Tutorial
Avaliação Parcial de Programas

Vladimir O. Di Iorio, Roberto S. Bigonha, Mariza A. S. Bigonha
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

1 Introdução

Suponha um programa P que tenha duas entradas, identificadas como in_1 e in_2. O resultado da *avaliação parcial* de P em relação à entrada in_1 é um novo programa P_{in_1}, designado por *residual* ou *especializado* [18]. O programa P_{in_1}, quando executado sobre a entrada restante in_2, produz o mesmo resultado que a execução de P sobre ambas as entradas. Avaliação parcial é um tipo de *especialização de programas*. A entrada in_1 é designada *entrada estática*, e in_2, *entrada dinâmica*.

O objetivo principal da avaliação parcial é o ganho em eficiência. Se parte dos dados de entrada de um programa é conhecida, as estruturas do programa que dependem apenas dessa parte podem ser previamente computadas. O programa especializado conterá apenas o código necessário para processar os dados ainda não conhecidos.

Um *avaliador parcial* para uma linguagem L é um programa que executa avaliação parcial sobre programas escritos em L, produzindo como resultado programas especializados que são escritos, geralmente, na mesma linguagem L. A Figura 1 apresenta um esquema de funcionamento de um avaliador parcial.

Um avaliador parcial realiza uma mistura de execução com geração de código, motivo pelo qual o processo foi designado "*mixed computation*", e o avaliador comumente chamado de `mix` [12, 20]. Avaliadores parciais já foram desenvolvidos com sucesso para linguagens de paradigma imperativo [1, 24], funcional [5, 17, 21] e lógico [30, 26].

Observe a função *Power(n,x)* na Figura 2(a), escrita na linguagem C, que computa o valor de x^n. A especialização de *Power* dado $n = 5$ é uma função com apenas um parâmetro x. Essa função deve produzir o mesmo resultado que *Power*, se for executada com entradas 5 e x, para qualquer valor de x.

Uma solução ingênua, mas que satisfaz a condição imposta acima, é exibida na Figura 2(b). De fato, o *Teorema s-m-n* de Kleene [18] mostra que sempre é possível obter um programa especializado, e sua prova exibe o projeto de um avaliador parcial. Esse teorema, entretanto, não se preocupa com questões de eficiência.

No exemplo em questão, o uso do valor estático n permite obter um código mais eficiente. Um avaliador parcial para a linguagem C deveria produzir um programa residual como o exibido na Figura 2(c), ao especializar a função Power com $n = 5$.

Este documento apresenta um tutorial sobre avaliação parcial de programas. Contém definições, exemplos, aplicações e tópicos de pesquisa. As principais referências para a construção desse texto foram o livro de Jones, Gomard e Sestoft [18], um relatório de Mogensen e Sestoft [27], e o material produzido por John Hatcliff para a Escola de Verão do DIKU de 1998 (*Partial Evaluation: Practice and Theory*).

2 Aplicações de Avaliação Parcial

A avaliação parcial de programas é utilizada em diversas áreas da computação. Para que sua utilização valha a pena, deve ser levado em consideração se o custo de produzir um programa especializado é compensado pelo ganho de velocidade na execução.

Se uma das entradas varia com freqüência bem menor que as outras, o custo de produção do programa especializado é diluído em um conjunto de execuções. Um exemplo onde isso acontece é o método usado em computação gráfica conhecido como *ray tracing*, para renderização de figuras. O algoritmo geral recebe duas entradas, uma cena e um raio de luz.. A avaliação parcial do algoritmo de *ray tracing* em relação a uma

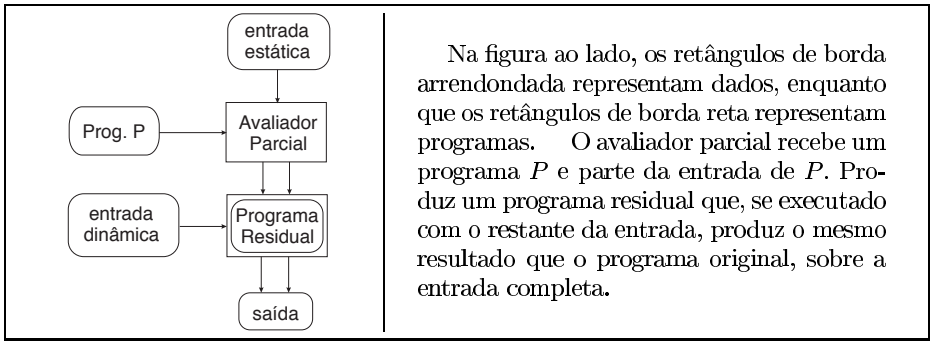


Figura1. Esquema da utilização de um avaliador parcial.

```
int Power (int n, int x) {
    int p = 1;
    while (n > 0)
        if (n%2 == 0) {
            x = x * x;
            n = n / 2;
        }
        else {
            p = p * x;
            n = n - 1;
        }
    return p;
}
```

(a) Função Power(n,x) = x^n.

```
int Power_5 (int x) {
    return Power (5, x);
}
```

(b) Especialização ingênua.

```
int Power_5 (int x) {
    int p = x;
    x = x * x;
    x = x * x;
    p = p * x;
    return p;
}
```

(c) Especialização eficiente.

Figura2. Exemplo de Avaliação Parcial.

cena específica resulta em um procedimento eficiente para traçar raios para aquela cena [3]. O tempo gasto para se construir o algoritmo especializado é compensado pela eficiência obtida, uma vez que o mesmo é utilizado inúmeras vezes para diferentes raios. Experiências realizadas por Mogensen mostraram um *ray tracer* especializado que era de 8 a 12 vezes mais rápido que o original [25].

Problemas que tenham natureza interpretativa constituem outra classe que pode se beneficiar imensamente da avaliação parcial. Esses problemas envolvem geralmente o uso de uma linguagem que permite ao usuário especificar o formato da entrada e parâmetros especiais. Alguns exemplos são: simulação de circuitos, redes neuronais, casamento de padrões e problemas cujas entradas são tabelas especificando transições.

Simuladores de circuitos recebem a descrição de um circuito elétrico como entrada, constroem equações diferenciais que descrevem seu comportamento e usam métodos numéricos para resolver essas equações. A especialização de um simulador geral em relação a um circuito específico gera um programa eficiente para esse circuito. Um ganho substancial em velocidade pode ser obtido [4].

O treinamento de redes neuronais geralmente é um processo muito caro, dispendendo um tempo longo de computação. Do ponto de vista da avaliação parcial, esse problema é semelhante ao anterior: um simulador geral pode ser especializado em relação a uma dada topologia de rede.

Um algoritmo de casamento de padrões em textos recebe como entradas uma seqüência de caracteres definindo um padrão e um texto onde

esse padrão deverá ser pesquisado. A avaliação parcial do algoritmo em relação a um padrão específico resulta em um programa que implementa um autômato finito determinístico que executa as ações necessárias para a identificação desse padrão em qualquer texto. Uma experiência interessante de Consel e Danvy mostra a geração de um algoritmo equivalente ao método de Knuth, Morris e Pratt (KMP) a partir de algoritmo de casamento de padrões ingênuo (força bruta) [9].

Análise léxica e sintática podem ser implementadas usando-se algoritmos que seguem tabelas de transições. Essas tabelas associam, a um dado estado e um dado caractere de entrada, um novo estado e uma ação semântica correspondente. A especialização desses algoritmos em relação a tabelas específicas resulta na "compilação" da tabela diretamente para um código executável, muito mais rápido.

Finalmente, um dos usos mais importantes de avaliação parcial é na compilação e geração de compiladores dirigida por semântica. Essa aplicação é discutida em detalhe na Seção 9.

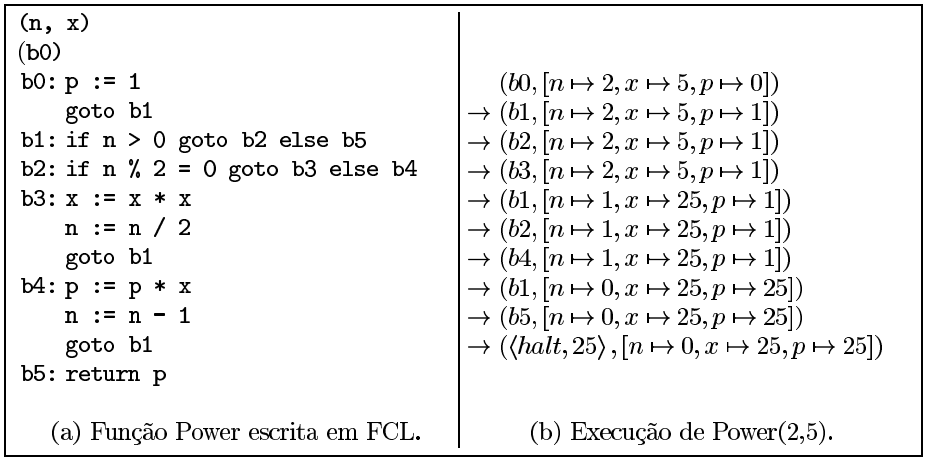
3 Linguagem de Fluxograma FCL

Para apresentar as técnicas empregadas na avaliação parcial de programas, vamos utilizar uma linguagem de fluxograma designada FCL (*Flowchart Language*) [14]. Sendo bastante simples, essa linguagem é ideal para apresentarmos os conceitos básicos de avaliação parcial.

