

Optimizatie – verbetering van de code

- Verbeteringen: overbodige berekeningen, assignments, stukken code weglaten, operaties vervangen door meer efficiënte versies,...
- Door **lokale** transformaties, waarbij slechts een klein deel van het programma betrokken is
- Abstractieniveau: eenvoudige versie van intermediaire code (P-machine code) – zie definitie van de P-machine instructies
- Noodzaak van formele (operationele) semantiek en andere formele methoden: de programma-transformaties mogen de betekenis van het programma **nooit** veranderen!

Abstractieniveau: eenvoudige versie van intermediaire code (P-machine code) – zie definitie van de P-machine instructies

- variables : x
- arithmetic expressions : e
- assignments : $x \leftarrow e$
- reading access to memory : $x \leftarrow M[e]$
- writing access to memory : $M[e_1] \leftarrow e_2$
- conditional statement : $\text{if}(e) s_1 \text{ else } s_2$
- unconditional jump : $\text{goto } L$

Nog geen machinetaal: die heeft meer gespecialiseerde instructies (adresseermogelijkheden), registers, geen gewone variabelen,...

Voorbeeld: swap

```
void swap ( int i, int j) {  
    int t;  
    if (a[i] > a[j]) {  
        t ← a[j];  
        a[j] ← a[i];  
        a[i] ← t;  
    }  
}
```

Adresberekeningen gebeuren meermaals; beter is:

```
void swap (int * p, int * q) {  
    int t, ai, aj;  
    ai ← *p; aj ← *q;  
    if (ai > aj) {  
        t ← aj;  
        *q ← ai;  
        *p ← t;  
    }  
}
```

Gecompileerde versie:

```
A1 ← A0 + 1 * i;           //    A0 = &a[0]
R1 ← M[A1];                //    R1 = a[i]
A2 ← A0 + 1 * j;
R2 ← M[A2];                //    R2 = a[j]
if (R1 > R2) {
    A3      ← A0 + 1 * j;
    t       ← M[A3];
    A4      ← A0 + 1 * j;
    A5      ← A0 + 1 * i;
    R3      ← M[A5];
    M[A4] ← R3;
    A6      ← A0 + 1 * i;
    M[A6] ← t;
}
```

//

Evident is

$$1 * x = x$$

Bovendien $A_1 = A_5 = A_6$ en $A_2 = A_3 = A_4$

en $M[A_1] = M[A_5]$, $M[A_1] = M[A_5]$,

dus ook $R_1 = R_3$, $R_2 = t$

en er blijft:

```
 $A_1 \leftarrow A_0 + i;$ 
 $R_1 \leftarrow M[A_1];$ 
 $A_2 \leftarrow A_0 + j;$ 
 $R_2 \leftarrow M[A_2];$ 
if ( $R_1 > R_2$ ) {
     $M[A_2] \leftarrow R_1;$ 
     $M[A_1] \leftarrow R_2;$ 
}
```

Vergelijking van de benodigde operaties in de twee versies:

