Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 13 - Análise de Programas e a *Framework Monótona*

Da necessidade e natureza da analise estática de programa

Questões sobre programas

- Será que o programa termina?
- Qual é o tamanho da *heap* necessária para a execução de um programa?
- Quais são os *output* possíveis?
- etc.

```
foo(p,x) {
  var f,q;
  if (*p≡0) { f=1; }
  else {
    q = malloc;
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
   }
  return f;
}
```

$\Longleftarrow$ qualquer ponto do programa

$=$

conjunto dos valores para o *Program Counter*

Invariante:

Uma propriedade que é válida num determinado ponto de programa qualquer que seja o estado que a execução produza quando chega a este ponto

Questões sobre pontos do programa

- será o valor de x lido em passos seguintes da execução ?
- poderá ser o apontador p o apontador *null* ?
- para que variáveis p pode apontar ?
- estará x inicializada antes da sua leitura ?
- qual é o intervalo de valores que a variável x pode tomar ?
- em que pontos do programa x pode tomar o seu valor actual ?
- Poderão ser p e q apontadores para zonas distintas da *heap*?
- Poderá um determinado *assert* falhar ?

Qual interesse das respostas ?

- **Ter garantias de correção**:
  - verificar o comportamento
  - apanhar os bugs **o mais cedo possível**
  - descobrir falhas de segurança

- **Aumentar a eficiência do código**:
  - guiar/alavancar os processos de optimização de código com informação comportamental
  - tirar proveito da informação sobre o uso de recursos necessários a execução

E que tal os testes ou a simulação?

"Program testing can be used to show the presence of bugs, but never to show their absence."

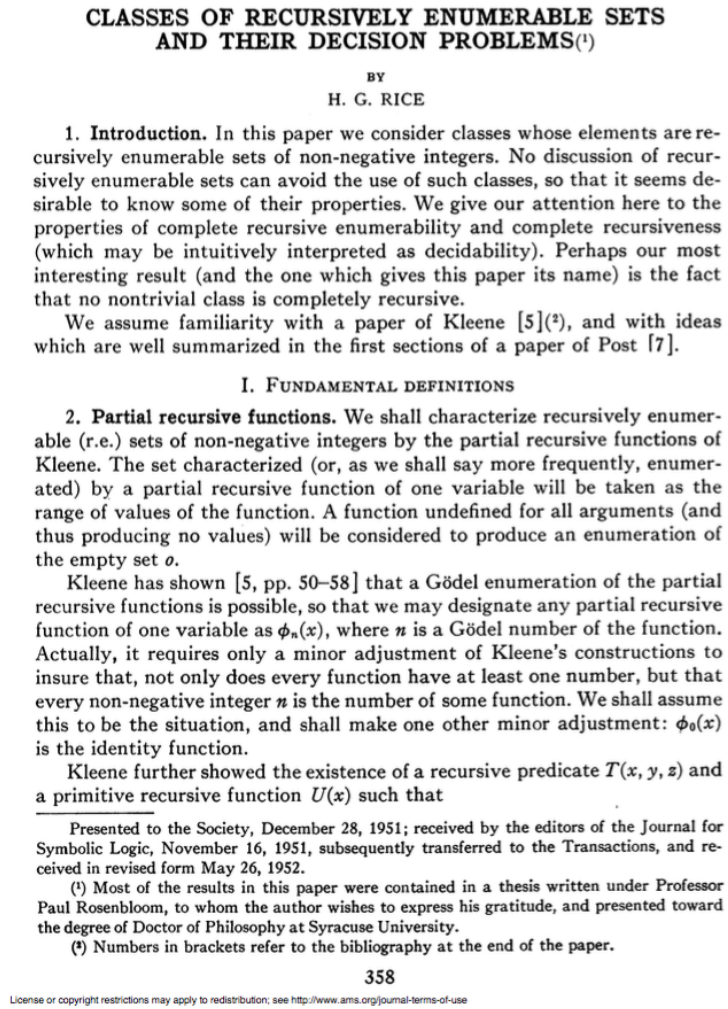
[Dijkstra, 1972]

- Os testes tomam "as vezes" 50% do tempo / custo do desenvolvimento
- Os testes podem ser complicados. Pense por exemplo testes a *drivers*. É preciso dispor dos dispositivos!
- Erros de concorrência são difíceis de (re)produzir com testes.

Corollary B. There are no nontrivial c.r. classes by the strong definition.

[Rice, 1953]

traduzido em termos leigos: determinar uma qualquer propriedade não trivial do comportamento de um programa escrito numa linguagem *Turing-Complete* não é (um problema) decidível.



CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS(1)

BY

H. G. RICE

1. **Introduction.** In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact that no nontrivial class is completely recursive.

We assume familiarity with a paper of Kleene [5](2), and with ideas which are well summarized in the first sections of a paper of Post [7].

I. FUNDAMENTAL DEFINITIONS

2. **Partial recursive functions.** We shall characterize recursively enumerable (r.e.) sets of non-negative integers by the partial recursive functions of Kleene. The set characterized (or, as we shall say more frequently, enumerated) by a partial recursive function of one variable will be taken as the range of values of the function. A function undefined for all arguments (and thus producing no values) will be considered to produce an enumeration of the empty set o.

Kleene has shown [5, pp. 50–58] that a Gödel enumeration of the partial recursive functions is possible, so that we may designate any partial recursive function of one variable as $\phi_n(x)$, where n is a Gödel number of the function. Actually, it requires only a minor adjustment of Kleene's constructions to insure that, not only does every function have at least one number, but that every non-negative integer n is the number of some function. We shall assume this to be the situation, and shall make one other minor adjustment: $\phi_0(x)$ is the identity function.

Kleene further showed the existence of a recursive predicate $T(x, y, z)$ and a primitive recursive function $U(x)$ such that

Presented to the Society, December 28, 1951; received by the editors of the Journal for Symbolic Logic, November 16, 1951, subsequently transferred to the Transactions, and received in revised form May 26, 1952.

(1) Most of the results in this paper were contained in a thesis written under Professor Paul Rosenbloom, to whom the author wishes to express his gratitude, and presented toward the degree of Doctor of Philosophy at Syracuse University.

(2) Numbers in brackets refer to the bibliography at the end of the paper.

358

Por redução ao problema da paragem.

Poderemos decidir se uma variável tem um valor constante?

```
x = 17; if (TM(j)) x = 18;
```

Aqui, x é constante se e só se a j-ésima máquina de Turing não pára sobre o *input* vazio.

Aproximações!

propor respostas aproximadas pode ser decidível.

Que aproximações são interessante? As aproximações **conservadoras**!

- i.e. que se enganam eventualmente, de forma defensiva - por exemplo propor uma recusa quando, na verdade, não há problema.
- desafio: propor aproximações que se enganam somente desta forma

Os problemas para os quais vamos propor aproximações são problema de decisão (resposta binária: sim/não).

Há processos de aproximações, outros do que os de decisão, que são interessantes: comportamento w.r.t a memória, determinar os locais onde apontam os apontadores, etc.

Natureza das aproximações

SLOGAN

ERRAR DO LADO SEGURO

TROCAR PRECISÃO COM EFICIÊNCIA

Exemplos de aproximações

decidir se uma determinada função pode ou não ser chamada em tempo de execução

- caso negativo: remover a função do código (i.e. *dead code*)
- caso positivo: nada por fazer
- aspecto conservador: as respostas negativas são as respostas que temos de garantir como certas

decidir se um *cast* `(A) x` resultará sempre

- caso positivo: não produzir, aquando do processo de compilação, o código correspondente ao teste
- caso negativo: produzir o código que corresponde ao *cast*
- aspecto conservador: as respostas positivas são as respostas que temos de garantir como certas

Para além dos problemas de decisão binária

- Que variáveis podem ser o alvo de um determinado apontador (digamos `p`)?
- A forma da resposta depende do objectivo
- Se queremos evitar uma dereferenciação (substituir `*p` por `x`, para um dado `x`) :
  - responder `&x` se é garantido que é o único alvo de `p`
  - responder `?` nos outros casos
- Se queremos saber o tamanho máximo de `*p` :
  - uma possível resposta : `{&x,&y,&z}`
  - se a resposta for imprecisa demais, pode indicar demasiados possíveis alvos: resposta pouco útil.

Um desafio de engenharia

- um algoritmo de aproximação embora correcto pode simplesmente devolver uma resposta sistematicamente pouco útil (demasiada imprecisa)
- o **desafio de engenharia** nesta área é conceber algoritmo que consegue dar respostas úteis com uma frequência útil para alimentar a aplicação cliente
- ... e isso num tempo razoável e com recursos (memória) em tamanho razoável
- é a parte complicada (mas muito... gratificante) da analise estática de programas!

Bugs?... Bugs!!

```
int main() {
  char *p,*q;
  p = NULL; printf("%s",p);
  q = (char *)malloc(100); p = q;
  free(q);
  *p = 'x';
  free(p);
  p = (char *)malloc(100);
  p = (char *)malloc(100); q = p;
  strcat(p,q);
}
```

resultado de :

gcc -Wall foo.c
lint foo.c

$\Longrightarrow$ **Nada a apontar!**

Exercício

Descreva todos os erros que encontra no programa anterior

Outros desafios

saber lidar com

- funções, métodos, funções de ordem superior
- Estruturas mutáveis, objectos, apontadores, vectores
- inteiros (máquina), cálculos com números flutuantes
- *Dynamic dispatching*
- herança e subtipagem
- exceções
- recurso a mecanismos de reflexão
- etc.

