Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 14 - Análise de Fluxo de Dados

Agenda

1. **catalogação das análises estáticas: *May*, *Must*, *Forward*, *Backward***

2. **analises insensíveis ao fluxo de dados:**
   - analise de vivacidade (*liveness analysis*)
   - análise das expressões disponíveis (*available expressions analysis*)
   - análise das expressões atarefadas (*very busy expressions analysis*)
   - análise do alcance de definições (*reaching definitions analysis*)
   - análise da propagação de constantes (*constant propagation analysis*)

3. **técnicas de alargamento e estreitamento e as suas aplicações**:
   - analise de intervalos (*interval analysis*)
   - técnica do alargamento na procura de pronto fixo
   - técnica do estreitamento na procura de pronto fixo

May, Must, Forward, Backward

Várias análises, um padrão

As analises que vamos explorar definem as suas restrições de forma diferente mas

- participam dos mesmos princípios, das mesmas estruturas
- que aplicam de forma diferente

$\Longrightarrow$ todas são instâncias da **framework monótona** que podemos **catalogar** com base em **duas dimensões**

Forward vs. Backward / **May vs. Must**

Forward vs. Backward

Uma analise prospectiva (*forward analysis*):

- calcula informação num ponto com base nos pontos anteriores (i.e. no comportamento passado)
- ou seja, caracterizam-se pelo RHS das restrições se referir ao conjunto dos nodos predecessores (o conjunto *pred*)
- a analise começa no *entry node* e propaga-se para a frente
- exemplos: *available expressions*, *reaching definitions*

Uma análise em retrocesso (*backward analysis*):

- calcula informação num ponto com base nos pontos que seguem (i.e. a analise do comportamento futuro determina o presente)
- ou seja, caracterizam-se pelo RHS das restrições se referir ao conjunto dos nodos sucessores (o conjunto *succ*)
- a análise começa no *exit node* e propaga-se para trás
- exemplos: *liveness*, *very busy expressions*

May vs. Must

uma análise **may** (*may analysis*):

- descreve informação que é possivelmente verdade
- é uma sobre-aproximação
- exemplos: *liveness*, *reaching definitions*

uma analise **must** (*must analysis*):

- descreve informação que é sem dúvida verdade
- é uma sub-aproximação
- exemplos: *available expressions*, *very busy expressions*

Resumo da classificação

	Forward	Backward
May	exemplo: definições alcançáveis $[\![v]\!]$ descreve os estados que seguem v $Join(v) = \sqcup_{w \in pred(v)} [\![w]\!] = \cup_{w \in pred(v)} [\![w]\!]$	exemplo: vivacidade $[\![v]\!]$ descreve os estados antecedem v $Join(v) = \sqcup_{w \in succ(v)} [\![w]\!] = \cup_{w \in succ(v)} [\![w]\!]$
Must	exemplo: expressões disponíveis $[\![v]\!]$ descreve os estados que seguem v $Join(v) = \sqcup_{w \in pred(v)} [\![w]\!] = \cap_{w \in pred(v)} [\![w]\!]$	exemplo: expressões atarefadas $[\![v]\!]$ descreve os estados antecedem v $Join(v) = \sqcup_{w \in succ(v)} [\![w]\!] = \cap_{w \in succ(v)} [\![w]\!]$

Análise de vivacidade

comecemos por redescobrir uma análise conhecida

uma variável está **viva** num determinado ponto do programa se esta pode ser lida de forma segura daí em diante durante a execução do programa

esta propriedade é indecídivel, mas pode ser aproximada por uma **analise de vivacidade**

consideramos aqui o reticulado das partes do conjunto das variáveis $Vars_P$ do programa P analizado

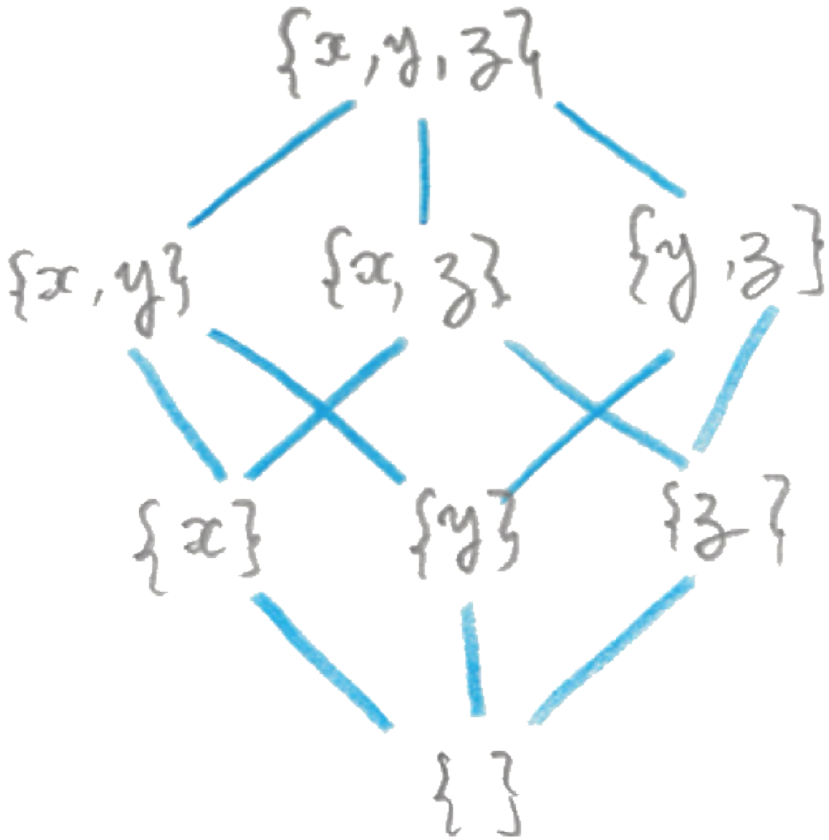
$$L = (\mathcal{P}(Vars_P), \subseteq)$$

fala-se de uma analise parametrizada (o reticulado em causa depende de P)