A linguagem FCL é uma linguagem simples de fluxograma que contém apenas comandos de atribuição, desvio condicional e incondicional, e retorno de valores. O código é dividido em blocos básicos, identificados por rótulos.

Um programa FCL inicia-se com a definição dos parâmetros de entrada, seguida do rótulo do primeiro bloco a ser executado, seguido de uma série de blocos básicos, onde cada um possui uma única entrada e única saída. Os blocos são constituídos por uma seqüência de comandos de atribuição possivelmente vazia, seguida de exatamente um comando de desvio ou comando `return`. A cada bloco está associado um rótulo diferente.

Os comandos de atribuição têm o formato `v := exp`, onde `v` é uma variável e `exp` uma expressão contendo operações que envolvem variáveis e constantes. Um comando de desvio pode ser incondicional (`goto label`) ou condicional (`if boolexp goto label1 else label2`). Para retornar o valor de uma expressão calculada, é usado o comando `return exp`, que termina a execução do programa. Todas as variáveis não pertencentes à



(a) Função Power escrita em FCL.

(b) Execução de Power(2,5).

Figura3. Exemplo de Programa FCL.

entrada são inicializadas com zero. A Figura 3(a) mostra a codificação em FCL da função Power, exibida na Figura 2(a).

Observe que um comando de desvio é obrigatório ao final de todo bloco que não termine com o comando `return`, mesmo que o desvio seja para o bloco subseqüente no código. A execução do programa sempre é finalizada com um comando `return`.

A execução de um programa FCL pode ser representada por uma seqüência de estados. Cada estado pode ser representado por um par (l, σ), onde l é um ponto do programa indicado por um rótulo e σ é o valor das variáveis antes da execução do comando indicado por l. A execução da função Power da Figura 3(a), com $n = 2$ e $x = 5$, é apresentada na Figura 3(b). Um novo rótulo, não existente no programa, foi introduzido para representar o término da execução e retorno de um valor: $\langle halt, 25\rangle$. Nessa representação escolhida para execuções de FCL, o próximo comando a ser executado em um estado é sempre o primeiro comando de um bloco básico.

4 Especialização Polivariante

A principal técnica de avaliação parcial é conhecida como *especialização polivariante*, sendo utilizada em linguagens de diversos paradigmas. O programa é visto como um grafo, onde os vértices são *pontos de programa*, conectados por *arestas de fluxo de controle*. Na linguagem FCL, os pontos de programa e as arestas de fluxo de controle são, respectivamente, os

blocos básicos rotulados e os *desvios*; em uma linguagem funcional, eles seriam as *funções definidas* e as *chamadas de funções*.

Essa técnica é chamada de especialização polivariante porque um único ponto de programa do programa original pode dar origem a vários pontos de programa no programa especializado. Para entender o processo, considere como a execução exibida na Figura 3(b) deveria ser alterada se apenas parte dos dados de entrada fosse fornecida. Intuitivamente, podemos prever que alguns dos valores das variáveis representadas em cada estado seriam desconhecidos. Algumas das expressões, atribuições, e desvios poderiam ser computados, se dependessem apenas dos dados conhecidos, chamados de *entrada estática*. Outras estruturas do programa não poderiam ser computadas, se dependessem dos valores não conhecidos, chamados de *entrada dinâmica*.

O processo procura estabelecer todos os estados alcançáveis, utilizando apenas os dados estáticos. Em seguida, constrói os blocos básicos do programa especializado. Cada bloco é derivado de um estado alcançado: o bloco construído a partir de (l, σ) é o resultado da computação de toda informação estática do bloco l, utilizando os valores conhecidos que aparecem em σ.

Podemos ver agora que o formato escolhido para a representação da execução de um programa FCL é adequada ao processo descrito acima. Cada estado da execução está associado ao início de um bloco básico, coincidindo com os pontos de programa utilizados no método de especialização polivariante.

Resumindo, o processo é composto de três passos, sendo que a maioria dos avaliadores parciais executa esses três passos em paralelo:

1. Determinação de todos os estados alcançáveis *(reachable states)*: começando pelo estado inicial (l_0, σ_0), onde σ_0 contém apenas os dados conhecidos, reunir todos os estados alcançáveis no conjunto S.
2. Especialização dos pontos de programa *(program point specialization)*: para cada $(l_i, \sigma_i) \in S$, inserir no programa residual um novo bloco, resultado da especialização de l_i em relação aos valores de σ_i.
3. Compressão das transições *(transition compression)*: otimização do programa residual por meio da fusão de alguns blocos; blocos compostos por apenas um desvio incondicional estão entre os candidatos a serem eliminados.

Como visto, o processo inicia com a divisão dos dados de entrada em conhecidos (estáticos) e não conhecidos (dinâmicos). Para determinar se os objetos restantes do programa são estáticos ou dinâmicos, existem

ainda duas abordagens diferentes: métodos *online* e *offline*. Nos métodos *online*, os valores são classificados como estáticos ou dinâmicos durante a especialização, enquanto que, nos métodos *offline*, uma análise do programa é feita antes de iniciar o processo de especialização, determinando a divisão. A seguir, discutimos essas duas abordagens e fornecemos exemplos da aplicação da especialização polivariante usando cada uma delas.

Nas seções seguintes, vamos utilizar as seguintes notações:

$[y_o \ldots y_j]$ representa uma lista de tamanho j, com elementos y_o, ... y_j.
$(x_o x_1 \ldots x_i)$ representa uma tupla de i componentes; o k-ésimo componente, $0 \leq k \leq i$, é x_k.

5 Métodos *Online*

Nas técnicas *online*, a avaliação parcial é geralmente executada em uma única fase, e os valores são definidos como estáticos ou dinâmicos durante a especialização. Nesta seção, vamos mostrar a abordagem *online* por meio de um exemplo, seguindo os três passos da especialização polivariante. Como dissemos antes, esses três passos são comumentes executadas em paralelo. Optamos por mostrar a execução de cada passo separadamente nos primeiros exemplos, por questões didáticas.

5.1 Determinação dos Estados Alcançáveis

Para demonstrar o processo, vamos voltar à execução da função Power, exibida na Figura 3(b). Desta feita, vamos considerar apenas que o valor da entrada estática n é 2, mas o valor de x não é conhecido. O símbolo D vai ser utilizado para representar o valor da entrada dinâmica x. Sendo assim, o estado inicial terá $b0$ como ponto de programa e $[n \mapsto 2, x \mapsto D, p \mapsto 0]$ como valor das variáveis. O valor inicial de p, embora não seja uma entrada conhecida, será 0, que é o valor *default* das variáveis. Apenas as entradas dinâmicas são inicializadas com D.

Mais tarde, quando apresentarmos um algoritmo para especialização *online*, veremos que é necessário elaborar um pouco mais essa representação: o valor de uma variável será denotado por um par $(type, val)$, onde *type* será D ou S, indicando valor dinâmico ou estático, e *val* será o valor real da variável, se for estática.

A execução da função Power, com valor n igual a 2, é exibida na Figura 4. O que acontece a cada passo:

Passo 1: No bloco $b0$, o valor de p foi aterado para 1, e o próximo ponto de programa calculado é $b1$.

$(b0, [n \mapsto 2, x \mapsto D, p \mapsto 0])$	passo 5: $(b2, [n \mapsto 1, x \mapsto D, p \mapsto 1])$
passo 1: $(b1, [n \mapsto 2, x \mapsto D, p \mapsto 1])$	passo 6 $(b4, [n \mapsto 1, x \mapsto D, p \mapsto 1])$
passo 2: $(b2, [n \mapsto 2, x \mapsto D, p \mapsto 1])$	passo 7: $(b1, [n \mapsto 0, x \mapsto D, p \mapsto D])$
passo 3: $(b3, [n \mapsto 2, x \mapsto D, p \mapsto 1])$	passo 8: $(b5, [n \mapsto 0, x \mapsto D, p \mapsto D])$
passo 4: $(b1, [n \mapsto 1, x \mapsto D, p \mapsto 1])$	passo 9: $(halt, [n \mapsto 0, x \mapsto D, p \mapsto D])$

Figura4. Execução de Power com Entrada Parcialmente Conhecida.

Passo 2: Como n é conhecido, a condição do desvio pôde ser computada, definindo que o próximo ponto de programa é $b2$.

Passo 3: De maneira similar ao passo anterior, o próximo ponto de programa calculado é $b3$.

Passo 4: O valor de n é alterado para 1, mas como x não é conhecido, $x * x$ não pôde ser calculado.

Passo 5: De maneira equivalente ao passo 2, o próximo ponto de programa é $b2$.

Passo 6: Como o valor de n é ímpar, o próximo ponto de programa é $b4$.