TIP

introduzimos uma linguagem de programação minimal que dispõe de todas as construções programáticas relevantes para exemplificar de forma realista os algoritmos de analise estática

designamo-la de TIP (*Tiny Imperative Programming Language*). Trata-se de um mini-C

```
E -> int
    | id
    | E+E | E-E | E*E | E/E | E>E | E==E
    | (E)
    | input
```

- o comando `input` lê um inteiro do *buffer* de entrada
- os operadores de comparação devolvem 0 (falso) ou 1 (verdade)
- `id` representa um identificador

```
S -> id = E;
     | output E;
     | S S
     | skip
     | if (E) {S} [else {S}]?
     | while (E) {S}
```

- numa instrução condicional, o `else` é opcional
- a instrução `output` escreve um inteiro no *buffer* de saída.

```
F ->  id (id ,..., id) { [var id,...,id;]?  S  return E; }
```

- uma função pode ter um número qualquer de parâmetros e retorna um unico valor
- o bloco de instruções `var` introduz um conjunto de variáveis não inicializadas
- as chamadas a funções aumentam o conjunto das expressões possíveis

```
E ->  id ( E,...,E )
```

aumentamos as expressões com operadores de alocação dinámica de memória

```
E -> &id
     | malloc
     | *E
     | null
```

as instruções são aumentadas com operações de dereferenciação de apontadores

```
S ->  *id = E
```

não autorizamos aritmética explícita de apontadores

Apontadores de funções

os identificadores de função denotam apontadores para funções

assim, generalizamos as chamadas a funções por forma a poder dereferenciar apontadores de funções

```
E ->  (E)( E , ... , E)
```

apontadores de funções podem ser considerados aqui como modelos simples para os conceitos de objectos, ou funções de ordem superior

um programa é uma coleção de funções.

a função final é considerada como o ponto de entrada da execução

- os seus argumentos são retirados do *buffer de entrada*
- o seu resultado é colocado no *buffer de saída*

assumimos que todos os identificadores declarados são únicos

```
P ->  F ... F
```

Exemplo: factorial

Versão iterativa

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
        f = f*n;
        n = n-1;
  }
  return f;
}
```

Versão recursiva

```
rec(n) {
    var f;
    if (n==0) {
      f=1;
       }
    else {
      f=n*rec(n-1);
    }
    return f;
}
```

Outro exemplo

```
foo(p,x) {
  var f,q;
  if (*p≡0) {
    f=1;
  }
  else {
    q = malloc;
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
  }
  return f;
}
```

```
main() {
  var n;
  n = input;
  return foo(&n,foo);
}
```

Poderíamos também considerar

- variáveis globais