	Before	After
+	6	2
*	6	0
load	4	2
store	2	2
>	1	1
←	6	2

Mogelijke optimizaties

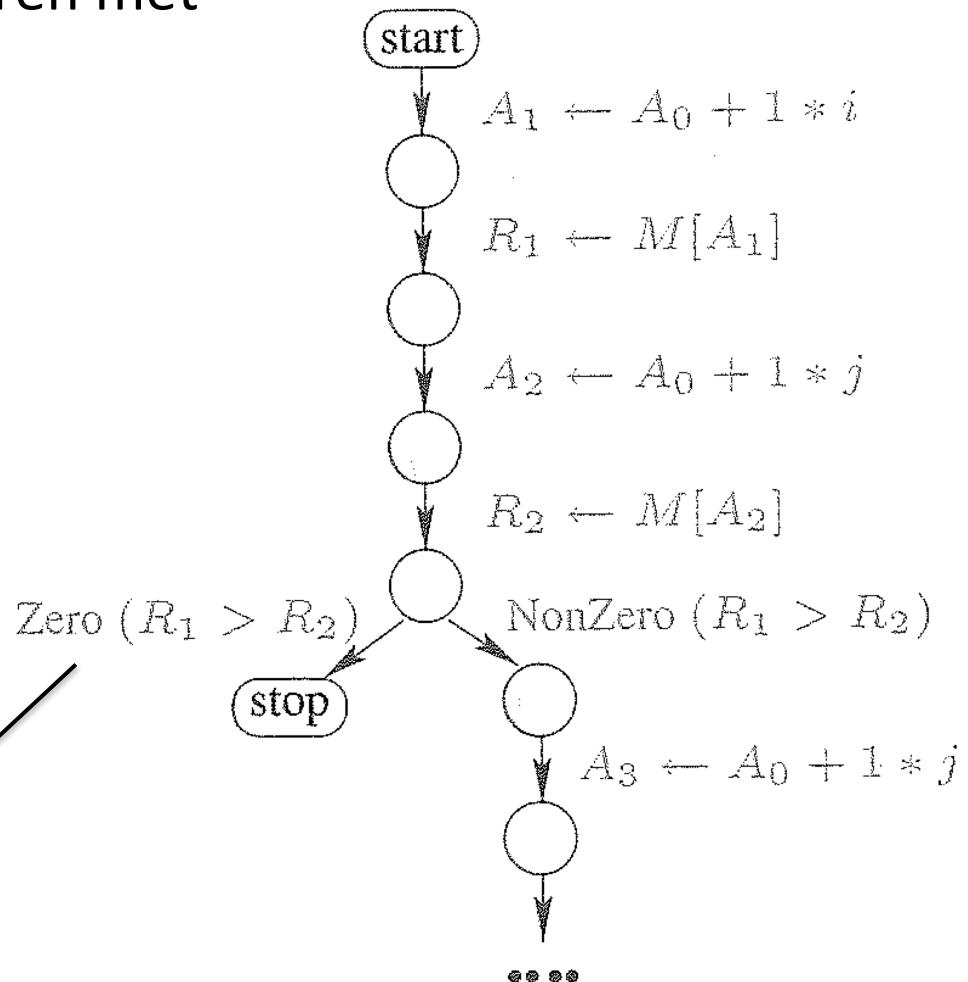
- Constant folding: detectie van expressies waarvan de waarde al op compile-time vastligt (partial compilation).
- Reduction in strength: vervangen van operaties door goedkoper equivalent, bv. $2*a$ door $a + a$ of $a + 0$ door a
- Inlining
- Hergebruik van assignments
