Principles of Abstract Interpretation
MIT press

Ch. 41, Dataflow Analysis

Patrick Cousot

pcousot.github.io

PrAbsInt@gmail.com   github.com/PrAbsInt/

These slides are available at
http://github.com/PrAbsInt/slides/slides-41--data-flow-analysis-PrAbsInt.pdf

Chapter 41

Ch. 41, Dataflow Analysis

(Classic) data flow analysis

- Dataflow analysis: at each point of a computer program gather information about how variables and/or expressions are used and modified
- Method:
  - Sets (of variables, expressions, etc.) are finite and represented by bit vectors
  - Program → control flow graph → equations → fixpoint computation → solution
- Mainly used for compilation
- Examples:
  - Reaching definitions: which assignments to x reach a use of a variable y in an expression at a program point without another assignments to x in between
  - Available expressions: which expressions previously computed expressions need not be recomputed at a program point (since its variables are unchanged)
  - Live variable analysis: which variables hold a value that may be used later
  - …

Soundness of a data flow analysis

- Originally postulated equationally [Allen & Cocke, NYU/IBM, 70's]
- Postulated on paths in the control flow graph to justify the equations [Kildall, 1973]
- The dataflow analysis on a path can be specified by a temporal logic formula (considering the control flow graph as a transition system) [Steffen, 1991, 1993]
- The control flow chart is an abstract interpretation of the program semantics [Schmidt, 1998]

→ The soundness is not with respect to the program semantics but with respect to an abstraction of the program semantics (i.e. the control flow chart)

→ This is problematic

- We study a correct alternative: soundness by abstract interpretation with respect to a semantics [P. Cousot and R. Cousot, 1979, Section 7.2.0.6.3 *Justifying the Data Flow Equations of "Available Expressions"*]

Live and dead variables analysis

- A variable x is live at a program point on a program path iff it may be used before being modified.
- A variable x is dead at a program point on a program path iff it will definitely be modified before being used

ℓ_1 $\longleftarrow$ x and y dead (modified in ℓ_1 and ℓ_2 before being used)

$\quad$ x = 1 ;

ℓ_2 $\longleftarrow$ x live (used in expression at ℓ_2) and y dead (modified in ℓ_2 before use)

$\quad$ y = x ;

ℓ_3 $\longleftarrow$ x dead (modified in ℓ_3 before use) and y live (used in expression at ℓ_3)

$\quad$ x = y + 1 ;

ℓ_4 $\longleftarrow$ x and y live (used in assigned expression at ℓ_4)

$\quad$ x = x − y ;

ℓ_5 $\longleftarrow$ x live and y dead (by hypothesis)

$L_e = \{\mathtt{x}\}$ $\longleftarrow$ hypothesis that x is live and y is dead on statement exit.

Liveness, informally

"a variable is potentially/definitely live at some program point ℓ if it holds a value that may/must be used in the future before the next time the variable is modified".

The liveness abstraction of a trace

$\downarrow$ This l is for "liveness"

$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0\ell,\ \ell\pi\rangle$

- ℓ is the program label $\ell = \mathsf{at}[\![\mathsf{S}]\!]$
- $\pi_0\ell$ is an initial trace of the program component S arriving $\ell = \mathsf{at}[\![\mathsf{S}]\!]$
- $\ell\pi$ continues the initial trace π_0 from $\ell = \mathsf{at}[\![\mathsf{S}]\!]$ on
- use defines the set $use[\![\mathsf{a}]\!]\rho$ of variables which value is used when executing action a in environment ρ
- mod defines the set $mod[\![\mathsf{a}]\!]\rho$ of variables which value is modified when executing action a in environment ρ.
- L_b is the set of variables live on exit of S by a **`break ;`**, if any
- L_e is the set of variables live on exit of S by a normal exit, if ever

$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0\ell,\ \ell\pi\rangle$ is the set of variables live at ℓ in $\ell\pi$ continuing $\pi_0\ell$.