- estruturas
- objectos
- funções aninhadas
- etc.

Grafos de Fluxo de Controlo

um grafo de fluxo de controlo (CFG) é um grafo dirigido

- os vértices correspondem aos pontos de programa (ambos antes e depois de instruções)
- arestas representam possíveis fluxos do controlo

um CFG tem sempre

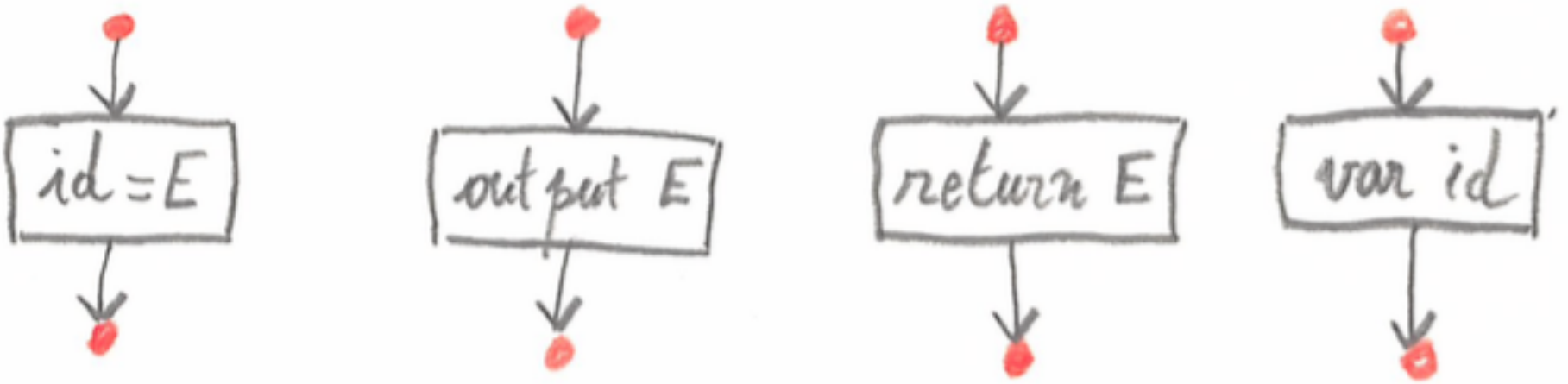
- um ponto de **entrada**
- um ponto de **saída**

seja v um vértice de um CFG.

- $pred(v)$ é o conjunto dos nodos que o antecedem
- $succ(v)$ é o conjunto dos nodos que sucedem

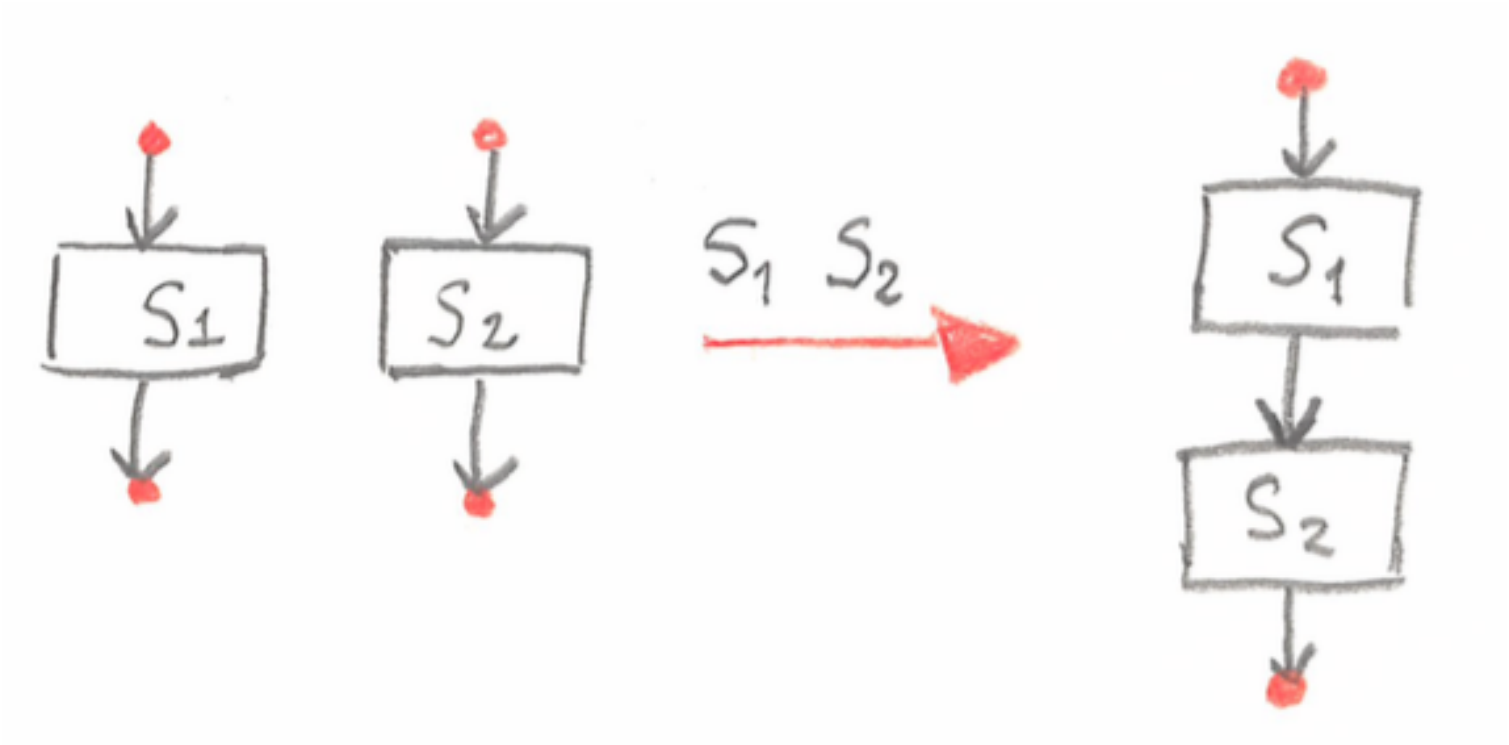
os CFGs são construidos por indução

Base. Os CFGs para as instrução de base são:

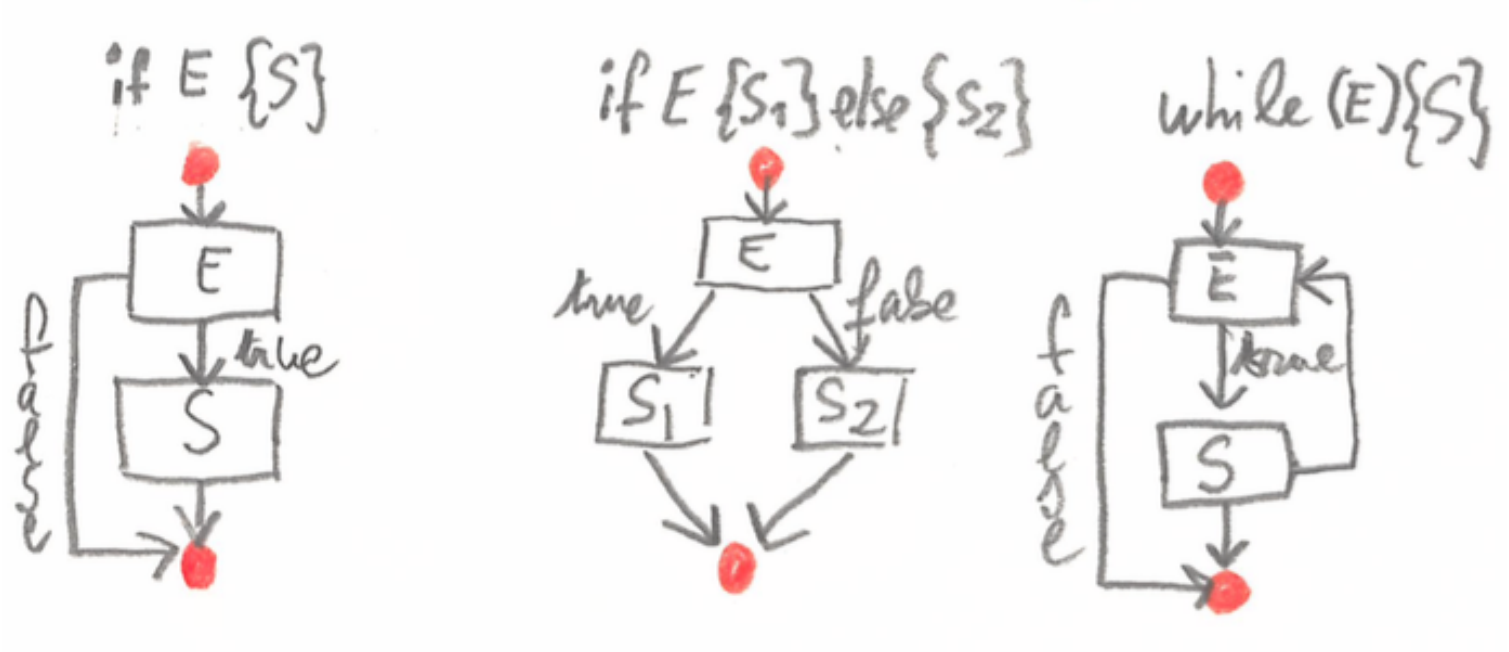


para a sequência S_1 S_2

- eliminar o nodo de saída do CFG de S_1, e o nodo de entrada de S_2
- concatenar o primeiro grafo por cima do segundo

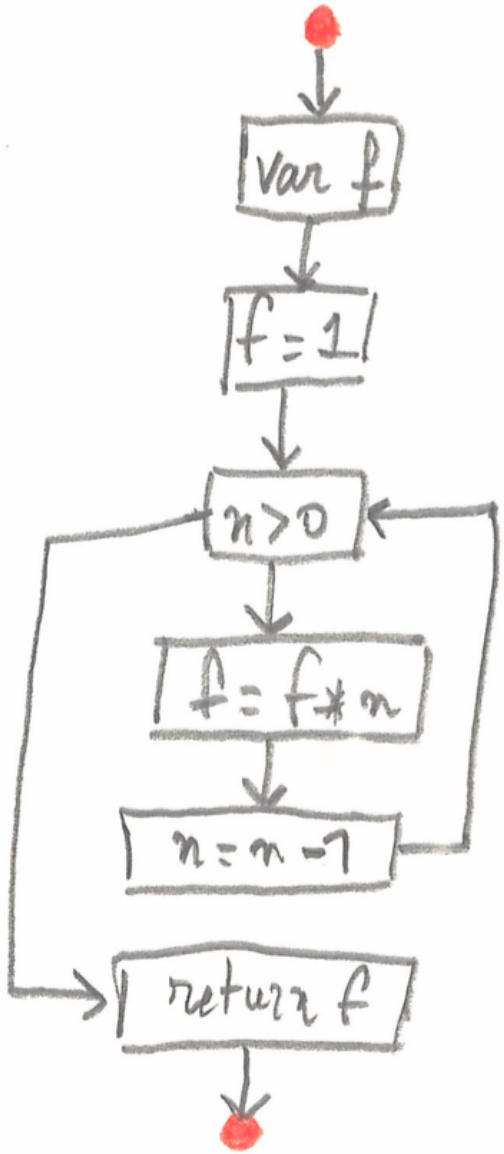


de forma semelhante, para as estruturas de controlo:



Um exemplo de CFG

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```



Reticulados e pontos fixos

Fluxo da informação e o seu controlo

vimos em aulas anteriores que os sistemas de tipos são mecanismos ricos que permitam um controlo fino de **muitas propriedades**, que vão muito para além da noção de tipos de dados (segurança, controlo da concorrência, controlo da utilização de recursos etc.)

mas os sistemas de tipos não são em geral sensíveis ao fluxo da informação gerada por um programa, e.g.

- as instruções podem ser trocadas entre elas sem alterar a tipagem
- as restrições de tipagem resultam da exploração da árvore de sintaxe abstracta e não do CFG

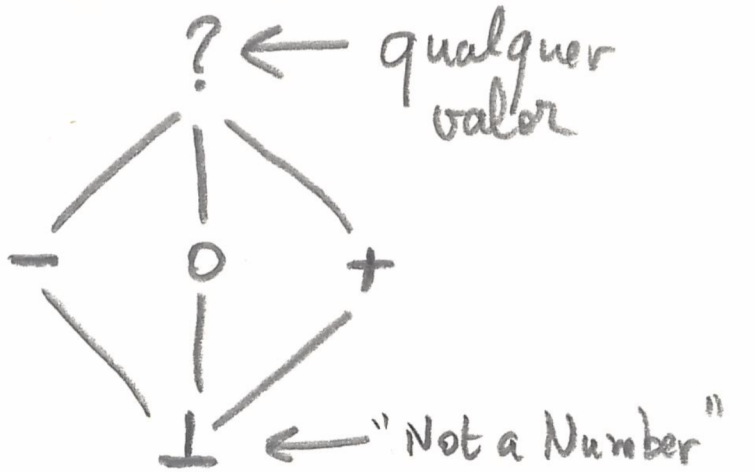
mas para algumas propriedades, precisamos de tomar em conta este fluxo

- as instruções quando trocadas devem alterar a análise
- as restrições resultantes devem ser extraídas da exploração do CFG

vamos introduzir aqui os fundamentos e as frameworks necessárias para tais analises, de **fluxo de informação**

o desafio é determinar de forma estática o sinal $(-, 0, +)$ de todas as expressões contidas num programa

para tal usamo o **reticulado dos sinais** (*Sign Lattice*)

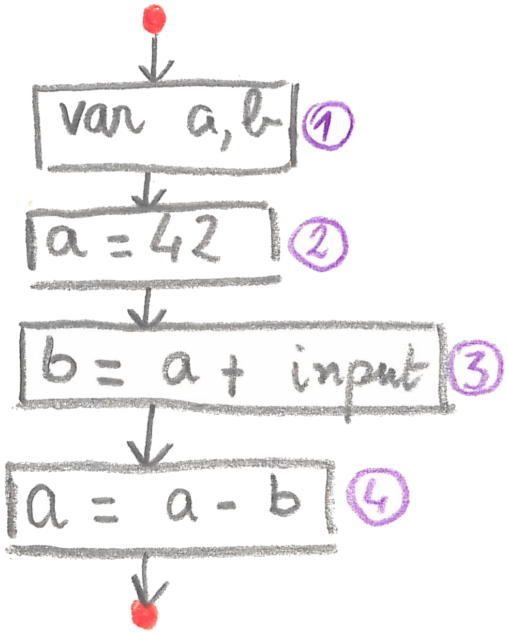


(*o vocabulário e os métodos usados aqui serão definidos com rigor mais adiante - trata-se aqui de dar uma ideia dos conceitos envolvidos)*

os estados da memória que nos interessam aqui são modelados como mapas $Vars \rightarrow Sign$ onde Var é o conjunto das variáveis do programa

Interessa-nos olhar para o estado das variáveis em cada ponto do programa (antes ou depois de cada vértice do CFG)

