Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 15 - *Path Sensitivity*, análise interprocedimental, análise de fluxo de controlo

Análises dependentes do caminho

Dependência do caminho

até agora ignoramos os valores resultantes das condições em estruturas condicionais/cíclicas assumindo que estas eram não-deterministas.
as análises resultantes desta assunção são designadas de **insensíveis ao caminho** (*path insensitive*)

no entanto as condições podem fornecer informações valiosas para as análises estáticas
considere o exemplo seguinte:
a análise de intervalo com alargamento irá concluir que no final do ciclo

- a variável x está no intervalo $[-\infty, \infty]$
- a variável y está no intervalo $[0, \infty]$
- a variável z está no intervalo $[-\infty, \infty]$

$\Rightarrow$ aproximação demasiada pessimista!

```
x = input;
y = 0;
z = 0;
while (x > 0) {
   z = z+x;
  if (17 > y) { y = y+1; }
  x = x-1;
}
```

Modelar as condições

vamos incluír estes valores na informação disponível para cada restrição associada aos nodos do CFG

juntamos uma instrução `assert` E à linguagem.

semântica intencional: afirma-se que a expressão booleana E é verdade neste ponto do programa

- se E não for verdade, então a execução é interrompida com um **erro de execução**
- a intenção é inserir *asserts* ***defensivos***, i.e. que pretendemos serem sempre verificados numa execução padrão.

a informação extraída de um nodo $[\![assert\ E]\!]$ permitirá juntar informação para análises como a analise de intervalo
em alternativa poderíamos aumentar as restrições de cada nodo que segue uma condição (i.e. condicional ou ciclo)

vantagem do `assert` E: E pode ser **mais informativa** do que a condição da instrução condicional C (i.e. $E \implies C$) e pode ser colocada em **qualquer ponto do programa**

juntamos `assert` E após cada nodo que introduz uma condição E no início de cada ramo em que sabemos E ser verificado

juntamos um `assert` $\neg E$ da mesma forma, no primeiro onto onde sabemos que E não é mais verificado

a semântica original do programa **é mantida**

```
x = input;
y = 0;
z = 0;
while (x>0) {
  assert(x>0);
  z = z+x;
  if (17>y) {
    assert (17 > y);
    y = y+1;
    }
  x = x-1;
  }
assert(!(x>0));
```

depende fortemente da análise em causa que deverá tentar tirar o máximo proveito da informação extraída de E.

no limite minimo $[\![v]\!] = \mathit{Join}(v)$ com $v = \mathit{assert}\ E$ está correcto, mas não é de todo informativo (não ganhamos informação nenhuma)

para a analise de intervalos, uma solução pode ser (com $v = \mathit{assert}\ (x > E)$):

$$[\![v]\!] = \mathit{Join}(v)[x \to \mathit{gt}(\mathit{Join}(v)(x), \mathit{eval}(\mathit{Join}(v), E))]$$

onde $\mathit{gt}([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \sqcap [l_2, \infty]$
é possível extender naturalmente para casos como $x = E$, $x < E$ ou $x \leq E$

no caso geral `assert` E (E condição booleana qualquer) é desafiante retirar informação para a análise de intervalo. $\Rightarrow$ **área activa de investigação**

no ponto do programa assinalado a análise de intervalo pode agora concluir

$$x = [-\infty, 0]$$

$$y = [0, 17]$$

$$z = [0, \infty]$$

```
x = input;
y = 0;
z = 0;
while (x>0) {
  assert(x>0);
  z = z+x;
  if (17>y) {
    assert (17 > y);
    y = y+1;
    }
  x = x-1;
  }
assert(!(x>0));
// Neste ponto!
```

Exercícios

- comprove que as restrições para o assert são correctas e monótonas
- discuta mais casos possíveis para E em que se pode gerar restrições úteis para a análise de intervalo

com a instrução `assert` temos uma forma simples de *path sensitivity* designada as vezes de sensibilidade ao controlo *control sensitivity* ou de analise de ramificação *branched analysis*

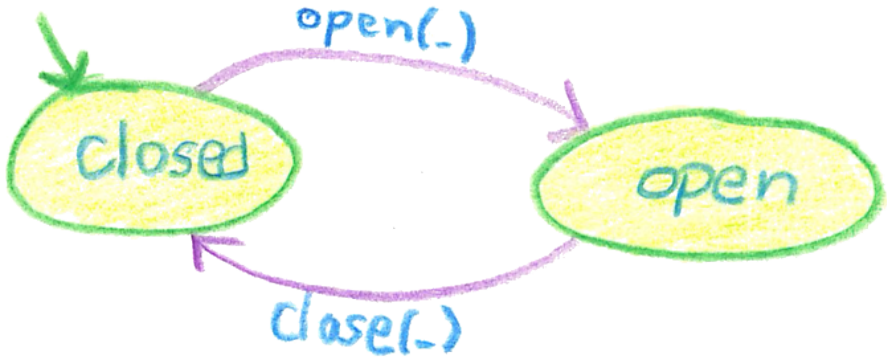
mas não é suficiente para interligar (correlacionar) os ramos

```
if (17 > x) { ... }
... // instruções que não mudam x
if (17 > x) { ... }
...
```

Abertura e fecho de ficheiros

imaginemos que dispomos de duas funções `open(f)` e `close (f)` que operam (abrem e fecham, resp.) sobre o ficheiro `f`

requisitos destas instruções: nunca fechar um ficheiro fechado, nunca abrir um ficheiro aberto



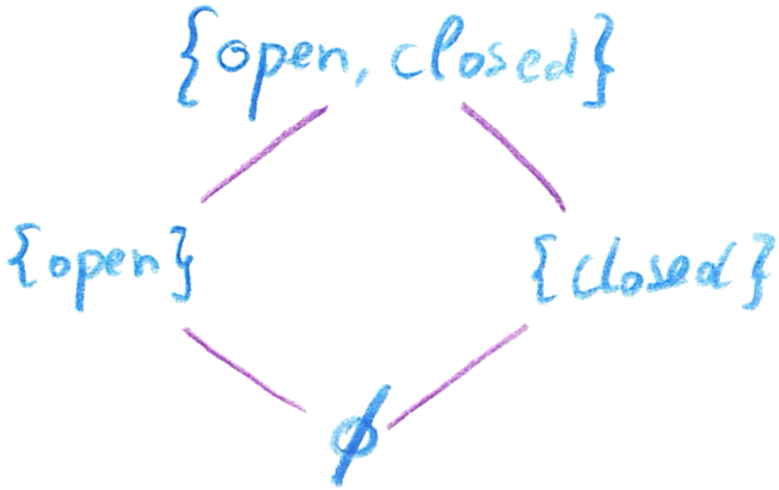
queremos dispor de uma analise estática que saiba verificar o cumprimento destes requisitos

por exemplo, neste caso, onde assumimos a manipulação de um único ficheiro

```
if (condition) {
  open();
  flag = 1;
} else {
flag = 0;
}
... //sem open
... //nem close
...// nem alterações
...// a  flag
if (flag) {
close();
}
```

o reticulado por usar é:

$$L = (\mathcal{P}(\{open, closed\}), \subseteq)$$



aqui, para um nodo v do CFG

$$Join(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

para cada vértice v do CFG, $[\![v]\!]$ aponta para o possível estatuto do ficheiro **após** o vértice

As restrições para os vértices relevantes são

$$[\![entry]\!] = \{closed\}$$

$$[\![open()]\!] = \{open\}$$

$$[\![close()]\!] = \{closed\}$$

para os vértices v restantes

$$[\![v]\!] = Join(v)$$

antes da instrução `close()`, a análise de concluí que o ficheiro pode estar nos estados $\{open, closed\}$
o que é contra-intuitivo visto que esta instrução é invocada somente quando `open()` foi previamente invocada

```
if (condition) {
  open();
  flag = 1;
} else {
flag = 0;
}
...
if (flag) {
//aqui!
close();
}
```

Escreva as restrições produzidas pelo programa, calcule a solução e mostre que a solução no vertice `flag` (o vertice que corresponde a última instrução condicional) é $\{open, closed\}$

Uma análise mais súbtil

obviamente uma análise mais certeira deverá ser capaz de tomar conta do valor de `flag`

nesta nossa segunda tentativa, utilizamos o reticulado seguinte

$$L = (\mathcal{P}(\{open, closed\}) \times \mathcal{P}(\{flag, \neg flag\}), \subseteq \times \subseteq)$$

juntamos também `asserts` para modelar as condicionais

mas mesmo assim continuamos com o **conhecimento impreciso** $(\{open, closed\}, \{flag, \neg flag\})$ tanto no fim do primeiro if como no fim do programa