- Dead code eliminatie
- Alias analyse
- Loop-invariante code detecteren en buiten loop brengen
- Interprocedurele analyse (tail-calls,...)

Representatie: de flowgraph

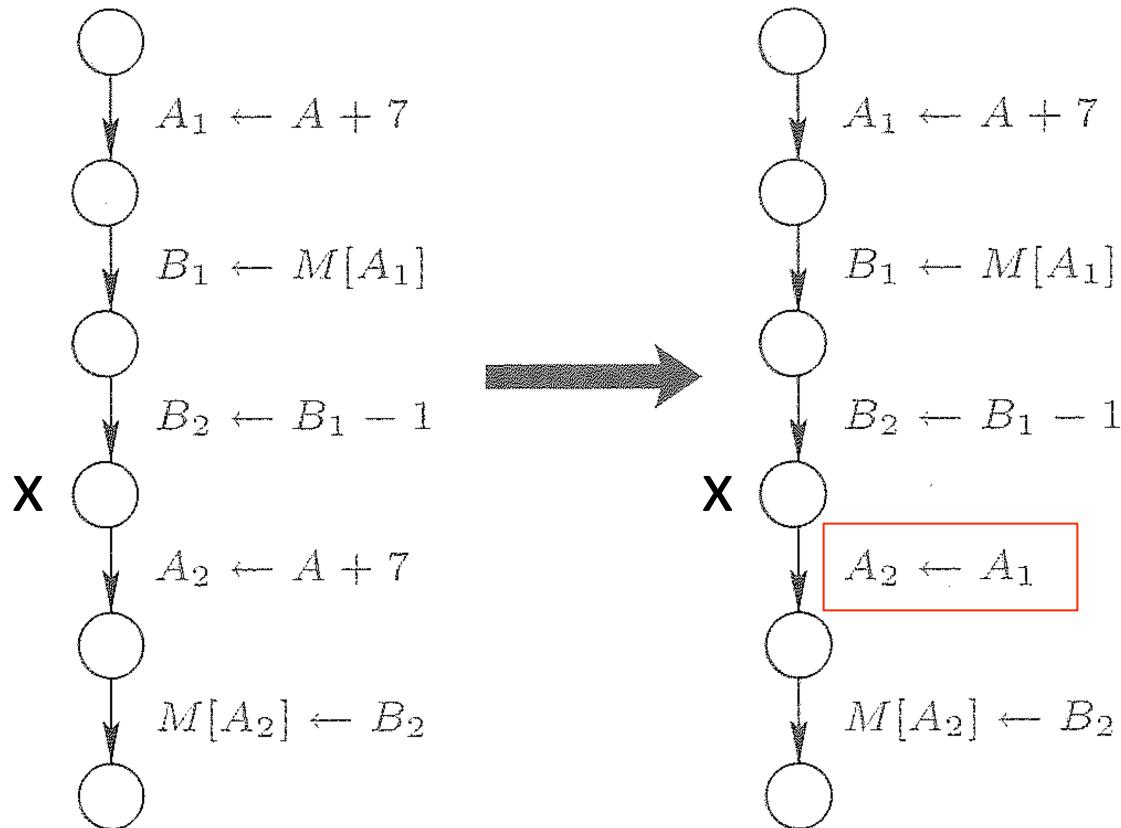
- Knopen: posities voor en na een instructie
 - Pijlen: opeenvolgende posities, met instructie als label
 - Eenvoudig te construeren met behulp van de AST
(zie Syntax-directed translation)