The liveness abstraction of a trace, formally & recursively

$$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\, L_b, L_e\, \langle \pi_0,\ \ell \rangle \quad \triangleq \quad \{\mathsf{x} \in \mathbb{V} \mid (\ell = \mathsf{after}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_e) \vee \qquad (\mathsf{a}_1) \qquad (41.2)$$

$$(\mathsf{escape}[\![\mathsf{S}]\!] \wedge \ell = \mathsf{break\text{-}to}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_b)\} \qquad (\mathsf{a}_2)$$

$$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\, L_b, L_e\, \langle \pi_0,\ \ell \xrightarrow{\mathsf{a}} \ell' \pi_1 \rangle \quad \triangleq \quad \{\mathsf{x} \in \mathbb{V} \mid \mathsf{x} \in use[\![\mathsf{a}]\!]\boldsymbol{\varrho}(\pi_0) \vee \qquad (\mathsf{b}_1)$$

$$(\mathsf{x} \notin mod[\![\mathsf{a}]\!]\boldsymbol{\varrho}(\pi_0) \wedge \mathsf{x} \in \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\, L_b, L_e\, \langle \pi_0 \frown \ell \xrightarrow{\mathsf{a}} \ell',\ \ell' \pi_1 \rangle)\} \qquad (\mathsf{b}_2)$$

The liveness abstraction of a trace, formally & recursively

$$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell\rangle \triangleq \{\mathsf{x} \in \mathbb{V} \mid (\ell = \mathsf{after}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_e) \vee \qquad (\mathsf{a}_1) \qquad (41.2)$$

$$(\mathsf{escape}[\![\mathsf{S}]\!] \wedge \ell = \mathsf{break\text{-}to}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_b)\} \qquad (\mathsf{a}_2)$$

$$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell \xrightarrow{\mathsf{a}} \ell'\pi_1\rangle \triangleq \{\mathsf{x} \in \mathbb{V} \mid \mathsf{x} \in \mathit{use}[\![\mathsf{a}]\!]\varrho(\pi_0) \vee \qquad (\mathsf{b}_1)$$

$$(\mathsf{x} \notin \mathit{mod}[\![\mathsf{a}]\!]\varrho(\pi_0) \wedge \mathsf{x} \in \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0 \frown \ell \xrightarrow{\mathsf{a}} \ell',\ \ell'\pi_1\rangle)\} \qquad (\mathsf{b}_2)$$

ℓ_1 $\longleftarrow$ $\neg(\mathsf{b}_1) \wedge \neg(\mathsf{b}_2)$ x and y dead (modified in ℓ_1 and ℓ_2 before being used)

x = 1 ;

ℓ_2 $\longleftarrow$ (b_2) x live (used in expression at ℓ_3) and $\neg(\mathsf{b}_1) \wedge \neg(\mathsf{b}_2)$ y dead (modified in ℓ_2 before use)

y = 1 ;

ℓ_3 $\longleftarrow$ (b_1) x live (used in ℓ_3) and (b_2) y live (not modified and used in expression at ℓ_4)

x = x + 1 ;

ℓ_4 $\longleftarrow$ (b_1) x and (b_1) y live (used in assigned expression at ℓ_4)

x = x - y ;

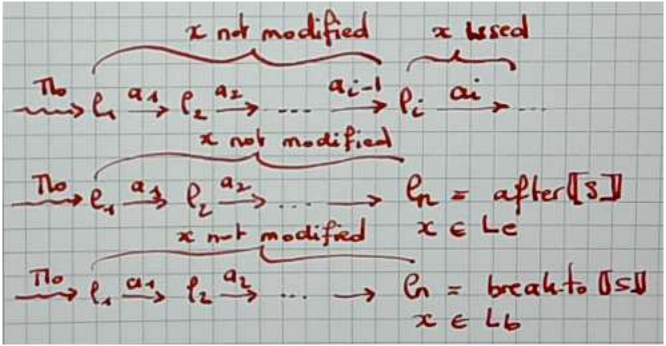
ℓ_5 $\longleftarrow$ (a_1) x live and $\neg(\mathsf{a}_1) \wedge \neg(\mathsf{a}_2)$ y dead (by hypothesis)

$L_e = \{\mathsf{x}\}$ $\longleftarrow$ hypothesis that x is live and y is dead on statement exit.