$\Rightarrow$ a analise só sabe que `open` ***pode*** ter sido invocado e que `flag` ***pode*** ser 0

```
if (condition) {
  assert(condition);
  open();
  flag = 1;
} else {
  assert(!condition);
  flag = 0;
}// <-- aqui
...
if (flag) {
  assert(flag);
  close();
} else {
  assert(!flag);
}// <-- e aqui
```

a analise anterior não é suficientemente precisa
não relaciona o estatuto do ficheiro com o valor de `flag`
designa-se de **independent atribute analysis**

precisamos aqui de uma análise que destaca e actua nas **relações** entre variáveis
isto obriga à gestão de estados abstractos **múltiplos** para cada ponto de programa
um por cada *contexto de caminho*

aqui: contexto de caminho = um predicado sobre os estados do programa
no exemplo considerado, precisamos do reticulado

$$L = P \to \mathcal{P}(\{open, closed\}) \qquad \text{(isomórfico a } \mathcal{P}(P \times \{open, closed\})\text{)}$$

onde $P = \{flag, \neg flag\}$ é o conjunto (finito) dos contextos de caminho

definição da analise e das suas restrições com base nestes contextos de caminho

$$[\![entry]\!] = \lambda p.\{closed\}$$

$$[\![open()]\!] = \lambda p.\{open\}$$

$$[\![close()]\!] = \lambda p.\{closed\}$$

para as atribuições envolvendo a variável `flag`

$$[\![flag = 0]\!] = [\neg flag \rightarrow \bigcup_{p \in P} Join(v)(p), flag \rightarrow \emptyset]$$

$$[\![flag = n]\!] = [flag \rightarrow \bigcup_{p \in P} Join(v)(p), \neg flag \rightarrow \emptyset]$$

$$[\![flag = E]\!] = \lambda q. \bigcup_{p \in P} Join(v)(p)$$

onde n é uma constante inteira não nula, E uma outra expressão qualquer

os caminhos $\rightarrow \emptyset$ representam os caminhos **impraticáveis** (*infeasible path*)

para as instruções `assert`

$$assert(flag) = [flag \rightarrow Join(v)(flag), \neg flag \rightarrow \emptyset]$$

$$assert(\neg flag) = [\neg flag \rightarrow Join(v)(\neg flag), flag \rightarrow \emptyset]$$

para os restantes vértices

$$[\![v]\!] = Join(v) = \lambda p. \bigcup_{w \in pred(v)} [\![w]\!](p)$$

para o programa em exemplo

$$
\begin{array}{lcl}
[\![entry]\!] & = & \lambda p.\{closed\} \\
[\![\text{condição}]\!] & = & [\![entry]\!] \\
[\![assert(\text{condição})]\!] & = & [\![\text{condição}]\!] \\
[\![open()]\!] & = & \lambda p.\{open\} \\
[\![flag = 1]\!] & = & [flag \rightarrow \bigcup_{p \in P} [\![open()]\!](p), \neg flag \rightarrow \emptyset] \\
[\![asert(!\text{condição})]\!] & = & [\![assert(\text{condição})]\!] \\
[\![flat = 0]\!] & = & [flag \rightarrow \bigcup_{p \in P} [\![asert(!\text{condição})]\!](p), flag \rightarrow \emptyset] \\
[\![\ \ldots \ ]\!] & = & \lambda p.([\![flag = 1]\!](p) \cup [\![flag = 0]\!](p)) \\
[\![flag]\!] & = & [\![\ \ldots \ ]\!] \\
[\![assert(flag)]\!] & = & [\neg flag \rightarrow [\![flag]\!](flag), \neg flag \rightarrow \emptyset] \\
[\![close()]\!] & = & \lambda p.\{closed\} \\
[\![assert(!flag)]\!] & = & [\neg flag \rightarrow [\![flag]\!](\neg flag), flag \rightarrow \emptyset] \\
[\![exit]\!] & = & \lambda p.([\![close()]\!](p) \cup [\![assert(!flag)]\!](p))
\end{array}
$$

Solução mínima

a solução mínima é, para cada $[\![v]\!](p)$

	$\neg flag$	$flag$
$[\![entry]\!]$	$\{closed\}$	$\{closed\}$
$[\![$condição$]\!]$	$\{closed\}$	$\{closed\}$
$[\![assert($condição$)]\!]$	$\{closed\}$	$\{closed\}$
$[\![open()]\!]$	$\{open\}$	$\{open\}$
$[\![flag = 1]\!]$	$\emptyset$	$\{open\}$
$[\![asert($!condição$)]\!]$	$\{closed\}$	$\{closed\}$
$[\![flat = 0]\!]$	$\{closed\}$	$\emptyset$
$[\![\ \ldots \ ]\!]$	$\{closed\}$	$\{open\}$
$[\![flag]\!]$	$\{closed\}$	$\{open\}$
$[\![assert(flag)]\!]$	$\emptyset$	$\{open\}$
$[\![close()]\!]$	$\{closed\}$	$\{closed\}$
$[\![assert(!flag)]\!]$	$\{closed\}$	$\emptyset$
$[\![exit]\!]$	$\{closed\}$	$\{closed\}$

a análise produz o elemento do reticulado $[\neg flag \rightarrow \{closed\}, flag \rightarrow \{open\}]$ a seguir ao primeiro *if*

a restrição $assert(flag)$ elimina a possibilidade de o ficheiro estar fechado neste ponto

asseguramos assim que $close()$ **só é chamado quando o ficheiro está aberto**

Desafios nas análises sensíveis ao caminho

quem desenha análises estáticas destas deve escolher criteriosamente P

- são em geral combinações de predicados que aparecem como condições no código analisado
- abordagem complementar: **refinamento iterativo** (e.g. *counter-example guided abstraction refinement*)
  - P começa como um contexto de caminho trivial
  - gradualmente P é refinado juntando predicados relevantes até as propriedades desejadas serem provadas ou rejeitadas ou ainda se tornar óbvio que não se consegue selecionar mais predicados pertinentes

problema: é habitual estas combinações levaram a computação a uma explosão exponencial

- para k predicados, 2^k contextos diferentes
- o corte da redundância habitualmente moderar esta explosão

raciocinar sobre `assert`: como actualizar os elementos do reticulado de forma suficientemente precisa?
pode envolver **sistemas de prova**!

Exercícios

- na definição das restrições por nodo v da analise relacional, dê as restrições apropriadas para $assert(!flag)$
- no caso do exemplo de código *TIP* apresentado, definimos o reticulado com base numa aplicação $P \to \mathcal{P}(A)$. Mostre que é isomórfico a um reticulado com base em $\mathcal{P}(P \times A)$
- descreva uma variante da análise do programa TIP apresentada que tire proveito da combinação com a analise de propagação de constante

podemos por exemplo lançar análises auxiliares , por exemplo a análise de sinal e a análise de propagação de constante
com base nos resultados, temos mais informação para melhor antever as atribuições à variável `flag`
podemos mudar

$$[\![open()]\!] = \lambda p.\{open\}$$

para a definição mais precisa (mas correcta)

$$[\![open()]\!] = \lambda p.if\ \ Join(v)(p) = \emptyset\ then\ \emptyset\ else\ \{open\}$$

Exercícios

- no caso da alteração considerada a regra *open*, agrimente que a alteração é mais precisa (do que a regra original) mas também correcta
- construa mais uma variante do exemplo *open*/*close* onde a propriedade por garantir só pode ser estabelecida como uma escolha de P que inclua um predicado que **não ocorra** numa condicional do programa. Um programa como este é em geral bastante desafiante para análises que usam a técnica do refinamento iterativo

Exercício

considere o programa seguinte

```
if (condition) {
  flag = 1;
  } else {
  flag = 0;
  }
...
if (condition) {
  open();
  }
...
if (flag) {
  close();
}
```

mostre como uma analise sensível ao caminho pode provar que este programa invoca *close* se e só se *open* foi previamente invocado

Análise Interprocedimental

uma análise que se debruça sobre o corpo de uma função é designada de **análise intraprocedimental**

uma análise que considera um programa na sua integralidade com todas as chamadas à funções que este considera, é designada de **análise interprocedimental**

abordagem ingénua:

- analisar cada função separadamente
- ser pessimista na aproximação do resultado de cada chamada de funções
- mas... raramente a precisão global é adequada

CFG para programas

a ideia é

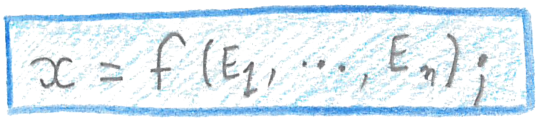
- construir um CFG para cada função
- colar todos estes CFGs por forma a espelhar as chamadas realizadas nos diferentes blocos de programa e os *returns*

para tal precisamos de ter em conta

- a passagem dos parâmetros
- o retorno de valores
- o comportamento dos valores das variáveis locais aquando das diferentes chamadas
  neste ultimo caso, temos de saber lidar com funções recursivas e as suas eventuais variáveis (que partilham os mesmos identificadores)

Hipótese de trabalho simplificadora

vamos assumir que todas as chamadas de função tem a forma seguinte:

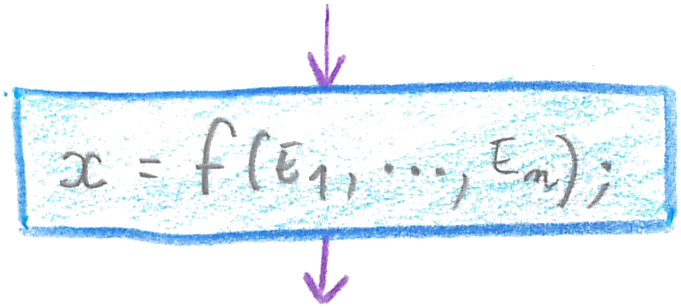


esta forma pode ser obtida para todos os programas, mediante transformação de programa