Conditional:

Zero = “yes”, NonZero = “no”

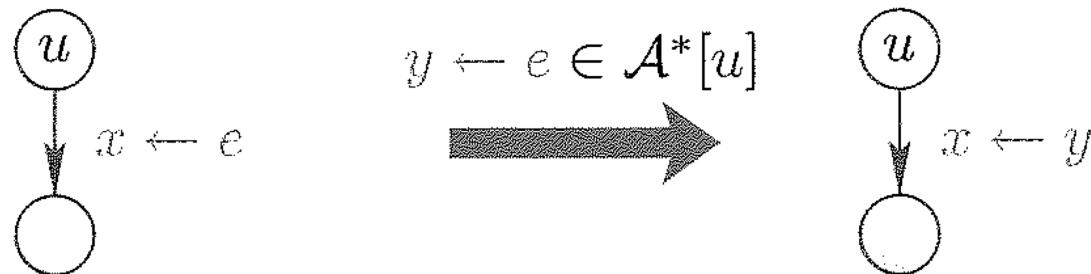


Hergebruik van assignments



Assignment $A_1 \leftarrow A + 7$ is beschikbaar op punt x

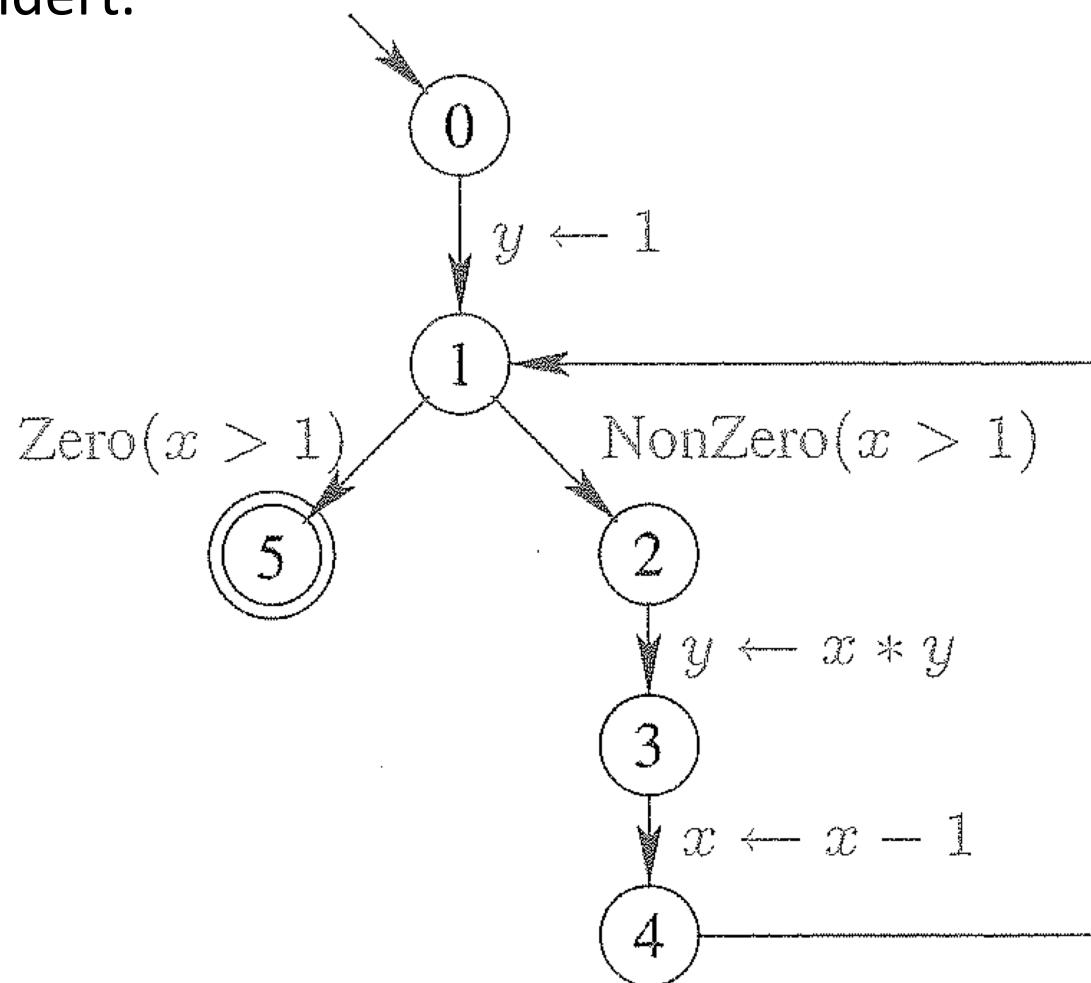
De transformatie:



Waar $\mathcal{A}^*[u]$ de set is van beschikbare assignments in punt u

Het komt er dus op aan die set te berekenen voor elke u

Algemene idee: ga na hoe de sets veranderen bij uitvoeren van een instructie (= doorlopen van een pijl in de flowgraph) en propageer de veranderingen tot er niets meer verandert.



$x \leftarrow y + 3;$

$x \leftarrow 7;$

$z \leftarrow y + 3;$

$x \leftarrow y + 3$ beschikbaar

$x \leftarrow y + 3$ niet meer beschikbaar

We kijken enkel naar de beschikbaarheid van assignments waarin geen enkele variabele zowel links als rechts voorkomt

Definieer een abstracte versie van de operationele semantiek, genoteerd $\llbracket \cdot \rrbracket^\sharp$, die werkt op sets van beschikbare assignments

$$\llbracket ; \rrbracket^\sharp A = A$$

$$\llbracket \text{NonZero}(e) \rrbracket^\sharp A = \llbracket \text{Zero}(e) \rrbracket^\sharp A = A$$

$$\llbracket x \leftarrow e \rrbracket^\sharp A = \begin{cases} (A \setminus \text{Occ}(x)) \cup \{x \leftarrow e\} & \text{if } x \notin \text{Vars}(e) \\ A \setminus \text{Occ}(x) & \text{otherwise} \end{cases}$$

