Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 16 - Analise estática de programas com apontadores

Programa

- Introdução à análise estática de apontador
- Análise de Andersen
- Análise de Steensgaard
- Análise interprocedimental de apontador
- Análise de apontador nulo
- Análise de forma

Introdução à análise estática de apontador

Analisar programas com apontadores

como fazer, por exemplo, uma análise de propagação de constantes em programas que manipulam apontadores ou referências para objectos?

```
...
*x= 42;
*y= 87;
z = *x;
// será z igual a 42 ou 87 ?
```

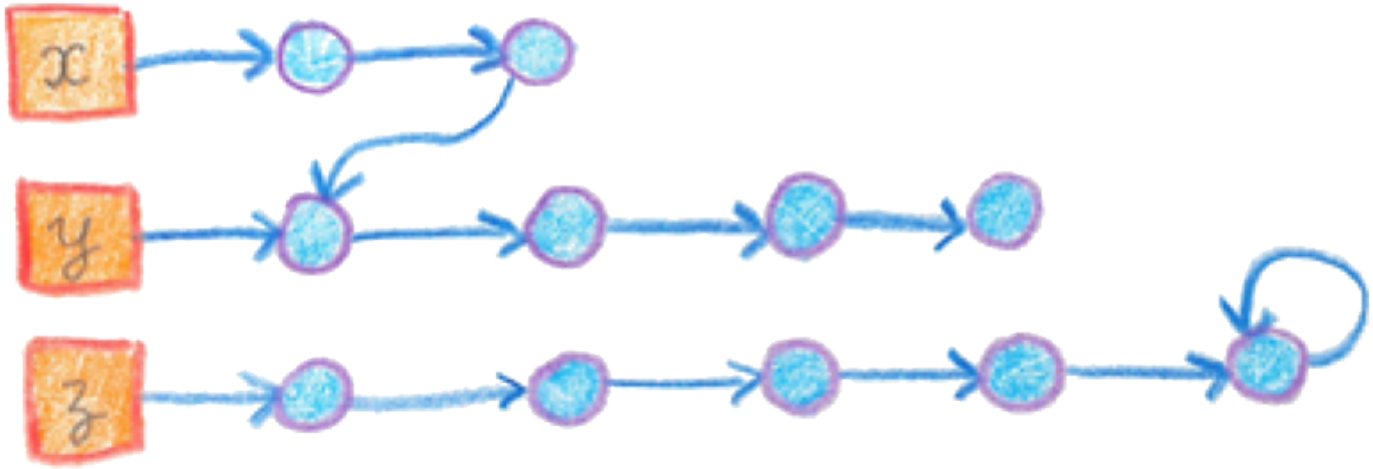
```
E -> &x
| malloc
| *E
| null
| ...
| (E) (E ,..., E)
| ...

S -> *x = E;
| ...
```

Apontadores para a *heap*

os apontadores na linguagem de estudo (TIP) são limitados

- `malloc` aloca uma única célula, somente
- só estruturas lineares podem ser construídas na *heap*



ignoremos por enquanto os apontadores de funções

mesmo assim, há neste contexto, desafios interessantes para as análises estáticas

Alvo dos apontadores

a questão fundamental sobre os apontadores é: **para que localizações podem apontar os apontadores?**

para responder, precisamos de uma abstracção adequada

o conjunto das células (abstractas) *Cell* contém

- *malloc-i* para cada local de alocação com índice *i*
- *x* para cada variável de nome *x* do programa

esta abstracção é conhecida como **abstracção dos locais de alocação** (*allocation site abstraction*)

o conjunto de todas os locais (abstractos), *Loc*, contem `&p` para cada célula *p*

cada célula abstracta pode corresponder a várias células de memória em tempo de execução

procura determinar para cada variável apontador x o **conjunto $pt(x)$ das células para as quais x pode apontar**

a análise é **conservadora** (*pode apontar para... - may point-to*)

- o conjunto pode ser abragente demais
- pode mostrar a ausência de *aliasing*: $pt(x) \cap pt(y) = \emptyset$

```
...
*x= 42;
*y= 87;
z = *x;
// será z igual a 42
//ou 87 ?
```

concentremo-nos nas análises que não são sensíveis ao fluxo

- que se debruçam sobre a *AST*
- antes ou aquando da análise de fluxo de controlo

Obter informação dos apontadores

uma análise simples (*address taken analysis*) :

- incluir todas as células `malloc-i`
- incluir a célula x se a expressão `&x` occorre no programa

no contexto de uma linguagem tipada, pode eliminar-se as células com tipos incompatíveis (com a variável apontador em causa)

para situações simples, a precisão desta análise é suficiente e é de facto computacionalmente bastante eficaz
precisamos de uma análise mais súbtil para os casos menos triviais

Normalização de apontadores

preparemos o contexto para as análises que seguem:

assuma que todas as utilizações programáticas de apontadores são normalizadas

```
x=malloc
x=&y
x=y
x=*y
*x=y
x=null
```

introduz muitas variáveis temporárias

mas agora todas as sub-expressões são simples (`malloc`, `null` ou nomes/localizações)

hipótese de trabalho: ignoramos o facto de as células criadas no momento da declaração e por malloc não são inicializadas

