é. Em qualquer dos casos pode ser tornada finita definindo uma gramática de estrutura clausal apropriada.

Chamamos a este procedimento a *estratégia iterativa pura.* Se fôr utilizado o método de variabilização completa cada exemplo pode dar, em cada iteração, um conjunto de cláusulas em vez de apenas uma. A estratégia de indução utilizada é escolhida através de uma declaração contida na especificação e corresponde a ligar e a desligar a condição de cobertura no Algoritmo 1 (contrutor de cláusulas).

Exemplo 5.10: Aqui mostramos como as estratégias de cobertura e iterativa pura podem ter diferentes resultados. A tarefa escolhida é de síntese múltipla dos predicados *sort/2* e *insert/3*. Por esse motivo a especificação contém informação relativa aos dois predicados (ver Secção 5.5.3). Neste exemplo damos o mesma especificação ao SKILit com cada uma das estratégias e comparamos os resultados.

Entrada:

sort([3,2,1],[1,2,3]). *environment(list).*

```
insert(2,[1],[1,2]).
insert(6,[],[6]).
sort([],[]).
insert(1,[2],[1,2]).
sort([5,4],[4,5]).

-insert(2,[1],[2,1]).
-insert(1,[2],[2,1]).
-insert(3,[1,2],[3,1,2]).
-insert(3,[1,2],[1,3,2]).
-sort([1,2],[2,1]).
-sort([1,3,2],[1,3,2]).
-sort([3,2,1],[2,3,1]).
-sort([3,2,4,1],[2,3,4,1]).
-sort([2,3,1],[2,3,1]).
```

```
csg( decomp_test_rec1_comp_2 ).
adm_predicates( sort/2,
        [dest/3,const/3,insert/3,sort/2,'<'/2,null/1]).
adm_predicates( insert/3,
        [dest/3,const/3,'<'/2,null/1,insert/3]).

% escolher estratégia conforme apropriado.
strategy( pure_iterative).
strategy(covering).
```

Saída, estratégia de cobertura:

```
sort([],[]).
sort([A,B|C],D)←insert(B,[A,B|C],E), insert(A,[B|C],D).

insert(A,[B],[B,A]) ←B<A.
insert(A,[],[A]).
insert(A,[B|C],[A,B|C]) ←A<B.

Number of iterations: 2
575 refinements (total)
4.64 secs
```

Saída, estratégia iterativa pura:

```
sort([],[]).
sort([A,B|C],D) ←insert(B,[A,B|C],E), insert(A,[B|C],D).
sort([A,B],C) ←insert(A,[B],C).
sort([A|B],C) ←sort(B,D), insert(A,D,C).

insert(A,[B],[B,A]) ← B<A.
insert(A,[],[A]).
insert(A,[B|C],[A,B|C]) ←A<B.
insert(A,[B|C],[B|D]) ←B<A, insert(A,C,D).
insert(A,[],[A]).
insert(A,[B],[A,B]) ← A<B.
```

insert(A,[B],[B|C]) ← B<A, insert(A,[],C).

Number of iterations: 5
2230 refinements (total)
35.37 secs

Estes resultados são produzidos sem o CT. Note-se que dados os exemplos positivos naquela ordem específica, a estratégia de cobertura consegue encontrar um programa não recursivo que cobre todos os exemplos excepto *sort([3,2,1],[1,2,3])* (não há nenhuma cláusula dentro do espaço de procura que cubra este exemplo). A estratégia iterativa pura encontra cláusulas alternativas, e entre elas algumas recursivas. O programa final (a negro) pode ser encontrado por um módulo de compressão como o CT. Note-se que as heurísticas de poda e a técnica de variabilização simples foram usadas neste exemplo. ♦

A seguir caracterizamos o conjunto de cláusulas que o SKILit consegue sintetizar usando a técnica de indução iterative pura.

Teorema 5.1: Seja *S* um esboço, *P* um programa, *G* uma gramática de estrutura clausal e *C* uma cláusula. Se $C\theta \angle S$ ($C\theta$ é uma consolidação de *S*) para alguma substituição θ, e o conjunto de cláusulas aceites por *G* é finito, então o SKILit com a estratégia iterativa pura, sem heurísticas de poda e com a variabilização completa produz *C* em tempo finito dados *S*, *P* e *G*.

Demonstração: Se $C\theta \angle S$ então, pela completude do operador de refinamento de esboços, $C\theta \in \rho^*(S)$. Uma vez que *G* aceita apenas um número finito de cláusulas então $\rho^*(S)$ é finito. Usando a estratégia iterativa pura o SKILit constrói todas as consolidações de *S* e eventualmente encontra o esboço $C\theta$. Uma das suas variabilizações é necessariamente *C*. Logo *C* é encontrada em tempo finito pelo SKILit com a estratégia iterativa pura.♦

Em consequência, dados os exemplos $\{e_1, \hat{e}_2\}$, e conhecimento de fundo *BK* se há uma cláusula não recursiva C_p que juntamente com *BK* cobre $\hat{e}_2$ então o SKILit gera essa cláusula. Se há uma cláusula recursiva *C* com um CRBC $\{e_1, e_2\}$ e $C_p \cup BK$ cobre e_2 então o SKILit gera *C* a partir de $\{e_1, \hat{e}_2\}$. Analogamente, podemos ter vários literais recursivos na cláusula *C* ou uma cadeia de propriedades recursivas como no Exemplo 5.9.

5.4.3 Arquitectura do SKILit

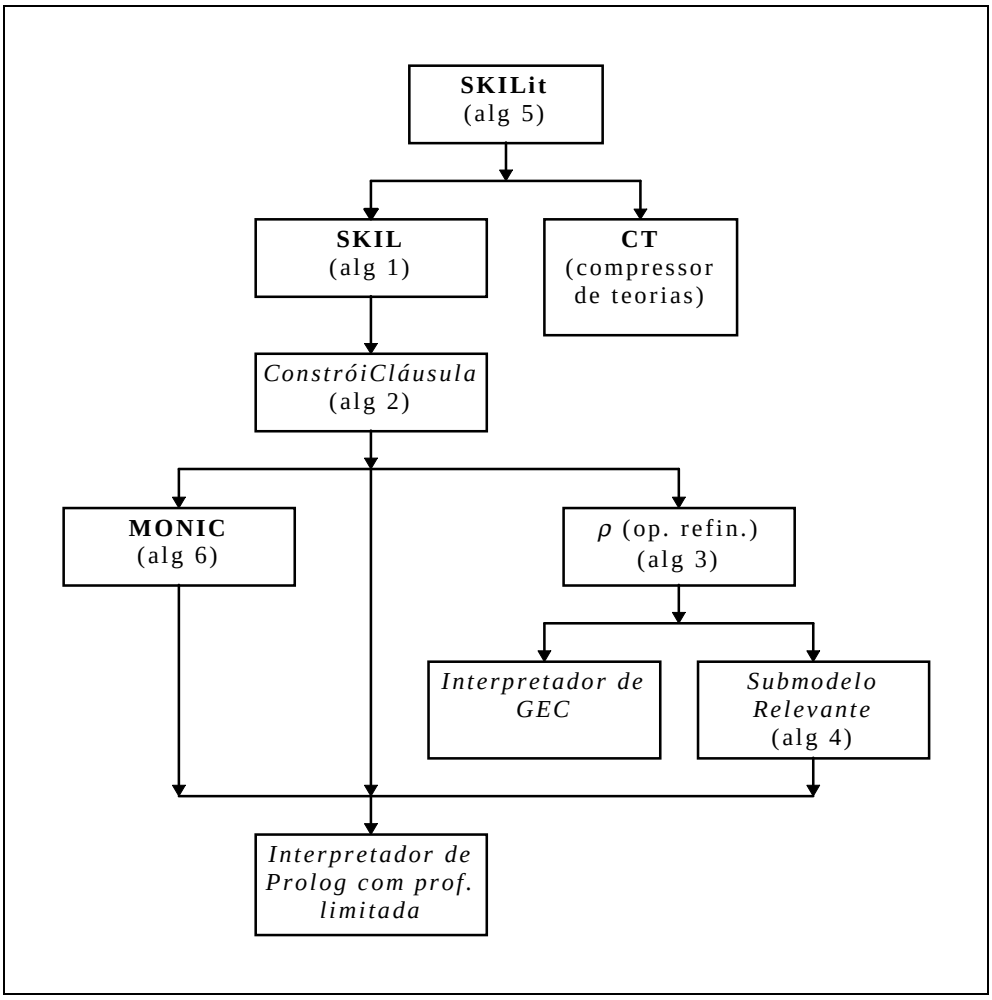

Figura 5.4: Arquitectura do sistema SKILit

O diagrama da Figura 5.4 mostra o relacionamento entre os principais módulos do sistema SKILit. Cada seta indica que o módulo na caixa de origem invoca o módulo na caixa apontada. O módulo no topo é o SKILit (Algoritmo 5), que chama iterativamente o sub-sistema SKIL (Algoritmo 1) e invoca também o compressor de teorias CT.

O módulo CT constitui o compressor de programas (ou compressor de teorias). O programa lógico obtido por indução iterativa pode conter cláusulas (propriedades) que, embora possam ter sido úteis ao processo de indução, não são necessárias no programa final. Algumas podem ser mesmo indesejáveis, causando não-terminação. De qualquer forma, é importante eliminar estas cláusulas por razões de eficiência e de legibilidade. Essa é a função do CT.

O CT recebe um programa induzido pelo método da indução iterativa e selecciona um subconjunto das suas cláusulas. Para tal usa uma estratégia de *selecção de sub-teorias* (sub-programas) úteis da teoria (programa) inicial. A partir da combinação desses sub-programas constrói um programa final que maximiza uma combinação de critérios definidos em termos de cobertura de exemplos positivos, compressão e comprimento da solução [15]. O compressor de teorias não será descrito em detalhe nesta tese.

5.5 Sessões

Aqui damos dois exemplos que mostram as capacidades do SKILit na síntese de predicados recursivos a partir de conjuntos esparsos de exemplos. O predicado *union/3* tem duas cláusulas recursivas interdependentes. O predicado *qsort/2* tem uma cláusula com dois literais recursivos. Mostramos também o emprego do SKILit numa tarefa de síntese múltipla de predicados. Nestas sessões, o SKILit utilizou a estratégia de cobertura, a variabilização simples e as heurísticas de poda.

5.5.1 Síntese de union/3

Mostramos um exemplo de síntese de uma definição do predicado *union/3* a partir de exemplos positivos e negativos e conhecimento de programação. Os exemplos dados são escolhidos seguindo a estratégia descrita anteriormente, variando a complexidade dos termos de entrada. Note-se, no entanto, que os exemplos positivos não são um conjunto representativo básico do programa, nem de um mesmo caminho de resolução.

Especificação:

```
mode( union(+,+,-) ).
type(union(list,list,list)).

union([],[2,3],[2,3]).
union([2],[2,3],[2,3]).
union([2],[3,4],[2,3,4]).
union([2,3],[4,2,5],[3,4,2,5]).
union([2,3],[4,5],[2,3,4,5]).

-union([3],[2],[3]).
-union([2],[3,4],[3,4]).
-union([2,3],[2],[2]).
-union([2,3],[4],[3,4]).
-union([2,3],[4],[2,4]).
-union([2],[1,2],[2,1,2]).
-union([1,2],[1,2],[1,1,2]).
```

Conhecimento de fundo e de programação:

```
background_knowledge(list).
adm_predicates( union/3,
      [dest/3,const/3,null/1,union/3,member/2,notmember/2]).
clause_structure(decomp_test_rec_comp_2).          % Apêndice C
```

Programa sintetizado (antes de eliminar as cláusulas redundantes)**:**

```
c(17):union([],A,A).
c(18):union([A],B,B)←                              % redundante
   member(A,B).
c(19):union([A],B,[A|B])←                          % redundante
   notmember(A,B).
c(20):union([A|B],C,D)←
   member(A,C),
   union(B,C,D).
c(21):union([A|B],C,[A|D])←
   notmember(A,C),
   union(B,C,D).
```

Número de iterações: 2
408 refinamentos (total)
18.22 secs

O programa sintetizado contém 3 cláusulas (c(17), c(20), c(21)), sendo duas delas recursivas. O módulo de compressão de teorias CT, elimina duas cláusulas redundantes (*c(18)* e *c(19)*). Estas são cláusulas intermédias que funcionam como propriedades e que são fundamentais na síntese das cláusulas recursivas. As cláusulas são apresentadas desachatadas, embora essa transformação seja feita pelo SKILit apenas para efeitos de apresentação do resultado.

O predicado *notmember/2* contorna o facto de o SKILit não prever a introdução de literais negados. Embora esta limitação do SKILit fosse fácil de eliminar, bastando para isso modificar o operador de refinamento, a procura poderia ficar mais pesada e os resultados poder-se-iam alterar.

5.5.2 Síntese de qsort/2

A síntese da definição do algoritmo de ordenação *quicksort* é um desafio clássico posto aos sistemas que sintetizam definições recursivas. Não queríamos por isso deixar de mostrar um resultado obtido pelo SKILit nessa tarefa. O que determina que o algoritmo de ordenação seja o *quicksort* e não outro qualquer são os predicados auxiliares admissíveis e a gramática de estrutura clausal.

Especificação:

```
mode( qsort(+,-) ).
type( qsort(list,list) ).

qsort([],[]).
qsort([3,1],[1,3]).
qsort([3,2,5,1,4],[1,2,3,4,5]).

-qsort([2,1],[2,1]).
-qsort([1,2],[2,1]).
-qsort([3,1,2],[1,3,2]).
-qsort([3,2,1],[2,1,3,1]).
-qsort([2,3,1],[2,3,1]).
-qsort([3,2,1],[2,1,3]).
```

Conhecimento de fundo e de programação:

```
background_knowledge( list ).
adm_predicates( qsort/2,
      [dest/3,const/3,part/4,append/3,qsort/2,'<'/2,null/1]).
clause_structure(decomp_test_rec_comp_2).


%parameters
max_num_of_refinement_nodes(2500).
```

Programa sintetizado(antes de eliminar as cláusulas redundantes)**:**

```
c(18):qsort([],[]).
c(19):qsort([A,B],[B,A])←                                  % redundante
   B<A.
c(20):qsort([A|B],C)←
   part(A,B,D,E),
   qsort(D,F),
   qsort(E,G),
   append(F,[A|G],C).

Número de iterações: 2
2780 refinamentos (total)
386.415 secs
```

Mais uma vez é sintetizada uma propriedade intermédia (cláusula c(19)), que o módulo CT se encarrega de eliminar.

5.5.3 Síntese múltipla de predicados

Aqui mostramos como o SKILit pode sintetizar um programa de dois predicados. Os predicados a definir são *sort/2* e *insert/3*. A especificação inclui exemplos positivos e negativos dos dois predicados assim como declarações de tipo e de modo. A gramática de estrutura clausal é uma só, mas cada predicado tem uma lista diferente de predicados auxiliares. Estes estão todos definidos no conhecimento de fundo *list*.

Especificação:

```
mode( sort(+,-) ).
type( sort(list,list) ).

mode( insert(+,+,-) ).
type( insert(int,list,list) ).

sort([3,2,1],[1,2,3]).
insert(6,[],[6]).
sort([],[]).
insert(1,[2],[1,2]).
sort([5,4],[4,5]).
insert(2,[1],[1,2]).
-insert(2,[1],[2,1]).

-insert(1,[2],[2,1]).
-insert(3,[1,2],[3,1,2]).
-insert(3,[1,2],[1,3,2]).
-sort([1,2],[2,1]).
-sort([1,3,2],[1,3,2]).
-sort([3,2,1],[2,3,1]).
-sort([3,2,4,1],[2,3,4,
```

Conhecimento de fundo e de programação:

```
background_knowledge( list ).

clause_structure(decomp_test_rec1_comp_2 ).
adm_predicates( sort/2,[dest/3,const/3,insert/3,sort/2,'<'/2,null/1]).
adm_predicates( insert/3,[dest/3,const/3,'<'/2,null/1,insert/3]).
```

No rasto do SKILit podemos ver a ordem de geração das cláusulas. Na primeira iteração temos cláusulas base e propriedades interessantes de ambos os predicados. A cláusula recursiva do *insert/3* também aparece na primeira iteração. Na segunda iteração temos a cláusula recursiva de *sort/2*.

Rasto do SKILit:

Iteração #1

exemplo a cobrir: sort([3,2,1],[1,2,3])
fila vazia.

exemplo a cobrir: insert(6,[],[6])
cláusula c(26) gerada após 7 refinamentos:
insert(A,[],[A]).

exemplo a cobrir: sort([],[])
cláusula c(27) gerada após 2 refinamentos:
sort([],[]).

exemplo a cobrir: insert(1,[2],[1,2])
cláusula c(28) gerada após 47 refinamentos:
insert(A,[B|C],[A,B|C])←
A<B.

exemplo a cobrir: sort([5,4],[4,5])
cláusula c(29) gerada após 101 refinamentos:
sort([A,B],C)←
insert(B,[A],C).

exemplo a cobrir: insert(2,[1],[1,2])
cláusula c(30) gerada após 105 refinamentos:
insert(A,[B|C],[B|D])←
B<A,
insert(A,C,D).

Iteração #2

exemplo a cobrir: sort([3,2,1],[1,2,3])

cláusula c(31) gerada após 14 refinamentos:
sort([A|B],C)←
sort(B,D),
insert(A,D,C).

exemplo a cobrir: insert(6,[],[6])
exemplo coberto pela cláusula c(26)

exemplo a cobrir: sort([],[])

exemplo coberto pela cláusula c(27)

exemplo a cobrir: insert(1,[2],[1,2])
exemplo coberto pela cláusula c(28)

exemplo a cobrir: sort([5,4],[4,5])
exemplo coberto pela cláusula c(29)

exemplo a cobrir: insert(2,[1],[1,2])
exemplo coberto pela cláusula c(30)

Programa sintetizado:

```
c(27):sort([],[]).
c(29):sort([A,B],C)←                                    % redundante
   insert(B,[A],C).
c(31):sort([A|B],C) ←
   sort(B,D),
   insert(A,D,C).

c(26):insert(A,[],[A]).
c(28):insert(A,[B|C],[A,B|C]) ←
   A<B.
c(30):insert(A,[B|C],[B|D]) ←
   B<A,
   insert(A,C,D).
```

Número de iterações: 2
351 refinamentos (total)

Embora o SKILit seja capaz de realizar síntese múltipla de predicados, nós não avaliamos cuidadosamente a nossa metodologia nesse tipo de tarefa de síntese. Em particular, não foi feita uma avaliação empírica sistemática que permita quantificar os sucessos e as limitações da nossa abordagem na síntese múltipla de predicados. Tencionamos fazer esse trabalho no futuro.

5.6 Limitações

Nesta Secção descrevemos as principais limitações da abordagem de síntese indutiva.

5.6.1 Programas específicos

Os programas sintetizados pelo SKILit são por vezes mais específicos do que os que seriam dados por outros sistemas capazes de induzirem definições recursivas a partir de conjuntos esparsos de exemplos positivos, como é o caso de CRUSTACEAN.

Exemplo 5.11: Dados os exemplos positivos

member(2,[1,2,3]).
member(3,[5,4,3]).

O SKILit gera um programa equivalente a:

member(X,[Y,X|Z]).
member(X,[Y|Z])←member(X,Z).

Outros sistemas, tais como CRUSTACEAN, sintetizariam um programa mais geral.

member(X,[X|Z]).
member(X,[Y|Z])←member(X,Z).

O programa do SKILit não cobre o exemplo *member(2,[2,1])*, ao contrário do segundo programa. ♦

Esta característica do SKILit pode ser encarada como uma limitação face a outros sistemas, no entanto nada nos garante que o programa que o utilizador teria em mente era o mais geral e não o mais específico. Por outras palavras, esta característica do SKILit é por vezes uma limitação, mas outras vezes pode ser também uma vantagem. Uma indicação disso é o facto de que o SKILit compete bem com CRUSTACEAN, como podemos ver na Secção 6.4.1.

5.6.2 Divisão de variáveis

A procura em largura feita pelo SKILit durante a construção de uma cláusula é comportável devido ao facto de se terem tomado algumas opções que reduzem esse mesmo espaço de procura. É o caso do processo de passagem das constantes a variáveis, a que chamamos variabilização, ocorrido no Algoritmo 2. O objectivo da variabilização é, dada uma cláusula totalmente instanciada, encontrar uma ou mais cláusulas que tenham a primeira como instância.

O processo de variabilização simples presentemente utilizado no SKILit tem a vantagem de ser eficiente (ver Secção 4.7.1) e de dar como resultado da variabilização de um determinado esboço uma única cláusula, evitando o problema vulgarmente designado por *divisão de variáveis* (*variable splitting*) [102]. Este facto ajuda a controlar a ramificação (*branching*) da árvore de procura. A desvantagem deste processo é não ter em conta o facto de uma mesma constante poder corresponder a duas variáveis diferentes, o que pode impedir a síntese de alguma cláusula desejável. Como já foi referido, o processo de variabilização completa resolve este problema.

