frama C

Software Analyzers

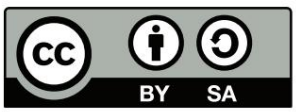
# Fatiamento — Técnica de documentação

# Documentação do plugin Slicing

## Projeto de uma ferramenta de corte

Anne Pacalet e Patrick Baudin






Lista CEA, Laboratório de Segurança e Proteção de Software, Saclay, F-91191

## Versões

Somente as versões Vx.0 são documentos completos; os outros devem ser considerados como rascunhos.

V4.0 - 2 de novembro de 2020:

— Documento sob licença Creative Commons “Attribution-ShareAlike 4.0 International”

https://creativecommons.org/licenses/by/4.0/.

V3.0 - 3 de dezembro de 2008:

— pequenas correções, —
acréscimos ao *fatiamento* de declarações globais, —
acréscimos ao gerenciamento de anotações.

V2.0 - 26 de junho de 2007:

— extração da documentação do CEO, — atualização
quanto ao estado da ferramenta, — fusão da
documentação da interprocessual, — reorganização da parte da
interprocessual, — apensação dos projetos não executados.

V1.0 - 27 de junho de 2006:

— aditamento da apresentação do documento, —
aditamento do capítulo "Uso" (manual), —
desenvolvimento do cálculo de determinados conjuntos em grafos de dependência, — atualização do
capítulo sobre o interprocedimento, — aditamento de uma
conclusão resumindo o progresso do trabalho.

V0.2 - 27 de abril de 2006:

— reorganização da definição de conjuntos de instruções, — correções diversas.

V0.1 - 22 de fevereiro de 2006:

— bibliografia sobre *fatiamento,* —
análise do documento [Baudin(2004)], — o que
pretendemos fazer, — primeiros
elementos sobre a construção do grafo de dependência, — integração e ordenação dos
arquivos do antigo documento (FLD ), — algoritmo para calcular dependências
de controle.

# Conteúdo

# Capítulo 1

# Introdução

O objetivo deste trabalho é projetar e desenvolver uma ferramenta *de fatiamento* baseada no kernel Frama-C para análise estática de programas C, inspirando-se no módulo equivalente já desenvolvido com base na ferramenta Caveat descrita em [Pacalet e Baudin (2005)].

## 1.1 Estado da arte

Vejamos primeiro o que diz a literatura sobre o assunto para nos posicionarmos em relação aos diferentes métodos.

Contamos aqui com a apresentação do trabalho de cálculo da PDG realizado em outro documento.

### 1.1.1 Origem

A técnica que consiste em reduzir um software - chamada em inglês *de slicing* - tem sido muito estudada desde sua introdução pela tese de [Weiser (1979)]. O artigo [Weiser (1981)] lembra que sua definição de *fatiamento* é a seguinte:

**Definição (fatiar selon Weiser)**

O programa original e o programa transformado devem ser idênticos se vistos através da janela definida pelo critério de fatiamento *definido* por um ponto do programa e uma lista de variáveis. Ou seja, a sequência de valores observada para essas variáveis neste ponto deve ser a mesma.
Além disso, o programa transformado deve ser obtido deletando certas instruções do programa original.

Essa definição foi posteriormente questionada com o argumento de que essa definição não é suficiente para traduzir o entendimento intuitivo que temos do *fatiamento.* O exemplo a seguir pode ser encontrado em [Kumar e Horwitz (2002)], mas essa crítica é repetida várias vezes.

**Exemplo 1**

| P | P1 | P2 |
|---|---|---|
| a = 1;<br>b = 3;<br>c = a*b;<br>a = 2; a = 2; b = 2; b<br>= 2; d = a+b; d =<br>a+b; d = a+b; | | a = 1;<br>b = 3; |

P1 é provavelmente o que esperamos de *fatiar* P se estivermos interessados na variável d após a última instrução, mas P2 também está correto do ponto de vista de Weiser.

Parece improvável construir um algoritmo que construa P2, mas, no entanto, vemos que a definição é insuficiente para especificar totalmente qual *deve ser o resultado de um fatiamento.*

[Choi e Ferrante (1994)] formaliza a definição da seguinte forma:

**Definição (Correct-Executable-Slice)**

Seja *Pslice* uma fatia executável de um programa *Porg,* S seja o conjunto de instruções em *Pslice* e *ProjS (Porg,* ÿ0) seja a projeção de *T race(Porg,* ÿ0) que contém apenas os traços de instrução produzidos por instruções em S. *Pslice* é uma fatia executável correta de *Porg* se *P rojS(Porg,* ÿ0) for idêntico a *T race(Porg,* ÿ0).

Mesmo que esta definição tenha as mesmas desvantagens que a original, ela permite que ele prove a correção de seus algoritmos com relação a esta formalização.

### 1.1.2 Gráficos de dependência

Como vimos em [Pacalet(2007)] os gráficos de dependência são a base dos cálculos. [Ottenstein e Ottenstein(1984)], então [Ferrante et al.(1987)Ferrante, Ottenstein e Warren] introduzem a noção de PDG *(Program Dependence Graph).* Tal gráfico é usado para representar as diferentes dependências entre as instruções de um programa. Eles o exploram para calcular um *fatiamento não executável* que se preocupa apenas com as instruções que influenciam o valor das variáveis, mas essa representação também será usada para calcular um *fatiamento* executável .

Essa representação, inicialmente intraprocedimento, foi estendida para interprocedimento em [Horwitz et al.(1988)Horwitz, Reps e Binkley] onde é chamada de SDG *(System Dependance Graph).* Neste artigo, Susan Horwitz está interessada em um *fatiamento* com a seguinte definição:

**Definição (cortando selon [Horwitz et al.(1988)Horwitz, Reps e Binkley])**

Uma fatia de um programa em relação ao ponto do programa p e à variável x consiste em um conjunto de instruções do programa que podem afetar o valor de x em p.

Mas também especifica que x deve ser definido ou usado em p. De fato, quando o grafo é construído, o *fatiamento* se resume a um problema de acessibilidade a um nó, mas um nó representa um ponto do programa e contém apenas as relações referentes às variáveis que ali aparecem.

Vimos em [Pacalet(2007)] que essa limitação pode ser eliminada mantendo uma correspondência entre os dados em um ponto do programa e os nós do grafo.

### 1.1.3 Fatias vs. Costeletas

Vimos que *o fatiamento* está interessado no que acontece antes de atingir um determinado ponto.

O conceito de *corte* é semelhante, exceto que está interessado nas instruções que serão influenciadas por um determinado dado em um determinado ponto.

### 1.1.4 Critérios *de divisão*

Até agora falamos apenas do chamado fatiamento *estático* **que** está interessado em todas as execuções de um programa, em oposição ao *fatiamento* **dinâmico** que considera a projeção do programa na execução de um determinado conjunto de entrada, ou **quasi-estático** que fixa apenas parte das entradas.

Uma versão mais geral que inclui ambos é chamada *fatia* **condicional** ([Fox et al. (2004)Fox, Danicic, Harman e Hierons]). Isso envolve especificar o estudo de caso na forma de propriedades no conjunto de entradas (pré-condições).

Posteriormente, a noção de *fatiamento* **condicional retrógrado** foi introduzida para vários propósitos diferentes:

— a primeira técnica descrita em [Comuzzi e Hart(1996)] e denominada p-slice, enfoca as instruções que participam do estabelecimento da pós-condição, — enquanto a segunda ([Fox et al. (2001) Fox, Harman, Hierons, e Danicic]) tem um objetivo um pouco diferente porque é visto como um auxílio à prova permitindo que o programa seja eliminado de todos os casos em que a propriedade é verificada (automaticamente), para manter apenas aqueles para os quais o resultado não é certo para que o usuário pode se concentrar em verificar esta parte.

Em [Harman et al.(2001a)Harman, Hierons, Fox, Danicic e Howroyd], esses autores continuam o trabalho anterior combinando a propagação direta e a propagação regressiva. Alguns exemplos e a apresentação de uma ferramenta que implementa o trabalho dessa equipe são apresentados em [Daoudi et al.(2002)Daoudi, Ouarbya, Howroyd, Danicic, Marman, Fox e Ward].

Eles definem seu objetivo da seguinte forma:

**Definição (corte pré-pós condicionado)**

As instruções são removidas se não puderem levar à satisfação da negação da pós-condição, quando executadas em um estado inicial que satisfaça a pré-condição.

- a terceiro abordagem apresentada em ([Chung et al.(2001)Chang, Lee, Yoon e Kwon] : também envolve especificar o critério de divisão por um pré-pós e combinar a propagação para frente e para trás de propriedades para manter apenas o que participa do estabelecimento da postagem no contexto da pré-condição. muito brevemente

Esses novos tipos de *fatiamento* não são mais baseados apenas em relacionamentos de dependência, mas requerem várias técnicas complementares, como a execução simbólica ou o cálculo do WP.

Como parte de nosso estudo, parece interessante explorar esse caminho visto que temos (ou teremos) tais ferramentas de análise no Frama-C.

### 1.1.5 Transformações

Em nosso trabalho anterior, exploramos algumas transformações de programas como especialização de função (e, portanto, transformação de chamada) e também propagação constante.

Vários artigos, incluindo [Harman et al.(2001b)Harman, Hu, Munro e Zhang], apresentam o que eles chamam de *fatiamento* **amorfo ,** no qual envolve a combinação *de corte* e técnicas de transformação de programa.

[Ward(2002)] (por algum motivo) prefere falar sobre *fatiamento* **semântico**

### 1.1.6 Ferramentas

Podemos citar duas ferramentas que implementam esta técnica em programas C-ANSI: — *unravel,*
apresentado em [Lyle e Wallace (1997)], está disponível na web (cf.
[[Desvendar()]]) ;
— e o *Wisconsin Program-Slicing Tool* ([Reps(1993)]), é comercializado pela *Gramma Tech* sob o nome *CodeSurfer* (cf. website [[CodeSurfer()]]).

Essas ferramentas permitem:
— selecionar as partes do código úteis para calcular uma variável em um ponto de verificação, — visualizar as instruções selecionadas no código-fonte do usuário, — e realizar operações de união e interseção entre vários descontos.

### 1.1.7 Para mais informações

Um estado da arte em 1995 é apresentado em [Tip(1995)].
Mais recentemente, [Xu et al. (2005) Xu, Qian, Zhang, Wu e Chen] fornecem uma revisão abrangente dos desenvolvimentos até 2005.

Além disso, um site - cujo endereço é dado na bibliografia sob a referência [SlicingWebLinks()] - manteve uma lista de projetos e uma bibliografia muito impressionante sobre *slicing* até 2003, mas este site não parece mais mantido.

## 1.2 O que queremos fazer

O documento [Baudin(2004)] especifica uma ferramenta de auxílio à análise de impacto depois de analisadas as necessidades nesta área. O módulo *de fatiamento* é apresentado como um componente importante desta ferramenta.

Vamos resumir aqui as principais características da ferramenta que serviu como especificação inicial: —

a ferramenta é usada para construir um programa que pode ser compilado a partir de um programa fonte, tendo os dois programas o mesmo comportamento em relação aos critérios fornecidos pelo do utilizador ;

— a construção deste programa de resultados é um processo interativo. - **comandos** de ferramentas

— correspondem a funções internas (em Ocaml) e devem permitir o acesso às diversas funcionalidades.

Eles podem ser usados diretamente, combinados em scripts *para* realizar operações mais avançadas ou até mesmo serem usados para construir um protocolo de comunicação com uma interface gráfica; — comandos **de consulta**

— são usados para consultar o sistema. Eles não devem modificar o estado interno da ferramenta. Mesmo que façam certos cálculos para responder, e que armazenem esses resultados para evitar um recálculo posterior, isso deve ser transparente para a continuação dos tratamentos; — os chamados comandos **de ação**

— devem possibilitar a construção do resultado do *fatiamento.* Esses comandos devem poder ser encadeados e combinados;

— a ferramenta mantém uma **lista de espera** de ações a serem executadas. Isso é útil porque alguns tratamentos podem ser divididos em várias ações que o usuário pode optar por aplicar ou não. Isso permite um controle mais fino dos cálculos; — o programa resultante só pode ser gerado quando a lista de espera estiver vazia; — o processamento deve ser modular; — cada função do programa fonte pode: — não aparecer no resultado, — ser simplesmente copiada no resultado, — corresponder a uma ou mais funções especializadas do resultado.

— na representação interna do resultado, as instruções são associadas a uma marcação que indica se devem ser ocultadas ou, se não, o motivo de sua presença, ou seja, por exemplo, se participam do controle ou do valor dos dados selecionados , etc

— identificação de instruções insignificantes (ou supérfluas)
— trata-se de saber distinguir o que se deve acrescentar quando não se faz uma especialização. Tome por exemplo o caso de uma chamada *L : f(a,* b); onde apenas *a* é necessário para o cálculo que nos interessa, mas que não queremos nos especializar em f(a);. Será então necessário adicionar o que permite calcular *b* em *L* às instruções selecionadas, mas preservando a informação de que são *supérfluas ;* — especialização de código
— trata-se de poder realizar
transformações sintáticas para reduzir ao máximo o código. Por exemplo, se depois de uma instrução y = x++;, só precisamos de x, devemos ser capazes de decompor essa instrução para manter apenas
x++;
Sendo este tipo de transformação já feito a montante do *módulo de fatiamento,* o problema já não se coloca para este tipo de instrução, mas continua a existir para a especialização de chamadas de função. De fato, se uma função *f* leva, por exemplo, dois argumentos dos quais apenas um é usado para calcular o que nos interessa, queremos ser capazes de construir uma função f1 que leva apenas um argumento e transformar a chamada para *f* em uma chamada para f1.
Além disso, também é possível considerar instruções de transformação quando o valor constante de uma variável é conhecido.

## 1.3 Mapa de documentos

Este documento apresenta apenas os princípios do que foi desenvolvido. Para mais detalhes sobre a implementação ou os comandos disponíveis, convida-se o leitor a consultar a documentação do código, pois ela sempre será mais detalhada e atualizada que este relatório.
Como a caixa de ferramentas *de corte* é um plugin Frama-C, sua interface é definida no módulo Db.Slicing. Além disso, a introdução da documentação interna do módulo dá uma ideia da arquitetura e dá pontos de entrada.

Os princípios para calcular os gráficos de dependência nos quais *o fatiamento* é baseado são apresentados em [Pacalet (2007)].

O Capítulo 2 apresenta como as instruções de uma função são marcadas para realizar uma redução intraprocedimento.

Em seguida, o capítulo 3 expõe como os cálculos são organizados para fazer uma propagação das reduções para toda a aplicação, e como propomos especializar as funções.

Por fim, o Capítulo 4 apresenta como utilizar o módulo desenvolvido e, portanto, oferece uma visão mais voltada para um usuário em potencial.

Sendo este relatório um documento de trabalho, irá evoluir à medida que o trabalho avança: — a versão 1.0 de 27 de junho de 2006 apresentou os desenvolvimentos em curso relativos aos grafos de dependência e à marcação de uma função. A apresentação do trabalho a ser feito para o manejo de um *fatiamento* interprocedimento foi menos detalhada por ser mais prospectiva. Seguiu-se um capítulo contendo um manual do usuário. Por fim, a conclusão apresentou o estado atual da ferramenta, e as perspectivas de evolução.

— na versão 2.0 de 26 de junho de 2007, a parte do gráfico de dependência foi extraída porque é assunto de um documento separado. Desta vez, o corte interprocedimento é apresentado em detalhes . As partes do documento de especificação cujas ideias não foram mantidas foram movidas para o apêndice D para registro. A parte sobre o uso da ferramenta foi mantida por enquanto, mesmo que não seja muito específica para o uso do plug-in de fatiamento .

No final de 2008, este documento pode ser considerado estável, pois mesmo que o desenvolvimento evolua, não se trata mais de modificar as funcionalidades básicas, mas sim de ajustar pontos de detalhe para melhorar a precisão, por exemplo.

# Capítulo 2

# Redução de uma função

## 2.1 Função especializada

Chamamos de **função especializada** a redução de uma função fonte obtida após a aplicação de uma ou mais ações. Este capítulo explica como essa redução é representada e calculada.

O colapso de uma função deve distinguir as instruções visíveis das invisíveis. A informação booleana, anexada a cada instrução, deveria assim ser suficiente para indicar se esta pertence ou não à função especializada, mas a implementação de uma ferramenta de navegação favorece o enriquecimento da informação calculada. Decidimos então ter uma anotação mais precisa do fluxo de dados, que chamamos de **marcação,** para especificar o motivo da presença ou ausência de determinado elemento. Isso pode possibilitar a visualização do impacto das instruções de uma função em um ponto do programa (controle, dados utilizados, declaração necessária, etc.)