Passo 7: A expressão $p*x$ não pôde ser calculada, pois x não é conhecido, assim o valor de p é alterado para D. O valor de n é alterado para 0.

Passo 8: A condição do desvio falhou desta vez, assim o próximo ponto de programa é $b5$.

Passo 9: Fim do programa. O valor de retorno não pôde ser calculado.

A execução do programa baseada nos valores conhecidos determina todos os estados alcançáveis a partir do estado inicial. Dois aspectos que valem a pena ser mencionados não ocorrem no exemplo mostrado nesta seção:

- Se for gerado um estado que já foi produzido anteriormente na execução, o algoritmo de determinação dos estados alcançáveis não precisa repetir todo o processo. Esse caso irá corresponder a um *loop* no programa residual.
- No exemplo exibido, todos os testes dos desvios condicionais dependiam apenas dos valores estáticos, assim foi sempre possível identificar qual seria o próximo bloco a ser executado. Se a condição depender de valores dinâmicos, não teremos como computá-la. Nesse caso, o algoritmo deve gerar os estados alcançáveis levando em conta os dois blocos referenciados no desvio condicional.

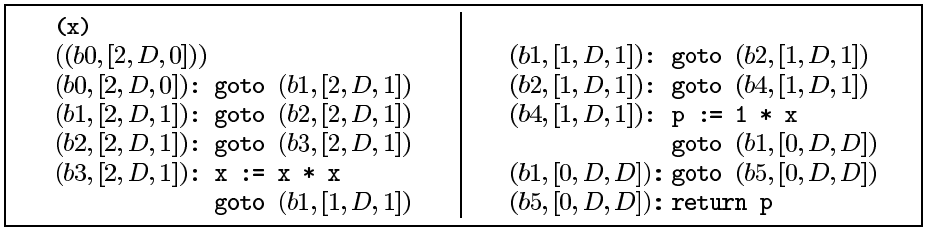


Figura5. Programa Residual, sem Compressão de Transições.

5.2 Especialização dos Pontos de Programa

Na execução exibida na Seção 5.1, pode-se ver que um mesmo ponto de programa aparece mais de uma vez, com diferentes valores para as variáveis. Por exemplo, $b1$ aparece com os seguintes valores para as variáveis:
$[n \mapsto 2, x \mapsto D, p \mapsto 1]$, $[n \mapsto 1, x \mapsto D, p \mapsto 1]$, $[n \mapsto 0, x \mapsto D, p \mapsto D]$.

Cada instância irá gerar um bloco básico especializado diferente no programa residual. Blocos diferentes recebem rótulos diferentes, assim os rótulos serão também especializados. Por motivo de simplicidade, vamos utilizar o par (l_i, σ_i) para denotar cada rótulo diferente. Por exemplo, a primeira instância do bloco $b1$ terá um rótulo como o seguinte: $(b1, [2, D, 1])$. Os rótulos referenciados nos comandos de desvio condicional e incondicional deverão também ser alterados.

O programa residual, antes das otimizações serem conduzidas, é exibido na Figura 5. A seguir, exibimos uma explicação de como o código de cada um dos blocos especializados foi obtido:

$(b0, [2, D, 0])$: A atribuição `p := 1;` não depende de nenhuma informação dinâmica, assim pode ser completamente computada. A instrução `goto b1` vai aparecer no código residual, especializada em relação ao novo valor das variáveis, ou seja, $[2, D, 1]$.

$(b1, [2, D, 1])$: Sempre que a condição de um desvio puder ser totalmente computada, ela não precisa aparecer no código residual. Além disso, a informação pode ser usada para decidir qual dos dois blocos especializar. No caso deste bloco, a condição é verdadeira e um desvio especializado para o bloco $b2$ é inserido no código residual.

$(b2, [2, D, 1])$: De forma semelhante ao caso anterior, um desvio especializado para o bloco $b3$ é inserido no código residual.

$(b3, [2, D, 1])$: Sempre que um dos valores de uma operação não for conhecido, a operação é classificada como D, ou seja, dinâmica. A variável x tem valor D, assim a expressão $x * x$ é classificada também como

D. Portanto, `x := x*x;` deve aparecer no código residual. Por outro lado, na operação $n/2$, os valores envolvidos são uma variável estática e uma constante, assim a expressão pode ser calculada. E como n é estática, a atribuição `n := n/2;` vai ser computada, não aparecendo no código residual. Finalmente, um desvio especializado é produzido.

$(b1, [1, D, 1])$**:** Similar a $(b1, [2, D, 1])$.

$(b2, [1, D, 1])$**:** Similar a $(b2, [2, D, 1])$.

$(b4, [1, D, 1])$**:** A operação $p * x$ é classificada como D, pois x é D. Entretanto, a residualização envolve o cálculo dos componentes estáticos da operação, nesse caso, a variável p. O resultado é a expressão $1 * x$. Sempre que um valor estático aparece em um contexto dinâmico, aplicamos uma operação conhecida como *lift*. Dizemos que o valor estático 1, calculado a partir de p, foi *lifted* para aparecer no código residual. Na atribuição `p := p * x`, a expressão do lado direito foi classificada como D, assim p passa também a ser D e a atribuição é residualizada. O resto do bloco é simples, consistindo do cálculo do novo valor de n e a residualização do desvio. O valor das variáveis agora é $[0, D, D]$.

$(b1, [0, D, D])$**:** Similar a $(b1, [2, D, 1])$ e $(b1, [1, D, 1])$.

$(b5, [0, D, D])$**:** A expressão p é classificada como D. Um comando `return` deve sempre ser residualizado. Se a expressão a ser retornada for estática, deve sofrer *lift* para aparecer no código residual.

5.3 Compressão das Transições

O programa produzido na Seção 5.2 contém vários blocos terminados por um desvio incondicional, alguns deles formados apenas pelo desvio. Esse fato ocorre com muita freqüência quando se utiliza o método de especialização polivariante, pois muitos comandos de blocos básicos do programa original não aparecem no programa residualizado.

A compressão de transições consiste em substituir um desvio para um rótulo *pp* pelo código do bloco indicado por *pp*. Os benefícios são evidentes: o programa pode ficar bem mais eficiente com a eliminação de muitos desvios no código. Se aplicarmos a compressão de transições no programa residual gerado na Seção 5.2, obteremos o resultado a seguir, onde os rótulos foram renomeados:

```
(x)
(b0)
b0: x := x * x
    p := 1 * x
    return p
```

Se aplicarmos a compressão de transições indiscriminadamente podemos ter dois tipos de problemas: duplicação de código e compressão infinita. A duplicação de código ocorre quando a compressão é aplicada a duas transições distintas para um mesmo ponto de programa. Quando o programa residual contém um *loop*, a compressão pode continuar indefinidamente.

Se a compressão de transições é executada como uma fase separada, após a geração do programa residual, a duplicação de código e compressão infinita podem ser mais facilmente evitadas. Um grafo de fluxo de controle do programa residual pode ser construído e uma análise pode ser conduzida de forma a identificar as compressões seguras. Entretanto, a especialização polivariante gera com freqüência inúmeros blocos contendo desvios suplérfluos, assim seria mais eficiente conduzir a compressão de transições durante o processo de especialização, ou seja, *transition compression on the fly*.

Executar a compressão de transições durante a especialização pode tornar bem mais difícil a tarefa de identificar as compressões seguras. Uma estratégia simples é usar a compressão em todas as transições que não façam parte de um desvio condicional residual. Na realidade, isso seria o máximo que conseguiríamos com a linguagem FCL, uma vez que um desvio condicional não pode conter comandos a serem executados, diferente da estrutura de blocos usada pela maioria das linguagens imperativas.

A estratégia descrita ainda pode gerar duplicação de código, mas experiências relatadas indicam que é um problema mínimo [18]. O mais importante é que essa estratégia não produz uma compressão infinita, a não ser que o programa residual contenha um *loop* infinito que dependa apenas da entrada estática. Nesse caso, o programa original também entraria em *loop* infinito com os mesmos valores estáticos, independente dos valores dinâmicos utilizados.

5.4 Especialização Online: Três Passos em Paralelo

A Figura 6 exibe um algoritmo para realizar especialização de programas FCL, que na Seção 1 chamamos de `mix`, executando os três passos em paralelo. O algoritmo tem como entradas o programa original `program`, o rótulo $\texttt{pp}_0$ do bloco inicial do programa e o valor inicial $\texttt{vs}_0$ das variáveis.