Exemplo 5.12: Segundo o processo empregue pelo SKILit, a variabilização da cláusula

$$p(a,z) \leftarrow q(a,c),t(a,c,z). \quad (1)$$

é obtido substituindo todas as ocorrências de uma constante por uma variável, correspondendo variáveis diferentes a diferentes constantes. O resultado é

$$p(A,Z) \leftarrow q(A,C),t(A,C,Z).$$

Este processo de variabilização é simples e eficiente e tem como resultado uma única cláusula. No entanto a cláusula

$$p(A,Z) \leftarrow q(B,C),t(A,C,Z).$$

tem também a cláusula (1) como instância. Um processo de variabilização cujo resultado fossem todas as cláusulas com (1) como instância daria uma extensa lista de cláusulas como

p(A,Z)←q(A,C),t(B,C,Z).
p(A,B)←q(C,D),t(E,F,G).

etc.♦

5.7 Trabalho relacionado

5.7.1 Aprendizagem em circuito fechado

Michalski descreve em [68] a noção de *sistema em circuito fechado* (*closed-loop learning system*) como um sistema de aprendizagem que é capaz de utilizar os conceitos aprendidos numa fase de aprendizagem numa em fases subsequentes. Se os conceitos aprendidos não forem aproveitados internamente pelo sistema designa-se por *sistema em circuito aberto*. Michalski faz notar que ao contrário dos sistemas de aprendizagem humanos, os sistemas de aprendizagem automática são tipicamente em circuito aberto. A indução iterativa utiliza esta filosofia do sistema em circuito fechado para aprender cláusulas recursivas. As cláusulas aprendidas em iterações iniciais são empregues pelo processo de aprendizagem em iterações seguintes.

Uma das três estratégias de indução que o sistema MIS de Shapiro [109] pode usar é a *estratégia adaptativa*. Nesse caso, o MIS trabalha em sistema de circuito fechado, usando as próprias cláusulas induzidas para auxiliar a indução de novas cláusulas, tal como acontece na indução iterativa. Todavia, no MIS, a procura das cláusulas é exaustiva, enquanto no SKILit esta procura é guiada pelos exemplos através da estratégia de consolidação de esboços. Por outro lado, o SKILit pode partir para a procura de uma cláusula começando num qualquer esboço, enquanto o MIS parte sempre da cláusula vazia.

Os sistemas CHILLIN [125] e RTL [40] usam também estratégias iterativas para a indução de cláusulas recursivas. Todavia, parecem existir diferenças significativas entre estas abordagens e o método de indução iterativa que propomos.

O sistema CHILLIN intercala uma fase de generalização de cláusulas com uma fase de especialização. As duas fases são repetidas até não ser possível compactar mais o programa construído. Na fase de generalização utiliza o operador de generalização menos geral (*least general generalization*). Na fase de especialização faz uma procura descendente guiada por uma heurística semelhante à do FOIL [96]. Embora esta heurística possa funcionar bem com conjuntos de exemplos relativamente grandes, não parece adequada à síntese de definições recursivas a partir de pequenos conjuntos de exemplos.

O sistema RTL usa um método iterativo para a definição de definições recursivas. No primeiro passo o sistema produz definições não recursivas, as quais são subsequentemente transformadas em recursivas. O SKILit procede de forma análoga pois frequentemente começa também por produzir definições não recursivas em primeiro lugar. No entanto, é difícil prever que resultados o RTL obteria com pequenos conjuntos de exemplos positivos, uma vez que em [40] não são indicados resultados experimentais nesse sentido. Contudo, cremos que o RTL não se comportaria de forma muito positiva devido ao facto de também empregar uma heurística semelhante à do FOIL.

Outras abordagens à síntese de cláusulas recursivas a partir de conjuntos de exemplos positivos pequenos e esparsos foram já referidas no início deste Capítulo (Secção 5.2).

5.7.2 Conjuntos esparsos de exemplos

Já aqui referimos abordagens à síntese de cláusulas recursivas a partir de conjuntos esparsos e pequenos de exemplos positivos (Secção 5.2). Sistemas como FORCE2 [12], LOPSTER [60], e CRUSTACEAN [1] são vocacionados para esse problema específico.

Embora eficientes, estes sistemas induzem uma classe de programas muito restrita. CRUSTACEAN, por exemplo, induz programas da forma

$p(\ldots)$.
$p(\ldots) \leftarrow p(\ldots)$.

e não permite a utilização de conhecimento de fundo. A classe de programas sintetizável por FORCE2 é descrita por

$p(\ldots) \leftarrow q_1(\ldots), \ldots q_n(\ldots)$.
$p(\ldots) \leftarrow r_1(\ldots), \ldots, r_m(\ldots), p(\ldots)$.

Cada predicado q_i e r_j é um predicado do conhecimento de fundo. Um aspecto negativo importante do FORCE2 é o de que o utilizador tem de indicar ao sistema quais os exemplos que vão ser cobertos pela cláusula base e quais os que não vão ser.

Tal como foi descrito, o SKILit pode induzir programas com um variável de cláusulas recursivas e não recursivas e com um número variável de literais recursivos. Outra característica importante do SKILit que não é partilhada por aquelas abordagens é que o seu resultado não é necessariamente um programa recursivo (a menos que isso seja imposta pela gramática de estrutura clausal). Uma solução não recursiva é construída sempre que for apropriado. As soluções recursivas aparecem apenas se envolverem cláusulas mais curtas do que as soluções recursivas.

Os sistemas TIM [49] e SMART [74] apresentam também abordagens ao problema de aprender cláusulas recursivas a partir de conjuntos esparsos de exemplos. Todavia, estes sistemas foram apresentados simultaneamente com o SKILit [53].

O sistema SMART de Mofizur et al. é capaz de induzir teorias consistindo numa cláusula base e numa cláusula recursiva. Enquanto a cláusula base é induzida usando um processo de decomposição de termos semelhante ao do CRUSTACEAN, a cláusula recursiva é construída segundo uma estratégia descendente (do geral para o específico) de forma semelhante ao MIS [109]. O sistema restringe a procura examinando dependências entre

as variáveis. O sistema é capaz de aprender definições de vários predicados que processam listas a partir de pequenos conjuntos de exemplos. A classe de programas sintetizáveis é, contudo, mais restrita do que no caso do SKILit.

O sistema TIM, em vez de procurar regularidades dentro dos termos dos exemplos como faz o CRUSTACEAN, constrói explicações dos exemplos em termos dos predicados do conhecimento de fundo. Estas explicações são designadas por *saturações*. Depois de construir saturações para todos os exemplos positivos, TIM procura regularidades em pares de saturações e usa essas regularidades para construir a cláusula recursiva. A procura de sequências de literais comuns é dispendiosa, mas pode levar a resultados bastante bons. Comparando resultado experimentais do TIM e do SKILit que foram realizados em condições semelhantes (mas não necessariamente as mesmas), concluímos que não há um vencedor claro. De qualquer das formas alguns programas estão simplesmente fora do alcance do TIM. Este constrói definições sintacticamente semelhantes às de FORCE2

Note-se que tanto TIM como SMART assumem que a solução é um programa recursivo, ao contrário de SKILit.

5.8 Sumário

O sistema SKILit é uma extensão do sistema SKIL também aqui apresentado. O SKILit usa uma estratégia de indução iterativa que lhe permite sintetizar definições recursivas a partir de conjuntos esparsos de exemplos positivos.

A indução iterativa consiste em invocar o SKIL repetidamente utilizando as cláusulas produzidas numa iteração como entrada para as iterações subsequentes. Nas primeiras iterações surgem cláusulas tipicamente não-recursivas que generalizam os exemplos positivos. Estas cláusulas são designadas por propriedades e vão servir para ajudar a introduzir literais recursivos nas iterações seguintes.

A indução iterativa ultrapassa o problema da indução de cláusulas recursivas a partir de conjuntos esparsos de exemplos positivos. Caracterizamos os conjuntos de bons exemplos para a síntese de cláusulas recursivas usando indução iterativa e descrevemos duas estratégias alternativas: a estratégia de cobertura e a estratégia iterativa pura. Mostramos quais as cláusulas que são produzidas pelo SKILit usando a estratégia iterativa pura.

6. Avaliação Empírica

Apresentamos uma avaliação empírica do sistema SKILit. Descrevemos a metodologia de avaliação e mostramos os resultados de algumas experiências. Experiências comparativas do SKILit com outros sistemas são também apresentadas.

Este Capítulo sumaria as experiências realizadas de forma a obter uma avaliação empírica do sistema SKILit. O objectivo da avaliação é o de confirmar experimentalmente as vantagens e desvantagens da nossa metodologia. Em particular, estamos interessados em validar a adequação do sistema à síntese de programas lógicos recursivos a partir de conjuntos esparsos de exemplos positivos para alguns programas lógicos escolhidos para teste. A metodologia experimental descrita aqui tenta simular um utilizador humano de um sistema de síntese que não conheça os programas pretendidos de antemão.

As questões que queremos ver respondidas são as seguintes:

- Qual é o desempenho do sistema SKILit na síntese de definições (recursivas) a partir de conjuntos esparsos de exemplos positivos?

- Como se compara o sistema SKILit com outros sistemas estado-da-arte em PLI?

Para responder a estas perguntas utilizamos uma metodologia experimental que descrevemos na Secção seguinte. Até há pouco tempo atrás, os novos sistemas de PLI propostos não eram testados sistematicamente segundo uma metodologia experimental, sendo apenas demonstrados sobre alguns exemplos escolhidos que põem em evidência as virtudes ou fraquezas dos sistemas [20,38,109]. Isso é, obviamente, insuficiente. Recentemente, alguns trabalhos que abordam o problema da indução de definições recursivas a partir de conjuntos esparsos de exemplos utilizam uma metodologia de avaliação sistemática, a qual permite avaliar a robustez dos sistemas propostos face a variações na escolha dos exemplos positivos e negativos [1, 49, 125].

6.1 Metodologia de experimentação

Decidimos adaptar uma estratégia de avaliação que é comum em aprendizagem automática às necessidades da PLI. Cada experiência consiste em fazer correr o sistema de síntese indutiva (por exemplo, o SKILit) sobre um conjunto de exemplos positivos e negativos, designado por *conjunto de treino*, e avaliar o programa lógico produzido sobre um outro conjunto de exemplos positivos e negativos, designado por *conjunto de teste*. A avaliação sobre o conjunto de teste é feita essencialmente com base no número de exemplos positivos e negativos cobertos pelo programa induzido (Figura 6.1).

Para avaliar a robustez do sistema de síntese face à escolha dos exemplos de treino, é feita uma série de experiências (10 ou 20 repetições). Em cada uma delas o conjunto de treino é construído aleatoriamente a partir de um universo de exemplos e de um univeros de exemplos negativos definidos a priori. O conjunto de treino é dado como uma especificação ao SKILit. O programa resultante é então avaliado num conjunto de teste. Cada conjunto de teste é também construido aleatoriamente a partir de universos de exemplos positivos e negativos. Enquanto que, para cada experiência, se constrói um

novo conjunto de treino, o conjunto de teste é o mesmo para toda a série de experiências para o mesmo predicado.

O *universo de exemplos positivos* de uma dada relação é um sub-conjunto dos elementos dessa relação. A probabilidade de cada um dos exemplos ser extraído é também estabelecida. O *universo de exemplos negativos* contém elementos que não pertencem à relação.

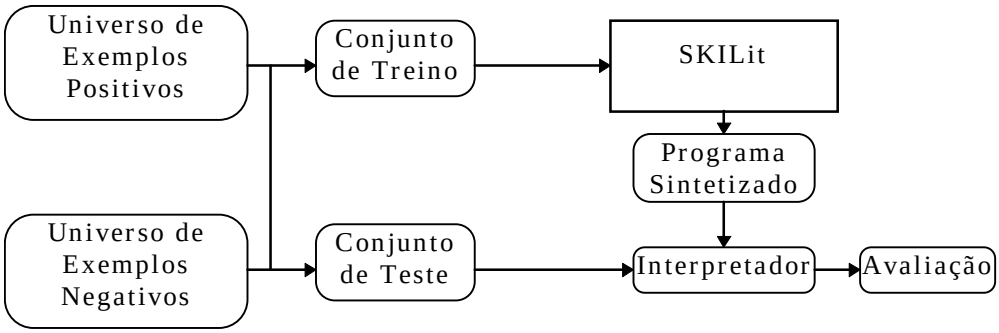

Figura 6.1: Metodologia de experimentação

A avaliação do desempenho do sistema numa experiência consiste em calcular o grau de acerto do programa induzido sobre o conjunto de teste, medindo o tempo de CPU[6] gasto pelo sistema durante a indução. Para cada série de experiências é também medida a percentagem de programas que tem grau de acerto igual a 1. Esta medida é designada pela percentagem de programas totalistas.

Outras medições foram feitas para cada experiência, tais como contar o número de cláusulas em cada programa. Contudo esses resultados não são aqui relatados.

[6] Unidade Central de Processamento (Central Processor Unit).

6.1.1 O grau de acerto, programas totalistas e tempo de CPU.

O **grau de acerto** (*success rate*) de um programa lógico P sobre um conjunto de teste CT com $\#E^-$ exemplos negativos e $\#E^+$ exemplos positivos é

$$ga(P,CT) = \frac{cob(P,E^+)+(\#E^- - cob(P,E^-))}{\#E^+ + \#E^-}$$

em que $cob(P,E^*)$ é o número de exemplos positivos ou negativos cobertos por P. O teste de cobertura é intensional, e é feito com um interpretador de profundidade limitada (Secção 4.7.4).

Depois de medir o grau de acerto de cada programa induzido P_1, P_2,…, P_n, sobre um conjunto de teste TS numa série de n experiências, podemos calcular a *percentagem de programas totalistas*.

$$ppt(ProgSint,TS) = \frac{\#(\{P \mid ga(P,CT)=1,\ P \in ProgSint\})}{n} \times 100\%$$

em que $ProgSint = \{ P_1, P_2,\ldots, P_n \}$, e # corresponde à cardinalidade de um conjunto.

A percentagem de programas totalistas estima a probabilidade de obtermos um programa correcto numa sessão do SKILit. Por outro lado, é interessante consideramos a percentagem de programas totalistas, e não apenas ao grau de acerto. É diferente o sistema sintetizar um programa aceitável em 90% das ocasiões ou sintetizar sempre um programa com um grau de acerto de 0.90. Na nossa opinião a primeira situação é preferível quando se trata de síntese de programas.

O tempo de CPU gasto para cada experiência foi medido num computador SUN com um processador SPARC10. O SKILit foi implementado com o compilador de Prolog Yap [2].

6.1.2 O universo de exemplos positivos

A distribuição de exemplos positivos $p(X_1,...,X_k)$ de uma relação *p/k* num universo de exemplos positivos é definida a partir das distribuições dos tipos de cada um dos argumentos X_i. Um tipo que corresponde a um conjunto de termos não estruturados, tal como *int* (0,1,2,...,9), tem uma distribuição uniforme. Um tipo estruturado, tal como *list* tem uma distribuição uniforme na sua dimensão até um certo limite. No caso de uma listas, a sua dimensão é o seu comprimento. As listas com um comprimento superior ao limite não são consideradas. O comprimento da lista *[]* é 0 e o comprimento de uma lista *[X|Y]* é 1+comprimento de *Y*. O universo de exemplos positivos envolvendo estruturas com dimensão menor ou igual a 4 é designado por U4(+). O universo de exemplos positivos envolvendo estruturas com dimensão maior ou igual a 3 e menor ou igual a 5 designa-se por U3:5(+). Para a escolha dos sub-termos de um termo estruturado, toma-se em consideração o tipo do sub-termo.

Podemos ter também exemplos positivos para os quais apenas os argumentos de entrada são limitados em dimensão. O universo U2i(+) é constituído por exemplos positivos em que a dimensão máxima dos argumentos é 2.

A extracção de um exemplo positivo $p(X_1,...,X_k)$ é feita extraindo cada termo X_i de tipo T_i segundo a distribuição de T_i. Evidentemente que os $p(X_1,...,X_k)$ que não pertencem à relação não são considerados. A tarefa de extracção de um exemplo positivo é facilitada tirando partido da declaração de modo do predicado *p/k*. Assim, extraem-se apenas os termos de entrada, sendo os de termos de saída determinados por estes. No caso da relação ser não determinística, os termos de saída devem ser escolhidos aleatoriamente entre as várias respostas possíveis.

6.1.3 O universo de exemplos negativos

Nas nossas experiências consideramos inicialmente dois tipos de exemplos negativos: aleatórios e 'near misses'. Seguidamente descrevemos os dois tipos de exemplos.

Um exemplo negativo $p(X_1,...,X_k)$ designado por aleatório é gerado de forma idêntica a um exemplo positivo, i.e., extraindo cada um dos termos X_i segundo a distribuição do seu tipo T_i e verificando que de facto se trata de um exemplo negativo da relação. O universo de exemplos negativos aleatórios envolvendo listas de comprimento igual ou menor do que 4 é designado por U4(-).

Os exemplos negativos 'near misses' são sintacticamente próximos dos exemplos positivos, mas sem pertencerem à relação. A extracção de um 'near miss' é feita corrompendo sintacticamente um exemplo positivo e verificando que o facto resultante não pertence à relação. As operações de corrupção aplicáveis a um exemplo positivo são definidas à priori. Uma lista, por exemplo, é corrompida apagando aleatoriamente um elemento, acrescentando um elemento, ou trocando a ordem de dois elementos consecutivos. A escolha da operação de transformação também é aleatória. O universo de exemplos negativos 'near miss' envolvendo listas de comprimento igual ou menor do que 4 é designado por Unm4(-). De forma análoga podemos ter Unm3:5(-), Unm2i(-), etc.

Os exemplos negativos aleatórios são mais simples de gerar do que os 'near misses', uma vez que requerem menos processamento. Na nossa opinião, todavia, os 'near misses' tendem a simular melhor o tipo de exemplos negativos que um utilizador real daria. Por esse motivo relatamos aqui apenas os resultados obtidos com exemplos negativos do tipo 'near miss'.

6.1.4 Os parâmetros do SKILit

Para cada experiência, é importante termos em conta o estado dos parâmetros do SKILit.

parâmetro	**valor por defeito**	**significado**
solver_depth	6	controla a profundidade do interpretador nos testes de cobertura

max_effort_limit	300	número máximo de refinamentos gerados durante a construção de uma cláusula.
dcg	decomp_test_rec_comp_2	a gramática de estrutura clausal usada para definir a circunscrição de linguagem.

Tabela 6.1: Parâmetros do SKILit.

Quando os valores dos parâmetros para uma experiência não são mencionados explicitamente consideram-se os valores por defeito.

6.1.5 Predicados utilizados na avaliação

Os predicados utilizados na avalição são alguns predicados de processamento de listas.

- *member(int,list)*: Este predicado é verdadeiro se o inteiro no primeiro argumento estiver contido na lista do segundo argumento.
- *last_of(int,list)*: O inteiro que é o último elemento da lista.
- *delete(int,list,list)*: A segunda lista é obtida a partir da primeira apagando a primeira ocorrência do inteiro no primeiro argumento. Se o inteiro não está na primeira lista, o predicado falha.
- *rv(list,list)*: A segunda lista tem os elementos da primeira em ordem inversa.
- *append(list,list,list)*: A terceira lista é obtida concatenando a primeira lista com a segunda.
- *split(list,list,list)*: A segunda lista contém os elementos que estão em posições ímpares na primeira lista. A terceira lista contém os elementos que estão em posição par.
- *union(list,list,list)*: Cada lista representa um conjunto que se assume não conter elementos repetidos. A terceira lista contém todos os elementos das duas primeiras, sem repetições.

Todos os predicados acima estão definidos no Apêndice A.

6.1.6 Resumo das experiências realizadas

Na primeira série de experiências o SKILit foi avaliado isoladamente. Os exemplos negativos usados foram do tipo 'near miss'. Os exemplos positivos usados para teste são mais complexos do que os usados para treino. A razão para isso é que exemplos de teste mais exigentes (U3:5(+)) reduzem a possibilidade de termos um programa menos bom (no sentido de que não seria aceite por um programador humano) atingir um grau de acerto alto. Esta opção foi motivada pelos resultados que obtivemos em algumas experiências preliminares que não são aqui descritas (vêr). Nessas experiências foram usados conjuntos de teste menos exigentes (U4(+)) e foi observado que alguns programas sintetizados pelo SKILit atingiram o grau de acerto máximo, apesar de serem claramente imperfeitos. Na Secção 6.3 descrevemos experiências realizadas com o predicado *union/3* e que permitem ilustar algumas limitações da metodologia de avaliação e da metodologia de síntese. Na Secção 6.4 damos resultados de experiências com os sistemas CRUSTACEAN e Progol. Na Secção 6.5 mostramos o resultado de outras experiências realizadas com o SKILit.