```
1 : var a,b;
2 : a =42;
3 : b = a + input;
4 : a = a - b;
```



$$\Longrightarrow \quad \begin{array}{lcl} x_1 & = & [a \to ?, b \to ?] \\ x_2 & = & x_1[a \to +] \\ x_3 & = & x_2[b \to x_2(a) + ?] \\ x_4 & = & x_3[a \to x_3(a) - x_3(b)] \end{array}$$

Conjunto de restrições sobre os sinais

a notação $[\![v]\!]$ denota um mapa que devolve o sinal para cada variável do programa após o vértice v do CFG corespondente ($[\![.]\!] : \mathit{Vars} \to \mathit{Sign}$)

para o vértice de entrada de uma função com parâmetros $p_1, \cdots, p_n$ temos

$$[\![v]\!] = [p_1 \to ?, \cdots, p_n \to ?]$$

para as declarações de variáveis, temos

$$[\![var\ x_1, \cdots, x_n]\!] = \mathit{JOIN}(v)[x_1 \to ?, \cdots, x_n \to ?]$$

para as atribuições, temos

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

para os outros vértices, temos

$$[\![v]\!] = \mathit{JOIN}(v)$$

onde

$$\mathit{JOIN}(v) = \bigsqcup_{w \in \mathit{pred}(v)} [\![w]\!]$$

para completar, falta-nos definir a função
$eval : (Vars \to Sign) \to E \to Sign$. trata-se de uma função de avaliação sobre o domínio dos sinais

$$eval(\rho, x) = \begin{cases} \rho(x) & se\ x \in Vars \\ sign(x) & se\ x\ constante\ inteira \\ \overline{op}(eval(\rho, E_1), eval(\rho, E_2)) & se\ x = E_1\ op\ E_2 \end{cases}$$

onde:
$\rho : Vars \to Sign$ designa um estado abstracto (o valor das variáveis em termos de sinais, e não de valores inteiros)

$Sign : E \to Sign$ devolve o sinal associado a um inteiro

$\overline{op}$ designa a extensão da operação *op* no domínio *Sign* dos sinais como apresentado no acetato seguinte

Operações abstratas

+	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	?
-	⊥	-	-	?	?
+	⊥	+	?	+	?
?	⊥	?	?	?	?

-	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	?
-	⊥	-	?	-	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

*	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	0	0	0
-	⊥	0	+	-	?
+	⊥	0	-	+	?
?	⊥	0	?	?	?

/	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	?	0	0	?
-	⊥	?	?	?	?
+	⊥	?	?	?	?
?	⊥	?	?	?	?

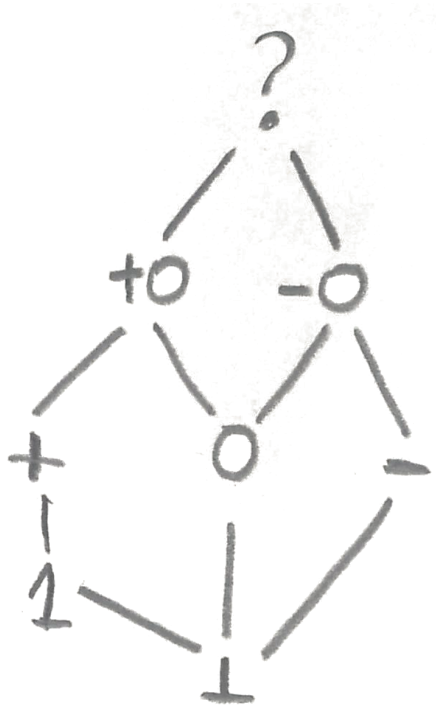
>	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	0	?
-	⊥	0	?	0	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

==	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	+	0	0	?
-	⊥	0	?	0	?
+	⊥	0	0	?	?
?	⊥	?	?	?	?

temos neste esquema alguma perda desnecessária de precisão. Podemos fazer melhor!

- $(2 > 0) == 1$ é avaliado em ?
- $+/+$ é avaliado em ?, (visto que $\frac{1}{2}$ fica arredondado para baixo, i.e. 0)

podemos usar um reticulado mais informativo



Os operadores são agora definidos
custa de uma tabela de 8 por 8

Exercício

- argumente que as tabelas de operação do acetato 41 são as mais precisas possíveis tento en conta o reticulado dos sinais original e tendo em conta a desejada propriedade de inocuidade (**soudness**);
- produza o sistema de equações para o exemplo de programa dado e resolva o sistema com base num dos algoritmos apresentados nesta aula;
- defina os seis operadores com base neste novo reticulado (acetato 42). Vertifique que aas operações propostas são todas monótonas.

Teoria das Ordens, Reticulados e Pontos Fixos
um resumo

dado um conjunto S, uma **ordem parcial** $\sqsubseteq$ é uma relação binária sobre S que satisfaz:

- reflexividade. $\forall x \in S, x \sqsubseteq x$
- transitividade.
  $\forall x, y, z \in S, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$
- anti-simetria. $\forall x, y \in S, x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$

uma **ordem estrita** é uma relação $\sqsubset$ binária sobre S que é transitiva e irreflexiva ($\forall x \in S, x \not\sqsubset x$). Por definicão, não é uma ordem parcial.

uma ordem (parcial ou estrita) R é **total** quando $\forall x, y \in S, x \ R \ y \ \vee \ y \ R \ x$

$(S, \sqsubseteq)$ é designado de **conjunto ordenado** quando $\sqsubseteq$ é uma ordem sobre S