O valor de cada variável é representado por um par $(type, val)$, onde $type$ é S (estático) ou D (dinâmico). Se a variável for dinâmica, val contém uma expressão representando a própria variável; se for estática, val contém a constante associada. No exemplo desenvolvido nesta seção,

```
pending := {(pp0,vs0)};
marked := {};
while (pending ≠ {}) do begin
    retire um elemento (pp,vs) de pending;
    marked := marked ∪ {(pp,vs)};
    bb := lookup (pp, program);
    (* bb é o bloco rotulado por pp no programa original *)
    code := newblock (pp,vs);
    (* cria novo bloco rotulado por (pp,vs) *)
    while (bb não estiver vazio) do begin
        command := first_command (bb);
        bb := rest (bb);
        case command of
            ...
    end;
    residual := extend (residual, code);
    (* adiciona bloco ao programa residual *)
end
```

Figura6. MIX - Algoritmo para Especialização de Programas.

a lista de valores iniciais $\texttt{vs}_0$ seria: $[n \mapsto (S,2)\,,\ x \mapsto (D,x)\,,\ p \mapsto (S,0)]$. Em métodos *online*, a utilização dos "*tags*" S e D para diferenciar valores estáticos e dinâmicos é essencial.

O conjunto `pending` contém os blocos do programa residual que ainda não foram gerados. Esses blocos são identificados por um par (pp, vs), onde pp é um rótulo do programa original e vs é uma lista de valores para as variáveis. O conjunto `marked` é utilizado para armazenar os rótulos dos blocos gerados no programa residual, para evitar o processamento de um bloco que já foi gerado.

O *loop* mais interno processa cada comando FCL do bloco corrente. Esses comandos podem alterar o estado vs das variáveis e também gerar novos blocos a serem inseridos em `pending`. O código para processar cada comando é exibido na Figura 7.

Primeiro discutiremos o processamento dos comandos de desvio. A definição de FCL impõe que qualquer comando de desvio sempre esteja posicionado no final de um bloco. Se o comando é um desvio incondicional `goto pp'`, a compressão da transição será realizada. No algoritmo, isso é feito pelo comando `bb := lookup (pp', program)`, lembrando que a variável `bb` contém o restante do bloco que está sendo processado em um dado momento. Nesse caso, os resultados da especialização do bloco corrente e de `pp'` aparecerão no mesmo bloco do programa residual.

```
case command of

(* se for comando return *)
return exp :
begin
    (type,val) := eval (exp, vs);
    if (type = S) then val = lift (val);
    code := extend (code, return val);
end

(* se for comando de atribuição *)
X := exp :
begin
    (type,val) := eval (exp, vs);
    vs := vs[X ↦ (type,val)];
    if (type = D) then
        code := extend (code, X := val);
end;

(* se for desvio incondicional *)
goto pp' :
    (* compressão da transição *)
    b := lookup (pp', program);

(* se for desvio condicional *)
if exp goto pp' else pp'' :
begin
    (type,val) := eval (exp, vs);
    if (type = S) then (* compressão da transição *)
        if (val = TRUE) then
            bb := lookup (pp', program);
        else
            bb := lookup (pp'', program);
    else begin (* type = D *)
        pending := pending ∪ ({(pp',vs)}\ marked);
        pending := pending ∪ ({(pp'',vs)}\ marked);
        code := extend (code,
            if val goto (pp',vs) else (pp'',vs) );
    end
end;
```

Figura7. Método *Online* para Especialização de Comandos FCL.

Para processar um desvio condicional `if exp goto pp' else pp''`, o primeiro passo é a avaliação da condição `exp`, executada pela função `eval(exp,vs)`. Essa função avalia uma expressão do programa original, usando uma lista de valores para as variáveis. O valor retornado é um par `(type,val)`, onde `type` pode ser S (estático) ou D (dinâmico). Se `type` é S, a expressão não depende de valores dinâmicos e `val` contém o valor constante resultado de sua avaliação. Se `type` é D, a expressão depende de valores dinâmicos e deve ser residualizada. Nesse caso, `val` contém o código da expressão especializado em relação às variáveis de `vs`.

Se a avaliação de `exp` no desvio condicional `if exp goto pp' else pp''` resultar em um valor estático, este será a constante TRUE ou FALSE. Se for TRUE, é executada uma compressão sobre a transição `pp'`, caso contrário, sobre `pp''`. Esse é um processo análogo ao que é conduzido quando o desvio é incondicional. Se o resultado da avaliação da condição for D, significa que ela depende de valores dinâmicos. Nesse caso, a compressão de transição não é realizada. Um comando de desvio condicional é gerado no final do bloco corrente. Esse comando é construído utilizando a condição especializada armazenada em `val`. Além disso, dois novos blocos podem ser inseridos em `pending`, se ainda não foi gerado código para eles. O algoritmo usa `marked` para determinar se código já foi gerado para os blocos em questão.

Na parte inicial do código exibido na Figura 7, pode-se ver o processamento dos comandos de atribuição e retorno de valor. Em um comando `return`, primeiramente a expressão é processada. Se resultar em um valor estático, ela deve sofrer *lift* para aparecer no código residual, uma vez que todo comando `return` é residualizado. Em um comando de atribuição, primeiramente o lado direito é processado. Se o resultado for um valor estático, o valor da variável é modificado e nenhum código é gerado. Se for dinâmico, a expressão especializada é utilizada na construção do comando de atribuição que irá aparecer no programa residual; além disso, a variável passa a ser classificada como dinâmica. A notação utilizada para modificação do valor de uma variável é `vs[`$X \mapsto$ `(type,val)]`, que significa: retorne uma nova lista `vs'` idêntica a `vs`, com exceção do valor da variável X, que é alterado para `(type,val)`.

6 Métodos *Offline*

Vimos que, nos métodos *online*, a determinação dos valores estáticos e dinâmicos é realizada junto com a especialização. Na abordagem *offline*,

por outro lado, o processo de especialização é geralmente dividido em duas fases.

A primeira fase de um método *offline* é designada BTA (*binding time analysis* ou *análise de tempo de definição*). Como nos métodos *online*, o processo começa determinando que partes da entrada do programa são estáticas e que partes são dinâmicas. Entretanto, o algoritmo de BTA não utiliza os valores especificados para as entradas estáticas. É feita uma análise sobre o programa, determinando quais variáveis dependem direta ou indiretamente das entradas estáticas e dinâmicas. Com isso, é produzida uma *divisão* de todas as variáveis nessas duas classes. Na maioria das vezes, essa informação é utilizada também para classificar cada estrutura do programa, por exemplo, comandos e operações, como estática ou dinâmica. Assim, BTA geralmente produz um *programa anotado* que identifica exatamente quais estruturas vão ser computadas e quais vão ser residualizadas.

A segunda fase de um método *offline* é a especialização propriamente dita. Como nos métodos *online*, a especialização gera os estados alcançáveis, especializa os pontos de programa e realiza a compressão das transições. Dados os valores das variáveis estáticas de entrada, o algoritmo segue estritamente as anotações geradas pela BTA para produzir o programa residual. Ao contrário dos métodos *online*, não é necessário determinar se uma operação é estática ou dinâmica, cada vez que ela for analisada.

Vamos mostrar a execução desses processos usando novamente a função Power da Figura 3(a).

6.1 Análise de Tempo de Definição

Suponha que a função Power deva ser especializada em relação a n, o primeiro parâmetro. Essa é a única informação necessária para executar a BTA. De modo diverso dos métodos *online*, não será utilizado, por enquanto o valor fornecido para a entrada estática n.

O objetivo inicial é descobrir quais variáveis utilizadas no programa são estáticas e dinâmicas, isto é, computar uma *divisão* das variáveis. Em seguida, um programa anotado é gerado.

A computação da divisão das variáveis pode seguir dois métodos diferentes: iteração até atingir um ponto fixo ou *resolução de restrições* (do inglês *constraint solving*). O primeiro método é o mais simples e utilizado pela maioria dos avaliadores parciais mais antigos. Nesta seção utilizaremos apenas esse método, que executa diversos passos sobre o programa, até atingir um ponto fixo na divisão das variáveis. Referências sobre

análise de tempo de definição usando resolução de restrições podem ser encontradas em [5, 6, 8].

Se a função Power, da Figura 3(a), vai ser especializada em relação a n, o algoritmo de iteração até atingir um ponto fixo é disparado com a especificação $[n \mapsto S, x \mapsto D]$, indicando que n é estático (S) e x é dinâmico (D). A seguinte *divisão inicial* é computada:

$$\Delta = [n \mapsto S, x \mapsto D, p \mapsto S]\,,$$

que casa com a especificação de entrada e determina que as demais variáveis são estáticas. Podemos assumir que as variáveis não pertencentes à entrada são inicialmente estáticas, pois todas as variáveis são inicializadas com 0 em FCL.

O algoritmo executa passadas seqüenciais sobre o programa, modificando a divisão, até que mais nenhuma modificação seja processada. O princípio utilizado para as modificações é a *congruência*: qualquer variável que dependa de uma variável dinâmica deve também ser classificada como dinâmica. No caso de FCL, a dependência é dada pelos comandos de atribuição: `V := exp` indica que `V` depende de quaisquer variáveis que apareçam em `exp`.