6.2 Resultados com o SKILit

Nesta primeira série de experiências avaliamos o desempenho do sistema SKILit em função do número de exemplos de treino positivos e negativos escolhidos aleatoriamente. Os exemplos positivos de treino foram retirados de universo U4(+) de cada relação, e os exemplos negativos do universo Unm4(-). Quanto aos conjuntos de teste, os exemplos positivos foram aleatoriamente extraidos do universo U3:5(+).

6.2.1 Grau de acerto

Na Figura 6.2 podemos observar a curva de aprendizagem do grau de acerto médio obtido pelo SKILit para cada um dos seis predicados considerados. Para cada predicado,

são apresentadas quatro curvas. Uma para 0 exemplos negativos, as outras para 5, 20 e 100. Cada curva mostra a média do grau de acerto obtida em 10 repetições para 2,3,5,10 e 20 exemplos positivos.

Para cinco dos seis predicados considerados nesta experiência, o SKILit conseguiu atingir um grau de acerto igual a 1 com 20 exemplos positivos e 100 exemplos negativos de treino. Uma excepção foi o predicado *append/3* que ficou por um nível máximo de 0.85.

Alguns predicados mais simples (*delete/3*, *last_of/2*, *member/2*, *split/3*) conseguiram atingir o máximo de acerto com 10 exemplos positivos e 5 negativos. O sistema consegue bons resultados, em termos de acerto, mesmo perante a ausência de exemplos negativos. Isso deve-se fundamentalmente ao seguinte factor. As diversas circunscrições do SKILit (gramática de estrutura clausal, conhecimento de fundo, parâmetros, estratégias de construção de cláusulas, etc) são suficientes para eliminar muitas definições que cobrem exemplos negativos no conjunto de teste. Foi o que aconteceu, por exemplo, na indução do predicado *member/2*, que obteve resultados excelentes com 10 exemplos positivos e 0 negativos. Para quase todos os predicados observamos nestas experiências pouca variação no grau de acerto relativamente ao número de exemplos negativos de treino.

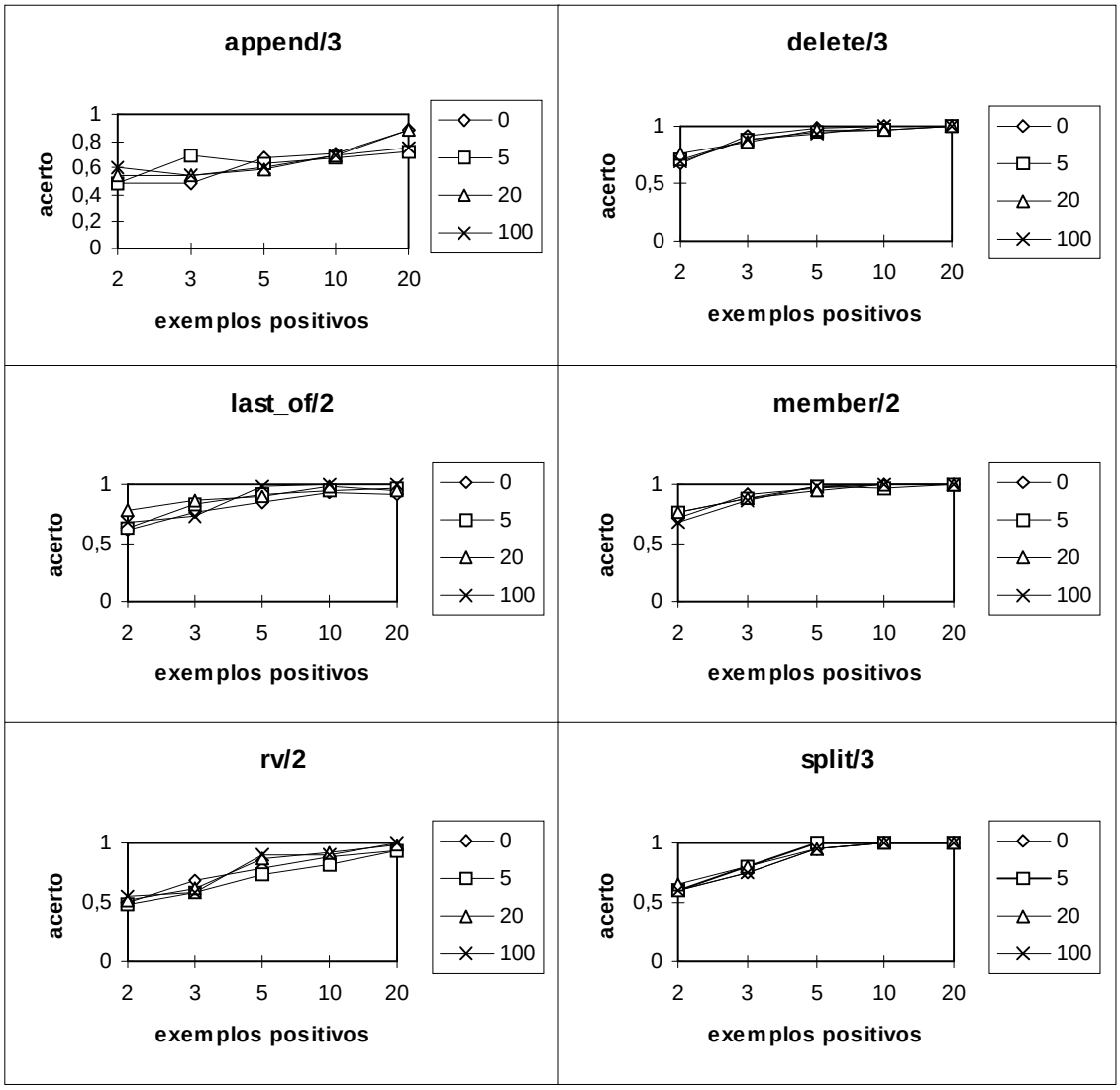

Figura 6.2: Grau de acerto face ao número de exemplos de treino.

6.2.2 Percentagem de programas totalistas

Na Figura 6.3 mostramos as curvas de aprendizagem para a percentagem de programas totalistas.

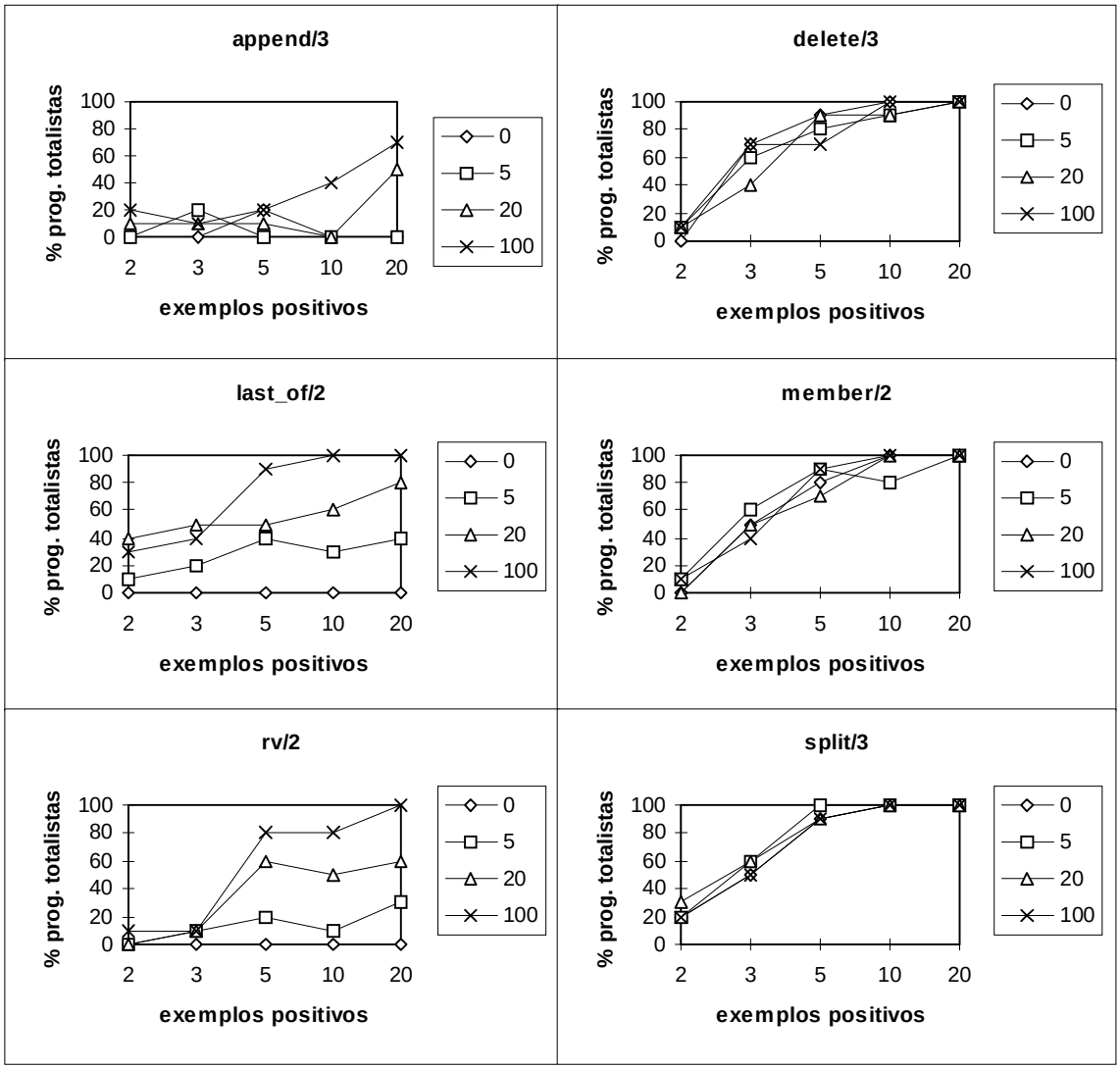

Figura 6.3: Percentagem de programas totalistas face ao número de exemplos de treino.

Ao contrário do que acontece com o grau de acerto, a percentagem de programas totalistas varia mais perceptivelmente com o número de exemplos negativos. Isto acontece, por exemplo, nos predicados *append/3*, *last_of/2* e *rv/2*. Em particular, estes predicados obtêm 0% de programas totalistas quando lhes são fornecidos 0 exemplos negativos. Mais uma vez, o sistema converge rapidamente para os 100% quando o número de exemplos aumenta. A excepção é ainda o predicado *append/3*.

6.2.3 Tempo de CPU

O tempo médio de CPU gasto nas diversas experiências é mostrado na Figura 6.4.

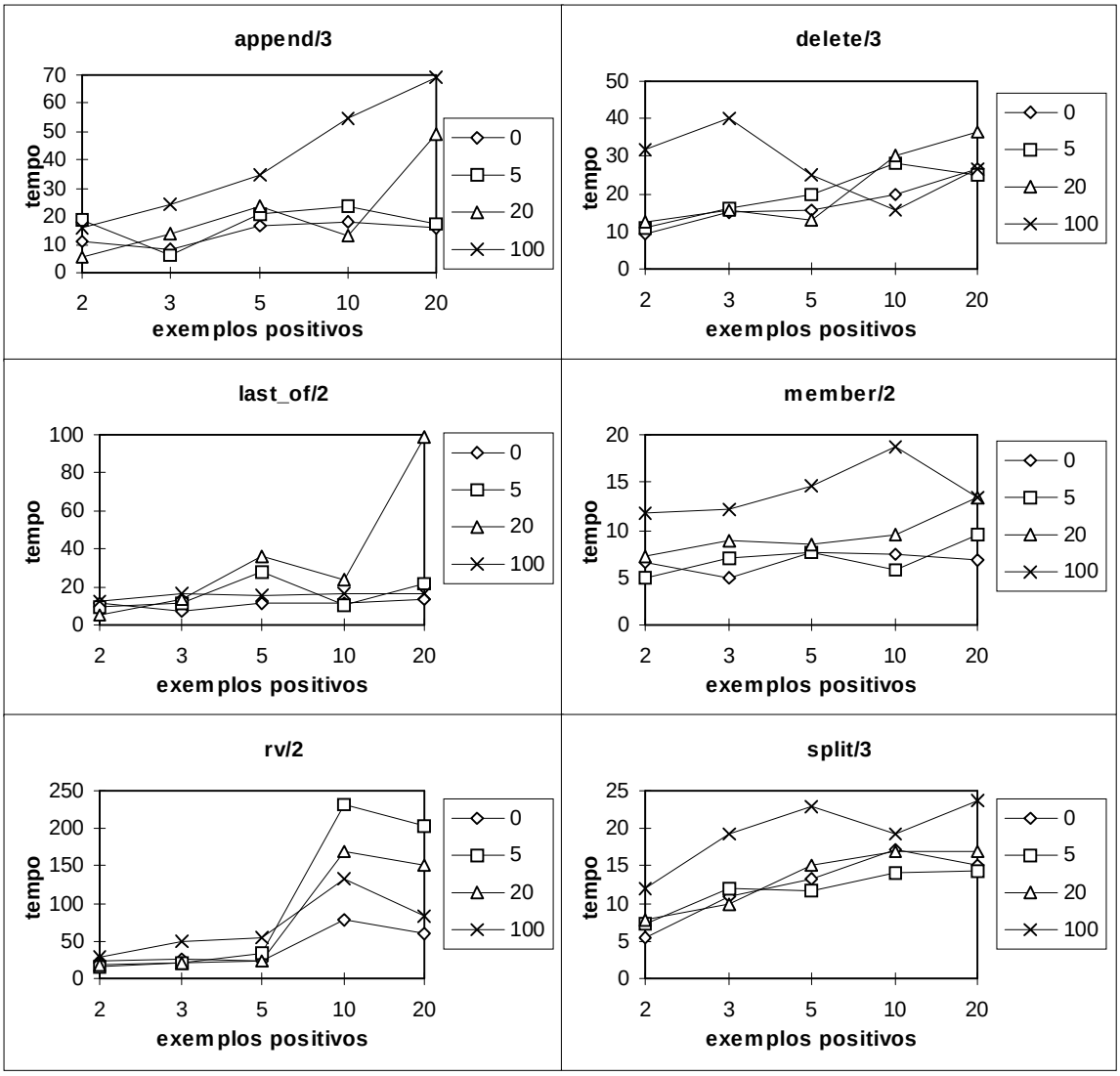

Figura 6.4: Tempo de CPU (segundos).

Podemos observar alguma irregularidade nas curvas de tempo gasto em função do número de exemplos. No entanto, o SKILit mostra um bom comportamento com o aumento do número de exemplos de treino, positivos ou negativos. Na maior parte dos casos, o tempo de CPU não aumenta dramaticamente com o número de exemplos de

treino. O tempo de CPU gasto pelo SKILit parece ser influenciado mais pela qualidade dos exemplos positivos, do que pela sua quantidade.

6.3 Experiências com union/3

Outros predicados foram utilizados para avaliação sistemática seguindo a metodologia descrita acima. Para alguns desses predicados (*factorial/2*, *extNth/2*, *noneiszero/1*), os resultados foram semelhantes aos obtidos para os seis predicados usados na Secção anterior.

Todavia, para predicados mais complexos, tais como *union/3*, *quicksort/2* (*qsort/2*), *insert/3* e *partition/4*, a nossa metodologia de avaliação tem mais dificuldade em gerar aleatoriamente bons exemplos positivos e/ou bons exemplos negativos. Estes predicados têm por vezes mais do que uma cláusula base ou mais do que uma cláusula recursiva, cada uma delas envolvendo um predicado de teste em particular. Outros têm mais do que um literal recursivo.

Isto não significa necessariamente que a nossa metodologia de síntese precisa, para esses predicados, de exemplos cuidadosamente escolhidos. Também pode ser o caso de que simular um utilizador humano através da geração aleatória de exemplos é mais difícil para predicados mais complexos. Por exemplo, muitos dos exemplos negativos gerados aleatoriamente para o predicado *union/3* tendem a ser variantes de poucos casos diferentes. Por outro lado, é improvável que alguns exemplos negativos importantes sejam gerados aleatoriamente. Devemos notar que o processo de geração aleatoria dos exemplos foi o mesmo para todos os predicados avaliados (de *member/2* a *partition/4*).

Descreveremos agora algumas experiências realizadas coma síntese do predicado *union/3* que ajudam a descrever algumas destas dificuldades.

Corremos o SKILit sobre conjuntos de 30 exemplos positivos e 100 exemplos negativos, todos gerados aleatoriamente. Os exemplos positivos foram retirados do

universo U2i(+), e os exemplos negativos do universo Unm2i(-). Os exemplos de teste foram retirados de U3:5(+) e Unm2i(-). O número de repetições por experiência foi 50.

Com conjuntos de treino construídos aleatoriamente, o SKILit não sintetizou um programa totalista em 50 execuções (Tabela 6.2). Podemos melhorar os resultados mudando a gramática de estrutura clausal de forma a que cada cláusula seja forçada a ter pelo menos um literal de teste (a gramática anterior permitia cláusulas sem literais de teste). A nova GEC chama-se decomp_+test_rec_comp_2. Em 50 execuções, o SKILit encontrou 3 programas totalistas.

Positivos	*Negativos*	*GEC*	*acerto*	*totalistas*	*tempo*
aleatórios	'near misses'	decomp_test_rec_comp_2	0.532	0	570.484
aleatórios	'near misses'	decomp_+test_rec_comp_2	0.585	6	543.441
aleatórios	9 escolhidos	decomp_test_rec_comp_2	0.753	34	204.748
aleatórios	9 escolhidos	decomp_+test_rec_comp_2	0.740	34	130.772
5 escolhid.	'near misses'	decomp_test_rec_comp_2	0.794	58	63.106
5 escolhid.	'near misses'	decomp_+test_rec_comp_2	0.821	64	50.838

Tabela 6.2: Resultados experimentais para *union/3*.

Serão estes maus resultados devidos à falta de bons exemplos positivos ou à falta de bons exemplos negativos? Para responder a esta questão, corremos de novo o SKILit com 50 conjuntos de 30 exemplos positivos aleatórios. Desta vez porém foram escolhidos manualmente 9 exemplos negativos. Os resultdos melhoraram claramente independentemente da gramática utilizada. Todavia, para a gramática decomp_+test_rec_comp_2, o tempo de CPU foi consideravelmente menor (Tabela 6.2). Os exemplos negativos escolhidos manualmente encontram-se na Tabela 6.3.

Escolhemos então 5 exemplos positivos (Tabela 6.3) e corremos o SKILit com 50 conjuntos de 100 exemplos negativos aleatorios. Os resultados obtidos são bastante bons com a gramática habitual e ainda melhoram se utilizarmos a gramática que impõe os literais de teste. Podemos concluir que é provável (34%) encontrar bons conjuntos de exemplos aleatorios positivos quando os exemplos negativos são escolhidos manualmente. É também provável (58%, 64%) encontrar bons conjuntos de exemplos

negativos aleatórios quando os positivos são escolhidos. Contudo, encontrar simultaneamente dois bons conjuntos aleatoriamente tem uma probabilidade baixa (0%, 6%) (Tabela 6.2).

Exemplos positivos escolhidos	*Exemplos negativos escolhidos*
union([],[2,3],[2,3]). *union([2],[2,3],[2,3]).* *union([2],[3,4],[2,3,4]).* *union([2,3],[4,2,5],[3,4,2,5]).* *union([2,3],[4,5],[2,3,4,5]).*	*-union([2],[1,2],[2,1,2]).* *-union([2],[3,4],[3,4]).* *-union([3],[2],[3]).* *-union([2,3],[2],[2]).* *-union([2,3],[4],[3,4]).* *-union([2,3],[4],[2,4]).* *-union([2],[2],[2,2]).* *-union([2,1],[2],[2,1,2]).* *-union([1,2],[1,2],[1,1,2]).*

Tabela 6.3: Exemplos escolhidos, positivos e negativos, utilizados nas experiências.

Uma direcção possível é dar ao utilizador meios mais poderosos de transmitir exemplos negativos ao sistema. Isto motivou o nosso trabalho com restrições de integridade, apresentado no Capítulo seguinte.

6.4 Comparação com outros sistemas

Aqui concentramo-nos na comparação do SKILit com os sistemas CRUSTACEAN e Progol que consideramos representativos do estado-da-arte. No entanto, outros trabalhos já aqui descritos são também relevantes.

6.4.1 CRUSTACEAN

Uma comparação entre o sistema SKILit e sistema CRUSTACEAN, feita sobre alguns predicados foi já publicada em [53] e está resumida na Tabela 6.4 e é descrita graficamente na Figura 6.5. Os valores mostrados para o sistema CRUSTACEAN foram retirados de [1]. Os valores do sistema SKILit foram obtidos em experiências realizadas por nós em condições tanto quanto possível idênticas às descritas por Aha et al. Por esse motivo, devemos considerar estes valores apenas como indicativos. De qualquer das

formas, perante um número bastante reduzido de exemplos positivos e negativos, o sistema SKILit obtém resultados de acerto no mínimo comparáveis ao sistema CRUSTACEAN. Tendo em conta que o SKILit utiliza uma circunscrição de linguagem muito mais fraca do que o CRUSTACEAN, este é um resultado importante (3.5.4).