Análise de Andersen

para cada célula c, introduzimos uma variável de restrição $[\![c]\!]$ que toma valor sobre os conjuntos de localizações, i.e. $[\![.]\!] : \mathit{Cell} \to \mathcal{P}(\mathit{Loc})$

$$
\begin{array}{lcl}
x = \mathit{malloc} & \dashrightarrow & \&\mathit{malloc}\text{–}i \in [\![x]\!] \\
x = \&y & \dashrightarrow & \&y \in [\![x]\!] \\
x = y & \dashrightarrow & [\![y]\!] \in [\![x]\!] \\
x = *y & \dashrightarrow & \&\alpha \in [\![y]\!] \implies [\![\alpha]\!] \in [\![x]\!] \\
*x = y & \dashrightarrow & \&\alpha \in [\![x]\!] \implies [\![y]\!] \in [\![\alpha]\!] \\
x = \mathit{null} & \dashrightarrow & \text{sem restrições}
\end{array}
$$

o mapa de apontadores (o mapa *points-to*) é definido como

$$pt(x) \triangleq \{\alpha \mid \&\alpha \in [\![x]\!]\}$$

as restrições geradas podem ser resolvidas pela framework cúbica!

a solução mínima única é dada em $\mathcal{O}(n^3)$
na prática, para Java (por exemplo), a complexidade é $\mathcal{O}(n^2)$

a análise é *flow-insentitive* mas é **dirigida** (sabemos em que direção os valores fluem nas atribuições)