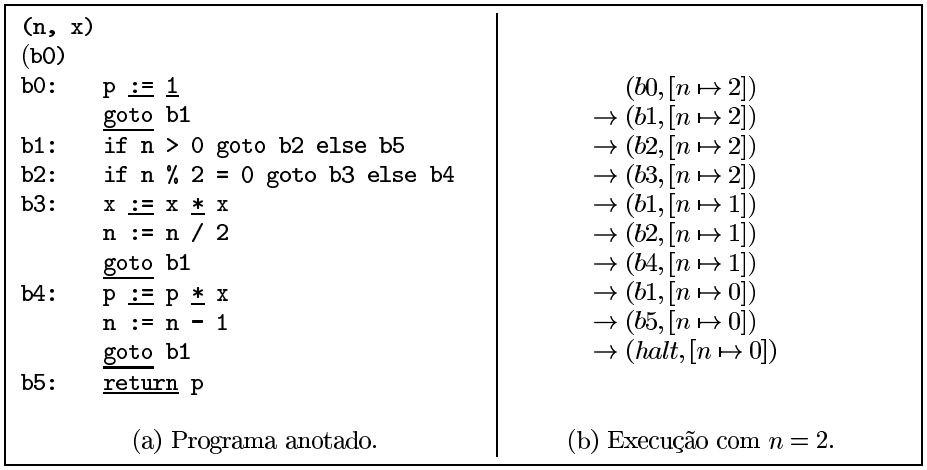
Assim, uma alteração na divisão só ocorre se uma variável que antes era classificada com estática passar a ser dinâmica. Como o número de variáveis é finito, garante-se que o processo sempre termina. Essa estratégia é conservadora, pois uma variável é considerada dinâmica se o for em qualquer ponto do programa. É conhecida por *divisão uniforme*, pois vale para o programa inteiro.

Voltando ao exemplo, a primeira iteração do algoritmo determina que a variável p deve ser dinâmica, uma vez que depende de x no comando `p := p * x`. Na segunda iteração, nenhuma mudança é realizada, assim o algoritmo termina com a seguinte divisão:

$$\Delta' = [n \mapsto S, x \mapsto D, p \mapsto D]\,.$$

O próximo passo é gerar um programa anotado. As estruturas do programa que dependem das variáveis dinâmicas são marcadas como *residualizáveis*. Se um dos operandos de uma operação é dinâmico, a operação é classificada como dinâmica. Um comando de atribuição é dinâmico se a variável do lado esquerdo o for.

Na Figura 8(a), é exibido o código anotado da função Power, a partir da divisão computada acima.. A codificação é feita usando-se uma linguagem de dois níveis [29, 28]. As estruturas dinâmicas aparecem sublinhadas.



(a) Programa anotado. (b) Execução com $n = 2$.

Figura8. Programa Power Anotado e Execução com Entrada Parcialmente Conhecida.

As variáveis não precisam ser anotadas porque sua classificação pode ser obtida diretamente da divisão computada. Todos os comandos de atribuição cujo lado esquerdo é p ou x são anotados como residualizáveis, pois essas variáveis são dinâmicas. No primeiro bloco, a constante 1 é anotada, indicando que irá aparecer no código residual. Como visto na Seção 5, sempre que um valor estático aparece em um contexto dinâmico, ele sofre um *lift*. Assim, a anotação da constante 1 é equivalente a escrever `lift(1)`, mas neste caso a operação de *lift* está *compilada* no código anotado. Por enquanto, vamos deixar a compressão de transições de lado, assim todos os `goto` são anotados.

6.2 Geração dos Estados Alcançáveis

Esse é o primeiro passo da especialização, seguindo a geração do programa anotado produzido na fase de BTA. Lembramos novamente que os três passos da especialização são geralmente conduzidos paralelamente, mas neste exemplo vamos executá-los em seqüência.

A propriedade de congruência da divisão das variáveis garante que nenhuma variável estática depende de uma dinâmica. Desse modo, a representação dos estados pode ser mais simples que a empregada nos métodos *online*. Cada estado será representado apenas pelos valores das variáveis estáticas.

No nosso exemplo, o estado inicial será $(b0, [n \mapsto 2])$. A seqüência de estados gerada é exibida na Figura 8(b). Para gerar essa seqüência de

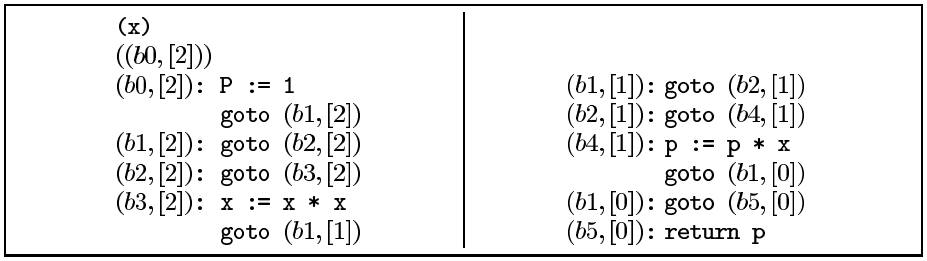


Figura9. Programa Residual, sem Compressão de Transições.

estados, basta seguir as anotações. As construções sublinhadas são ignoradas, uma vez que não contribuem para a determinação dos valores estáticos.

6.3 Especialização dos Pontos de Programa

Como nos métodos *online*, para cada estado (l_i, σ_i), uma versão especializada do bloco l_i é criada, usando os valores das variáveis estáticas de σ_i. A diferença é que não é necessário verificar se cada operação é S ou D, isso já foi estabelecido pelas anotações.

Por exemplo, no bloco `b3` da Figura 8(a), todos os componentes do primeiro comando de atribuição estão sublinhados, assim são copiados diretamente para o código residual. O segundo comando, ao contrário, gera uma atualização do valor de n. Finalmente, o comando de desvio é especializado é copiado para o código residual.

O programa residual é exibido na Figura 9.

6.4 Compressão de Transições

A compressão de transições é feita da mesma maneira que nos métodos *online*. Assim, o programa residual terá o seguinte formato, após as otimizações dos desvios:

```
(x)
(b0)
b0: P := 1
    x := x * x
    p := p * x
    return p
```

6.5 Especialização Offline: Três Passos em Paralelo

Um algoritmo para avaliação parcial de programas usando um método *offline* tem o mesmo formato geral proposto para os métodos *online*, exibido na Figura 6. A diferença reside em como os comandos são tratados. Nos métodos *online*, o caráter estático ou dinâmico de cada componente é calculado durante a execução da especialização. Nos métodos *offline*, um programa anotado com informações de análise de tempo de definição é produzido e depois simplesmente as anotações são seguidas durante a especialização.

No método *online*, o algoritmo da Figura 6 tinha como entradas o programa original `program`, o rótulo $\texttt{pp}_0$ do bloco inicial do programa e o valor inicial $\texttt{vs}_0$ das variáveis. O método *offline* precisa dessas mesmas entradas, e de uma quarta, que é a divisão das variáveis computada pela BTA. Além disso, `program` é o programa anotado produzido pela BTA, e não mais o programa original.

O exemplo estudado nesta seção, que foi a especialização *offline* da função Power, nos mostrou que é necessário representar apenas o valor das variáveis estáticas em cada estado. Desse modo, elimina-se a necessidade de utilizar *tags* para identificar se um valor de uma variável é estático ou dinâmico, durante a especialização. O estado inicial $\texttt{vs}_0$ das variáveis, no caso da função Power especializada em relação a n, será $[n \mapsto 2]$.

Como fizemos na Seção 5, o código *offline* para processar cada comando FCL, que são: `return`, atribuição, desvio condicional e incondicional, é apresentado separadamente do *loop* principal do algoritmo de especialização. Esse código pode ser observado na Figura 10.

As funções utilizadas na Figura 10 para avaliar expressões são `eval` e `reduce`. A função `eval` só é aplicada a expressões classificadas pela BTA como estáticas, retornando uma constante resultante da avaliação dessas expressões. A função `reduce`, por outro lado, só é aplicada a expressões classificadas como dinâmicas. O resultado da aplicação de `reduce` a uma expressão é uma nova expressão, onde todas as operações estáticas foram computadas. Isso configura uma política completamente diferente da adotada na abordagem *online*, onde a avaliação de uma expressão poderia resultar em um valor estático ou dinâmico, necessitando da propagação de valores com *tags*.

6.6 Assegurando a Terminação

No algoritmo de BTA apresentado na Seção 6.1, uma variável é classificada como dinâmica quando depende de um valor dinâmico em algum

```
case command of

(* se for comando return *)
return exp :
    code := extend (code, return reduce(exp,vs) );

(* se for comando de atribuição *)
 X := exp :
    if (X foi classicada como estática) then
        vs := vs[X ↦ eval(exp,vs)];
    else
        code := extend (code, X := reduce(exp,vs) );

(* se for desvio incondicional *)
goto pp' :
    (* compressão da transição *)
    b := lookup (pp', program);

(* se for desvio condicional *)
if exp goto pp' else pp'' :
    if (exp foi classificada como estática) then
        if (eval(exp,vs) = TRUE) then
            (* compressão da transição *)
            bb := lookup (pp', program);
        else
            (* compressão da transição *)
            bb := lookup (pp'', program);
    else begin
        (* desvio condicional dinâmico *)
        pending := pending ∪ ({(pp',vs)}\ marked);
        pending := pending ∪ ({(pp'',vs)}\ marked);
        code := extend (code,
            if reduce(exp,vs) goto (pp',vs) else (pp'',vs) );
    end
```

Figura10. Método *Offline* para Especialização de Comandos FCL.

ponto do programa, o que convencionamos chamar de *princípio da congruência*. Como veremos a seguir, essa estratégia simples pode levar o avaliador parcial a um *loop* infinito.