Para cada predicado, fizemos variar o número de exemplos positivos entre 2 e 5, enquanto o número de exemplos negativos se manteve constante (=10). O exemplos positivos foram retirados do universo U4(+). Para os exemplos negativos usou-se o universo Unm4(+). Os resultados mostram médias obtidas a partir de 5 experiências.

	SKILit			*CRUSTACEAN*	
	número de exemplos positivos de treino				
	2	3	5	2	3
append/3	0.76	0.80	0.89	0.63	0.74
delete/3	0.75	0.88	1.00	0.62	0.71
rv/2	0.66	0.85	0.87	0.80	0.86
member/2	0.70	0.89	0.95	0.65	0.76
last_of/2	0.71	0.72	0.94	0.74	0.89

Tabela 6.4: Grau de acerto do SKILit face ao CRUSTACEAN

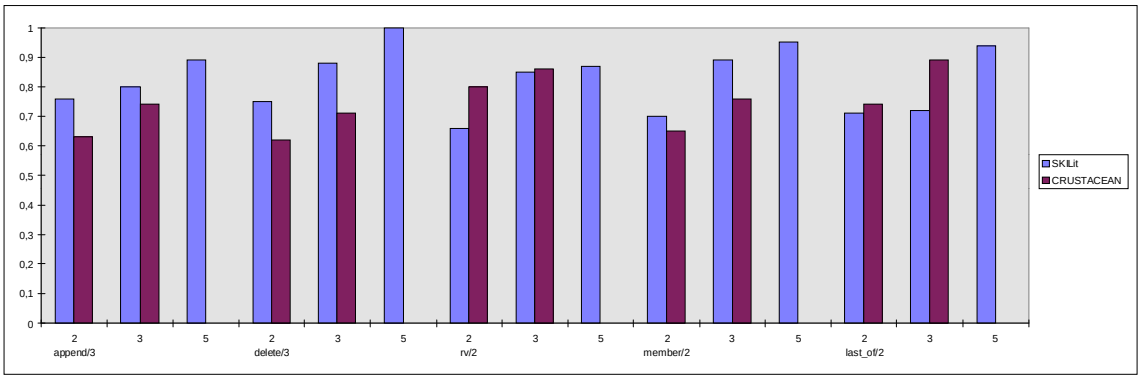

Figura 6.5: Graus de acerto de SKILit vs. CRUSTACEAN.

6.4.2 Progol

Para estabelecermos alguma comparação entre o SKILit e o sistema Progol [80] decidimos utilizar um dos ficheiros de entrada distribuídos com o próprio Progol. Esse

ficheiro contém 17 exemplos positivos e 8 exemplos negativos do predicado append/3. Com esses exemplos, a versão do Progol de que dispomos sintetiza uma definição do *append/3* perfeita. A experiência de comparação com o SKILit consistiu em fazer correr ambos os sistemas sobre 20 subconjuntos dos exemplos positivos. Estes subconjuntos foram construídos aleatoriamente e fornecidos a cada um dos sistemas juntamente com a totalidade dos exemplos negativos. Os resultados estão representados na Tabela 6.5 e graficamente na Figura 6.6.

	Acerto		Desvio padrão		% Totalistas	
	SKILit	*Progol*	*SKILit*	*Progol*	*SKILit*	*Progol*
3	0.625	0.500	0.217	0.000	25	0
5	0.750	0.525	0.250	0.109	50	5
7	0.800	0.699	0.245	0.244	60	35
9	0.900	0.949	0.200	0.150	80	85
11	0.975	0.945	0.109	0.149	95	60
13	0.850	0.996	0.229	0.006	70	70
15	0.975	0.998	0.109	0.006	95	90
17	1.000	0.998	0.000	0.006	100	90

Tabela 6.5: Comparação entre SKILit e Progol. Predicado *append/3*.
(A primeira coluna mostra o número de exemplos positivos)

Comparando as percentagens de programas totalistas obtidas por cada um dos sistemas, observamos uma clara superioridade do SKILit. Tendo em conta que estamos num contexto de síntese de programas, este é um resultado bastante positivo. No que diz respeito ao grau de acerto obtido o SKILit parece atingir melhores resultados para conjuntos de exemplos muito pequenos (3 a 7 exemplos). Os valores de desvio padrão fornecidos permitem uma comparação mais rigorosa.

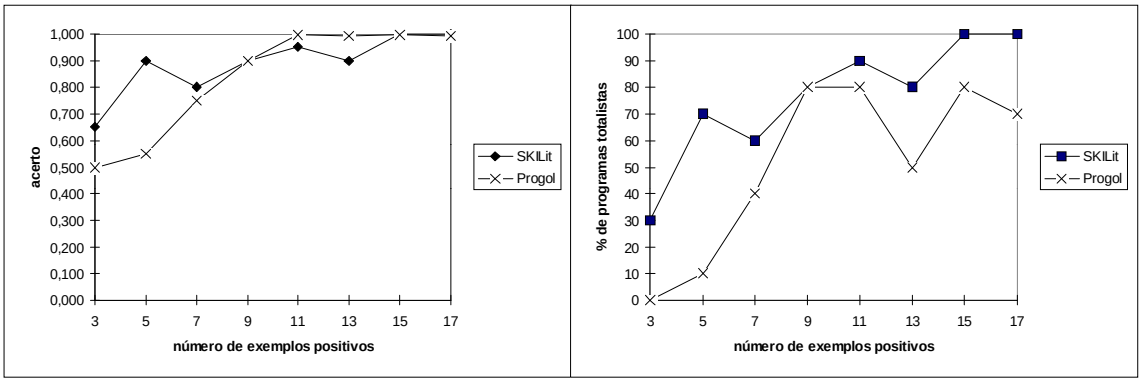

Figura 6.6: Comparação entre SKILit e Progol para o predicado *append/3*.

6.5 Outras experiências

Nesta Secção mostramos algumas experiências isoladas realizadas com o SKILit utilizando exexmplos seleccionados manualmente. Estas experiências servem para ilustrar a classe de programas que o SKILit é capaz de induzir.

Note-se que nestas experiências foi usada a gramática de estrutura clausal decomp_test_rec_comp_2. O uso de outras gramáticas é explicitamente indicado. Aquela gramática é a mesma que foi utilizada noutras experiências relatadas nesta tese.

O conhecimento de fundo é ou 'integer' ou 'list' dependendo dos predicados auxiliares admissíveis. Infelizmentes, uma lista de predicados admissíveis tem de ser dada ao SKILit. Esta é uma limitação importante que deve ser investigada no futuro.

6.5.1 Factorial

O predicado *factorial(X,Y)* é verdadeiro se $Y = X!$.

Entrada:

mode(factorial(+,-)).
factorial(2,2).
factorial(3,6).
-factorial(3,3).
-factorial(4,12).

factorial(4,24).

background_knowledge(integer).
adm_predicates(factorial/2,[succ/2,pred/2,zero/1,multb/3,factorial/2]).
clause_structure(decomp_test_rec_comp_2).

Saída:

factorial(A,A)←pred(A,B),pred(B,C),zero(C).
factorial(A,B)←pred(A,C),factorial(C,D),multb(D,A,B).

Número de iterações: 3
507 refinamentos (total)

6.5.2 Multiply

O predicado *multiply(A,B,C)* instancia *C* com *A*B*.

Entrada:

mode(multiply(+,+,-)).
type(multiply(int,int,int)).

multiply(0,4,0).
multiply(1,5,5).
multiply(2,3,6).
multiply(3,4,12).

-multiply(2,2,2).
-multiply(3,3,6).
-multiply(2,2,5).
-multiply(2,2,6).
-multiply(2,2,8).
-multiply(2,3,4).

background_knowledge(integer).
adm_predicates(multiply/3,[pred/2,succ/2,zero/1,plus/3,one/1,multiply/3]).
clause_structure(decomp_test_rec_comp_2).

Saída:

multiply(A,B,A)←zero(A).
multiply(A,B,C)←pred(A,D), multiply(D,B,E), plus(E,B,C).

Número de iterações: 3
795 refinamentos (total)

Propriedades geradas (eliminadas por CT):

multiply(1,A,A).

6.5.3 Insert

O predicado *insert(I,L1,L2)* introduz um inteiro *I* numa lista ordenada *L1* obtendo *L2* deforma a que *L2* seja ordenada.

Entrada:

mode(insert(+,+,-)).
type(insert(int,list,list)).

insert(2,[],[2]).
insert(1,[2],[1,2]).
insert(2,[1],[1,2]).
insert(3,[1,5],[1,3,5]).

-insert(2,[1],[2,1]).
-insert(1,[2],[2,1]).
-insert(3,[1,2],[3,1,2]).
-insert(3,[1,2],[1,3,2]).

background_knowledge(list).
adm_predicates(insert/3,[dest/3,const/3,'</2,null/1,insert/3]).
clause_structure(decomp_test_rec_comp_2).

Saída:

insert(A,[],[A]).
insert(A,[B|C],[A,B|C])←A<B.
insert(A,[B|C],D)←insert(A,C,E),insert(B,E,D).

Esta definição de *insert/3* parece correcta do ponto de vista lógico. Computacionalmente tem o seguinte problema. Interpretando o programa e pondo a pergunta

←insert(8,[1,3,5,7],R).

obtemos uma série de respostas idênticas

R = [1,3,5,7,8] ;
R = [1,3,5,7,8] ;
R = [1,3,5,7,8] ;
R = [1,3,5,7,8] ;

R = [1,3,5,7,8] ;
R = [1,3,5,7,8]
....

Este tipo de problema computacional poderia ser facilmente detectado limitando o número de respostas idênticas obtidas a partir dos exemplos de treino. Contudo, este filtro não está correntemente implementado no SKILit.

Se alterarmos a GEC de forma a que a recursividade dupla não seja permitida (decomp_test_rec1_comp_2, Apêndice C) obtemos a seguinte definição para *insert/3*:

insert(A,[],[A]).
insert(A,[B|C],[A,B|C])←A<B.
insert(A,[B|C],[B|D])←B<A,insert(A,C,D).

6.5.4 Partition

O predicado *partition(I,L,S,G)* divide a lista *L* em duas listas *S* e *G*, de forma a que *S* os elementos de *L* menores ou iguais a *I*, e *G* contenha os elementos de *L* maiores do que *I*.

Entrada:

mode(partition(+,+,-,-)).
mode(partition(int,list,list,list)).

partition(2,[],[],[]).
partition(4,[2,6],[2],[6]).
partition(3,[1,2,5],[1,2],[5]).
partition(3,[6,2,5],[2],[6,5]).

-partition(2,[2],[2],[2]).
-partition(2,[1,3,4],[1,4],[3,4]).
-partition(2,[3,1],[3],[1]).
-partition(2,[1,0],[1],[0]).
-partition(2,[4,5],[4],[5]).
-partition(4,[1,2],[1],[2]).
-partition(2,[3],[3],[]).
-partition(2,[1],[],[1]).

background_knowledge(list).
adm_predicates(partition/4,[dest/3,const/3,null/1,'<'/2,partition/4]).
clause_structure(decomp_test_rec_comp_2).

max_num_of_refinement_nodes (1000).

Saída:

partition(A,[],[],[]).
partition(A,[B|C],[B|D],E)←B<A, partition(A,C,D,E).
partition(A,[B|C],D,[B|E])←A<B,partition(A,C,D,E).

Propriedades geradas:

partition(A,[B,C],[B],[C])←B<A,A<C.

6.5.5 Insertion sort

O predicado *isort(A,B)* ordena a lista *A*. A lista *B* é a lista ordenada. Os predicados admissíveis escolhidos (*insertb/3*) determinam a estratégia de ordenação.

Entrada:

mode(isort(+,-)).
type(isort(list,list)).

isort([],[]).
isort([2,1],[1,2]).
isort([3,2,1],[1,2,3]).

-isort([1,2],[2,1]).
-isort([1,3,2],[1,3,2]).
-isort([3,2,1],[2,3,1]).
-isort([3,2,4,1],[2,3,4,1]).

background_knowledge(list).
adm_predicates(isort/2,[dest/3,const/3,insertb/3,isort/2,'<'/2,null/1]).
clause_structure(decomp_test_rec_comp_2).

Saída:

isort([],[]).
isort([A|B],C)←isort(B,D),insertb(A,D,C).

Número de iterações: 3
193 refinamentos (total)

Propriedades geradas:

isort([A,B],C)←insertb(A,[B],C).

6.6 Trabalho relacionado respeitante a avaliação

Alguns sistemas de PLI têm sido avaliados empiricamente e de forma sistemática pelos seus autores. A seguir descrevemos o trabalho mais relevante respeitante a avaliação.

Em 1993 Quinlan apresentou o seu "*Midterm Report*" onde avaliava o sistema FOIL [97] usando uma metodologia experimental que ele designou como "mais pragmática" do que a usada habitualmente em sistemas de aprendizagem. Quinlan criticou em particular o facto de muitos sistemas serem avaliados usando um conhecimento de fundo muito limitado, assim como um conjunto de exemplos de treino cuidadosamente escolhido. A sua metodologia de avaliação do FOIL consiste em fazer o sistema sintetizar uma série de predicados de manipulação de listas do livro de Bratko "*Prolog Programming for Artificial Intelligence*" [10]. O conhecimento de fundo é inicialmente vazio e vai acumulando os predicados à medida que aprende.

Quanto aos exemplos de treino, Quinlan considera o conjunto das listas com comprimento menor ou igual a três, e com elementos do conjunto {1,2,3}, a que chama universo U3,e um outro conjunto, universo U4, com listas de comprimento menor ou igual a quatro com elementos em {1,2,3,4}. Para cada relação e para cada universo construiu o conjunto de exemplos positivos que envolvem as listas do respectivo universo e pertencem à relação. Os que não pertencem à relação constituem o conjunto de exemplos negativos. Todos os exemplos positivos são dados para treino, assim como todos os exemplos negativos até ao limite de 90 000. Os resultados obtidos são satisfatórios para a maior parte dos predicados embora com limitações para alguns deles.

Sobre o trabalho de Quinlan destacamos o seguinte. Apesar dos conjuntos de treino não serem cuidadosamente escolhidos, eles são completos para cada universo (U3, U4) considerado. Este facto facilita a indução de definições recursivas. Num enquadramento de síntese indutiva em que os exemplos são fornecidos manualmente pelo utilizador, não podemos esperar que os conjuntos de treino sejam completos. Em lugar disso tendem a ser pequenos e esparsos.

Em 1994 Aha et al. [1] avaliam o sistema CRUSTACEAN usando conjuntos de treino construídos aleatoriamente, de forma a demonstrarem que o sistema consegue sintetizar definições recursivas a partir de conjuntos pequenos (e esparsos) de exemplos. Para cada predicado, foi medido o desempenho (grau de acerto) de CRUSTACEAN com conjuntos de treino de 2 e 3 exemplos positivos e 10 exemplos negativos todos escolhidos aleatoriamente. Os valores de desempenho foram obtidos a partir da média de 10 execuções para cada número diferente de exemplos positivos. Os predicados envolvidos foram *append/3*, *delete/3*, *extractNth/3*, *factorial/2*, *last_of/2*, *member/2* ao , *noneIsZero/1*, *plus/3*, *reverse/2* e *split/3*.

A nossa metodologia de experimentação baseia-se, em boa medida, neste trabalho de Aha et al. Lá encontramos a preocupação de submeter um sistema de indução a situações em que os conjuntos de treino não são completos, mas sim esparsos. A principal limitação desta metodologia é a de, a nosso ver, dificilmente se estender a sistemas com uma circunscrição de linguagem menos forte do que a utilizada pelo sistema CRUSTACEAN. Um sistema como o SKILit, por exemplo, que pode induzir uma grande variedade de cláusulas, necessita de exemplos negativos que possam eliminar os programas sobre-gerais, sendo pouco provável que tais exemplos negativos relevantes sejam gerados aleatoriamente na metodologia de Aha et al.

7. Restrições de Integridade

Apresentamos o verificador de integridade MONIC que utiliza uma estratégia de Monte Carlo para procurar inconsistências entre um programa e restrições de integridade. MONIC é eficiente, íntegro mas incompleto. MONIC é integrado no sistema SKILit e apresentamos alguns resultados experimentais na síntese a partir de exemplos positivos e restrições de integridade.

7.1 Introdução

Geralmente, os exemplos negativos aceites pelos sistemas de PLI são factos fechados (sem variáveis). No nosso entender, um facto fechado transmite muito pouca informação quando é dado como exemplo negativo. Daqui resulta que o número de exemplos negativos dados a um sistema tende a ser elevado. O facto de os sistemas de PLI vulgarmente necessitarem de um grande número de exemplos negativos é o principal problema abordado neste Capítulo.

Para ilustrar este problema damos dois exemplos referidos na literatura.

- Segundo o "*mid-term report*" de Quinlan [97] o sistema FOIL aprende o predicado *reverse/2* usando entre 1560 a 92796 exemplos negativos.

- Zelle et al. [125] referem que o sistema CHILLIN aprende o predicado *member/2* com uma precisão média de cerca de 50% dados mais de 80 exemplos negativos.

Estes factos restringem as aplicações dos sistemas de PLI, principalmente quando os exemplos são manualmente fornecidos pelo utilizador, como acontece na síntese de programas.

O problema do excesso de número negativos atraiu já alguma atenção da comunidade de PLI. Alguns sistemas como FORCE2 [12], LOPSTER [60] e CRUSTACEAN [1] usam uma circunscrição de linguagem muito forte, o que diminui o número de exemplos negativos necessários. Contudo, esses sistemas parecem difíceis de estender de forma a lidar com uma maior variedade de programas lógicos. O sistema FOIL [96] permite o uso de um *pressuposto de mundo fechado* (*closed world assumption*). Se um facto não é dado ao sistema explicitamente como um exemplo positivo então é considerado um exemplo negativo. Esta técnica não é prática em muitas situações de aprendizagem pois força o utilizador a fornecer um conjunto completo de exemplos positivos.

Uma alternativa prometedora baseia-se no uso de *restrições de integridade*. Estas são cláusulas da lógica de primeira ordem da forma $a \wedge ... \wedge b \rightarrow c \vee ... \vee d$ que podem ser usadas para transmitir ao sistema restrições a respeitar pelo predicado a sintetizar, da mesma forma que são usados os exemplos negativos. A diferença principal é que as restrições de integridade permitem uma representação mais compacta. Luc de Raedt sugeriu que um sistema de PLI podia usar restrições para verificar os programas gerados [21].

Apesar de as restrições de integridade não serem cláusulas normais, o *procedimento de prova SLDNF* pode ser usado para verificar se um programa lógico satisfaz uma restrição de integridade, transformando a restrição numa pergunta (*query*) e pondo essa pergunta ao programa lógico. Esta estratégia, contudo, sofre de sérios problemas de

eficiência, uma vez que para encontrar uma instância violante de uma restrição de integridade podemos ter que tentar todas as instanciações possíveis das suas variáveis. Existem outros métodos mais sofisticados de lidar com restrições de integridade tais como SATCHMO [65], mas parecem ainda demasiado pesados computacionalmente para serem de uso prático para a PLI.

Nós propomos um novo método para lidar com restrições de integridade que torna possível o uso de restrições de integridade em sistemas de PLI sem grandes custos de eficiência. Resultados experimentais mostram que, usando este método, conseguimos induzir programas lógicos recursivos eficientemente. Na verdade, observamos nas nossas experiências que, para um mesmo nível de precisão, o nosso sistema é mais rápido com restrições de integridade do que com exemplos negativos.

O nosso verificador de restrições de integridade (MONIC) usa uma estratégia de 'Monte Carlo'. Para verificar se um programa *P* satisfaz uma restrição de integridade *I*, MONIC gera aleatoriamente uma série de consequências lógicas de *P* e verifica se estas satisfazem *I*. Esta é uma forma muito eficiente de lidar com restrições, embora incompleta. Todavia, podemos controlar o nível de incompletude variando o número de consequências lógicas de *P* que são geradas.

7.2 O número de exemplos negativos

Como já foi referido no Capítulo 5 muitos sistemas de PLI requerem um número excessivo de exemplos positivos para induzirem definições de predicados, o que constitui um obstáculo ao uso dos sistemas de PLI, especialmente no contexto de síntese de programas. O nosso sistema SKILit lida precisamente com a falta de exemplos positivos cruciais gerando propriedades que depois reutiliza iterativamente. Estas são cláusulas que capturam regularidades nos exemplos positivos, generalizando-os, permitindo a introdução de cláusulas recursivas mais complexas.