O documento que apresenta o cálculo do PDG expõe os elementos que compõem o gráfico de dependência. A maioria corresponde a uma instrução. Algumas exceções no entanto:

— os elementos que representam as entradas/saídas de uma função não estão associados a uma instrução,
— o rótulo de uma instrução é representado por um elemento além daqueles que representam a instrução,
— uma chamada de função é representada por vários elementos.

Não consideramos, a princípio, a especialização das funções chamadas que serão estudadas no capítulo 3. A chamada de função é, portanto, vista por enquanto simplesmente como uma instrução representada por vários elementos.

Por outro lado, distinguimos a visibilidade de um rótulo da instrução associada. A seguir, o rótulo será frequentemente considerado como uma instrução por si só.

Podemos, portanto, dizer que a função especializada contém uma **marcação** que mapeia uma **marca** para as instruções e rótulos de uma função.

## 2.2 Marcação

Calculamos a marcação de uma função em resposta a uma consulta. Isso geralmente é expresso em termos de elementos do CEO.

Inicialmente, a marcação pode conter o resultado de cálculos anteriores ou ser recém-criada, ou seja, vazia.

O cálculo elementar é muito simples: consiste em percorrer o PDG, calcular a marca para cada elemento e associá-lo à instrução ou rótulo correspondente combinando-o com o possível valor anterior.

Uma ideia do algoritmo aplicado é apresentada em A.2.

### 2.2.1 Mudando para um ponto do programa

Uma primeira solicitação consiste em marcar os elementos do fluxo que são usados para determinar o que permite passar para um ponto de programa.

A marca propagada é chamada de **marca de controle** e é denotada por *mC.*

Para realizar este cálculo, a marca é propagada nas dependências de controle do ponto escolhido, depois recursivamente em todas as dependências dos pontos assim selecionados. Isso corresponde a marcar *mC* todos os elementos do conjunto RL0.

**Exemplo 2**

| | |
|---|---|
| *mC* x = y+1; a = 0; b<br>= 10;<br><br>...<br>*mC* se (x > 0) {<br>...<br>L: a += b;<br>...<br>} | Trata-se aqui de selecionar o que controla a passagem para o ponto L. O teste x>0 é, portanto, marcado *mC,* assim como o cálculo de x do qual este teste depende. |

### 2.2.2 Valor dos dados

Você também pode pedir para selecionar o que torna possível calcular um dado em um ponto do programa. Podemos, por exemplo, pedir para manter apenas o que permite calcular uma das saídas de uma função.

O primeiro passo é encontrar o elemento correspondente, ou elementos, no grafo. Elementos particulares representam as saídas da função. Para cálculos internos, podemos usar a identificação de uma atribuição para falar sobre os dados afetados,

ou você deve ter o estado usado durante a construção do grafo. É então possível encontrar os elementos que participaram do cálculo em um ponto de qualquer datum, pelo menos quando o ponto considerado estiver dentro de seu alcance.

A marca associada ao cálculo de um dado é chamada **de marca de valor** e é denotada por *mV.* É utilizado, como o próprio nome sugere, para anotar os elementos envolvidos no cálculo do valor dos dados solicitados.

Os dados usados para calcular o endereço da caixa modificada (parte esquerda da atribuição) são anotados por uma **marca de endereço** *(mA).* Isso é, por exemplo, o cálculo do índice de um elemento de array ou um ponteiro.

Os elementos que atingem o ponto do programa são marcados como *mC.*

**Exemplo 3**

```
mCA x = y+1; mV b =
 10;
                ...
 mC se (x > 0) {
                ...
               L: t[x] = b;
                ...
               }
```

Trata-se de selecionar o que participa do cálculo da instrução localizada no ponto L. O teste x>0 é, portanto, marcado como *mC ,* assim como o cálculo de x do qual este teste depende. O valor de b participa do cálculo da parte direita e, portanto, é marcado em *mV.* A parte esquerda depende de x que deve, portanto, ter a marca *mA.* Podemos, portanto, ver que x combina duas informações e, portanto, é marcado como *mCA.*

### 2.2.3 Elementos desnecessários

Vimos que a marcação é relativa às instruções, e que certas instruções (chamadas de função) são representadas por vários elementos do PDG. Se você não quiser quebrar as instruções (ou seja, especializar as chamadas de função), pode acontecer que a mesma instrução corresponda a elementos visíveis e outros invisíveis. O último, e suas dependências, são então marcados como **supérfluos** (marque mS). Algumas instruções tornam-se então visíveis, embora não sejam estritamente necessárias para o cálculo solicitado.

**Exemplo 4**

```
int G;
int f (int x, int y) {
    G = x + 1;
    retornar y + 2;
}
int g (vazio) { int a
    = 1; int b =
    a+1; retornar f
    (b, a);
}
```

Se o usuário pedir para selecionar o que permite calcular o valor de retorno de g, na verdade precisaríamos apenas do valor de a, mas como não especializamos f, também devemos marcar a instrução que calcula b como supérflua.

### 2.2.4 Marcas Registradas

Em resumo, as marcas possíveis são:

| |
|---|
| *mV :* marca de valor |
| *mC :* marca de seleção |
| *mA :* marca de endereço |
| *mS :* marca para item supérfluo |

As marcas *mV, mC, mA* podem se sobrepor quando um elemento participa do cálculo por vários motivos. Por exemplo , *mV A* denotará a marca associada a um elemento que participa do valor e do cálculo do endereço. Posteriormente, qualquer marca desta família é chamada de marca *mCAV .*

A marca *mS* é o elemento neutro do traje.

# Capítulo 3

# redução interprocedimento

## 3.1 Finalidade

A filtragem elementar vista anteriormente é apenas um pré-requisito para a redução de uma aplicação, pois quando uma função foi reduzida, suas chamadas também devem ser processadas para ficarem consistentes, e você também pode querer reduzir as funções chamadas.

### 3.1.1 Especificação do problema

Para estender a marcação intraprocedural já realizada para uma computação interprocedural, uma primeira abordagem seria propagar a marcação por meio de chamadas de função, mas isso leva a muitas funções especializadas se todas as chamadas forem diferenciadas ou a muita perda de precisão se combiná-los. Este problema é explicado com mais detalhes em §3.3.3. O objetivo é, portanto, obter um compromisso aceitável e configurável.

Assim que se decide poder ter várias especializações da mesma função fonte na aplicação final, o mecanismo que permite obter um resultado coerente torna-se relativamente complexo. Por exemplo, quando estamos interessados no cálculo de certos dados *d* em uma função g, vimos no capítulo anterior como calcular a marcação dos elementos do PDG para poder reduzir g. Se *g* chama h, e todas as saídas de *h* não são necessárias para o cálculo de *d* em g, também podemos solicitar a especialização de h. Além disso, se *g* for chamado, é possível substituir uma ou mais chamadas por uma chamada para g1. Esse processamento deve ser aplicado recursivamente, pois assim que modificamos a marcação de uma função, também podemos precisar modificar as funções chamadas e chamadoras.

Decidiu-se, portanto, decompor um tratamento completo em várias ações elementares perfeitamente definidas. Eles podem então ser combinados automaticamente de acordo com as opções de nível superior.

### 3.1.2 Organização dos resultados

O processo completo consiste em aplicar sucessivamente um certo número de consultas expressas pelo usuário para construir uma nova aplicação. Chamaremos de **projeto atual** o estado desse novo aplicativo em um determinado momento. É composto:

— um conjunto de resultados (inicialmente vazio) sobre as funções como elas eram apresentado em 2, ou seja, funções especializadas;
— e uma lista de ações que ainda devem ser aplicadas para obter uma aplicação coesa interesse.

Quando o usuário expressa solicitações, elas são traduzidas em **ações** que são armazenadas na lista de tarefas pendentes. As ações são aplicadas em sequência, ou seja, você não pode aplicar uma ação se a anterior não tiver sido concluída. A aplicação de uma ação possivelmente pode gerar novas ações.

Uma ação pode, em certos casos, ser excluída ou modificada: este ponto é abordado em §3.6.1.

Ao final da análise, não há mais ação a ser aplicada, podendo ser gerada a aplicação resultante.

### 3.1.3 Chamada de função

Durante a marcação intraprocedimento são obtidas as notas estritamente necessárias para o cálculo solicitado. Em particular, podemos extrair as assinaturas de chamadas de função. Para cada um deles, assim que houver pelo menos um elemento visível, você terá que decidir qual função será chamada. Vimos, por exemplo, que podemos optar por não especializar as chamadas de função: as entradas invisíveis das chamadas terão que ser marcadas como supérfluas.

### 3.1.4 Propagação para funções chamadas

Para deixar a possibilidade de desenvolver a ferramenta, decidimos oferecer várias possibilidades: —
uma especialização por chamada: para cada chamada, determinamos as saídas úteis e gera a função chamada correspondente (se ainda não existir),
— nenhuma especialização de função: assim que precisamos de uma das saídas de uma função, deixamos a chamada e, portanto, precisamos selecionar todos os argumentos de chamada,
— apenas uma especialização por função: agrupamento de todas as especializações nascidas necessários para a aplicação em uma única função,
— agrupamento de funções especializadas *a posteriori,* a pedido do usuário.

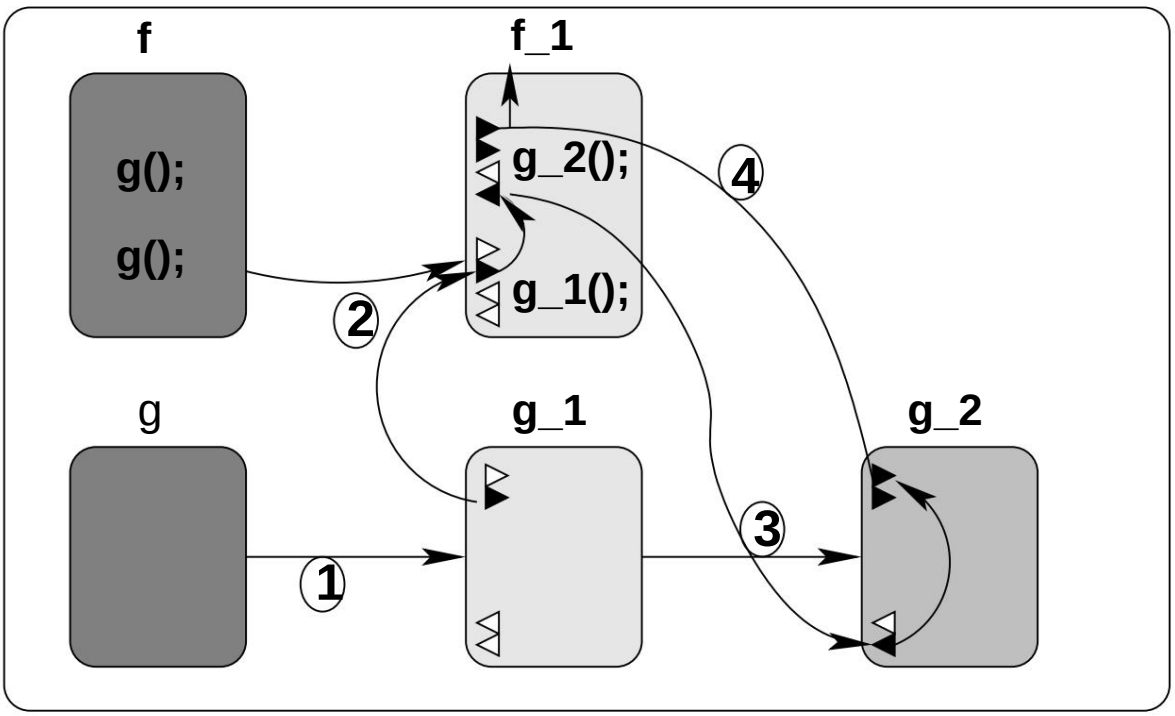
### 3.1.5 Propagação para chamar funções

Quando uma função especializada foi calculada, pode-se querer chamá-la em vez da função inicial e, assim, propagar a redução. Para fazer isso, você precisa ser capaz de criar uma consulta que designe o ponto de chamada e a função especializada a ser usada. Esta consulta equivale a estar interessada no cálculo das entradas visíveis da especialização acima do ponto de chamada.

A aplicação de tal solicitação pode, por sua vez, gerar uma solicitação de especialização, possivelmente da mesma função do exemplo abaixo.

**Exemplo 5**

A figura a seguir mostra um exemplo das diferentes etapas da transformação de chamadas para *g* em *f* por chamadas para a função filtrada g1 e como se pode considerar a propagação da informação.




1. cálculo da função g1 : apenas a segunda entrada é visível, as saídas não são visíveis; 2.
substituição da segunda chamada a *g* em *f* por uma chamada a g1 e seleção dos elementos que permitem calcular a entrada visível. Vemos que esse cálculo requer o cálculo da segunda saída da primeira chamada para g1. Portanto, é necessário calcular uma segunda função g2.
3. cálculo da função g2 : deve ter os mesmos elementos de g1 e também calcular sua segunda saída. Vemos que isso requer a visibilidade da primeira entrada.
4. seleção do que permite calcular a primeira entrada de g2 em f.

### 3.1.6 Seleção de chamadas de função

Além dos critérios *de fatiamento* já vistos, que consistem em selecionar um dado em um ponto do programa, o aspecto interprocessual leva à definição de um novo critério que permite manter apenas o que permite chamar uma determinada função e calcular os contextos de chamada.
A função pode ser: —
uma função externa para a qual não temos o
código, — uma função presente no código
fonte, — uma função resultante de uma
filtragem anterior, podendo optar
por selecionar: — apenas o que permite 'alcançar pontos do programa pede isso
função,
— ou também o que permite calcular as entradas.

Essa filtragem pode, por exemplo, ser usada para reduzir um aplicativo para permitir uma análise de vazamentos de memória. Basta selecionar as funções de alocação (alloc, malloc, etc.) e a função de liberação (livre).

## 3.2 Princípio e notações

Recordamos aqui os aspectos do cálculo intraprocedimento, e diversas notações, que serão utilizadas posteriormente. Contamos amplamente com a representação de entradas/saídas e chamadas de função no PDG apresentado em [Pacalet (2007)].

### 3.2.1 Marcas elementares

Consideramos que dispomos de um conjunto de marcas elementares, aqui não detalhadas, dotadas de:

— uma ordem parcial (ÿ), — uma operação de união (+), — um menor elemento *(ÿm)* usado para marcar elementos invisíveis, — e um maior elemento *(m )* correspondente à marcação dos elementos de função fonte.

Consideramos também um elemento chamado *Spare* tal que: — *ÿmm = ÿm* ÿ *m = Spare* ÿ *Spare* ÿ *m* Temos então:

$$ÿmm = ÿm \text{ ÿ } m + Sobressalente = m$$

Esta marca é, portanto, a menor que torna um elemento visível. É usado para marcar os chamados *elementos supérfluos,* ou seja, aqueles que não seriam necessários se a redução fosse mais precisa.

### 3.2.2 Marcação

Vimos no capítulo 2 que temos um procedimento que permite marcar uma instrução e propagá-la de acordo com as dependências (PDG).

Durante este cálculo, as relações de dependência entre as entradas e saídas das funções chamadas são dadas pela especificação [1].

No resultado obtido, quando um elemento de função e1 é utilizado por um elemento de função e2, a marca m1 associada a e1 é maior ou igual à marca m2 associada a e2 porque m1 é a união das marcas de todos os elementos que dependem em e1.

#### 3.2.3 Funções

Cada função de origem pode ser especializada uma ou mais vezes. No nível intraprocedimento , isso significa que é dada uma certa marcação.

A seguir, notamos:

1. pode ser um problema fazer a quebra de ramificação, porque as dependências podem ser recolhidas por tal especialização.

— *f, g, h, ...* qualquer função (fonte ou especializada), — uma função fonte tem um índice 0: f0, g0, h0, ... — as funções especializadas têm outro índice: *fi , gj , hk, ...* (salvo indicação em contrário, quando o índice é especificado, assume-se, portanto, que é diferente de 0).

### 3.2.4 Entradas/saídas

O gráfico de dependência contém elementos específicos que representam as entradas e saídas da função. Denotamos por *InOut(f)* o conjunto desses elementos para a função f.