Observe o trecho de programa FCL a seguir, adaptado de [18]:

```
b0: if y ≠ 0 goto b1 else b2
b1: x := x + 1
    y := y - 1
    goto b0
b2: ...
```

Suponha que y seja classificada como dinâmica. Se x não é atualizada com um valor dinâmico em nenhum outro ponto do programa, a nossa estratégia simples iria classificar x como estática.

Por simplicidade, vamos supor que a única variável estática do programa seja x. Vamos supor também que, em algum momento da especialização, o bloco rotulado por $b0$ é atingido com x valendo 0. O seguinte código residual seria gerado:

```
(b0,[x ↦ 0]): if y ≠ 0 goto (b1,[x ↦ 0]) else (b2,[x ↦ 0])
(b1,[x ↦ 0]): y := y - 1
              goto (b0,[x ↦ 1])
(b2,[x ↦ 0]): ...
```

Podemos observar que um novo estado foi gerado: $(b0, [x \mapsto 1])$. O avaliador deve gerar um bloco especializado para esse estado:

```
(b0,[x ↦ 1]): if y ≠ 0 goto (b1,[x ↦ 1]) else (b2,[x ↦ 1])
(b1,[x ↦ 1]): y := y - 1
              goto (b0,[x ↦ 2])
(b2,[x ↦ 1]): ...
```

O processo continuaria indefinidamente. O conjunto de estados alcançáveis seria infinito. O problema é que, embora x dependa apenas de constantes e de si mesmo, o conjunto de valores que pode receber é ilimitado, pois os valores são computados sob um controle dinâmico. Uma solução é permitir que BTA classifique todas as variáveis calculadas sob controle dinâmico como dinâmicas.

O processo de se classificar uma variável como dinâmica, mesmo quando o princípio da conguência permite que seja estática, é chamado de *generalização*. Esquemas para resolver esse problema podem ser encontrados em [16, 31].

7 Comparação Entre os Métodos *Online* e *Offline*

Para tecer comparações entre as duas abordagens propostas para avaliação parcial, vamos observar os programas residuais produzidos para a função Power, nas Seções 5 e 6.

Ambas as abordagens levaram à produção de nove estados diferentes, mas o método *online* possibilitou uma maior exploração do valor estático da variável p. Recordando o raciocínio desenvolvido, o comando de atribuição `p := 1;` foi residualizado pelo método *offline*, mas foi computado pelo método *online* e não apareceu no programa residual. Isso aconteceu porque a variável p foi identificada pela BTA como dependendo de valores dinâmicos em algum ponto do programa. Portanto o método *offline* classificou toda operação envolvendo p como dinâmica no programa anotado. O comando de atribuição `p := p * x;` também foi completamente residualizado pelo método *offline*, enquanto a especialização *online* pôde inferir o valor constante 1 a partir de p.

Os resultados alcançados, embora simples, mostram que os métodos *offline* são muitas vezes mais conservadores do que a abordagem *online*. Uma análise mais cuidadosa revela, entretanto, que muitas vantagens são conseguidas quando se utiliza a abordagem *offline*.

O que deve ser pesado quando se escolhe uma das abordagens é:

- ter que lidar com a propagação de *tags* que identificam se um valor é estático ou dinâmico durante o processo de especialização; ou
- ter que produzir um programa anotado e realizar a especialização em duas fases.

Uma analogia com as políticas de verificação de tipos de linguagens de programação pode dar uma idéia clara das diferenças. Avaliação parcial *online* é análoga à verificação de tipos em tempo de execução, onde os valores posuem *tags* associados indicando o tipo. Avaliação parcial *offline*, por outro lado, é análoga à verificação de tipos estática.

Da mesma maneira que verificação de tipos dinâmica, especialização *online* tem a vantagem de ser mais flexível e menos conservadora, uma vez que pode se basear nos valores reais das variáveis para tomar decisões sobre o caráter estático ou dinâmico de uma construção do programa analisado. Na abordagem offline, a fase de BTA não tem acesso aos valores das variáveis, tendo que decidir a classificação baseada apenas nos *tags* S e D.

Da mesma forma que a verificação de tipos estática, avaliação parcial *offline* tem a vantagem de ser mais eficiente e permitir uma verificação

mais fácil das propriedades do programa residual, antes que ele seja gerado. Uma vez que a divisão das variáveis é executada uma única vez, elimina-se o esforço adicional de se lidar com *tags* durante a especialização.

Os primeiros avaliadores parciais usavam sempre métodos *online*. As técnicas *offline* receberam um grande impulso quando se buscou a compilação e geração de compiladores usando avaliação parcial. Essas tarefas envolvem a auto-aplicação do avaliador parcial, processo que introduz muitos problemas para a abordagem *online*, mas que puderam ser resolvidos de maneira satisfatória com os métodos *offline*. Um dos motivos pelos quais os métodos *offline* facilitam a auto-aplicação é a divisão do processamento em duas fases. O código do avaliador parcial que é submetido a si próprio consiste em um programa anotado mais simples, pois executa apenas a segunda fase do método, que consitui o núcleo da especialização. No Capítulo 9 discutiremos a auto-aplicação com mais detalhe.

8 Exemplo: Casamento de Padrões

Nesta seção, mostraremos a aplicação do método de especialização polivariante a um exemplo um pouco mais complexo do que os utilizados nas Seções 5 e 6. O exemplo compreende um algoritmo ingênuo de casamento de padrões escrito na linguagem FCL. O método de especialização vai empregar técnicas *offline*.

Um programa para realizar casamento de padrões em textos tem duas entradas, ambas consisitindo de uma seqüência de caracteres:

- p é o padrão procurado, tem tamanho M e é indexado de 0 a $M-1$;
- A é o texto onde o padrão será pesquisado, tem tamanho N e é indexado de 0 a $N-1$.

O programa retorna -1 se não encontrou nenhuma ocorrência de p em A. Caso contrário, retorna a posição da primeira ocorrência encontrada. Um algoritmo ingênuo, codificado em FCL, é apresentado na Figura 11.

A tupla (p, M, A, N) indica que a entrada é formada, respectivamente, pelo padrão e seu comprimento, e pelo texto e seu comprimento. A variável j indica o caractere corrente do padrão e i indica o caractere corrente do texto. O algoritmo é ingênuo porque, se $p[j] \neq A[i]$, j passa indicar novamente o primeiro caractere de p e i avança apenas uma posição no texto, a partir do início do último prefixo correto.

Vamos nos concentrar agora na especialização do programa apresentado em relação a um padrão específico. Ou seja, vamos produzir um novo

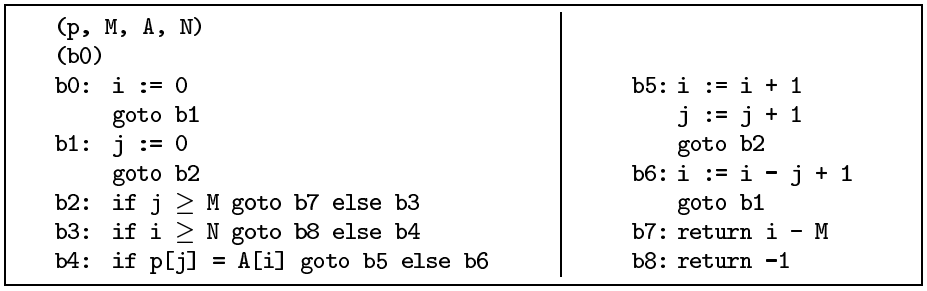


Figura11. Programa FCL para Casamento de Padrões.

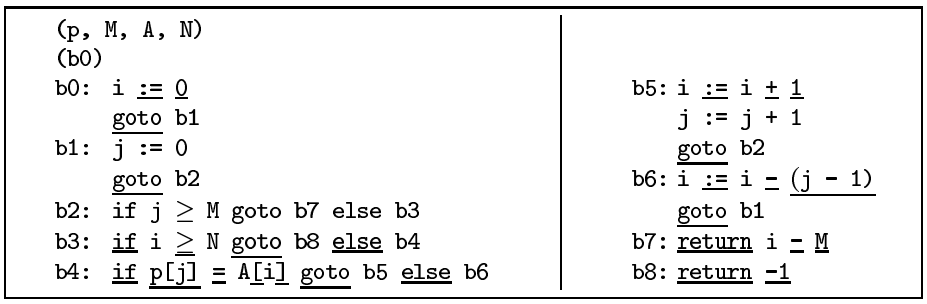


Figura12. Programa Anotado.

programa, mais eficiente, que realiza a busca de um padrão específico em qualquer texto. As entradas p e M seriam, portanto classificadas como estáticas, enquanto que A e N seriam dinâmicas. Usando um método *offline*, a primeira providência é executar uma análise de tempo de definição. A divisão resultante da BTA seria:

$$\Delta = [p \mapsto S,\ M \mapsto S,\ A \mapsto D,\ N \mapsto D,\ j \mapsto S,\ i \mapsto D]\,.$$

Observe que as variáveis i e j são atualizadas apenas com resultados de operações envolvendo constantes e o valor de si próprias. Na Seção 6.6, mostramos que uma divisão congruente produzida pela BTA pode levar o processo de especialização a um *loop* infinito. No exemplo desta seção, uma análise mais elaborada mostra que j tem crescimento limitado por M, um valor conhecido, enquanto que i tem crescimento limitado por N, um valor desconhecido. Assim, é necessária uma generalização da variável i, ou seja, classificá-la como dinâmica, embora o princípio da congruência permita que seja estática.