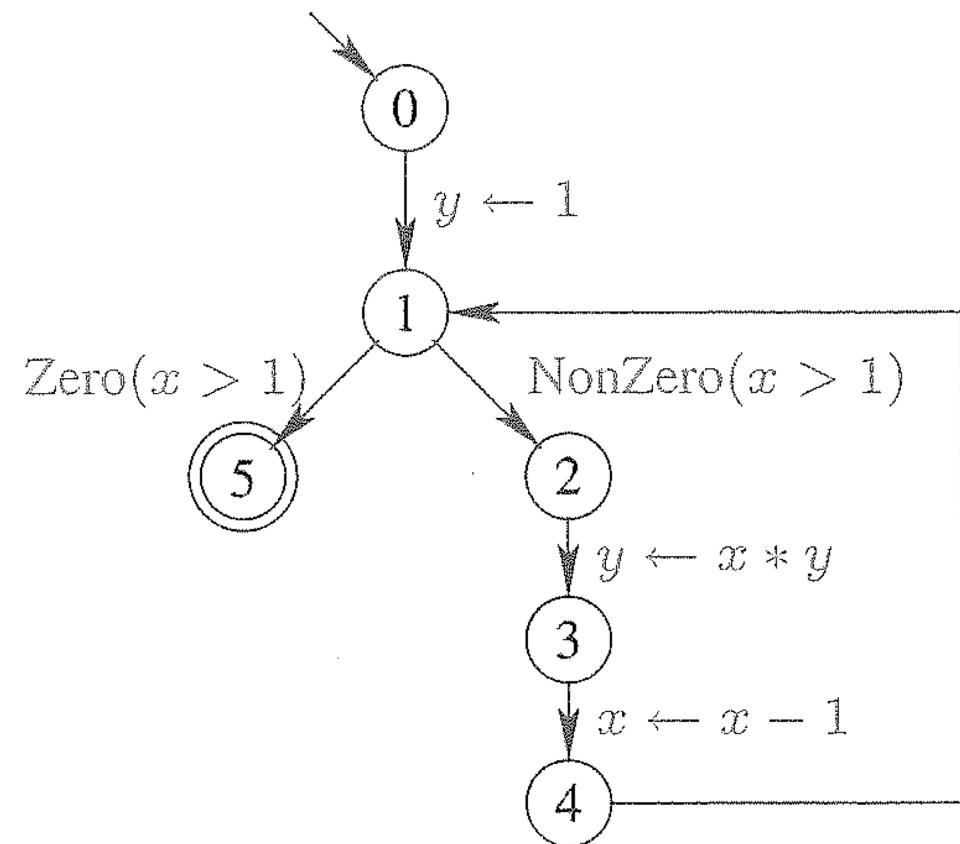
$$\llbracket x \leftarrow M[e] \rrbracket^\sharp A = A \setminus \text{Occ}(x)$$

$$\llbracket M[e_1] \leftarrow e_2 \rrbracket^\sharp A = A$$

$\text{Occ}(x)$: de assignments waarin x voorkomt

$\text{Vars}(e)$: de variabelen die voorkomen in e

Toepassing op het voorbeeld:



$$\mathcal{A}[0] \subseteq \emptyset$$

$$\mathcal{A}[1] \subseteq (\mathcal{A}[0] \setminus \text{Occ}(y)) \cup \{y \leftarrow 1\}$$

$$\mathcal{A}[1] \subseteq \mathcal{A}[4]$$

$$\mathcal{A}[2] \subseteq \mathcal{A}[1]$$

$$\mathcal{A}[3] \subseteq \mathcal{A}[2] \setminus \text{Occ}(y)$$

$$\mathcal{A}[4] \subseteq \mathcal{A}[3] \setminus \text{Occ}(x)$$

$$\mathcal{A}[5] \subseteq \mathcal{A}[1]$$

Inclusie, geen gelijkheid!

Je kan dit ook schrijven als

$$\mathcal{A}[start] \subseteq \emptyset$$

$$\mathcal{A}[v] \subseteq \bigcap\{\llbracket k \rrbracket^\sharp(\mathcal{A}[u]) \mid k = (u, lab, v) \text{ edge}\} \text{ for } v \neq start$$

en ook als

$X \subseteq F(X)$, waar $X = (x_0, x_1, \dots, x_n)$, $n+1$ is het aantal knopen en

$$F(x_0, x_1, \dots, x_n) = (f_0(x_0, x_1, \dots, x_n), \dots, f_n(x_0, x_1, \dots, x_n))$$

Verder zijn de functies $\llbracket k \rrbracket^\sharp$ en dus ook de f_i en F ,
monotoon (bewaren de partiële orde)

We zoeken **maximale** sets $A[0], A[1], \dots$ zo dat geldt, voor alle knopen en pijlen:

$$\mathcal{A}[start] \subseteq \emptyset$$

$$\mathcal{A}[v] \subseteq \llbracket k \rrbracket^\sharp(\mathcal{A}[u]) \quad \text{for an edge } k = (u, lab, v)$$

Daarvoor bestaat een algemene methode, gebaseerd op **complete lattices** en de (Knaster – Tarski) **fixpuntstelling**.