Um reticulado para a vivacidade

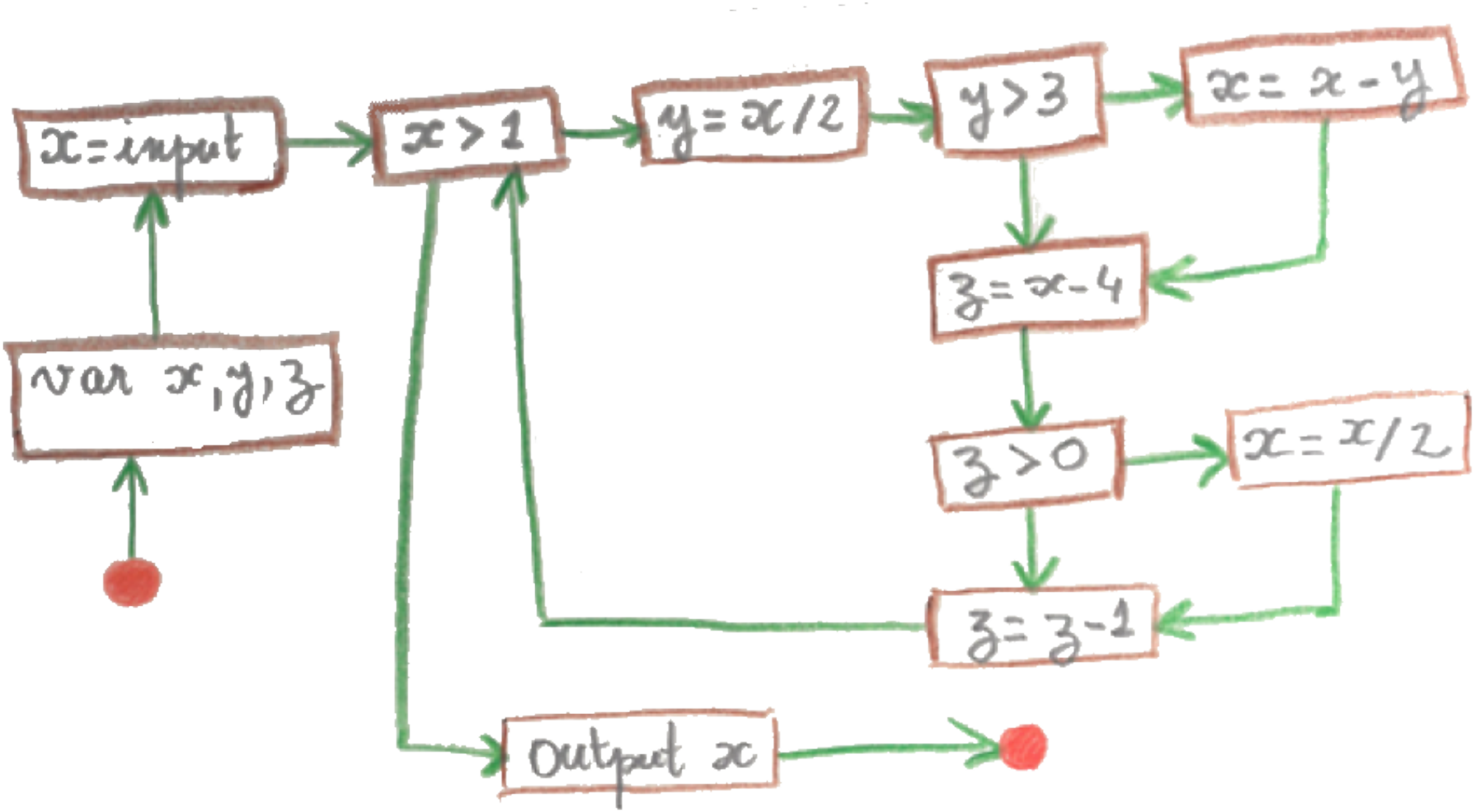
vamos exemplificar com base no exemplo seguinte:

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```



$L = (\mathcal{P}(\{x, y, z\}), \subseteq)$

o CFG na analise de vivacidade



Definição da analise de vivacidade

para cada vértice v do CFG introduzimos as restrições $[\![v]\!]$ associadas a este ponto do CFG.

estas denotam o conjunto das variáveis vivas **antes** deste ponto

teremos o cuidado de calcular uma analise conservadora, já que o conjunto em causa terá a tendência em ser demasiado abrangente.

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

admitimos a existência de uma função $vars(E)$ que devolve o conjunto das variáveis que ocorrem em E

para o nodo de entrada

$$[\![entry]\!] = \emptyset$$

para as declarações

$$[\![var\ id_1, \cdots, id_n]\!] = Join(v) \setminus \{id_1, \cdots, id_n\}$$

para as condições e *output*

$$[\![if\,(E)]\!] = [\![output\ E]\!] = Join(v) \cup vars(E)$$

para as atribuições

$$[\![x = E]\!] = Join(v) \setminus \{x\} \cup vars(E)$$

para qualquer outro nodo v : $[\![v]\!] = Join(v)$

Exercício

Mostre que as equações das restrições para a análise de vivacidade são monótonas, i.e. definem funções monótonas

$$
\begin{array}{lcl}
[\![\mathit{entry}]\!] & = & \emptyset \\
[\![\mathit{var}\ x, y, z]\!] & = & [\![x = \mathit{input}]\!] \setminus \{x, y, z\} \\
[\![x = \mathit{input}]\!] & = & [\![x > 1]\!] \setminus \{x\} \\
[\![x > 1]\!] & = & ([\![y = x/2]\!] \cup [\![\mathit{output}\ x]\!]) \cup \{x\} \\
[\![y = x/2]\!] & = & ([\![y > 3]\!] \setminus \{y\}) \cup \{x\} \\
[\![y > 3]\!] & = & [\![x = x - y]\!] \cup [\![z = x - 4]\!] \cup \{y\} \\
[\![x = x - y]\!] & = & ([\![z = x - 4]\!] \setminus \{x\}) \cup \{x, y\} \\
[\![z = x - 4]\!] & = & ([\![z > 0]\!] \setminus z) \cup \{x\} \\
[\![z > 0]\!] & = & [\![x = x/2]\!] \cup [\![z = z - 1]\!] \cup \{z\} \\
[\![x = x/2]\!] & = & ([\![z = z - 1]\!] \setminus \{x\}) \cup \{x\} \\
[\![z = z - 1]\!] & = & ([\![x > 1]\!] \setminus \{z\}) \cup \{z\} \\
[\![\mathit{output}\ x]\!] & = & [\![\mathit{exit}]\!] \cup \{x\} \\
[\![\mathit{exit}]\!] & = & \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
[\![\mathit{entry}]\!] & = & \emptyset \\
[\![\mathit{var}\ x, y, z]\!] & = & \emptyset \\
[\![x = \mathit{input}]\!] & = & \emptyset \\
[\![x > 1]\!] & = & \{x\} \\
[\![y = x/2]\!] & = & \{x\} \\
[\![y > 3]\!] & = & \{x, y\} \\
[\![x = x - y]\!] & = & \{x, y\} \\
[\![z = x - 4]\!] & = & \{x\} \\
[\![z > 0]\!] & = & \{x, z\} \\
[\![x = x/2]\!] & = & \{x, z\} \\
[\![z = z - 1]\!] & = & \{x, z\} \\
[\![\mathit{output}\ x]\!] & = & \{x\} \\
[\![\mathit{exit}]\!] & = & \emptyset
\end{array}
$$

constatamos que

- as variáveis y e z nunca estão vivas em simultáneo
- o valor calculado na atribuição $z = z - 1$ nunca mais é lido (utilizado)