Não especificamos qual função *f* é, pois consideramos o mesmo conjunto de entradas/saídas para todas as especializações:

$$\ddot{y}i.InOut(f0) = InOut(fi)$$

### 3.2.5 Assinatura

Chamamos de **assinatura,** e denotamos *sig,* uma função que associa uma marca aos elementos de um conjunto de entradas/saídas. *s*
nós definimos que representa uma assinatura em que todos os elementos têm uma marca m :

$$dizer = {}_{s}\ \ddot{y}\ \ddot{y}e\ \ddot{y}\ \text{dom(sig).sig(e)} = {}_{m}$$

e similarmente, *ÿs* que representa uma assinatura cujos elementos têm uma marca *ÿm :*

$$sig = \ddot{y}s\ \ddot{y}\ \ddot{y}e\ \ddot{y}\ \text{dom(sig).sig(e)} = \ddot{y}m$$

Definimos *sigf(f)* a **assinatura de uma função** *f* tal que: — *dom(sigf(f))* = *InOut(f)* — sigf(f0) = — *sigf(fi)* *(e)* é a marca ${}_{s}$ atribuída a *e* em *fi .*

Às vezes denotamos *inSigf(f)* e *outSigf(f)* as funções cujo domínio é reduzido às entradas ou saídas.

### 3.2.6 Chamada de função

Como estamos falando aqui sobre a parte interprocedural do processamento, estamos interessados principalmente em chamadas de função. Considera-se que é conhecido como identificar exclusivamente uma chamada de função c, (por exemplo, pela identificação da função de chamada e um elemento do PDG, ou um número de sequência). A seguir, para especificar o nome da função chamada, observaremos *cg* uma chamada para uma função g.

Denotamos *call(f)* o conjunto de chamadas de função de f. Observe que a especialização não é especificada, porque:

$$\ddot{y}i.call(fi) = \text{call(f0)}$$

Para cada chamada de função *cg* de *fi ,* o grafo de dependência contém elementos que representam as entradas/saídas da função chamada.

Definimos a **assinatura de uma** chamada de função *c* na função *fi ,* e notamos *sigc(c,* fi), a função que fornece as marcas desses elementos em *fi .*

## 3.3 Cálculo interprocessual

### 3.3.1 Gerenciamento de funções

No nível intraprocessual , uma função especializada é caracterizada por uma certa marcação de seus elementos. No interprocedimento, estamos interessados na propagação de marcas para chamadas de função. Propõe-se assim adicionar às funções especializadas uma função *Call ,* descrita em §3.3.2, que associa, a cada chamada de função, a identificação da função a chamar.
O objetivo é substituir algumas chamadas por chamadas para funções especializadas.

A redução de uma aplicação consiste na construção de um **projeto** que contém uma lista de funções, inicializada com a lista das funções fonte, e complementada durante o estudo pelas funções especializadas calculadas.

O objetivo é obter um projeto coerente, conforme definido em §3.3.5.

### 3.3.2 Chamadas de funções

Chamamos de Call(fi) a função que mapeia cada chamada de *fi* para uma função chamada.
*Call(fi)(c)* portanto, fornece a função chamada por *fi* para a chamada c.

Denotamos *Call(fi)(c)* = *ÿc* quando a chamada é invisível.

Além disso, *Call(fi)(c)* = a função *c* significa que a chamada ainda não foi atribuída, ou seja, a ser chamada ainda não foi escolhida.

Se a função chamada for determinada acessando um ponteiro de função, ela só é considerada na *Chamada* se a função chamada for estaticamente conhecida a partir da análise de valor. Nos demais casos, deixaremos a chamada como está na função source.

Funções de origem necessariamente chamam funções de origem:

*ÿc* ÿ chamada(f), *ÿg. Chamada(f0)(c)* = g0

Quando a chamada é atribuída, corresponde necessariamente a uma especialização da função inicialmente chamada:

*ÿc* ÿ chamada(f), *Chamada(f0)(c)* = g0 ÿ *ÿfi* .(Chamada(fi)(c) = ÿcÿChamada(fi)(c) = cÿÿj. *Chamada(fi) (c) = gj )*

Além disso, se *Call(fi)(c) = gj* queremos que a assinatura da função chamada *sigf(gj )* seja *compatível* com a assinatura da chamada *sigc(c,* fi). Agora vamos ver o que isso significa...

### 3.3.3 Problema de compatibilidade de assinatura

A primeira ideia que surge quando se deseja propagar a marcação para as funções chamadas é aplicar nas dependências as mesmas regras de propagação do cálculo intraprocedimento. Mas quando você deseja usar a mesma função especializada em diferentes contextos, isso leva a uma perda muito grande de precisão. Queremos, por exemplo, poder marcar como *Spare* as entradas que não são realmente usadas para uma determinada chamada, mesmo que tenham outras marcas em outro lugar.

Por exemplo, no seguinte caso:

```
int X, Y;

void g (int x, int y) {
    X=x; e = e; }
```

```
int f_a (int x_a, int y_a)
        { g (x_a,
    y_a); retornar X; }
```

```
int f_b (int x_b, int y_b)
        { g (x_b,
    y_b); retorna Y; }
```

se o usuário solicitar a marcação das saídas 0 de f_a e f_b, e a construção de uma única especialização para g, gostaríamos que x_a e y_b fossem marcados como *Spare*

Para obter tal comportamento, decidimos estender a marcação da seguinte maneira.

### 3.3.4 Par de marcas

Para manter a precisão máxima, é necessário distinguir as seleções realmente feitas pelo usuário das marcas introduzidas por uma aproximação. Para isso, cada elemento de uma especialização não é mais marcado por uma única marca, mas por um par < m1, m2 > onde: a marca m1 corresponde à propagação de

uma seleção pelo usuário na função de origem e em seus chamadores, MAS não o propagamos diretamente nas funções chamadas porque isso leva a uma perda muito grande de precisão. Deve-se notar que m1 não pode, em princípio, ser *Spare,* a menos que o usuário o solicite explicitamente.

— a marca m2 corresponde à propagação de uma marca m1 de um chamador. Portanto, quando a saída de uma chamada tiver uma marca m1, a saída correspondente da função chamada será marcada como m2. Além disso, se uma função tiver uma entrada marcada como *m* =< m1, m2 > com m2 = *ÿm,* seus chamadores devem propagar *m* =< m1, *Spare* > na entrada correspondente. Atenção, como veremos adiante, isso não significa que as entradas dos chamadores tenham a marca *m ,* pois é necessário combinar essa marca com as marcas do chamador.

Do ponto de vista do usuário, a marca associada a um elemento é a união de m1 e m2.

No exemplo anterior, as entradas *x* e *y* de *g* serão marcadas em m2, e isso realmente levará à marcação *Spare* das entradas inúteis em f_a e *f_b.*

### 3.3.5 Coerência do projeto

Antes de ver como calcular o resultado, vamos ver as propriedades que uma marcação final deve ter para ser consistente. Os elementos que nos interessam são as assinaturas de funções e chamadas.

A seguir falamos apenas de marcas propagadas, sabendo que além disso, cada uma também pode conter uma marca colocada manualmente pelo usuário.

— as marcas das saídas de uma chamada *(cg, fi)* dependem apenas do contexto da chamada, ou seja, as marcas de suas dependências em *fi .* No entanto, parece que uma marca

m2 só pode vir de marcas m2 nas saídas de *fi* , exceto a marca *Spare* que pode vir de entradas de chamada de função. — as

marcas das saídas de uma função especializada são determinadas pelas marcas das saídas das chamadas a esta função. Só podemos ter m1 = *ÿm* se o usuário tiver explicitamente colocado uma marca na saída. A marca m2 é a união das marcas m1 e m2 das saídas de chamada.

— as marcas das entradas de uma função são determinadas exclusivamente pela propagação das marcas dos outros elementos da função. Uma marca m1 decorre necessariamente de uma seleção do usuário na função considerada ou em uma de suas funções chamada.

— as marcas de entrada de uma chamada *(cg, fi)* são uma combinação das marcas de entrada de *ge* as saídas de *(cg, fi)* Se a função chamada for substituída, essas marcas de entrada devem ser recalculadas.

Em resumo, considere uma entrada *e* e uma saída *s* de uma função especializada *gi* , na nota *m(e) = sigf(gi)(e), m(s) = sigf(gi)(s).*

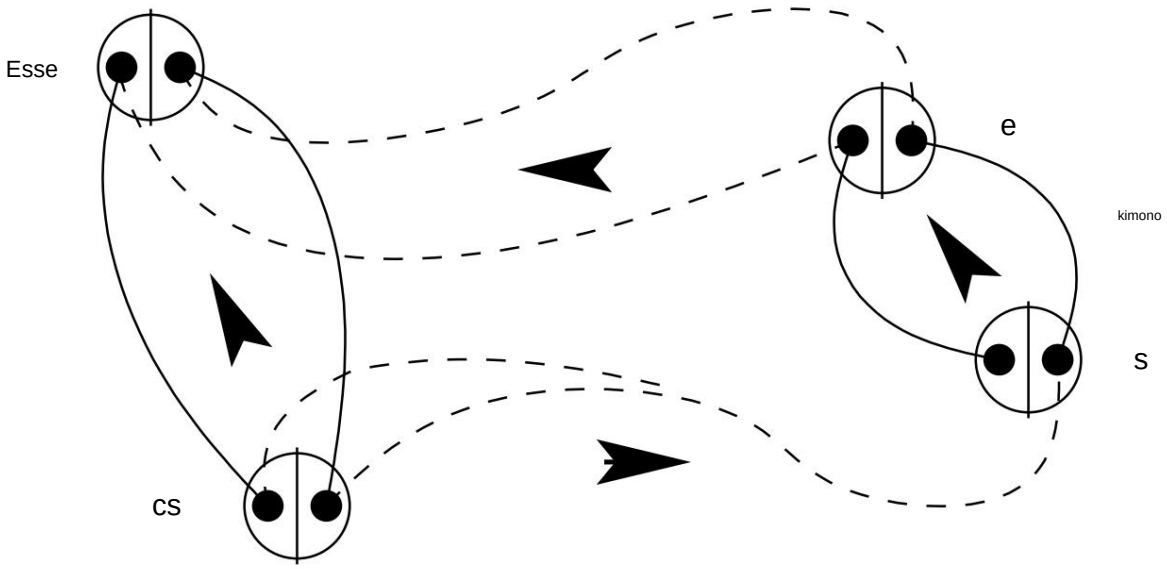



**Figura 3.1 –** *Propagação de marcas em chamadas de funções*

Seja *c* qualquer chamada para esta função, notamos *m(ce) = sigc(c)(e)* e *m(cs) = sigc(c)(s).*
Para ter um projeto coerente, devemos ter as seguintes propriedades:

— m2(s) ÿ *m1(cs) + m2(cs)* —
m1(e) ÿ *m1(ce)* —
m2(e) = *ÿm* ÿ *m1(ce)* ÿ *Sobressalente*

Além disso, se a saída *s* depende da entrada *e :*

— *m1(cs)* ÿ m2(s) ÿ m2(e) ÿ *m2(ce)* —
*m1(cs)* ÿ *m1(ce)* —
m1(s) ÿ m1( e) ÿ *m1(ec)*

Vimos que o cálculo do resultado final é dividido em ações elementares. Entre duas ações, o projeto deve ser **parcialmente consistente,** ou seja, para toda *Call(fi)(c) : — Call(fi)(c) = ÿc* ÿÿ *sigc(c, fi) =*

*ÿs — Call(fi )(c)* = e deve haver uma ação pendente para atribuir a chamada, *— Call(fi)(c) = gj* e: *— gj* deve existir e ter uma assinatura compatível

com l call, ou uma ação de modificação pendente (porque neste caso, não sabemos necessariamente avaliar a compatibilidade das assinaturas).

— ou *gj* ainda não existe, mas tem uma ação de compilação pendente.

Quando a lista de ações estiver vazia, o projeto deve ser **consistente ,** ou seja, todas as chamadas de função devem ser atribuídas a uma função com uma assinatura compatível com a da chamada.

## 3.4 Ações para computação interprocedimento

A dificuldade do tratamento é vincular as ações corretamente para obter uma aplicação coerente. Além disso, para ter uma caixa de ferramentas suficientemente flexível, as ações devem corresponder a processos suficientemente elementares e perfeitamente especificados (especialmente seu comportamento em relação às diferentes opções).

Fica, portanto, decidido que qualquer ação deve ser decomposta e traduzida em uma sequência **de ações elementares** definidas abaixo. Devem ser simples e o menos configuráveis possível, sendo a flexibilidade proveniente da tradução de uma ação de alto nível em ações elementares. Estes últimos, portanto, não são todos acessíveis ao usuário, mas podem ser vistos como etapas de cálculo que serão combinadas posteriormente de acordo com as necessidades.

A aplicação de uma ação elementar: — pode eventualmente gerar novas ações, — mas só pode criar ou modificar uma única função (ou nenhuma).

Esta parte, portanto, apresenta primeiro as ações elementares e os detalhes da aplicação de cada uma delas. A seguir, veremos que a geração dessas ações depende do modo de operação escolhido. Os modos propostos são explicados em §3.4.4.

As ações estão resumidas na forma de planilhas de dados, no apêndice C.

### 3.4.1 Criando uma especialização: *NewSlice*

Chamamos de *fi = NewSlice(f0)* a ação que permite construir uma especialização *fi* de f0 . Inicialmente, todas as marcas são definidas para *ÿm* e a função *Call(fi)(c)* = modificações c. O podem ser feitas usando as ações mostradas abaixo. Isso permite um processamento consistente.

### 3.4.2 Editando uma especialização

As seguintes ações modificam a marcação de uma especialização, não criam novas funções.

**Aviso :** por enquanto, as ações **adicionam** marcas. Queremos poder reduzir uma especialização?

Em qualquer caso, quando a marcação de uma função é modificada, é necessário gerenciar suas chamadas de função. Todas as ações de modificação abaixo, portanto, geram uma solicitação *ExamineCalls* que permite que essa tarefa seja executada. Sua aplicação é apresentada em §3.4.2.

**Adicionando uma marca de usuário: *AddUserMark*** A ação

*AddUserMark(fi ,(e,* m) permite adicionar e propagar uma marca em qualquer elemento de *fi .É* uma marca de usuário, portanto é adicionada em m1.

**Propagando uma marca de saída: *AddOutputMarks*** A ação

*AddOutputMarks(fi , outSigc)* é gerada quando *fi* é chamado, mas não calcula o suficiente. Para cada saída s, se *outSigc(s)* =< m1, m2 >, adicione < *ÿm,* m1 + m2 > a outSigf(fi)(s) e propague a nova marca em *fi .*

**Marcando as entradas de uma chamada: *ModifCallInputs*** A ação

*ModifCallInputs(cg, fi)* é gerada quando as marcas de entrada de uma chamada *gj = Call(fi)(cg)* in *fi* são insuficientes para a função chamada. Para cada entrada *e* da chamada, começamos recalculando sua marca de acordo com as marcas de suas dependências.
Isso permite obter um resultado correto mesmo se precisarmos de mais entrada antes de chamar *gj .* Então, se *inSigf(gj )(e)* =< m1, m2 >, adicione < m1, *Spare* > a *inSigc(cg, fi)* se m2 = *ÿm* e < m1, *ÿm* > caso contrário, e propague a marca de notícias assim obtido em *fi .*

**Gerenciamento de chamadas: *ExamineCalls***

A ação *ExamineCalls(fi)* é aplicada automaticamente após qualquer modificação da marcação da função *fi* porque é necessário examinar Call(fi) para verificar sua consistência, ou seja, para ver se as funções chamadas ainda são adequadas .

Para cada chamada, portanto, olhamos para:

— se *outSigc(c, fi) = ÿs* (não usamos as saídas da chamada de função), então
*Call(fi)(c) = ÿc,* —
else (algumas saídas são marcadas):
— if *Call(fi)(c) = ÿc :* nada foi chamado anteriormente, faça como se
*Call(fi)(c) = c* (veja abaixo), — se *Call(fi)*
*(c) =* : nenhuma função ainda foi atribuída a esta chamada. Uma ação *ChooseCall* é responsável por atribuir uma função a esta chamada (consulte §3.4.3). — se *Call(fi)(c) = gj :* devemos comparar *outSigc(c, fi)* com *outsigf(gj )* como faremos para vê-lo.