o programa em análise

```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

restrições geradas

$$
\begin{array}{rcl}
\&malloc-1 \in [\![p]\!] & & \\
[\![y]\!] \subseteq [\![x]\!] & & \\
[\![z]\!] \subseteq [\![x]\!] & & \\
\&\alpha \in [\![p]\!] & \Longrightarrow & [\![z]\!] \subseteq [\![\alpha]\!] \\
[\![q]\!] \subseteq [\![p]\!] & & \\
\&y \in [\![q]\!] & & \\
\&\alpha \in [\![p]\!] & \Longrightarrow & [\![\alpha]\!] \subseteq [\![x]\!] \\
\&z \in [\![p]\!] & &
\end{array}
$$

menor solução

$pt(p) = \{malloc\text{–}1, y, z\}$

$pt(q) = \{y\}$

Exercício

mostre que a normalização dos programas previamente necessária à aplicação do algoritmo de Andersen pode ser automatizada com base na introdução adequada de variáveis temporárias frescas

Análise de Steensgaard

Um desvio necessário

os mecanismos de resolução necessários a análise de Steengaard assentam nos mesmos princípios envolvidos na tipagem
revisitamos o algoritmo de tipagem visto aqui como uma ***framework*** **de unificação**

sobre a AST

para cada instrução/expressão E (incluido identificadores) temos o seu tipo como a solução de $[\![E]\!]$

assim

$$
\begin{array}{lcl}
\mathit{intconst} & \dashrightarrow & [\![\mathit{intcons}]\!] = \mathit{int} \\
E_1\ \mathit{op}\ E_2 & \dashrightarrow & [\![E_1]\!] = [\![E_2]\!] = [\![E_1\ \mathit{op}\ E_2]\!] = \mathit{int} \\
E_1\ ==\ E_2 & \dashrightarrow & [\![E_1]\!] = [\![E_2]\!]\ \mathit{land}\ [\![E_1\ \mathit{op}\ E_2]\!] = \mathit{int} \\
\mathit{input} & \dashrightarrow & [\![\mathit{input}]\!] = \mathit{int} \\
x\ =\ E & \dashrightarrow & [\![x]\!] = [\![E]\!] \\
\mathit{output}\ E; & \dashrightarrow & [\![E]\!] = \mathit{int} \\
\mathit{if}\ (E)\{S\} & \dashrightarrow & [\![E]\!] = \mathit{int} \\
\mathit{if}\ (E)\{S_1\}\ \mathit{else}\ \{S_2\} & \dashrightarrow & [\![E]\!] = \mathit{int} \\
\mathit{while}\ (E)\{S\} & \dashrightarrow & [\![E]\!] = \mathit{int}
\end{array}
$$

já sabemos que é possível generalizar para incluir subtipagem, polimorfismo, ordem superior, etc.

Restrições para a tipagem

$$
\begin{array}{lcl}
f(p_1,\ldots,p_n)\{\ldots\ \mathit{return}\ E;\} & \dashrightarrow & [\![f]\!] = ([\![p_1]\!],\ldots,[\![p_n]\!]) \to [\![E]\!] \\
(E)(E_1,\ldots,E_n) & \dashrightarrow & [\![E]\!] = ([\![E_1]\!],\ldots,[\![E_n]\!]) \to [\![(E)(E_1,\ldots,E_n)]\!] \\
\&x & \dashrightarrow & [\![\&x]\!] = \&[\![x]\!] \\
\mathit{malloc} & \dashrightarrow & [\![\mathit{malloc}]\!] = \&\alpha \\
\mathit{null} & \dashrightarrow & [\![\mathit{null}]\!] = \&\alpha \\
*E & \dashrightarrow & [\![E]\!] = \&[\![*E]\!] \\
*x = E & \dashrightarrow & [\![x]\!] = \&[\![E]\!]
\end{array}
$$

de realçar que introduzimos uma variável de tipo α nas restrições

Problema da unificação

o problema da unificação trata da condições necessárias para igualar dois termos com variáveis

um termo (com variável) é construído com base em símbolos de função (designados por constructores) com respectiva aridade e símbolos de variáveis

por exemplo tendo os símbolos de funções a, b, c (aridade 0 - constantes), d, e (aridade 1), f, g, h (aridade 2), i, j, k (aridade 3)
a, $d(a)$, $h(a, g(d(a), b))$ são termos sem variáveis
$f(X, b)$, $h(X, g(Y, Z))$ são termos com variáveis

assim, concretamente um problema de unificação é encontrar a substituição ρ mais geral das variáveis (***most general unifier***) que permitam aos dois termos se tornarem sintaticamente iguais
por exemplo

$$k(X, bY) \stackrel{?}{=} k(f(Y, Z), Z, d(Z))$$

sim se $\rho = [X = f(d(b), b),\ Y = d(b),\ Z = b]$

não há soluções possíveis a um problema de unificação quando situações como as seguintes ocorram:

tentativa de identificação de símbolos de função diferentes

$$d(X) \stackrel{?}{\neq} e(X)$$

tentativa de identificação de termos com aridades erradas ou diferentes (eventualmente com os mesmo símbolo à cabeça)

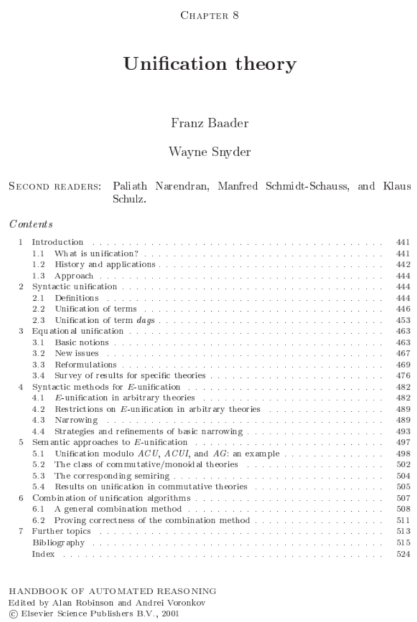
$$a \stackrel{?}{\neq} a(X)$$

Algoritmos para a unificação

nos casos de **termos de tamanho finito**, podemos usar o algoritmo de Paterson e Wegman (1978) em $\mathcal{O}(n)$
encontra o unificador mais geral ou informa que este não existe

nos casos em que se considera **termos infinitos** (mas com uma imposição de um **padrão de regularidade** sobre os termos infinitos) podemos usar o algoritmo de unificação de G. Huet (1976) cuja implementação faz uso da técnica de ***union-find***. O algoritmo é relativamente eficiente.

(ver aulas sobre a Análise de Tipos, T e TP onde introduzimos em particular tanto o algoritmo W como o da unificação)



CHAPTER 8

Unification theory

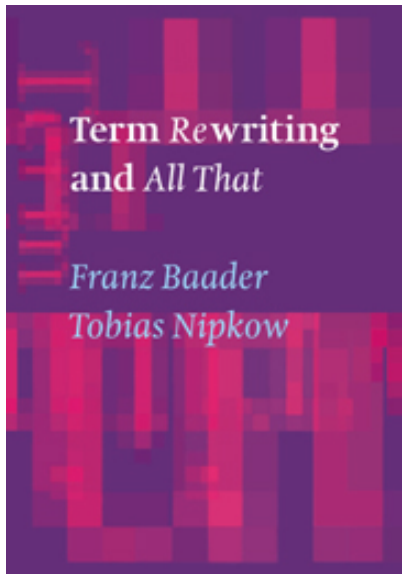
Franz Baader

Wayne Snyder

SECOND READERS: Paliath Narendran, Manfred Schmidt-Schauss, and Klaus Schulz.

Contents

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov



o conceito de base: ver as atribuições como fluxos bi-direcionais: igualdades

as restrições geradas são :

$x = \mathit{malloc}$	$\dashrightarrow$	$\&\mathit{malloc}\text{–}i \in [\![x]\!]$
$x = \&y$	$\dashrightarrow$	$\&y \in [\![x]\!]$
$x = y$	$\dashrightarrow$	$[\![y]\!] = [\![x]\!]$
$x = *y$	$\dashrightarrow$	$\&\alpha \in [\![y]\!] \implies [\![\alpha]\!] = [\![x]\!]$
$*x = y$	$\dashrightarrow$	$\&\alpha \in [\![x]\!] \implies [\![y]\!] = [\![\alpha]\!]$
$x = \mathit{null}$	$\dashrightarrow$	sem restrições

geramos também as restrições adicionais seguintes (restrições de congruência):

$$\&\alpha_1 \&\alpha_2 \in [\![t]\!] \implies [\![\alpha_1]\!] = [\![\alpha_2]\!]$$

$$[\![t_1]\!] \cap [\![t_2]\!] \neq \emptyset \implies [\![t_1]\!] \cap [\![t_2]\!]$$

o mapa *points-to* é

$$pt(x) \triangleq \{\alpha \mid \&[\![\alpha]\!] \in [\![x]\!]\}$$

a procura de solução requer um algoritmo de unificação (como no caso da resolução do problema de tipagem, via por exemplo *union-find*)

Um exemplo

o programa em análise