podemos assim durante a compilação realizar a optimização seguinte:

```
var x,yz;
x = input;
while (x>1) {
  yz = x/2;
  if (yz>3) x = x-yz;
  yz = x-4;
  if (yz>0) x = x/2;
}
output x;
```

façamos uma estimativa na utilização do algoritmo natural apresentado na aula anterior.
assunções de base: o CFG tem n vértices e o programa tem k variáveis

o reticulado $\mathcal{P}^n$ tem assim uma altura igual a $k \times n$. Esta altura limita o úmero de iterações que podem ser realizadas pelo algoritmo
cada elemento do reticulado pode ser representado por um *bitvector* de tamanho $k \times n$
em cada iteração devemos realizar $\mathcal{O}(n)$ intersecções, diferenças ou igualdades sobre conjuntos de tamanho k. O custo destas é assim $\mathcal{O}(k \times n)$
assim a complexidade temporal global é

$$\mathcal{O}(k^2 \times n^2)$$

Exercício

Qual é a complexidade no pior caso da análise de vivacidade com o algoritmo com **lista de afazeres**

Análise de expressões disponíveis

uma expressão (não-trivial) está disponível num determinado ponto de programa se o seu valor actual já foi calculado num ponto anterior da execução

A aproximação natural em geral incluí muito poucas expressões com estatuto de disponibilidade conhecido.

- a analise só pode reportar ***disponível*** se a expressão está totalmente disponível
- uma expressão disponível pode **não ser calculada novamente**

Um reticulado para as expressões disponíveis

vamos exemplificar com base no exemplo seguinte:

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

as 4 sub-expressões não triviais deste programas são:

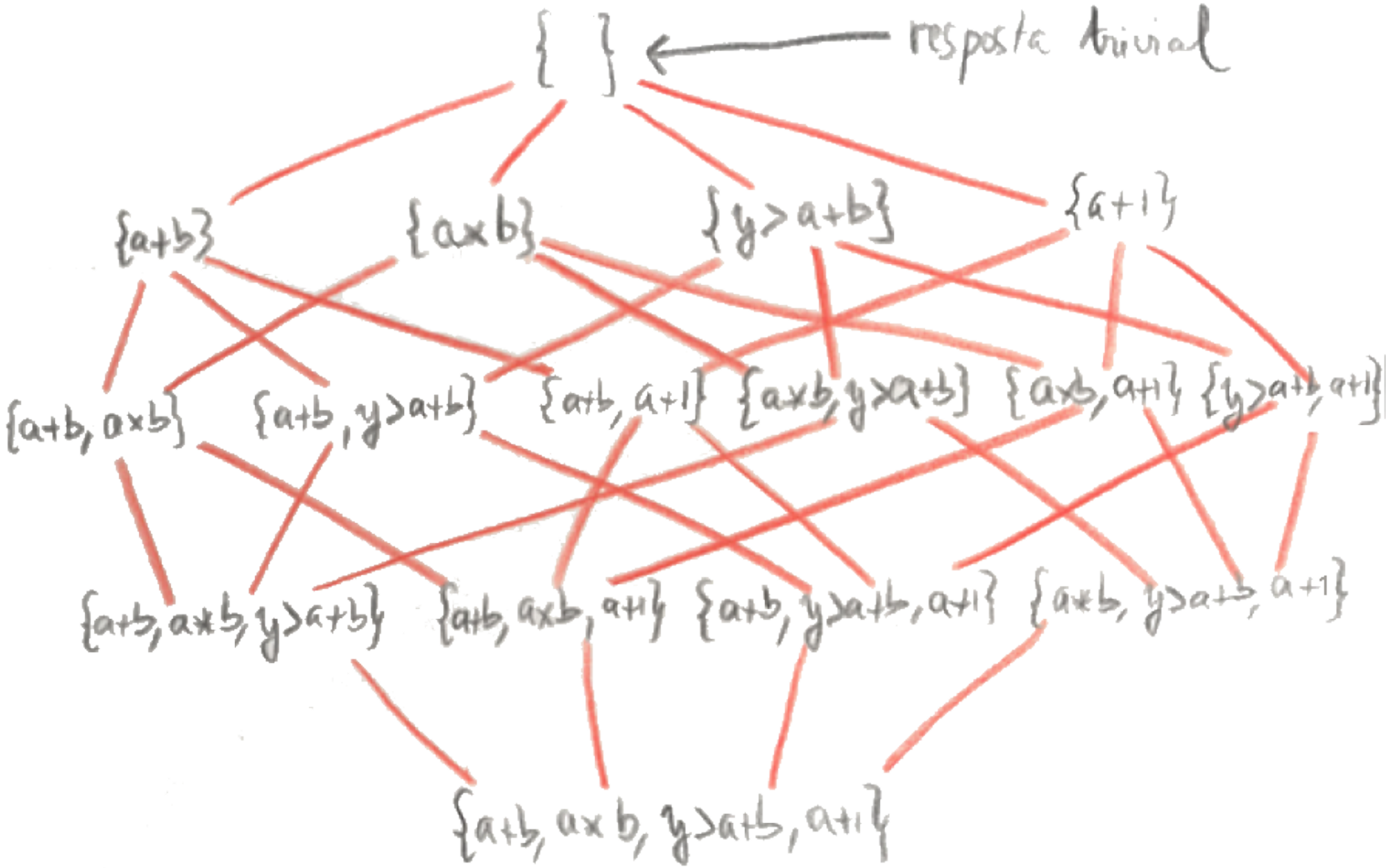
$$a + b,\ a \times b,\ y > a + b,\ a + 1$$

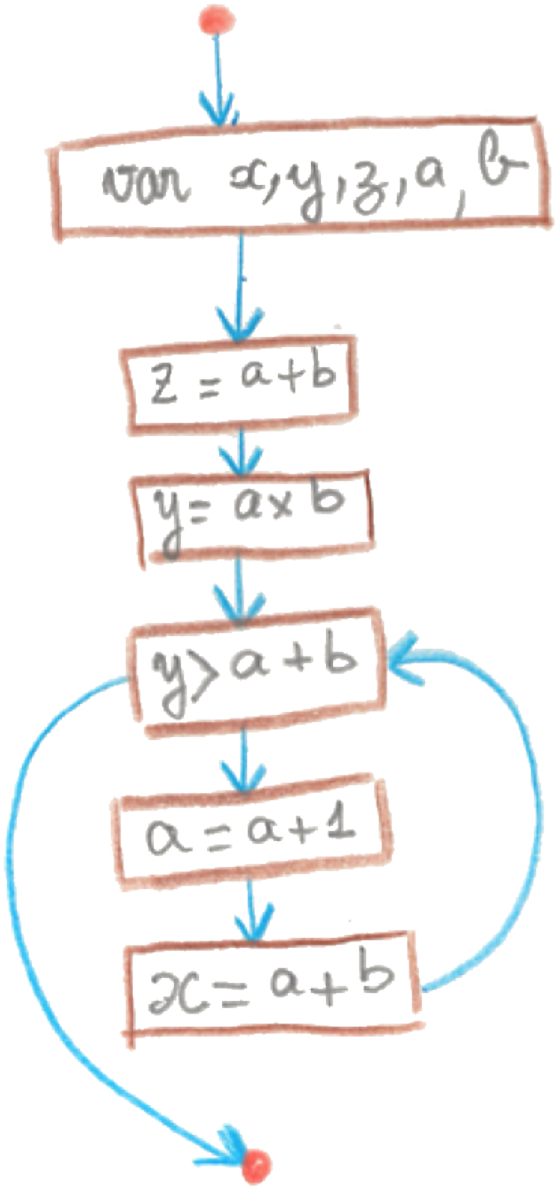
o reticulado por considerar é:

$$L = (\mathcal{P}(\{a + b, a \times b, y > a + b, a + 1\}), \supseteq)$$

o **reticulado subconjunto invertido**

O reticulado subconjunto invertido



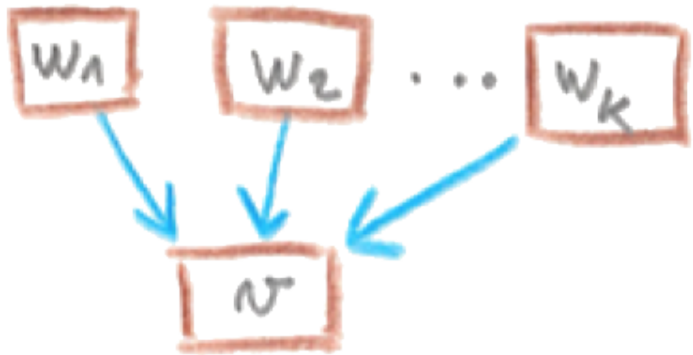

var x,y,z,a,b
z = a+b
y= a x b
y> a+b
a = a+1
x= a+b

para cada nodo V do CFG, temos a variável $[\![v]\!]$ definida como o conjunto das variáveis do programa que estão disponíveis no ponto do programa que **segue** v

os conjuntos calculados podem ser **demasiados pequenos**, pode isso optamos por uma análise conservadora que permitirá retirar informação relevante, mesmo nestes casos.

definição do *Join*:

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



a função $X \downarrow_x$ remove todas as expressões de X que contêm referências para variável x

a função $exps(E)$ é definida por recursão estrutural como

$$exps(E) = \begin{cases} \{E_1 \ op \ E_2\} \cup exps(E_1) \cup exps(E_2) & se \ E = E_1 \ op \ E_2 \\ & \text{sendo a sub-expressão} \\ & \texttt{input} \text{ excluída} \\ \emptyset & se \ E = intconst \ (\in \mathbb{N}) \\ \emptyset & se \ E = x \ (\in Vars) \\ \emptyset & se \ E = input \end{cases}$$

para o nodo de entrada

$$[\![entry]\!] = \emptyset$$

para as condições e *output*

$$[\![if\,(E)]\!] = [\![output\ E]\!] = Join(v) \cup exps(E)$$

para as atribuições

$$[\![x = E]\!] = (Join(v) \cup exps(E))\downarrow_x$$

para qualquer outro nodo v

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

Intuição: Uma expressão está disponível em v se está disponível a partir de todas as arestas de entrada ou é calculada precisamente em v. É no entanto preciso tomar em conta que uma atribuição pode alterar este estado

Exercício: Verifique que as restrições são monótonas

$$
\begin{array}{lcl}
[\![entry]\!] & = & \emptyset \\
[\![var\ x, y, z, a, b]\!] & = & [\![entry]\!] \\
[\![z = a + b]\!] & = & exps(a + b) \downarrow_z \\
[\![y = a \times b]\!] & = & ([\![z = a + b]\!] \cup exps(a \times b)) \downarrow_y \\
[\![y > a + b]\!] & = & ([\![y = a \times b]\!] \cap [\![x = a + b]\!]) \cup exps(y > a + b) \\
[\![a = a + 1]\!] & = & ([\![y > a + b]\!] \cup exps(a + 1)) \downarrow_a \\
[\![x = a + b]\!] & = & ([\![a = a + 1]\!] \cup exps(a + b)) \downarrow_x \\
[\![exit]\!] & = & [\![y > a + b]\!]
\end{array}
$$

$$
\begin{array}{lcl}
[\![entry]\!] & = & \emptyset \\
[\![var\ x, y, z, a, b]\!] & = & \emptyset \\
[\![z = a + b]\!] & = & \{a + b\} \\
[\![y = a \times b]\!] & = & \{a + b,\ a \times b\} \\
[\![y > a + b]\!] & = & \{a + b,\ y > a + b\} \\
[\![a = a + 1]\!] & = & \emptyset \\
[\![x = a + b]\!] & = & \{a + b\} \\
[\![exit]\!] & = & \{a + b\}
\end{array}
$$

temos várias respostas não triviais

notamos em particular que $a + b$ está disponível antes do ciclo

o programa pode assim ser (ligeiramente) optimizado da seguinte forma

```
var x,y,z,a,b, aplusb;
aplusb = a+b;
z=aplusb;
y = a*b;
while (y > aplusb) {
  a = a+1;
  aplusb = a+b;
  x = aplusb;
}
```

enquanto garantimos o respeito semântico do programa original

para um CFG com n nodos e k expressões não triviais, o reticulado tem uma altura de $n \times k \Longrightarrow$ limita o numero das iterações.

em cada iteração efectuamos $\mathcal{O}(n)$ operações (intersecções, uniões, igualdades) que tomam no global $\mathcal{O}(k.n)$, ou seja $\mathcal{O}(k^2.n^2)$

Análise de expressões atarefadas

Análise de expressões atarefadas

uma expressão (não trivial) é **atarefada** (*very busy*) se vai certamente ser avaliada novamente antes de ver o seu valor mudar

mais uma vez, temos de nos socorrer de uma aproximação para poder detectar tais casos
as aproximações que consideraremos aqui costumam serem prudente (e assim incluir poucas expressões)