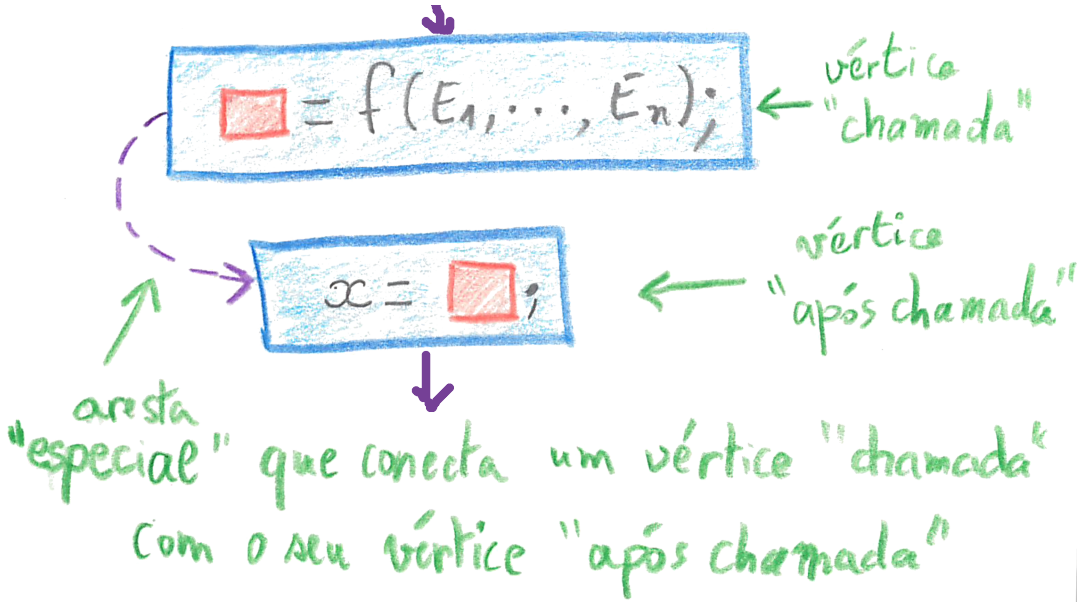
assim esta hipótese não é redutora

defina esta transformação de programa, i.e. mostre como cada programa pode ser rescrito introduzindo eventualmente novas variáveis (*frescas*) temporárias

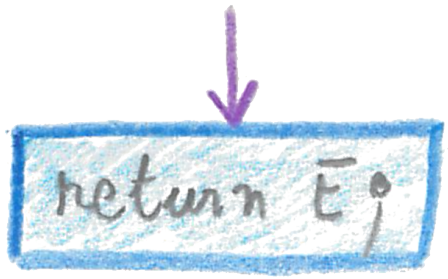
partimos o vértice da chamada original



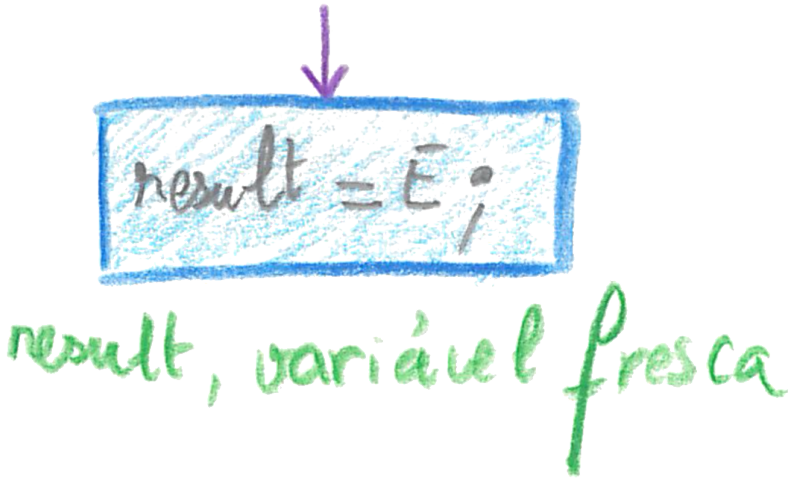
em dois nodos



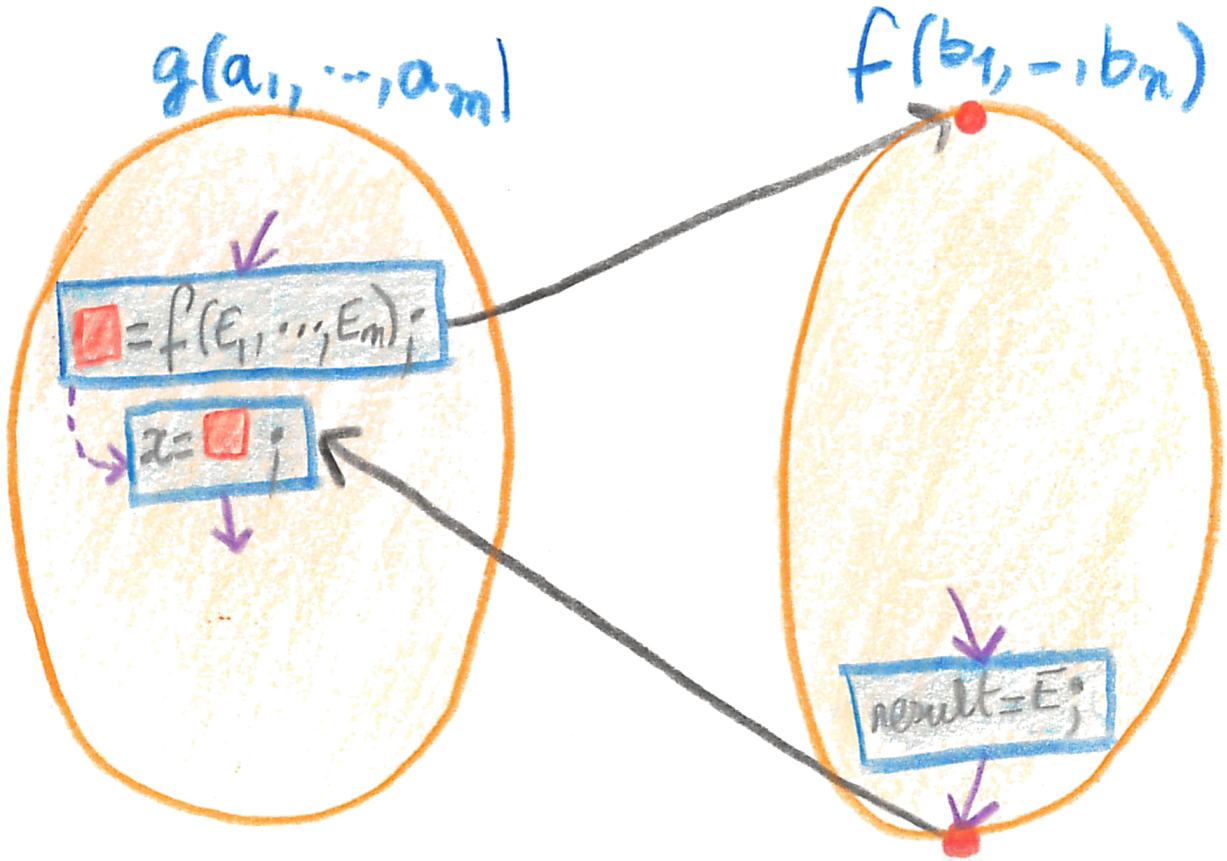
transforma-se cada vértice *return*



numa atribuição



e junta-se arestas de chamada e de retorno



podemos doravante aplicar novamente a framework monótona!

para vértices *call/entry*: ter em atenção os modelos de avaliação dos parâmetros antes da ligação aos (identificadores dos) parâmetros formais é um aspecto importante no caso das funções recursivas

para vértices *after-call/exit*

- como no caso de uma atribuição $x = \mathit{result}$
- mas também recupera as variáveis locais activa antes da chamada via a aresta *call* $\longrightarrow$ after-call

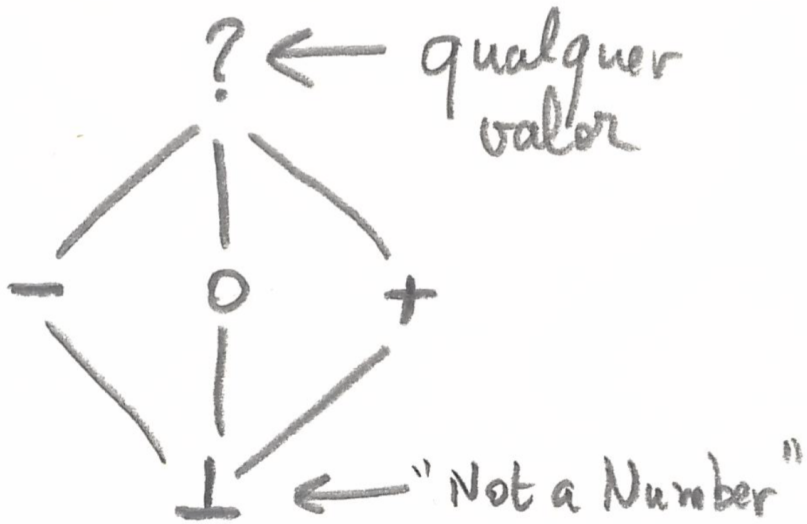
os detalhes restantes dependem especificamente da análise em causa

Exercício

Quantas arestas pode ter um CFG interprocedimental de um programa com CFG (intraprocedimental) de n vértices ?