Potential and definite liveness of a statement, formally & iteratively

Lemma 1 If $\pi_1 = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} \ell_n$ and $\langle \pi_0,\ \pi_1 \rangle \in \mathcal{S}^*[\![\mathsf{S}]\!]$ then

$$
\begin{aligned}
\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle \pi_0,\ \pi_1 \rangle \quad = \quad & \{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, n-1] \,.\, \forall j \in [1, i-1] \,. \\
& \mathsf{x} \notin \mathit{mod}[\![\mathsf{a}_j]\!]\varrho(\pi_0 \frown \ell_1 \xrightarrow{\mathsf{a}_1} \ell_2 \dots \xrightarrow{\mathsf{a}_{j-1}} \ell_j) \wedge \mathsf{x} \in \mathit{use}[\![\mathsf{a}_i]\!]\varrho(\pi_0 \frown \ell_1 \xrightarrow{\mathsf{a}_1} \ell_2 \dots \xrightarrow{\mathsf{a}_{i-1}} \ell_i)\} \\
& \cup \left(\!\left| \ \ell_n = \mathsf{after}[\![\mathsf{S}]\!] \ ?\ L_e \ ⁏\ \varnothing \ \right|\!\right) \cup \left(\!\left| \ \mathsf{escape}[\![\mathsf{S}]\!] \wedge \ell_n = \mathsf{break\text{-}to}[\![\mathsf{S}]\!] \ ?\ L_b \ ⁏\ \varnothing \ \right|\!\right). \qquad \square
\end{aligned}
$$



Potential and definite liveness of a statement, formally

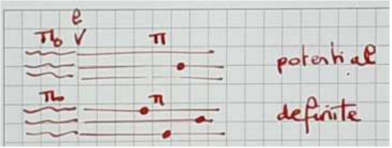
The potential and definite liveness are abstractions of the maximal trace semantics $\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathsf{S}]\!]$ is by merge over all traces

$$\alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ L_b, L_e \quad \triangleq \quad \bigcup_{\langle\pi_0,\,\pi\rangle\,\in\,\boldsymbol{\mathcal{S}}} \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi\rangle \qquad \text{potential liveness} \qquad (41.3)$$

$$\alpha^{\forall l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ L_b, L_e \quad \triangleq \quad \bigcap_{\langle\pi_0,\,\pi\rangle\,\in\,\boldsymbol{\mathcal{S}}} \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi\rangle \qquad \text{definite liveness} \qquad (41.4)$$

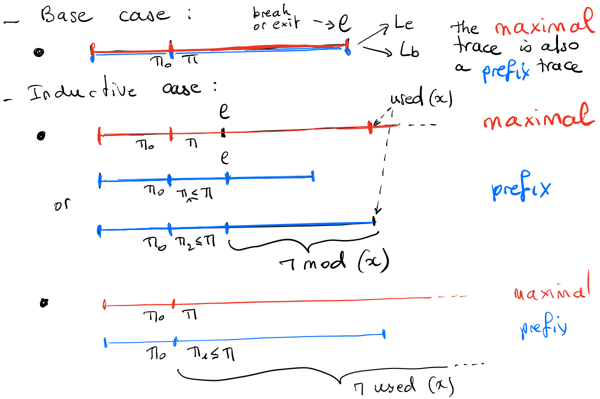
$$\alpha^{\exists d}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ D_b, D_e \quad \triangleq \quad \neg\alpha^{\forall l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ \neg D_b, \neg D_e \qquad \text{potential deadness} \qquad (41.5)$$

$$\alpha^{\forall d}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ D_b, D_e \quad \triangleq \quad \neg\alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ \neg D_b, \neg D_e \qquad \text{definite deadness} \qquad (41.6)$$