$(S, \sqsubseteq)$ pode ser representada de forma elegante por um diagrama de Hasse se for finito

Emmy Noether
(1882-1935)

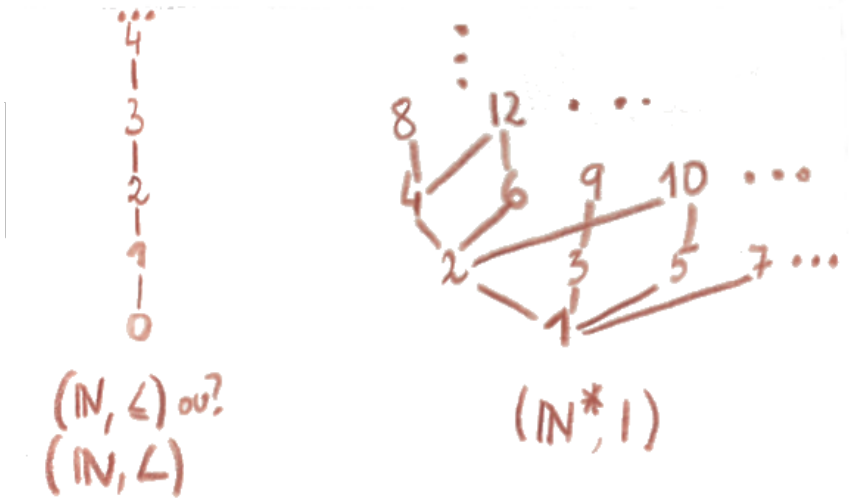
Alguns factos

$(\mathbb{N}, \leq)$ é um conjunto ordenado por uma ordem parcial total

$(\mathbb{N}, <)$ é um conjunto ordenado por uma ordem estrita total

Seja C um conjunto qualquer, $(\mathcal{P}(C), \subseteq)$ é um conjunto ordenado por uma ordem pacial (não total)

$(\mathbb{N}^*, |)$, onde $a \mid b$ é a relação de divisibilidade inteira (a divide b por inteiro), é um conjunto ordenado por uma ordem pacial (não total)



Daí em diante e exceto menção explícita em contrário, quando falarmos de conjunto ordenado, subentenderemos que a ordem é parcial.

Elementos mínimos e máximos

seja $(S, \sqsubseteq)$ um conjunto ordenado e um subconjunto S' de S ($S' \subseteq S$). Um elemento x é

- **elemento mínimo** de S' sse
  $x \in S' \ \wedge \ \forall y \in S', x \sqsubseteq y$
- **elemento máximo** de S' sse
  $x \in S' \ \wedge \ \forall y \in S', y \sqsubseteq x$

designamos por **supremo (top, $\top$)** o elemento máximo de S (i.e quando consideramos $S' = S$)

designamos por **ínfimo (bottom, $\bot$)** o elemento mínimo de S

os elementos mínimos e máximos ($\top$ e $\bot$ também) **podem não existir**

seja X um subconjunto de S ($X \subseteq S$)

diz-se de $y \in S$ que é um **limite superior** de X (notação $X \sqsubseteq y$) quando

$$\forall x \in X, x \sqsubseteq y$$

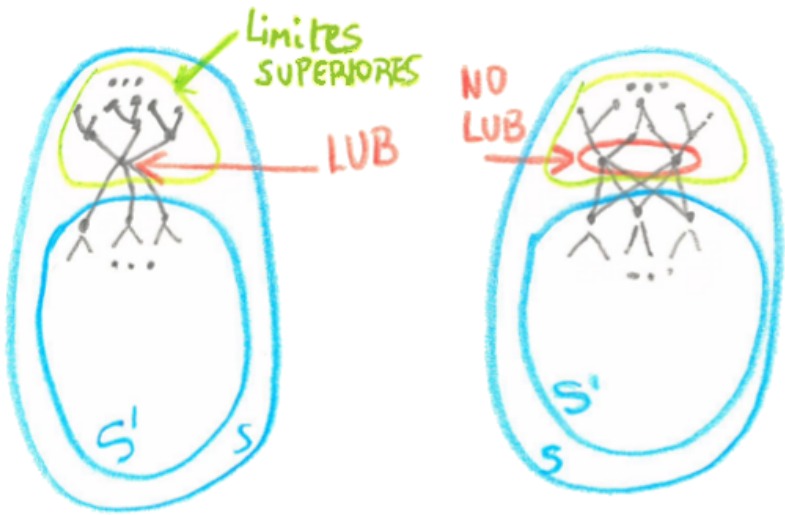
diz-se de $y \in S$ que é um **limite inferior** de X (notação $y \sqsubseteq X$) quando

$$\forall x \in X, y \sqsubseteq x$$

seja X um subconjunto de S ($X \subseteq S$)

designa-se por **menor limite superior / *least upper bound* / lub** de X (notação $\bigsqcup X$) o elemento definido por

$$X \sqsubseteq \bigsqcup X \ \wedge \ \forall y \in S, X \sqsubseteq y \implies \bigsqcup X \sqsubseteq y$$



designa-se por **maior limite inferior / *greatest lower bound* / glb** de X (notação $\bigsqcap X$) o elemento definido por

$$\bigsqcap X \sqsubseteq X \ \wedge \ \forall y \in S, y \sqsubseteq X \implies y \sqsubseteq \bigsqcap X$$

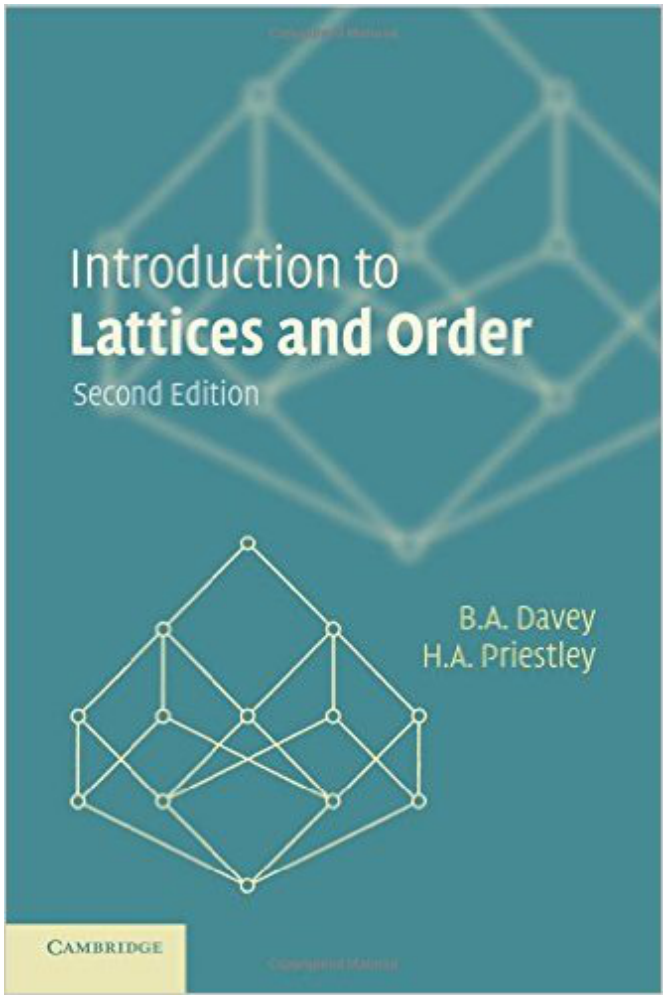
dependendo do conjunto ordenado em causa, estes limites podem igualmente **não existir**

um **reticulado completo** $(S, \sqsubseteq)$ é um conjunto S ordenado por uma ordem parcial $\sqsubseteq$ tal que $\bigsqcap X$ e $\bigsqcup X$ existam qualquer que seja $X \subseteq S$

num reticulado completo é garantido que $\top$ e $\bot$ existam
mais, $\top = \bigsqcap \emptyset = \bigsqcup S$ e $\bot = \bigsqcup \emptyset = \bigsqcap S$

é comum dar uma definição alternativa onde os reticulados são vistos como estruturas algébricas $(S, \sqsubseteq, \top, \bot, \sqcup, \sqcap)$.

um **reticulado** (não necessariamente completo) é um tuplo $(S, \sqsubseteq, \top, \bot, \sqcup, \sqcap)$ tal que para cada par $(x, y) \in S \times S$, $\{x\} \sqcap \{y\}$ e $\{x\} \sqcup \{y\}$ existam (notação $x \sqcap y$ e $x \sqcup y$).



todo o reticulado finito (i.e. em que S é finito) é um reticulado completo

considere o conjunto $S \triangleq \{0, 1, 2\}$ e a relação binária $R \triangleq \{(0,0), (1,1), (2,2), (0,1), (0,2)\}$. Esta relação é uma relação de ordem parcial. Assim (S, R) é um conjunto ordenado, mas não é um reticulado (nem reticulado completo). Isto porque $1 \sqcup 2$ não existe.

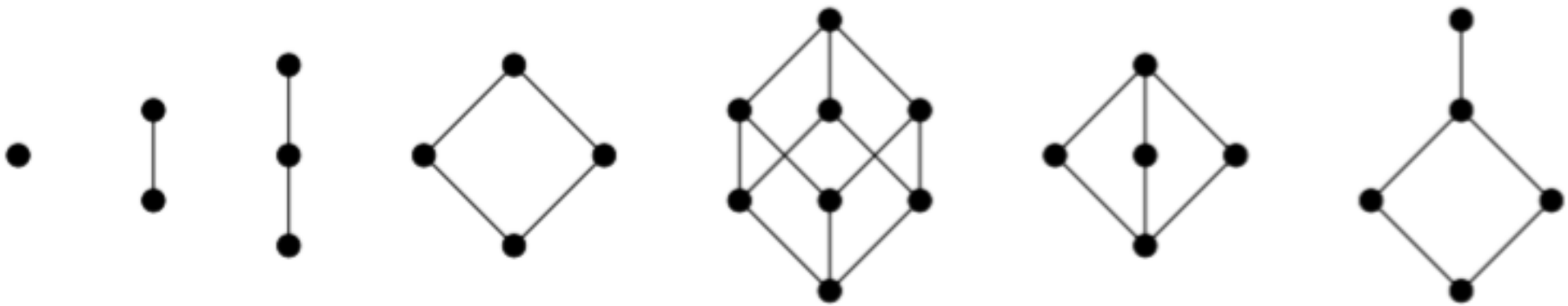
qualquer conjunto ordenado totalmente forma um reticulado com esta ordem.

seja C um conjunto, $(\mathcal{P}(C), \subseteq)$ é um reticulado (completo). Basta considerar $\bot = \emptyset, \top = C, \sqcup = \cup, \sqcap = \cap$.

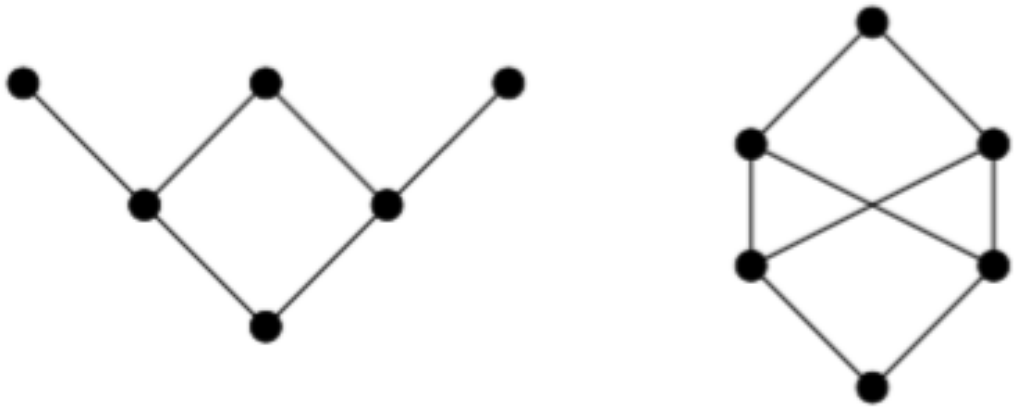