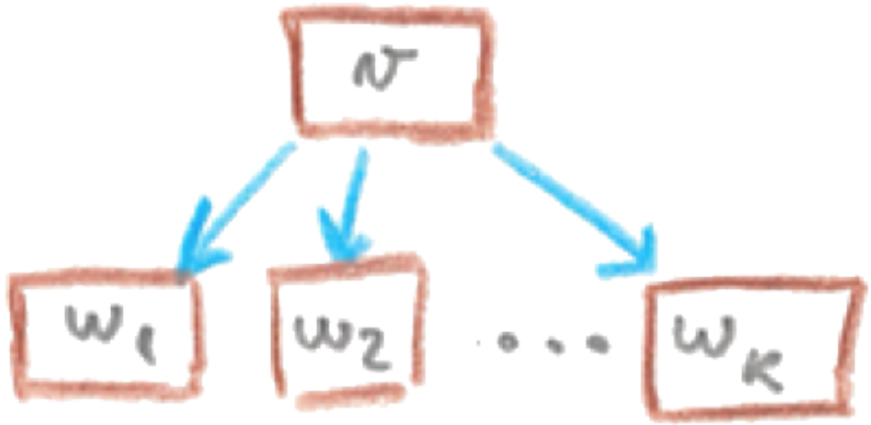
- as (eventualmente poucas) respostas "**atarefada**" devem ser certeiras
- as expressões atarefadas podem ser **pre-calculada**

boa nova: podemos reutilizar o reticulado da análise das expressões disponíveis

para cada vértice v do CFG temos a restrição $[\![v]\!]$ que consiste no subconjunto de variáveis do programa que estão atarefadas no vértice **anterior** a v.
como a análise é conservadora, o conjunto calculado pode ser de tamanho diminuto

definition do *Join*:

$$Join(v) = \bigcap_{w \in succ(v)} [\![w]\!]$$



no caso do vértice *v*=*exit*:

$$[\![exit]\!] = \emptyset$$

para as condicionais ($v = if\ (E)$)e a operação *output* ($v = output\ E$):

$$[\![if\,(E)]\!] = [\![output\ E]\!] = Join(v) \cup exps(E)$$

no caso das atribuições ($v = \ x = E$):

$$[\![x = E]\!] = Join(v) \downarrow_x \cup exps(E)$$

para todos os outros nodos:

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

Um exemplo

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
   output a*b-x;
   x = x-1;
}
output a*b;
```

a analise das expressões atarefadas destaca que $a \times b$ está atarefada dentro do ciclo

Desenvolva, à semelhança dos exemplos das análises previamente expostos,

- a definição das restrições para o exemplo aqui apresentado e
- o calculo do ponto fixo que resulta na solução apresentada ($a \times b$ **é expressão atarefada**)

Consequências e respectivo aproveitamento

o compilador pode, com base nesta analise, realizar uma optimização (designada de ***code hoisting***): puxar a avaliação da expressões atarefada para o ponto que antecede o primeiro local onde ficou determinada que está atarefada.

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
   output a*b-x;
   x = x-1;
}
output a*b;
```

$\Longrightarrow$

```
var x,a,b,atimesb;
x = input;
a = x-1;
b = x-2;
atimesb=a*b;
while (x>0) {
   output atimesb-x;
   x = x-1;
}
output atimesb;
```

Qual é a análise de complexidade no pior caso desta análise estática?

Análise do alcance de definições

Análise do alcance de definições: Princípios

a análise do alcance de definições (***reaching definitions analysis***) para um ponto de programa permite saber que atribuições definem neste ponto do programa os valores das variáveis.

o desafio da aproximação conservadora que vamos definir é aqui **não falhar nenhuma atribuições** que possa influenciar os valores das variáveis num determinado ponto do programa. Neste sentido, esta analise estática pode ter de considerar demasiadas possíveis atribuições.

o reticulado que vamos aqui usar é o reticulado construído sobre o **conjunto das partes do conjunto das atribuições** do programa em causa, ordenado pela **inclusão**.

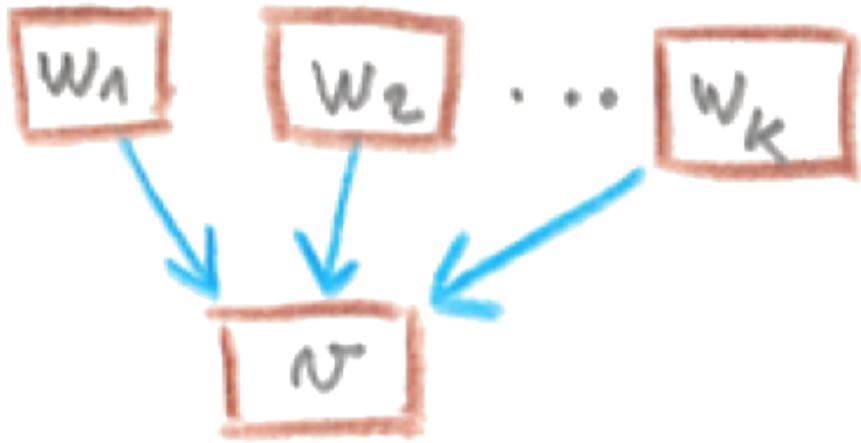
assim para o programa seguinte:

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

o reticulado é

$$L = (\mathcal{P}(\{x = input, y = x/2, x = x-y, z = x-4, x = x/2, z = z-1\}), \subseteq)$$