We zorgen ervoor dat de set van mogelijke (abstracte) waarden die we toekennen aan de programma-posities (in ons geval dus sets van beschikbare assignments) een complete lattice vormt.

Een **complete lattice** is een set D uitgerust met een partiële orde, die zo is dat elke deelverzameling X van D een **least upper bound** heeft

Partiële orde: een binaire relatie met de eigenschappen

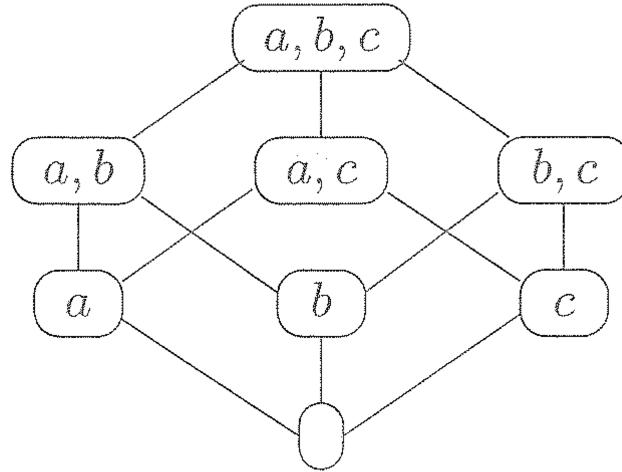
$$\begin{array}{ll} a \sqsubseteq a & \textit{reflexivity} \\ a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b & \textit{antisymmetry} \\ a \sqsubseteq b \wedge b \sqsubseteq c \implies a \sqsubseteq c & \textit{transitivity} \end{array}$$

a is upper bound van X als $x \sqsubseteq a$ voor elke x in X

Least upper bound: een upper bound b zo dat voor elke andere upper bound a geldt dat $b \sqsubseteq a$

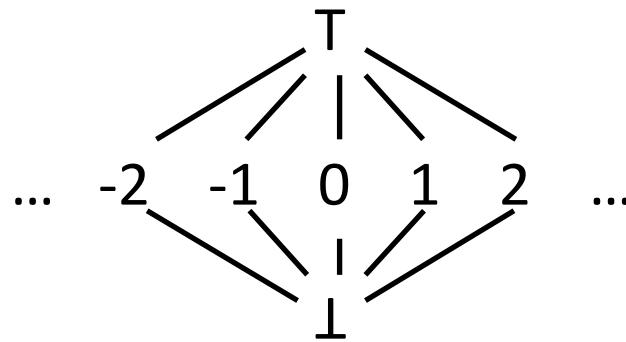
Voorbeelden

1. De set van alle subsets van $\{a,b,c\}$ met \subseteq :



lub nemen = unie nemen

2. Gehele getallen \mathbb{Z} , met Top (T) en Bottom (\perp) :



Bestaat altijd!

Stellingen:

1. Elke subset X van een complete lattice D heeft een greatest lower bound.
2. Laat $D \sqsubseteq$ een complete lattice zijn en $f: D \rightarrow D$ een monotone functie. Dan is de rij $f(\perp), f^2(\perp), f^3(\perp) \dots$ stijgend, en als $f^{n-1}(\perp) = f^n(\perp)$ dan is $f^n(\perp)$ het kleinste element d zo dat $f(d) \sqsubseteq d$
3. (Knaster – Tarski) Laat $D \sqsubseteq$ een complete lattice zijn en $f: D \rightarrow D$ een monotone functie. Dan heet f een kleinste fixpunt d_0 en dat is gelijk aan de kleinste oplossing van $f(x) \sqsubseteq x$
(fixpunt: oplossing van $f(x) = x$)

Toepassing op het probleem van de beschikbare assignments:

We hebben voor elk punt in het programma de **grootste** set nodig die aan de voorwaarden voldoet. Dat moet de **kleinste** oplossing zijn voor de gekozen partiële orde \subseteq . Dus kiezen we voor \subseteq de superset relatie \supseteq . (Dus niet \subseteq) Het “bottom” element is dus de hele set van mogelijk beschikbare assignments. (Dus niet de lege set) In het voorbeeld bestaat die set uit één assignment: $y \leftarrow 1$. Uit de stellingen volgt dat we een algoritme hebben om die oplossing te berekenen: begin met bottom en pas f telkens opnieuw toe tot de bekomen sets (voor elk punt) niet meer veranderen.