$(\mathbb{N}, |)$ é um reticulado (completo). Basta pensar o $\sqcup$ como sendo o menor múltiplo comum, e $\sqcap$ como o maior divisor comum. $\top$ é 0 (todos dividem 0, visto que $\forall n \in \mathbb{N}, n \times 0 = 0$) e $\bot$ é 1.

seja $Q = \{q \in \mathbb{Q} | 0 \leq q \leq 1\}$. $(Q, \leq)$ é um reticulado, mas não é completo. **Exercício**: porquê?

Exemplos de reticulados



Exercício: porquê?



Porque estes dois conjuntos ordenados não formam reticulados?

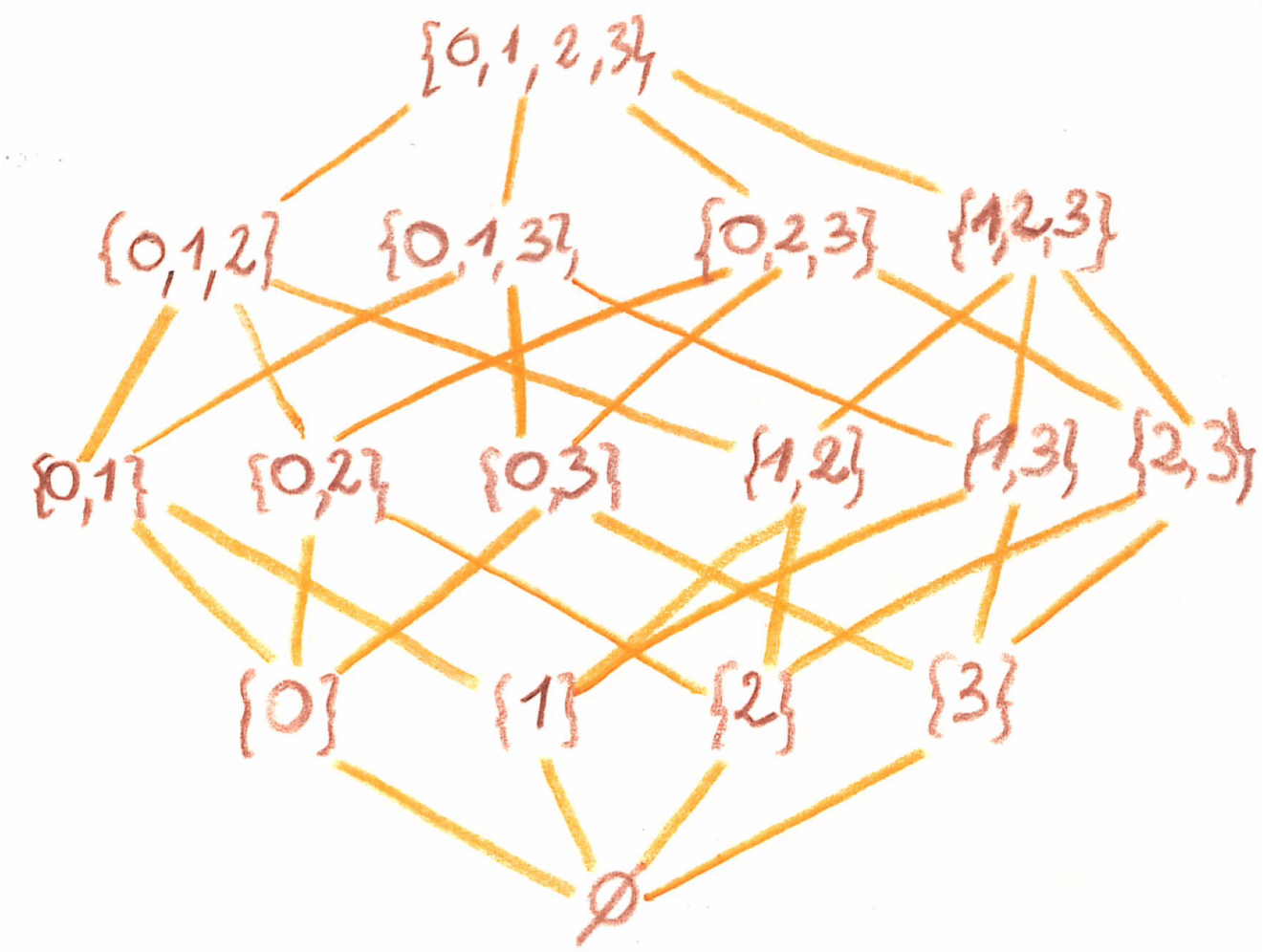
seja $(S, \sqsubseteq)$ um reticulado completo.

1. demonstre que $\top$ e $\bot$ são limites (superior, inferior, resp.) únicos de S
2. demonstre que $\bigsqcup S = \bigsqcap \emptyset$ e $\bigsqcap S = \bigsqcup \emptyset$

Reticulados como Legos: reticulado das partes

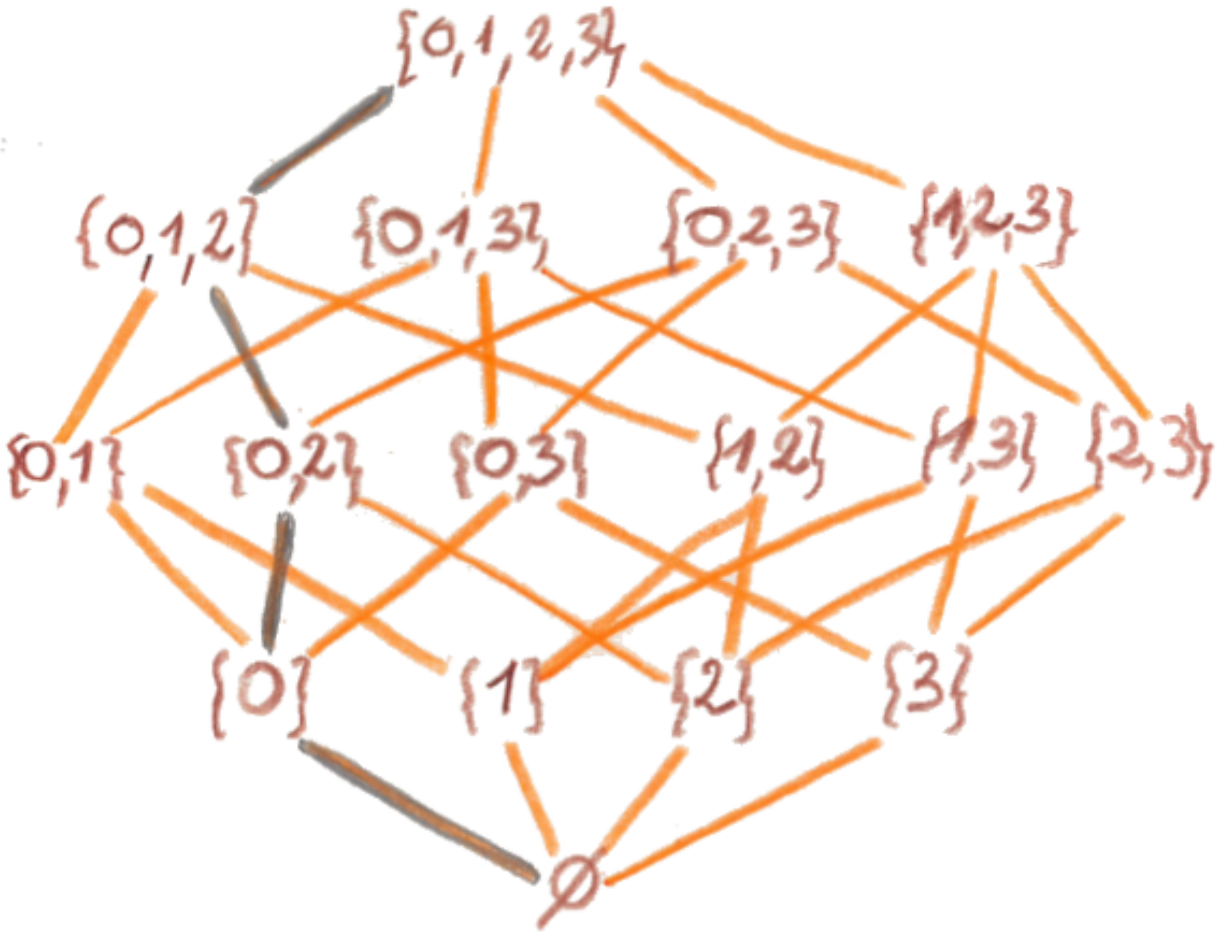
cada conjunto finito A define um reticulado $(\mathcal{P}(A), \subseteq)$ onde $\top = A$, $\bot = \emptyset$, $\sqcap = \cap$, $\sqcup = \cup$: o **reticulado das partes (powerset lattice)**

um exemplo (com $A = \{0, 1, 2, 3\}$)



Uma noção auxiliar: **altura de um reticulado**

é o tamanho do maior caminho entre o $\bot$ até ao $\top$ (pode ser ω)



Reticulados como Legos: reticulados produto e soma

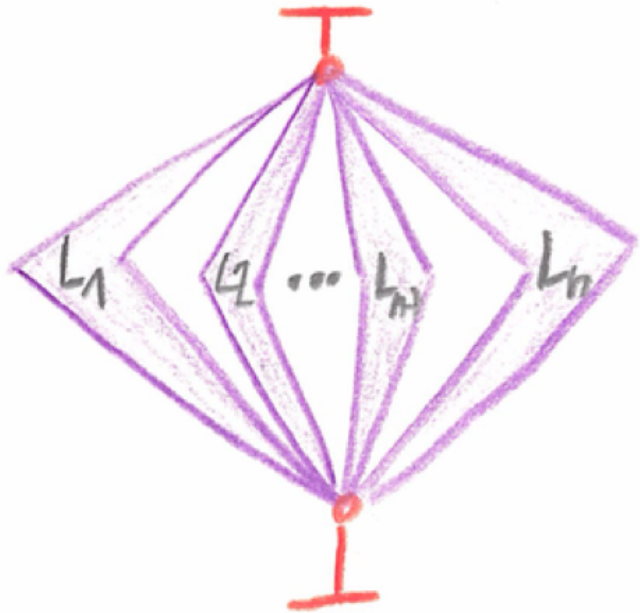
Sejam $(L_1, \sqsubseteq_1), (L_2, \sqsubseteq_2), \ldots, (L_n, \sqsubseteq_n)$ reticulados com altura finita

o **reticulado produto** $(L_p, \sqsubseteq_p)$ é definido como

- $L_p \triangleq L_1 \times L_2 \times \cdots \times L_n$
- $\sqsubseteq_p$ definido como
  $(x_1, \ldots, x_n) \sqsubseteq_p (y_1, \ldots, y_n)$ se e só se
  $(x_1 \sqsubseteq_1 y_1) \wedge \ldots \wedge (x_n \sqsubseteq_n y_n)$
- assim: $altura(L_p) = \Sigma_{1 \leq i \leq n} altura(L_i)$

o **reticulado soma** $(L_s, \sqsubseteq_s)$ é definido como

- $L_s \triangleq L_1 + L_2 + \cdots + L_n = \{(i, x_i) \mid x_i \in L_i \setminus \{\top, \bot\}\} \cup \{\top, \bot\}$
- $\sqsubseteq_s$ definido como $(i, x_i) \sqsubseteq_s (j, y_j)$ se e só se
  $i = j \ \wedge \ x_i \sqsubseteq_i y_i$
- e assim: $altura(L_s) = max_{1 \leq i \leq n} altura(L_i)$



reticulado soma

Reticulados como Legos: reticulados *lift* e *flat*

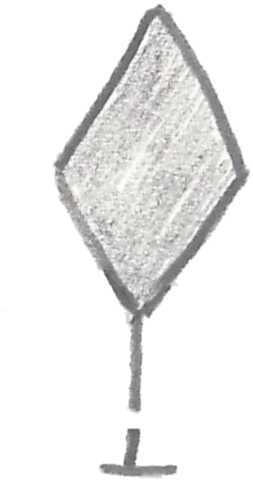
Seja $R = (L, \sqsubseteq)$ um reticulado e A um conjunto

o **reticulado *lift***, denotado por $Lift(R)$ é o reticulado definido como

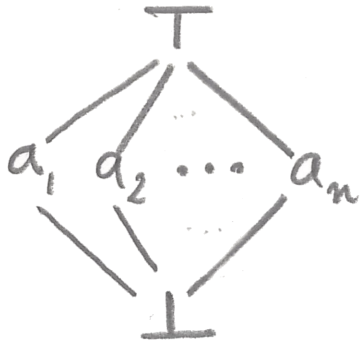
- o conjunto de suporte é $R \cup \{\bot\}$
- a ordem é uma extensão simples de $\sqsubseteq$, $\sqsubseteq_{Lift} = \sqsubseteq \cup (\bot, \bot_R)$
- assim: $altura(Lift(R)) = altura(R) + 1$

o **reticulado flat**, denotado por $Flat(A)$ é definido como

- o conjunto de suporte é $A \cup \{\bot, \top\}$
- a ordem $\sqsubseteq_{flat}$ é definida como $(\forall x \in A, \bot \sqsubseteq_{flat} x)$, $(\forall x \in A, x \sqsubseteq_{flat} \top)$, $(\forall x, y \in A, x \neq y \implies x \not\sqsubseteq_{flat} y)$
- e assim: $altura(Flat(A)) = 2$



reticulado *lift*



reticulado *flat*

Se A é um conjunto e $(L, \sqsubseteq_L)$ é um reticulado então podemos contruir um reticulado, designado de **reticulado mapa** (notação $A \mapsto L$)

o conjunto de suporte é $A \mapsto L = \{[a_1 \mapsto x_1, \ldots, a_n \mapsto x_n] | a_i \in A \text{ e } x_i \in L\}$

a ordem é: $f \sqsubseteq g \triangleq \forall a_i \in A, f(a_i) \sqsubseteq_L g(a_i)$

a altura é então: $altura(A \mapsto L) = |A| \times altura(L)$

Exercício

- Porque razão podemos ver um reticulado *Flat* como um reticulado soma
- Mostre que cada reticulado produto é isomórfico a um reticulado mapa
- Demonstre a afirmação feita sorbe a altura de um reticulado mapa

os reticulados (mas também CPOs, que deixamos de lado por enquanto) formam os fundamentos que suportam a **framework** que vamos agora apresentar.
Esta gera restrições calculadas localmente aos pontos do programa e infere uma solução global ao programa... porque está sustentada num reticulado