Prefix versus maximal trace semantics

Lemma 2 41.9 Using the prefix trace semantics $\boldsymbol{\mathcal{S}}^{*}[\![\mathsf{S}]\!]$ or the maximal trace semantics $\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathsf{S}]\!]$ in the definition of potential liveness $\alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]$ is equivalent. □



Semantic liveness/deadness abstractions $\mathcal{S}^{\exists l}[\![\mathsf{S}]\!] \triangleq \alpha^{\exists l}_{\text{use,mod}}[\![\mathsf{S}]\!]$

- An action a uses variable y in a given environment ρ if and only if it is possible to change the value of y so as to change the effect of action a on program execution.

$$
\begin{array}{rcll}
\mathsf{use}[\![\mathsf{skip}]\!]\,\rho & \triangleq & \varnothing & (41.10)\\
\mathsf{use}[\![\mathsf{x} = \mathsf{A}]\!]\,\rho & \triangleq & \{\mathsf{y} \mid \exists \nu \in \mathbb{V} \,.\, \mathscr{A}[\![\mathsf{A}]\!]\,\rho \neq \mathscr{A}[\![\mathsf{A}]\!]\,\rho[\mathsf{y} \leftarrow \nu] \wedge \rho(\mathsf{x}) \neq \mathscr{A}[\![\mathsf{A}]\!]\,\rho\} & \\
\mathsf{use}[\![\mathsf{a}]\!]\,\rho & \triangleq & \{\mathsf{y} \mid \exists \nu \in \mathbb{V} \,.\, \mathscr{B}[\![\mathsf{a}]\!]\,\rho \neq \mathscr{B}[\![\mathsf{a}]\!]\,\rho[\mathsf{y} \leftarrow \nu]\} & \mathsf{a} \in \{\mathsf{B}, \neg(\mathsf{B})\}
\end{array}
$$

Example, $\mathsf{y} \notin \mathsf{use}[\![\mathsf{x} = \mathsf{y} - \mathsf{y}]\!]\,\rho$ and $\mathsf{x} \notin \mathsf{use}[\![\mathsf{x} = \mathsf{x}]\!]\,\rho$.

- An action a modifies variable x in an environment ρ if and only the execution of action a in environment ρ changes the value of x.

$$
\mathsf{mod}[\![\mathsf{a}]\!]\,\rho \triangleq \{\mathsf{x} \mid \mathsf{a} = (\mathsf{x} = \mathsf{A}) \wedge (\rho(\mathsf{x}) \neq \mathscr{A}[\![\mathsf{A}]\!]\,\rho)\}
$$

Classic syntactic liveness/deadness abstractions $\mathcal{S}^{\exists l}[\![\mathsf{S}]\!] \triangleq \alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}[\![\mathsf{S}]\!]$

The set $\mathsf{use}[\![\mathsf{a}]\!]$ of variables used and the set $\mathsf{mod}[\![\mathsf{a}]\!]$ of variables assigned to/modified in an action $\mathsf{a} \in \mathcal{A}$ are postulated to be as follows (the parameter ρ is useless but added for consistency with (41.2)).

$$
\begin{array}{rcl}
\mathsf{use}[\![\mathsf{x = A}]\!]\,\rho & \triangleq & \mathsf{vars}[\![\mathsf{A}]\!] \\
\mathsf{use}[\![\mathsf{skip}]\!]\,\rho & \triangleq & \varnothing \\
\mathsf{use}[\![\mathsf{B}]\!]\,\rho & \triangleq & \mathsf{use}[\![\neg(\mathsf{B})]\!]\,\rho \ \triangleq\ \mathsf{vars}[\![\mathsf{B}]\!]
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{mod}[\![\mathsf{x = A}]\!]\,\rho & \triangleq & \{\mathsf{x}\} \\
\mathsf{mod}[\![\mathsf{skip}]\!]\,\rho & \triangleq & \varnothing \\
\mathsf{mod}[\![\mathsf{B}]\!]\,\rho & \triangleq & \mathsf{mod}[\![\neg(\mathsf{B})]\!]\,\rho \ \triangleq\ \varnothing
\end{array}
$$

where $\mathsf{vars}[\![\mathsf{E}]\!]$ is the set of program variables occurring in arithmetic or boolean expression E.