E quanto aos exemplos negativos? Dar todos os exemplos negativos cruciais a um sistema de PLI pode ser uma tarefa monótona. Estes podem ser numerosos pois cada exemplo negativo fechado fornece pouca informação ao sistema. Para além disso, o utilizador não sabe quais os exemplos negativos mais apropriados para cada tarefa de síntese, tendo a tendência de dar ao sistema mais exemplos negativos do que os necessários.

7.3 Restrições de integridade

Da mesma forma que uma propriedade pode representar um conjunto de exemplos positivos, os exemplos negativos podem também ser substituídos, ou complementados, por cláusulas mais expressivas. Tais cláusulas são chamadas *restrições de integridade* (*integrity constraints*).

Exemplo 7.1: podemos exprimir que nenhum termo é membro da lista vazia através da restrição de integridade $member([],X) \rightarrow false$. ♦

Exemplo 7.2: A cláusula $sort(X,Y) \rightarrow sorted(Y)$, representa uma restrição de integridade que exprime a condição "o segundo argumento do predicado *sort/2* é uma lista ordenada". Da mesma forma, podemos dizer que a lista *Y* é uma permutação da lista *X* com $sort(X,Y) \rightarrow permutation(X,Y)$. ♦

As restrições de integridade, tal como os exemplos negativos, podem ser usadas por um sistema de PLI para detectar e rejeitar programas sobre-gerais. De facto, os exemplos negativos podem ser vistos como um caso particular das restrições de integridade. Por exemplo, $member([],2) \rightarrow false$ representa o exemplo negativo *member([],2)*. Uma restrição de integridade representa intensionalmente um conjunto de exemplos negativos, possivelmente infinito, pelo que as restrições de integridade podem exprimir a informação negativa em termos mais conpactos do que os exemplos negativos fechados.

Definição 7.1: Uma restrição de integridade é uma cláusula de primeira ordem da forma $A_1 \vee ... \vee A_n \vee \neg B_1 \vee ... \vee \neg B_m$. Os A_i e os B_i são átomos. Os A_i são chamados condições positivas e os B_i condições negativas. ♦

Note-se que $A_1 \vee ... \vee A_n \vee \neg B_1 \vee ... \vee \neg B_m$ pode ser escrita como $B_1 \wedge ... \wedge B_m \rightarrow A_1 \vee ... \vee A_n$. Neste trabalho adoptaremos uma notação baseada no Prolog tal como fizemos para outras cláusulas[7]. Os operadores de disjunção e de conjunção substituem-se por vírgulas como em $B_1,...,B_m \rightarrow A_1,...,A_n$. As vírgulas do lado do antecedente representam conjunções, as do lado do consequente representam disjunções. Mantemos no entanto a seta ($\rightarrow$) tal como fizemos para as cláusulas do programa. A negação é interpretada como negação por falha.

Exemplo 7.3: A restrição de integridade

union(A,B,C),member(X,C)→member(X,A),member(X,B)

exprime a condição de que se *X* pertence ao terceiro argumento de *union/3* (argumento de saída) então ou pertence ao primeiro ou ao segundo argumento (argumentos de entrada). ♦

As restrições de integridade são geralmente definidas como cláusulas '*range restricted*' [21,105]. Ser 'range restricted' significa que qualquer variável que ocorra na regra deve ter uma ocorrência numa condição positiva da regra. Neste trabalho, nós não consideramos esta restrição, uma vez que os programas que sintetizamos não são, eles próprios 'range restricted'.

Os exemplos positivos e negativos podem ser representados como restrições de integridade. Um exemplo positivo *p* corresponde à restrição *verdade→p*. Um exemplo negativo *n* é representado pela restrição *n→falso*. Embora exemplos e restrições possam

[7] É claro que as restrições de integridade são mais expressivas do que as cláusulas do Prolog. Nas restrições podemos ter mais do que um literal na cabeça da cláusula.

ser teoricamente tratados de uma maneira uniforme, nós fazêmo-lo separadamente uma vez que usamos estratégias diferentes para tratar exemlos positivos, negativos e restrições de integridade.

7.3.1 Satisfação de restrições

De forma a evitar a indução de programas sobre-gerais, o sistema de PLI deve testar em certos momentos se o programa candidato satisfaz restrições de integridade. Se for esse o caso o programa é aceite e o processo indutivo prossegue.

A seguir definimos as noções de *satisfação*, *violação* e *instância violante* [21]. Ao agente que verifica satisfação de restrições chamaremos *verificador de restrições de integridade* ou simplesmente *verificador de integridade*.

Definição 7.2: Dada uma restrição $B_1,...,B_m \rightarrow A_1,...,A_n$ e um programa P, a restrição é *satisfeita* por P se e só se a pergunta $\leftarrow B_1,...,B_m$,not $A_1,...,$not A_n falha em P. Se P não satisfaz I diz-se que P *viola* a restrição I. Se IT é um conjunto de restrições de integridade, P satisfaz IT se satisfaz todas as restrições em IT. ♦

Definição 7.3: Seja I uma restrição de integridade $B_1,...,B_m \rightarrow A_1,...,A_n$. $I\theta$ é uma *instância violante* de I se e só se θ é uma substituição resposta possível para a pergunta $\leftarrow B_1,...,B_m$,not $A_1,...,$not A_n, quando posta a P. ♦

Exemplo 7.4: A restrição de integridade *sort(X,Y)→sorted(Y)* não é satisfeita pelo programa {*sort(X,X)*}∪{definição de *sorted/1*}. Podemos verificar isso transformando a restrição na pergunta

$\leftarrow$*sort(X,Y)*,not *sorted(Y)*.

Esta pergunta sucede em {*sort(X,X)*}∪{definição de *sorted/1*} com a substituição resposta {*X/[1,0], Y/[1,0]*}.. Assim, uma instância violante é

sort([1,0],[1,0])→sorted([1,0]).

♦

Dada a Definição 7.2, podemos verificar se um programa *P* satisfaz uma restrição transformando a restrição numa pergunta e pondo essa pergunta a *P*, usando SLDNF. Embora esta seja uma forma simples de verificar a consistência, é potencialmente ineficiente. É simples porque porque não requer o emprego de demonstradores de teoremas especiais. A sua ineficiência é devida à natureza de geração e teste do SLD(NF).

Exemplo 7.5: A restrição de integridade *sort(X,Y)*→*sorted(Y)* podes ser transformada numa pergunta ←*sort(X,Y)*,not *sorted(Y)*. Para verificar a consistência da restrição e do programa *P*, colocamos a pergunta a *P*. A resolução SLDNF constrói todas as instanciações possíveis do literal *sort(X,Y)* e, para cada valor de *Y*, testa se esse valor é ou não uma lista ordenada (assumindo que *X* e *Y* tomam valores no domínio das listas). Quando uma lista não ordenada é encontrada, temos uma instância violante da restrição. Este processo pode ser muito ineficiente. Suponhamos que *X* e *Y* tomam valores sobre listas de tamanhos 0,1,2,3 e 4, cujos elementos são inteiros em {0,1,…,9}. Isto representa um universode mais de 10000 listas. Para responder a essa pergunta, o SLDNF pode ter de experimentar todos os valores possíveis. Este problema cresce exponencialmente em dificuldade com a aridade do predicado do primeiro literal. ♦

Na área da aprendizagem automática, programação lógica por indução incluída, tem sido dada relativamente pouca atenção às restrições de integridade. Luc De Raedt empregou restrições de integridade no seu sistema CLINT [21]. As restrições são transformadas em perguntas e confrontadas com os programas induzidos, tal como sugere a Definição 7.2. Por esse motivo, a procura de uma instância violante é ineficiente. Se alguma instância violante é encontrada, aquele sistema tenta determinar qual o predicado que está incorrectamente definido através da ajuda de um oráculo.

Podem-se encontrar outros verificadores de integridade na literatura de programação lógica, tais como SATCHMO [65], e o de Sadri e Kowalski [105]. O problema com este tipo de verificadores de integridade é a sua ineficiência.

7.4 MONIC e a estratégia Monte Carlo

Nesta Secção descrevemos MONIC (*Monte Carlo Integrity Checker*), um *método de Monte Carlo*[8] [103] para lidar com restrições de integridade, que foi integrado no nosso sistema de PLI SKILit. Como já vimos no Capítulo 5 o sistema SKILit constrói um programa lógico P adicionando uma cláusula C de cada vez a uma teoria inicial P_0. O Algoritmo 6 descreve o processo de indução é mostra onde é feita a verificação de integridade.

$P := P_0$
enquanto P não satisfaz algum critério de paragem
    constrói nova cláusula C
    se $P \cup \{C\} \cup BK$ satisfaz restrições de integridade
        $P := P \cup \{C\}$
    fim de se
fim de enquanto

Algoritmo 6: Descrição alto nível do SKILit.

Em cada ciclo, depois da geração de uma cláusula C, há um teste de consistência que envolve o novo programa $P \cup \{C\}$. Este novo programa só é aceite se satisfizer as restrições de integridade. Descrevemos agora de que forma serão processadas as restrições de integridade.

[8] De acordo com Rubinstein [103], o termo "Monte Carlo" foi introduzido por von Neumann e Ulam durante a Segunda Grande Guerra, como uma palavra de código para o trabalho secreto em Los Alamos, tendo o Método de Monte Carlo sido aplicado em problemas relacionados com a bomba atómica. Hoje em dia, ainda segundo Rubinstein, é o mais poderoso e o mais utilizado método em análise de problemas complexos de simulação, com um vasto leque de aplicações.

7.4.1 Restrições de integridade operacionais

MONIC lida com restrições de integridade da forma $A_1,...,A_n \rightarrow B_1,...,B_m$, conforme definido antes. Para além disso, impomos mais duas condições a uma restrição de integridade *I* para um programa *P* definindo um predicado *p/k*.

1. O literal mais à esquerda do antecedente deve ser um literal não negado com predicado *p/k*.

2. Se *I* é transformada numa pergunta $\leftarrow B_1,...,B_m$,not $A_1,...,$not A_n e os argumentos de entrada de B_1 estiverem instanciados, então a pergunta deve ser uma pergunta *aceitável* com respeito aos modos de entrada/saída dos predicados em *Q*.

Uma pergunta $\leftarrow L$, em que *L* é $p(X_1,...,X_n)$ ou *not* $p(X_1,...,X_n)$, é aceitável se todos os argumentos de entrada X_i estiverem totalmente instanciados. Uma pergunta $\leftarrow p(X_1,...,X_n)$, *MaisLiterais* é *aceitável* se, após instanciação de todos os argumentos X_i, $\leftarrow$*MaisLiterais* é uma pergunta aceitável, onde *MaisLiterais* é uma conjunção de literais. Uma pergunta $\leftarrow$*not* $p(X_1,...,X_n)$, *MaisLiterais* é *aceitável* se $\leftarrow$*MaisLiterais* é uma pergunta aceitável. O teste de aceitabilidade de uma pergunta é trivial dadas as declarações de modo dos predicados envolvidos. Esta condição garante que os modos de entrada/saída dos predicados envolvidos serão respeitados.

A primeira condição garante que a restrição de integridade condiciona o predicado *p/k*, uma vez que o literal com predicado *p/k* se encontra no corpo da restrição. O facto deste literal ter de estar na posição mais à esquerda permite procurar uma instância violante da restrição de integridade partindo de um consequência lógica fechada do programa *P*.

As restrições de integridade aceites por MONIC são *restritivas* (*restrictive*) no sentido definido por De Raedt [21]. Neste tipo de restrições, os literais relativos ao predicado a ser induzido estão no antecedente. Um exemplo de uma restrição de integridade restritiva em relação ao predicado *union/3* é

union(A,B,C),member(X,A)→member(X,C).

Esta restrição diz que, se a lista *C* é o resultado de reunirmos as listas *A* e *B*, então todo o elemento *X* de *A* deve ser um elemento de *C*.

Uma restrição de integridade com os literais do predicado a ser induzido no consequente são chamadas *generativas*. Um exemplo de uma restrição generativa relativamente ao predicado *union/3* é *true→union(A,A,A)*. Aqui não consideramos este tipo de restrições embora pareça possível estender o nosso verificador de integridade de forma a que as possa tratar.

7.4.2 O algoritmo para verificação de integridade

O nosso algoritmo de verificação de integridade (MONIC, Algoritmo 7) pega num determinado programa *P* definindo algum predicado *p/k* e um conjunto de restrições de integridade *IT*, e dá uma de duas respostas possíveis. Ou *P* e *IT* são *inconsistentes* (alguma restrição $I \in IT$ é violada), ou *P* e *IT* não são dados como inconsistentes e são consideradas *provavelmente consistentes*.

O método de Monte Carlo é baseado na geração aleatória de factos sobre o predicado *p/k* que são consequências lógicas do programa *P*. Cada um desses factos é usado para procurar uma instância logicamente falsa de alguma restrição $I \in IT$. Se uma tal instância for encontrada, temos a certeza de que *P* e *IT* são de facto inconsistentes. Se, pelo contrário, nenhum dos factos resulta numa instância violante de alguma $I \in IT$, o método pára após um número limitado de tentativas. Nesse caso, *P* e *IT* não são dados como inconsistentes mas apenas provavelmente consistentes.

A geração aleatória de consequências lógicas fechadas do programa *P* é central ao algoritmo de verificação de integridade, pelo que merece alguma atenção. Para obter o facto *f*, tal que $P \vdash f$, começamos pelo termo mais geral $p(X_1,...,X_k)$ de *p/k* ($X_1,...X_k$ são variáveis). Por uma questão de clareza, vamos supor que k=2. Suponhamos também que *mode(p(+,-))* e $type(p(tipo_X,tipo_Y))$, e que o termo mais geral é *p(X,Y)*. Agora queremos

uma pergunta $\leftarrow p(X,Y)$, onde X está instanciado por um termo de tipo $tipo_X$ (note-se que X é um argumento de entrada). Para isso, escolhemos um termo t_{in} do tipo $tipo_X$. Após pormos a pergunta ao programa P, a variável Y é instanciada com o termo t_{out}. O facto f é $p(t_{in},t_{out})$.

A natureza aleatória de f vem da escolha dos argumentos de entrada. Cada termo escolhido de um determinado tipo é retirado aleatoriamente de uma dada população de termos com uma distribuição fixa (ver Secção 7.4.3).

entrada: Programa P definindo o predicado p/k;
Declarações de modo e de tipo do predicado p/k;
Um conjunto de restrições de integridade IT;
Inteiro n.
saída: Uma resposta de entre {inconsistente, provavelmente consistente}

1. Gera pergunta Q
   $p(X,Y)$ é o termo mais geral de p/k
   (X representa os argumentos de entrada,Y os de saída)
   Para cada variável $V_i \in X$, instancia-a aleatoriamente com t_i do tipo $tipo(V_i)$;
   $\theta_{in} = \{V_i/t_i\}$
   $Q := \leftarrow P(X,Y)\theta_{in}$
2. Põe pergunta Q a P
   Se Q falha então volta ao passo 1;
   Se não obtemos uma substituição resposta θ_{out}
   (havendo várias substituições, consideramos cada uma delas)
3. Gera facto f
   $f = P(X,Y)\theta_{in}\theta_{out}$
4. Para cada $I \in IT$, procura instância violante de I.
   Transforma I numa query $\leftarrow L, MaisLiterais$
   θ_{uni} é o unificador do literal mais à esquerda L com f
   Põe pergunta $\leftarrow MaisLiterais\theta_{uni}$ a P
   Se a pergunta sucede então P viola I
   guarda f como um exemplo negativo
   retorna 'inconsistente'
5. Após n perguntas retorna 'provavelmente consistente'
   Caso contrário volta ao passo 1.

Algoritmo 7: MONIC: O verificador de integridade.

Dado um facto, este é unificado com o literal mais à esquerda do antecedente de $I \in IT$. A restrição pode agora ser transformada numa pergunta aceitável. A resposta obtida pondo a pergunta a *P* é sucesso ou falha. O sucesso significa que uma instância violante de *I* foi encontrada, resultando daí que *P* e *I* são inconsistentes. A falha significa que, embora não fosse encontrada uma instância violante, *P* e *I* podem ser ainda inconsistentes. Contudo, quanto maior for o número de factos que violam *I*, mais provável é que *P* e *I* sejam consistentes. Mais adiante na Secção 7.7.1 darei uma medida desta probabilidade.

Exemplo 7.6: O programa *P* abaixo contém uma definição incorrecta de *rv/2* que supostamente inverte a ordem dos elementos de uma dada lista. Definições para *append/3* e *last/2* são também dadas como parte do conhecimento de fundo.

```
mode(rv(+,-)).
type(rv(list,list)).
rv([A,B|C],[B,A|C]).
rv([A|B],C)←rv(B,D),append(D,[A],C).

mode(append(+,+,-)).
type(append(list,list,list)).
append([],A,A).
append([A|B],C,[A|D])←append(B,C,D).

mode(last(+,-)).
type(last(list,int)).
last([X],X).
last([X|Y],Z)←last(Y,Z).
```

A seguinte restrição de integridade *I* impõe que em cada facto *rv(X,Y)*, o primeiro elemento da lista *X* é o último elemento da lista *Y*.

$$rv(X,Y),X=[A|B] \rightarrow last(Y,A).$$

Sigamos agora uma iteração do MONIC (Algoritmo 7).

Passo 1: *rv(X,Y)* é o termos mais geral;
X é o único argumento de entrada e tem tipo *list*;
Uma escolha aleatória de um termo do tipo *list* dá *t=[4,1,5]*;

A pergunta *Q* é *rv([4,1,5],X)*.

Passo 2: A pergunta *Q* sucede em *P* e obtemos θ_{out}={*X/[1,4,5]*}.

Passo 3: *f* é *rv([4,1,5],[1,4,5])*.

Passo 4: *I* é transformada na pergunta

←*rv([4,1,5],[1,4,5]), [4,1,5]=[4|[1,5]],* not *last([4,1,5],4)*.

A pergunta sucede. *I* é violada.

Guarda *rv([4,1,5],[1,4,5])* como exemplo negativo.

Retorna 'inconsistente'. ♦

7.4.3 Tipos e distribuições

Os factos aleatórios são obtidos gerando aleatoriamente os argumentos de entrada de uma pergunta que é colocada a um programa *P*. A geração aleatória de cada argumento é feita segundo uma distribuição definida para o tipo desse argumento. Aqui, a cada tipo é associada uma *distribuição* que pode estar pré-definida no sistema de PLI, ou pode ser definida pelo próprio utilizador. Actualmente definimos a distribuição de um tipo especificando a probabilidade de obter termos de comprimento 0, 1, 2, etc. Uma alternativa para a definição de distribuições de tipos são os programas lógicos estocásticos de Muggleton [81].

Exemplo 7.7: Definimos a distribuição para o conjunto de listas de comprimento 0 a 4, com elementos 0 a 9, da forma seguinte.

Probabilidade(comprimento da lista *L*=*n*)=0.2 para *n*=0,...4.

Probabilidade(um dado elemento de *L* ser *d*)=0.1 para *d*=0,...,9.

Em consequência, 0.2 é a probabilidade de obter a lista vazia (*[]*). A probabilidade de obter a lista *[3]* é 0.2×0.1=0.02. ♦

Embora se tenha que definir uma distribuição para cada tipo para usar MONIC, a escolha das distribuições não parece difícil. Na realidade, as distribuições que utilizamos nas experiências foram praticamente a nossa primeira escolha.

7.5 Avaliação

Realizamos algumas experiências para avaliar o desempenho do SKILit juntamente com o módulo MONIC de verificação de restrições de integridade. Para isso escolhemos dois predicados e para cada um deles construímos vários conjuntos de restrições de integridade. A metodologia de avaliação é idêntica à descrita na Secção 6.1, excepto que aqui são dadas também as restrições de integridade ao sistema. Na Secção 7.5.2 descrevemos experiências com o predicado *union/3*. Em todas as experiências, o número de perguntas geradas pelo Algoritmo 7 (Inteiro *n*) foi 100.

7.5.1 append/3 e rv/2

Para o predicado *append/3* foram utilizados os seguintes quatro conjuntos de restrições de integridade (ver Apêndice A para definições de *member/2* e *sublist/2*):

ic1: *append(X,Y,Z),member(A,X)→member(A,Z).*
append(X,Y,Z),member(A,Y)→member(A,Z).

ic2: *append(X,Y,Z),sublist([A,B],X)→sublist([A,B],Z).*
append(X,Y,Z),sublist([A,B],Y)→sublist([A,B],Z).

ic3: *append(X,Y,Z),sublist(A,X)→sublist(A,Z).*
append(X,Y,Z),sublist(A,Y)→sublist(A,Z).

ic4: *append(X,Y,Z),sublist(A,X)→sublist(A,Z).*
append(X,Y,Z),sublist(A,Y)→sublist(A,Z).
append([_|_],X,X)→false.