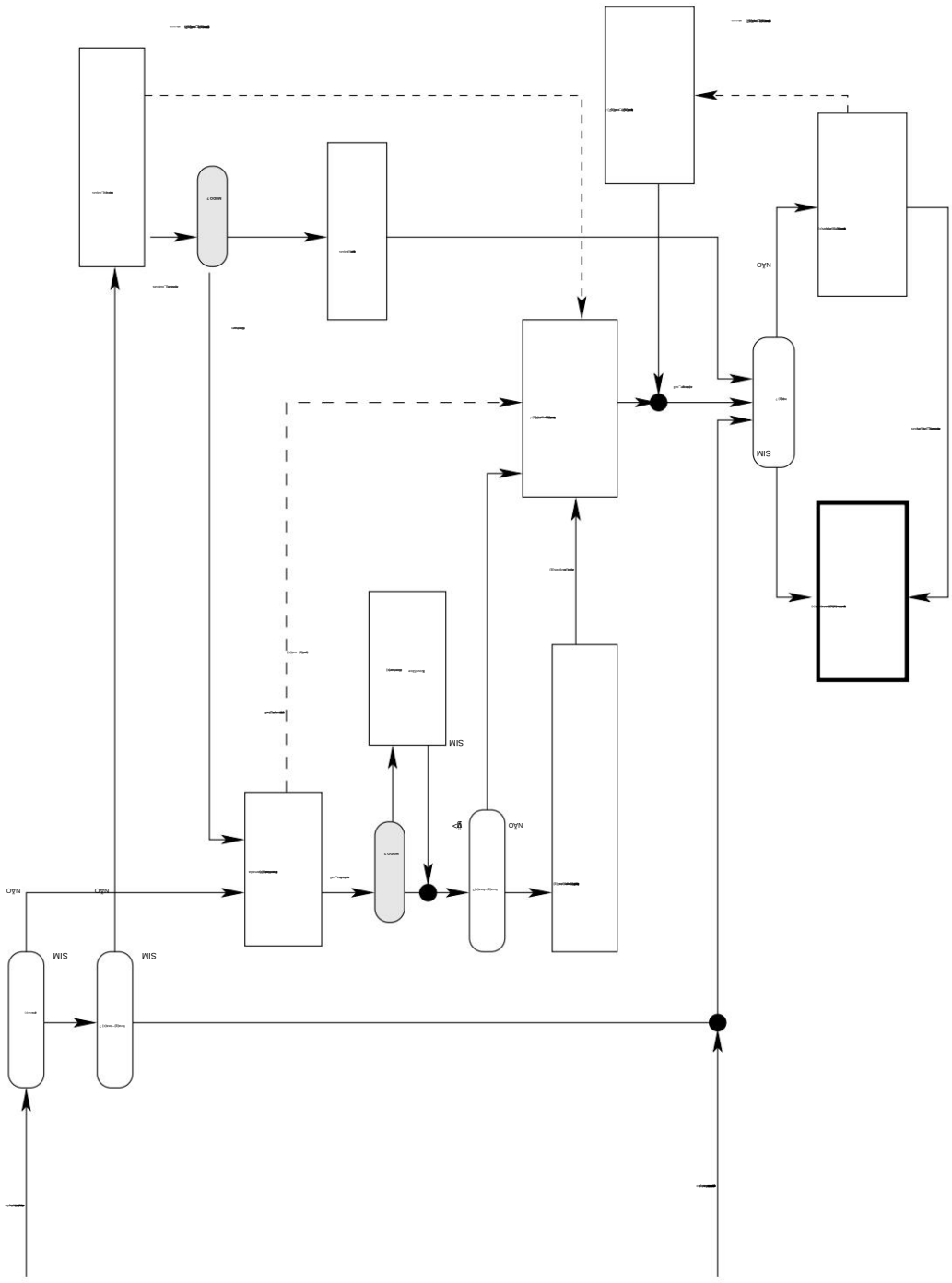
Para que a função chamada *gj* seja adequada, ela deve calcular pelo menos todas as saídas necessárias para esta chamada. Ou seja, se uma saída da chamada for marcada como < m1, m2 > e a saída correspondente de *gj* for marcada como <*m* $_{1}$ , *m* $_{2}$ >, devemos ter:

$$m2 \ \ddot{y}\ m1 + m2$$

Se não for esse o caso, uma ação *MissingOutputs* é gerada (consulte §3.4.3). Poderíamos gerar *AddOutputMarks(gj , outSigc(c, fi)* diretamente , mas isso não permitiria escolher outra função *g* em vez de estender *gj .*

Por outro lado, *ExamineCalls* também se encarrega de propagar os tokens para os chamadores, ou seja, se *fi* for chamado por *hk* e os tokens de suas entradas tiverem sido alterados, propague-os para chamar as entradas. Para fazer isso, geramos uma *ação MissingInputs (cf.* §3.4.3). Pode-se gerar diretamente *ModifCallInputs(cf ,* hk), mas isso não permitiria modificar a chamada de função para escolher chamar outra especialização de *f* em *hk.*

Os passos para atribuir e depois modificar chamadas de função podem parecer complexos à primeira vista, especialmente porque sua sequência depende de um modo de operação que será explicado em §3.4.4 e possivelmente de intervenções específicas do usuário. A Figura 3.4.2 tenta resumir esse processo.



Figo. 3.1: Manipulando uma chamada *c* para uma função *g*

seguindo a modificação das marcas da assinatura de *c* na função de chamada ou das entradas de g.

### 3.4.3 Atribuir ou modificar uma chamada de função

Vimos que ao criar ou modificar a marcação de uma função *fi ,* as ações *ChooseCall(c,* fi), *MissingOutputs(c, fi)* ou *MissingInputs(c, fi)* podem ser geradas quando uma chamada *c* deve ser atribuída ou modificada. Em todos os casos, nos perguntamos a função a ser chamada porque ainda não a escolhemos ou não é mais adequada.

#### Atribuir uma chamada: *ChooseCall*

Uma ação *ChooseCall(c, fi)* pode ser gerada (dependendo do modo escolhido) quando *Call(fi)(c)* deve ser modificado porque a marcação de *fi* mudou: é uma questão de encontrar uma função *gk* para substituir a antiga valor de *Call(fi)(c)* que poderia ser, mas o último não é *ÿc* ou mesmo *gj ,* não é mais adequado.

Durante a aplicação desta ação, além do modo de operação, o principal critério de escolha é a marcação das saídas na chamada *(outSigc(c,* fi)) que é confrontada com a marcação das especializações de *g* já calculadas , possivelmente aplicando um tratamento especial a *gj ,* a função inicialmente chamada.

Dependendo do modo:

— ou encontramos um *gk* adequado , —
ou encontramos um *gk* que queremos usar, mas que não calcula coisas suficientes:
devemos, portanto, criar *AddOutputMarks(gk, sigc(c,* fi)),
— ou não encontramos uma especialização existente e então iniciamos a criação de uma nova especialização a partir de g0, ou a partir de *gj* (a ver...)

Em todos os casos, após ter escolhido *gk,* uma ação ChangeCall é aplicada *(cf.* §3.4.3).

#### Ação *MissingInputs*

Lembre-se que uma ação *MissingInputs(c, fi)* é gerada quando a função *gj* atribuída a *Call(fi)(c) = gj* foi modificada, e que requer a marcação de entradas que não são modificadas *sigc(c,* fi ).

Dependendo do modo de operação, tal ação pode levar a:

— modificando a marcação de *fi ,* ou seja, aplicando *ModifCallInputs(c,* fi), — escolhendo outra função, ou seja, aplicando *ChooseCall(c, fi)* (cujo resultado depende do modo atual), — uma transformação manual pelo usuário
dessa ação por a *ChangeCall(c, fi , gk)* onde *gk* deve necessariamente calcular saídas suficientes.

#### *Saídas ausentes* de ação

A ação *MissingOutputs(c, fi)* é gerada quando a marcação de *fi* foi alterada e a função *gj* atribuída a *Call(fi)(c) = gj* não computa saídas suficientes para o novo marcação.

Dependendo do modo de operação, tal ação pode levar a: modificação da marcação de *gj ,* ou seja, a aplicação de
*AddOutputMarks(gj , sigc(c,* fi)),

— a escolha de outra função, ou seja, a aplicação de *ChooseCall(c, fi)* (cujo resultado depende do modo atual), — uma transformação
manual pelo usuário desta ação por uma *ChangeCall (c, fi , gk)* onde *gk* deve necessariamente calcular saídas suficientes.

**Alteração de uma chamada: *ChangeCall*** A

ação *ChangeCall(c, fi , gj )* permite transformar *fi* para ter *Call(fi)(c) = gj* . Só pode ser aplicado se *gj* calcular todas as saídas necessárias para a chamada *c* em *fi* . Portanto, é necessário aplicar *AddOutputMarks* a *gj* antes de aplicar *ChangeCall* se estiver faltando.

Aplicar *ChangeCall(c, fi , gj )* consiste em modificar *Call(fi)(c),* mas também aplicar *ModifCallInputs(c, fi)* caso a assinatura de *gj* exija entradas que não são visíveis em *sigc(c,* fi).

### 3.4.4 Modos de operação

A geração dessas ações, bem como o efeito de sua aplicação no projeto, depende da escolha de um modo de operação que determina a precisão das especializações. Este modo pode oferecer muitas possibilidades, mas é inicialmente limitado a quatro comportamentos: 1. nenhuma especialização de chamadas de
função **(DontSliceCalls)** (cf. §3.4.4) ; 2. sem especialização, mas propagação da
marcação para as funções chamadas **(PropagateMarksOnly)(cf.** §3.4.4) ; 3. o mínimo possível de especializações **(MinimizeNbSlice)** (ver
§3.4.4) : a ferramenta não cria mais de uma especialização por função; 4. as especializações mais precisas possíveis **(PreciseSlices)** (cf. §3.4.4) :
as funções com a mesma visibilidade são, no entanto, agrupadas, mesmo que a marcação não seja a mesma.

Para maior flexibilidade, do ponto de vista do kit de ferramentas descrito aqui, assume-se que o modo pode ser alterado a qualquer momento.

#### Redução com *DontSliceCalls*

Esta é a escolha que permite que a marcação intraprocedimento seja utilizada sem se propagar para as funções chamadas. Pode ser usado se alguém estiver interessado nos links de dependência em uma função, sem estar interessado no restante do aplicativo. Todas as chamadas visíveis são, portanto, atribuídas às funções de origem:

$$\ddot{y}c.\ddot{y}o.outSigc(c, \text{fi})(o) = \ddot{y}m\ \ddot{y}\ Call(fi)(c) = Call(f0)(c)$$

Nesse caso, não adianta gerar uma *ação ChooseCall,* e os *MissingInputs* (marcação de inputs não utilizados como *Spare )* são aplicados diretamente. Além disso, não há ações *MissingOutputs ,* pois as funções de origem calculam todas as saídas.

Se outro modo foi usado, ainda é possível aplicar as seguintes ações:

— *ChooseCall(c, fi)* converte em *ChangeCall(c, fi ,* g0), —
*MissingInputs(c, fi)* em *ModifCallInputs(c, fi)*
— *MissingOutputs(c, fi)* e *AddOutputMarks ?* ou *ChangeCall(c, fi ,* g0) ?

**Redução com** *PropagateMarksOnly* Este modo

deve ser usado se você quiser ver a propagação de dependências em todo o aplicativo sem calcular especializações. É, portanto, muito semelhante ao modo anterior, exceto que uma marcação também é calculada para as funções chamadas. No entanto, não é uma redução propriamente dita, pois tudo permanece visível.

**Redução com** *MinimizeNbSlice*

Quando uma ação *ChooseCall* é aplicada a uma chamada para uma função *g* e ainda não há especialização, uma nova é criada.

Se houver uma especialização *gj* (e apenas uma), ela é escolhida para ser chamada, independentemente de sua marcação.

Se existirem vários *gj* , é *o mais próximo* [2] qual deve ser escolhido.

Se faltarem marcas nas saídas da função selecionada, elas devem ser adicionadas. Para isso, não há geração de *MissingOutputs* , pois queremos ter apenas uma especialização: ela é substituída diretamente por *AddOutputMarks.*

Finalmente, a ação *ChangeCall* permite que você chame a função escolhida. É esta última ação que então se encarrega de propagar a marcação das entradas em *fi* aplicando automaticamente o *ModifCallInputs(c,* fi).

**Redução com** *PreciseSlices*

Neste modo, ao aplicar *ChooseCall(c,* fi), a escolha da função a ser chamada é feita da seguinte forma:

— selecionamos as especializações que têm a mesma visibilidade que *outSigc(c, fi)* (mesmo que não possuem exatamente a mesma marcação);
— se não houver nenhum, um novo é criado; — se houver apenas um, é escolhido; — se forem vários, deve-se escolher um. No caso, escolhemos por enquanto a primeira especialização encontrada. É claro que essa escolha pode ser melhorada se definirmos critérios de seleção.

Deve-se notar que esta situação só pode ocorrer se o usuário estiver inter veio.

Observação: se *Call(fi)(c)* já estiver atribuído antes de ChooseCall ser aplicado , seu valor antigo será ignorado.

Neste modo, aplicar as ações *MissingOutputs(c, fi)* e *MissingInputs(c, fi)* equivale a aplicar *ChooseCall(c,* fi).

Vale ressaltar que este modo também pode ser escolhido caso você queira controlar todas as escolhas manualmente, pois as ações são todas geradas, e a ferramenta só faz uma escolha quando elas são aplicadas. Eles podem, portanto, ser previamente transformados em um *ChangeCall* pelo usuário.

2. Ficam por definir os critérios de seleção.

### 3.4.5 Outras ações

**Duplicando uma especialização: *CopySlice*** A ação

*fj* = *CopySlice(fi)* simplesmente duplica *fi* e, portanto, cria uma nova função especializada *fj* . A marcação, bem como a função Call(fi) são simplesmente copiadas.

Note que inicialmente *fj* não é chamado.

**Combinando especializações: *Combine*** Você

pode combinar duas especializações (ou seja, torná-las uma união) usando f3 = *Combine(f1,* f2).

A tabela de marcação é calculada simplesmente combinando as marcas, sem necessidade de propagar. Em seguida, é necessário resolver as chamadas de função, ou seja, calcular Call(f3). O que é feito para uma determinada chamada *c* depende de sua assinatura no resultado f3 em comparação com o que temos em f1 e f2. Sabemos que, por construção *sigc(c,* f3) = *sigc(c,* f1) ÿ *sigc(c,* f2) e que é portanto maior ou igual.

— si *sigc(c,* f1) = *sigc(c,* f2) :
— si *Chamada(f1)(c) = Chamada(f2)(c) : Chamada(f3)(c) = Chamada(f1)*
*(c)* — si *Chamada(f1)(c) =* $_c$ *: Chamada(f3)(c) = Chamada(f2)*
*(c)* — si *Chamada(f2)(c) =* $_c$ *: Call(f3)(c) = Call(f1)(c)* —
caso contrário, significa que as duas chamadas são atribuídas a diferentes funções:
escolhemos aquele com a menor assinatura (ou seja, o mais preciso).
— else, if *sigc(c,* f3) = *sigc(c,* f1) : *Call(f3)(c) = Call(f1)(c)* — else,
if *sigc(c,* f3) = *sigc(c,* f2) : *Call(f3)(c) = Call(f2)(c)* — caso contrário,
*Call(f3)(c) = gi* , e criamos uma ação *gi = Combine(Call(f1)(c),* Call(f2) (vs)).

Deve-se notar que a criação de f3 não modifica nem f1 nem f2. Se o objetivo for substituir f1 e f2 por f3, isso pode ser feito por uma combinação de ações elementares (cf. §3.6.4).

**Excluindo uma especialização: *DeleteSlice*** Uma

função especializada pode ser excluída usando a *ação DeleteSlice(fi),* mas esta ação só pode ser aplicada se *fi* não for chamado, a menos que seja chamado por si só. Isso significa que suas chamadas devem ser excluídas previamente usando ações *ChangeCall* (ou *DeleteSlice* no chamador...).

## 3.5 Seleção Persistente

Após um primeiro uso das funções apresentadas acima, parecia que a necessidade do usuário também poderia ser marcar uma instrução (ou um ponto de dados em um ponto do programa) para todas as especializações de uma função. Este tipo de seleção tem sido chamado de **persistente.** Isto significa que cada função tem, em paralelo com as suas diferentes marcações, uma marcação mínima que é utilizada aquando da criação de qualquer nova especialização. Quando o usuário adiciona uma seleção persistente, ela é adicionada tanto nesta marcação mínima, quanto em todas as especializações existentes. Adicionar uma seleção persistente a uma função também pode tornar todas as suas chamadas visíveis (se a opção estiver definida) porque o usuário deseja que o programa minimizado salte para o ponto marcado sempre que possível.

o programa fonte: isso, portanto, assume que todos os caminhos que levam à função são visíveis.

## 3.6 Gerenciar ações elementares

As ações elementares apresentadas acima devem ser combinadas para responder a consultas de nível superior. O usuário expressa suas solicitações adicionando ações à lista de projetos, mas estas devem ser decompostas para serem aplicadas, e a aplicação de ações elementares gera outras.

