Static analysis and all that

Martin Steffen

IfI UiO

Spring 2016

Plan

- approx. 15 lectures, details see web-page
- flexible time-schedule, depending on progress/interest
- covering parts/following the structure of textbook [2], concentrating on
  - overview
  - data-flow
  - control-flow
  - type- and effect systems
- on request, new parts possible
- helpful prior knowledge: having at least heard of
  - typed lambda calculi (especially for CFA)
  - simple type systems
  - operational semantics
  - lattice theory, fixpoints, induction

  but things needed will be covered . . .

1. Data flow analysis
   - Intraprocedural analysis
   - Theoretical properties
   - Monotone frameworks
   - Equation solving
   - Interprocedural Analysis
   - Shape analysis

Plan

- traditional form of program analysis
- again while-language
- number of analyses: available expr., reaching def's, very busy expr., live variables . . .
- general setting: monotone frameworks
- advanced topics:
    - interprocedural data flow
    - shape analysis

Initial and final labels

$$init : \mathbf{Stmt} \rightarrow \mathbf{Lab} \qquad final : \mathbf{Stmt} \rightarrow 2^{\mathbf{Lab}} \tag{1}$$

$$\begin{array}{c|cc} [x := a]^l & l & \{l\} \\ [\text{skip}]^l & l & \{l\} \\ S_1 ; S_2 & init(S_1) & final(S_2) \\ \texttt{if}[b]^l\,\texttt{then}\,S_1\,\texttt{else}\,S_2 & l & final(S_1) \cup final(S_2) \\ \texttt{while}[b]^l\,\texttt{do}\,S & l & \{l\} \end{array} \tag{2}$$

Blocks

$$\begin{aligned} blocks([x := a]^l) &= \\ blocks([\text{skip}]^l) &= \\ blocks(S_1; S_2) &= \\ blocks(\texttt{if}[b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2) &= \\ blocks(\texttt{while}[b]^l\ \texttt{do}\ S) &= \end{aligned} \tag{3}$$

Blocks

$$
\begin{aligned}
\mathit{blocks}([x := a]^l) &= [x := a]^l \\
\mathit{blocks}([\text{skip}]^l) &= [\text{skip}]^l \\
\mathit{blocks}(S_1; S_2) &= \mathit{blocks}(S_1) \cup \mathit{blocks}(S_2) \\
\mathit{blocks}(\texttt{if}[b]^l\,\texttt{then}\,S_1\,\texttt{else}\,S_2) &= \{[b]^l\} \cup \mathit{blocks}(S_1) \cup \mathit{blocks}(S_2 \\
\mathit{blocks}(\texttt{while}[b]^l\,\texttt{do}\,S) &= \{[b]^l\} \cup \mathit{blocks}(S)
\end{aligned}
\tag{3}
$$

Labels and flows = flow graph

$$\mathit{labels} : \textbf{Stmt} \to 2^{\textbf{Lab}} \qquad \mathit{flow} : \textbf{Stmt} \to 2^{\textbf{Lab} \times \textbf{Lab}}$$

$$\mathit{labels}(S) = \{l \mid [B]^l \in \mathit{blocks}(S)\} \tag{4}$$

$$\begin{aligned} \mathit{flow}([x := a]^l) &= \\ \mathit{flow}([\mathrm{skip}]^l) &= \\ \mathit{flow}(S_1 ; S_2) &= \\ \mathit{flow}(\texttt{if}\,[b]^l\,\texttt{then}\,S_1\,\texttt{else}\,S_2) &= \\ \mathit{flow}(\texttt{while}\,[b]^l\,\texttt{do}\,S) &= \end{aligned} \tag{5}$$

Labels and flows = flow graph

$$labels : \mathbf{Stmt} \rightarrow 2^{\mathbf{Lab}} \qquad flow : \mathbf{Stmt} \rightarrow 2^{\mathbf{Lab} \times \mathbf{Lab}}$$

$$labels(S) = \{l \mid [B]^l \in blocks(S)\} \tag{4}$$

$$\begin{aligned}
flow([x := a]^l) &= \emptyset \\
flow([\text{skip}]^l) &= \emptyset \\
flow(S_1; S_2) &= flow(S_1) \cup flow(S_2) \\
&\quad \cup \{(l, init(S_2)) \mid l \in final(S_1)\} \\
flow(\texttt{if}[b]^l\,\texttt{then}\,S_1\,\texttt{else}\,S_2) &= flow(S_1) \cup flow(S_2) \\
&\quad \cup \{(l, init(S_1)), (l, init(S_2))\} \\
flow(\texttt{while}[b]^l\,\texttt{do}\,S) &= flow(S_1) \cup \{l, init(S)\} \\
&\quad \cup \{(l', l) \mid l' \in final(S)\}
\end{aligned} \tag{5}$$

Flow and reverse flow

- *flow*: for forward analyses

$$labels(S) = init(S) \cup \{l \mid (l, l') \in flow(S)\} \cup \{l' \mid (l, l') \in flow(S)\}$$

- reverse flow $flow^R$: simply invert the edges of *flow*.

Program of interest

- S_*: program being analysed, top-level statement
- analogously $\mathbf{Lab}_*$, $\mathbf{Var}_*$, $\mathbf{Blocks}_*$
- trivial expression: a single variable or constant
- $\mathbf{AExp}_*$: non-trivial arithmetic sub-expr. of S_*, analogous for $\mathbf{AExp}(a)$ and $\mathbf{AExp}(b)$.
- useful restrictions
  - isolated entries: $(l, init(S_*)) \notin flow(S_*)$
  - isolated exits $\forall l_1 \in final(S_*).\quad (l_1, l_2) \notin flow(S_*)$
  - label consistency

    $$[B_1]^l, [B_2]^l \in blocks(S) \quad \text{then} \quad B_1 = B_2$$

    "*l* labels *the* block *B*"
  - even better: unique labelling

Avoid recomputation: Available expressions

- example:

$$[x := a + b]^1; [y := a * b]^2; \; \texttt{while} \; [y > a + b]^3 \; \texttt{do} \; ([a := a + 1]^4; [x := a + b]^5$$

Avoid recomputation: Available expressions

- example:

$$[x := a + b]^1; [y := a * b]^2; \ \texttt{while} \ [y > a + b]^3 \ \texttt{do} \ ([a := a + 1]^4; [x := a + b]^5$$

Goal

for each program point: which expressions must have already been computed (and not later modified), on all paths to the program point.

- usage: avoid re-computation

Available expressions: general

- given as flow equations (not constraints)[1]
- uniform representation of effect of basic blocks (= intra-block flow)

2 ingredients of intra-block flow

- kill: flow information "eliminated" passing through the basic block
- generate: flow information "generated new" passing through the basic block

- later example analyses: presented similarly
- different analyses ⇒ different kill- and generate-functions/different kind of flow information.

[1] but not too crucial, as we know already

Available expressions: types

- interest in sets of expressions: $2^{\mathbf{AExp}_*}$
- generation and killing:

$$\mathit{kill}_{\mathsf{AE}}, \mathit{gen}_{\mathsf{AE}} : \mathbf{Blocks}_* \to 2^{\mathbf{AExp}_*}$$

- analysis: pair of functions

$$\mathsf{AE}_{\mathit{entry}}, \mathsf{AE}_{\mathit{exit}} : \mathbf{Lab}_* \to 2^{\mathbf{AExp}_*}$$

Available expressions analysis: kill and generate

core of the intra-block flow specification

$$
\begin{aligned}
kill_{\mathsf{AE}}([x := a]^l) &= \\
kill_{\mathsf{AE}}([\mathsf{skip}]^l) &= \\
kill_{\mathsf{AE}}([b]^l) &=
\end{aligned}
$$

$$
\begin{aligned}
gen_{\mathsf{AE}}([x := a]^l) &= \\
gen_{\mathsf{AE}}([\mathsf{skip}]^l) &= \\
gen_{\mathsf{AE}}([b]^l) &=
\end{aligned}
$$

Available expressions analysis: kill and generate

core of the intra-block flow specification

$$
\begin{aligned}
kill_{\mathsf{AE}}([x := a]^l) &= \{a' \in \mathbf{AExp}_* \mid x \in fv(a')\} \\
kill_{\mathsf{AE}}([\mathsf{skip}]^l) &= \emptyset \\
kill_{\mathsf{AE}}([b]^l) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
gen_{\mathsf{AE}}([x := a]^l) &= \{a' \in \mathbf{AExp}(a) \mid x \notin fv(a')\} \\
gen_{\mathsf{AE}}([\mathsf{skip}]^l) &= \emptyset \\
gen_{\mathsf{AE}}([b]^l) &= \mathbf{AExp}(b)
\end{aligned}
$$

Flow equations: $\mathrm{AE}^{=}$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

Flow equations for AE

$$
\begin{aligned}
\mathrm{AE}_{entry}(l) &= \begin{cases} \emptyset & l = init(S_*) \\ \bigcap\{\mathrm{AE}_{exit}(l') \mid (l, l') \in flow(S_*)\} & \text{otherwise} \end{cases} \\
\mathrm{AE}_{exit}(l) &= \mathrm{AE}_{entry}(l) \setminus kill_{\mathrm{AE}}(B^l) \cup gen_{\mathrm{AE}}(B^l)
\end{aligned}
$$

where $B^l \in blocks(S_*)$

- note the "order" of kill/ generate

Remarks

- forward analysis (as RD)
- interest in largest solution (unlike RD) $\Rightarrow$ must analysis[2]
- expression is available: if no path kills it
- remember: informal description of AE: expression available on "all paths" (i.e., not killed on any)
- remember: reaching definitions
- illustration

[2] as opposed to may-analysis.

Example

Reaching definitions

- remember the intro
- here: same analysis, but based on the new definitions: kill, generate, flow ...
- example:

$$[x := 5]^1; [y := 1]^2; \texttt{while}[x > 1]^4\ \texttt{do}([y := x{*}y]^4; [x := x{-}1]^5)$$

Reaching definitions: types

- interest in sets of tuples of var's and program points/labels: $2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$ ($\mathbf{Lab}_*^? = \mathbf{Lab}_* + \{?\}$)
- generation and killing:

$$kill_{\mathrm{RD}}, gen_{\mathrm{RD}} : \mathbf{Blocks}_* \to 2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$$

- analysis: pair of functions

$$\mathrm{RD}_{entry}, \mathrm{RD}_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$$

Reaching defs: kill and generate

$$
\begin{aligned}
kill_{\mathsf{RD}}([x := a]^l) &= \\
kill_{\mathsf{RD}}([\mathsf{skip}]^l) &= \\
kill_{\mathsf{RD}}([b]^l) &= \\
\\
gen_{\mathsf{RD}}([x := a]^l) &= \\
gen_{\mathsf{RD}}([\mathsf{skip}]^l) &= \\
gen_{\mathsf{RD}}([b]^l) &=
\end{aligned}
$$

Reaching defs: kill and generate

$$
\begin{array}{rcl}
kill_{\mathrm{RD}}([x := a]^l) & = & \{(x, ?)\} \cup \\
 & & \bigcup \{(x, l') \mid B^{l'} \text{ is assgm. to } x \text{ in } S_*\} \\
kill_{\mathrm{RD}}([\text{skip}]^l) & = & \emptyset \\
kill_{\mathrm{RD}}([b]^l) & = & \emptyset \\
 & & \\
gen_{\mathrm{RD}}([x := a]^l) & = & \{(x, l)\} \\
gen_{\mathrm{RD}}([\text{skip}]^l) & = & \emptyset \\
gen_{\mathrm{RD}}([b]^l) & = & \emptyset
\end{array}
$$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