Explicamos agora por palavras o significado de algumas restrições. A primeira restrição em ic1, por exemplo, diz que se a lista *Z* é o resultado de concatenarmos as listas *X* e *Y*, então qualquer elemento *A* de *X* deve ser um elemento de *Z*. A segunda restrição em ic2

diz que se *A* e *B* são dois elementos consecutivos de *Y*, devem também ser dois elementos consecutivos de *Z*. A terceira restrição de ic4 diz que se concatenarmos uma lista com pelo menos um elemento (*[_|_]*) à lista *X*, não devemos obter a mesma lista *X*.

Para *rv/2* temos dois conjuntos de restrições:

ic1: *rv(X,Y),sublist([A,B],X)→sublist([B,A],Y).*

ic2: *rv(X,Y),length(X,N)→length(Y,N).*
rv(X,Y),member(A,X)→member(A,Y).
rv(X,Y),member(A,Y)→member(A,X).

A primeira restrição diz que se *Y* resulta da inversão da lista *X* então quaisquer ois elementos consecutivos de *X* devem ser também consecutivos, mas en ordem inversa, em *Y*. A primeira restrição em ic2 diz que a lista invertida tem o mesmo comprimento do que a lista original.

Nas experiências realizadas, o sistema SKILit + MONIC consegue melhores resultados em termos de grau de acerto do que o SKILit com exemplos negativos apenas (Secção 6.2.1). Para a restrição de integridade em ic1, a síntese de *rv/2* atinge os 100% de acerto com apenas 10 exemplos positivos aleatoriamente escolhidos. Os resultados com o predicado *append/3* também melhoraram em relação aos resultados obtidos com exemplos negativos, principalmente para os conjuntos de restrições de integridade ic1 e ic2 (ver Figura 7.1).

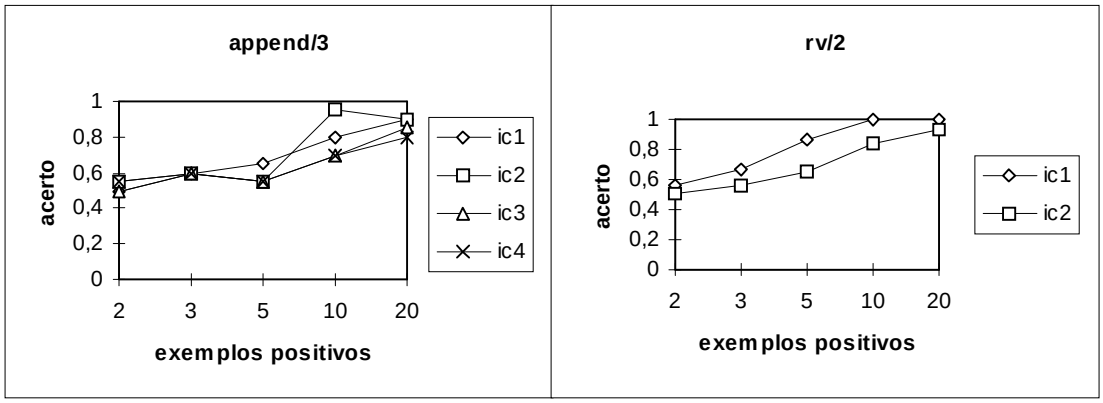

Figura 7.1: SKILit + MONIC: grau de acerto obtido.

Em termos de percentagem de programas totalistas, os resultados também foram superiores aos obtidos com exemplos negativos para grande parte dos conjuntos de restrições de integridade escolhidos. Isso significa que o utilizador, quando dispõe de restrições de integridade, pode aumentar as possibilidades de sintetizar o programa que pretende (ver Figura 7.2).

Tal como acontece com exemplos positivos e negativos, a escolha de restrições de integridade adequadas é importante. O conjunto de restrições ic2 para o predicado *rv/2* não conseguiu bons resultados pois essas restrições não cobrem muitos exemplos negativos importantes, tais como *rv([1,2],[1,2])* (as restrições dizem apenas que os dois argumentos de *rv(X,Y)* devem ter o mesmo número de elementos e que todo o elemento do argumento de entrada é elemento do argumento de saída).

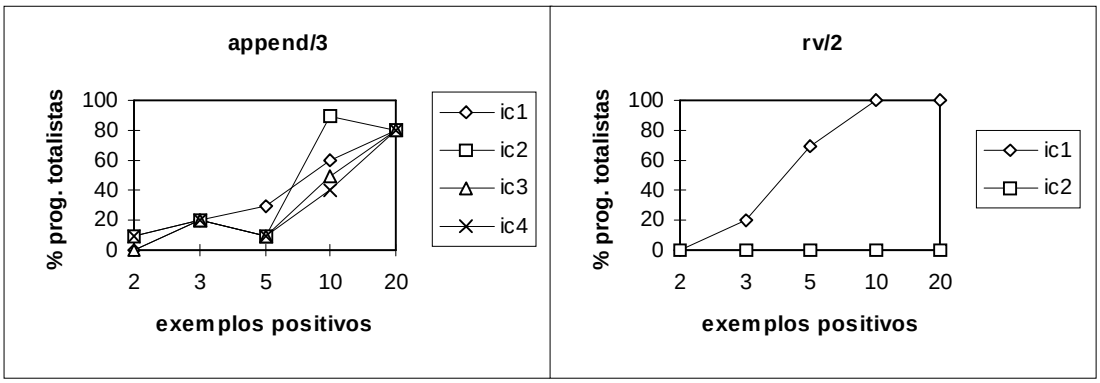

Figura 7.2: SKILit + MONIC: percentagem de programas totalistas.

Como podemos ver pela Figura 7.3, o custo de utilização de restrições de integridade em termos de tempo gasto na síntese é comparável ou melhor do que o emprego exclusivo de exemplos negativos (Secção 6.2).

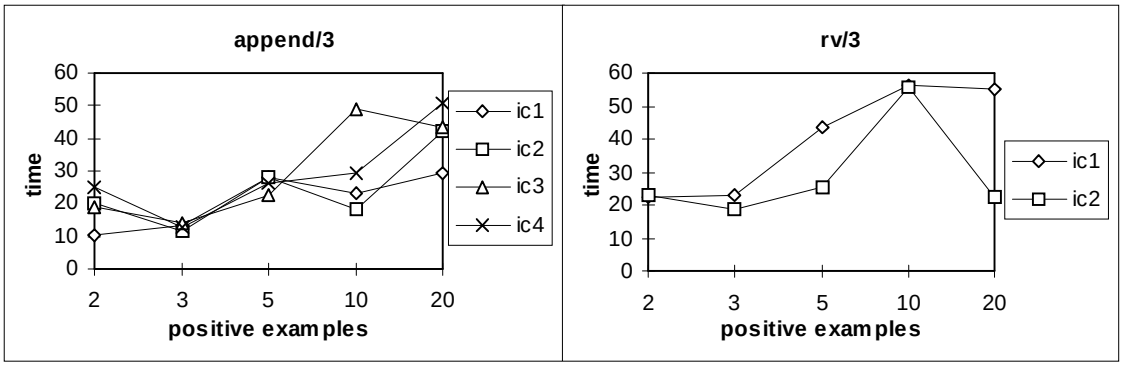

Figura 7.3: SKILit + MONIC: tempo de processador gasto (segundos).

7.5.2 union/3

Para o predicado *union/3* usamos dois conjuntos de restrições de integridade.

ic1: *union(A,B,C),member(X,A)→member(X,C).*
union(A,B,C),member(X,B)→member(X,C).
union([X|B],C,D),member(X,C),D=[X|E]→ C=[X|F].

ic2: *union(A,B,C),member(X,A) →member(X,C).*
union(A,B,C),member(X,B) → member(X,C).
union(A,B,C),member(X,A),member(X,B),append(A,B,C) → false.

Em ic1 a primeira restrição diz que, para cada facto *union(A,B,C)*, todo elemento da lista *A* deve estar em *C*. A segunda restrição diz que todo elemento de *B* deve estar em *C*. A terceira restrição aplica-se quando quando o primeiro elemento *X* da primeira lista de entrada pertence à outra lista de entrada (*C*) e simultaneamente *X* é o primeiro elemento da lista de saída (*D*). Neste caso *X* deve também ser o primeiro elemento de *C* (*union([1],[1,2],[1,2])* é um exemplo positivo enquanto *union([1],[2,1],[1,2,1])* é um exemplo negativo).

Em ic2, as duas primeiras restrições são idênticas às duas primeiras de ic1. A terceira restrição diz que se *A* e *B* têm um elemento em comum então o resultado de *union/3* deve ser diferente do resultado obtido pela concatenação das duas listas (*union([2],[1,2],[1,2])* e *append([2],[1,2],[2,1,2])*).

conj. de restrições	*GEC*	*acerto*	*totalistas*	*tempo*
ic2	decomp_test_rec_comp_2	0.689	24	251.802
ic1	decomp_test_rec_comp_2	0.758	38	147.188
ic2	decomp_+test_rec_comp_2	0.780	44	122.487

Tabela 7.1: Experiências com *union/3* e restrições de integridade.

Estas experiências com *union/3* foram realizadas num enquadramento semelhante ao usado na Secção 6.3. Para cada conjunto de restrições, corremos SKILit+MONIC 50 vezes dando em cada execução 30 exemplos positivos retirados aleatoriamente de U2i(+). Não foram dados exemplos negativos. Utilizamos também duas gramáticas de estrutura clausal diferentes: decomp_test_rec_comp_2, e decomp_+test_rec_comp_2. Esta última força as cláusulas sintetizadas a terem literais de teste.

Os resultado são dados na Tabela 7.1. O acerto e a percentagem de programas totalistas obtidas com as restrições de integridade em ic1 e ic2 foram claramente superiores aos resultados obtidos com exemplos negativos aleatórios (Tabela 6.2), e muito semelhantes aos obtidos com exemplos negativos escolhidos manualmente. No caso de ic2 com a GEC decomp_+test_rec_comp_2, os resultados foram claramente superiores aos obtidos com exemplos negativos escolhidos. O tempo de síntese foi comparável ao das experiências da Secção 6.3.

7.6 Trabalho relacionado

Existem outros procedimentos para a verificação da satisfação de restrições de integridade. Sadri e Kowalski propõem um procedimento de prova que verifica a satisfação de integridade, dado um programa lógico e um conjunto de restrições [105]. Este procedimento verifica se uma alteração ao programa viola alguma das restrições. Para uma única restrição de integridade, o procedimento de Sadri e Kowalski é equivalente ao SLDNF (Secção 3.2.3). Um outro demonstrador de teoremas, o SATCHMO, foi proposto por Manthey e Bry [65].

Este tipo de verificadores de integridade usa uma abordagem sistemática para procurar inconsistências. Para encontrar uma instância violante de uma restrição de integridade tem que se considerar um grande número de instanciações possíveis. A estratégia de Monte Carlo tenta apenas uma amostragem dessas instanciações e reduz drasticamente o esforço de procura de uma instância violante de uma restrição de integridade.

Luc De Raedt utilizou restrições de integridade para melhorar o seu sistema CLINT [21]. No seu trabalho, uma restrição de integridade é verificada transformando-a numa pergunta e pondo-a ao programa gerado.

7.7 Discussão

7.7.1 O número de perguntas

O verificador de integridade proposto permite encontrar uma instância violante de uma restrição de integridade *I* gerando um número limitado de perguntas. Cada pergunta é posta ao programa *P* sendo obtidos factos a partir das respostas. Com estes factos tentamos obter uma instância violante da restrição *I*. Se tal instância é encontrada, podemos estar certos de que o programa e a restrição são inconsistentes. Caso contrário, não podemos ter a certeza de que são consistentes. MONIC é incompleto.

No entanto, a distribuição binomial diz-nos que a probabilidade, após *n* perguntas, de encontrar uma inconsistência é de $\alpha=(1-p)^n$, onde *p* é a probabilidade de uma dada pergunta obtida a partir de *I* e posta ao programa *P* suceder. Esta distribuição diz-nos que quanto mais alto for o valor de *n*, mais provável é que *P* e *I* sejam consistentes no caso de não encontrarmos nenhuma inconsistência.

Intuitivamente, o valor de *p* mede o nível de consistência da restrição de integridade com o programa *P*. O verdadeiro valor de *p* é desconhecido. Podemos, no entanto, escolher um limite inferior para o valor de *p*, significando que, se há uma inconsistência, então pelo menos $100\times p\%$ das perguntas dão uma instância violante de *I*. Chamamos a isto o

"pressuposto da generalidade das restrições de integridade". Após *n* perguntas não violantes, e dado uma valor a *p*, podemos estar $100\times(1-\alpha)\%$ seguros que *P* e *I* são consistentes.

Infelizmente, o utilizador não pode dizer, a priori, se este pressuposto vai ser ou não verificado, pois isso depende de *P*, o qual não é conhecido. Podemos, no entanto, ter uma noção intuitiva da generalidade de uma dada restrição, e assim preferir restrições mais 'gerais' tais como *sort(X,Y)→sorted(Y)* às mais 'específicas' como *sort([2,3],[3,2])→false*.

7.7.2 Integridade e completude

Quando MONIC encontra uma instancia violante de uma restrição de integridade, isso significa que o programa não satisfaz a restrição. Nesse sentido MONIC é *correcto* ou íntegro (*sound*), uma vez que não encontra falsas inconsistências. No entanto, MONIC é apenas tão correcto quanto o procedimento de prova que estiver a utilizar para responder às perguntas. Para garantir a integridade do procedimento de prova deve ser usada uma *regra de computação segura* (*safe*) [46].

Tal como vimos anteriormente, MONIC pode não encontrar uma instância violante mesmo que uma exista. Por esse motivo é *incompleto*. No entanto, quando MONIC não encontra uma inconsistência temos um nível de confiança associado (α). Por outro lado, podemos controlar esse nível de confiança escolhendo um número apropriado de perguntas.

7.7.3 Limitações

A utilização de restrições de integridade em especificações, independentemente da forma como elas são verificadas, representa um esforço adicional para o utilizador, uma vez que escrever um conjunto adequado de restrições de integridade pode ser tão complexo quanto escrever o próprio programa. No entanto, a utilização de restrições de

integridade na especificação é facultativa. Se são faceis de escrever, podem ser exploradas pelo sistema. Caso contrário o sistema pode usar exemplos negativos apenas.

As limitações mais importantes, específicas da nossa abordagem de Monte Carlo, são a incompletude da procura de uma instância violante (discutida na Secção 7.7.2), a necessidade de se associarem funções de distribuição de probabilidade aos tipos utilizados (Secção 7.4.3) e o facto de não lidarmos com qualquer restrição de integridade (Secção 7.4.1).

Embora as distribuições que se associam aos tipos possam ter influência no comportamento do método, não fizemos grande esforço para procurar uma distribuição (*tuning*) para conseguirmos os resultados alcançados. Por isso, em termos práticos, esta limitação não parece significativa.

As restrições de integridade restritivas tratadas por MONIC parecem ser as mais adequadas à representação de informação negativa num contexto de síntese a partir de especificações incompletas. Por esse motivo, não consideramos outros tipos de restrições, embora tal extensão pudesse ser contemplada no futuro.

8. Conclusões

8.1 Sumário

Nesta dissertação apresentamos uma metodologia de construção automática de programas lógicos e que serve de base ao sistema SKILit. O utilizador que pretenda obter um programa *P* fornece ao sistema informação que descreve alguns aspectos desse programa. O sistema assume que a informação que lhe é dada é *incompleta*. A partir desses dados, a que chamamos *especificação*, o SKILit constrói um programa *P'* que os satisfaz. Caso *P'* não satisfaça o utilizador, este pode fornecer mais dados para a especificação e tornar a correr o SKILit. O SKILit é *indutivo*, pois a partir de uma dada especificação incompleta *S* sintetiza um programa *P* que pode ter consequências lógicas não descritas em *S*. Em suma, o SKILit é um sistema de síntese indutiva de programas lógicos a partir de especificações incompletas (Capítulos 1 e 5). Uma especificação dada ao SKILit é composta por elementos diversos, dos quais se destacam os *exemplos positivos* e os *exemplos negativos*.

A estratégia de *indução iterativa* usada pelo SKILit (Capítulo 5) permitiu responder a uma importante limitação de sistemas de PLI que representam o estado da arte, tais

como GOLEM [82], FOIL [96] e Progol [80]. Ao contrário destes sistemas, o SKILit consegue sintetizar programas recursivos a partir de conjuntos esparsos de exemplos positivos. Para ser bem sucedido na tarefa de síntese, o utilizador do SKILit não tem de dar uma lista de exemplos tão completa quanto possível, nem de adivinhar os exemplos de acordo com possíveis caminhos de resolução do programa pretendido. Ambos os procedimentos seriam contrários ao espírito da programação através de exemplos. A indução iterativa representa um avanço no estado da arte da síntese de programas recursivos.

O sistema SKILit tem uma *classe de programas sintetizáveis* bastante alargada e ao mesmo tempo está preparado para lidar com conjuntos de exemplos incompletos. Sistemas anteriores a SKILit, como LOPSTER [60], FORCE2 [12] e CRUSTACEAN [1], são também capazes de sintetizar definições recursivas a partir de poucos exemplos. Todos eles no entanto sintetizam programas dentro de uma classe muito restrita e bem definida. As estratégias que utilizam, embora interessantes, a isso os obrigam. Dentro dos programas sintetizados pelo SKILit mas não pelos sistemas acima citados encontramos, por exemplo, o *quicksort/2*, que tem dois literais recursivos numa cláusula, e o *union/3* que tem duas cláusulas recursivas.

A classe de programas sintetizáveis pelo SKILit numa determinada tarefa de síntese pode ser limitada por uma *gramática de estrutura clausal* (GEC). Embora não sendo necessária ao funcionamento do sistema, uma GEC permite uma construção mais eficiente do programa pretendido. Uma GEC serve para transmitir ao SKILit uma determinada estratégia de programação tal como “dividir-e-conquistar”, “geração-e-teste”, etc. Uma vez que cada estratégia serve um vasto conjunto de programas, as GEC são altamente reutilizáveis.

O conhecimento de fundo (CF) tem um papel muito importante no processo de síntese. É a partir dos predicados definidos no CF que se constitui o vocabulário que o SKILit vai utilizar na construção das cláusulas. Assim sendo, um conhecimento de fundo

adequado pode facilitar a tarefa de síntese. Por outro lado, o insucesso da tarefa de síntese pode-se dever à pobreza do CF (Secção 4.9).

A construção de uma cláusula é feita procurando uma *ligação relacional* entre os argumentos de entrada e os argumentos de saída de um exemplo positivo ou de um esboço de algoritmo. Esta estratégia do SKILit permite que o conhecimento de fundo condicione a construção da cláusula uma vez que da ligação fazem parte apenas consequências lógicas do CF, dos exemplos positivos e de cláusulas entretanto sintetizadas pelo SKILit.

A estratégia de construção de cláusulas usada pelo SKILit permite que o CF seja definido *intensionalmente* (Secção 4.4) como qualquer programa em Prolog, facilitando a sua criação e manutenção por parte do utilizador.

O utilizador do SKILit pode especificar o programa pretendido totalmente através de exemplos embora disponha de meios complementares para o fazer. Um *esboço* (Secção 4.5.1), por exemplo, fornece ao sistema informação parcial sobre como um determinado exemplo positivo é processado. O SKILit explora os esboços que lhe são dados consolidando-os e transformando-os em cláusulas operacionais (Capítulo 5). Esboços e exemplos são tratados de maneira uniforme. Mostra-se que o operador de refinamento de esboços utilizado pelo SKILit é completo sob os pressupostos adequados.

Se um sistema que exige um número excessivo de exemplos positivos não é adequado à síntese de programas, também não o seria um sistema que exigisse um grande número de exemplos negativos. Por esse motivo estendemos o SKILit acrescentando a possibilidade deste lidar com *restrições de integridade* (MONIC). Uma restrição de integridade pode substituir um grande número de exemplos negativos fechados, o que permite ao utilizador uma maior expressividade e economia de tempo.

A *estratégia de Monte Carlo* criada para lidar com as restrições parece bastante eficiente o que a torna apropriada para o uso num contexto de síntese indutiva. Outras estratégias existentes, mais pesadas, podem-se tornar quase impraticáveis neste contexto.

8.2 Problemas em aberto

O SKILit não é ainda uma ferramenta útil de apoio num ambiente de programação em lógica. Demos, no entanto, alguns passos no sentido de valorizar a abordagem indutiva à síntese de programas. Muitos problemas importantes ainda têm, no entanto, que ser resolvidos. As soluções que propomos poderão ser melhoradas. É destes problemas deixados e das linhas de investigação que podem indicar que falaremos nesta Secção.

8.2.1 A selecção dos predicados auxiliares

Um sistema de síntese indutiva deve ter um conhecimento de fundo (CF) bastante rico, de forma a poder responder a um grande número de problemas. Nas experiências que habitualmente se fazem com os sistemas de PLI o CF define um conjunto de predicados auxiliares praticamente necessários e suficientes para que a síntese suceda.