A granularidade em que o usuário pode intervir fica por definir de acordo com o que se deseja fazer com a ferramenta.

Primeiro examinaremos como a lista de ações pode ser gerenciada (§3.6.1), depois veremos um certo número de exemplos de combinação de ações elementares para responder a solicitações de nível superior.

### 3.6.1 Modificação ou exclusão de ações

As ações do usuário podem ser deletadas por ele antes de serem aplicadas.
Por outro lado, as ações geradas como *ChooseCall, MissingInputs* ou *MissingOutputs* consistem em tornar uma chamada de função consistente, portanto, não podem ser excluídas. Por outro lado, o usuário pode transformá-los em *ChangeCall* para especificar a função que deseja ver chamada. Nesse caso, essa nova ação não pode, por sua vez, ser excluída.

### 3.6.2 Ordem das ações

A priori, estando a marcação perfeitamente definida, não depende da ordem dos cálculos.
No entanto, o resultado obtido pode depender da ordem de aplicação das ações, pois se aplicarmos uma ação que construa uma nova função *gj ,* então outra para construir *fi* que chama uma função g, *gj* poderá ser escolhida para a chamada , enquanto que se a primeira ação ainda não tivesse sido aplicada, outra especialização poderia ter sido chamada ou uma nova ação teria sido gerada. Além disso, algumas ações possuem uma pré-condição que só é possivelmente satisfeita se as ações geradas forem aplicadas em ordem. Portanto, inicialmente não está planejado poder modificar a ordem das ações.

### 3.6.3 Propagação para chamadores

As ações elementares apenas propagam a marcação das entradas de uma determinada função *fi* para chamadas *(c, hk)* como *Call(hk)(c) = fi .* Se quisermos construir novas funções especializadas para todas as funções *h* que chamam *f* para propagar a marcação, devemos encadear diferentes ações elementares.

Vejamos, por exemplo, como controlar a adição de uma marca de usuário *m* em um elemento *e* de *f* com propagação nos chamadores:

— *fi = NewSlice(f0)*

— *AddUserMark(fi ,(e,* m))
— *ÿh0.ÿcf* ÿ call(h0) ÿ *hk = NewSlice(h0)* ÿ *ÿcf .ChangeCall(cf , hk, fi)*

#### 3.6.4 Fusão de duas especializações

Se duas especializações f1 e f2 foram calculadas, o usuário pode querer mesclá-las. Para isso, as seguintes ações devem ser aplicadas: —
f3 = *Combine(f1,* f2) —
*ÿc, hk. Call(hk)(c)* = f1 ÿ *ChangeCall(c, hk,* f3) — *ÿc,*
*hk. Call(hk)(c)* = f2 ÿ *ChangeCall(c, hk,* f3)
— *DeleteSlice(f1), DeleteSlice(f2)*

## 3.7 Produção do resultado

Lembre-se que o objetivo é gerar arquivos fonte compilados. No entanto, após ter realizado todo o trabalho de filtragem apresentado anteriormente, obtém-se um projeto contendo, para cada função, zero, uma ou várias funções filtradas, havendo para cada função filtrada uma tabela de marcação que permite determinar as instruções visíveis, e chamadas de função. Vamos ver o que falta para chegar ao resultado final.

#### 3.7.1 Declarações globais

Para produzir arquivos de resultados, você deve acima de tudo ser capaz de colocar neles o que está fora das funções, ou seja, as declarações globais.

Teria sido possível calcular links de dependência para declarações globais, mas a remoção de declarações desnecessárias foi considerada um bom recurso em si, então isso é feito como uma passagem extra no resultado.

#### 3.7.2 Funções

Para uma função que foi analisada, foi calculada a representação da função de origem, bem como uma ou mais especializações. As funções não analisadas não aparecerão no resultado porque não participam do cálculo solicitado.

**função especializada**

Para gerar o código correspondente a uma função especializada, deve-se atribuir a ela um nome que não conflite com o das funções fonte ou outras especializações. Também é necessário ter sua assinatura para saber se tem valor de retorno, e conhecer a lista de seus argumentos visíveis.

**Se deve ou não manter a função de origem**

Uma função de origem só precisa estar presente no aplicativo gerado se for chamada. O processamento de uma chamada de função que não leva à chamada de uma especialização deve, portanto, memorizar um link com a função de origem.

### Variáveis locais estáticas

Quando uma função está presente em várias cópias no resultado, é preciso pensar na saída das possíveis variáveis estáticas comuns a várias especializações para que sejam compartilhadas pelas diferentes funções.

### Editando Chamadas

Para gerar uma chamada a uma função especializada, é necessário dispor das mesmas informações que foram utilizadas para a definição da função, nomeadamente:

— seu nome,
— a lista de parâmetros formais necessários para calcular as saídas selecionadas,
— o tipo de retorno para saber se a função produz um resultado.

**Geração de novos protótipos** Ao produzir

funções especializadas, também é necessário gerar os protótipos associados.
Como eles têm necessariamente o mesmo escopo da função fonte, basta gerar esses novos protótipos no local onde se encontra o protótipo inicial.

#### Gerenciamento de blocos

Mesmo que o gráfico de dependência contenha elementos que representam os blocos e, portanto, o *fatiamento* associe marcas a esses elementos, é difícil excluir um bloco, mesmo que ele seja invisível. De fato, isso às vezes pode alterar a semântica e, em um if (c) { S1; } senão { S2; }, você pode até obter um resultado sintaticamente incorreto se não for cuidadoso.

Como também pode ser útil ter uma ferramenta de limpeza de código que remova adequadamente os blocos vazios, decidimos não processá-los no *nível de fatiamento.* Todos os blocos são, portanto, considerados visíveis, independentemente de sua marca, e a limpeza é realizada durante uma passagem adicional no código gerado.

### 3.7.3 Anotações

As anotações podem ser usadas para construir consultas de fatiamento , mas além dessa possibilidade, deve-se determinar se as anotações presentes no código-fonte devem ser colocadas no resultado. Para isso, verificamos se os dados utilizados para avaliar a propriedade são preservados no programa reduzido: se sim, a anotação será mantida, caso contrário, será fatiada.

# Capítulo 4

# Usar

Neste capítulo tenta apresentar algumas bases que permitem testar o toolkit de *slicing* a partir do interpretador ou de um arquivo de comando. Para usar a interface gráfica, consulte a documentação correspondente. Também é possível realizar algumas ações básicas na linha de comando. Veja as opções oferecidas usando:

```
bin/toplevel.top --help
```

## 4.1 Utilização interativa

Ao iniciar a ferramenta de fatiamento, você deve fornecer o(s) arquivo(s) para análise e, pelo menos, a opção -deps que permite realizar a análise de dependência (use –help para ver as outras opções):

```
bin/toplevel.top -deps fichier.c
```

Em seguida, terminamos com um prompt # que indica que estamos sob um interpretador Ocaml.

Para sair do interpretador, use o comando:

```
#desistir;;
```

(o # não é o prompt, faz parte do comando). Você também pode usar Ctrl-D.

**Truque**

Como qualquer interpretador Ocaml, ele não tem capacidade de editar a linha de comando (para corrigir um erro de digitação, por exemplo), nem o histórico (para recuperar um comando anterior). Portanto, é altamente recomendável, mesmo que não seja essencial, instalar o mencionado acima, que pode ser obtido no endereço: ftp://ftp.inria.fr/INRIA/Projects/cristal/Daniel.de_Rauglaudre /Tools/. Em seguida , basta preceder o nome do comando com o dito . O homem disse ter mais informações sobre as possibilidades desta ferramenta.

No interpretador, você pode usar qualquer instrução Ocaml e todos os comandos específicos da ferramenta.

Para aqueles que não estão familiarizados com Ocaml, para iniciar um comando, você deve digitar seu nome seguido dos argumentos ou parênteses vazios se não houver argumentos e terminar com dois pontos e vírgulas.

O interpretador exibe o tipo de retorno do comando (a unidade corresponde ao tipo vazio), possivelmente seguido do valor retornado. Na maioria dos casos, a primeira informação não interessa ao usuário, mas esta exibição não pode ser desabilitada. Às vezes pode ser útil encontrar a assinatura de comandos < por exemplo: módulo S = Db.Slicing;;

exibe o tipo do novo módulo S e, portanto, permite que você veja a lista de comandos do módulo Db.Slicing.

A organização do Frama-C agrupa todos os comandos do módulo Db. O submódulo Db.Slicing contém comandos de divisão . O nome de um comando normalmente deve ser prefixado pelo nome do módulo ao qual ele pertence. Para poder usar nomes sem prefixo, você deve abrir o módulo usando: open Db;;

Você também pode optar por renomear o módulo para ter um prefixo mais curto: module
S = Db.Slicing;;

Comandos de outros módulos também são úteis ao usar *fatiamento.* O Kui, em particular, fornece funções para navegar pelas informações geradas por outros analisadores.

## 4.2 Arquivo de comando

Os comandos podem ser digitados diretamente, mas também é possível carregar arquivos contendo os comandos com o comando #use "cmds.ml";;. O conteúdo do arquivo cmds.ml é então interpretado como se tivesse sido digitado, e o controle é retomado no interpretador ao final de sua execução.

Se você deseja apenas executar um arquivo em lote, pode usar o redirecionamento do Unix.
O interpretador será encerrado ao encontrar o EOF (fim do arquivo).

**Truque**

Ao utilizar dito (veja o truque na página anterior) também obtemos a possibilidade de memorizar o histórico e, portanto, ter um registro dos comandos lançados (muito prático como base para escrever um script ou para reproduzir uma sequência). Para fazer isso, basta digitar: said -h cmds.ml bin/toplevel.top ... arguments...

## 4.3 Exemplos

Alguns exemplos de uso podem ser encontrados no diretório test. Os dois arquivos a seguir fornecem algumas funções que facilitam o uso:

— libSelect.ml oferece uma interface simplificada para seleções simples, — libAnim.ml permite gerar representações gráficas do projeto em diferentes estágios de sua construção para visualizar as relações entre as especializações.

# capítulo 5

# Conclusão

Em conclusão, podemos dizer que as funcionalidades básicas da ferramenta não evoluíram muito em 2008, mas ganharam robustez e precisão: esse foi o principal objetivo do ano.

O principal desenvolvimento diz respeito ao gerenciamento de anotações, tanto na produção do resultado (o que mantemos?) quanto como critério de fatiamento . Este ponto ainda está em desenvolvimento porque requer o uso de funções externas ao módulo que ainda não existem.

O kit de ferramentas *de fatiamento* pode ser considerado estável, embora ainda possa evoluir para atender a novas necessidades,

O Apêndice D apresenta em particular alguns projetos que foram mencionados, alguns dos quais poderiam complementar a ferramenta.

# Anexo A

# Algoritmos

## A.1 Premissas

Damos a nós mesmos alguns tipos e funções básicas:

```
(* Point de program *) type
t_program_point ;;

(* Instrução *) digite
t_stmt ;;

(* a instrução localizada em um ponto de verificação (após) *) val get_pp_stmt :
t_program_point -> t_stmt ;;

(* CFG : gráfico de fluxo de controle *)
digite t_cfg ;; (*
sucessos no cfg. *) val get_cfg_succ : t_cfg
-> t_stmt -> t_stmt list ;; (* predecessores dans le cfg. *) val get_cfg_prev :
t_cfg -> t_stmt -> t_stmt list ;;


(* PDG : gráfico de dependências do programa *)
digite t_pdg ;; (*
elemento que compõe o PDG *) type
t_elem ;; (* coloque
a lista de dependências diretas do elemento no PDG *) val get_dpds : t_elem -> t_pdg -> t_elem list ;;
val get_all_dpds : t_pdg -> t_elem -> lista t_elem ;; val get_list_all_dpds :
t_pdg -> lista t_elem -> lista t_elem ;; val get_list_control_dpds : t_pdg ->
lista t_elem -> lista t_elem ;; val get_list_all_control_dpds : t_pdg -> lista t_elem -> lista
t_elem ;; val merge : lista t_elem -> lista t_elem -> lista t_elem ;;



val get_pp_elems : t_pdg -> t_program_point -> lista t_elem ;;

(* correspondência entre instruções e elementos PDG *) type t_stmt_elems ;;


(* encontre a instrução correspondente a um elemento *) val get_stmt: t_elem
-> t_stmt_elems -> t_stmt ;;
```

```
(* encontre as instruções correspondentes aos elementos *) val get_stmts: t_elem
list -> t_stmt_elems -> t_stmt list ;; (* encontre os elementos correspondentes a uma
 instrução *) val get_elems: t_stmt -> t_stmt_elems -> t_elem list ;;


digite t_state

digite t_data

val get_state : t_program_point -> t_state ;; val get_defs_data :
t_state -> t_data -> t_elem list ;; (* type des marques *) type t_mark ;; (* la
 marque correspondente àmS :
superflu. *) val
 spare_mark : t_mark ;; (* combinaison de deux marques *) val
combine_mark : t_mark -> t_mark ->
 t_mark ;;


(* tipo correspondente à marcação das instruções de uma função. *) digite t_ff ;; (* leia a marca
associada a
 uma instrução na marca *) val get_stmt_mark : t_stmt -> t_ff -> t_mark ;; (* substitua a
marca associada a uma instrução na marca *) val
 replace_stmt_mark : t_ff -> t_stmt -> t_mark -> t_ff ;;
```

## A.2 Marcação de uma função

Trata-se de ver como são marcadas as instruções de uma função a partir de um pedido dando um elemento do CEO e uma marca.

O algoritmo aqui apresentado é uma versão simplificada do que existe na ferramenta para ser mais fácil de entender. É muito ineficiente e corre o risco de entrar em loop, pois não testa se a instrução a ser marcada já contém a nova marca (teste de parada).

Esta é mais uma versão não especializada de chamadas de função.

```
(* Chamamos H de módulo de hipótese. *) módulo H =
AlgoH ;;

(* produz uma nova função especializada a partir de [ff]
      marcando o elemento [e] e todas as suas dependências com a marca [m]. *)
 deixe rec mark_rec_pdg_elem pdg stmt_elems me ff =
    let new_ff = add_elem_mark pdg stmt_elems me ff in let dpds =
    H.get_dpds e pdg in
       List.fold_right (mark_rec_pdg_elem pdg stmt_elems m) dpds new_ff (* ;; *) and (*


[add_elem_mark] ajoute la marque [m] àl'instruction correspondente à
      o elemento [e] e marque quaisquer outros elementos como supérfluos. *)
           add_elem_mark pdg stmt_elems me ff =
    let stmt = H.get_stmt e stmt_elems in let old_m =
    H.get_stmt_mark stmt ff in let new_m =
    H.combine_mark old_m m in
```

```
let new_ff = H.replace_stmt_mark ff stmt new_m in let elems =
H.get_elems stmt stmt_elems in let (_, other_elems)
= List.partition (fun elem -> elem = e) elems in let mark_spare_elem e ff =
mark_rec_pdg_elem pdg stmt_elems H. spare_mark e ff em List.fold_right mark_spare_elem other_elems
  new_ff
```

# Anexo B

# Exemplo de marcação interprocedimento

Esta parte apresenta como as ações elementares apresentadas anteriormente podem ser usadas para responder a solicitações de usuários de nível superior.

## B.1 Apresentação do exemplo

Veremos diferentes exemplos de marcação no exemplo abaixo. Em todos os casos, considera-se que o usuário deseja ter apenas uma especialização por função fonte no resultado, e que solicita sistematicamente a propagação de sua marcação aos chamadores.

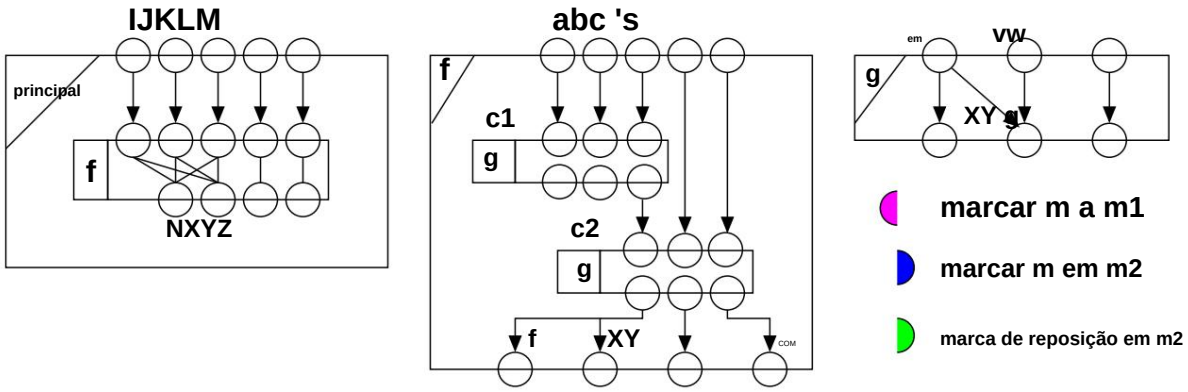
Para simplificar a apresentação, como estamos interessados aqui apenas na propagação interprocedural, usamos apenas qualquer marca elementar *m* que torne um elemento visível e a marca *Spare* já mencionada.

```
int X, Y; int
g (int u, int v, int w) { lg1:
        X = u; lg2:
    Y = u + v;
    retornar w; }
```

```
intZ; int
f (int a, int b, int c, int d,
        int e) {

    int r;
    lf1: r = g (a, b, c); lf2: Z = g
    (r, d, e); retornar X; }
```

```
int I, J, K, L, M, N; int main
() { lm1: /* ... */
    lm2: N = f (I, J, K,
    L, M); lm3: /* ... */ }
```




A legenda indica como as marcas são representadas nas figuras a seguir.