Flow equations for RD

$$
\begin{aligned}
RD_{entry}(l) &= \\
RD_{exit}(l) &= RD_{entry}(l) \setminus kill_{\mathrm{RD}}(B^l) \cup gen_{\mathrm{RD}}(B^l)
\end{aligned}
$$

where $B^l \in blocks(S_*)$

- same order of kill/generate

Flow equations: $RD^{=}$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

Flow equations for RD

$$
\begin{aligned}
RD_{entry}(l) &= \begin{cases} \{(x,?) \mid x \in fv(S_*)\} & l = init(S_*) \\ \bigcup\{RD_{exit}(l') \mid (l,l') \in flow(S_*)\} & \text{otherwise} \end{cases} \\
RD_{exit}(l) &= RD_{entry}(l) \setminus kill_{RD}(B^l) \cup gen_{RD}(B^l)
\end{aligned}
$$

where $B^l \in blocks(S_*)$

- same order of kill/generate

Flow equations: $AE^{=}$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

Flow equations for AE

$$\mathrm{AE}_{entry}(l) = \begin{cases} \emptyset & l = init(S_*) \\ \bigcap\{\mathrm{AE}_{exit}(l') \mid (l, l') \in flow(S_*)\} & \text{otherwise} \end{cases}$$

$$\mathrm{AE}_{exit}(l) = \mathrm{AE}_{entry}(l) \setminus kill_{\mathrm{AE}}(B^l) \cup gen_{\mathrm{AE}}(B^l)$$

where $B^l \in blocks(S_*)$

- note the "order" of kill/ generate

Example

Very busy expressions

- $$
\begin{array}{ll}
\texttt{if} & [a > b]^1 \\
\texttt{then} & [x := b - a]^2; [y := a - b]^3 \\
\texttt{else} & [a := b - a]^4; [x := a - b]^5
\end{array}
$$

Definition (Very busy expression)

an expr. is very busy at the exit of a label, if for all paths from that label, the expression is used before any of its variables is "redefined" (= overwritten).

- use: expression "hoisting"

Goal

for each program point, which expressions are very busy at the exit of that point.

Very busy expr.: types

- interested in: sets of expressions: $2^{\mathbf{AExp}_*}$
- generation and killing:

$$kill_{\mathsf{VB}}, gen_{\mathsf{VB}} : \mathbf{Blocks}_* \rightarrow 2^{\mathbf{AExp}_*}$$

- analysis: pair of functions

$$\mathsf{VB}_{entry}, \mathsf{VB}_{exit} : \mathbf{Lab}_* \rightarrow 2^{\mathbf{AExp}_*}$$

Very busy expr.: kill and generate

core of the intra-block flow specification

$$
\begin{aligned}
kill_{\mathrm{VB}}([x := a]^l) &= \\
kill_{\mathrm{VB}}([\mathrm{skip}]^l) &= \\
kill_{\mathrm{VB}}([b]^l) &= \\
\\
gen_{\mathrm{VB}}([x := a]^l) &= \\
gen_{\mathrm{VB}}([\mathrm{skip}]^l) &= \\
gen_{\mathrm{VB}}([b]^l) &=
\end{aligned}
$$

core of the intra-block flow specification

$$
\begin{array}{rcl}
kill_{\mathrm{VB}}([x := a]^l) & = & \{a' \in \mathbf{AExp}_* \mid x \in fv(a')\} \\
kill_{\mathrm{VB}}([\mathrm{skip}]^l) & = & \emptyset \\
kill_{\mathrm{VB}}([b]^l) & = & \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
gen_{\mathrm{VB}}([x := a]^l) & = & \mathbf{AExp}(a) \\
gen_{\mathrm{VB}}([\mathrm{skip}]^l) & = & \emptyset \\
gen_{\mathrm{VB}}([b]^l) & = & \mathbf{AExp}(b)
\end{array}
$$

Available expressions analysis: kill and generate

core of the intra-block flow specification

$$
\begin{aligned}
kill_{\text{AE}}([x := a]^l) &= \{a' \in \mathbf{AExp}_* \mid x \in fv(a')\} \\
kill_{\text{AE}}([\text{skip}]^l) &= \emptyset \\
kill_{\text{AE}}([b]^l) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
gen_{\text{AE}}([x := a]^l) &= \{a' \in \mathbf{AExp}(a) \mid x \notin fv(a')\} \\
gen_{\text{AE}}([\text{skip}]^l) &= \emptyset \\
gen_{\text{AE}}([b]^l) &= \mathbf{AExp}(b)
\end{aligned}
$$

Flow equations.: $\text{VB}^=$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

however: everything works backwards now

Flow equations: VB

$$\begin{aligned} \text{VB}_{exit}(l) &= \\ \text{VB}_{entry}(l) &= \end{aligned}$$

where $B^l \in blocks(S_*)$

Flow equations.: $\text{VB}^{=}$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

however: everything works backwards now

Flow equations: VB

$$\text{VB}_{exit}(l) = \begin{cases} \emptyset & l = final(S_*) \\ \bigcap\{\text{VB}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases}$$

$$\text{VB}_{entry}(l) = \text{VB}_{exit}(l) \setminus kill_{\text{VB}}(B^l) \cup gen_{\text{VB}}(B^l)$$

where $B^l \in blocks(S_*)$

Example

When can var's be "thrown away": Live variable analysis

$$[x := 2]^1; [y := 4]^2; [x := 1]^3;$$
$$(\texttt{if}[y > x]^4\ \texttt{then}[z := y]^5\ \texttt{else}[z := y * y]^6); [x := z]^7$$

When can var's be "thrown away": Live variable analysis

$$[x := 2]^1; [y := 4]^2; [x := 1]^3;$$
$$(\texttt{if}[y > x]^4\, \texttt{then}[z := y]^5\, \texttt{else}[z := y * y]^6); [x := z]^7$$

Live variable

a variable is live (at exit of a label) = there exists a path from the mentioned exit to the use of that variable which does not assign to the variable (i.e., redefines its value)

- use: dead code elimination, register allocation

Goal

for each program point: which variables may be live at the exit of that point.

Live variables: types

- interested in sets of variables $2^{\mathbf{Var}_*}$
- generation and killing:

$$kill_{\mathsf{LV}}, gen_{\mathsf{LV}} : \mathbf{Blocks}_* \rightarrow 2^{\mathbf{Var}_*}$$

- analysis: pair of functions

$$\mathsf{LV}_{entry}, \mathsf{LV}_{exit} : \mathbf{Lab}_* \rightarrow 2^{\mathbf{Var}_*}$$

Live variables: kill and generate

$$\begin{aligned}
\mathit{kill}_{\mathrm{AE}}([x := a]^l) &= \\
\mathit{kill}_{\mathrm{LV}}([\mathrm{skip}]^l) &= \\
\mathit{kill}_{\mathrm{LV}}([b]^l) &=
\end{aligned}$$

$$\begin{aligned}
\mathit{gen}_{\mathrm{LV}}([x := a]^l) &= \\
\mathit{gen}_{\mathrm{LV}}([\mathrm{skip}]^l) &= \\
\mathit{gen}_{\mathrm{LV}}([b]^l) &=
\end{aligned}$$

Live variables: kill and generate

$$
\begin{aligned}
kill_{\mathsf{AE}}([x := a]^l) &= \{x\} \\
kill_{\mathsf{LV}}([\mathsf{skip}]^l) &= \emptyset \\
kill_{\mathsf{LV}}([b]^l) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
gen_{\mathsf{LV}}([x := a]^l) &= fv(a) \\
gen_{\mathsf{LV}}([\mathsf{skip}]^l) &= \emptyset \\
gen_{\mathsf{LV}}([b]^l) &= fv(b)
\end{aligned}
$$

Flow equations $LV^{=}$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

however: everything works backwards now

Flow equations LV

$$\begin{aligned} LV_{exit}(l) &= \\ LV_{entry}(l) &= \end{aligned}$$

where $B^l \in blocks(S_*)$

Flow equations $\mathsf{LV}^{=}$

split into

- intra-block equations, using kill/generate
- inter-block equations, using flow

however: everything works backwards now

Flow equations LV

$$
\begin{aligned}
\mathsf{LV}_{exit}(l) &= \begin{cases} \emptyset & l \in final(S_*) \\ \bigcup\{\mathsf{LV}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases} \\
\mathsf{LV}_{entry}(l) &= \mathsf{LV}_{exit}(l) \setminus kill_{\mathsf{LV}}(B^l) \cup gen_{\mathsf{LV}}(B^l)
\end{aligned}
$$

where $B^l \in blocks(S_*)$

Example

Relating programs with analyses

- analyses
  - intended as (static) abstraction/overapprox. of real program behavior
  - so far: without real connection to programs
- soundness of the analysis: “safe” analysis
- but: we have not defined yet the behavior/semantics of programs
- here: “easiest” semantics: operational
- more precisely: small-step SOS (structural operational semantics)

states, configs, and transitions

fixing some data types

- state σ : **State** $=$ **Var** $\rightarrow$ **Z**
- configuration: pair of statement $\times$ state or (terminal) just a state
- transitions

$$\langle S, \sigma\rangle \rightarrow \acute{\sigma} \quad \text{or} \quad \langle S, \sigma\rangle \rightarrow \langle \acute{S}, \acute{\sigma}\rangle$$

Semantics of expressions

$$[\![\_]\!]^{\mathcal{A}}_{\_} : \textbf{AExp} \to (\textbf{State} \to \textbf{Z})$$
$$[\![\_]\!]^{\mathcal{B}}_{\_} : \textbf{BExp} \to (\textbf{State} \to \textbf{T})$$

simplifying assumption: no errors

$$\begin{array}{rcl} [\![x]\!]^{\mathcal{A}}_{\sigma} & = & \sigma(x) \\ [\![n]\!]^{\mathcal{A}}_{\sigma} & = & \mathcal{N}(n) \\ [\![a_1 \ \mathtt{op}_a \ a_2]\!]^{\mathcal{A}}_{\sigma} & = & [\![a_1]\!]^{\mathcal{A}}_{\sigma} \ \textbf{op}_a \ [\![a_2]\!]^{\mathcal{A}}_{\sigma} \end{array}$$

$$\begin{array}{rcl} [\![\text{not } b]\!]^{\mathcal{B}}_{\sigma} & = & \neg[\![b]\!]^{\mathcal{B}}_{\sigma} \\ [\![b_1 \ \mathtt{op}_b \ b_2]\!]^{\mathcal{B}}_{\sigma} & = & [\![b_1]\!]^{\mathcal{B}}_{\sigma} \ \textbf{op}_b \ [\![b_2]\!]^{\mathcal{B}}_{\sigma} \\ [\![a_1 \ \mathtt{op}_r \ a_2]\!]^{\mathcal{B}}_{\sigma} & = & [\![a_1]\!]^{\mathcal{A}}_{\sigma} \ \textbf{op}_r \ [\![a_2]\!]^{\mathcal{A}}_{\sigma} \end{array}$$

clearly:

$$\forall x \in fv(a).\ \sigma_1(x) = \sigma_2(x) \text{ then } [\![a]\!]^{\mathcal{A}}_{\sigma_1} = [\![a]\!]^{\mathcal{A}}_{\sigma_2}$$