Um exemplo: os sinais, outra vez

relembremos a analise intraprocedimental de sinal cujo reticulado *Sign* para os valores abstracto era



e cujo reticulado para os estados abstractos era $\mathit{Vars} \to \mathit{Sign}$

Um exemplo: os sinais, outra vez

Para proceder a uma analise interprocedimental de sinal, temos de ter agora em conta dois tipos de vértices do CFG suplementar

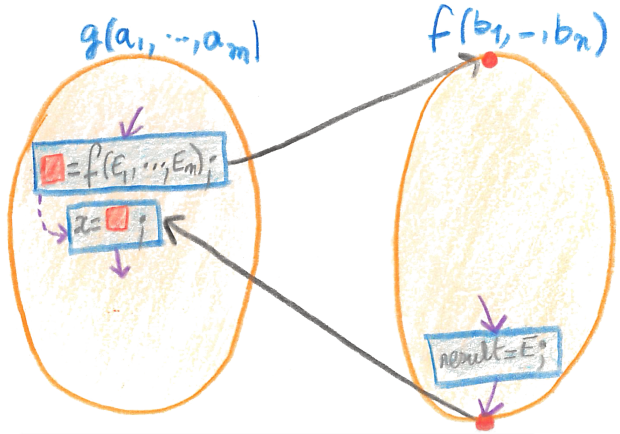
as restrições para um vértice de entrada v de uma função são $f(b_1, \cdots, b_n)$

$$[\![v]\!] = \sqcup_{w \in pred(v)} \bot[b_1 \mapsto eval([\![w]\!], E_1), \cdots, b_n \mapsto eval([\![w]\!], E_n)]$$

as restrições para os vértices *after-call* v de label $x = \square$ com uma chamada para o vértice v'

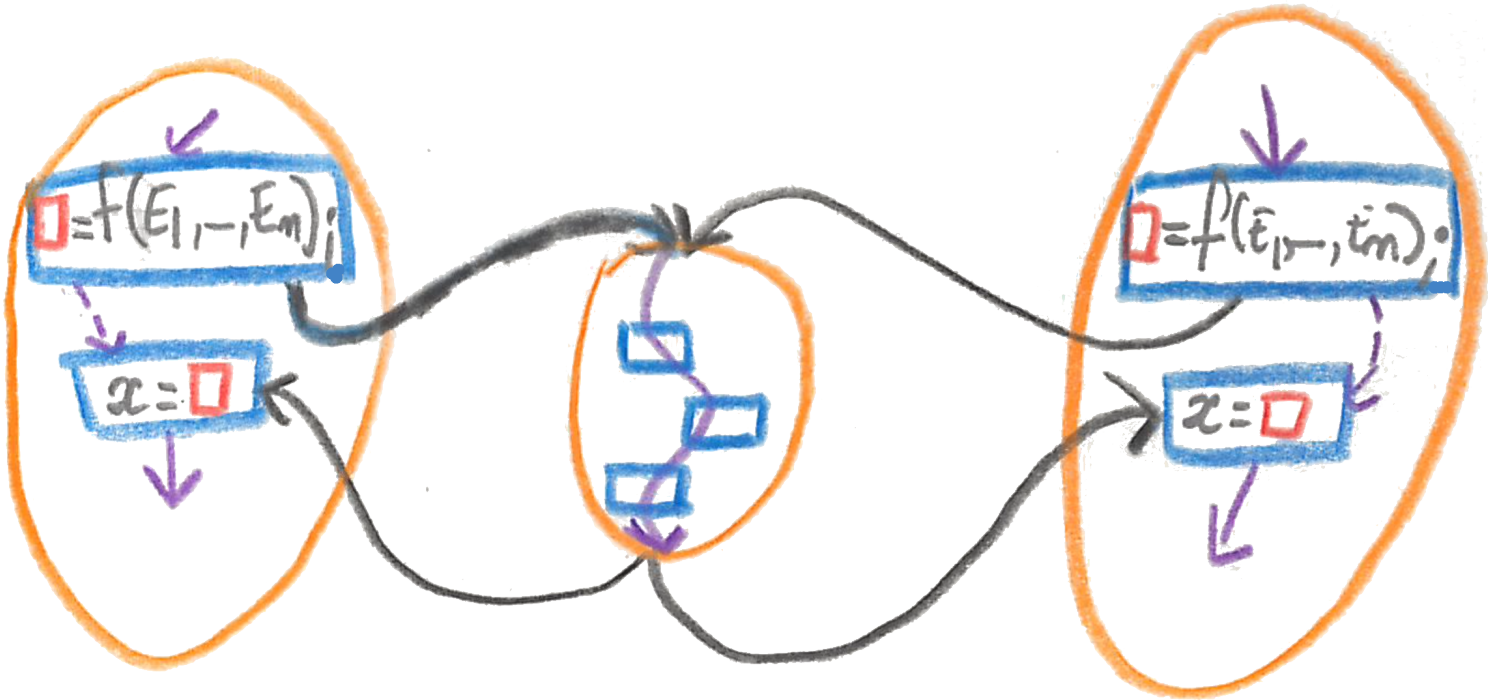
$$[\![v]\!] = [\![v']\!][x \mapsto [\![w]\!](result)] \quad \text{onde } w \in pred(v)$$

de realçar que não temos nesta fase na linguagem TIP nem variáveis globais, nem *heap*, nem funções aninhadas, nem funções de ordem superior



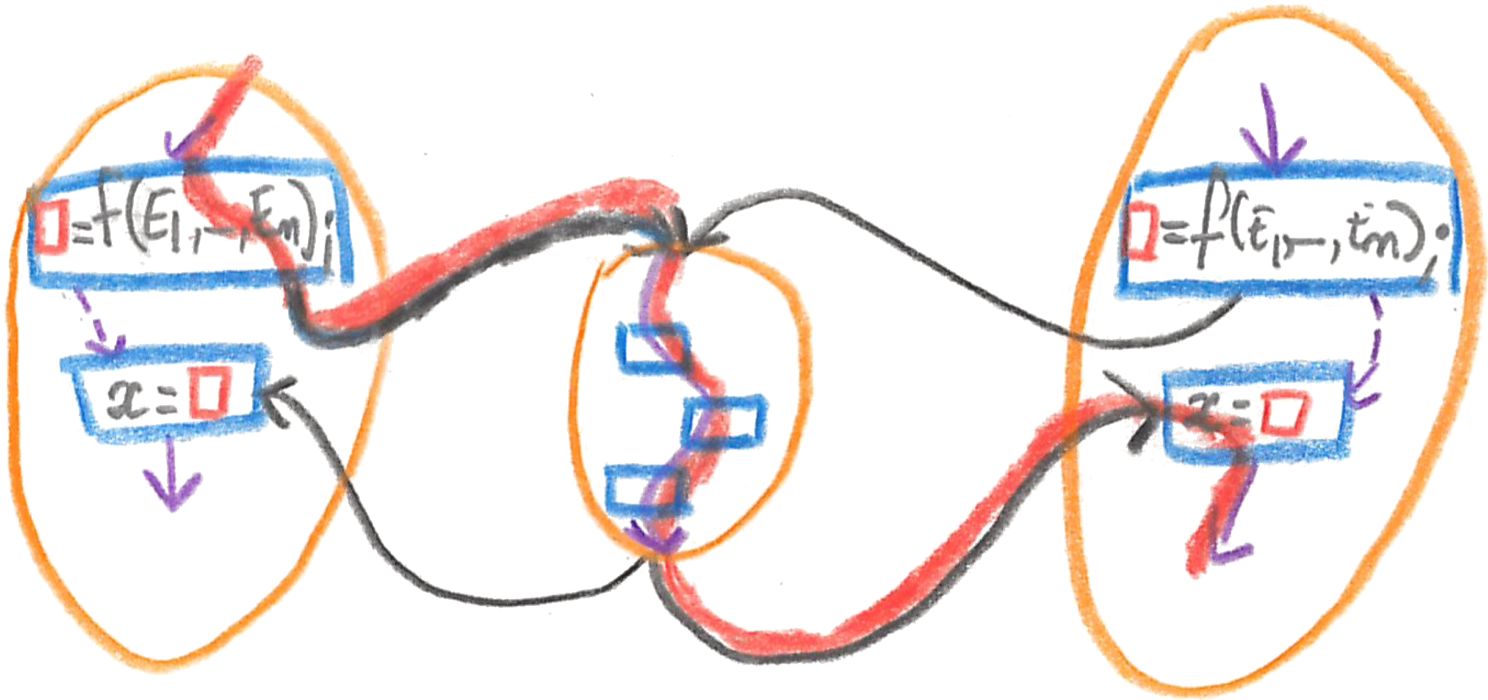
Caminho interprocedimental inválido

considere o CFG seguinte



Caminho interprocedimental inválido

este caminho não é válido (retorno errado) e deve então ser detectado e evitado



qual é o sinal do valor de retorno de g

```
f(z) {
  return z*42;
}

g() {
  var x,y;
  x = f(0);
  y = f(87);
  return x + y;
}
```

a analise apresentada devolve ?