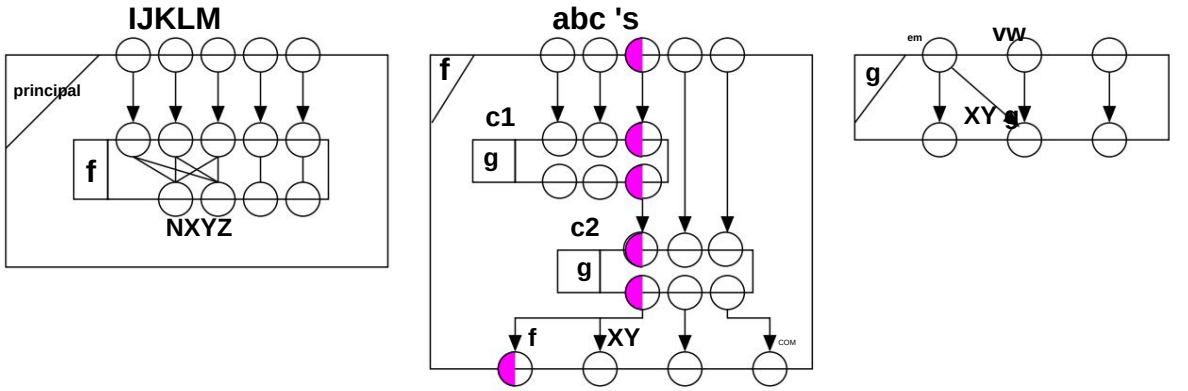
## B.2 Caso 1

Suponha primeiro que o usuário deseja selecionar a saída 0 de f e veja como o cálculo procede.

Vimos em §3.6.3 que essa solicitação do usuário resulta na seguinte sequência de ações elementares:

— f1 = *NewSlice(f0),*
— *AddOutputMarks(f1,(out0,* m)) —
*main1* = NewSlice(main0), —
*ChangeCall(c, main1,* f1).

Primeiro calculamos a marcação de f1 por propagação simples:

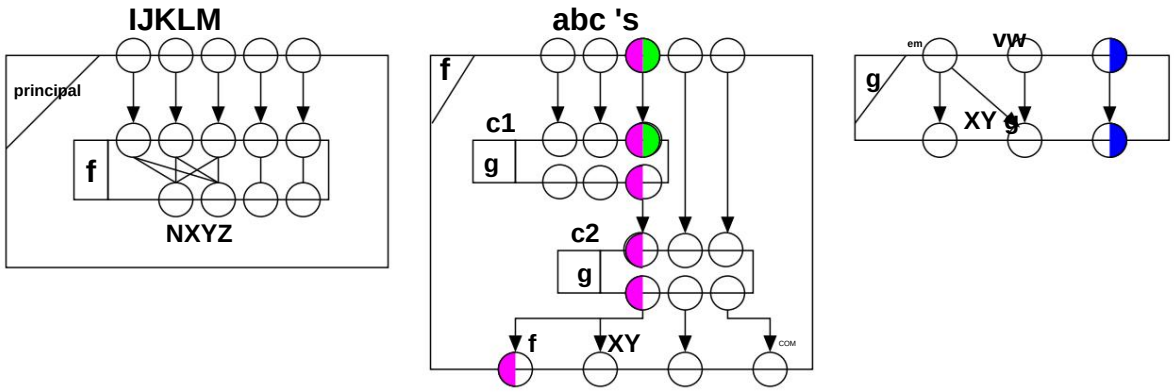



Em seguida, a segunda ação gera *ChooseCall(c1,* f1) e *ChooseCall(c2,* f1).

Ao aplicar a primeira dessas novas ações, como ainda não há especialização para g, geramos:
— g1 = *NewSlice(g0), —*

*AddOutputMarks(g1,*
*(out0,* m))
— *ChangeCall(c1,* f1, g1)

Após a construção de g1, a entrada *w* tem uma marca m2. A aplicação do *ChangeCall ,* portanto, acionará um *ModifCallInputs(c1,* f1) que levará à marcação da entrada *c* de f1 como *Spare* (além de *m* em m1).



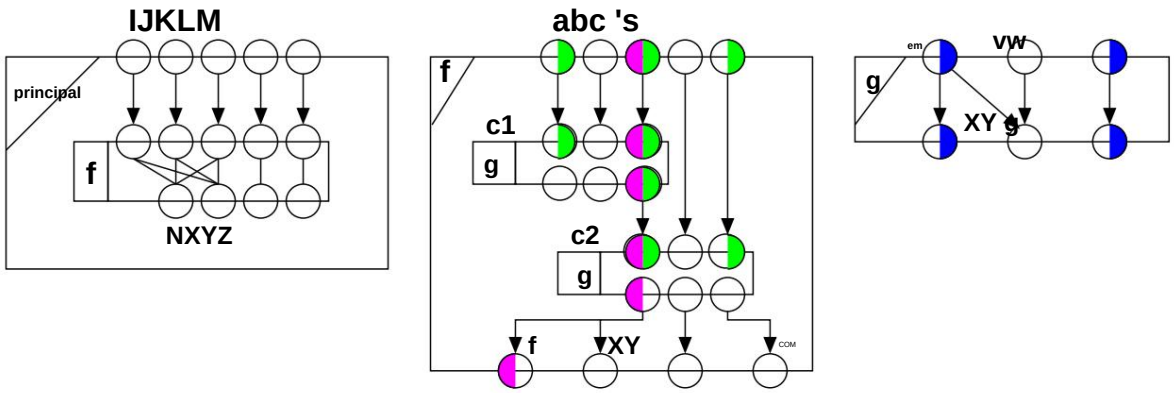

A aplicação *de ChooseCall(c2,* f1) produz:

— *AddOutputMarks(g1,(outX,* m)) já que optamos por ter apenas uma especialização por função source,

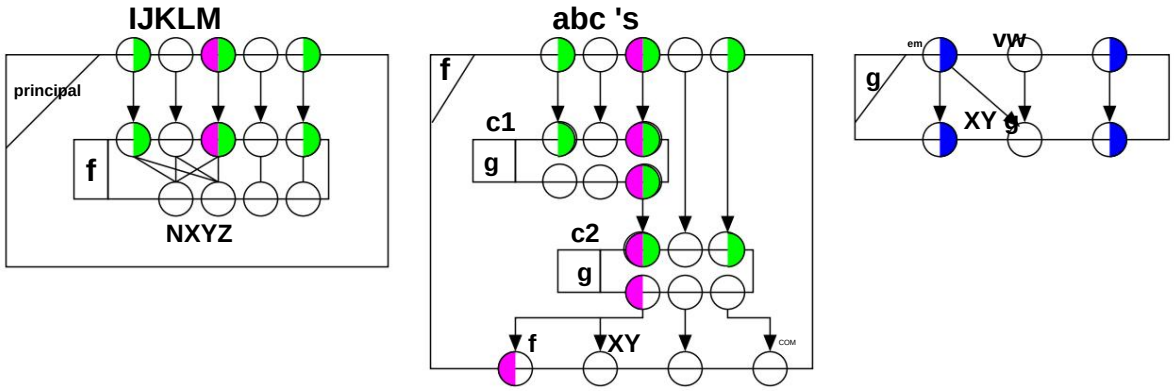
— e *ChangeCall(c2,* f1, g1).

Como g1 é chamado em c1, modificar sua marcação gera *MissingInputs(c1,* f1), que, dadas as opções, é traduzido diretamente por *ModifCallInputs(c1,* f1).

Em seguida, o *ChangeCall* acionará *ModifCallInputs(c2,* f1) que marcará *e* como *Spare.*



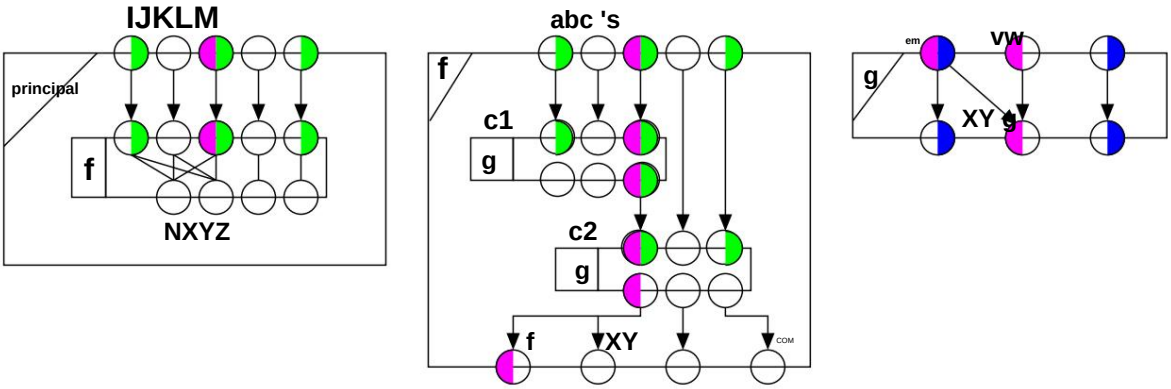

Por fim, resta apenas aplicar *ChangeCall(c, main1,* f1) que leva à aplicação de *ModifCallInputs(c, main1)* e, portanto, à propagação da marcação de f1 em *main1.*



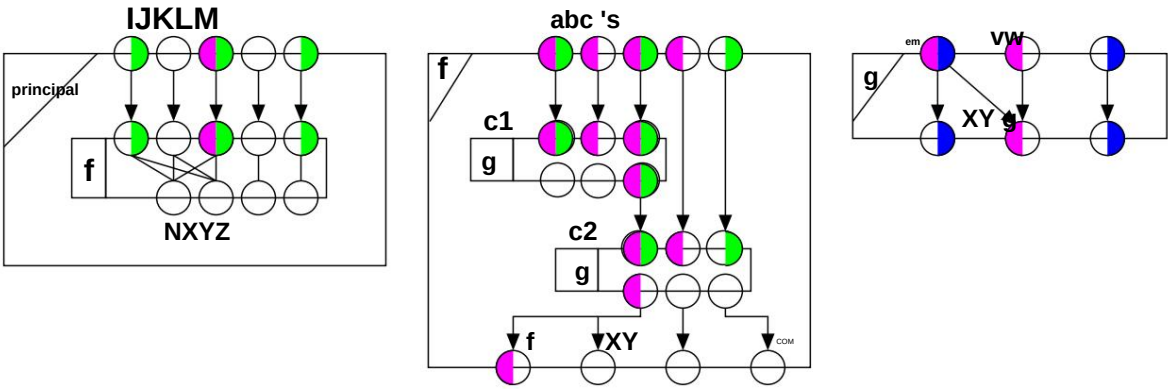
## B.3 Caso 2

Do resultado do caso 1, o usuário deseja selecionar o cálculo de *Y* em g1.

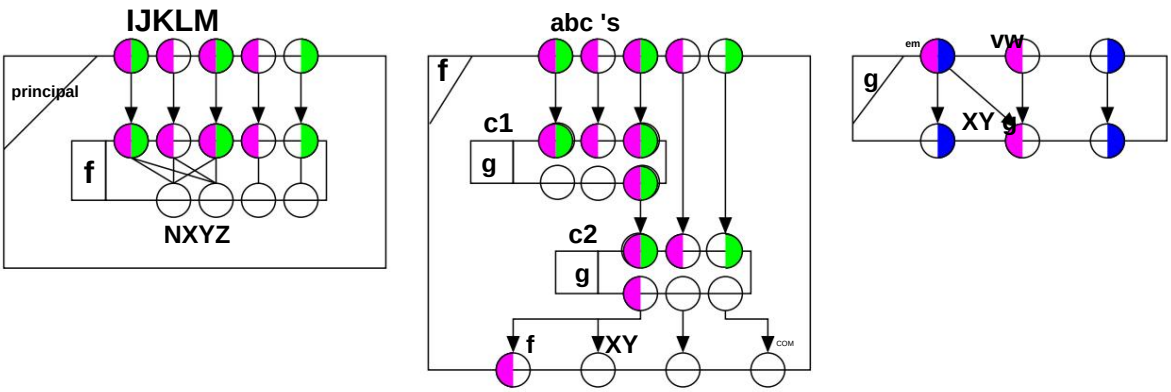
*AddUserMark(g1,(outY ,* m)) propaga a marcação em m1 para a entrada *v* de g1 *(u* já está marcado).




Então, quando g1 é chamado e há marcas ausentes, duas ações *MissingInputs* são geradas. As opções indicam que devem ser traduzidas para *ModifCallInputs(c1,* f1) e *ModifCallInputs(c2,* f1). O que leva a marcar em m1 as entradas *a, b, d* de f1.




Da mesma forma, a propagação será realizada em *main1* aplicando *ModifCallInputs(c, main1).*



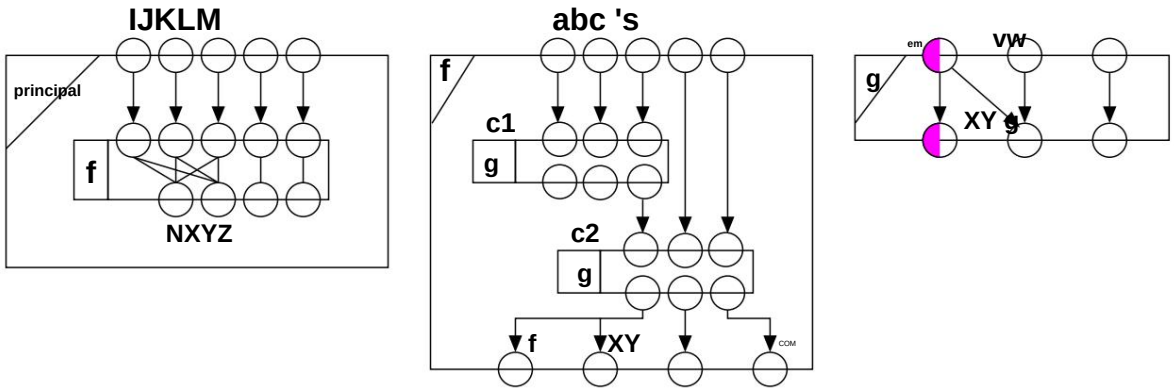
## B.4 Caso 3

Em um novo estudo, o usuário deseja selecionar o cálculo de *X* em g.

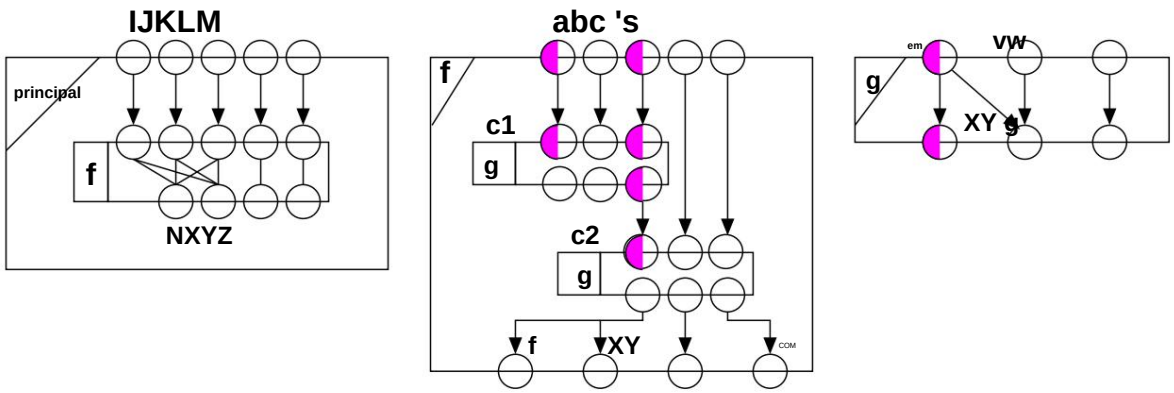
Como no caso 1, esta requisição resulta na criação de uma função especializada g1 e na propagação de sua marcação para todas as suas

chamadas: — g1 =
*NewSlice(g0),* — *AddOutputMarks(g1,*
*(outX,* m)) — f1 =
*NewSlice (f0),* — *ChangeCall(c1,* f1, g1).
— *ChangeCall(c2,* f1, g1).
— *main1* = NewSlice(main0),
— *ChangeCall(c, main1,* f1).