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



no caso das atribuições

$$[\![x = E]\!] = Join(v) \downarrow_x \cup \{x = E\}$$

para todos os outros nodos

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

onde a função $X \downarrow_x$ remove a atribuição x de X

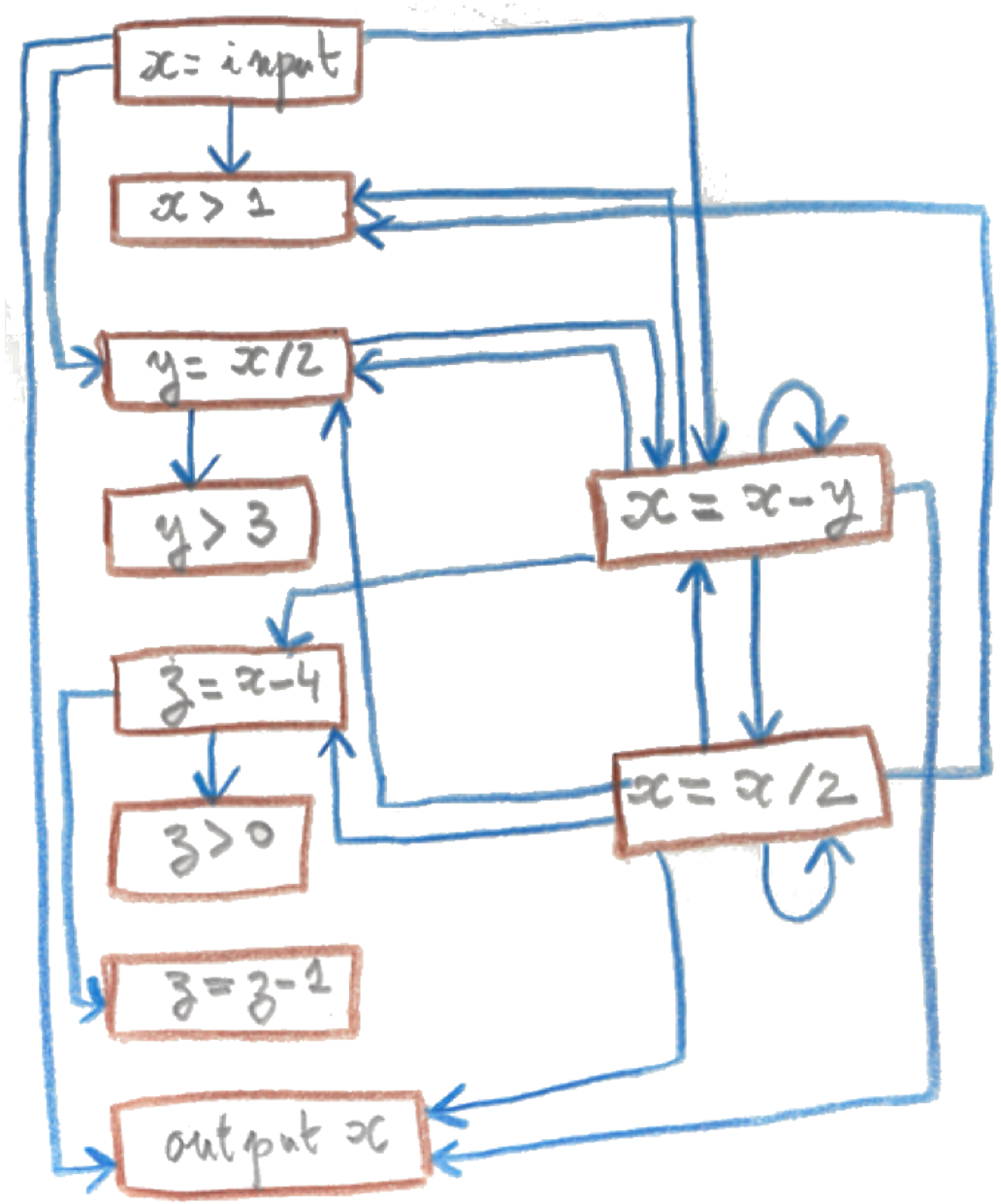
O grafo *def-use*

desta vez e para esta análise, o grafo que vamos usar é uma variante do CFG: o grafo ***def-use***
os vértices são os vértices do CFG

as arestas vão dos vértices de definições para vértices que possivelmente as usam

esta analise **forma a base de várias transformações**, como as eliminações de código morto, ou *code-motion*

O grafo *def-use*, um exemplo



Exercício

- mostre que o grafo *def-use* é sempre um sub-grafo do fecho transitivo do CFG
- conduza a definição do sistema de restrições e o calculo da menor solução desta análise sobre o exemplo introduzido
- que proveito podemos retirar da solução (explique o seu uso nas duas transformações citadas no acetato 44)?

Análise das variáveis inicializadas

Análise das variáveis inicializadas

objectivo caracterizar cada variável com a informação seguinte: se foi devidamente inicializada antes da sua leitura

é uma analise ***must* em processo**

em cada ponto do programa queremos saber as variáveis que **com toda a certeza** foram inicializadas **antes**

o reticulado de suporte é o reticulado inverso das partes do conjunto das variáveis do programa

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

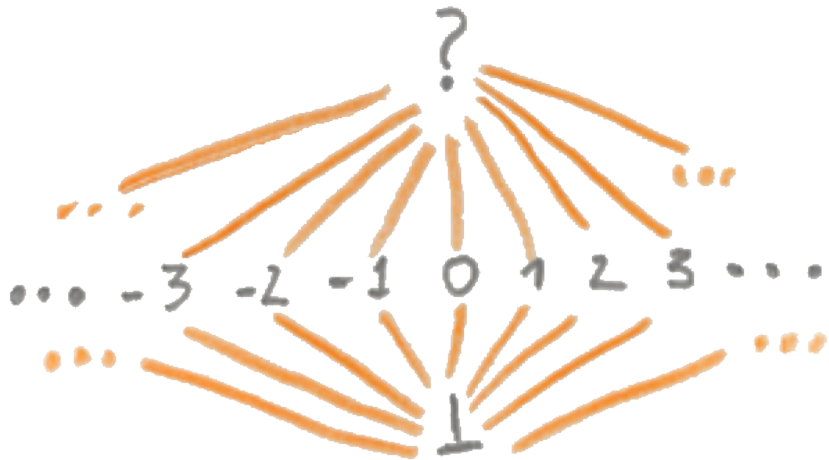
a definição $[\![v]\!]$ das restrições para cada nodo v é

$$[\![entry]\!] = \emptyset \qquad [\![x = E]\!] = Join(v) \cup \{x\}$$
$$[\![v]\!] = Join(v) \text{ para todos os outros nodos } v$$

Análise da propagação de constantes

objectivo. determinar em cada ponto de programas que variáveis tem um valor constante

é uma variante da analise de sinal mas sobre um reticulado da forma



$m \dot{+} n \triangleq n, m \mapsto \mathit{if}\ (n = \bot \vee m = \bot)\ \mathit{then}\ \{\bot\}\ \mathit{else}\ \mathit{if}\ (n = ? \vee m = ?)\ \mathit{then}\ \{?\}\ \mathit{else}\ \{n + m\}$

Exercício

- defina as operações abstractas em falta
- defina a função $Join(v)$ para esta análise e para os vértices v de um CFG
- defina a função $[\![v]\!]$ para cada tipo de nodo v do CFG
- verifique que esta função é monótona (relativamente ao reticulado em causa)
- defina o conjunto das restrições associadas ao exemplo apresentado no acetato 52
- execute o algoritmo de calculo de ponto fixo (escolhe entre as diferentes variantes apresentadas) e apresente os cálculos intermédios

Com base na solução desta análise, podemos transformar o primeiro programa no segundo programa
Por sua vez, com base numa analise *reaching definitions* e uma eliminação de código morto podemos reduzir o código na forma do terceiro programa

```
var x,y,z;
x = 27;
y = input;
z = 2*x+y;
if (x<0) {y = z-3;}
   else {y = 12;}
output y;
```

$\Longrightarrow$

```
var x,y,z;
x = 27;
y = input;
z = 54+y;
if (0) {y = z-3;}
   else {y = 12;}
output y;
```

$\Longrightarrow$

```
var y;
y = input;
output 12;
```

Análise de intervalo

uma análise de intervalo calcula para cada variável o **limite superior** e o **limite inferior** dos seus possíveis valores

pode ser útil para optimizações
exemplos: se uma variável é usada como índice de vector e o intervalo dos seus valores estarem dentro dos limites do vector, podemos abdicar da verificação sistemática do *array bound checking*

mas também: *numerical overflows*, optimizações ligadas à representação óptima de valores numéricos, etc.