Inlining de funções

substituí a chamada a uma função pelo seu corpo e insere atribuições suplementares para lidar explicitamente com passagem de parâmetros e o valor de retorno

permite evitar caminhos interprocedimental inválidos

mas para caminhos aninhados (funções aninhadas etc.) temos uma explosão exponencial do CFG
pior... não sabe lidar com funções recursivas

usa-se heurísticas para determinar quando aplicar esta técnica (baseadas em compromissos entre o tamanho do CFG e a precisão)

Exemplo, com inlining

qual é o sinal do valor de retorno de g

```
f(z) {
  return z*42;
}

g() {
  var x,y;
  x = 0 * 42;
  y = 87 * 42;
  return x + y;
}
```

após duas aplicações da técnica de *inlining*, a analise apresentada devolve $+$

o *inlining* de funções permite uma certa sensibilidade ao contexto (que designamos de **análise polivariante**)

podemos, em alternativa ao inlining baseado em cópia física, realizar esta operação de forma **lógica**

para tal, substituímos o reticulados para os estados abstractos L por $C \to L$ onde C é o conjunto dos **contextos**

- os contextos são abstracções do estado à entrada da função
- C deve ser finito para garantir a altura finita de $C \to L$
- uma algoritmo *lista de afazeres* pode calcular cada mapa a medida da necessidade (*by-need* - uma função é analizada só se num determinado contexto a sua resposta é necessária)

a escolha de C não é imediata; existam várias estratégias para a escolha

Inlining de um nível

sejam $c_1, \cdots, c_n$ os vértices de chamadas de funções do programa considerado

definimos $C = \{c_1, \ldots, c_n\} \cup \{\epsilon\}$

- cada vértice define agora o seu próprio **contexto de chamada** (*call context*)
  (ϵ representa o contexto de chamada da função principal)
- o contexto assemelha-se ao endereço de retorno da tabela de activação que está no topo da pilha de chamada

tem o mesmo efeito do que o *inlining* físico (de um nível), mas de facto sem necessitar a cópia de CFG

em geral é fácil generalizar as restrições de uma análise insensível ao contexto para este cenário e este reticulado

A abordagem *string de chamada*

sejam $c_1, \cdots, c_n$ os vértices de chamadas de funções do programa considerado

definimos C como o conjunto das strings sobre o alfabeto $c_1, \cdots, c_n$ e de comprimento máximo k

- uma string destas representa as localizações das k primeiras chamadas na pilha de chamadas
- a string ϵ representa, mais uma vez, o contexto de chamada da função principal

para $k = 1$, estamos no contexto do *inlining* de nível 1

Exemplo: análise interprocedimental de sinal com strings de chamada $k = 1$

reticulado para estados abstractos : $C \to (Vars \to Sign)$ onde C é o conjunto dos vértices "chamadas" do CFG

```
 f(z)  {
    var t1,t2;
    t1 = z*6;
    t2 = t1*7;
    return t2;//<---
}
  ...
x=f(0); //c1
y = f(87); // c2
...
```

$$[c_1 \mapsto [z \mapsto 0, t_1 \mapsto 0, t_2 \mapsto 0]$$

$$c_2 \mapsto [z \mapsto +, t_1 \mapsto +, t_2 \mapsto +]]$$

(implicitamente os outros valores são mapeados para $\perp$)

proponha um exemplo de programa onde é preciso fixar $k = 2$ para não perder precisão.

a abordagem *strings de chamada* consider o fluxo de controlo mas surge a questão: porquê distinguir dois locais de chamada se os seus estados abstractos são os mesmos?

a abordagem funcional considera em alternativa o fluxo de dado

um caso especial: se o reticulado original para os estados abstractos for L (e L é finito), substitui-lo por $L \to L$ (ou seja, o conjunto C é L)

como é habitual em análises sensíveis ao fluxo, associamos um elemento do reticulado a cada ponto do programa

cada elemento do reticulado é agora um mapa m que devolve um elemento $m(x)$ do reticulado original para cada x possível onde x descreve o estado à entrada da função

Exemplo: análise interprocedimental de sinal com strings de chamada $k = 1$

reticulado para estados abstractos : $C \to (Vars \to Sign)$ onde $C = (Vars \to Sign)$

```
 f(z)  {
    var t1,t2;
    t1 = z*6;
    t2 = t1*7;
    return t2;//<---
}
  ...
x=f(0); //c1
y = f(87); // c2
...
```

$$[[z \mapsto 0] \mapsto [z \mapsto 0, t_1 \mapsto 0, t_2 \mapsto 0]$$

$$[z \mapsto +] \mapsto [z \mapsto +, t_1 \mapsto +, t_2 \mapsto +]]$$

(implicitamente os outros valores são mapeados para $\perp$)

nota: nesta análise, contextos e estados abstractos consideram somente as variáveis que estão no alcance (*scope*)

o elemento do reticulado para um vértice de saída de uma função é um ***resumo de função*** que mapea o *input* de função abstracta para respectivo *output*

esta informação pode ser aproveitada nos vértices de chamadas

quando se entra numa função tendo em conta um estado abstracto x

- considerar o resumo de função s para esta função
- se $s(x)$ já fo calculado, usar esta informação para modelar integralmente o corpo da função, continuar a análise directamente a partir do vértice *after-call*

evita o problema com caminhos interprocedimental inválidos

mas pode ser **impraticável computacionalmente** se L for demasiado grande

Análise de fluxo de controlo

uma linguagem que suporta funções de ordem superior, apontadores de função, objectos (etc.) vê o fluxo de controlo e o fluxo de dados interligarem-se de forma complexa e subtil

a cada chamada não é trivial perceber que código é realmente invocado.

o CFG deixa de ser de construção fácil, mas a sua definição é importante

nestas situações é necessário proceder a uma análise prévia para construir o CFG (e proceder para a outras análises que dependem do CFG)

Análise de fluxo de controlo

uma análise de fluxo de controlo (***CFA***) tem por objectivo aproximar o CFG

- de forma conservadora, calcula o destino de todas as possíveis chamadas no local em que estas são feitas
- resposta trivial: **todas**!

as análises de fluxo de controlo são em geral **insensíveis ao caminho**

- suportam-se na AST
- o próprio CFG... não existe ainda e está a ser construido!
- (alternativamente: pode ser construído ***on-the-fly*** durante a análise de fluxo de informação)

uma analise posterior pode usar o CFG:

- uma CFA sensível ao caminho pode ser menos conservadora
- esta pode ser iterada

o lambda calculo oferece o contexto de base para uma ilustração da construção de CFA

$$
\begin{array}{rcll}
E & ::= & \lambda x.E & \text{(definição de função)} \\
 & | & E_1\ E_2 & \text{(aplicação de função)} \\
 & | & x & \text{(variável)}
\end{array}
$$

para simplificar assumimos aqui que não temos fenômenos de captura de variáveis (todas as variáveis, livres e ligadas tem identificadores distintos)

lembrete: um **fecho** $\lambda x.$ abstraí a função $\lambda x.E$ dos contextos onde esta pode evoluir (resolve a dependência de todas as variáveis livres e representa **concretamente** a função $\lambda x.E$)

objectivo: cada local de chamada $(E_1\ E_2)$ determina quais são as possíveis funções para E_1 dentro do conjunto $\{\lambda x_1, \lambda x_2, \ldots, \lambda x_n\}$

seja $C = \{\lambda x_1, \lambda x_2, \ldots, \lambda x_n\}$ o conjunto dos fechos de um programa p considerado para a análise

para cada nodo v da AST de p, introduzimos a variável $[\![v]\!] \in \mathcal{P}(C)$ que representa os fechos que este nodo por vir a ser após a sua avaliação

assim, para $v = \lambda x.E$, temos a restrição

$$\lambda x \in [\![\lambda x.E]\!]$$

para $v = E_1\ E_2$ temos a restrição condicional

$$\lambda x \in [\![E_1]\!] \implies ([\![E_2]\!] \subseteq [\![x]\!] \wedge [\![E]\!] \subseteq [\![E_1\ E_2]\!]) \quad \text{(para cada função } \lambda x.E)$$

Exercício: e para um nodo variável $v = x$?

mostre que as restrições resultantes podem ser transformadas para um contexto standard de inequações monótonas e resolvidas como um clássico cálculo de ponto fixo

a análise de fecho é uma instância de uma análise mais geral designada de **framework cúbica**

imaginemos que temos

- um conjunto de *tokens* $T = \{t_1, t_2, \ldots, t_k\}$
- um conjunto de variáveis $\{x_1, \ldots, x_n\}$ que tomam valor no conjunto das partes de T
- uma coleção de restrições da forma
  - $t \in x$ ou
  - $t \in x \implies y \subseteq z$

o objectivo é calcular a solução mínima
esta solução é **única**, visto que as soluções são fechadas por intersecção.