	1	2	3	4	5
0	\emptyset	\emptyset	\emptyset	\emptyset	
1	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	\emptyset	\emptyset	
2	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	\emptyset	
3	\emptyset	\emptyset	\emptyset	\emptyset	ditto
4	$\{y \leftarrow 1\}$	\emptyset	\emptyset	\emptyset	
5	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	\emptyset	

Efficienter berekening: gebruik telkens de laatst berekende waarde (round robin)

	1	2	3
0	\emptyset	\emptyset	
1	$\{y \leftarrow 1\}$	\emptyset	
2	$\{y \leftarrow 1\}$	\emptyset	
3	\emptyset	\emptyset	ditto
4	\emptyset	\emptyset	
5	\emptyset	\emptyset	

Verwijderen van assignments aan dode variabelen

Een variabele is dood op een bepaald punt als hij na dat punt niet gebruikt wordt voor er opnieuw een waarde aan toegekend wordt.

De waarden toegekend aan de knopen zijn uiteraard sets van dode variabelen. Die sets vormen weer een complete lattice.

Keuze van \sqsubseteq ?

Dit is een voorbeeld van **achterwaardse** (backward) analyse. Men keert dus a.h.w. de richting van de pijlen om, en men vertrekt van een set X van variabelen die op het eind nog “**live**” zijn: ze worden in het vervolg van het programma nog gebruikt.

De functies voor de abstracte interpretatie?

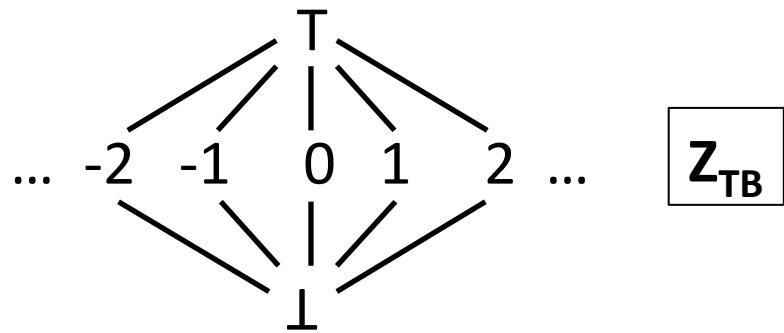
- variables : x
- arithmetic expressions : e
- assignments : $x \leftarrow e$
- reading access to memory : $x \leftarrow M[e]$
- writing access to memory : $M[e_1] \leftarrow e_2$
- conditional statement : $\text{if}(e) s_1 \text{ else } s_2$
- unconditional jump : $\text{goto } L$

Constant Folding: detecteren welke variabelen een constante waarde hebben

Constant Folding: detecteren welke variabelen een constante waarde hebben (stel: enkel integer variabelen)

Keuze van complete lattice:

Maps van Variabelen op \mathbf{Z}_{TB} : gehele getallen, met bovendien een Top en een Bottom element



$A \sqsubseteq B$ als in B elke variabele tenminste “even bepaald” is als in A
 \perp = nog geen waarde, i = heeft 1 waarde gekregen,
 T = heeft meerdere waarden gekregen, en varieert dus.

De functies

$$\llbracket ; \rrbracket^\sharp D = D$$

$$\llbracket \text{NonZero}(e) \rrbracket^\sharp D = \begin{cases} \perp & \text{if } 0 = \llbracket e \rrbracket^\sharp D \\ D & \text{otherwise} \end{cases}$$