o reticulado $L_{interval}$ dos intervalos define-se por

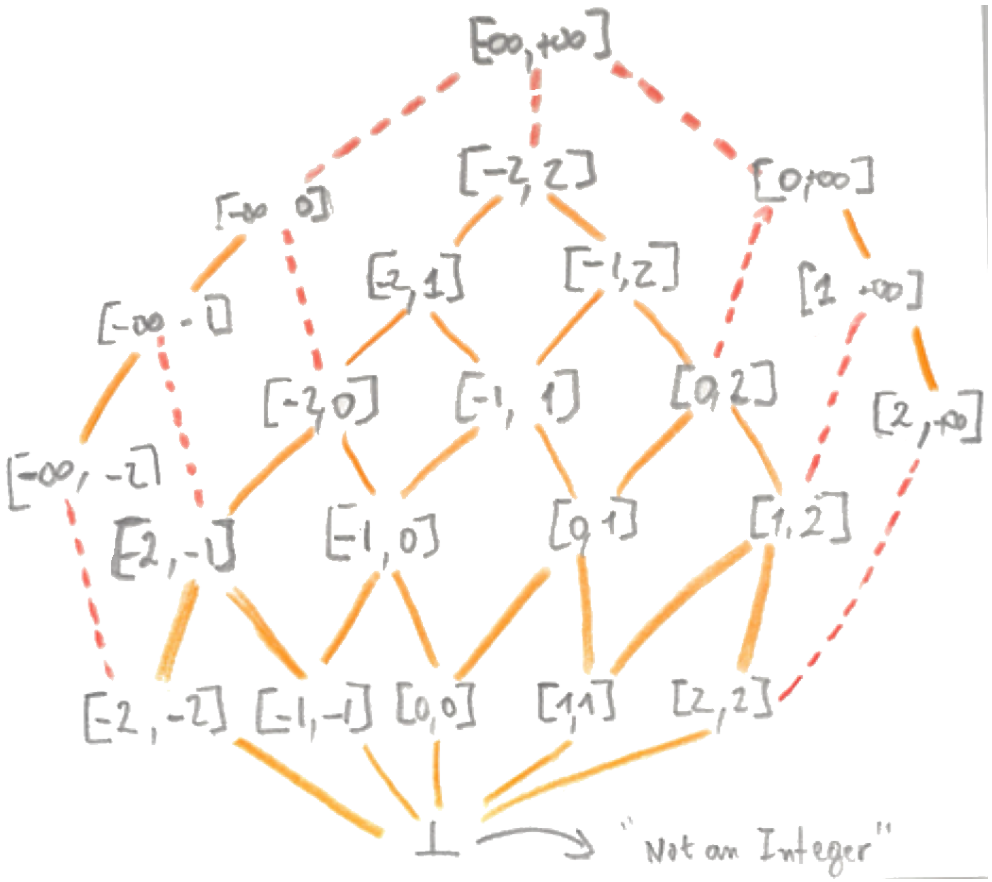
$$lift(\{[l, h] \mid l, h \in \mathbb{N} \wedge l \leq h\})$$

e

$$[l_1, h_1] \sqsubseteq [l_2, h_2] \triangleq l2 \leq l1 \wedge h_1 \leq h_2$$

∞ e $-\infty$ são considerados valores (não são uma abstração)

o reticulado tem uma altura infinita!! temos de adaptar o algoritmo de procura de ponto fixo.



o reticulado ***Interval***

o reticulado por contemplar para cada ponto do CFG é

$$L \triangleq \mathit{Vars} \to \mathit{Inverval}$$

que associa um intervalo a cada variável inteira

mas (mais uma vez...)

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \cdots [0,n] \cdots$$

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

para o nodo de entrada

$$[\![entry]\!] = \lambda x.\bot$$

para as atribuições

$$[\![x = E]\!] = Join(b)[x \rightarrow eval(Join(v), E)]$$

para os nodos restantes

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

a função $eval(\rho, E)$ é definida da seguinte forma:

$$eval(\rho, E) \triangleq \begin{cases} \rho(x) & se\ E = x, x \in Vars \\ \text{[i,i]} & se\ E = i, i \in \mathbb{N} \\ \\ \overline{op}(eval(\rho, E_1), eval(\rho, E_2)) & se\ E = E_1\ opE_2 \end{cases}$$

onde

$$\overline{op}([l_1, h_1], [l_2, h_2]) = [(min_{x \in [l_1, h_1], y \in [l_2, h_2]} x\ op\ y),\ (max_{x \in [l_1, h_1], y \in [l_2, h_2]} x\ op\ y)]$$

exemplo

$$\overline{+}([1, 10], [-7, 7]) = [1 - 5, 10 + 7] = [-4, 17]$$

como anunciado, o algoritmo de procura de ponto fixo tem de ser adaptado aos casos dos reticulados de altura infinita
porque em L^n, a sequência de aproximações $F_i(\bot, \bot, \ldots, \bot)$ pode não convergir

restringir a representação dos inteiros a 32 bits não é uma solução prática

Exercício

1. mostre que estas definições definam operadores monótonos sobre o reticulado dos intervalos
2. mostre que os operadores abstractos de comparação podem ser definidos de uma forma mais precisa sem romper as propriedades esperadas deles
3. dê um exemplo de programa sobre o qual, em L^n, a sequência de aproximações $F_i(\bot, \bot, \ldots, \bot)$ não converge

As técnicas de *widening* e de *narrowing*

esta técnica permite contornar em parte a dificuldade apontada

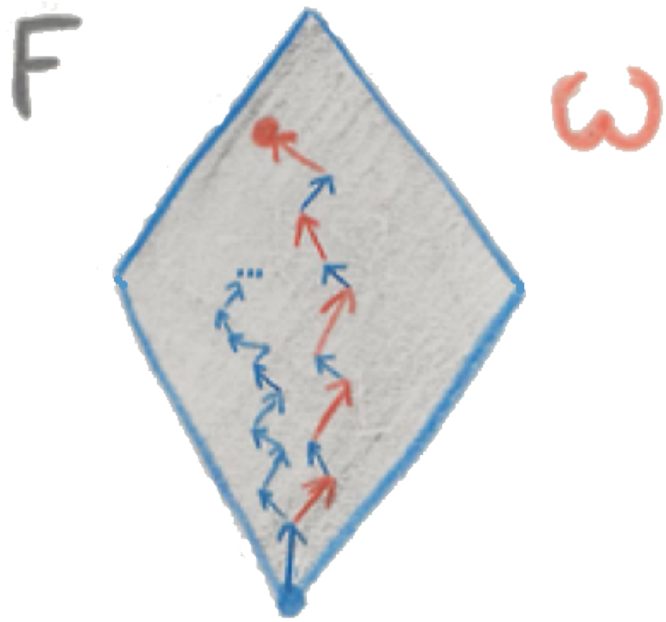
introduzimos para esse efeito uma **função de alargamento** (*widening function*)
$\omega : L^n \to L^n$ tal que