Existe um algoritmo cuja complexidade é cúbica

em que sentido o facto das soluções ao conjunto das restrições serem fechadas por intersecção implica que a solução mínima é única

cada variável é mapeada para um nodo de uma DAG (*directed acyclic graph*)

cada nodo possuí um *bitvector* em $\{0,1\}^k$ (inicializado com 0s)

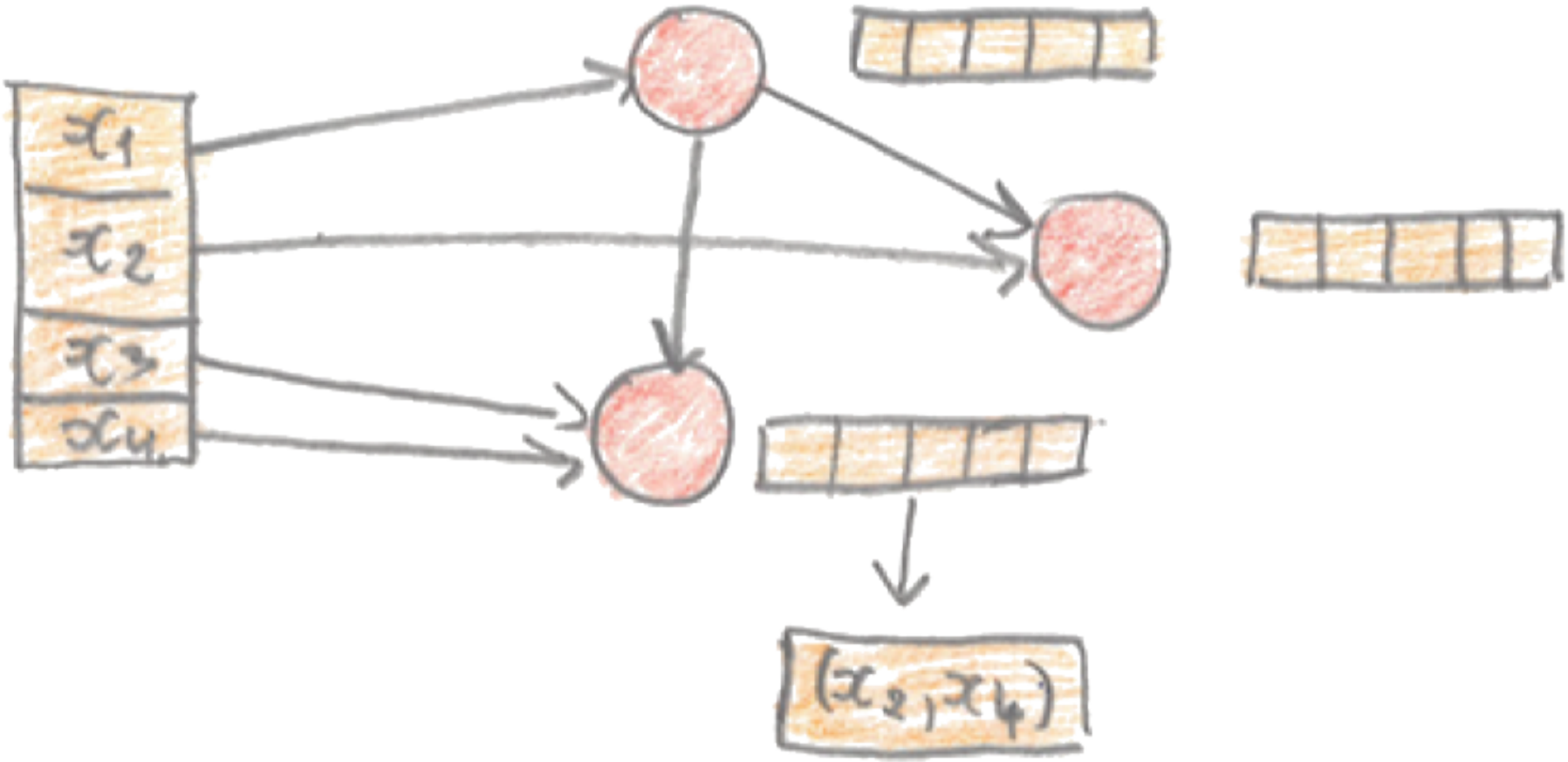
cada bit está relacionado com uma lista de pares de variáveis
esta lista modela as restrições condicionais

as arestas modelam a inclusão de restrições

invariante do método: os diferentes *bitvectors* modelam em cada instante da procura da solução a solução mínima das restrições analisada até este instante

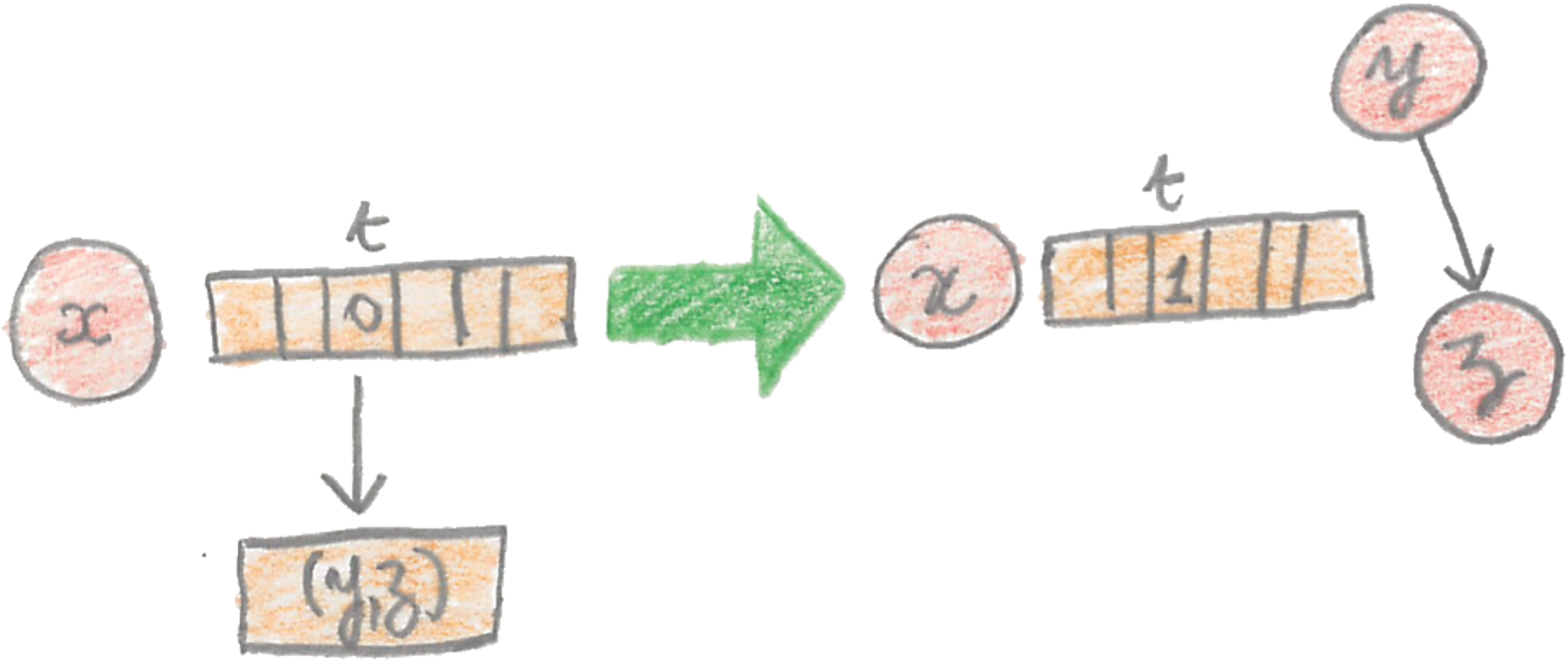
as restrições são acrescentadas uma por uma