Embora a estratégia de seguir uma ligação relacional permita ao SKILit filtrar muitos predicados auxiliares irrelevantes, o desempenho do sistema degrada-se quando o número de predicados auxiliares admissíveis é muito grande (Secção 3.4.5). A solução aqui adoptada involve o utilizador, o qual indica ao sistema quais os predicados admissíveis a ser considerados. É evidente que esta não é uma solução inteiramente satisfatória.

O SKILit falha também se os predicados do CF não forem suficientes, pois não faz invenção de predicados. Esta é uma tarefa difícil por si só [114]. No entanto, valeria a pena estender o SKILit, uma vez que um sistema utilizado na prática deve colmatar as falhas existentes no conhecimento de fundo.

8.2.2 Interactividade

Quando a especificação que o utilizador fornece ao SKILit não é suficiente para que o sistema construa um programa adequado, o remédio é examinar o resultado do SKILit e alterar a informação dada concordantemente.

Um sistema de síntese deve guiar o utilizador na construção e refinamento da especificação. Os sistemas interactivos tipicamente sugerem ao utilizador os exemplos a fornecer durante o próprio processo de síntese e desta forma libertam-no da ingrata tarefa de ter que adivinhar quais são os bons e os maus exemplos. Infelizmente, tais sistemas tendem a pôr demasiadas perguntas ao utilizador, perturbando-o. Por esse motivo optamos por uma solução não interactiva.

No entanto, seria importante que um sistema como o SKILit tivesse algumas ferramentas interactivas para avaliação dos programas produzidos e para depuração de especificações. Este tipo de interactividade pós-síntese seria menos incomodativa para o utilizador porque entraria em acção apenas se o programa final fosse incorrecto ou incompleto.

8.2.3 Muitos exemplos

O SKILit especializou-se na síntese a partir de poucos exemplos. Cada cláusula é construída para cobrir um exemplo positivo, e não são directamente tomados em conta os restantes exemplos positivos. O SKILit explora a estrutura interna do próprio exemplo, mas ignora os padrões que se possam repetir entre exemplos diferentes. Por essa razão o SKILit pode ter alguma dificuldade em lidar com grandes quantidades de dados.

8.3 Avaliação da abordagem

As experiências com o SKILit realizadas no Capítulo 6 mostraram que o sistema é capaz de sintetizar predicados manipuladores de listas e de outras estruturas a partir de conjunto de exemplos positivos mal escolhidos.

8.4 Contribuições principais para o estado da arte

Apresentamos a noção de esboço de algoritmo como um formalismo para descrever computações de forma parcial. Definimos a noção de refinamento e de consolidação de esboço e apresentamos um operador de refinamento que encontra todas as consolidações operacionais de um esboço. Este operador de refinamento é a base do motor indutivo da nossa metodologia de síntese de programas lógicos.

A indução iterativa desenvolvida e utilizada no SKILit permite a síntese de definições recursivas a partir de conjuntos esparsos de exemplos. Este resultado é importante por si só e principalmente considerando não haver no SKILit grandes restrições à classe de programas sintetizáveis. Em particular, o SKILit não assume que o programa a construir é recursivo (a menos que a gramática clausal o imponha). As soluções recursivas são preferidas em relação às não recursivas só se as primeiras envolverem cláusulas mais curtas. Este é um aspecto importante que não encontramos, por exemplo, em CRUSTACEAN, TIM ou SYNAPSE. Pelo facto de não depender de heurísticas de cobertura ou de ganho de informação (Secção 3.4.5) o SKILit pode ainda lidar com conjuntos de exemplos muito pequenos.

A classe de programas pode ser definida pela utilização de uma gramática de estrutura clausal. Esta gramática permite a representação de conhecimento genérico de programação. O sistema pode funcionar, no entanto, sem nenhuma gramática.

Por fim, a estratégia de Monte Carlo adaptada à verificação de integridade permite que a especificação inclua restrições de integridade bastante mais expressivas do que os

tradicionais exemplos negativos. A principal vantagem do nosso verificador de integridade MONIC é a sua eficiência.

8.5 O Futuro

Num plano mais imediato, o SKILit poderia beneficiar de uma reimplementação mais eficiente. Haveria também vantagens em dispôr de outras estratégias de procura para além da utilizada, que permitissem o processamento de grandes volumes de exemplos, a utilização de grandes bases de conhecimento de fundo e a síntese de cláusulas mais complexas. Seria importante que o sistema pudesse trabalhar com pouca informação negativa (exemplos negativos ou restrições de integridade). O sistema poderia também tirar mais partido da informação contida em esboços de algoritmo.

Mas o futuro ideal para um sistema como o SKILit seria estar integrado num ambiente de desenvolvimento de programas. Ser uma ferramenta entre outras, benefeciar de um interface gráfico que minimizasse o esforço do utilizador em construir uma especificação incompleta e que permitisse a descrição de esboços de algoritmo. Comunicar com outras ferramentas de desenvolvimento, desde simples editores de texto a módulos de análise estática e dinâmica de programas.

As técnicas indutivas de programação podem, no futuro, ser usadas por programadores iniciados, assim como por programadores experimentados. A programação por exemplos pode ser um paradigma de especificação importante para programadores iniciados que necessitam da programação para interagirem com aplicações complexas de uso quotidiano como processadores de texto, folhas de cálculo ou sistemas de gestão de bases de dados. As técnicas indutivas manterão os programadores involuntários tão longe do código quanto possível. Os programadores experientes poderão beneficiar de ferramentas baseadas na programação por exemplos inseridas em editores de texto avançados. As especificações incompletas poderão estar associadas ao código sendo utilizadas para síntese de programas e verificação.

Na sua base, a síntese automática de programas terá de ser mais baseada em conhecimento. Diferentes aspectos do conhecimento de programação serão representados e fornecidos aos sistemas de síntese. Estes diferentes tipos de conhecimento serão, por um lado, representados separadamente e por outro terão de ser combinados de forma a se obter, para cada tarefa, uma solução integrada. Nesta tese caminhamos nessa direcção, focando a nossa atenção em dois tipos de conhecimento de programação: conhecimento específico (esboços) e conhecimento genérico (gramáticas de estrutura).

Os sistemas de programação automática terão de possuír conhecimento acerca dos programas auxiliares e dos programas sintetizados. Ter uma lista de predicados como conhecimento de fundo não é suficiente, particularmente se essa lista fôr muito longa. Um sistema de programação automática deveria, por exemplo, ter facilidade em reconhecer exemplos positivos de um programa que já conhece.

Em suma, a programação automática proporciona desafios de investigação muito relevantes. A sua relevância deriva da importância da própria programação. Qualquer ferramenta ou metodologia que simplifique a tarefa de programar significa que poderão surgir aplicações mais poderosas. Seja devido à simplificação do desenvolvimento de aplicações, seja porque o interface das aplicações tornou estas mais fáceis de usar. Aplicações mais poderosas e mais fáceis de usar significam que as tarefas assistidas por computador (programação incluída) serão concretizadas mais eficientemente, deixando a quem tem que as desempenhar mais tempo precioso de sobra.

Referências

[1] Aha, D. W., Lapointe, S., Ling, C. X., Matwin S (1994): "Inverting Implication with Small Training Sets". Proceedings of the European Conference on Machine Learning, ECML-94. Ed. F. Bergadano, L. De Raedt. Springer Verlag.

[2] Azevedo, R., Costa, V.S., Damas, L., Reis, R. (1990): "YAP Reference Manual". Centro de Informática da Universidade do Porto.

[3] Banerji, R. B. (1964): "A Language for the Description of Concepts". *General Systems*, 9, pp. 135-141.

[4] Baroglio, C., Giordana, A., Saitta, L. (1992): "Learning Mutually Dependent Relations". *Journal of Intelligent Information Systems*, 1, pp. 159-176. Kluwer Academic Publishers, Boston.

[5] Bergadano, F., (1993): "Towards an Inductive Logic Programming Language". *Deliverable no. TO1 of ILP project*.

[6] Bergadano, F., Gunetti, D. (1993): "The Difficulties of Learning Logic Programs with Cut". *Journal of Artificial Intelligence Research* 1, pp. 91-107.

[7] Biermann, A. W., (1978): "The inference of regular LISP Programs from Examples". *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-8, No. 8, Agosto 1978.

[8] Biermann, A. W., (1990): "Automatic Programming". *Encyclopedia of Artificial Intelligence*. Ed. Stuart C. Shapiro. Wiley Interscience.

[9] Blum, L. and Blum, M. (1975): "Toward a Mathematical Theory of Inductive Inference". *Information and Control*, 28, pp. 125-155.

[10] Bratko, I. (1986): *Prolog Programming for Artificial Intelligence*. Addison-Wesley.

[11] Bratko, I., Muggleton, S., Varšek, A. (1992): "Learning Qualitative Models of Dynamic Systems". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[12] Brazdil, P., Jorge, A. (1992): "Modular Approach to ILP: Learning from interaction between Modules". *Logical Approaches to Machine Learning, Workshop notes*. ECAI 92.

[13] Brazdil, P. (1981): *A Model for Error Detection and Correction*. PhD Thesis. University of Edinburgh.

[14] Brazdil, P., Jorge, A. (1994): "Learning by Refining Algorithm Sketches". *Proceedings of ECAI-94*. Ed. T. Cohn. Wiley.

[15] Brazdil, P., Jorge, A. (1997): "Induction with Subtheory Selection". *ECML 97 - Poster Papers*. Ed. M. van Someren, G. Widmer. Laboratory of Intelligent Systems, Faculty of Informatics and Statistics, University of Economics, Prague.

[16] Buntine, W. (1988): "Generalized Subsumption and its Application to Induction and Redundancy". *Artificial Intelligence* 36, pp 149-176, Elsevier Sience Publishers B.V. (North Holland).

[17] Calejo, M. (1991): *A Framework for Declarative Prolog Debugging*. Dissertação de Doutoramento. Universidade Nova de Lisboa.

[12] Cohen, W. W. (1993): "Pac_learning a restricted class of recursive logic programs". *Proceedings of the third International Workshop on Inductive Logic Programming* (pp. 73-86). Bled, Slovenia. J. Stefan Institute.

[18] Cohen, W. W. (1993): "Rapid prototyping of ILP systems using explicit bias". *Proceedings of 1993 IJCAI Workshop on ILP*.

[19] Cypher, A. (Ed.) (1993): *Watch What I Do: Programming by Demonstration*. MIT Press.

[20] De Raedt, L. (1991): *Interactive Concept Learning*. PhD thesis. Katholieke Universiteit Leuven.

[21] De Raedt, L. (1992): *Interactive Theory Revision: An Inductive Logic Pogramming Approach*. Academic Press.

[22] De Raedt, L., Bruynooghe, M. (1993): "A theory of Clausal Discovery". *Proceedings of IJCAI-93*. Chamberry, França. R. Bajcsy (Ed.). Morgan Kaufmann.

[23] De Raedt, L., Idestam-Almquist, P, Sablon, G (1997): "θ-subsumption for Structural Matching". *Proceedings of ECML-97*. Prague. M. van Someren, G. Widmer (Ed.). Springer.

[24] De Raedt, L. Lavrac, N. (1995): "Multiple Predicate Learning in two Inductive Logic Programming settings". *Journal of Pure and Applied Logic*, 4(2):227-254.

[25] De Raedt, L., Lavrac, N., Dzeroski, S. (1993): "Multiple Predicate Learning". *Proceedings of IJCAI-93*. Chamberry, França. R. Bajcsy (Ed.). Morgan Kaufmann.

[26] Deville, Y. (1990): *Logic Programming, Systematic Program Development*. Addison-Wesley Publishing Company.

[27] Deville, Y., Lau, K.,(1994): "Logic Program Synthesis". *The Journal of Logic Programming, special issue Ten Years of Logic Programming,* volumes 19,20, May/July 1994.

[28] Diller, A. (1991): *Z, an introduction to formal methods*. Wiley.

[29] Dolšak, B. and Muggleton, S. (1992): "The Application of Inductive Logic Programming to Finite Element Mesh Design". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[30] Dromey, G. (1989): *Program Derivation, the development of programs from specification*. Addison-Wesley.

[31] Ducassé, M. (1994): "Logic Programming Environments: Dynamic Program Analysis and Debugging". *The Journal of Logic Programming, special issue Ten Years of Logic Programming,* volumes 19,20, May/July 1994.

[32] Esposito, F., Malerba, G., Semeraro, G. and Pazzani, M. (1993): "Document understanding: a machine learning approach". *Real-World Applications of Machine Learning, Workshop notes*. Ed. Y. Kodratoff, P. Langley. ECML-93, Viena.

[33] Feng, C. (1992): "Inducing Temporal Fault Diagnostic Rules from a Qualitative Model". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[34] Feng, C. and Muggleton, S. (1992): "Towards Inductive Generalization in Higher Order Logic". *Proceedings of the Ninth Interantional Workshop ML92*. Ed. Derek Sleeman and P. Edwards. Morgan Kaufmann.

[35] Fisher, A., (1988): *CASE: Using Software Development tools*. Wiley.
[36] Flach, P. (1995): *Conjectures: an inquiry concerning the logic of induction*. PhD Thesis. ITK dissertation series - 1.
[37] Flener, P. (1995): *Logic Program Synthesis From Incomplete Information*. Kluwer Academic Publishers.
[38] Flener, P., Deville, Y. (1992): "Logic Program Synthesis from Incomplete Specifications". Research Report RR 92-22. Université Catholique de Louvain, Unite d'Informatique.
[39] Flener, P., Popelínský, L., (1994): "On the use of Inductive Reasoning in Program Synthesis: Prejudice and Prospects". *Joint Proc. of LOPSTR'94 and META'94,* LNCS, Springer-Verlag.
[40] Giordana, A., Saitta, L., Baroglio, C. (1993): "Learning Simple Recursive Theories". *Proceedings of the 7th International Symposium, ISMIS'93*. Lecture Notes in Artificial Intelligence. Springer-Verlag.
[41] Grobelnik, M. (1992): "MARKUS: An Optimal Model Inference System". *Proceedings on ECAI-92 Workshop on Logical Approaches to Machine Learning*. Rouveirol, C. (Ed.).
[42] Heidorn, G. E. (1975): "Automatic Programming Through Natural Language Dialogue: A Survey". [99].
[43] Helft, N. (1987): "Inductive Generalization: a Logical Framework". *Proceedings of EWSL 87*. Ed. I. Bratko, N. Lavrac. Sigma Press.
[44] Helft, N. (1989): "Induction as nonmonotonic inference". *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pp 149-156. Morgan-Kaufmann.
[45] Hoare, C.A.R (1986): "Mathematics of Programming". BYTE 11(8).
[46] Hogger, C. J. (1990): *Essentials of Logic Programming*. Graduate texts in computer science series. Oxford University Press.
[47] Idestam-Almquist, P. (1993): "Generalization under implication by recursive anti-unification". *Proceedings of ILP-93*. Technical Report. Jozef Stefan Institute.
[48] Idestam-Almquist, P. (1993): *Generalization of Clauses*. PhD thesis. Report Series No. 93-025. Stockholm University, Royal Institute of Technology, Department of Computer and Systems Sciences.
[49] Idestam-Almquist, P. (1995): "Efficient Induction of Recursive Definitions by Structural Analysis of Saturations". *Proceedings of the Fifth International Workshop on Inductive Logic Programming*. Ed. L. De Raedt. Scientific Report. Departement of Computer Science, K.U. Leuven.
[50] Jaquet, J.-M. (Ed.) (1993): *Constructing Logic Programs*. Wiley Professional Computing. Wiley.
[51] Jazza, A. (1995): "Toward Better Software Automation". *Software Engineering Notes*, vol. 20 no. 1. ACM SIGSOFT.
[52] Joch, A. (1995): "How Software doesn't work". BYTE, 20(12).
[53] Jorge, A. and Brazdil, P. (1996): "Architecture for Iterative Learning of Recursive Definitions". *Advances in Inductive Logic Programming*. Ed. Luc De Raedt. IOS Press.

[54] Jorge, A. and Brazdil, P. (1996): "Integrity Constraints in ILP using a Monte Carlo approach". *Inductive Logic Programming, 6th International Workshop, ILP-96*. Ed. Stephen Muggleton. LNAI 1314. Springer..

[55] Jüllig, R. K. (1993): "Applying Formal Software Synthesis". *IEEE Software*. Vol. 10, No. 3, pp. 11-22.

[56] Kietz, J. and Wrobel, S. (1992): "Controlling the Complexity of Learning in Logic". *Inductive Logic Programming*. Ed. Stephen Muggleton. Academic Press Limited.

[57] Kijsirikul, B., Numao M. and Shimura, M. (1992): "Discrimination-Based Constructive Induction of Logic Programs". *Proceedings of AAAI-92*. Morgan-Kaufmann.

[58] Klingspor, V. (1994): "GRDT: Enhancing Model-Based Learning for Its Application in Robot Navigation". *Proceedings of the Fourth International Workshop on Inductive Logic Programming (ILP-94)*. GMD-Studien Nr. 237. GMD, Alemanha.

[59] Korf, R. E., (1990): "Search". *Encyclopedia of Artificial Intelligence*. Ed. Stuart C. Shapiro. Wiley Interscience.

[60] Lapointe, S., Matwin, S., (1992): "Sub-unification: A tool for efficient induction of recursive programs". *Proceedings of the Ninth International Conference on Machine Learning* (pp. 273-281). Aberdeeen, Scotland. Morgan Kaufmann.

[61] Lavrac, N., Dzeroski, S. (1992): "Inductive Learning of Relations from Noisy Examples". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press Limited.

[62] Lavrac, N., Dzeroski, S. (1994): *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

[63] Ling, C. X., (1991): "Inductive Learning From Good Examples". *12th International Joint Conference on Artificial Intelligence*. Ed. J. Mylopoulos, R. Reiter. Morgan Kaufmann.

[64] Lloyd, J. W., (1987) *Foundations of Logic Programming* (second, extended edition). Springer-Verlag.

[65] Manthey, R., Bry, F. (1988): "SATCHMO: a theorem prover implemented in Prolog". *Proceedings of CADE 88 (9th Conference on Automated Deduction)*. Springer-Verlag.

[66] Michalski, R. S. (1983): "A Theory and Methodology of Inductive Learning". [69].

[67] Michalski, R. S. (1990): "Learning, Machine". *Encyclopedia of Artificial Intelligence*. Ed. S. Shapiro. Wiley Inter-Science.

[68] Michalski, R. S. (1994): "Inferential Theory of Learning: Developing Foundations for Multistrategy Learning". *Machine Learning, A Multistrategy Approach, Volume IV*. Ed. Ryszard Michalski, Gheorghe Tecuci. Morgan Kaufmann.

[69] Michalski, R. S., Carbonell, J. and Mitchell, T. (1983) *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishing Company.

[70] Michalski, R. S., Larson, J. B. (1978): "Selection of most representative training examples and incremental generation of VL1 hypotheses: The underlying methodology and description of programs ESEL and AQ11". Technical report 867. Computer Science Department, University of Illinois, Urbana-Champaign.

[71] Minker, J. (Ed.) (1988): *Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers.
[72] Mitchell, T. (1982): "Generalization as Search". *Artificial Intelligence*, 18, pp. 203-226.
[73] Mitchell, T. (1990): "The need for biases in learning generalizations". *Readings in Machine Learning.* Ed. J. Shavlik and T. Dietterich. Morgan Kaufmann.
[74] Mofizur, C. R. and Numao, M. (1995): "Top-down Induction of Recursive Programs from Small Number of Sparse Examples". *Proceedings of the Fifth International Workshop on Inductive Logic Programming*. Ed. L. De Raedt. Scientific Report. Departement of Computer Science, K.U. Leuven.
[75] Morik, K., Potamias, G. and Moustakis, V. (1993): "Knowledgeable Learning Using MOBAL - A Case Study in A Medical Domain". *Real-World Applications of Machine Learning*, Workshop notes. Ed. Y. Kodratoff, P. Langley. ECML-93, Viena.
[76] Morik, K., Wrobel, S., Kietz, J. and Emde, W. (1993): *Knowledge Acquisition and Machine Learning: Theory Methods and Applications.* Academic Press.
[77] Muggleton, S. (1994): "Inverting Implication". Versão preliminar.
[78] Muggleton, S. (1992): "Inductive Logic Programming". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press. Também em *Proceedings of the First International Conference on Algorithmic Learning Theory*, Ohmsha, Tokyo, 1990.
[79] Muggleton, S. (1993): "Inductive Logic Programming: derivations, successes and shortcomings". *Proceedings of ECML-93*. Springer-Verlag.
[80] Muggleton, S. (1995): "Inverse Entailment and Progol". *New Generation Computing Journal*, vol. 13, May 1995.
[81] Muggleton, S. (1995): "Stochastic Logic Programs: extended abstract". *Proceedings of ILP-95*. Ed. Luc De Raedt. Scientific Report. Katholiek Universiteit Leuven.
[82] Muggleton, S., Feng, C. (1990): "Efficient Induction of Logic Programs". *Proceedings of the 1st Conference on Algorithmic Learning Theory*, Ohmsha, Tokyo.
[83] Muggletton, S., De Raedt, L., (1994): "Inductive Logic Programming". *The Journal of Logic Programming, special issue Ten Years of Logic Programming*. Vol. 19,20, May/July 1994.
[84] Muggleton, S., Mizoguchi, F. and Furukawa, K. (1995): Preface of the Special Issue on Inductive Logic Programming, *New Generation Computing*. Vol. 13, Nos. 3,4. Springer-Verlag.
[85] Muggleton, S., King, R., Sternberg, M. (1992): "Protein secondary structure prediction using logic". *Proceedings of the 2nd International Workshop on ILP*. Muggleton, S. (Ed.). Report ICOT-TM 1182, pp. 228-259.
[86] Muggleton, S., King, R., Sternberg, M. (1992): "Protein secondary structure prediction using logic". *Protein Engineering*. 7:647-657.