Unsoundness of the syntactic liveness/deadness abstractions

$$\mathcal{S}^{\exists l}[\![\mathtt{S}]\!] \;\dot{\not\subseteq}\; \mathcal{S}^{\exists l'}[\![\mathtt{S}]\!]$$

- Counter-example: ℓ_1 $\mathtt{x = y - y}$;ℓ_2 where x is live at ℓ_2 on exit
- Syntactically,
  - x is not used in y−y
  - x is modified by the assignment so x is always syntactically dead at ℓ_1.
- Semantically,
  - x is not used in y−y (since changing the value of x at ℓ_1 will not change the value of y−y which is always 0).
  - x is not always modified by the assignment $\mathtt{x = y - y}$; (in case x was 0 before)
- The problem is that

$$\exists \rho \in \mathbb{E}\mathbb{v} \,.\, \mathtt{y} \in \mathrm{use}[\![\mathtt{a}]\!]\,\rho \Rightarrow \forall \rho \in \mathbb{E}\mathbb{v} \,.\, \mathtt{y} \in \mathbb{use}[\![\mathtt{a}]\!]\,\rho$$

but

$$\exists \rho \in \mathbb{E}\mathbb{v} \,.\, \mathtt{x} \in \mathbb{mod}[\![\mathtt{a}]\!]\,\rho \wedge \mathtt{x} \notin \mathrm{mod}[\![\mathtt{a}]\!]\,\rho$$

What could go wrong when optimizing programs?

Consider a compiler that successively performs

1. a (syntactic) liveness analysis $\mathcal{S}^{\exists l}$;
2. next, a code optimization by removal
   (a) of assignments to variables that are dead after this assignment,
   (b) of assignments to variables that do not change the value of this variable (using Kildall's constancy analysis [Kildall, 1973] or better symbolic constancy analysis [Haghighat and Polychronopoulos, 1996; Wegman and Zadeck, 1991]);
3. next, a register allocation such that
   (a) simultaneously live variables are stored in different registers,
   (b) when no register is left and one is needed, one of those containing the value of a dead variable is preferred (to avoid saving the value of the variable to its memory location as would be needed for live variables).

For the following program (where all variables are dead on exit)

		semantically		syntactically	
		live	dead	live	dead
`x=0; scanf(y);`					
`if (x==0){`					
ℓ_1 `...` `x` and `y` neither used nor modified `...`	ℓ_1	{x}	{y}	{y}	{x}
ℓ_2 `x = y - y; }`	ℓ_2	{x}	{y}	{y}	{x}
`else {`					
`x=42;`					
`}`					
ℓ_3 `print(x);`	ℓ_3	{x}	{y}	{x}	{y}

- Code elimination ((2.b)) suppresses the assignment at ℓ_2 since the value of x is unchanged.
- Assume x is in a register at ℓ_1 and a fresh register is needed but none is left available. By ((3.b)) the register containing x may be selected since its value need not be saved to memory because x is syntactically dead at ℓ_1.
- Then the value of x is lost at ℓ_3, a compilation bug.
- This error does not occur with semantic liveness.

Solutions not to go wrong

- Prevent program transformations (such as (2.b) and (3.b) above) that do not preserve the soundness of the semantic liveness $\mathcal{S}^{\exists l}$.
- Move elimination of assignments to variables that do not change the value of this variable ((2.b)) before liveness analysis.
- Redo the liveness analysis after any program transformation that does not preserve the information.
- A better solution is adopted in CompCert [Leroy, 2009]: the liveness analysis and code elimination are performed simultaneously and the liveness analysis is updated to be valid *after* code elimination.