SOS

$$\langle [x := a]^l, \sigma \rangle \to \sigma[x \mapsto [\![a]\!]^{\mathcal{A}}_{\sigma}] \quad \textsc{Ass} \qquad \langle [\mathsf{skip}]^l, \sigma \rangle \to \sigma \quad \textsc{SKIP}$$

$$\frac{\langle S_1, \sigma \rangle \to \langle \acute{S}_1, \acute{\sigma} \rangle}{\langle S_1; S_2, \sigma \rangle \to \langle \acute{S}_1; S_2, \acute{\sigma} \rangle}\ \textsc{Seq}_1 \qquad \frac{\langle S_1, \sigma \rangle \to \acute{\sigma}}{\langle S_1; S_2, \sigma \rangle \to \langle S_2, \acute{\sigma} \rangle}\ \textsc{Seq}_2$$

$$\frac{[\![b]\!]^{\mathcal{B}}_{\sigma} = \top}{\langle \mathtt{if}[b]^l\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \sigma \rangle \to \langle S_1, \sigma \rangle}\ \textsc{If}_1$$

$$\frac{[\![b]\!]^{\mathcal{B}}_{\sigma} = \top}{\langle \mathtt{while}[b]^l\ \mathtt{do}\ S, \sigma \rangle \to \langle S; \mathtt{while}[b]^l\ \mathtt{do}\ S, \sigma \rangle}\ \textsc{While}_1$$

$$\frac{[\![b]\!]^{\mathcal{B}}_{\sigma} = \bot}{\langle \mathtt{while}[b]^l\ \mathtt{do}\ S, \sigma \rangle \to \sigma}\ \textsc{While}_2$$

Derivation sequences

- derivation sequence: “completed” execution:
  - finite sequence: $\langle S_1, \sigma_1 \rangle, \ldots, \langle S_n, \sigma_n \rangle, \sigma_{n+1}$
  - infinite sequence: $\langle S_1, \sigma_1 \rangle, \ldots, \langle S_i, \sigma_i \rangle, \ldots$
- note: labels do not influence the semantics

Lemma

1. $\langle S, \sigma \rangle \rightarrow \sigma'$, *then* $final(S) = \{init(S)\}$
2. $\langle S, \sigma \rangle \rightarrow \langle \acute{S}, \acute{\sigma} \rangle$, *then* $final(S) \supseteq \{final(\acute{S})\}$
3. $\langle S, \sigma \rangle \rightarrow \langle \acute{S}, \acute{\sigma} \rangle$, *then* $flow(S) \supseteq \{flow(\acute{S})\}$
4. $\langle S, \sigma \rangle \rightarrow \langle \acute{S}, \acute{\sigma} \rangle$, *then* $blocks(S) \supseteq blocks(\acute{S})$*; if* S *is label consistent, then so is* $\acute{S}$

Correctness of live analysis

- LV as example
- given as constraint system (not as equational system)

LV constraint system

$$
\mathsf{LV}_{exit}(l) \supseteq \begin{cases} \emptyset & l \in final(S_*) \\ \bigcup\{\mathsf{LV}_{entry}(l') \mid (l', l) \in flow^R(S_*)\} & \text{otherwise} \end{cases}
$$

$$
\mathsf{LV}_{entry}(l) \supseteq \mathsf{LV}_{exit}(l) \setminus kill_{\mathsf{LV}}(B^l) \cup gen_{\mathsf{LV}}(B^l)
$$

$$
live_{entry}, live_{exit} : \mathbf{Lab}_* \to 2^{\mathbf{Var}_*}
$$

"*live* solves constraint system $\mathsf{LV}^{\subseteq}(S)$"

$$
live \models \mathsf{LV}^{\subseteq}(S)
$$

(analogously for equations $\mathsf{LV}^{=}(S)$)

When can var's be "thrown away": Live variable analysis

$$[x := 2]^1; [y := 4]^2; [x := 1]^3;$$
$$(\texttt{if}[y > x]^4 \texttt{ then}[z := y]^5 \texttt{ else}[z := y * y]^6); [x := z]^7$$

Live variable

a variable is live (at exit of a label) = there exists a path from the mentioned exit to the use of that variable which does not assign to the variable (i.e., redefines its value)

- use: dead code elimination, register allocation

Goal

for each program point: which variables may be live at the exit of that point.

Equational vs. constraint analysis

Lemma

- *If live* $\models \mathsf{LV}^{=}$*, then live* $\models \mathsf{LV}^{\subseteq}$
- *The least solutions of live* $\models \mathsf{LV}^{=}$ *and live* $\models \mathsf{LV}^{\subseteq}$ *coincide.*

Intermezzo: orders, lattices. etc.

as a reminder:

- partial order $(L, \sqsubseteq)$
- upper bound l of $Y \subseteq L$:
- least upper bound (lub): $\bigsqcup Y$ (or *join*)
- dually: lower bounds and greatest lower bounds: $\bigsqcap Y$ (or *meet*)
- complete lattice $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top)$: po-set where meets and joins exist for all subsets, furthermore $\bot = \bigsqcap \emptyset$ and $\top = \bigsqcup \emptyset$.

Fixpoints

given complete lattice L and monotone $f : L \to L$.

- fixpoint: $f(l) = l$

$$Fix(f) = \{l \mid f(l) = l\}$$

- f reductive at l, l is a pre-fixpoint of f: $f(l) \sqsubseteq l$:

$$Red(f) = \{l \mid f(l) \sqsubseteq l\}$$

- f extensive at l, l is a post-fixpoint of f: $f(l) \sqsupseteq l$:

$$Ext(f) = \{l \mid f(l) \sqsupseteq l\}$$

$$lfp(f) \triangleq \bigsqcap Fix(f) \text{ and } gfp(f) \triangleq \bigsqcup Fix(f)$$

Tarski's theorem

Theorem

L: complete lattice, $f : L \to L$ monotone.

$$\begin{array}{rcccl} \mathit{lfp}(f) & \triangleq & \bigsqcap \mathit{Red}(f) & \in & \mathit{Fix}(f) \\ \mathit{gfp}(f) & \triangleq & \bigsqcup \mathit{Ext}(f) & \in & \mathit{Fix}(f) \end{array} \quad (6)$$

Fixpoint iteration

- often: iterate, approximate least fixed point from below $(f^n(\bot))_n$:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \dots$$

- not assured that we "reach" the fixpoint ("within" ω)

$$\begin{array}{lll} \bot \sqsubseteq f^n(\bot) \sqsubseteq \bigsqcup_n f^n(\bot) & \sqsubseteq & \mathit{lfp}(f) \\ & & \mathit{gfp}(f) \quad \sqsubseteq \bigsqcap_n f^n(\top) \sqsubseteq f^n(\top) \sqsubseteq (\top) \end{array}$$

- additional requirement: continuity on f for all ascending chains $(l_n)_n$

$$f(\bigsqcup_n (l_n)) = \bigsqcup (f(l_n))$$

- ascending chain condition: $f^n(\bot) = f^{n+1}(\bot)$, i.e., $\mathit{lfp}(f) = f^n(\bot)$
- descending chain condition: dually

Equational vs. constraint analysis

Lemma

- *If live* $\models \mathsf{LV}^{=}$, *then live* $\models \mathsf{LV}^{\subseteq}$
- *The least solutions of live* $\models \mathsf{LV}^{=}$ *and live* $\models \mathsf{LV}^{\subseteq}$ *coincide.*

Basic preservation results

Lemma ("Smaller" graph $\rightarrow$ less constraints)

Assume live $\models \mathrm{LV}^{\subseteq}(S_1)$. *If flow*$(S_1) \supseteq$ *flow*(S_2) *and blocks*$(S_1) \supseteq$ *blocks*(S_2), *then live* $\models \mathrm{LV}^{\subseteq}(S_2)$.

Corollary ("subject reduction")

If live $\models \mathrm{LV}^{\subseteq}(S)$ *and* $\langle S, \sigma \rangle \rightarrow \langle \acute{S}, \acute{\sigma} \rangle$, *then live* $\models \mathrm{LV}^{\subseteq}(\acute{S})$

Lemma (Flow)

Assume live $\models \mathrm{LV}^{\subseteq}(S)$. *If* $l \rightarrow_{flow} l'$, *then* $live_{exit}(l) \supseteq live_{entry}(l')$.

Correctness relation

- basic intuitition: only live variables influence the program
- proof by induction

Correctness relation on states:

Given V = set of variables:[a]

$$\sigma_1 \sim_V \sigma_2 \text{ iff } \forall x \in V . \sigma_1(x) = \sigma_2(x) \tag{7}$$

[a] V is intended to be "live variables" but in $\sim_V$ just set of vars.

$\Rightarrow$

$$\begin{array}{ccccccccc}
\langle S, \sigma_1 \rangle & \longrightarrow & \langle S', \sigma_1' \rangle & \longrightarrow & \ldots & \longrightarrow & \langle S'', \sigma_1'' \rangle & \longrightarrow & \sigma_1''' \\
\Big| \sim_V & & \Big| \sim_{V'} & & & & \Big| \sim_{V''} & & \Big| \sim_{X(l)} \\
\langle S, \sigma_2 \rangle & \longrightarrow & \langle S', \sigma_2' \rangle & \longrightarrow & \ldots & \longrightarrow & \langle S'', \sigma_2'' \rangle & \longrightarrow & \sigma_2'''
\end{array}$$

Notation:

- $N(l) = live_{entry}(l)$, $X(l) = live_{exit}(l)$

Example

Correctness (1)

Lemma (Preservation inter-block flow)

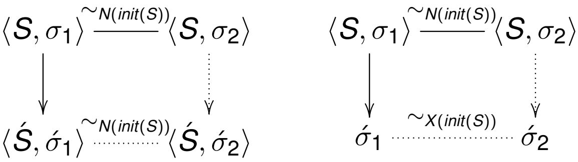
Assume live $\models \mathsf{LV}^{\subseteq}$. *If* $\sigma_1 \sim_{X(l)} \sigma_2$ *and* $l \rightarrow_{flow} l'$, *then* $\sigma_1 \sim_{N(l')} \sigma_2$.

Correctness

Theorem (Correctness)

Assume live $\models \mathrm{LV}^{\subseteq}(S)$.

- *If* $\langle S, \sigma_1 \rangle \rightarrow \langle \acute{S}, \acute{\sigma}_1 \rangle$ *and* $\sigma_1 \sim_{N(init(S))} \sigma_2$, *then there exists* $\acute{\sigma}_2$ *s.t.* $\langle S, \sigma_2 \rangle \rightarrow \langle \acute{S}, \acute{\sigma}_2 \rangle$ *and* $\acute{\sigma}_1 \sim_{N(init(\acute{S}))} \acute{\sigma}_2$.
- *If* $\langle S, \sigma_1 \rangle \rightarrow \acute{\sigma}_1$ *and* $\sigma_1 \sim_{N(init(S))} \sigma_2$, *then there exists* $\acute{\sigma}_2$ *s.t.* $\langle S, \sigma_2 \rangle \rightarrow \acute{\sigma}_2$ *and* $\acute{\sigma}_1 \sim_{X(init(S))} \acute{\sigma}_2$.