```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

restrições geradas (sem as restrições de congruência)

$$
\begin{array}{rcl}
\&malloc-1 \in [\![p]\!] & & \\
[\![y]\!] = [\![x]\!] & & \\
[\![z]\!] = [\![x]\!] & & \\
\&\alpha \in [\![p]\!] & \Longrightarrow & [\![z]\!] = [\![\alpha]\!] \\
[\![q]\!] = [\![p]\!] & & \\
\&y \in [\![q]\!] & & \\
\&\alpha \in [\![p]\!] & \Longrightarrow & [\![\alpha]\!] = [\![x]\!] \\
\&z \in [\![p]\!] & &
\end{array}
$$

menor solução

$pt(p) = \{malloc\text{–}1, y, z\}$
$pt(q) = \{malloc\text{–}1, y, z\}$

menos preciso... mas mais rápido

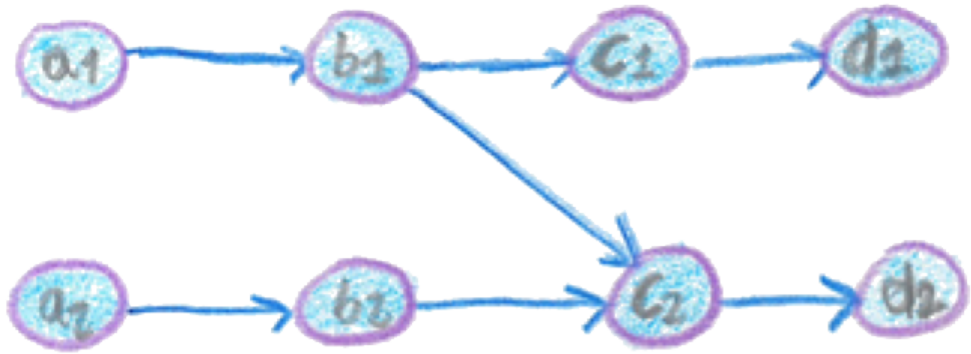
Exercício

quais são as restrições de congruência por contemplar no exemplo anterior?

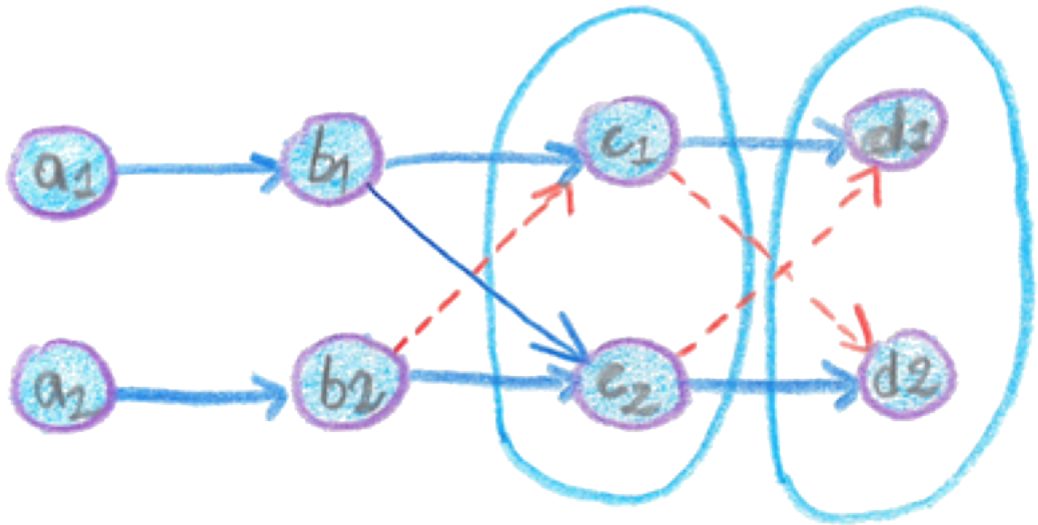
Outro exemplo

```
a1 = &b1;
b1 = &c1;
c1 = &d1;
a2 = &b2;
b2 = &c2;
c2 = &d2;
b1 = &c2;
```

Andersen:



Steensaard:



para distinguir entre diferentes apontadores para a mesma célula marcamos os constructores de apontadores com $\&_a$ em que a é uma marca fresca
unificar $\&_a\alpha$ com $\&_b\alpha$ não provoca um erro de unificação

actualizamos as restrições geradas :

$$\begin{array}{lll} x = \mathit{malloc} & \dashrightarrow & [\![x]\!] = \&_v[\![\mathit{malloc\text{–}i}]\!] \\ x = \&y & \dashrightarrow & [\![x]\!] = \&_v[\![y]\!] \\ x = y & \dashrightarrow & [\![x]\!] = \&_{v_1}\alpha \wedge [\![y]\!] = \&_{v_2}\alpha \\ x = *y & \dashrightarrow & [\![x]\!] = \&_{v_1}\alpha \wedge [\![y]\!] = \&_{v_2}\&_{v_3}\alpha \\ *x = y & \dashrightarrow & [\![x]\!] = \&_{v_1}\&_{v_2}\alpha \wedge \&_{v_3}\alpha \\ x = \mathit{null} & \dashrightarrow & \text{sem restrições} \end{array}$$

juntamos a actualização à restrição de congruência

$$\&_a\alpha_1 = \&_b\alpha_2 \implies \alpha_1 = \alpha_2$$

o mapa *points-to* é agora

$$pt(x) \triangleq \{c \mid \&_a[\![c]\!] \in [\![x]\!]\}$$

De volta ao exemplo

o programa em análise

```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

restrições geradas (sem congruência)

$$
\begin{array}{rcl}
[\![p]\!] & = & \&_1[\![malloc\text{–}1]\!] \\
[\![x]\!] & = & \&_2\alpha_1 \\
[\![y]\!] & = & \&_3\alpha_1 \\
[\![x]\!] & = & \&_4\alpha_2 \\
[\![z]\!] & = & \&_5\alpha_2 \\
[\![p]\!] & = & \&_6\&_7\alpha_3 \\
[\![z]\!] & = & \&_8\alpha_3 \\
[\![p]\!] & = & \&_9\alpha_4 \\
[\![q]\!] & = & \&_{10}\alpha_4 \\
[\![q]\!] & = & \&_{11}[\![y]\!] \\
[\![x]\!] & = & \&_{12}\alpha_5 \\
[\![p]\!] & = & \&_{13}\&_{14}\alpha_5 \\
[\![p]\!] & = & \&_{15}[\![z]\!]
\end{array}
$$

menor solução - mais uma vez....

$pt(p) = pt(q) = \{malloc\text{–}1, y, z\}$

olhando só para os apontadores e admitindo que o tipo da construção *malloc* é tipo dos apontadores para valores de tipo (variável) α (notação $\&\alpha$)
então as restrições de tipagem das construções que manipulam apontadores (que por comodidade notamos também $[\![.]\!]$) são

$$
\begin{array}{lcl}
x = \mathit{malloc} & \dashrightarrow & [\![x]\!] = \&\alpha \\
x = \&y & \dashrightarrow & [\![x]\!] = \&[\![y]\!] \\
x = y & \dashrightarrow & [\![x]\!] = [\![y]\!] \\
x = *y & \dashrightarrow & \&[\![x]\!] = [\![y]\!] \\
*x = y & \dashrightarrow & [\![x]\!] = \&[\![y]\!] \\
\&[\![t_1]\!] = \&[\![t_2]\!] & \Longrightarrow & [\![t_1]\!] = [\![t_2]\!] (\text{congruência})
\end{array}
$$

assumindo que o programa analizado se submete com sucesso à tipagem, haverá semelhanças entre a solução da fase de tipagem com a solução gerada pelo algoritmo de Steensgaard?

Análise interprocedimental de apontador

se os apontadores de funções são distintos dos apontadores para a *heap* então **executar primeiro** uma CFA **seguido de** uma análise de Andersen ou Steengaard

mas se ambos os tipos de apontadores se confundem (é o caso no TIP) então a CFA e a analise de apontador devem ser **realizadas em simultáneo**!

exempo de apontadores mixtos em TIP: `(***x)(1,2,3)`

Normalização das chamadas de função

assumimos que todas as chamadas a funções são da forma

$$x = (y)(a_1, \ldots, a_n)$$

assumimos que todas instruções de retorno de funções são da forma

$$\mathit{return}\ z;$$

todos os identificadores de função estão coletados no conjunto Loc

Exercício

mostre como realizar esta normalização de forma sistemática

em particular mostre como esta transformação obriga a uma (potencialmente volumosa) introdução de variáveis temporárias

como primeira nota, apontemos as semelhanças estruturais que a análise de Andersen tem com a análise de fluxo de controlo
no caso da chamada de função $x = (y)(a_1, \ldots, a_n)$ e das ocorrências de $f(x_1, \ldots, x_n)\{\cdots, \mathit{return}\ z;\}$
juntamos as restrições

$$\begin{array}{l} f \in [\![f]\!] \\ f \in [\![y]\!] \end{array} \implies ([\![a_i]\!] \subseteq [\![x_i]\!]\ \mathit{para}\ i = 1, \ldots, n\ \wedge\ [\![z]\!] \subseteq [\![x]\!])$$

estas restrições no caso das chamadas tradicionais são semelhantes

o conjunto de restrições pode ser resolvido com a ajuda da *framework* cúbica

no caso da chamada de função $x = (y)(a_1, \ldots, a_n)$ e das ocorrências de $f(x_1, \ldots, x_n)\{\cdots, \mathit{return}\ z;\}$
juntamos as restrições

$$\begin{array}{l} f \in [\![f]\!] \\ f \in [\![y]\!] \end{array} \implies ([\![a_i]\!]{=}[\![x_i]\!]\ \mathit{para}\ i = 1, \ldots, n\ \wedge\ [\![z]\!]{=}[\![x]\!])$$

estas restrições no caso das chamadas tradicionais são semelhantes

o conjunto de restrições pode ser resolvido aem surpresas com a ajuda da *framework* de unificação