```
1 : var a,b;
2 : a =42;
3 : b = a + input;
4 : a = a - b;
```

$$\Longrightarrow \begin{array}{lcl} x_1 & = & [a \to ?, b \to ?] \\ x_2 & = & x_1[a \to +] \\ x_3 & = & x_2[b \to x_2(a) + ?] \\ x_4 & = & x_3[a \to x_3(a) - x_3(b)] \end{array}$$

para cada programa sob analise temos um conjunto de variáveis (associados aos pontos do programa) $x_1 \ldots x_n \in L$ das quais queremos analizar o comportamento (relativamente a uma propriedade)

para tal desenhamos um conjunto de restrições (equações por cumprir) que dependem (de cada ponto) do CFG. Este conjunto estebalece restrições locais. Este conjunto tem a forma

$$\begin{array}{lcl} x_1 & = & F_1(x_1, \cdots, x_n) \\ x_2 & = & F_2(x_1, \cdots, x_n) \\ & \cdots & \\ x_n & = & F_n(x_1, \cdots, x_n) \end{array}$$

estas restrições podem ser agrupadas numa só função $F : L^n \to L^n$:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

os valores de $x_1, \ldots, x_n$ tais que $x_1, \ldots, x_n = F(x_1, \ldots, x_n)$ representam valores para os quais a propriedade desejada é cumprida.
dos valores possíveis, queremos o **menor** tuplo $(x_1, \ldots, x_n)$ (i.e. **os valores com mais precisão**). Trata se do **menor ponto fixo** de F

consideramos daí em diante o reticulado $(L, \sqsubseteq)$

uma função $f : L \to L$ é dita **monótona** quando
$\forall x, y \in L, x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$

uma função com múltipos argumentos é monótona, quando o é em cada argumento

as funções monótonas são fechadas por composição

visto como funções, $\sqcup$ e $\sqcap$ são monótonas

uma função é **extensiva** em L quando $\forall x \in L, x \sqsubseteq f(x)$

a monotonia difere da extensibilidade. Por exemplo, todas as funções constantes são monótonas

a operação $\sqcup$ e a actualização dos mapas (associação entre as variáveis e os seus sinais) são monótonas

a composição em uso preserva a monotonia

serão os operadores abstractos $\overline{op}$ utilizados monótonos?

isto pode ser verificado, de forma manual mas *dispendiosa*

ou então por uma análise baseada num algoritmo em $\mathcal{O}(n^3)$ numa tabela de *op* de tamanho n por n

$$\forall x, y, x' \in L.\ x \sqsubseteq x' \implies (x\ \overline{op}\ y) \sqsubseteq (x'\ \overline{op}\ y)$$

$$\forall x, y, y' \in L.\ y \sqsubseteq y' \implies (x\ \overline{op}\ y) \sqsubseteq (x\ \overline{op}\ y')$$

$x \in L$ é um ponto fixo de $f : L \to L$ se e só se $f(x) = x$

Teorema do menor ponto fixo

Num reticulado com altura finita, cada função monótona f tem um **menor ponto fixo único**

$$fix(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

é trivial constatar que $\bot \sqsubseteq f(\bot)$

como f é monótona, temos igualmente $f(\bot) \sqsubseteq f^2(\bot)$ (onde $f^2(\bot) = f(f(\bot))$)

por indução chegamos a constatação de que $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$

o que significa que

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots \sqsubseteq f^i(\bot) \sqsubseteq \ldots$$

é uma cadeia crescente

L tem uma altura finita, então para um dado k, $f^k(\bot) = f^{k+1}(\bot)$

definimos assim $fix(f) = f^k(\bot)$

Prova de unicidade

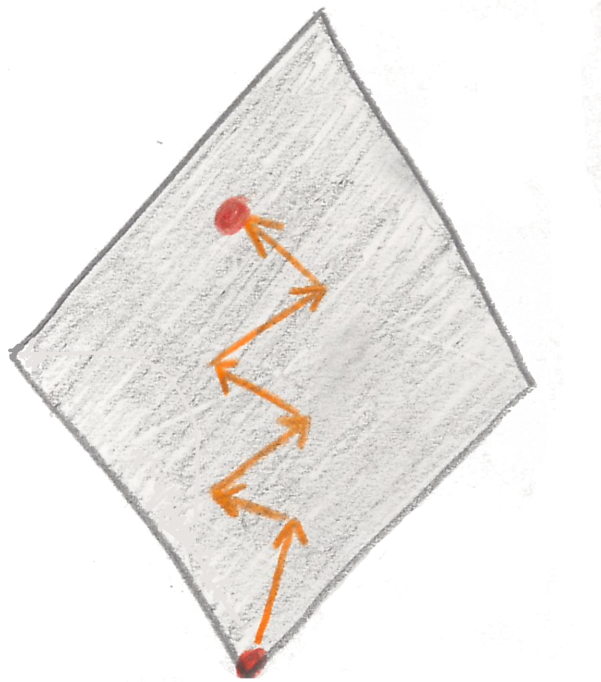
assumimos que existe um outro ponto fixo $x = f(x)$
é fácil ver que $\bot \sqsubseteq x$
por indução , $f^i(\bot) \sqsubseteq f^i(x) = x$

em particular , $fix(f) = f^k(\bot) \sqsubseteq x$, i.e. $fix(f)$ é um menor ponto fixo

dizer que é **o** menor ponto fixo segue por anti-simetria

a complexidade do calculo de $fix(f)$ depende de

- a altura do reticulado
- o custo do cálculo de f
- o custo do teste de igualdade



$x = \bot$
do {
  $t = x$;
  $x = f(x)$;
} while $(x \neq t)$

relembramos que a função $f : L \to L$ é monótona quando

$$\forall x, y \in L . x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$$

quando $x \sqsubseteq y$ dizemos que y é uma **aproximação segura** de x, ou então que é uma **sobre aproximação** de x, ou então que x é **pelo menos tão preciso quanto** y.

assim, f é uma função que **optimiza aproximações**

quando f é monótona, temos o *lema*

more precise input cannot lead to less precise output

comecemos por relembrar que assumimos aqui um reticulado $(L, \sqsubseteq)$ de altura finita

o sistema de equações que tetamos revolver é da forma

$$\begin{array}{lcl} x_1 & = & F_1(x_1, \cdots, x_n) \\ x_2 & = & F_2(x_1, \cdots, x_n) \\ & \cdots & \\ x_n & = & F_n(x_1, \cdots, x_n) \end{array}$$

onde os x_i são vaxiáveis em L e $F_i : L^n \to L$

de notar que L^n é um reticulado produto (**Exercício:** explique porquê?)

cada sistema de equação tem uma solução menor única que é o menor ponto fixo da função $F = L^n \to L^n$ definida como

$$F(x_1, \cdots, x_n) = (F_1(x_1, \cdots, x_n), \ldots, F_n(x_1, \cdots, x_n))$$

Uma solução é sempre um ponto fixo

e o menor deles é o mais preciso

um sistema de inequação no nosso contexto dos reticulados tema forma

$$\begin{array}{lcl} x_1 & \sqsubseteq & F_1(x_1, \cdots, x_n) \\ x_2 & \sqsubseteq & F_2(x_1, \cdots, x_n) \\ & \cdots & \\ x_n & \sqsubseteq & F_n(x_1, \cdots, x_n) \end{array} \quad \text{ou} \quad \begin{array}{lcl} x_1 & \sqsupseteq & F_1(x_1, \cdots, x_n) \\ x_2 & \sqsupseteq & F_2(x_1, \cdots, x_n) \\ & \cdots & \\ x_n & \sqsupseteq & F_n(x_1, \cdots, x_n) \end{array}$$

pode ser resolvido da mesma forma de um sistema de equações se aproveitarmos os seguintes factos

$$x \sqsubseteq y \;\Leftrightarrow\; x = x \sqcap y$$

e

$$x \sqsupseteq y \;\Leftrightarrow\; x = x \sqcup y$$

Exercício: demonstre estas duas equivalências

consideramos aqui um CFG por analisar, com os nodos $V = \{v_1, v_2, \cdots, v_n\}$

dispomos igualmente de um reticulado de tamanho finito L das possíveis respostas

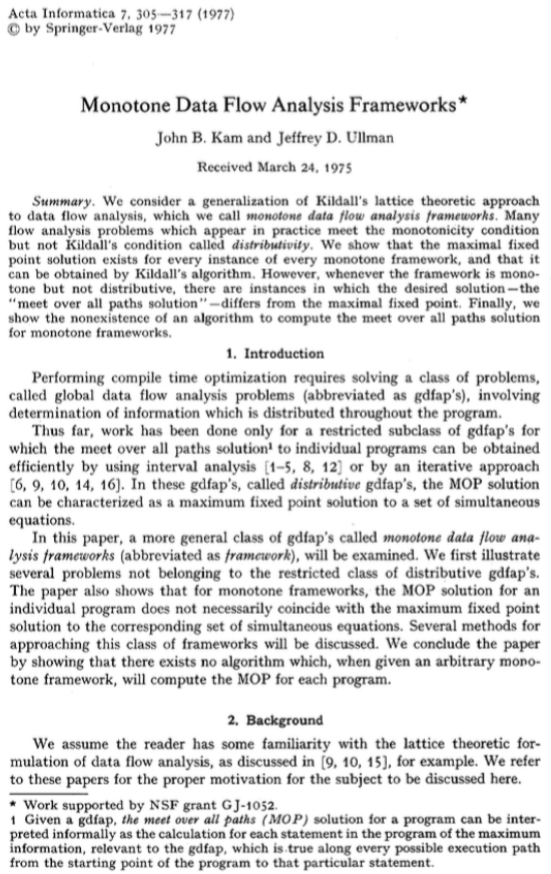
este é fixo ou depende do dado programa por analisar

consideramos uma variável (um ponto de restrição) $[\![v]\!] \in L$ para cada vértice v do CFG

uma restrição (uma equação, ou inequação) de fluxo de dado associada a cada construção da linguagem

- que liga o valor de $[\![v]\!]$ às variáveis de outros vértices
- tipicamente um vértice está ligado aos seus vizinhos
- as restrições **devem** ser funções monótonas

$$[\![v_i]\!] = F_i([\![v_1]\!], [\![v_2]\!], \ldots, [\![v_n]\!])$$



Acta Informatica 7, 305—317 (1977)
© by Springer-Verlag 1977

Monotone Data Flow Analysis Frameworks*

John B. Kam and Jeffrey D. Ullman

Received March 24, 1975

Summary. We consider a generalization of Kildall's lattice theoretic approach to data flow analysis, which we call *monotone data flow analysis frameworks*. Many flow analysis problems which appear in practice meet the monotonicity condition but not Kildall's condition called *distributivity*. We show that the maximal fixed point solution exists for every instance of every monotone framework, and that it can be obtained by Kildall's algorithm. However, whenever the framework is monotone but not distributive, there are instances in which the desired solution—the "meet over all paths solution"—differs from the maximal fixed point. Finally, we show the nonexistence of an algorithm to compute the meet over all paths solution for monotone frameworks.

1. Introduction

Performing compile time optimization requires solving a class of problems, called global data flow analysis problems (abbreviated as gdfap's), involving determination of information which is distributed throughout the program.

Thus far, work has been done only for a restricted subclass of gdfap's for which the meet over all paths solution[1] to individual programs can be obtained efficiently by using interval analysis [1–5, 8, 12] or by an iterative approach [6, 9, 10, 14, 16]. In these gdfap's, called *distributive* gdfap's, the MOP solution can be characterized as a maximum fixed point solution to a set of simultaneous equations.