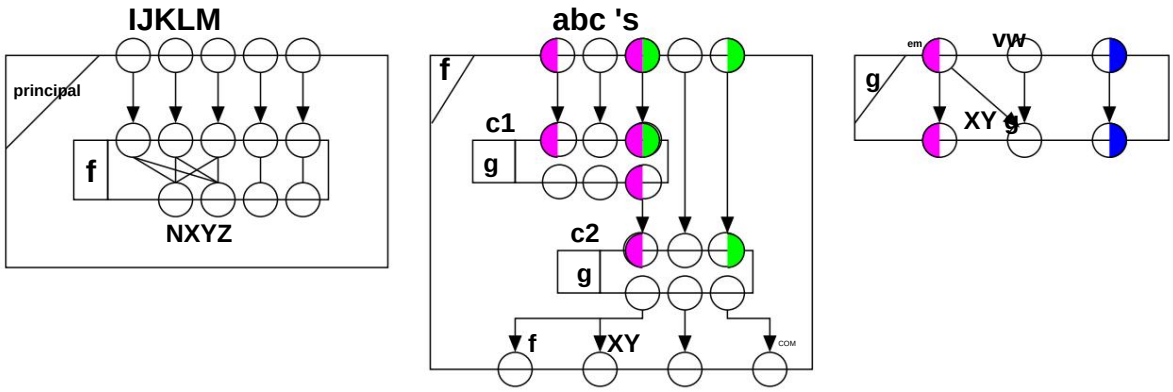
Primeiro calculamos a marcação de g1 :



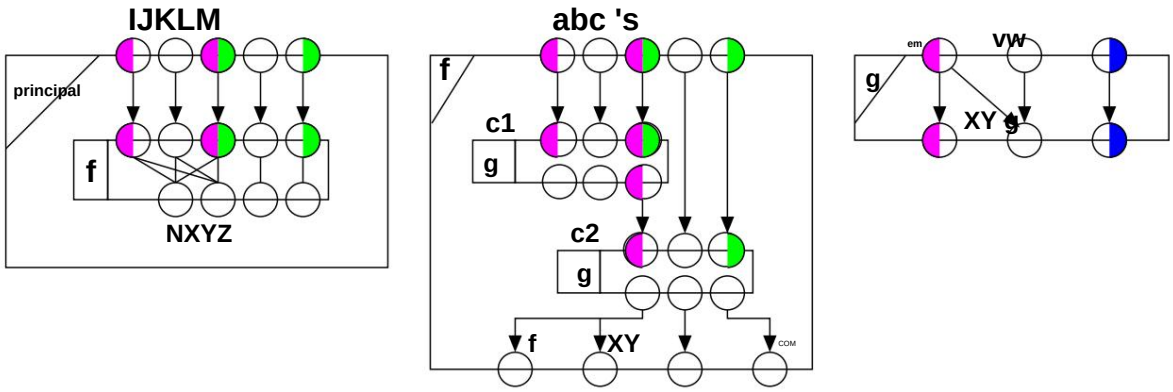

Então, as duas *ChangeCalls* em f1 levam a propagar m1 :




O segundo commit (ao modificar c2) aciona *MissingOutputs(c1,* f1) que, dadas as opções, se transforma em *AddOutputMarks(g1, outSigc(c1)).* Isso leva à marcação de m2 como saída 0 de g1. Os *MissingInputs* gerados se transformam em *ModifCallInputs(c1,* f1) e *ModifCallInputs(c2,* f1), que propaga *Spare* para as entradas *c* e *e* de f1.



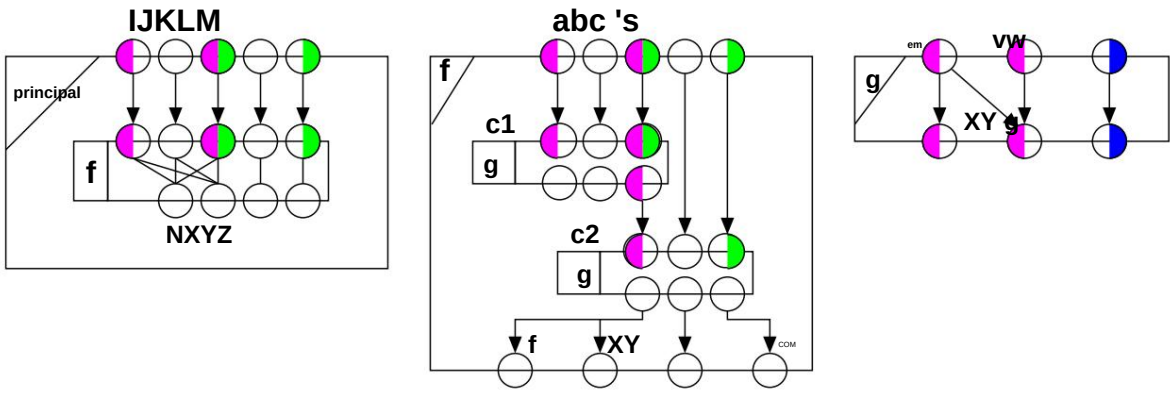

Finalmente, o *ChangeCall(c, main1,* f1) propaga a marcação de f1 para *main1.* Notamos que mesmo que essa alteração tivesse sido feita anteriormente, a propagação da marcação teria sido feita graças ao *MissingInputs.*



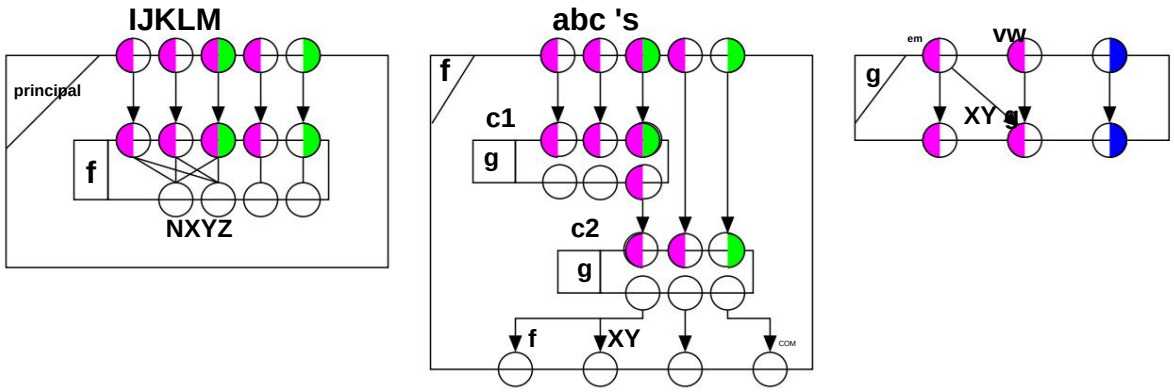

## B.5 Cas 4

A partir do caso 3, o usuário deseja adicionar a seleção do cálculo de *Y* em g1.

*AddUserMark(g1,(outY ,* m)) resulta na marcação *Y* em m1, então, por propagação, *v* em m1 também:




As ações *MissingInputs* transformadas em *ModifCallInputs* propagam m1 para as entradas *b* e *d* de f1, depois para as entradas *J* e *L* de *main1.*



# Anexo C

# Detalhes das ações

Este capítulo apresenta uma folha de dados para cada ação do cálculo interprocedimento apresentado no capítulo 3. Correspondem aos comandos conforme foram especificados, podendo haver algumas diferenças do que foi implementado, mas são diferenças menores (principalmente nomes) . Para mais detalhes, convida-se o leitor a consultar o documento do código que estará sempre o mais atualizado.

As seguintes indicações são dadas para cada ação:

— **Parâmetros :** indica o significado dos parâmetros de ação,
— **Pré-condição :** indica quaisquer condições para criação e/ou aplicação,
— **Criação de usuário :** indica se o usuário pode criá-lo,
— **Editável pelo usuário :** indica se o usuário pode modificá-lo quando estiver na lista de espera,
— **Geração automática :** indica se esta ação pode ser gerada pela ferramenta, e especifica em quais
casos, — **Aplicação :** detalha o que acontece quando esta ação é aplicada, —
**Geração :** fornece a lista de ações que podem ser geradas, —
**Modificações :** indica se uma especialização pode ser criada ou modificada.

## C.1 NewSlice

*NewSlice(f0)*

— **Parâmetros :** f0 é a função de origem para a qual queremos criar um novo especialização.
— **Pré-condição :** nenhuma.
— **Usuário criado :** sim.
— **Editável pelo usuário :** sim (exclusão).
— **Geração automática :** sim, por *ChooseCall.*
— **Aplicação :** cria uma nova especialização *fi ,* inicialmente sem marcações (tudo invisível).
— **Geração :** nenhuma.
— **Modificações :** criação de um novo *fi .*

## C.2 AddUserMark

*AddUserMark(fi ,(e,* m))

— **Parâmetros :** *fi* é a função cuja marcação queremos modificar, *e* um elemento a marcar, *m* a marca a adicionar. Diferentes versões desta ação podem ser propostas para facilitar a designação de *e* e a escolha da marca m.
— **Pré-condição :** *fi* deve existir e *e* denota um elemento válido de f.
— **Usuário criado :** sim.
— **Editável pelo usuário :** sim (exclusão).
— **Geração automática :** não.
— **Aplicação :** anexa *m* para marcar m1 de *e* em *fi* e propaga para dependências.
— **Geração :** *ExamineCalls.*
— **Modificações :** modificação de *fi .*

## C.3 Examinar Chamadas

*ExamineCalls(fi)*

— **Parâmetros :** *fi* é a função cuja marcação foi modificada e para a qual as chamadas devem ser verificadas.
— **Pré-condição :** *fi* deve existir.
— **Usuário criado :** não.
— **Editável pelo usuário :** não.
— **Geração automática :** sim.
— **Aplicação :** verifica a consistência da marcação de cada chamada de função *c* de *call(f)* e também qualquer chamada para *fi .*
— **Geração :** *ChooseCall* e/ou *MissingOutputs* se a marcação de certas chamadas de função foi modificada, e *MissingInputs* se *fi* for chamado e as entradas de certas chamadas forem marcadas insuficientemente.
— **Alterações :** nada.

## C.4 Escolher Chamada

*EscolhaCall(c, fi)*

— **Parâmetros :** chame *c* na função especializada *fi .*
— **Pré-condição :** nenhuma.
— **Usuário criado :** não.
— **Modificável pelo usuário :** sim, pode ser substituído por um *ChangeCall(c, fi , gj )* (consulte as condições para criar esta ação).
— **Geração automática :** sim, quando a marcação de *fi* é modificada e a chamada *c* torna-se visível, esta ação é gerada para atribuir uma função a chamar.
— **Aplicação :** determina a função a chamar tendo em conta o modo de função ção.
— **Geração :** *ChangeCall* e possivelmente *NewSlice* e *AddOutputMarks.*
— **Alterações :** nada.

## C.5 ChangeCall

*ChangeCall(c, fi , gj )*

— **Parâmetros :** consideramos a chamada *c* de *fi , gj* é a função a chamar.
— **Pré-condição :** *Call(f0)(c)* = g0 e a assinatura de saída de *gj* deve ser compatível com *sigc(c,* fi).
— **Usuário criado :** sim.
— **Editável pelo usuário :** sim, mas somente se ele o criou inicialmente.
— **Geração automática :** sim, por *ChooseCall.*
— **Aplicação :** *Call(fi)(c) = gj* e as marcas das entradas de *gj* são propagadas em *fi (ModifCallInputs(c,* fi)).
— **Geração :** a ação *ModifCallInputs* é aplicada diretamente, mas pode gerar para iniciar outras ações (ver §C.7).
— **Modificações :** *fi .*

## C.6 Entradas ausentes

*MissingInputs(c, fi)*

— **Parâmetros :** chamada *c* na função especializada *fi ,* —
**Pré-condição :** a função chamada requer mais entradas do que computa *fi .*
— **Criação pelo usuário :** não, —
**Modificável pelo usuário :** sim, pode ser substituído por um *ChangeCall(c, fi , gj )* (consulte as condições para criar esta ação).
— **Geração automática :** sim, quando a função *Call(fi)(c) = gj* for modificada, e requer a marcação de entradas que não sejam *sigc(c,* fi).
— **Aplicação :** equivalente a *ModifCallInputs* ou *ChooseCall* dependendo do modo de funcionando.
— **Geração :** depende da aplicação (ver acima).
— **Modificações :** depende da aplicação (ver acima).

## C.7 ModifCallInputs

*ModifCallInputs(c, fi)*

— **Parâmetros :** chame *c* na função especializada *fi .*
— **Pré-condição :** *Call(fi)(c) = gj* —
**Usuário criado :** não.
— **Editável pelo usuário :** não.
— **Geração automática :** sim, aplicando *MissingInputs* em alguns modos.
— **Aplicação :** propaga as marcas das entradas de *gj* ao nível da chamada *c* de *fi* ee nas dependências.
— **Geração :** *ChooseCall* e/ou *MissingOutputs* se a marcação de certas chamadas de função foi modificada, e *MissingInputs* se *fi* for chamado e as entradas de certas chamadas forem marcadas insuficientemente.
— **Modificações :** *fi .*

## C.8 Saídas ausentes

*MissingOutputs(c, fi)*

— **Parâmetros :** chame *c* na função especializada *fi* —
**Pré-condição :** *Call(fi)(c) = gj* —
**Usuário criado :** não.
— **Modificável pelo usuário :** sim, pode ser substituído por um *ChangeCall(c, fi , gk)* (consulte as condições para criar esta ação).
— **Geração automática :** sim, quando a marcação de *c* em *fi* é modificada, a chamada é atribuída a *gj* e as saídas de *gj* são insuficientes.
— **Aplicação :** depende do modo: —
*AddOutputMarks* —
ou *ChooseCall*
— **Geração :** depende da aplicação (ver acima).
— **Modificações :** depende da aplicação (ver acima).

## C.9 AddOutputMarks

*AddOutputMarks(fi , outSigf)*

— **Parâmetros :** *fi* é a função cuja marcação queremos modificar, *outSigf* indica as marcas que devem ser adicionadas às saídas.
— **Pré-condição :** *fi* deve existir e *outSigf* deve corresponder a uma assinatura das saídas de f.
— **Usuário criado :** não.
— **Editável pelo usuário :** não.
— **Geração automática :** sim, aplicando *MissingOutputs.*
— **Aplicação :** as marcas de *outSigf* são adicionadas às marcas m2 das saídas de *fi* e propagado.
— **Geração :** *ChooseCall* e/ou *MissingOutputs* se a marcação de certas chamadas de função foi modificada, e *MissingInputs* se *fi* for chamado e as entradas de certas chamadas forem marcadas insuficientemente.
— **Modificações :** modificação de *fi .*

## C.10 CopySlice

*CopySlice(fi)*

— **Parâmetros :** *fi* é a função especializada a ser copiada.
— **Pré-condição :** *fi* deve existir.
— **Usuário criado :** sim.
— **Modificável pelo usuário :** sim (exclusão), —
**Geração automática :** não.
— **Aplicação :** cria uma nova especialização *fj* de *f* cuja marcação e chamadas são idênticas às de *fi .* Atenção, ao final desta ação, *fj* não é chamado.
— **Geração :** não.
— **Modificações :** criação de um novo *fj .*

## C.11 Combinar

*Combine(fi , fj )*

— **Parâmetros :** *fi* e *fj* são as funções especializadas a serem combinadas.
— **Pré-condição :** *fi* e *fj* devem existir.
— **Usuário criado :** sim.
— **Modificável pelo usuário :** sim (exclusão), —
**Geração automática :** não.
— **Aplicação :** calcula uma nova especialização *fk* —
**Geração :** *ChooseCall* —
**Modificações :** criação e cálculo de *fk.*

## C.12 Excluir Fatia

*DeleteSlice(fi)*

— **Parâmetros :** *fi* é a função especializada a ser removida.
— **Pré-condição :** *fi* não deve ser chamado.
— **Usuário criado :** sim.
— **Editável pelo usuário :** sim (exclusão).
— **Geração automática :** não.
— **Aplicação :** *fi* é excluído.
— **Geração :** nenhuma.
— **Alterações :** *fi* foi removido.