Correctness (many steps)

Assume *live* $\models \mathrm{LV}^{\subseteq}(S)$

- If $\langle S, \sigma_1 \rangle \rightarrow^* \langle \acute{S}, \acute{\sigma}_1 \rangle$ and $\sigma_1 \sim_{N(init(S))} \sigma_2$, then there exists $\acute{\sigma}_2$ s.t. $\langle S, \sigma_2 \rangle \rightarrow^* \langle \acute{S}, \acute{\sigma}_2 \rangle$ and $\acute{\sigma}_1 \sim_{N(init(\acute{S}))} \acute{\sigma}_2$.
- If $\langle S, \sigma_1 \rangle \rightarrow^* \acute{\sigma}_1$ and $\sigma_1 \sim_{N(init(S))} \sigma_2$, then there exists $\acute{\sigma}_2$ s.t. $\langle S, \sigma_2 \rangle \rightarrow^* \acute{\sigma}_2$ and $\acute{\sigma}_1 \sim_{X(l)} \acute{\sigma}_2$ for some $l \in final(S)$.

Monotone framework: general pattern

$$
\begin{array}{rcll}
\mathit{Analysis}_\circ(l) & = & \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\mathit{Analysis}_\bullet(l') \mid (l', l) \in F\} & \text{otherwise} \end{cases} \\
\mathit{Analysis}_\bullet(l) & = & f_l(\mathit{Analysis}_\circ(l))
\end{array} \tag{8}
$$

- $\bigsqcup$: either $\bigcup$ or $\bigcap$
- F: either $\mathit{flow}(S_*)$ or $\mathit{flow}^R(S_*)$.
- E: either $\{\mathit{init}(S_*)\}$ or $\mathit{final}(S_*)$
- ι: either the initial or final information
- f_l: transfer function for $[B]^l \in \mathit{blocks}(S_*)$.

Monotone frameworks

- direction of flow:
    - forward analysis:
        - $F = flow(S_*)$
        - $Analysis_\circ$ for entry and $Analysis_\bullet$ for exits
        - assumption: isolated entries
    - backward analysis: dually
        - $F = flow^R(S_*)$
        - $Analysis_\circ$ for exit and $Analysis_\bullet$ for entry
        - assumption: isolated exits
- sort of solution
    - may analysis
        - properties for some path
        - smallest solution
    - must analysis
        - properties of all paths
        - greatest solution

Without isolated entries

$$
\begin{aligned}
Analysis_\circ(l) &= \iota_E^l \sqcup \bigsqcup\{Analysis_\bullet(l') \mid (l', l) \in F\} \qquad (9)\\
&\quad \text{where } \iota_E^l = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases}\\
Analysis_\bullet(l) &= f_l(Analysis_\circ(l))
\end{aligned}
$$

where $l \sqcup \bot = l$

Basic definitions: property space

- property space L, often complete lattice
- combination operator: $\bigsqcup : 2^L \rightarrow L$ ($\sqcup$: binary case).
- $\bot = \bigsqcup \emptyset$
- often: ascending chain condition (stabilization)

Transfer functions

$$f_l : L \to L$$

with $l \in \mathbf{Lab}_*$

- associated with the blocks[3]
- requirements: monotone
- $\mathcal{F}$: monotone functions over L:
  - containing all transfer functions
  - containing identity
  - closed under composition

[3] One can do it also other way (but not in this lecture).

Framework (summary)

- complete lattice L, ascending chain condition
- $\mathcal{F}$ monotone functions, closed as stated
- distributive framework

$$f(l_1 \vee l_2) = f(l_1) \vee f(l_2)$$

(or rather $f(l_1 \vee l_2) \sqsubseteq f(l_1) \vee f(l_2)$)

Our 4 classical examples

- for a label consistent program S_*, all a instances of a monotone, distributive, framework:
- conditions:
  - lattice of properties: immediate (subset/superset)
  - ascending chain condition: finite set of syntactic entities
  - closure conditions on $\mathcal{F}$
    - monotone
    - closure under identity and composition
  - distributive: assured by using the kill- and generate-formulation

Instances: overview

	avail. epxr.	reach. def's	very busy expr.	live var's
L	$2^{\mathbf{AExp}_*}$	$2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?}$	$2^{\mathbf{AExp}_*}$	$2^{\mathbf{Var}_*}$
$\sqsubseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$
$\bigsqcup$	$\bigcap$	$\bigcup$	$\bigcap$	$\bigcup$
$\bot$	$\mathbf{AExp}_*$	$\emptyset$	$\mathbf{AExp}_*$	$\emptyset$
ι	$\emptyset$	$\{(x, ?) \mid x \in fv(S_*)\}$	$\emptyset$	$\emptyset$
E	$\{init(S_*)\}$	$\{init(S_*)\}$	$final(S_*)$	$final(S_*)$
F	$flow(S_*)$	$flow(S_*)$	$flow^R(S_*)$	$flow^R(S_*)$
$\mathcal{F}$	$\{f : L \to L \mid \exists l_k, l_g.\ f(l) = (l \setminus l_k) \cup l_g\}$			
f_l	$f_l(l) = (l \setminus kill([B]^l) \cup gen([B]^l))$ where $[B]^l \in blocks(S_*)$			

Solving the analyses

- given: set of equations (or constraints) over finite sets of variables
- domain of variables: complete lattices + ascending chain condition
- 2 solutions for the monotone frameworks
  1. MFP: "maximal fix point"
  2. MOP: "meet over all paths"

MFP

- terminology: historically “MFP” stands for *maximal* fix point (not minimal)
- iterative worklist algorithm:
  - central data structure: worklist
  - list (or container) of pairs
- related to chaotic iteration

Chaotic iteration

Input: example equations for reaching definitions

Output: least solution: $\vec{RD} = (RD_1, \ldots, RD_{12})$

Method: step 1: initialization
$RD_1 := \emptyset; \ldots; RD_{12} := \emptyset$

step 2: iteration

while $RD_j \neq F_j(RD_1, \ldots, RD_{12})$ for some j
do
$RD_j := F_j(RD_1, \ldots, RD_{12})$

Worklist algorithms

- fixpoint iteration algorithm
- general kind of algorithms, for DFA, CFA, ...
- same for equational and constraint systems
- "specialization"/determinization of chaotic iteration

⇒ worklist: central data structure, "container" containing "the work still to be done"

- for more details (different traversal strategies): see [2, Chap. 6]

WL-algo for DFA

- WL-algo for monotone frameworks

⇒ input: instance of monotone framework

- two central data structures
  - worklist: flow-edges yet to be (re-)considered:
    1. removed when effect of transfer function has been taken care of
    2. (re-)added, when point 1 endangers satisfaction of (in-)equations
  - array to store the “current state” of $Analysis_\circ$
- one central control structure (after initialization): loop until worklist empty

```
Input:  (L, 𝓕, F, E, ι, f)
Output: MFP∘, MFP•
Method: step 1: initialization
            W := nil;
            for all (l, l') ∈ F  do  W := (l, l') :: W;
            for all l ∈ F  or ∈ E  do
               if l ∈ E  then  Analysis[l] := ι
                               else  Analysis[l] := ⊥_L;
        step 2: iteration
            while  W ≠ nil  do
              (l, l')  :=  ( fst(head(W)), snd(head(W)));
              W  := tail W;
              if  f_l(Analysis[l]) ⋢ Analysis[l']
              then   Analysis[l'] := Analysis[l'] ⊔ f_l(Analysis[l]);
                     for all l'' with (l', l'') ∈ F  do
                            W := (l', l'') :: W;
         step 3: presenting the result:
              for all l ∈ F  or ∈ E  do
                MFP∘(l) := Analysis[l];
                MFP•(l) := f_l(Analysis[l])
```

MFP: properties

Lemma

The algo

- *terminates and*
- *calculates the least solution*

Proof.

- termination: ascending chain condition & loop is enlarging
- least FP:
  - invariant: array always below $Analysis_{\circ}$
  - at loop exit: array “solves” (in-)equations

□

Time complexity

- estimation of upper bound of number basic steps
  - at most b different labels in E
  - at most $e \geq b$ pairs in the flow F
  - height of the lattice: at most h
  - non-loop steps: $O(b + e)$
  - loop: at most h times addition to the WL

$\Rightarrow$

$$O(e \cdot h) \tag{10}$$

or $\leq O(b^2 h)$

MOP: paths

- terminoloy: historically: MOP stands for "meet over all paths"
- here: dually joins
- 2 versions of a path:
  1. path to entry of a block: blocks traversed from the "extremal block" of the program, but not including it
  2. path to exit of a block
- $$\begin{array}{lcl} path_{\circ}(l) & = & \{[l_1, \ldots \mathbf{l_{n-1}}] \mid l_i \to_{flow} l_{i+1} \wedge l_n = l \wedge l_1 \in E\} \\ path_{\bullet}(l) & = & \{[l_1, \ldots \mathbf{l_n}] \mid l_i \to_{flow} l_{i+1} \wedge l_n = l \wedge l_1 \in E\} \end{array}$$
- transfer function for paths $\vec{l}$

$$f_{\vec{l}} = f_{l_n} \circ \ldots f_{l_1} \circ id$$

MOP

- paths:
  - forward analyses: paths from init block to entry of a block
  - backward analyses: paths from exits of a block to a final block
- two components of the MOP solution (for given *l*):
  - up-to but not including *l*
  - up-to including *l*

$$MOP_\circ(l) = \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in path_\circ l\}$$
$$MOP_\bullet(l) = \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in path_\bullet l\}$$

MOP vs. MFP

- MOP: can be undecidable
- MFP approximates MOP ("$MFP \sqsupseteq MOP$")

Lemma

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet \tag{11}$$

In case of a distributive framework

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet \tag{12}$$

Adding procedures

- so far: very simplified language:
  - minimalistic imperative language
  - reading and writing to variables plus
  - simple controlflow, given as flow graph
- now: procedures: interprocedural analysis
- (possible) complications:
  - calls/returns (i.e., control flow)
  - parameter passing (call-by-value vs. call-by-reference)
  - scopes
  - potential aliasing (with call-by-reference)
  - higher-order functions/procedures
- here: top-level procedures, mutual recursion, call-by-value parameter + call-by-result

Syntax

- program: $\texttt{begin}\, D_* \; S_* \,\texttt{end}$

$$D_* ::= \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y) \overset{l_n}{\texttt{is}} S \overset{l_x}{\texttt{end}} \mid D \; D$$

- procedure names p
- statements

$$S ::= \ldots \; [\texttt{call}\, p(a, z)]^{l_c}_{l_r}$$

- note: call statement with 2 labels
- statically scoped language, CBV parameter passing (1st parameter), and CBN for second
- mutal recursion possible
- assumption: unique labelling, only declared procedures are called, all procedures have different names.