In this paper, a more general class of gdfap's called *monotone data flow analysis frameworks* (abbreviated as *framework*), will be examined. We first illustrate several problems not belonging to the restricted class of distributive gdfap's. The paper also shows that for monotone frameworks, the MOP solution for an individual program does not necessarily coincide with the maximum fixed point solution to the corresponding set of simultaneous equations. Several methods for approaching this class of frameworks will be discussed. We conclude the paper by showing that there exists no algorithm which, when given an arbitrary monotone framework, will compute the MOP for each program.

2. Background

We assume the reader has some familiarity with the lattice theoretic formulation of data flow analysis, as discussed in [9, 10, 15], for example. We refer to these papers for the proper motivation for the subject to be discussed here.

* Work supported by NSF grant GJ-1052.

1 Given a gdfap, *the meet over all paths (MOP)* solution for a program can be interpreted informally as the calculation for each statement in the program of the maximum information, relevant to the gdfap, which is true along every possible execution path from the starting point of the program to that particular statement.

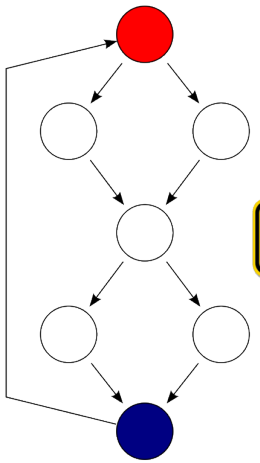
extrair todas as restrições referente ao CFG auscultado
resolver todas as restrições utulizando o algotimo de procura de ponto fixo

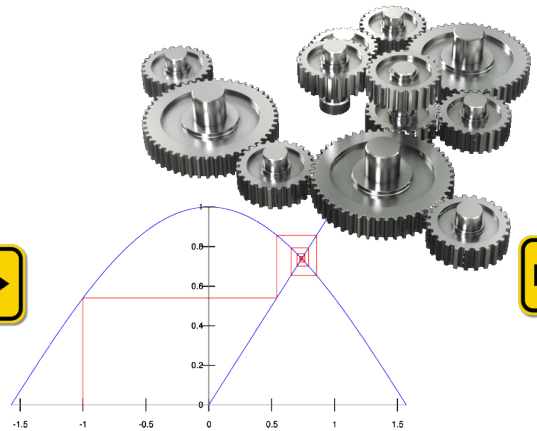
- trabalhamos no reticulado L^n
- calcular o menor ponto fixo da combinação das funções

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

esta solução dá uma resposta em L para cada vértice do CFG

Geração e resolução das restrições

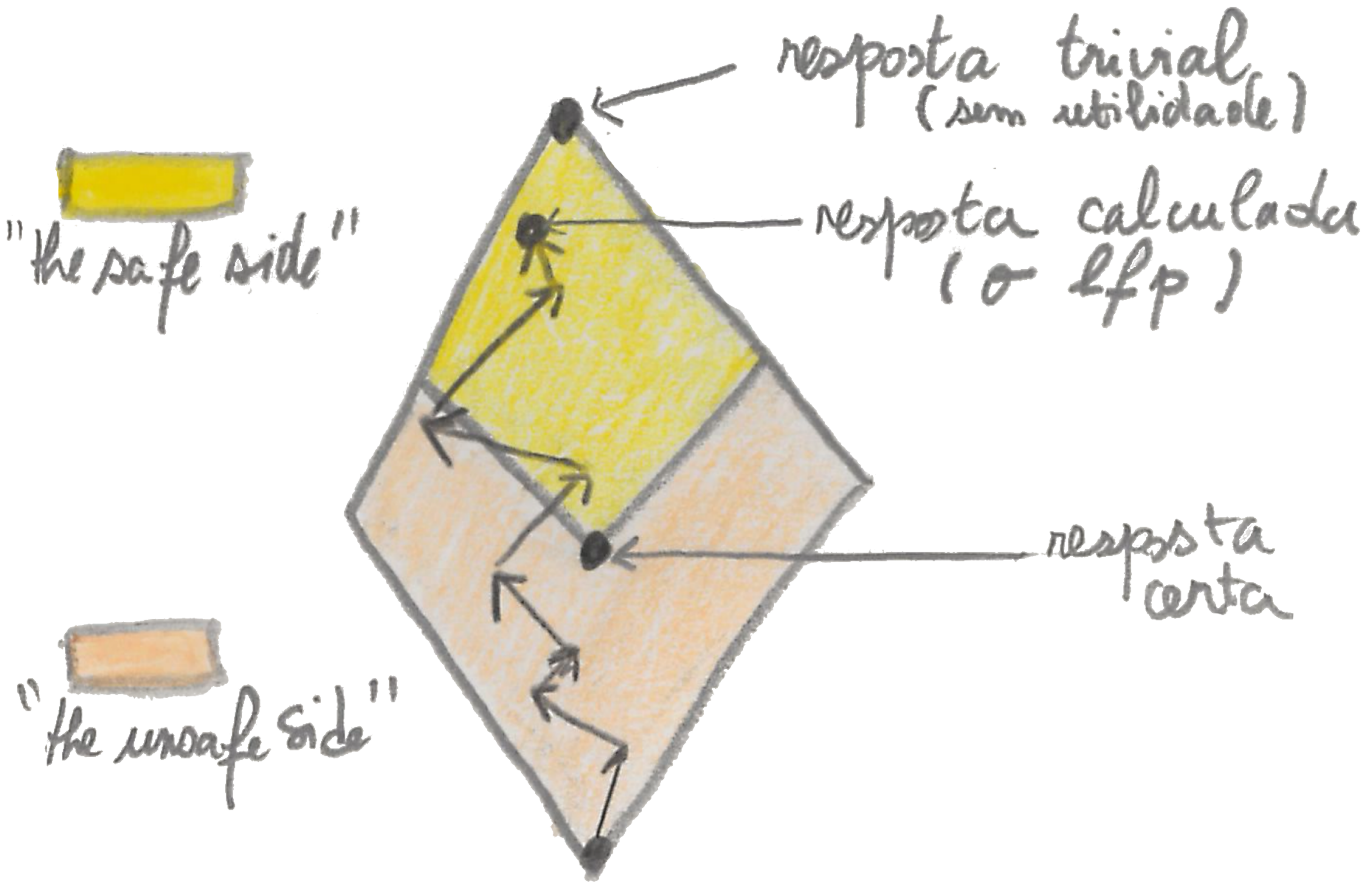




[[p]] = &int
[[q]] = &int
[[malloc]] = &int
[[x]] = ϕ
[[foo]] = ϕ
[[&n]] = &int
[[main]] = ()->int

CFG | Recolha de restrições | Fixed-Point Solver | Solução

Pontos do reticulado como respostas



Aproximações conservadoras...

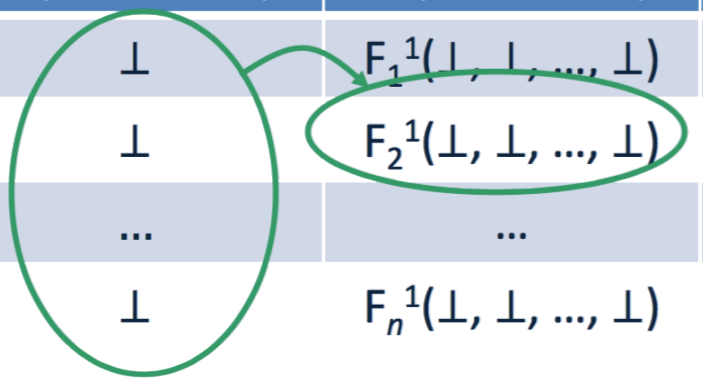
$x = (\bot, \bot, \cdots, \bot)$
do $\{$
  $t = x;$
  $x = F(x);$
$\}$ *while*$(x \neq t)$

a correção deste algoritmo é assegurado pelo teorema do ponto fixo

mas não há aqui nenhum proveito da estrutura de L^n lu de F (por exemplo do facto de $x \in L^n$ ou de $F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \cdots, F_n(x_1, \ldots, x_n))$)

Algoritmo natural

	$F^0(\bot, \bot, ..., \bot)$	$F^1(\bot, \bot, ..., \bot)$	...	$F^k(\bot, \bot, ..., \bot)$
1	$\bot$	$F_1^1(\bot, \bot, ..., \bot)$	...	$F_1^k(\bot, \bot, ..., \bot)$
2	$\bot$	$F_2^1(\bot, \bot, ..., \bot)$	...	$F_2^k(\bot, \bot, ..., \bot)$
...	...	...	...	...
n	$\bot$	$F_n^1(\bot, \bot, ..., \bot)$	...	$F_n^k(\bot, \bot, ..., \bot)$



calcular cada nova entrada numa coluna é feito com base na coluna anterior

- sem usar as entradas da coluna que se preenche
- muitas das entradas são susceptíveis de mudar

$x_1 = \bot;\ \cdots\ ;\ x_n = \bot$
while $(\exists i \in \mathbb{N}.\ x_i \neq F_i(x_1, \cdots, x_n))\{$
  escolher um i de forma não determinística tal que $x_i \neq F_i(x_1, \cdots, x_n)$
  $x_i = F_i(x_1, \cdots, x_n);$
$\}$

exploramos aqui explicitamente a estrutura de L^n

pode requerer um número maior de iterações, mas essas são em geral menos custosas

Assumindo que temos uma forma eficiente de determinar se a condição do ciclo é válida ou não, explique porque a iteração caótica é melhor (mais eficiente) do que o algoritmo natural

NB. é o estudo desta assunção que nos permite melhorar o algoritmo da iteração caótica

Correção da iteração caótica

seja x^j o valor de $x = (x_1, \cdots, x_n)$ na j-ésima iteração do algoritmo natural

seja $\underline{x^j}$ o valor de $x = (x_1, \cdots, x_n)$ na j-ésima iteração do algoritmo com iteração caótica

por indução sobre j, mostramos que $\forall j \in \mathbb{N}, \underline{x^j} \sqsubseteq x^j$

a iteração caótica termina em final de contas, no ponto fixo

este ponto fixo deve ser idêntico ao algoritmo natural, visto ser o menor ponto fixo

a escolha aleatória do i (em $\exists i \cdots$) no algoritmo por iteração caótica não é prática

ideia: prever i por uma análise e pela estrutura do programa

exemplo: na análise de sinal, quando terminamos o processamento de um vértice v do CFG, processar em seguida o $succ(v)$.

O algoritmo *lista de afazeres*

é uma especialização da iteração caótica que tira proveito da estrutura de F

em grande parte, as partes direitas dos F_i são escassas, i.e. as restrições correspondentes no CFG não envolvem todos os vértices, pelo contrário, só alguns poucos

usa-se assim um mapa $dep : V \to \mathcal{P}(V)$ que dá, para cada $v \in V$ as variáveis w tais que v ocorre na parte direita das restrições sobre w

O algoritmo *lista de afazeres*

$x_1 = \perp;\ \cdots;\ x_n = \perp$
$W = \{1, \cdots, n\}$; // a lista de a-fazeres
while $(w \neq \emptyset)$){
  $i = W.removeNext();$
  $y = F_i(x_1, \cdots, x_n);$
  if $(y \neq x_i)$ {
    for$(v_j \in dep(v_i))$ $W.add(j);$
    $x_i = y;$
  }
}

O algoritmo *lista de afazeres*

Que invariante de ciclo propor para possibilitar a prova de correcção do algoritmo?

1. $[\![v]\!] = f_v(\bigsqcup_{w \in pred(v)} [\![w]\!])$
2. $[\![v]\!] = \bigsqcup_{w \in pred(v)} f_v([\![w]\!])$

   pode ser mais preciso (a não ser que f_v seja distributiva)

   mas também mais custosa

   Exercício: Porquê?

3. $\forall w \in succ(v).f_v([\![v]\!]) \sqsubseteq [\![w]\!]$

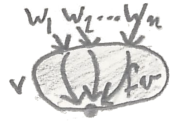
   pode requerer menos operaçãos *JOIN* se há o CFG tem muitas arestas

   mais adequado para analises inter-procedurais

O algoritmo *lista de afazeres* original

$x_1 = \bot;\ \cdots;\ x_n = \bot$
$W = \{1, \cdots, n\}$; // a lista de a-fazeres
while $(w \neq \emptyset)$){
  $i = W.removeNext()$;
  $y = F_i(x_1, \cdots, x_n)$; $\Longrightarrow$
  if $(y \neq x_i)$ {
    for$(v_j \in dep(v_i))$ $W.add(j)$;
    $x_i = y$;
  }
}

1. $f_i(\bigsqcup_{j \in pred(i)} x_j)$
   ou
2. $\bigsqcup_{j \in pred(i)} f_i(x_j)$



O algoritmo *lista de afazeres* com propagação

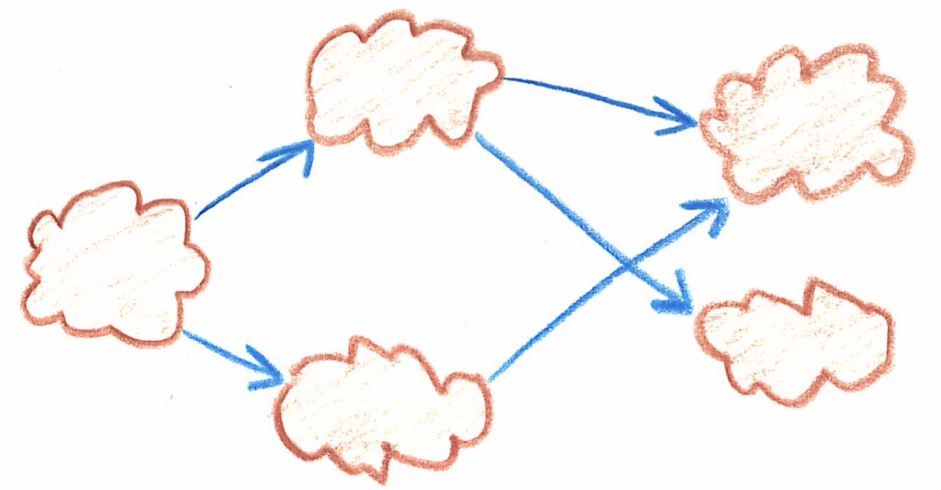
$x_1 = \bot;\ \cdots;\ x_n = \bot$
$W = \{1, \cdots, n\}$; // a lista de a-fazeres
while $(w \neq \emptyset)$){
  $i = W.removeNext()$;
  $y = f_i(x_i)$
  for$(v_j \in dep(v_i))$ {
    propagate(y, j);
  }
}

propagate$(y,\ j)$ {
  $t = x_j \sqcup y$;
  if$(t \neq x_j)$ {
    $W.add(j)$;
    $x_j = t$;
  }
}

Mais algumas melhorias

representar a lista de afazeres como uma fila com prioridades (há em geral heurísticas interessantes por explorar na definição das prioridades)

olhar para o grafo de dependência das arestas, ver como se organizam as componentes (fortemente) conexas do grafo e, tendo em conta o grafo resultante, resolver as restrições de baixo para cima



foi o que fizemos na analise de vivacidade no capítulo de optimização de código!

Analise de sinal, finalmente

Uma melhoria ao algoritmo *lista de afazeres*:

- juntar inicialmente somente o ponto de entrada
- deixar, então, o fluxo de dados propagar-se pelo programa, conforme as restrições definidas

neste ponto, e se as restrições para as declarações de variáveis fossem

$$[\![var x_1, \cdots, x_n]\!] = JOIN(v)[x_1 \to \bot, \cdots, x_n \to \bot]???$$

(isto pode fazer sentido, se tratamos o estado "não inicializado" como "sem valor" no logar de "qualquer valor")

Problema a iteração poderia parar antes do ponto fixo
Solução: substituir $Vars \to Sign$ por $lift(Vars \to Sign)$ (o que nos permite poder distinguir entre "não atingível" e "nenhuma variável tem valor")

com esta analise, temos uma forma (por enquanto, um pouco agressiva) de detectar potenciais divisões por zero. Para melhor, precisamos de uma analise **path sentitive**.

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*!).