Um exemplo



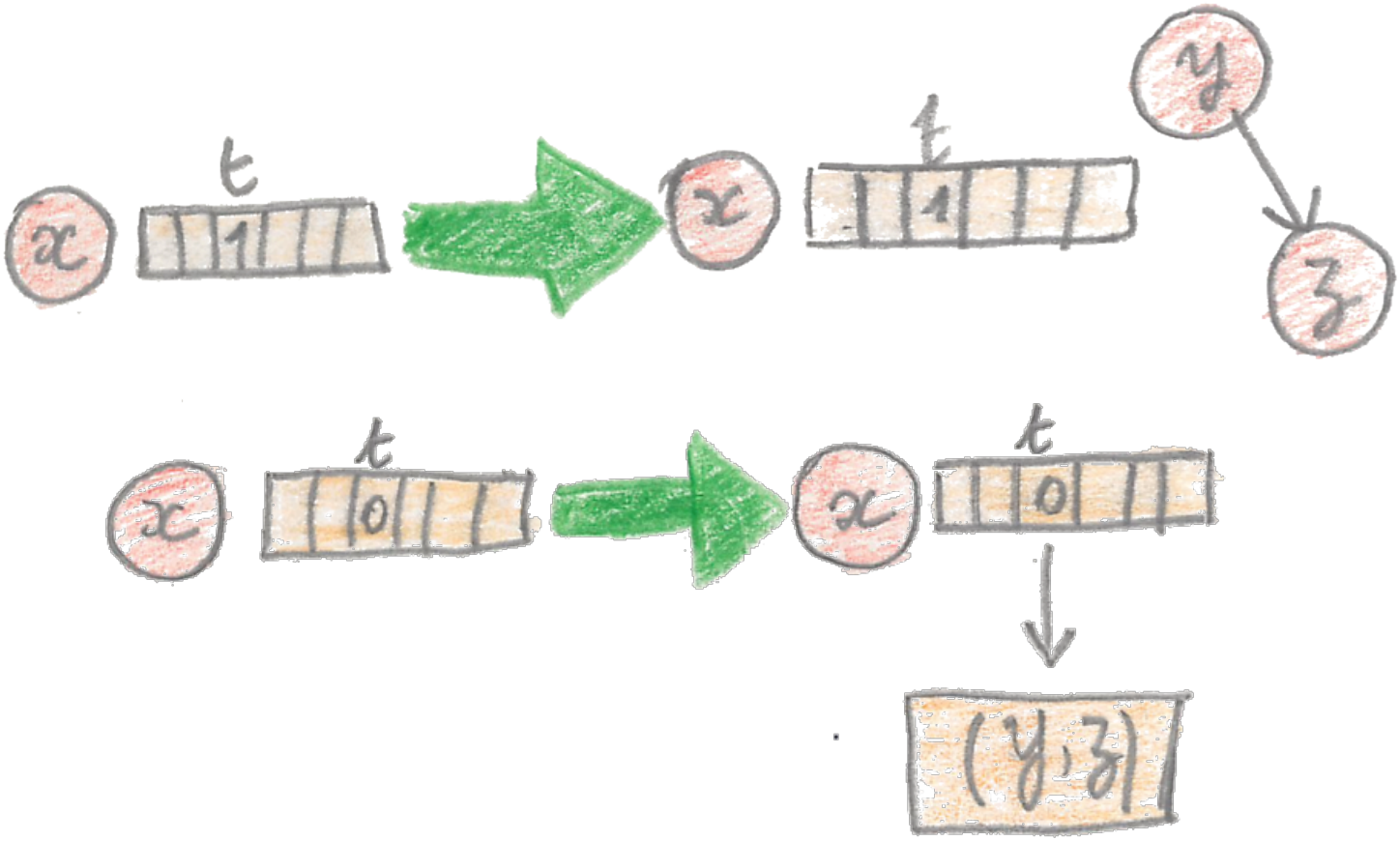
restrições da forma $t \in x$

- procurar o nodo associado a x
- atribuir 1 ao bit que corresponde a t
- se a lista de pares para t não está vazia, então juntar os vértices correspondentes ao par no DAG



restrições da forma $t \in x \implies y \subseteq z$

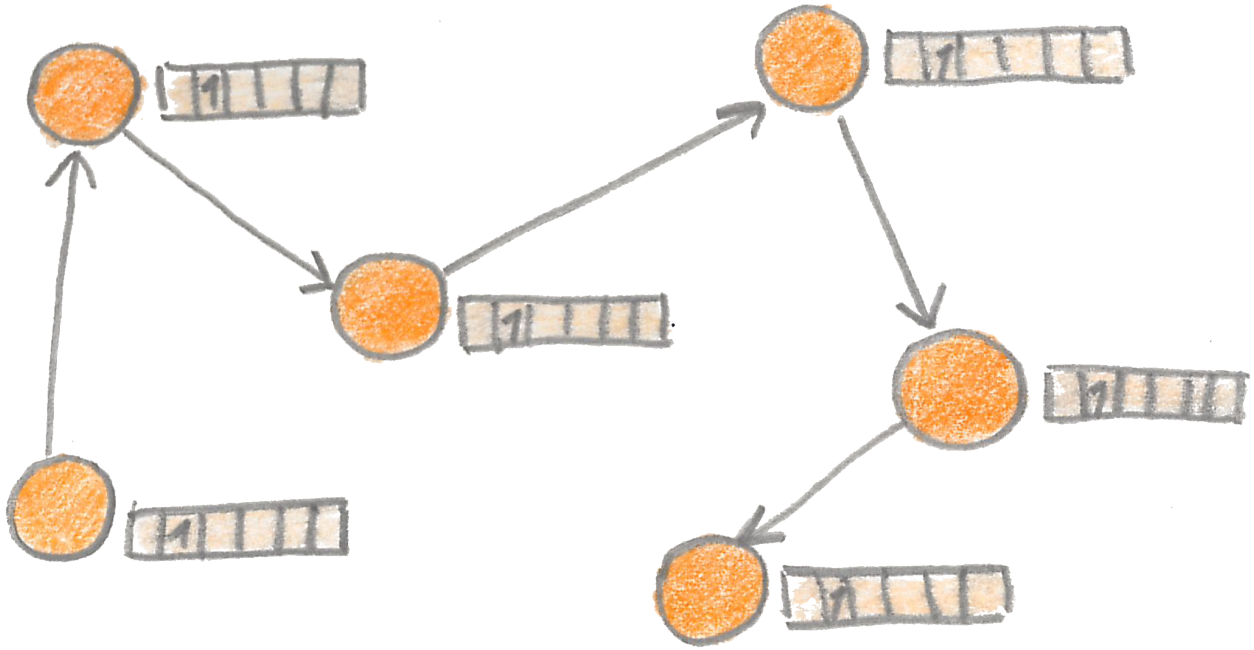
- verificar se o bit que corresponde a t é 1
- no caso positivo, juntar ao DAG arestas de y para z
- no caso negativo juntar (y, z) na lista de pares associados a t



se um vértice acrescentado forma um ciclo

- fundir os vértice que formam o ciclo num só vértice
- calcular a união dos respectivos *bitvectors*
- concatenar a lista dos pares
- actualizar o mapa das variáveis de acordo com as alterações

propagar os valores de todos os bits recentemente inicializados para todas as arestas do DAG



Complexidade temporal no pior caso

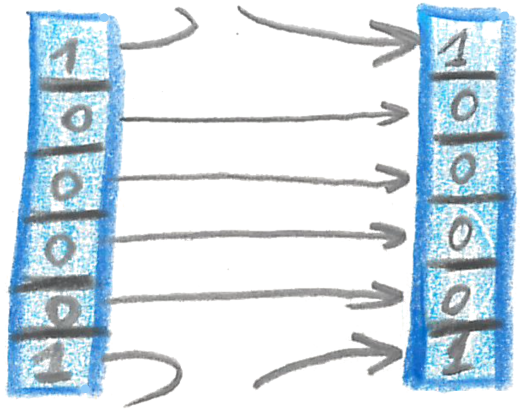
hipótese de trabalho: programa de tamanho n, com $\mathcal{O}(n)$ funções e $\mathcal{O}(n)$ aplicações

$\mathcal{O}(n)$ restrições simples, $\mathcal{O}(n^2)$ restrições condicionais

$\mathcal{O}(n)$ vértices, $\mathcal{O}(n^2)$ arestas, $\mathcal{O}(n)$ bits por vértices

tempo total para a propagação do bitvectors: $\mathcal{O}(n^3)$ tempo total para o colapso dos ciclos: $\mathcal{O}(n^3)$ tempo total para a gestão da lista dos pares: $\mathcal{O}(n^3)$

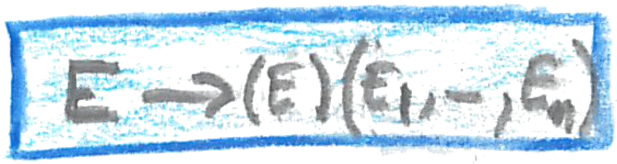
para poupar na propagação dos bits, podemos simular que cada correspondência é feita via um fio que quando usado uma vez é quebrado (logo, não há mais propagação por aí)



em suma, o pior caso é $\mathcal{O}(n^3)$

esta complexidade é conhecida como o *cubic time bottleneck*
variantes deste algoritmo conseguem um tempo em $\mathcal{O}(\frac{n^3}{log(n)})$

para uma chamada desta natureza



não é imediato ver que função é realmente invocada

uma aproximação grosseira mas correcta pode ser: qualquer função com o o número correcto de argumentos

com uma boa análise de fluxo de controlo, conseguimos muito melhores aproximações!

CFA para TIP com apontadores de função

notemos primeiro que $id(E_1, \cdots, E_n)$ pode ser visto como uma instância – açúcar sintáctico – do caso $(id)(E_1, \cdots, E_n)$

podemos assim generalizar a análise previamente exposta

os tokens por considerar são todas as funções $\{f_1, f_2, \cdots, f_k\}$

para cada nodo v do AST introduzimos a variável $[\![v]\!]$ que representa o conjunto das funções nas quais v se pode avaliar

para as definições de funções $f(\cdots)\{\ldots\}$

$$f \in [\![f]\!]$$

para as atribuições $x = E$

$$[\![E]\!] \subseteq [\![x]\!]$$

para chamadas de função *simples* $f(E_1, \cdots, E_n)$

$$[\![E_i]\!] \subseteq [\![a_i]\!] \;\; \textit{para } i = 1, \ldots, n \;\wedge\; [\![E']\!] \subseteq [\![f(E_1, \ldots, E_n)]\!]$$

onde f tem os argumentos $a_1, \ldots, a_n$ e devolve a expressão E'

para chamadas de função *calculadas* $(E)(E_1, \cdots, E_n)$

$$f \in [\![E]\!] \implies ([\![E_i]\!] \subseteq [\![a_i]\!] \;\; \textit{para } i = 1 \ldots, n \;\wedge\; [\![E']\!] \subseteq [\![(E)(E_1, \ldots, E_n)]\!])$$

para cada função f com argumentos $a_1, \ldots, a_n$ e que devolve a expressão E'

podemos usar o sistema de tipos para melhorar a análise
de facto podemos somente considerar as funções candidatas que são tipáveis aquando da chamada para gerar as restrições