Example

$$
\begin{array}{lllll}
\texttt{begin} & \texttt{proc}\ \mathit{fib}(\texttt{val}\ z, u, \texttt{res}\ v)\ \texttt{is}^1 & & \\
 & \qquad \texttt{if} & [z < 3]^2 \\
 & \qquad \texttt{then} & [v := u + 1]^3 \\
 & \qquad \texttt{else} & [\texttt{call}\ \mathit{fib}(z - 1, u, v)]_5^4; \\
 & & [\texttt{call}\ \mathit{fib}(z - 2, v, v)]_7^6 \\
 & \texttt{end}^8; & \\
 & [\texttt{call}\ \mathit{fib}(x, 0, y)]_{10}^9 & \\
\texttt{end} & &
\end{array}
$$

Blocks, labels, etc

$$
\begin{array}{rcl}
\mathit{init}([\texttt{call}\ p(a,z)]_{l_r}^{l_c}) & = & l_c \\
\mathit{final}([\texttt{call}\ p(a,z)]_{l_r}^{l_c}) & = & \{l_r\} \\
\mathit{blocks}([\texttt{call}\ p(a,z)]_{l_r}^{l_c}) & = & \{[\texttt{call}\ p(a,z)]_{l_r}^{l_c}\} \\
\mathit{labels}([\texttt{call}\ p(a,z)]_{l_r}^{l_c}) & = & \{l_c, l_r\} \\
\mathit{flow}([\texttt{call}\ p(a,z)]_{l_r}^{l_c}) & = &
\end{array}
$$

Blocks, labels, etc

$$\begin{array}{rcl} \mathit{init}([\mathtt{call}\ p(a,z)]^{l_c}_{l_r}) & = & l_c \\ \mathit{final}([\mathtt{call}\ p(a,z)]^{l_c}_{l_r}) & = & \{l_r\} \\ \mathit{blocks}([\mathtt{call}\ p(a,z)]^{l_c}_{l_r}) & = & \{[\mathtt{call}\ p(a,z)]^{l_c}_{l_r}\} \\ \mathit{labels}([\mathtt{call}\ p(a,z)]^{l_c}_{l_r}) & = & \{l_c, l_r\} \\ \mathit{flow}([\mathtt{call}\ p(a,z)]^{l_c}_{l_r}) & = & \{(\mathbf{l_c}; \mathbf{l_n}), (\mathbf{l_x}; \mathbf{l_r})\} \end{array}$$

where $\mathtt{proc}\ p(\mathtt{val}\ x, \mathtt{res}\ y)\ \mathtt{is}^{l_n}\ S\ \mathtt{end}^{l_x}$ is in D_*.

- two *new* kinds of flows:[4] calling and returning
- static dispatch only

[4] written slightly different(!)

For procedure declaration

$$
\begin{aligned}
init(p) &= \\
final(p) &= \\
blocks(p) &= \quad \cup\, blocks(S) \\
labels(p) &= \\
flow(p) &=
\end{aligned}
$$

For procedure declaration

$$
\begin{array}{rcl}
\mathit{init}(p) & = & l_n \\
\mathit{final}(p) & = & \{l_x\} \\
\mathit{blocks}(p) & = & \{\mathtt{is}^{l_n}, \mathtt{end}^{l_x}\} \cup \mathit{blocks}(S) \\
\mathit{labels}(p) & = & \{l_n, l_x\} \cup \mathit{labels}(S) \\
\mathit{flow}(p) & = & \{(l_n, \mathit{init}(S))\} \cup \mathit{flow}(S) \cup \{(l, l_x) \mid l \in \mathit{final}(S)\}
\end{array}
$$

Flow graph of complete program

$$
\begin{array}{rcl}
init_* & = & init(S_*) \\
final_* & = & final(S_*) \\
blocks_* & = & \bigcup\{blocks(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\, \texttt{is}^{l_n}\, S\, \texttt{end}^{l_x} \in D_*\} \\
 & & \cup blocks(S_*) \\
labels_* & = & \bigcup\{labels(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\, \texttt{is}^{l_n}\, S\, \texttt{end}^{l_x} \in D_*\} \\
 & & \cup labels(S_*) \\
flow_* & = & \bigcup\{flow(p) \mid \texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\, \texttt{is}^{l_n}\, S\, \texttt{end}^{l_x} \in D_*\} \\
 & & \cup flow(S_*)
\end{array}
$$

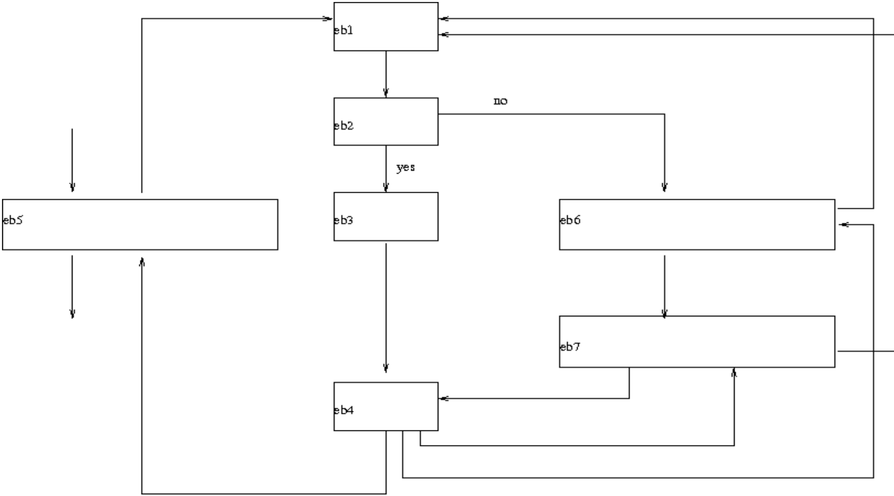
Interprocedural flow

- inter-procedural: from call-site to procedure, and back: $(l_c; l_n)$ and $(l_x; l_r)$.
- more precise (=better) capture of flow:

$$\textit{inter-flow}_* = \{(l_c, l_n, l_x, l_r) \mid P_* \text{ contains } \begin{array}{l}[\texttt{call}\, p(a,z)]^{l_c}_{l_r} \text{ and} \\ \texttt{proc}(\texttt{val}\, x, \texttt{res}\, y)\, \texttt{is}^{l_n}\, S\, \texttt{er}\end{array}$$

abbreviation: *IF* for $\textit{inter-flow}_*$ or $\textit{inter-flow}^R_*$

Example: fibonacci flow



Semantics: stores, locations,. . .

- not only new syntax
- new semantical concept: local data!
  - different "incarnations" of a variable $\Rightarrow$ locations
  - remember: $\sigma \in \textbf{State} = \textbf{Var}_* \to \textbf{Z}$

ξ	$\in$	$\textbf{Loc}$	locations
ρ	$\in$	$\textbf{Env} = \textbf{Var}_* \to \textbf{Loc}$	environment
ς	$\in$	$\textbf{Store} = \textbf{Loc} \to_{fin} \textbf{Z}$ (partial functions)	store

- $\sigma = \varsigma \circ \rho$: total $\Rightarrow$ $ran(\rho) \subseteq dom(\varsigma)$
- top-level environment: ρ_*: all var's are mapped to unique locations

Steps

- steps relative to environment ρ

$$\rho \vdash_* \langle S, \varsigma \rangle \to \langle \acute{S}, \acute{\varsigma} \rangle$$

or

$$\rho \vdash_* \langle S, \varsigma \rangle \to \acute{\varsigma}$$

- old rules needs to be adapted

$$\frac{\xi_1, \xi_2 \notin dom(\varsigma) \qquad v \in \mathbf{Z} \quad \texttt{proc}\; p(\texttt{val}\; x, \texttt{res}\; y)\; \texttt{is}^{l_n}\; S\; \texttt{end}^{l_x} \in D_* \quad \acute{\varsigma} =}{\rho \vdash_* \langle [\texttt{call}\; p(a, z)]^{l_c}_{l_r}, \varsigma \rangle \to \langle \text{bind}\; \rho[x \mapsto \xi_1][y \mapsto \xi_2]\; \texttt{in}\; S\; \texttt{then}\; z := y, \acute{\varsigma} \rangle}$$

Steps

- steps relative to environment ρ

$$\rho \vdash_* \langle S, \varsigma \rangle \to \langle \acute{S}, \acute{\varsigma} \rangle$$

or

$$\rho \vdash_* \langle S, \varsigma \rangle \to \acute{\varsigma}$$

- old rules needs to be adapted

$$\frac{\xi_1, \xi_2 \notin dom(\varsigma) \qquad v \in \mathbf{Z} \qquad \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{l_n}\ S\ \texttt{end}^{l_x} \in D_* \qquad \acute{\varsigma} = \varsigma[\xi_1 \mapsto [a]^{\mathcal{A}}_{\varsigma \circ \rho}][\xi_2 \mapsto v]}{\rho \vdash_* \langle [\texttt{call}\ p(a, z)]^{l_c}_{l_r}, \varsigma \rangle \to \langle \text{bind}\ \rho[x \mapsto \xi_1][y \mapsto \xi_2]\ \texttt{in}\ S\ \texttt{then}\ z := y, \acute{\varsigma} \rangle}$$

Bind-construct

$$\frac{\acute{\rho} \vdash_* \langle S, \varsigma\rangle \to \langle \acute{S}, \acute{\varsigma}\rangle}{\rho \vdash_* \langle \text{bind } \acute{\rho} \;\texttt{in}\; S \;\texttt{then}\; z := y, \varsigma\rangle \to} \;\textsc{Bind}_1$$

$$\frac{\acute{\rho} \vdash_* \langle S, \varsigma\rangle \to \acute{\varsigma}}{\rho \vdash_* \langle \text{bind } \acute{\rho} \;\texttt{in}\; S \;\texttt{then}\; z := y, \varsigma\rangle \to} \;\textsc{Bind}_2$$

- bind-syntax: “runtime syntax”

⇒ formulation of correctness must be adapted, too (Chap. 3)

Bind-construct

$$\frac{\acute{\rho} \vdash_* \langle S, \varsigma\rangle \to \langle \acute{S}, \acute{\varsigma}\rangle}{\rho \vdash_* \langle \text{bind } \acute{\rho} \texttt{ in } S \texttt{ then } z := y, \varsigma\rangle \to \langle \text{bind } \acute{\rho} \texttt{ in } \acute{S} \texttt{ then } z := y, \acute{\varsigma}\rangle} \text{ BIND}_1$$

$$\frac{\acute{\rho} \vdash_* \langle S, \varsigma\rangle \to \acute{\varsigma}}{\rho \vdash_* \langle \text{bind } \acute{\rho} \texttt{ in } S \texttt{ then } z := y, \varsigma\rangle \to \acute{\varsigma}[\rho(z) \mapsto \acute{\varsigma}(\acute{\rho}(y))]} \text{ BIND}_2$$

- bind-syntax: "runtime syntax"

⇒ formulation of correctness must be adapted, too (Chap. 3)

Naive formulation

- first attempt
- assumptions:
  - for each proc. call: 2 transfer functions: f_{l_c} (call) and f_{l_r} (return)
  - for each proc. definition: 2 transfer functions: f_{l_n} (enter) and f_{l_x} (exit)
- given: mon. framework $(L, \mathcal{F}, F, E, \iota, f)$
- inter-proc. edges $(l_c; l_n)$ and $(l_x; l_r)$ = ordinary flow edges (l_1, l_2)
- ignore parameter passing: *transfer* functions for proc. calls/proc definitions are identity

Equation system

$$
\begin{array}{rcl}
A_\bullet(l) & = & f_l(A_\circ(l)) \\
A_\circ(l) & = & \bigsqcup\{A_\bullet(l') \mid (l', l) \in F \text{ or } (l'; l) \in F\} \vee \iota_E^l
\end{array}
$$

with

$$
\iota_E^l = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases}
$$

- analysis: safe
- unnecessary unprecise/too abstract

MVP

- restrict attention to valid ("possible") paths

⇒ capture the nesting structure

- from MOP to MVP: "meet over all *valid* paths"
- complete path:
  - appropriate nesting
  - all calls are answered

Complete paths

- given $P_* = \texttt{begin}\, D_*\ S_*\, \texttt{end}$
- CP_{l_1,l_2}: complete paths from l_1 to l_2
- generated by the following productions (l's are the terminals)[5]

$$\frac{}{CP_{l,l} \longrightarrow l}$$

$$\frac{(l_1, l_2) \in F}{CP_{l_1,l_3} \longrightarrow l_1, CP_{l_2,l_3}}$$

$$\frac{(l_c, l_n, l_x, l_r) \in IF}{CP_{l_c,l} \longrightarrow l_c, CP_{l_n,l_x}, CP_{l_r,l}}$$

[5] We assume forward analysis here.