Análise de apontador nulo

Análise de apontador nulo

pretendemos saber quando, numa dereferência `*p`, `p` pode ser (ou não) nulo

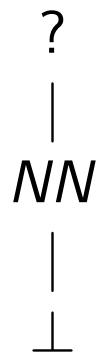
podemos usar a framework monótona, assumindo que um mapa *points-to* foi previamente calculado

porque não tratar `null` como uma localização especial nas análises de tipo Andersen ou Steensgaard

consideremos então aqui uma analise intra-procedimental (i.e. ignoramos as chamadas de funções)

Um reticulado para a análise de apontador `Null`

definimos o reticulado, designado de *Null*, seguinte



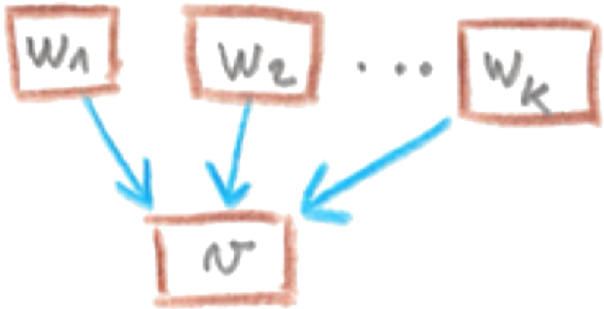
onde *NN* representa **"sem dúvida Não Nulo"**, e $\perp$ representa "não é um apontador"

usamos para cada ponto do programa o reticulado mapa $Cell \to Null$

trata-se de uma análise em processo

para cada nodo v do CFG, temos a variável $[\![v]\!]$
esta é uma mapa que atribuí valores abstactos a todas as células (de *Cell*)
no ponto de programa que segue v

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



Restrições para a análise de apontador nulo

relembramos as operações (os nodos do CFG) que involvem explicitamente apontadores

$x = malloc$
$x = \&y$
$x = y$
$x = *y$
$*x = y$
$x = null$

para os outros vértices v do CFG, as restições são definidas como

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

Restrições para a análise de apontador nulo

para uma operação `*x=y` opera/altera na *heap* (*strore operation*), precisamos de modelar as alterações na célula para onde `x` aponta (qualquer que seja ela)

são candidatas potenciais **várias** células abstractas (as células coletadas em `pt(x)`)

na abstracção em causa, cada célula abstracta da *heap*, `malloc-i` pode descrever **várias** células concretas

esta situação pode ser resolvida com o conceito de **actualização fraca** *(weak update)*
para um nodo v de tipo $*x = y$,

$$[\![v]\!] = \mathit{store}(\mathit{Join}(v), x, y)$$

onde $\mathit{store}(\sigma, x, y) \triangleq \sigma[\alpha \rightarrow \sigma(\alpha) \sqcup \sigma(y) \ \mid \ \forall \alpha \in \mathit{pt}(x)]$

Restrições para a análise de apontador nulo

para uma operação $x = *y$ que carrega vaores da *heap*, precisamos de modelar a alteração de uma variável de programa x

a nossa abstracção tem uma **única** célula abstracta para x
com a hipótese de trabalho tomada, que é realizar uma analise intra-procedimental, esta célula representa uma **única** célula concreta

neste contexto podemos nos socorrer da **actualização forte** (*strong update*) para definir as restrições associadas

para um nodo v de tipo $x = *y$,

$$[\![v]\!] = load(Join(v), x, y)$$

onde $load(\sigma, x, y) \triangleq \sigma[x \rightarrow \sqcup_{\alpha \in pt(y)} \sigma(\alpha)]$

Restrições para a análise de apontador nulo

para os restantes vértices v temos

$$
\begin{array}{lll}
x = \mathit{malloc} & \dashrightarrow & [\![v]\!] = \mathit{Join}(v)[x \rightarrow \mathit{NN}, \mathit{malloc}{-}i \rightarrow ?] \\
x = \&y & \dashrightarrow & [\![v]\!] = \mathit{Join}(v)[x \rightarrow \mathit{NN}] \\
x = y & \dashrightarrow & [\![v]\!] = \mathit{Join}(v)[x \rightarrow \mathit{Join}(v)(y)] \\
x = \mathit{null} & \dashrightarrow & [\![v]\!] = \mathit{Join}(v)[x \rightarrow ?]
\end{array}
$$

em cada caso, s atribuições modificam uma variável de programa

podemos então usar as actualizações fortes, como no caso das operações de carregamento

se assumimos que o programa analisado nunca lê de locais memórias não inicializados, podemos remover a substituição $\mathit{malloc}{-}i \rightarrow ?$ do primeiro caso

Exercício

Explique porque as quarto definições do acetato anteriores são monótonas e correctas

Actualizações fortes e fracas

actualização forte: $\sigma[c \rightarrow \mathit{novo\text{–}valor}]$

- possível se se abe de c que só aponta para uma célula única
- funciona para as atribuições para variáveis locais (na assunção da análise intraprocedimental)

actualização fraca: $\sigma[c \rightarrow \sigma(c) \sqcup \mathit{novo\text{–}valor}]$

- necessária se c refere-se a mais do que uma célula concreta
- má em termos de precisão, perde-se parte do poder do *flow-sentitivity*
- necessária no caso das atribuições na *heap* (a não ser que se estenda a abstracção usada)

Uso da análise de apontador Nulo

a de-referenciação de $*p$ é segura no vértice v se $Join(v)(p) = NN$

a qualidade desta análise depende da qualidade da análise de apontadores subjacentes

Um exemplo