O programa anotado, baseado na divisão calculada, é exibido na Figura 12. A forma de apresentação do programa anotado é a mesma adotada na Seção 6, com as operações dinâmicas sublinhadas. Os comandos `goto`

são todos sublinhados, uma vez que estamos considerando, por questões didáticas, uma fase separada para compressão de transições. As variáveis não são anotadas porque sua classificação pode ser obtida diretamente da divisão. As que aparecem sublinhadas no programa acima são devido a uma operação de *lift* implícita, ou seja, são valores estáticos que aparecem em contextos dinâmicos e são então transformados de modo a aparecer no código residual. No bloco `b6`, aplicamos uma otimização que altera a ordem das operações: a operação `(j - 1)` é realizada antes da subtração, pois envolve dois operandos estáticos e assim pode ser completamente computada.

Vamos supor agora que o programa anotado seja especializado em relação aos seguintes valores: $p =$ "*abc*" e $M = 3$. O estado inicial será $(b0, [\text{"}abc\text{"}, 3, 0])$, indicando o bloco `b0` como ponto de programa e os valores das variáveis p, M e j, respectivamente. Uma otimização bastante simples na geração dos estados alcançáveis é a seguinte: se uma variável estática não é modificada em nenhum ponto do programa, ela aparecerá com o mesmo valor em todos os estados, assim pode ser eliminada da representação dos estados sem prejuízo do processo. No nosso caso, p e M nunca são alteradas, assim podemos simplificar a representação dos estados, usando apenas a variável j.

Discutiremos a seguir a geração dos estados alcançáveis e a especialização dos pontos de programa, usando como estado inicial $(b0, [0])$, onde 0 é o valor inicial da variável j.

Inicialmente, o bloco

$$(b0, [0])\texttt{: i := 0}\quad \texttt{goto } (b1, [0])$$

é gerado, não envolvendo nenhuma computação de valores estáticos. Lembrando do algoritmo apresentado nas Figuras 6 e 10, o estado $(b1, [0])$ é acrescentado a `pending`, o conjunto dos estados ainda não gerados. O código associado a $(b1, [0])$ consiste apenas de um desvio para o bloco identificado por $(b2, [0])$. Nesse terceiro bloco, a condição do desvio é estática e computada como falsa. O código associado a esses dois blocos é exibido abaixo:

$$(b1, [0])\texttt{: goto } (b2, [0]) \qquad\qquad (b2, [0])\texttt{: goto } (b3, [0])$$

O quarto bloco apresenta a primeira ocorrência de uma situação onde a condição de um desvio é dinâmica:

$$(b3, [0])\texttt{: if i} \geq \texttt{N goto } (b8, [0])\texttt{ else } (b4, [0])$$

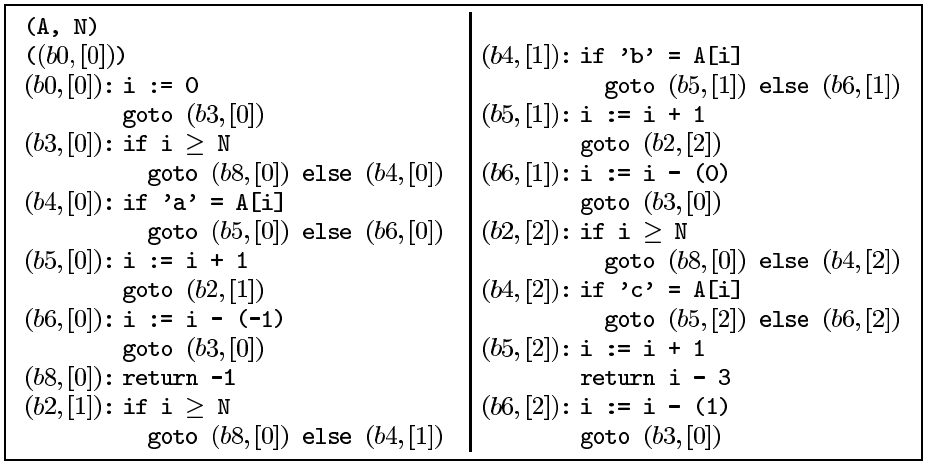


Figura13. Programa Residual.

Nesse caso, o conjunto `pending` é acrescido de dois novos estados, cuja ordem de processamento é irrelevante. O código associado a esses estados é exibido abaixo:

$(b8, [0])$: `return -1`
$(b4, [0])$: `if 'a' =` $=$ `A[i] goto` $(b5, [0])$ `else` $(b6, [0])$

O bloco identificado por $(b4, [0])$ produz dois novos estados: $(b5, [0])$ e $(b6, [0])$. A primeira vez que o valor de j é modificado é no bloco $(b5, [0])$ e o bloco $(b6, [0])$ nos apresenta o primeiro caso de loop produzido no código residual:

$(b5, [0])$: `i := i + 1` `goto` $(b2, [1])$

$(b6, [0])$: `i := i - (-1)` `goto` $(b1, [0])$

O bloco $(b1, [0])$ já foi gerado, assim não é acrescentado em `pending`. O conjunto `marked` é utilizado pelo algoritmo para determinar os blocos cujo código já foi gerado.

A geração dos demais blocos segue um processo similar ao descrito acima. O código residual, após a compressão das transições, é exibido na Figura 13. O programa residual age como um autômato, mudando de estado a cada vez que um dos caracteres do padrão é encontrado no texto, em seqüência. Se houver falha no casamento, retorna ao estado inicial, neste caso, representado por $(b3, [0])$. Uma otimização que não havia sido ainda comentada foi conduzida no código: os estados $(b8, [0])$, $(b8, [1])$ e $(b8, [2])$ foram fundidos em um único, rotulado como $(b8, [0])$, pois o código gerado para os três é idêntico.

9 Geração de Geradores de Programas

Nesta seção, vamos dar um tratamento um pouco mais formal à avaliação parcial de programas. Mostraremos como essa técnica pode ser utilizada para compilação e geração de compiladores dirigida por semântica.

Seguindo a notação utilizada em [18], se P é um programa escrito na linguagem L, $[\![P]\!]_L$ é uma função que denota a sua semântica. Quando não for importante, omitiremos o subscrito L da notação. Sendo assim, a definição equacional do avaliador parcial `mix` é a seguinte:

$$\begin{aligned} out &= [\![P]\!]_S\,(in_1, in_2) \\ P_{in_1} &= [\![mix]\!]_L\,(P, in_1) \\ out &= [\![P_{in_1}]\!]_T\,(in_2) \end{aligned}$$

O programa P, quando aplicado às entradas, in_1 e in_2, produz a saída out. O avaliador parcial `mix`, quando aplicado a P e parte de sua entrada (in_1), produz um novo programa identificado como P_{in_1}. O programa residual P_{in_1} produz a mesma saída out, quando a entrada restante (in_2) é submetida a ele. A avaliação parcial é vantajosa quando in_2 varia mais do que in_1 e P_{in_1} executa mais rápido sobre in_2 do que P sobre in_1 e in_2.

As linguagens envolvidas são L, usada para implementar o avaliador parcial mix; S, a linguagem fonte dos programas submetidos ao avaliador parcial; e T, a linguagem objeto dos programas especializados produzidos. Geralmente, S e T são idênticas, mas existem casos onde elas são distintas. Nos exemplos apresentados nas Seções 5, 6 e 8, as linguagens S e T são a linguagem de fluxogramas FCL. Não discutimos em qual linguagem o próprio avaliador parcial estaria implementado, embora tenhamos apresentado um código para mix nas Figuras 6, 7 e 10, usando uma linguagem algorítimica. Com pouco esforço, esses algoritmos podem ser traduzidos para a linguagem FCL.

9.1 Compilação e Geração de Compiladores

Como o avaliador `mix` é um programa com duas entradas, ele pode servir de entrada para si próprio. Futamura foi o primeiro a sugerir essa abordagem, apresentando três equações que passaram a ser conhecidas como *três projeções de Futamura* [13].