$$(\omega \circ F)^i(\bot, \bot, \bot, \cdots, \bot)$$

convirja para um ponto fixo que é uma **aproximação segura** (*safe*) de cada
$F^i(\bot, \bot, \bot, \cdots, \bot)$

ou seja, a função ω *exagera* a informação que manipula de forma suficiente para garantir terminação

iterações em modo *Turbo*



a função ω é definida ponto a ponto sobre L^n
é parametrizada com um subconjunto B finito fixo de $\mathbb{N}$ (i.e. $B \subset \mathbb{N}$)
este deve conter ∞ e $-\infty$ (para poder contar com $\top$)
tipicamente este conjunto é definido com base em todas as constantes que ocorrem no programa analizado (outras heurísticas existam)

ideia: encontrar o intervalo envolvente *mais justo/apertado* **permitido** neste caso dos intervalos:

$$\omega(I) = \begin{cases} [max\{i \in B \mid i \leq a\},\ min\{i \in B \mid b \leq i\}] & se\ I = [a, b] \\ \bot & se\ I = \bot \end{cases}$$

Exercício

mostre que ω é uma função extensiva e monótona

mostre que $\omega(\mathit{Interval})$ é um reticulado de altura finita

tendo em conta as propriedades anteriores de ω, mostre que a técnica de alargamento funciona garantidamente (na procura de um ponto fixo que é uma aproximação *safe*)

vamos em parte responder ao exercício anterior.
a correcção da técnica do alargamento depende fortemente de

- ω é uma função **extensiva** e **monótona** (w.r.t o reticulado em causa)
- $\omega(\mathit{Interval})$ é um reticulado de altura finita

assim, **safety**:

$$\forall i, F^i(\bot, \bot, \bot, \cdot, \bot) \sqsubseteq (\omega \circ F)^i(\bot, \bot, \bot, \cdot, \bot)$$

visto F ser monótono e ω ser extensiva

assim $\omega \circ F$ é uma função monótona de $\omega(\mathit{interval}) \rightarrow \omega(\mathit{Interval})$, logo o ponto fixo existe.

basta aplicar ω nas arestas de retorno (**back-edge**) do CFG

a técnica de alargamento em geral encontra uma solução demasiada aproximada (*aponta demasiado para cima*).

Dar marcha à ré

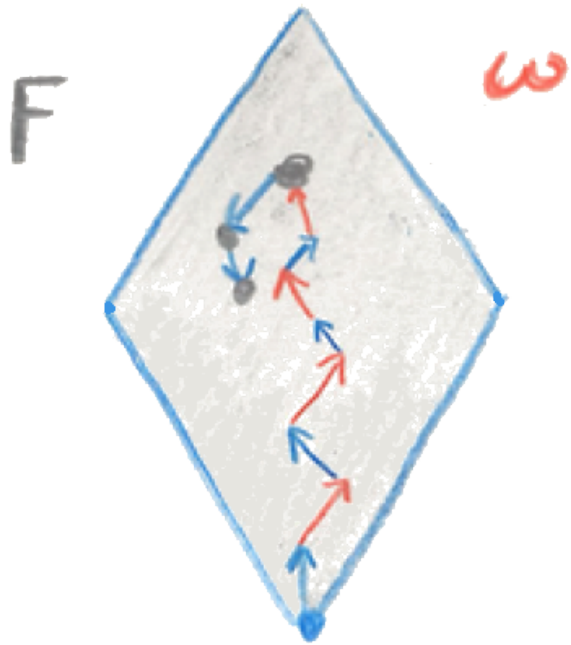
a técnica do estreitamento tenta minorar este efeito

se definirmos *fix* e *fix*ω como

$$fix \triangleq \sqcup F^i(\bot, \bot, \cdots, \bot)$$

$$fix\omega \triangleq \sqcup(\omega \circ F)^i(\bot, \bot, \cdots, \bot)$$

então $fix \sqsubseteq fix\omega$
mas temos também $fix \sqsubseteq F(fix\omega) \sqsubseteq fix\omega$



assim, aplicar F repetidamente **melhora** o resultado (que **permanece** *safe*)

podemos iterar as vezes que quisermos. Pode divergir, mas é seguro parar em qualquer altura

mostre que $\forall i, fix \sqsubseteq F^{i+1}(fix\omega) \sqsubseteq F^{i}(fix\omega) \sqsubseteq fix\omega$

tendo em conta que ω é expansivo temos

$$F(fix\omega) \sqsubseteq \omega(F(fix\omega)) = (\omega \circ F)(fix\omega) = fix\omega$$

- por indução temos que para cada i $F^{i+1}(fix\omega) \sqsubseteq F^i(fix\omega) \sqsubseteq fix\omega$
- i.e. $F^{i+1}(fix\omega)$ é pelo menos tão preciso quanto $F^i(fix\omega)$

$fix \sqsubseteq fix\omega$ consequentemente $F(fix) = fix \sqsubseteq F(fix\omega)$, por monotonia de F

- por indução sobre i temos igualmente $fix \sqsubseteq F^i(fix\omega)$
- i.e. $F^i(fix\omega)$ é uma aproximação segura de fix

Divergência em ação

```
y = 0;
x = 7;
x = x+1;
while (input) {
  x = 7;
  x = x+1;
  y = y+1;
} //  Neste ponto do programa
```

$$
\begin{array}{ll}
[x \to \bot & , y \to \bot] \\
[x \to [8,8] & , y \to [0,1]] \\
[x \to [8,8] & , y \to [0,2]] \\
[x \to [8,8] & , y \to [0,3]] \\
\ldots &
\end{array}
$$

Alargamento em ação

```
y = 0;
x = 7;
x = x+1;
while (input) {
  x = 7;
  x = x+1;
  y = y+1;
} //  Neste ponto do programa
```

$$
\begin{array}{ll}
[x \to \bot & , y \to \bot] \\
[x \to [7, \infty] & , y \to [0, 1]] \\
[x \to [7, \infty] & , y \to [0, 7]] \\
[x \to [7, \infty] & , y \to [0, \infty]] \\
\ldots &
\end{array}
$$

$com \quad B = \{-\infty, 0, 1, 7, \infty\}$

Estreitamento em acção

```
y = 0;
x = 7;
x = x+1;
while (input) {
  x = 7;
  x = x+1;
  y = y+1;
} //  Neste ponto do programa
```

$$[x \to \bot \quad , y \to \bot]$$
$$[x \to [7, \infty] \quad , y \to [0, 1]]$$
$$[x \to [7, \infty] \quad , y \to [0, 7]]$$

$$[x \to [8, 8] \quad , y \to [0, \infty]]$$

$com \quad B = \{-\infty, 0, 1, 7, \infty\}$

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*!).