```
p  = malloc;
q = &p;
n = null;
*q = n;
*p = n;
```

a análise de Andersen devolve

$$\begin{aligned} pt(p) &= \{malloc\text{–}1\} \\ pt(q) &= \{p\} \\ pt(n) &= \emptyset \end{aligned}$$

Um exemplo: restrições geradas

$$
\begin{array}{lcl}
[\![p = malloc]\!] & = & \bot[p \to NN, malloc\text{–}1 \to ?] \\
[\![q = \&p]\!] & = & [\![p = malloc]\!][q \to NN] \\
[\![n = null]\!] & = & [\![q = \&p]\!][n \to ?] \\
[\![*q = n]\!] & = & [\![n = null]\!][p \to [\![n = null]\!](p) \sqcup [\![n = null]\!](n)] \\
[\![*p = n]\!] & = & [\![* = n]\!][malloc\text{–}1 \to [\![*q = n]\!](malloc\text{–}1) \sqcup [\![*q = n]\!](n)]
\end{array}
$$

Um exemplo: menor solução

$$
\begin{array}{lcl}
[\![p = malloc]\!] & = & [p \to NN, q \to \bot, n \to \bot, malloc\text{–}1 \to ?] \\
[\![q = \&p]\!] & = & [p \to NN, q \to NN, n \to \bot, malloc\text{–}1 \to ?] \\
[\![n = null]\!] & = & [p \to NN, q \to NN, n \to ?, malloc\text{–}1 \to ?] \\
[\![*q = n]\!] & = & [p \to ?, q \to NN, n \to ?, malloc\text{–}1 \to ?] \\
[\![*p = n]\!] & = & [p \to ?, q \to NN, n \to ?, malloc\text{–}1 \to ?]
\end{array}
$$

no ponto do programa antes de $*q = n$, a analise sabe que q é definitivamente não nulo
antes de $*p = n$, o apontador p pode ser nulo

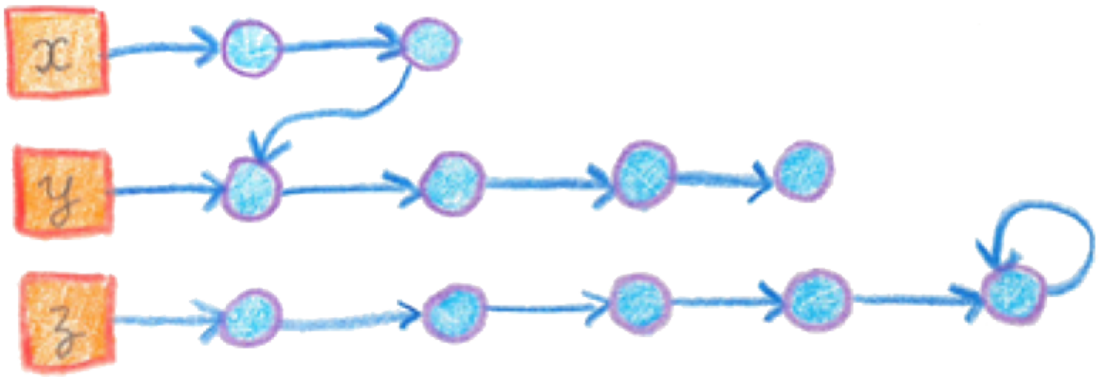
devido as actualizações fracas realizadas em todas as operações de arquivo (*store*) na *heap*, a precisão é má para as localizações *malloc–i*

Análise de forma

A *heap* como estrutura de primeira classe nas análises

a *heap* tem uma influência notória no comportamento de programas com apontadores
em particular a sua segmentação e a evolução desta segmentação no curso da execução é relevante
por exemplo saber decidir se apontadores evoluem em zonas distintas da memória (e.g. *heap*) tem impacto na análise comportamental de um programa

consideremos por exemplo esta estrutura não trivial de uma *heap*



é relevante poder decidir **as zonas de disjunção** da *heap*

- x e y não são disjuntos
- y e z são disjuntos

são grafos que descrevem possíveis *heap*

- vértices são células abstractas
- arestas são possíveis apontadores entre células

o reticulado de grafos de forma é

$$L \triangleq (\mathcal{P}(Cell \times Cell), \subseteq)$$

cada vértice v do CFG introduzimos uma variável de restrição $[\![v]\!]$ que descreve o estado **após** v

por enquanto, olhemos para uma análise intra-procedimental (i.e. ignoramos as chamadas de função)

tendo em conta que para um vértice v,

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

as restrições para operações que lidam com apontadores são

$$\begin{array}{lcl}
x = malloc & \dashrightarrow & [\![v]\!] = Join(v) \downarrow_x \cup \{(x, malloc\text{–}i)\} \\
x = \&y & \dashrightarrow & [\![v]\!] = Join(v) \downarrow_x \cup \{(x, y)\} \\
x = y & \dashrightarrow & [\![v]\!] = assign(Join(v), x, y) \\
x = *y & \dashrightarrow & [\![v]\!] = load(Join(v), x, y) \\
*x = y & \dashrightarrow & [\![v]\!] = store(Join(v), x, y) \\
x = null & \dashrightarrow & [\![v]\!] = Join(v) \downarrow_x
\end{array}$$

para todos os outros vértices v do CFG, $[\![v]\!] = Join(v)$

onde

$$
\begin{array}{lcl}
\sigma \downarrow_x & = & \{(s,t) \in \sigma \mid s \neq x\} \\
\mathit{assign}(\sigma, x, y) & = & \sigma \downarrow_x \cup \{(x,t) \mid (y,t) \in \sigma\} \\
\mathit{load}(\sigma, x, y) & = & \sigma \downarrow_x \cup \{(x,t) \mid (y,s) \in \sigma, (s,t) \in \sigma\} \\
\mathit{store}(\sigma, x, y) & = & \sigma \cup \{(s,t) \mid (x,s) \in \sigma, (y,t) \in \sigma\}
\end{array}
$$

de notar que a operação *store* utiliza **actualização fraca**

Exemplo