Example: Fibonacci

- grammar for fibonacci program:

$$
\begin{array}{rcl}
CP_{9,10} & \longrightarrow & 9, CP_{1,8}, CP_{10,10} \\
CP_{10,10} & \longrightarrow & 10 \\
CP_{1,8} & \longrightarrow & 1, CP_{2,8} \\
CP_{2,8} & \longrightarrow & 2, CP_{3,8} \\
CP_{2,8} & \longrightarrow & 2, CP_{4,8} \\
CP_{3,8} & \longrightarrow & 3, CP_{8,8} \\
CP_{8,8} & \longrightarrow & 8 \\
CP_{4,8} & \longrightarrow & 4, CP_{1,8}, CP_{5,8} \\
CP_{5,8} & \longrightarrow & 5, CP_{6,8} \\
CP_{6,8} & \longrightarrow & 6, CP_{1,8}, CP_{7,8} \\
CP_{7,8} & \longrightarrow & 7, CP_{8,8}
\end{array}
$$

- valid path:
  - start at extremal node (E),
  - all proc exits have matching entries
- generated by non-terminal VP_*

$$\frac{l_1 \in E \qquad l_2 \in \mathbf{Lab}_*}{VP_* \longrightarrow VP_{l_1,l_2}} \qquad \frac{}{VP_{l,l} \longrightarrow l}$$

$$\frac{(l_1, l_2) \in F}{VP_{l_1,l_3} \longrightarrow l_1, VP_{l_2,l_3}}$$

$$\frac{(l_c, l_n, l_x, l_r) \in IF}{VP_{l_c,l} \longrightarrow l_c, CP_{l_n,l_x}, VP_{l_r,l}} \qquad \frac{(l_c, l_n, l_x, l_r) \in IF}{VP_{l_c,l} \longrightarrow l_c, VP_{l_n,l}}$$

- adapt the definition of paths

$$
\begin{aligned}
vpath_{\circ}(l) &= \{[l_1, \ldots \mathbf{l_{n-1}}] \mid l_n = l \wedge [l_1, \ldots, l_n] \text{ valid}\} \\
vpath_{\bullet}(l) &= \{[l_1, \ldots \mathbf{l_n}] \mid l_n = l \wedge [l_1, \ldots, l_n] \text{ valid}\}
\end{aligned}
$$

- MVP solution:

$$
\begin{aligned}
MVP_{\circ}(l) &= \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in vpath_{\circ}(l)\} \\
MVP_{\bullet}(l) &= \bigsqcup\{f_{\vec{l}}(\iota) \mid \vec{l} \in vpath_{\bullet}(l)\}
\end{aligned}
$$

Contexts

- MVP/MOP *undecidable* but more precise than basic MFP

⇒ instead of MVP: "embellish" MFP

$$\delta \in \Delta \tag{13}$$

- for instance: representing/recording of the path taken

⇒ "embellishment":[6] adding contexts

embellished monotone framework

$$(\hat{L}, \hat{\mathcal{F}}, F, E, \hat{\iota}, \hat{f})$$

- intra-procedural (*independent* of Δ)
- inter-procedural

[6] Here, notationally indicated by a hat on top.

Intra-procedural

- this part: independent of Δ
  - property lattice: $\hat{L} = \Delta \to L$
  - mononote functions $\hat{\mathcal{F}}$
  - transfer functions: pointwise

$$\hat{f}_l(\hat{l})(\delta) = f_l(\hat{l}(\delta)) \tag{14}$$

- flow equations: “unchanged” for intra-proc. part

$$\begin{aligned} A_\bullet(l) &= \hat{f}_l(A_\circ(l)) \\ A_\circ(l) &= \bigsqcup\{A_\bullet(l') \mid (l', l) \in F \text{ or } (l'; l) \in F)\} \vee \iota_E^{\hat{l}} \end{aligned} \tag{15}$$

- in equation for $A_\bullet$: except for labels l for proc. calls (i.e., not l_c and l_r)

Sign analysis

- $\textbf{Sign} = \{-, 0, +\}$, $L_{sign} = 2^{\textbf{Var}_* \to \textbf{Sign}}$
- abstract states $\sigma^{sign} \in L_{sign}$
- for *expressions:* $[\_]^{\mathcal{A}_{sign}}_{\_} : \textbf{AExp} \to (\textbf{Var}_* \to \textbf{Sign}) \to 2^{\textbf{Sign}}$
- transfer function for $[x := a]^l$

$$f_l^{sign}(Y) = \bigcup \{\Phi_l^{sign}(\sigma^{sign}) \mid \sigma^{sign} \in Y\} \tag{16}$$

where $Y \subseteq \textbf{Var}_* \to \textbf{Sign}$ and

$$\phi_l^{sign}(\sigma^{sign}) = \{\sigma^{sign}[x \mapsto s] \mid s \in [a]^{\mathcal{A}_{sign}}_{\sigma^{sign}}\} \tag{17}$$

Sign analysis: embellished

$$\hat{L}_{sign} = \Delta \to L_{sign} = \Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})} \tag{18}$$

- transfer function for $[x := a]^l$

$$\hat{f}_l^{sign}(Z) = \bigcup \{ \{\delta\} \times \phi_l^{sign}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z \} \tag{19}$$

Inter-procedural

- procedure definition `proc(val` x, `res` y) `is`^{l_n} S `end`^{l_x}:

$$\hat{f}_{l_n}, \hat{f}_{l_x} : (\Delta \to L) \to (\Delta \to L) = id$$

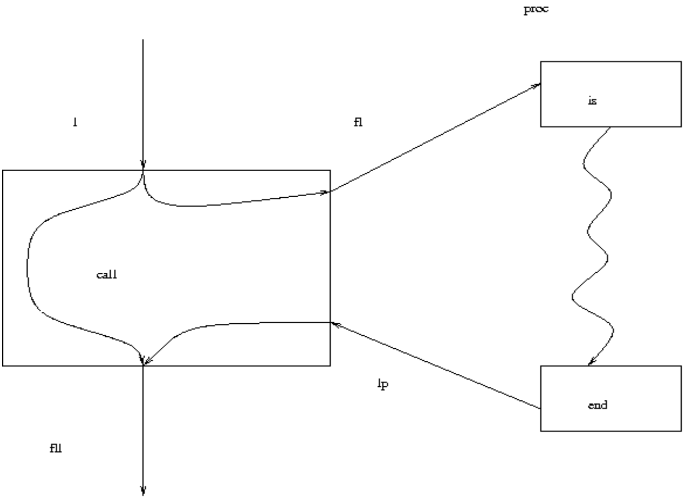
- procedure call: $(l_c, l_n, l_x, l_r) \in IF$
- here: forward analysis
- call: 2 transfer functions/2 sets of equations, i.e., for all $(l_c, l_n, l_x, l_r) \in IF$
  1. for calls:
     - $\hat{f1}_{l_c} : (\Delta \to L) \to (\Delta \to L)$

$$A_\bullet(l_c) = \hat{f1}_{l_c}(A_\circ(l_c)) \tag{20}$$

  2. for returns:
     - $\hat{f2}_{l_c,l_r} : (\Delta \to L) \times (\Delta \to L) \to (\Delta \to L)$

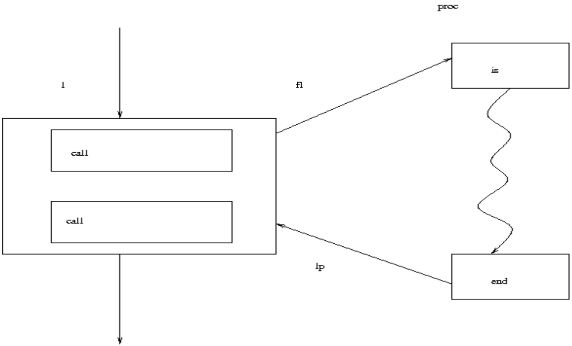
$$A_\bullet(l_r) = \hat{f2}_{l_c,l_r}(A_\circ(l_c), A_\circ(l_r))) \tag{21}$$

Procedure call



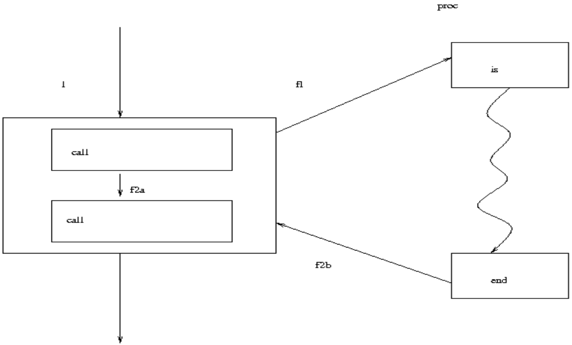
Ignoring call context

$$\hat{f}^2_{l_c,l_r}(\hat{l},\hat{l}') = \hat{f}^2_{l_r}(\hat{l}')$$



Merging call context

$$\hat{f}^{2}_{l_c,l_r}(\hat{l},\hat{l}') = \hat{f}^{2A}_{l_c,l_r}(\hat{l}) \vee \hat{f}^{2B}_{l_c,l_r}(\hat{l}')$$



Context sensitivity

- IF-edges: allow to relate returns to matching calls[7]
- context insensitive: proc-body analysed combining flow information from all call-sites.
- contexts: can be used to distinguish different call-sites

⇒ context sensitive analysis ⇒ more precision + more effort

In the following: 2 specializations:

1. control ("call strings")
2. data

[7] at least in the MVP-approach.