Programa exemplo

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = (f)(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

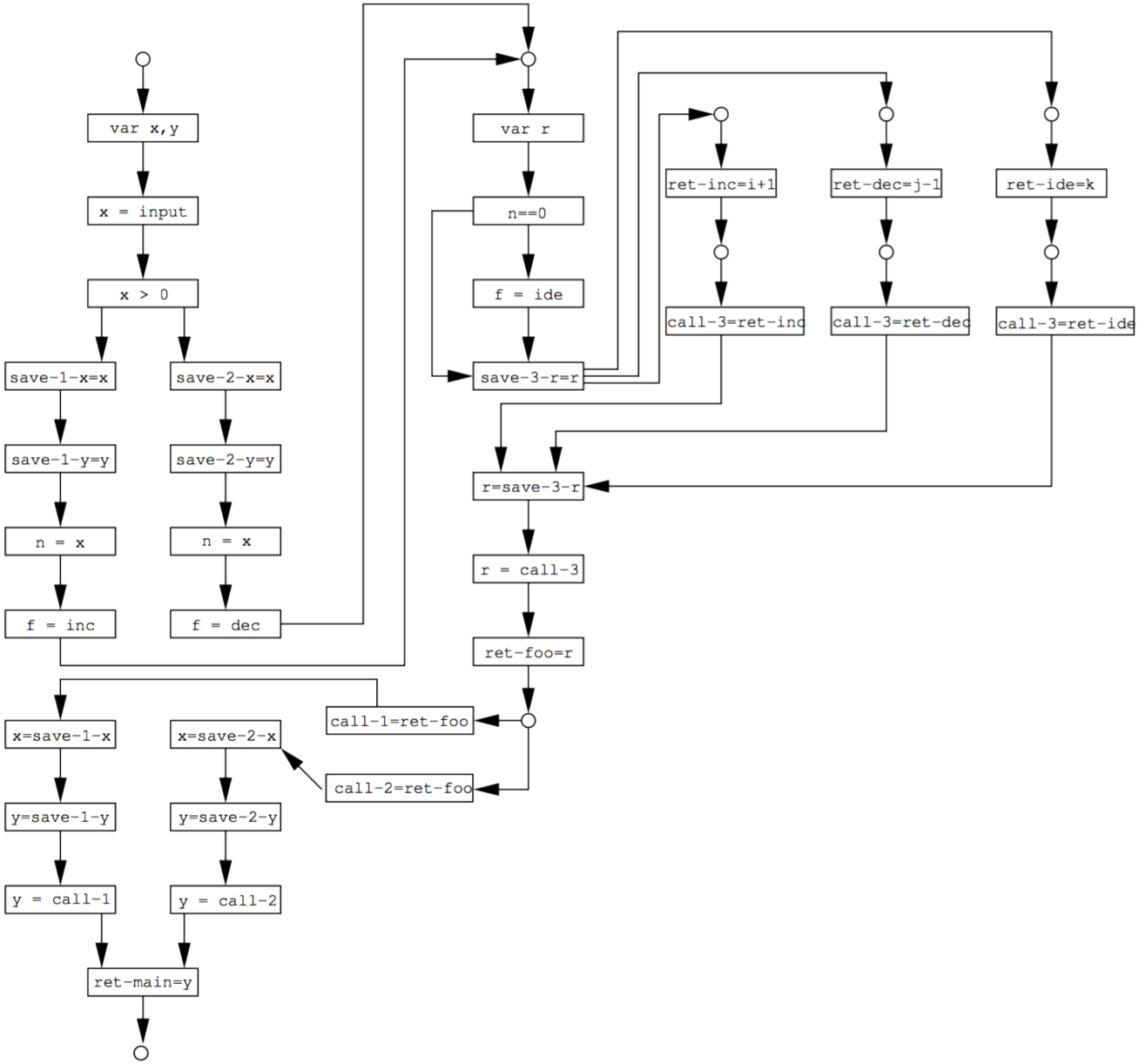
Restrições geradas

$$
\begin{array}{lcll}
inc & \in & [\![inc]\!] & \\
dec & \in & [\![dec]\!] & \\
ide & \in & [\![ide]\!] & \\
[\![ide]\!] & \subseteq & [\![f]\!] & \\
[\![(f)(n)]\!] & \subseteq & [\![r]\!] & \\
inc & \in & [\![f]\!] & \Longrightarrow \quad [\![n]\!] \subseteq [\![i]\!] \ \wedge \ [\![i+1]\!] \subseteq [\![(f)(n)]\!] \\
dec & \in & [\![f]\!] & \Longrightarrow \quad [\![n]\!] \subseteq [\![j]\!] \ \wedge \ [\![j-1]\!] \subseteq [\![(f)(n)]\!] \\
ide & \in & [\![f]\!] & \Longrightarrow \quad [\![n]\!] \subseteq [\![k]\!] \ \wedge \ [\![k]\!] \subseteq [\![(f)(n)]\!] \\
[\![input]\!] & \subseteq & [\![x]\!] & \\
[\![foo(x, inc)]\!] & \subseteq & [\![y]\!] & \\
[\![foo(x, dec)]\!] & \subseteq & [\![y]\!] & \\
foo & \in & [\![foo]\!] & \\
foo & \in & [\![foo]\!] & \Longrightarrow \quad [\![x]\!] \subseteq [\![n]\!] \ \wedge \ [\![inc]\!] \subseteq [\![f]\!] \ \wedge \ [\![(f)(n)]\!] \subseteq [\![foo(x, inc)]\!] \\
foo & \in & [\![foo]\!] & \Longrightarrow \quad [\![x]\!] \subseteq [\![n]\!] \ \wedge \ [\![dec]\!] \subseteq [\![f]\!] \ \wedge \ [\![(f)(n)]\!] \subseteq [\![foo(x, dec)]\!] \\
main & \in & [\![main]\!] &
\end{array}
$$

$$
\begin{array}{lcl}
[\![inc]\!] & = & \{inc\} \\
[\![dec]\!] & = & \{dec\} \\
[\![ide]\!] & = & \{ide\} \\
[\![f]\!] & = & \{inc, dec, ide\} \\
[\![foo]\!] & = & \{foo\} \\
[\![main]\!] & = & \{main\}
\end{array}
$$

com esta informação podemos construir as arestas de chamada e arestas de retorno do CFG interprocedimental

CFG resultante



Uma CFA simples para linguagens orientadas a objectos

no caso de uma invocação de método

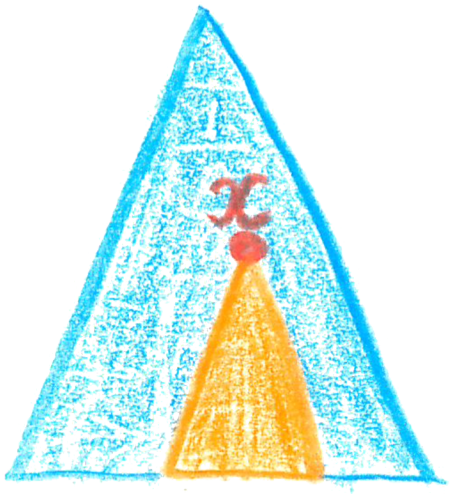
que implementação será realmente invocada?

a CFA anteriormente exposta pode ser utilizada, mas essa não tira proveito das características particulares que uma linguagem orientadas a objectos tem: a hierarquia de classes, subtipagem etc.

Uma CFA simples para linguagens orientadas a objectos

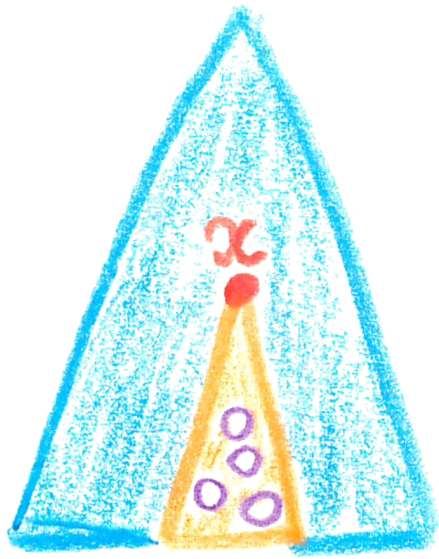
uma solução simples consiste na seleção de todos os métodos chamados *m* com 3 argumentos

a análise da hierarquia de classes (*CHA*) permite melhorar a precisão: considerar somente a parte da hierarquia de classes da qual a classe estática de *x* é raíz



Uma CFA simples para linguagens orientadas a objectos

outra variante, com precisão acrescida: análise rápida de tipos (*RTA*): restringir a CHA às classes que estão *de facto* utilizadas no programa em expressões *new*



mais uma variante, (análise variável de tipos - *VTA*) calcula uma CFA intraprocedimental enquanto faz assunções conservadoras sobre o resto do programa

Leitura

As aulas de Análise Estáticas de Programas desta UC baseam-se em duas fontes essenciais:

- Anders Møller and Michael I. Schwartzbach. Static Program Analysis (acetatos e sebenta).
- Flemming Nielson, Hanne R. Nielson, and Chris L. Hankin. Principles of Program Analysis (um *must read*!).