Calculational design of the structural syntactic potential liveness analysis

$$\begin{array}{rcl}
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{Sl}\ \ell]\!]\ L_e & \triangleq & \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{Sl}\ \ell]\!]\ \varnothing, L_e \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{x = A\ ;}]\!]\ L_b, L_e & \triangleq & \mathsf{use}[\![\mathtt{x = A}]\!] \cup (L_e \setminus \mathsf{mod}[\![\mathtt{x = A}]\!]) \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{;}]\!]\ L_b, L_e & \triangleq & L_e \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{Sl'\ S}]\!]\ L_b, L_e & \triangleq & \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{Sl'}]\!]\ L_b, (\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{S}]\!]\ L_b, L_e) \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\epsilon]\!]\ L_b, L_e & \triangleq & L_e \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{if\ (B)\ S}_t]\!]\ L_b, L_e & \triangleq & \mathsf{use}[\![\mathtt{B}]\!] \cup L_e \cup \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{S}_t]\!]\ L_b, L_e \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else}\ \mathtt{S}_f]\!]\ L_b, L_e & \triangleq & \mathsf{use}[\![\mathtt{B}]\!] \cup \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{S}_t]\!]\ L_b, L_e \cup \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{S}_f]\!]\ L_b, L_e \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{while\ (B)\ S}_b]\!]\ L_b, L_e & \triangleq & \mathsf{use}[\![\mathtt{B}]\!] \cup L_e \cup \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{S}_b]\!]\ L_b, L_e \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{break\ ;}]\!]\ L_b, L_e & \triangleq & L_b \\
\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\{\ \mathtt{Sl}\ \}]\!]\ L_b, L_e & \triangleq & \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathtt{Sl}]\!]\ L_b, L_e
\end{array} \tag{41.22}$$

No fixpoint iteration for the `while`, the solution can be directly computed by 1 iteration!.

Theorem 41.24 $\widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$ defined by (41.22) is syntactically sound that is

$$\alpha^{\exists l}_{\mathsf{use,mod}} \not\subseteq \alpha^{\exists l}_{\mathsf{use,mod}} (\mathcal{S}^*[\![\mathsf{S}]\!]) \mathrel{\dot{\subseteq}} \alpha^{\exists l}_{\mathbb{use,mod}}[\![\mathsf{S}]\!] (\mathcal{S}^*[\![\mathsf{S}]\!]) \mathrel{\dot{\subseteq}} \widehat{\mathcal{S}}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$$

Informally:

- A variable is live at a program component iff it is not semantically/syntactically used before being (syntactically) *assigned to*
- So, a compiler removing this assignment would be incorrect.

Proof by calculational design

Structural syntactic definite deadness analysis

$$\widehat{\mathcal{S}}^{\forall d}[\![\mathsf{S}]\!]\ D_b, D_e \triangleq \neg\widehat{\mathcal{S}}^{\exists l}[\![\mathsf{S}]\!]\ \neg D_b, \neg D_e$$