Call strings

- context = path
- concentrating on calls: flow-edges (l_c, l_n), where just l_c is recorded

$$\Delta = \mathbf{Lab}^* \qquad \text{call strings}$$

- extremal value

$$\hat{\iota}(\delta) =$$

Call strings

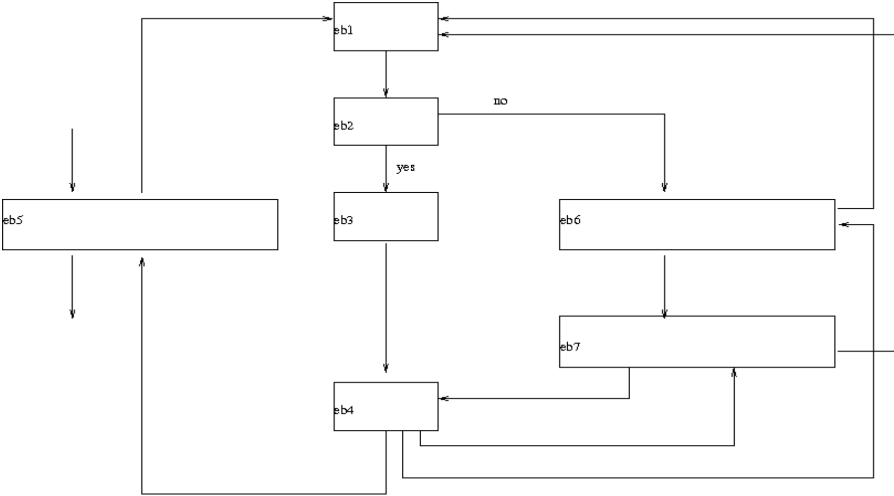
- context = path
- concentrating on calls: flow-edges (l_c, l_n), where just l_c is recorded

$$\Delta = \mathbf{Lab}^* \qquad \text{call strings}$$

- extremal value

$$\hat{\iota}(\delta) = \begin{cases} \iota & \text{if } \delta = \epsilon \\ \bot & \text{otherwise} \end{cases}$$

Example: fibonacci flow



Example: Fibonacci

some call strings:

$$\epsilon, [9], [9,4], [9,6], [9,4,4], [9,4,6], [9,6,4], [9,6,6], \ldots$$

Transfer functions for call strings

- here: forward analysis
- 2 cases: define $\hat{f}^1_{l_c}$ and $\hat{f}^2_{l_c,l_r}$
  - calls (basically: check that the path ends with l_c):

$$\begin{aligned} \hat{f}^1_{l_c}(\hat{l})([\delta, l_c]) &= f^1_{l_c}(\hat{l}(\delta)) \\ \hat{f}^1_{l_c}(\_) &= \bot \end{aligned} \tag{22}$$

  - returns (basically: match return with the call)

$$\hat{f}^2_{l_c,l_r}(\hat{l}, \hat{l}')(\delta) = f_{l_c,l_r}(\hat{l}(\delta), \hat{l}'([\delta, l_c])) \tag{23}$$

- Note: connection between the arguments (via δ) of f_{l_c,l_r}
- Notation: $[\hat{\delta}, l_c]$: concatenation of calls string
- l': at procedure exit.

Sign analysis

calls: abstract parameter-passing + glueing calls-returns

$$\Phi^{sign1}_{l_c}(\sigma^{sign}) \quad = \quad \{\sigma^{sign}[\mapsto][\mapsto] \mid s \in [a]^{\mathcal{A}_{sign}}_{\sigma^{sign}}, \}$$

returns (analogously)

$$\Phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) \quad = \quad \{\sigma^{sign}_2[\mapsto]\}$$

(formal params: x, y, call-site return variable z)

calls: abstract parameter-passing + glueing calls-returns

$$\Phi^{sign1}_{l_c}(\sigma^{sign}) \quad = \quad \{\sigma^{sign}[x \mapsto s][y \mapsto s'] \mid s \in [a]^{\mathcal{A}_{sign}}_{\sigma^{sign}},\ s' \in \{-, 0, +\}\}$$

returns (analogously)

$$\Phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) \quad = \quad \{\sigma^{sign}_2[x, y, z \mapsto \sigma^{sign}_1(x), \sigma^{sign}_1(y), \sigma^{sign}_2(y)]\}$$

(formal params: x, y, call-site return variable z)

Sign analysis

calls: abstract parameter-passing + glueing calls-returns

$$\begin{array}{rcl}\hat{f}_{l_c}^{sign1}(Z) & = & \bigcup\{\{\delta'\} \times \Phi_{l_c}^{sign1}(\sigma^{sign}) \mid (\delta', \sigma^{sign}) \in Z, \delta' = [\delta, l_c]\}\\ \Phi_{l_c}^{sign1}(\sigma^{sign}) & = & \{\sigma^{sign}[x \mapsto s][y \mapsto s'] \mid s \in [a]_{\sigma^{sign}}^{\mathcal{A}_{sign}},\ s' \in \{-, 0, +\}\}\end{array}$$

returns (analogously)

$$\begin{array}{rcl}\hat{f}_{l_c,l_r}^{sign2}(Z, Z') & = & \bigcup\{\{\delta\} \times \Phi_{l_c,l_r}^{sign2}(\sigma_1^{sign}, \sigma_2^{sign}) \mid \begin{array}{l}(\delta, \sigma_1^{sign}) \in Z\\ (\delta', \sigma_2^{sign}) \in Z'\\ \delta' = [\delta, l_c]\end{array}\}\\ \Phi_{l_c,l_r}^{sign2}(\sigma_1^{sign}, \sigma_2^{sign}) & = & \{\sigma_2^{sign}[x, y, z \mapsto \sigma_1^{sign}(x), \sigma_1^{sign}(y), \sigma_2^{sign}(y)]\}\end{array}$$

(formal params: x, y, call-site return variable z)

Call strings of bounded length

- recursion $\Rightarrow$ call-strings of unbounded length

$\Rightarrow$ restrict the length

$$\Delta = \mathbf{Lab}^{\leq k} \qquad \text{for some } k \geq 0$$

- for $k = 0$ context-insensitive ($\Delta = \{\epsilon\}$)

Assumption sets

- alternative to call strings
- not tracking the path, but assumption about the state
- assume here: $L = 2^D$

$\Rightarrow$ $\hat{L} = \Delta \to L \simeq 2^{\Delta \times D}$

- restrict to only the last call[8]
- dependency on data only $\Rightarrow$
- (large) assumption set context
  - $\Rightarrow$ $\Delta = 2^D$
  - $\hat{\iota} = \{(\{\iota\}, \iota)\}$ initial context

[8] corresponds to $k = 1$

Transfer functions

- calls

$$\hat{f}^1_{l_c}(Z) \quad = \quad \bigcup \{ \{\delta'\} \times \Phi^1_{l_c}(d) \mid \begin{array}{l} (\delta, d) \in Z \wedge \\ \delta' = \end{array} \}$$

where $\Phi^1_{l_c} : D \rightarrow 2^D$

- return

$$\hat{f}^2_{l_c,l_r}(Z, Z') \quad = \quad \bigcup \{ \{\delta\} \times \Phi^2_{l_c,l_r}(d, d') \mid \begin{array}{l} (\delta, d) \in Z \wedge \\ (\delta', d') \in Z' \wedge \\ \delta' = \end{array} \}$$

Transfer functions

- calls

$$\hat{f}^1_{l_c}(Z) \quad = \quad \bigcup\{\{\delta'\} \times \Phi^1_{l_c}(d) \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\end{array} \}$$

where $\Phi^1_{l_c} : D \to 2^D$

- return

$$\hat{f}^2_{l_c,l_r}(Z, Z') \quad = \quad \bigcup\{\{\delta\} \times \Phi^2_{l_c,l_r}(d, d') \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ (\delta', d') \in Z' \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\end{array} \}$$

Small assumption sets

- throw away even more information.

$$\Delta = D$$

- instead of $2^D \times D$: now only $D \times D$.
- transfer functions simplified
  - call

$$\hat{f}^1_{l_c}(Z) \quad = \quad \bigcup \{ \{\delta\} \times \Phi^1_{l_c}(d) \mid \; (\delta, d) \in Z \; \}$$

  - return

$$\hat{f}^2_{l_c,l_r}(Z, Z') \quad = \quad \bigcup \{ \{\delta\} \times \Phi^2_{l_c,l_r}(d, d') \mid \; \begin{array}{l} (\delta, d) \in Z \wedge \\ (\delta, d') \in Z' \end{array} \; \}$$

Flow-(in-)sensitivity

- “execution order” influences result of the analysis:

$$S_1\,;S_2 \quad \text{vs.} \quad S_2; S_1$$

- flow in-sensitivity: order is irrelevant
- less precise (but “cheaper”)
- for instance: *kill* is empty
- sometimes useful in combination with inter-proc. analysis

Set of assigned variables

- for procedure p: determine

$$\mathrm{IAV}(p)$$

  global variables that may be assigned to (also indirectly) when p is called
- two aux. definitions (straightforwardly defined, obviously flow-insensitive)
  - $\mathrm{AV}(S)$: assigned variables in S
  - $\mathrm{CP}(S)$: called procedures in S

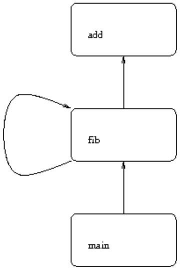
$$\mathrm{IAV}(p) = (\mathrm{AV}(S) \setminus \{x\}) \cup \bigcup \{\mathrm{IAV}(p') \mid p' \in CP(S)\} \quad (24)$$

  where $\texttt{proc}\, p(\texttt{val}\, x, \texttt{res}\, y)\, \texttt{is}^{l_n}\, S\, \texttt{end}^{l_x} \in D_*$
- CP $\Rightarrow$ procedure call graph (which procedure calls which one; see example)

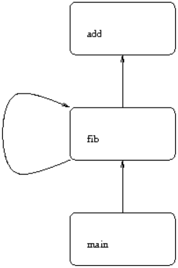
Example

```
begin   proc fib(val z) is
            if    [z < 3]
            then  [call add(a)]
            else  [call fib(z − 1)];
                  [call fib(z − 2)]
        end;
        proc add(val u) is (y := y + 1; u := 0)
        end
        y := 0; [call fib(x)]
end
```

Example



Example



$$\begin{aligned} \mathrm{IAV}(\mathit{fib}) &= (\emptyset \setminus \{z\}) \cup \mathrm{IAV}(\mathit{fib}) \cup \mathrm{IAV}(\mathit{add}) \\ \mathrm{IAV}(\mathit{add}) &= \{y, u\} \setminus \{u\} \end{aligned}$$

$\Rightarrow$ smallest solution

$$\mathrm{IAV}(\mathit{fib}) = \{y\}$$

- further extension of While-language
- *plus*: heap allocated data structures[9]
- use: warnings for illegal dereferencing
- also: “verification” for simple properties

[9] so far: global vars + stack allocated local vars

Syntax

- new: "cells" on the heap
- access via selectors:

$$sel \in \mathbf{Sel} \quad \text{selector names}$$

- example in Lisp: `car` and `cdr`
- in the notation here $x.cdr$
- here: no nested selector expressions (for simplicity)
- pointer expressions

$$p \in \mathbf{PExp}$$

$$p ::= x \mid x.sel$$

- nil: new constant

Syntax: Grammar

a	$::=$	$p \mid x \mid n \mid a \; \mathtt{op}_a \; a$	arithm. expressions
b	$::=$	$\text{true} \mid \text{false} \mid \text{not } b \mid b \; \mathtt{op}_b \; b \mid a \; \mathtt{op}_r \; a$	boolean expr.
S	$::=$	$[x := a]^l \mid [\text{skip}]^l \mid S_1 ; S_2$	statements
		$\mathtt{if}[b]^l \; \mathtt{then} \; S \; \mathtt{else} \; S \mid \mathtt{while}[b]^l \; \mathtt{do} \; S$	
	$\mid$	$[\mathtt{malloc} \; p]^l$	

Table: Abstract syntax

Syntax: Remarks

- note: no pointer arithmetic
- operations (expressions) on pointers
  - equality testing for pointers: new boolean expression
  - $\mathtt{op}_p$: some unary operators (is−nil or has−*sel* for each *sel* $\in$ **Sel**)
- assignment

$$p := a$$

two forms

- p is a variable: as before
- p is selector expression: heap update

Example: list reversal

$[y := \text{nil}]^1$

```
while   [not is−nil(x)]^2
do      ( [z := y]^3
          [y := x]^4
          [x := x.cdr]^5
          [y.cdr := z]^6 );
```