[87] Nienhuys-Cheng, S-H., de Wolf, R. (1995): "Least Generalizations and Greatest Specializations of Sets of Clauses". *Journal of Artificial Intelligence Research*. Volume 4, pp 341-363..

[88] O'Keefe, R. (1990): *The Craft of Prolog*. MIT Press.

[89] Olsson, R. (1995): "Inductive Functional Programming Using Incremental Program Transformation". *Artificial Intelligence* 74, pp 55-81. Elsevier.

[90] Paakki, J., Gyimóthy, T. and Horváth, T. (1994): "Effective Algorithm Debugging for Inductive Logic Programming". *Proceedings of the Fourth International Workshop on Inductive Logic Programming (ILP-94)*. GMD-Studien Nr. 237. GMD, Alemanha.

[91] Pereira, L. M. and Calejo, M. (1989): "Algorithmic Debugging of Prolog Side-Effects". *Proceedings of EPIA 89*, ed. by J. P. Martins and E. Morgado, Lecture Notes in Artificial Intelligence, Springer-Verlag.

[92] Plotkin, G. (1969): "A note on inductive generalization". *Machine Intelligence 5*. Ed. B. Meltzer, D. Michie. Edinburgh University Press.

[93] Plotkin, G. (1971): "A further note on inductive generalization" . *Machine Intelligence 6*. Ed. B. Meltzer, D. Michie. Edinburgh University Press.

[94] Popelínský, L., Flener P, Stepánková O, (1994) "ILP and Automatic Programming: Towards Three Approaches". *Proceedings of the 4th International Workshop on Inductive Logic Programming*. Volume 237, GMD-Studien.

[95] Popelinský, L., Stpánková, O. (1995): "WiM: A study on top-down ILP program". *Proceedings of AIT'95 Workshop*. Brno. ISBN 80-214-0673-9.

[96] Quinlan, J. R. (1990): "Learning Logical Definitions from Relations". *Machine Learning 5(3)*, pp.239-266.

[97] Quinlan, J. R. and Cameron-Jones, R. M. (1993): "FOIL: A Midterm Report". *Proceedings of the European Conference on Machine Learning ECML-93*. Ed. P. Brazdil. Springrer-Verlag.

[98] Quinlan, J. R. and Cameron-Jones, R. M. (1995): "Induction of Logic Programs: FOIL and related systems". *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4).

[99] Rich, C. and Waters, R. (Eds.) (1986): *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann.

[100] Richards, B., Mooney, R. (1995): "Refinement of First-Order Horn Clause Domain Theories". *Machine Learning*,Vol.19, No.2. Kluwer Academic Publishers.

[101] Richards, B., Mooney, R. (1992): "Learning relations by pathfinding". *Proceedings of the Tenth National Conference on Artificial Intelligence*. MIT Press.

[102] Rouveirol, C. (1992): "Extensions of Inversion of Resolution Applied to Theory Completion". *Proceedings of the 1st International Workshop on Inductive Logic Programming*. Technical Report. LIACC, Universidade do Porto.

[103] Rubinstein, R. (1981): *Simulation and the Monte Carlo Method*. John Wiley & Sons.

[104] Russel, S., Subramanian, G. (1990): "Mutual Constraints on Representation and Inference". *Machine Learning, Meta-Reasoning and Logics*. Ed. P. Brazdil, K. Konolige. Kluwer Academic Press.

[105] Sadri, F., Kowalski, R. (1988): "A Theorem Proving Approach to Database Integrity". *Deductive Databases and Logic Programming*. Ed. Jack Minker. Morgan Kaufmann Publishers.

[106] Sammut, C. (1993): "The Origins of Inductive Logic Programming: A Prehistoric Tale". *Proceedings of ILP-93*. Technical report. Jozef Stefan Institute.

[107] Sammut, C., Banerji, R. (1986): "Learning Concepts by asking Questions". *Machine Learning: An Artificial Intelligence Approach*, Vol. 2. Ed. R. Michalski, J. Carbonell, T. Mitchel. Morgan Kaufmann.

[108] Sernadas, A., Sernadas, C. and Costa, J.F. (1995): "Object-Specification Logic". *Journal of Logic and Computation* 5(5), pp. 603-630.

[109] Shapiro, E. Y., (1982) *Algorithmic Program Debugging*. MIT Press.

[110] Silverstein, G. and Pazzani, M. (1989): "Relational Clichés: constraining constructive induction during relational learning". *Proceedings of the Sixth International Workshop on Machine Learning*, Evanston, Illinois. Kaufmann.

[111] Smith, D. R., (1990): "KIDS: A Semiautomatic Program Development System". *IEEE Trans. on Software Engineering*, Vol. 16 No. 9.

[112] Solomonoff, R. J. (1964): "A formal Theory of Inductive Inference, Part I". *Information and Control*, 7, pp. 1-22.

[113] Sommerville, I., (1989): *Software Engineering* (third edition). Addison-Wesley Publishers Ltd.

[114] Stahl, I. (1993): "Predicate Invention in ILP - an Overview". *Proceedings of ECML-93*. Ed. P. Brazdil. Springer-Verlag.

[115] Stahl, I. (1995): "The Appropriateness of Predicate Invention as Bias Shift Operation in ILP". *Machine Learning*, 20. pp.95-117.

[116] Sterling, L., Shapiro, E. Y., (1986) *The Art of Prolog: Advanced Programming Techniques*. MIT Press.

[117] Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I., (1994): "Deductive composition of Astronomical Software from Subroutine Libraries". *Proceedings of the Twelfth International Conference on Automated Deduction*. Nancy, France. A. Bundy (Ed.). LNAI 814. Springer Verlag.

[118] Summers, P. (1977): "A Methodology for LISP Program Construction from Examples". *Journal of the Association for Computing Machinery*, Vol. 24, No. 1.

[119] Tausend, B. (1994): "Representing Biases for Inductive Logic Programming". *Proceedings of ECML-94*. Ed. F. Bergadano, L. De Raedt. Springer-Verlag.

[120] Tausend, B. (1994): "Biases and Their Effects in Inductive Logic Programming". *Proceedings of ECML-94*. Ed. F. Bergadano, L. De Raedt. Springer-Verlag.

[121] Ullman, J. (1989): *Principles of Databases and Knowledge Base Systems*. Volumes I e II. Computer Science Press.

[122] Vere, S. (1977): "Induction of Relational Productions in the Presence of background Knowledge". *Proceedings of the Fifth International Joint Conference in Artificial Intelligence*.

[123] Wirth, R. and O'Rorke (1992): "Constraints for predicate invention". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[124] Wrobel, S. (1994): *Concept Formation and Knowledge Revision*. Kluwer Academic Publishers.

[125] Zelle, J. M., Mooney, R. J., Konvisser, J. B., (1994): "Combining Top-down and Bottom-up Techniques in Inductive Logic Programming". *Proceedings of the Eleventh International Conference on Machine Learning ML-94*, Morgan-Kaufmann.

Anexos

Apêndice A

Predicados do conhecimento de fundo *list*:

*mode(**const**(+,+,-)).*
type(const(list,int,list)).

const(A,B,C)←A=[B|C].

*mode(**dest**(+,-,-)).*
type(dest(list,int,list)).

dest(A,B,C)←A=[B|C].

*mode(**null**(+)).*
type(null(list)).

null([]).

*mode(**addlast**(+,+,-)).*
type(addlast(list,int,list)).

addlast([],X,[X]).
addlast([A|B],X,[A|C])←
addlast(B,X,C).

*mode(**appendb**(+,+,-)).*
type(appendb(list,list,list)).

appendb([],A,A).
appendb([A|B],C,[A|D])←
appendb(B,C,D).

% O predicado *append/3* tem uma definição equivalente a *appendb/3*.

*mode(**delete**(+,+,-)).*
type(delete(elem,list,list)).

```
delete(A,[A|B],B).
delete(A,[B|C],[B|D])←
        delete(A,C,D).

mode(last_of(-,+)).
type(last_of(elem,list)).

last_of(A,[A]).
last_of(A,[C|D])←last_of(A,D).

mode(memberb(-,+)).
type(memberb(elem,list)).

memberb(A,[A|B]).
memberb(A,[B|C])←
        memberb(A,C).
```

% O predicado *member/2* tem uma definição equivalente a *memberb/2*.

```
mode(notmember(+,+)).
type(notmember(elem,list)).

notmember(A,B)←memberb(A,B).

mode(partb(+,+,-,-)).
type(partb(elem,list,list,list)).
```

% O predicado *partition/42* tem uma definição equivalente a *partb/4*.

```
partb(A,[B|C],[B|D],E)←
        A>B,partb(A,C,D,E).
partb(A,[B|C],D,[B|E])←
        A=<B,partb(A,C,D,E).
partb(A,[],[],[]).

mode(rv(+,-)).
type(rv(list,list)).

rv( [], []).
rv( [C|D], B) ←
   rv(D,E),
   addlast(E,C,B).
```

```
mode(singleton(+)).
type(singleton(list)).

singleton([X]).

mode(sort(+,-)).
type(sort(list,list)).

sort([A|B],C)←
        part(A,B,E,F),
        sort(E,G),sort(F,H),
        append(G,[A|H],C).
sort([],[]).

mode(split(+,-,-)).
type(split(list,list,list)).

split([],[],[]).
split([A,B|D],[A|E],[B|F])←split(D,E,F).

mode(union(+,+,-)).
type(union(list,list,list)).

union([],A,A).
union([A|B],C,D)←
        member(A,C),!,
        union(B,C,D).
union([A|B],C,[A|D])←
        union(B,C,D).

mode(insertb(+,+,-)).
type(insertb(int,list,list)).

insertb(A,[],[A]).
insertb(A,[B|C],[A,B|C])←A<B.
insertb(A,[B|C],[B|D])←B<A,insertb(A,C,D).
```

Predicados do conhecimeno de fundo *integer*:

```
mode(pred(+,-)).
type(pred(int,int)).

pred(X,Y)←Y is X-1,X>0.
```

*mode(**succ**(+,-)).*
type(succ(int,int)).

succ(X,Y)←Y is X+1.

*mode(**zero**(+)).*
type(zero(int)).

zero(0).

*mode(**one**(+)).*
type(one(int)).

one(1).

*mode(**plus**(+,+,-)).*
type(plus(int,int,int)).

plus(X,Y,Z)←Z is Y+X.

*mode(**multb**(+,+,-)).*
type(multb(int,int,int)).

*multb(X,Y,Z)←Z is X*Y.*

% O predicado *multiply/3* é equivalente a *multb/3*.

*mode(**even**(-)).*
type(even(peano)).

even(0).
even(s(s(X)))←even(X).

*mode(**odd**(+)).*
type(odd(peano)).

odd(s(0)).
odd(s(s(X)))← odd(X).

Outros predicados:

***sorted**([]).*

```
sorted([A]).
sorted([A,B|C])←
        A=<B,
        sorted([B|C]).

sublist(A,B)←
        prefix(A,B).
sublist(A,[B|C])←
        sublist(A,C).

prefix([],A).
prefix([A|B],[A|C])←
        prefix(B,C).
```

Apêndice B

Definições de tipos:

```
int(X)←member(X,[0,1,2,3,4,5,6,7,8,9]).

letter(X)←member(X,[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).

list([]).
list([A|B])←int(A),list(B).

peano(0).
peano(s(X))←peano(X).
```

Apêndice C

‘decomp_test_rec_comp_2’

```
body(P)-->decomp(+,2),test(*,2),recurs(*,2,P),comp(*,2).
body(P)-->test(+,2),comp(*,2).

decomp(_,N)-->lit_decomp,{N>0}.
decomp(_,N)-->lit_decomp,{N2 is N-1},decomp(+,N2).
decomp(*,N)-->[].
```

lit_decomp-->[dest/3];[pred/2];[partb/4].

test(_,N)-->lit_test,{N>0}.
test(_,N)-->lit_test,{N2 is N-1},test(+,N2).
test(,N)-->[].*

lit_test-->[null/1];[memberb/2];[notmemberb/2];[zero/1].

comp(_,N)-->lit_comp,{N>0}.
comp(_,N)-->lit_comp,{N2 is N-1},comp(+,N2).
comp(,N)-->[].*

lit_comp-->[appendb/3];[insertb/3];[addlast/3];[const/3].

recurs(_,N,P)-->lit_recurs(P),{N>0}.
recurs(_,N,P)-->lit_recurs(P),{N2 is N-1},rec(+,N2,P).
recurs(,N,P)-->[].*

lit_recurs(P)-->[P].

‘decomp_+test_rec_comp_2’

body(P)-->decomp(+,2),test(+,2),rec(,2,P),comp(*,2).*
body(P)-->test(+,2),comp(,2).*

No restante idêntica a decomp_test_rec_comp_2.dcg.

‘decomp_test_rec1_comp_2’

body(P)-->decomp(+,2),test,2),rec(*,1,P),comp(*,2).*
body(P)-->test(+,2),comp(,2).*

No restante idêntica a decomp_test_rec_comp_2.dcg.

Traduções

Abreviaturas:

AA: Aprendizagem Automática

ES: Engenharia de 'software'

IA: Inteligência Artificial

L: Lógica

PL: Programação Lógica

PLI: Programação Lógica por Indução

SP: Síntese de Programas

***θ*-equivalente** : (PLI) *θ*-equivalent. v. *subsunção-θ.*
***θ*-subsume** : (PLI) *θ*-subsumes. v. *subsunção-θ.*
aprendido, programa : (AA) v. *programa sintetizado*.
aprendizagem : v. *aprendizagem automática*.
aprendizagem automática : machine learning.
aprofundamento iterativo: (IA) iterative deepening.
argumentos de entrada : (PL) input arguments.
argumentos de saída : (PL) output arguments
aridade : (PL) arity.
átomo : (PL) atom.
base de Herbrand : (PL) Herbrand base.
cabeça de cláusula : (PL) clause head.
CASE : (ES) Sigla, Computer Aided Software Engineering.
circunscrição : (AA) bias
circunscrição de linguagem : (AA) language bias. Traduções alternativas: *viés de linguagem*, *pressupostos de linguagem.*
cláusula de Horn : (PL) Horn clause.
cláusula definida : (PL) definite clause..
cláusula fechada : (PL) ground clause.
cláusula maximamente específica : (PLI) maximally specific clause
cláusula maximamente geral : (PLI) maximally general clause.
cláusula recursiva : (PL) recursive clause.
cláusula reduzida : (PL) reduced clause.
cobertura : (AA) coverage.
cobertura extensional : (PLI) extensional coverage.
cobertura intensional : (PLI) intensional coverage.
cobertura, estratégia de : (AA) covering strategy.
compatível, tipo : (PL) compatible type.
completamento de um programa : (PL) completion, Comp(P).
completude: (L) completeness.
conhecimento de fundo : (AA) background knowledge.
conjunto de teste : (AA) test set.
conjunto de treino : (AA) training set.

conjunto representativo básico : (PLI) basic representative set.
consequência lógica : (L) logical consequence.
consolidação de um esboço : (PLI) skectch consolidation
corpo de cláusula : (PL) clause body.
declaração de modo (de entrada/saída) : (PLI) input/output mode declaration.
declaração de tipo : (PLI) type declaration.
depuração de programas : (PL) program debugging.
determinações : (PLI) determinations.
esboço de algoritmo : (PLI) algorithm sketch.
esboço mínimo : (PLI) black box sketch.
esboço operacional : (PLI) operational sketch.
esboço sintacticamente ordenado : (PLI) syntactically ordered sketch.
especialização mais geral : (AA) most general specialization.
especificação : (SP) specification.
especificação completa : (SP) complete specification.
especificação formal : (SP) formal specification.
especificação incompleta : (SP) incomplete specification.
especificação informal : (SP) informal specification.
esquema de cláusula : (PLI) clause scheme.
exemplo fechado : (PLI) ground example.
exemplo negativo : (AA) negative example.
exemplo positivo : (AA) positive example
extensional, conhecimento de fundo :(PLI) extensional background knowledge.
facto : (PL) fact.
ferramentas CASE :(ES) CASE tools.
functor :(PL) functor.
gananciosa, método de procura : (IA) greedy search method.
generalização menos geral : (PLI) least general generalization. Usa-se frequentemente a sigla lgg.
gerador de código : (ES) code generator. v. *ferramentas CASE*.
globalmente completo : (PLI) globally complete.
grafo de dependências : (PLI) dependence graph.
gramática de cláusulas definidas : (PL) definite clause grammar. Usa-se a sigla DCG.
gramática de estrutura clausal : (PLI) clause structure grammar. Usa-se a sigla GEC.
implicação-T : (PLI) T-implication.
inclusão generalizada : (PLI) generalized subsumption.
subsunção-θ : (PLI) θ-subsumption.
indução : (AA) induction.
indução iterativa : (PLI) bootstrap iterative induction.
induzido, programa : (PLI) induced program.
integridade: (L) soundness, integrity.
intensional, conhecimento de fundo : (PLI) intensional background knowledge.
linguagem de hipóteses : (AA) hypothesis language, concept language.
linguagem de muito alto nível : (ES) very high level language.
linguagem livre de functores : (PL) functor free language.
literal : (PL) literal.
literal de esboço : (PLI) sketch literal.
literal fechado : (PL) ground literal.
literal operacional : (PLI) operational literal.
localmente completo : (PLI) locally complete.
lógica de primeira ordem : (L) first order logic.
modelo de Herbrand : (PL) Herbrand model.

modelo de regras : (PLI) rule model.
modelo mínimo : (PL) minimal model.
modo de entrada/saída : (PL) input output mode.
operador de especialização : (AA) specialization operator.
operador de especialização óptimo : (AA) optimal specialization operator.
operador de generalização : (AA) generalization operator.
operador de refinamento : (PLI) refinement operator.
padrão de cláusulas : (PLI) clause template.
pergunta : (PL) query.
predicado de esboço : (PLI) sketch predicate.
procedimento de prova : (L) proof procedure.
procura de caminhos relacionais : (PLI) relational pathfinding.
procura descendente : (IA) top-down search.
procura em largura : (IA) breadth-first search.
procura heurística : (IA) heuristic search.
programa lógico : (PL) logic program.
programação automática : automatic programming.
programação extensiva : (SE) programming in the large.
programação lógica : logic programming. Usa-se a sigla PL.
programação lógica por indução : inductive logic programming. Usa-se a sigla PLI.
programação não extensiva : (ES) programming in-the-small.
propriedade : (PLI) property.
prova : (L) proof.
resolução-SLD : (PL) SLD-resolution.
restrição de integridade : (PL) integrity constraint.
retrocesso : (PL) backtracking.
semente, exemplo : (PLI) seed example.
síntese de programas : program synthesis.
síntese de programas lógicos : (PL) logic program synthesis.
síntese indutiva : inductive synthesis.
síntese indutiva a partir de exemplos : inductive synthesis from examples.
síntese indutiva de programas a partir de especificações incompletas : inductive program synthesis from incomplete specifications.
sintetizado, programa : synthesized program.
subida mais rápida : (IA) hill-climbing. Traduções alternativas: *trepa-colina*, *subida-da-colina*.
substituição : (PL) substitution.
substituição resposta : (PL) answer substitution.
termo : (PL) term.
termo fechado : (PL) ground term.
tipo : (PL) type.
unificador : (PL) unifier.
unificador mais geral : (PL) most general unifier.
universo de exemplos : (AA) universe of examples.
variável : (PL) variable.
vocabulário : (AA) vocabulary.

Lista de Figuras

Lista de Algoritmos

Lista de Exemplos

Lista de Definições

Índice Remissivo

—*D*—

—*E*—

—*F*—

—*G*—

—*H*—

—*I*—

—*L*—

—M—

—N—

—O—

—P—

—R—

—*S*—

—*T*—

—*U*—

—*V*—