# Anexo D

# Projetos

Este capítulo reúne várias ideias que foram discutidas em um momento ou outro e que foram abandonadas ou adiadas.

## D.1 Outros critérios

#### D.1.1 Especificando o que você não quer mais

Outro critério interessante de *fatiamento,* que não aparece em nenhum artigo, seria, creio eu, poder especificar o que se quer retirar. Por exemplo, *"o que acontece com o programa se não estivermos interessados em tal saída ou tal processamento".*

Além disso, a análise de dependência usada para *divisão* também pode ser usada para computar informações úteis para outras análises. Alguém pode, por exemplo, estar interessado nas variáveis cujo valor determina a saída de um loop; que pode ser usado como uma pista para as ampliações a serem feitas em uma interpretação abstrata.

#### D.1.2 Cálculo de uma variável global

A primeira filtragem global visa selecionar as instruções que participam do cálculo de uma determinada variável global. Como também temos acesso à lista de saídas de cada função, basta gerar os filtros permitindo que esta variável seja calculada sobre cada função tendo-a como saída.

## D.2 Uso de semântica

Esta parte reúne algumas ideias de evoluções para melhorar o *fatiamento* de uma função usando a semântica das instruções. Alguns deles foram implementados na ferramenta anterior; outras são apenas caminhos a serem explorados.

#### D.2.1 Redução relacionada a uma restrição

Na ferramenta anterior, um comando permitia cortar um galho substituindo o teste da condição do caminho por um assert.

De forma mais geral, estamos interessados aqui em uma restrição que representa uma asserção no código. Gostaríamos de mover essa restrição para determinar um ponto onde ela é impossível (reduzida a *falso).* Poderíamos então *cortar* o galho correspondente.

O teste condicional que permite acesso a este ramo pode ser transformado em uma asserção ou ser deletado (opção?). No primeiro caso, os dados usados no teste são visíveis, enquanto no segundo, eles podem ser apagados se não forem usados em outro lugar.

Você também pode remover instruções que não são mais úteis. A detecção de código morto permite, por exemplo, excluir elementos que foram usados apenas no ramo excluído.

**Exemplo 6**

Na seguinte sequência:

x = c+1; a = 2; se (c) x += a ; y = x+1;

se pedirmos para selecionar as dependências de y, então para deletar o ramo se (c), obtemos:

x = c+1; a ~~= 2;~~ **~~afirmar~~ (!c);** x += a~~; y = x+1;~~

onde vemos que a instrução a=2; desaparece e depois não serve mais.

Veremos a seguir como também podemos determinar que x vale 1 usando o cálculo de WP e que, portanto, y vale 2 usando a propagação de constantes.

---

O problema geral com a imposição de restrições é que elas também podem influenciar os resultados de outras análises, como a análise de valor. Veremos um caso especial abaixo com propagação constante.

## D.2.2 Movendo para um ponto do programa

A restrição que acabamos de discutir também pode ser dada implicitamente pela especificação de um ponto do programa. É então uma questão de poder deletar o que não é usado quando a execução da função passa por este ponto. Tal consulta pode, por exemplo, ser usada no contexto de uma análise de impacto.

Um primeiro passo pode ser remover ramificações incompatíveis com o ponto observando apenas a sintaxe.

**Exemplo 7**

Na seguinte sequência:

se (c) P : x += a ; senão y = x+1 ;

se o usuário deseja examinar a passagem para o ponto P, o desvio else pode ser eliminado:

**afirmar (c);** P : x += a ; sen~~ão y = x+1 ;~~

Atenção, isso só é verdade se a sequência não estiver em loop...

Mas pode ser mais interessante calcular uma pré-condição que garanta que passemos pelo ponto determinado e usar esse resultado como uma restrição.

**Nota 1 :** Atenção, deve-se notar que esta pré-condição geralmente não pode ser calculada por um simples WP porque isso garante que uma propriedade seja verdadeira se a função for executada com entradas que satisfaçam a pré-condição, mas não garante que não haja outras possíveis casos. Isso se deve à aproximação introduzida pelo cálculo do WP dos loops.

## D.2.3 Propagação de constantes

Outras reduções podem ser realizadas explorando as constantes do programa.

**Exemplo 8**

Na seguinte sequência:

```
x = c? 1:-1; se (x<0) f(x); senão g(x);
```

se estudarmos esta sequência com um valor inicial 0 para c, gostaríamos de saber como determinar isso:

— x=-1, —
portanto o segundo teste é verdadeiro, —
portanto g não será chamado, — e f será
chamado neste contexto com o valor -1.

Queremos, portanto, obter:

```
afirmar (c == 0); x = -1; f(-1);
```

Este cálculo pode ser facilmente realizado por uma análise de valor externa, mas deve ser possível transmitir a restrição para ela (aqui: *c* = 0;).

Na ferramenta anterior, o módulo *de fatiamento* (FLD) era acoplado a um módulo de propagação constante (CST) da seguinte forma: — para cada função
filtrada, são calculados os estados CST correspondentes, — se uma ramificação é
excluída a pedido do usuário , a CST é informada
reduzir o estado,
— se uma constante estiver associada a um dado, (por exemplo, ao especializar uma função com um
parâmetro constante), o CST também é informado, — ao calcular uma função
filtrada, para cada salto condicional, o CST é consultado para saber se um ramo está inacessível (ÿ), e se
assim for, os elementos correspondentes são marcados *mM* (código morto), — para cada chamada de
função, se for conhecido por determinar que uma
ou mais entradas são
constantes, a especialização desta função é solicitada,
— para cada chamada de função, se soubermos determinar que uma ou mais saídas são constantes,
fornecemos a informação ao CST para que ele possa propagá-la.

**Exemplo 9**

Vamos voltar ao exemplo anterior onde sabemos que *c* = 0 em 1:

```
/*1*/ x = c ? /*2*/1 : /*3*/-1; /*4*/if (x<0) /*5*/f(x); else /*6*/g(x);
```

Então nós temos :

— em /*1*/: *c* = 0, —
em /*2*/: ÿ já que o teste é falso, — em /*4*/:
*x* = ÿ1, — em /*5*/:
sempre *x* = ÿ1, — em /*6*/: ÿ já que
o teste é sempre verdadeiro,

Esta sequência é, portanto, reduzida a:

```
x = c? 1:-1; se (x>0) f_1(x); senão g(x);
```

onde f_1 é a função especializada f com sua entrada em -1.

Quando a análise possibilita determinar o valor preciso de uma variável, também pode ser interessante utilizar essa informação para realizar uma transformação do programa antes de produzir o resultado. Não podemos realmente fazer isso antes, porque podemos ter que agrupar várias funções onde o valor pode ser diferente.

A implementação dos seguintes pontos não foi feita na ferramenta anterior, pelo que seria necessário ver se é possível integrá-los na nova:

— exclui o argumento correspondente a uma entrada constante. De fato, durante o cálculo da função especializada, não sabíamos como excluir o argumento correspondente, mas propagamos o valor da mesma forma para excluir quaisquer ramos mortos.

**Exemplo 10**

No exemplo anterior, podemos especializar f em f_1 porque sabemos que *x* = ÿ1.
Mas também gostaríamos de transformar: void f_1 (int x) { ... }
em : void f_1 (void) { int x = -1; ... }

— suprime o cálculo de uma condição if que é sempre verdadeira quando não há outra.
Isso não foi feito porque no módulo de propagação constante não havia nenhum ponto de programa correspondente à ramificação else ausente . O estado ÿ que teria sido associado a ele, portanto, não estava lá para permitir essa exclusão.

### D.2.4 Usando WP

Como vimos, a propagação constante pode tornar possível especializar funções e cortar ramificações do programa.

Este tipo de cálculo é realizado a priori por propagação direta. Para fazer a propagação reversa, gostaríamos de usar um cálculo WP.

O princípio de uso seria o seguinte:

**Exemplo 11**

O usuário só está interessado na ramificação *then* do *if.* Portanto, introduzimos a afirmação $y < 0$:

| Fonte do programa | ramo cortado |
|---|---|
| int f (int c, int x) {<br><br>int y = c ? 1:-1; se (y < 0)<br>{ int z = y - 1;<br>g(z); retornar z +<br>x; }<br><br>outro<br>retornar 0;<br>} | int f (int c, int x) {<br><br>int y = c ? 1:-1; **afirmar (y**<br>**< 0);** { int z = y - 1;<br>g(z); retornar z +<br>x; }<br><br>} |
| **WP** | **Propagação constante** |
| int f (int c, int x) { **/* (c == 0); */**<br>int y = c ? 1:-1;<br>afirmar (y < 0); { int z = y -<br>1; g(z); retornar z + x;<br><br>}<br><br>} | int f (int c, int x) { /* (c == 0); */<br>int y = **/* 0 ? 1 : */** -1;<br>afirmar ( -1 < 0); { int z = **-1** - 1; g<br>( -2); retornar **-2** + x;<br><br>}<br><br>} |

Neste exemplo, o uso do WP permite determinar que *c* vale 0, e que então é possível propagar esta constante.

A principal dificuldade na utilização de tal cálculo é o controle das operações.

A mais simples é deixar o usuário controlar a propagação usando comandos interativos.

A ideia é permitir que ele forneça uma restrição, rastreá-la usando WP e deduzir outras propriedades, utilizáveis, por exemplo, por análise de valor.

Tenha cuidado, aqui novamente, como na observação 1 na página 67, você deve primeiro se certificar de que a aproximação introduzida pelo WP dos loops está de fato na direção certa em relação ao que você deseja obter...

# Bibliografia

[Baudin(2004)] P. Baudin. Análise de impacto: especificação da ferramenta. Relatório Técnico DTSI/SOL/04-187, CEA, junho de 2004.

[Choi e Ferrante(1994)] J.-D. Choi e J. Ferrante. Corte estático na presença de instruções goto. ACM Trans. Programa. Lang. Syst., 16(4):1097–1113, 1994. ISSN 0164-0925. URL http://doi.acm.org/10.1145/183432.183438.

[Chung et al.(2001)Chung, Lee, Yoon e Kwon] IS Chung, WK Lee, GS Yoon e YR Kwon. Divisão do programa com base na especificação. Em SAC '01: Proceedings of the 2001 ACM Symposium on Applied Computing, páginas 605–609, Nova York, NY, EUA, 2001. Imprensa ACM. ISBN 1-58113-287-5 URL http://doi.acm.org/10.1145/372202.372784.

[CodeSurfeur()] CodeSurfeur. Codesurfeur. URL http://www.codesurfer.com/.

[Comuzzi e Hart(1996)] JJ Comuzzi e JM Hart. Fatiamento do programa usando as pré-condições mais fracas. Em FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods, páginas 557–575, Londres, Reino Unido, 1996. Springer-Verlag. ISBN 3-540-60973-3. URL http://portal.acm.org/citation.cfm?id=729684.

[Daoudi et al.(2002)Daoudi, Ouarbya, Howroyd, Danicic, Marman, Fox e Ward] M. Daoudi, L. Ouarbya, J. Howroyd, S. Danicic, M. Marman, C. Fox e MP Ward. Consus: Uma abordagem escalável para corte condicional, 2002. URL http://citeseer.ist.psu.edu/daoudi02consus.html.

[Ferrante et al. (1987) Ferrante, Ottenstein e Warren] J. Ferrante, KJ Ottenstein e JD Warren. O grafo de dependência do programa e seu uso na otimização. ACM Trans. Programa. Lang. Syst., 9(3): 319–349, 1987. ISSN 0164-0925. URL http://doi.acm.org/10.1145/24039.24041.

[Fox et al. (2001) Fox, Harman, Hierons e Danicic] C. Fox, M. Harman, R. Hierons e S. Danicic. Condicionamento retrógrado: uma nova técnica de especialização de programas e sua aplicação à compreensão de programas. Em IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension, páginas 89–97, Washington, DC, EUA, 2001. IEEE Computer Society. URL http://citeseer.ist.psu.edu/fox01backward.html.

[Fox et al. (2004) Fox, Danicic, Harman e Hierons] C. Fox, S. Danicic, M. Harman e RM Hierons. Consit: Um fatiador de programa condicionado totalmente automatizado. Software: Practice and Experience, 34(1):15–46, 2004. URL http://dx.doi.org/10.1002/spe.556.

[Harman et al. (2001a) Harman, Hierons, Fox, Danicic e Howroyd] M. Harman, R. Hierons, C. Fox, S. Danicic e J. Howroyd. Fatiamento pré/pós condicionado. Em ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), página 138, Washington, DC, EUA, 2001a. Sociedade de Computação IEEE. ISBN 0-7695-1189-9.

[Harman et al.(2001b)Harman, Hu, Munro e Zhang] M. Harman, L. Hu, MC Munro e X. Zhang. GUSTT : Um sistema de fatiamento amorfo que combina fatiamento e

transformação. Em Working Conference on Reverse Engineering, páginas 271–280, 2001b.
URL http://citeseer.ist.psu.edu/harman01gustt.html.

[Horwitz et al. (1988) Horwitz, Reps e Binkley] S. Horwitz, T. Reps e D. Binkley. Em fatiamento interprocedimento usando gráficos de dependência. Em Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, volume 23-7, páginas 35–46, Atlanta, GA, junho de 1988. URL http://citeseer.nj.nec.com/horwitz90interprocedural.html .

[Kumar e Horwitz (2002)] S. Kumar e S. Horwitz. Melhor divisão de programas com saltos e interruptores. Em Abordagens fundamentais para engenharia de software, páginas 96–112, 2002.
URL http://citeseer.ist.psu.edu/kumar02better.html.

[Lyle e Wallace (1997)] J. Lyle e D. Wallace. Usando a ferramenta de fatiamento do programa unravel para avaliar o software de alta integridade. Em Proceedings of Software Quality Week, maio de 1997.
URL http://citeseer.ist.psu.edu/lyle97using.html.

[Ottenstein e Ottenstein(1984)] KJ Ottenstein e LM Ottenstein. O grafo de dependência do programa em um ambiente de desenvolvimento de software. Em SDE 1: Anais do primeiro simpósio de engenharia de software ACM SIGSOFT/SIGPLAN em ambientes práticos de desenvolvimento de software, páginas 177–184, Nova York, NY, EUA, 1984. ACM Press.
ISBN 0-89791-131-8. URL http://doi.acm.org/10.1145/800020.808263.

[Pacalet(2007)] A.Pacalet. Cálculo de dependências em um programa c. Relatório técnico, INRIA, junho de 2007. (Nota: *documento de trabalho em desenvolvimento).*

[Pacalet e Baudin(2005)] A. Pacalet e P. Baudin. Análise de impacto; a ferramenta de corte.
Relatório Técnico DTSI/SOL/05-141, CEA, maio de 2005. (Nota: *Uma versão mais recente deste documento está disponível internamente).*

[Reps(1993)] T. Reps. Manual de referência do sistema de integração de programas de Wisconsin: Release, 1993. URL http://citeseer.nj.nec.com/reps93wisconsin.html.

[SlicingWebLinks()] FatiandoWebLinks. Cortando links da web. URL http://www.infosun.fmi.uni-passau.de/st/staff/krinke/slicing/.

[Dica(1995)] F. Dica. Uma pesquisa sobre técnicas de divisão de programas. Journal of Programming Languages, 3:121–189, 1995. URL http://citeseer.nj.nec.com/tip95survey.html.

[Desvendar()] Desvendar. Desvendar. URL http://www.itl.nist.gov/div897/sqg/unravel/unravel.html.

[Ward (2002)] M. Ward. Fatiamento de programa por meio de transformações fermat, 2002. URL http://citeseer.ist.psu.edu/ward02program.html.

[Weiser(1979)] M. Weiser. Fatias de programa: investigações formais, psicológicas e práticas de um método automático de abstração de programas. Tese de doutorado, Universidade de Michigan, Ann Arbor, 1979.

[Weiser(1981)] M. Weiser. Corte do programa. Em ICSE '81: Proceedings of the 5th international conference on Software engineering, páginas 439–449, Piscataway, NJ, EUA, 1981. IEEE Press. ISBN 0-89791-146-6.

[Xu et al.(2005)Xu, Qian, Zhang, Wu e Chen] B. Xu, J. Qian, X. Zhang, Z. Wu e L. Chen. Uma breve pesquisa sobre divisão de programas. SIGSOFT Softw. Eng. Notes, 30(2):1–36, 2005. ISSN 0163-5948. URL http://doi.acm.org/10.1145/1050849.1050865.