```
var x,y,n,p,q;
x = malloc;
y = malloc;
*x = null;
*y = y;
n = input;
while (n>0) {
   p = malloc;
   q = malloc;
   *p = x; *q = y;
   x = p; y = q;
   n = n-1;
}
```

com base na análise de forma, obtemos o grafo seguinte após o ciclo

concluimos que x e y serão sempre distintos

De mapas *point-to* para grafos de forma

a analise de forma é também uma análise *points-to* sensível ao fluxo visto que

$$pt(x) = \{t \mid (x, t) \in [\![v]\!]\}$$

é um mapa *points-to*

mais custos mais também mais preciso:

```
x=&y;
x=&z;
// <---- aqui
```

no ponto assinalado

- análise de Andersen: $pt(x) = \{y, z\}$
- análise de forma : $pt(x) = \{z\}$

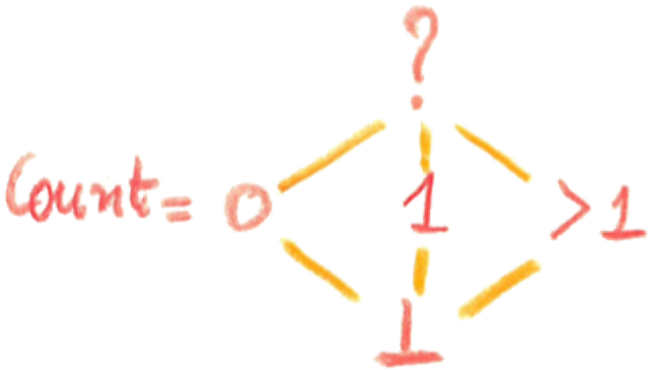
podem ser combinadas, usando a análise de Andersen para a construção do CFG e utilizando a análise de forma para melhorar a análise de apontador nulo

a análise de forma está a falhar na recolha de informação: o vértice *malloc*–2 forma sistematicamente um ciclo para ele próprio no exemplo

para lidar com este tipo de fenómenos, temos de usar um reticulado mais informativo

$$\mathcal{P}(Cell \times Cell) \times (Cell \to Count)$$

onde se guarda para cada célula o registo de quantas células concretas cada célula abstracta descreve



esta informação permite **actualizações fortes** nas células que descrevem **exactamente** uma célula concreta

defina o conjunto novo de restrições associado a este reticulado e a esta análise de forma para cada nodo ($x = \mathit{malloc}$, $*x = y$, etc.) do CFG

com o exemplo anterior e após o ciclo obtemos então o grafo de forma seguinte

já se consegue neste caso perceber que o vértice *malloc*–2 forma sistematicamente um ciclo para ele próprio

para extender a análise no contexto interprocedimental é necessário considerar

- passagem de parâmetros
- actualização fraca de células localizadas na pilha (**frame/operand stack**)
- o fenómeno de fuga de células da pilha (**escape of stack cells**)

Análise de escape - *escape analysis*

para lidar com o referido fenómeno podemos proceder da seguinte forma

realizar uma análise de forma

olhar para a expressão de retorno

avaliar no grafo de forma a acessibilidade (*reachability*) dos argumentos ou variáveis definidas na função em si

se nenhuma desses está acessível , então não há fuga de células de pilha

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz();
  *p=1;
  return *p;
}
```

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*!).