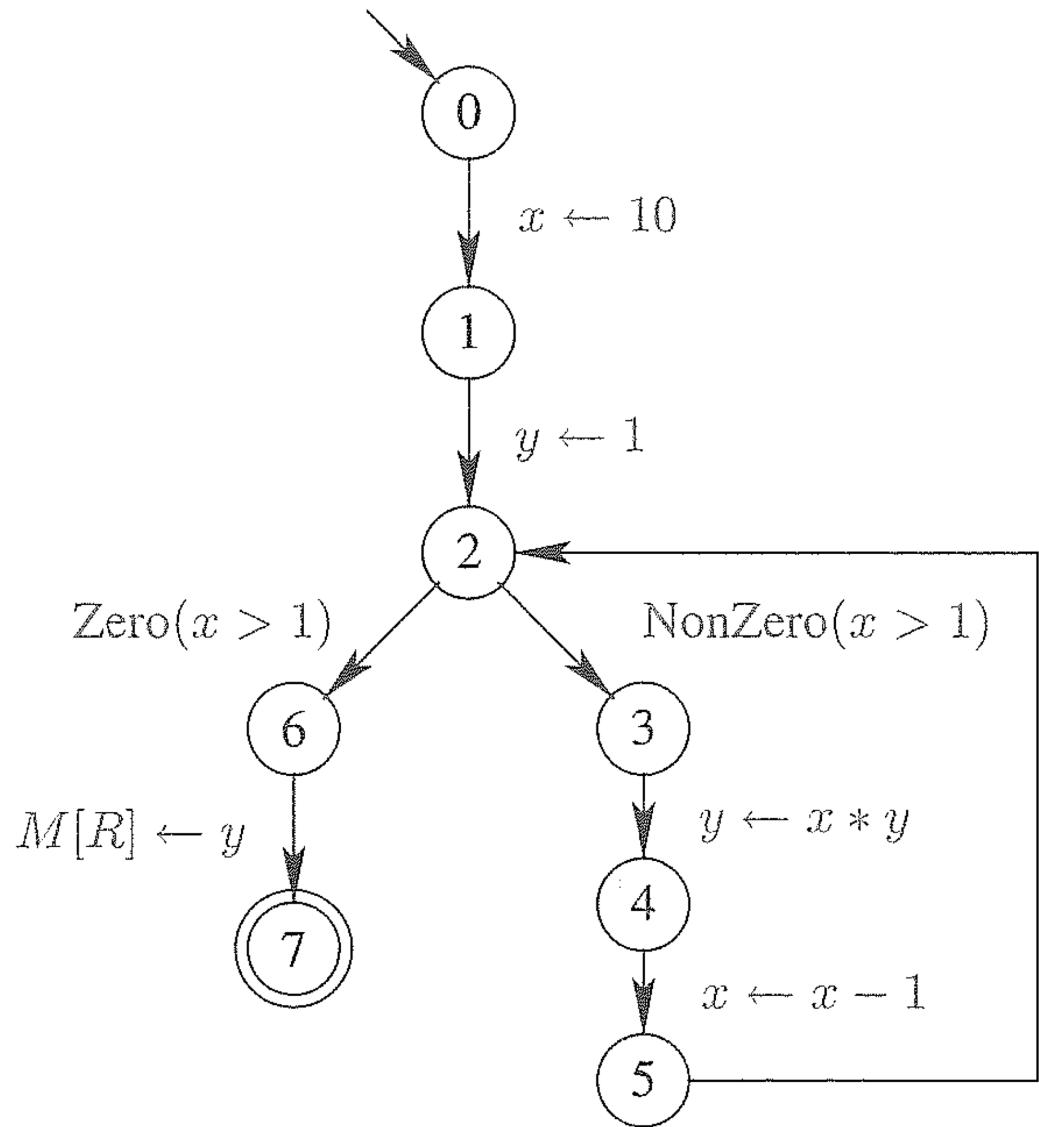
$$\llbracket \text{Zero}(e) \rrbracket^\sharp D = \begin{cases} \perp & \text{if } 0 \not\sqsubseteq \llbracket e \rrbracket^\sharp D \\ D & \text{if } 0 \sqsubseteq \llbracket e \rrbracket^\sharp D \end{cases}$$

cannot be zero
could be zero

$$\llbracket x \leftarrow e \rrbracket^\sharp D = D \oplus \{x \mapsto \llbracket e \rrbracket^\sharp D\}$$

$$\llbracket x \leftarrow M[e] \rrbracket^\sharp D = D \oplus \{x \mapsto \top\}$$

$$\llbracket M[e_1] \leftarrow e_2 \rrbracket^\sharp D = D$$



	1		2		3	
	x	y	x	y	x	y
0	T	T	T	T	T	
1	10	T	10	T		
2	10	1	T	T		
3	10	1	T	T		
4	10	10	T	T		
5	9	10	T	T		
6	⊥		T	T		
7	⊥		T	T		

ditto