Supondo *int* um interpretador de uma linguagem qualquer, escrito em S, a primeira projeção de Futamura mostra que compilação via avaliação

parcial sempre gera programas corretos:

$$\begin{aligned} out &= [\![source]\!]\,(input) \\ &= [\![int]\!]\,(source, input) \\ &= [\![[\![mix]\!]\,(int, source)\,]\!]\,(input) \\ &= [\![target]\!]\,(input) \end{aligned}$$

Assim temos $target = [\![mix]\!]\,(int, source)$, ou seja, o programa objeto é resultado da avaliação parcial de um interpretador em relação a um programa fonte específico. Um interpretador para uma linguagem L pode ser visto como a descrição da semântica de L. Assim, a primeira projeção de Futamura mostra que é possível realizar *compilação dirigida por semântica*, usando um avaliador parcial `mix`.

O avaliador parcial `mix` é um programa que recebe duas entradas: um programa P a ser especializado e parte dos dados de entrada de P. Portanto, o próprio `mix` pode ser especializado em relação a P.

A segunda projeção de Futamura refere-se à geração de compiladores por meio da auto-aplicação de `mix`:

$$\begin{aligned} target &= [\![mix]\!]\,(int, source) \\ &= [\![[\![mix]\!]\,(mix, int)\,]\!]\,(source) \\ &= [\![compiler]\!]\,(source) \end{aligned}$$

Temos então $compiler = [\![mix]\!]\,(mix, int)$. Um compilador é gerado via avaliação parcial do próprio avaliador parcial, em relação a um interpretador específico: *geração de compiladores dirigida por semântica*.

Observe que havíamos feito a suposição de que S era a linguagem fonte dos programas submetidos a `mix`. No caso da auto-aplicação, isso quer dizer que o próprio `mix` deve ser escrito na linguagem S. Nos exemplos das Seções 5, 6 e 8, tanto a linguagem fonte dos programas submetidos a `mix`, quanto a linguagem dos programas residuais produzidos, eram a linguagem FCL. Para podermos aplicar os procedimentos descritos nesta seção, seria necessário codificar `mix` em FCL.

A auto-aplicação de mix pode ir ainda mais longe. A terceira projeção de Futamura envolve geração de geradores de compiladores:

$$\begin{aligned} compiler &= [\![mix]\!]\,(mix, int) \\ &= [\![[\![mix]\!]\,(mix, mix)\,]\!]\,(int) \\ &= [\![cogen]\!]\,(int) \end{aligned}$$

O programa `cogen` é chamado de gerador de compiladores, porque recebe um interpretador para uma linguagem L como entrada, produzindo um compilador de L para a linguagem dos programas residuais de `mix`. Através da auto-aplicação de um avaliador parcial, é possível produzir automaticamente um gerador de compiladores.

9.2 Resultados Práticos

O primeiro avaliador parcial auto-aplicável foi construído por Jones, Sestoft e Sondergaard, para uma linguagem de equações recursivas de primeira ordem. A primeira versão [19] requeria anotações prévias introduzidas pelo usuário, mas na versão seguinte [20] era completamente automática.

Jorgensen realizou experimentos que envolviam a geração de um compilador para uma linguagem funcional de avaliação *lazy*, usando um avaliador parcial escrito em uma linguagem funcional de avaliação estrita [22]. Os experimentos mostraram que a velocidade de execução do código compilado foi equivalente ao produzido por compiladores comerciais.

Muitos experimentos envolvendo a geração automática de geradores de compiladores `cogen`, usando a auto-aplicação de um avaliador parcial, foram bem sucedidos. A maioria tinha como linguagem fonte uma linguagem não tipada [20, 14, 10, 21, 22, 15]. Para linguagens fortemente tipadas, resultados mais eficientes são conseguidos escrevendo o programa `cogen` à mão, em vez de gerá-lo automaticamente por meio da terceira projeção de Futamura [2, 7, 23]. Essa abordagem é conhecida como *geração de extensões*.

10 Leitura Adicional

O texto base para entendimento de avaliação parcial de programas é o livro de Jones, Gomard e Sestoft [18]. O livro cobre tópicos iniciais, como as definições básicas dos conceitos envolvidos, algoritmos para implementação de avaliadores parciais e exemplos, de forma bastante didática. Trata também aspectos relacionados à especialização de programas escritos em linguagens de paradigmas diversos: imperativo, funcional e lógico. Os capítulos finais apresentam tópicos mais avançados, como garantia de terminação da avaliação parcial, supercompilação etc.

Um segundo livro publicado sobre o assunto três anos depois, tendo como autores Danvy, Glück e Thiemann [11], reúne artigos completos, procurando resumir o estado da arte e as perspectivas futuras da avaliação parcial de programas.

O artigo de Mogensen e Sestoft [27] é ideal para iniciantes em avaliação parcial, pois é um tutorial mais atualizado. Em poucas páginas, aborda a maioria dos aspectos mais importantes relacionados ao tema. Em particular, trata a abordagem de geração de extensões com muito mais detalhe que o livro de Jones, Gomard e Sestoft. Algumas referências bibliográficas citadas são mais recentes.

A Escola de Verão de 1998 do Departamento de Ciência da Computação da Universidade de Copenhagen (DIKU) abordou o tema: *Avaliação Parcial - Teoria e Prática*. Os textos dos seminários apresentados no evento são também uma leitura muito interessante. Infelizmente, os anais não foram publicados por completo, cabendo ao leitor interessado uma pesquisa no *site* do grupo de pesquisa TOPPS: www.diku.dk/research-groups/topps. O material não publicado inclui os textos e transparências utilizadas por John Hatcliff, que serviram como umas das principais bases do texto apresentado neste capítulo, especialmente as Seções 5 e 6.

11 Conclusão

Como este documento se trata de um tutorial, procuramos nos concentrar nos aspectos introdutórios e mais práticos. O objetivo principal é passar ao leitor um conceito claro de avaliação parcial de programas, as diferenças práticas entre as abordagens *online* e *offline* e uma introdução à geração automática de compiladores usando avaliação parcial.

Todos os exemplos utilizados para especialização de programas foram escritos na linguagem FCL. As Seções 5 e 6 apresentam detalhadamente algoritmos de especialização de programas FCL usando as abordagens *online* e *offline*. Os princípios básicos discutidos podem ser utilizados para construir um avaliador parcial para linguagens imperativas reais ou que seguem outros paradigmas, como as linguagens funcionais e lógicas.

Por motivos de espaço, importantes otimizações a serem introduzidas nos algoritmos de especialização não foram discutidas neste documento. Entre elas, podem ser citadas: análise de estruturas parcialmente estáticas, expansão de chamadas de funções, BTA com divisão não uniforme, BTA com divisão polivariante e outras. Outro aspecto importante que não foi explorado é a abordagem de *geração de extensões*, que consiste em escrever à mão o gerador de compiladores `cogen`. Quando a linguagem fonte é fortemente tipada, essa abordagem pode produzir resultados mais eficientes na geração automática de compiladores dirigida por semântica. A bibliografia citada na Seção 10 pode ser consultada para se obter informações sobre esses tópicos.

A área de avaliação parcial de programas tem apresentado recentemente resultados práticos mais expressivos, principalmente com a utilização de avaliadores parciais para linguagens imperativas de implementação eficiente. Nesse sentido, o projeto C-mix tem sido muito importante (consultar www.diku.dk/research-groups/topps). Esse projeto consiste na implementação de um avaliador parcial eficiente para a linguagem ANSI C e tem sido utilizado em diversas aplicações, algumas citadas na Seção 2.

Referências

1. L. Andersen. C program specialization. Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. Student Project 91-12-17.
2. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.
3. P. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.
4. A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.
5. L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993. DIKU Research Report 93/22.
6. L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 61–71, 1994.
7. L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September, 1994. (Lecture Notes in Computer Science, Vol. 844)*, pages 198–214. Berlin: Springer-Verlag, 1994.
8. L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, September 1995.
9. C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.
10. C. Consel and S. Khoo. Semantics-directed generation of a prolog compiler. Technical Report YALEU/DCS/RR-781, Yale University, New Haven, Connecticut, May 1990.
11. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
12. A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
13. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
14. C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, Amsterdam: North-Holland, 1989.

15. R. Heldal. Generating more practical compilers by partial evaluation. In R. Heldal, C. Kehler Holst, and P. Wadler, editors, *Functional Programming, Glasgow 1991*, pages 158–163. Berlin: Springer-Verlag, 1992.
16. N. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. Amsterdam: North-Holland, 1988.
17. N. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, pages 49–58. New York: IEEE Computer Society, 1990.
18. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
19. N. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.
20. N. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
21. J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.
22. J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992*, pages 258–268. New York: ACM, 1992.
23. J. Launchbury and C. Holst. Handwriting cogen to avoid problems with static typing. In *Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218, 1991.
24. M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.
25. T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.
26. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *LOPSTR 92. Workshops in Computing*. Berlin: Springer-Verlag, Jan. 1993.
27. T. Æ. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.
28. F. Nielson and H. R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.
29. F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science **vol. 34**. Cambridge University Press, 1992.
30. D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991. Report TRITA-TCS-9101.
31. P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. Amsterdam: North-Holland, 1988.