$$\begin{array}{rcl}
\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{Sl}\ \ell]\!]\ D_e &=& \widehat{\mathcal{S}}^{\forall d}[\![\mathtt{Sl}\ \ell]\!]\ \mathbb{V}, D_e \\
\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{x} = \mathtt{A}\ ;]\!]\ D_b, D_e &=& \neg\,\mathrm{use}[\![\mathtt{x} = \mathtt{A}]\!] \cap (D_e \cup \mathrm{mod}[\![\mathtt{x} = \mathtt{A}]\!]) \\
\widehat{\mathcal{S}}^{\forall d}[\![;]\!]\ D_b, D_e &=& D_e \\
\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{Sl}'\ \mathtt{S}]\!]\ D_b, D_e &=& \widehat{\mathcal{S}}^{\forall d}[\![\mathtt{Sl}']\!]\ D_b, (\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{S}]\!]\ D_b, D_e) \\
\widehat{\mathcal{S}}^{\forall d}[\![\ \epsilon\ ]\!]\ D_b, D_e &=& D_e \\
\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{if}\ (\mathtt{B})\ \mathtt{S}_t]\!]\ D_b, D_e &=& \neg\,\mathrm{use}[\![\mathtt{B}]\!] \cap D_e \cap \widehat{\mathcal{S}}^{\forall d}[\![\mathtt{S}_t]\!]\ D_b, D_e \\
\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{if}\ (\mathtt{B})\ \mathtt{S}_t\ \mathtt{else}\ \mathtt{S}_f]\!]\ D_b, D_e &=& \neg\,\mathrm{use}[\![\mathtt{B}]\!] \cap \widehat{\mathcal{S}}^{\forall d}[\![\mathtt{S}_t]\!]\ D_b, D_e \cap \widehat{\mathcal{S}}^{\forall d}[\![\mathtt{S}_f]\!]\ D_b, D_e \\
\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{while}\ (\mathtt{B})\ \mathtt{S}_b]\!]\ D_b, D_e &=& \neg\,\mathrm{use}[\![\mathtt{B}]\!] \cap D_e \cap \widehat{\mathcal{S}}^{\forall d}[\![\mathtt{S}_b]\!]\ D_b, D_e \\
\widehat{\mathcal{S}}^{\forall d}[\![\mathtt{break}\ ;]\!]\ D_b, D_e &=& D_b \\
\widehat{\mathcal{S}}^{\forall d}[\![\{\ \mathtt{Sl}\ \}]\!]\ D_b, D_e &=& \widehat{\mathcal{S}}^{\forall d}[\![\mathtt{Sl}]\!]\ D_b, D_e
\end{array} \tag{41.6}$$

Conclusion

- Correct compilation should preserve the source semantics
- Using syntactic reasonings may be be problematic if not well-understood with respect to the source, intermediate, and object code semantics
- The tradition of using a control graph, boolean equations, and fixpoint iteration may be inefficient
- Structural induction is more efficient (and no compiler writer looks to know that!)

References I

Bibliography

Allen, Frances E. (1970). "Control Flow Analysis.". *SIGPLAN Not.*. 5.7, pp. 1–19.

— (1971). "A Basis for Program Optimization.". In *IFIP Congress (1)*. Pp. 385–390.

— (1974). "Interprocedural data flow analysis.". In Jack L. Rosenfeld, ed. *Information Processing 74*. North-Holland Pub. Co., pp. 398–402.

Allen, Frances E. and John Cocke (1976). "A Program Data Flow Analysis Procedure.". *Commun. ACM*. 19.3, pp. 137–147.

Cousot, Patrick and Radhia Cousot (1979). "Systematic Design of Program Analysis Frameworks.". In *POPL*. ACM Press, pp. 269–282.

Haghighat, Mohammad R. and Constantine D. Polychronopoulos (1996). "Symbolic Analysis for Parallelizing Compilers.". *ACM Trans. Program. Lang. Syst.*. 18.4, pp. 477–518.

References II

Kildall, Gary A. (1973). “A Unified Approach to Global Program Optimization.”. In *POPL*. ACM Press, pp. 194–206.

Leroy, Xavier (2009). “Formal verification of a realistic compiler.”. *Commun. ACM.* 52.7, pp. 107–115.

Schmidt, David A. (1998). “Data Flow Analysis is Model Checking of Abstract Interpretations.”. In *POPL*. ACM, pp. 38–48.

Steffen, Bernhard (1991). “Data Flow Analysis as Model Checking.”. In *TACS*. Vol. 526. Lecture Notes in Computer Science. Springer, pp. 346–365.

— (1993). “Generating Data Flow Analysis Algorithms from Modal Specifications.”. *Sci. Comput. Program.*. 21.2, pp. 115–139.

Wegman, Mark N. and F. Kenneth Zadeck (1991). “Constant Propagation with Conditional Branches.”. *ACM Trans. Program. Lang. Syst.*. 13.2, pp. 181–210.

Home work

Read Ch. 41 “Dataflow Analysis” of

Principles of Abstract Interpretation
Patrick Cousot
MIT Press

The End, Thank you