$[z := \text{nil}]^7$

State and heap

$$\xi \in \textbf{Loc} \quad \text{locations}$$

states

$$\sigma \in \textbf{State} = \textbf{Var}_* \to (\textbf{Z} + \textbf{Loc} + \{\diamond\})$$

$\diamond$: constant.

heap

$$\mathcal{H} \in \textbf{Heap} = (\textbf{Loc} \times \textbf{Sel}) \to_{fin} (\textbf{Z} + \textbf{Loc} + \{\diamond\}) \tag{25}$$

- $\to_{fin}$: partial function: newly created cells: uninitialized

Pointer expressions

semantics function for pointer expressions

$$[\![\_]\!]^{\mathcal{P}}_{\_} : \textbf{PExp}_* \to$$

$$\begin{aligned}[\![x]\!]^{\mathcal{P}}_{\sigma,\mathcal{H}} &= \\ [\![x.sel]\!]^{\mathcal{P}}_{\sigma,\mathcal{H}} &= \end{aligned}$$

Pointer expressions

semantics function for pointer expressions

$$[\![\_]\!]^{\mathcal{P}}_{\_} : \mathbf{PExp}_* \rightarrow (\mathbf{State} \times \mathbf{Heap}) \rightarrow_{fin} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

$$\begin{aligned}
[\![x]\!]^{\mathcal{P}}_{\sigma,\mathcal{H}} &= \sigma(x) \\
[\![x.sel]\!]^{\mathcal{P}}_{\sigma,\mathcal{H}} &= \begin{cases} \mathcal{H}(\sigma(x), sel) \\ \quad \text{if } \sigma(x) \in \mathbf{Loc} \text{ and } \mathcal{H} \text{ is defined on } (\sigma(x), sel) \\ undef \\ \quad \text{if } \sigma(x) \notin \mathbf{Loc} \text{ or } \mathcal{H} \text{ is undefined on } (\sigma(x), sel) \end{cases}
\end{aligned}$$

Arithmetic expressions

$$[\![\_]\!]^{\mathcal{A}}_{\_} \quad : \quad \textbf{AExp} \rightarrow (\textbf{State} \times \textbf{Heap}) \rightarrow_{fin} (\textbf{Z} + \textbf{Loc} \rightarrow \{\diamond\})$$

$$\begin{array}{rcl}
[\![p]\!]^{\mathcal{A}}_{\sigma,\mathcal{H}} & = & [\![p]\!]^{\mathcal{P}}_{\sigma,\mathcal{H}} \\
[\![n]\!]^{\mathcal{A}}_{\sigma,\mathcal{H}} & = & \mathcal{N}(n) \\
[\![a_1 \ \mathrm{op}_a \ a_2]\!]^{\mathcal{A}}_{\sigma,\mathcal{H}} & = & [\![a_1]\!]^{\mathcal{A}}_{\sigma,\mathcal{H}} \ \textbf{op}_a \ [\![a_2]\!]^{\mathcal{A}}_{\sigma,\mathcal{H}} \\
[\![\mathrm{nil}]\!]^{\mathcal{A}}_{\sigma,\mathcal{H}} & = & \diamond
\end{array}$$

- $\textbf{op}_a$: (re-)interpreted "strictly": both arguments must be defined integers

Boolean expressions

$$
\begin{array}{rcl}
[\_]^{\mathcal{B}}_{\_} & : & \textbf{BExp} \rightarrow (\textbf{State} \times \textbf{Heap}) \rightarrow_{fin} \textbf{B} \\
[a_1 \; \texttt{op}_r \; a_2]^{\mathcal{B}}_{\sigma,\mathcal{H}} & = & [a_1]^{\mathcal{A}}_{\sigma,\mathcal{H}} \; \textbf{op}_r \; [a_2]^{\mathcal{A}}_{\sigma,\mathcal{H}} \\
[\texttt{op}_p \; p]^{\mathcal{B}}_{\sigma,\mathcal{H}} & = & \textbf{op}_p \; ([p]^{\mathcal{P}}_{\sigma,\mathcal{H}})
\end{array}
$$

- $\textbf{op}_r$: likewise (re-)interpreted "strictly": both arguments must be defined and both integers or both pointers
- $\textbf{op}_p$: as needed, for instance

$$
\textbf{is}{-}\textbf{nil}(v) = \begin{cases} \mathit{true} & \text{if } v = \diamond \\ \mathit{false} & \text{otherwise} \end{cases}
$$

Semantics: statements

$$\frac{[a]^{\mathcal{A}}_{\sigma,\mathcal{H}} \text{ is defined}}{\langle [x := a]^l, \sigma, \mathcal{H}\rangle \rightarrow} \textsc{Assgn}_{\text{state}}$$

$$\frac{}{\langle [x.sel := a]^l, \sigma, \mathcal{H}\rangle \rightarrow} \textsc{Assgn}_{\text{heap}}$$

$$\frac{}{\langle [\texttt{malloc}\, x]^l, \sigma, \mathcal{H}\rangle \rightarrow} \textsc{Malloc}_{\text{state}}$$

$$\frac{\xi \text{ fresh} \qquad \sigma(x) \in \mathbf{Loc}}{\langle [\texttt{malloc}\, x.sel]^l, \sigma, \mathcal{H}\rangle \rightarrow} \textsc{Malloc}_{\text{heap}}$$

Semantics: statements

$$\frac{[a]^{\mathcal{A}}_{\sigma,\mathcal{H}} \text{ is defined}}{\langle [x := a]^l, \sigma, \mathcal{H}\rangle \rightarrow \langle \sigma[x \mapsto [a]^{\mathcal{A}}_{\sigma,\mathcal{H}}], \mathcal{H}\rangle} \ \textsc{Assgn}_{\text{state}}$$

$$\frac{\sigma(x) \in \textbf{Loc} \qquad [a]^{\mathcal{A}}_{\sigma,\mathcal{H}} \text{ is defined}}{\langle [x.sel := a]^l, \sigma, \mathcal{H}\rangle \rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto [a]^{\mathcal{A}}_{\sigma,\mathcal{H}}]\rangle} \ \textsc{Assgn}_{\text{heap}}$$

$$\frac{\xi \text{ fresh}}{\langle [\texttt{malloc}\ x]^l, \sigma, \mathcal{H}\rangle \rightarrow \langle \sigma[x \mapsto \xi], \mathcal{H}\rangle} \ \textsc{Malloc}_{\text{state}}$$

$$\frac{\xi \text{ fresh} \qquad \sigma(x) \in \textbf{Loc}}{\langle [\texttt{malloc}\ x.sel]^l, \sigma, \mathcal{H}\rangle \rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \xi], \mathcal{H}\rangle} \ \textsc{Malloc}_{\text{heap}}$$

Shape graphs

- heap can be arbitrarily large

⇒ finite, abstract representation: shape graphs $(\mathrm{S}, \mathrm{H}, \mathrm{is})$

  - abstract state: S
  - abstract heap: H
  - sharing information: is.

- 5 invariants to regulate/describe their connection

Abstract locations

- notation n_X

$$\mathbf{ALoc} = \{n_X \mid X \subseteq \mathbf{Var}_*\} \tag{26}$$

- for $x \in X$, n_X represents location $\sigma(x)$
- $n_\emptyset$: abstract summary location: locations to which the σ does not point directly.

> **Invariant 1:** If two abstract locations n_X and n_Y occur in the same shape graph, then either
> - $X = Y$, or
> - $X \cap Y = \emptyset$.

Abstract states

- abstraction of state

$\Rightarrow$ mapping var's to abstract locations

Invariant 2: If x mapped to n_X by the abstract state, then $x \in X$

$$S \in \textbf{AState} = 2^{\textbf{Var}_* \times \textbf{ALoc}} (\simeq \textbf{Var}_* \rightarrow 2^{\textbf{ALoc}}) \qquad (27)$$

- locations occurring in S:

$$ALoc(S) = \{n_x \mid \exists x.\ (x, n_X) \in S\}$$

Abstract heaps

$$\mathsf{H} \in \textbf{AHeap} = 2^{\textbf{ALoc}\times\textbf{Sel}\times\textbf{ALoc}} (= \textbf{ALoc} \times \textbf{Sel} \to 2^{\textbf{ALoc}}) \quad (28)$$

$$ALoc(\mathsf{H}) = \{n_v, n_w \mid \exists sel.\ (n_V, sel, n_W) \in \mathsf{H}\}$$

- "abstraction":

$$\begin{array}{ccc} n_V & \xrightarrow{sel} & n_W \\ \uparrow & & \uparrow \\ \xi_1 & \xrightarrow{\mathcal{H}(\_,sel)} & \xi_2 \end{array}$$

Abstract heap (2)

- concrete heap: selection is "functional"
- abstract heap: almost, but not quite, exception: $n_\emptyset$

Invariant 3: Whenever (n_V, sel, n_W) and $(n_v, sel, n_{W'})$ are in the abstract heap, then either $V = \emptyset$ or $W = W'$.

Example: list reversal

$$S_2 =$$
$$H_2 =$$

Example: list reversal

$$\begin{array}{rcl} S_2 & = & \{(x, n_{\{x\}}), (y, n_{\{y\}}), (z, n_{\{z\}})\} \\ H_2 & = & (n_{\{x\}}, \text{cdr}, n_\emptyset), (n_\emptyset, \text{cdr}, n_\emptyset), (n_{\{y\}}, \text{cdr}, n_{\{z\}}) \end{array}$$

- no edge $(n_{\{z\}}, \text{cdr}, n_\emptyset)$

Sharing information

- we have sharing for locations reachable by var's (aliasing) but not further
- we can do better

$\Rightarrow$ is

  - predicate/subset of abstract locations
  - characterizing sharing aliasing on the heap
  - contains: locations shared by pointers on the heap

- also implicit[10] sharing, sharing on the abstract heap

[10] the explicit one is the one as inherited from the real heap, and captured in is.

Sharing information

Invariant 4: If $n_X \in is$, then either

- $(n_\emptyset, sel, n_X)$ is in the abstract heap for some *sel*, or
- there exists 2 distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap (i.e., either $sel_1 \neq sel_2$ or $V \neq W$)

Invariant 5: Whenever there are 2 distinct triples (n_v, sel_1, n_X) and (n_w, sel_2, n_X) in the abstract heap and $n_X \neq n_\emptyset$, then $n_X \in is$.

Shape graphs: summary

$$
\begin{array}{llll}
\mathsf{S} & \in & \mathbf{AState} & = 2^{\mathbf{Var}_* \times \mathbf{ALoc}} \\
\mathsf{H} & \in & \mathbf{AHeap} & = 2^{\mathbf{ALoc} \times \mathbf{Sel} \times \mathbf{ALoc}} \\
\mathsf{is} & \in & \mathbf{IsShared} & = 2^{\mathbf{ALoc}}
\end{array}
$$

- shape graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ compatible
  1. $\forall n_V, n_W \in ALoc(\mathsf{S}) \cup ALoc(\mathsf{H}) \cup \mathsf{is}.\ V = W$ or $V \cap W = \emptyset$
  2. $\forall (x, n_X) \in \mathsf{S}.\ x \in X$
  3. $\forall (n_V, sel, n_W), (n_V, sel, n_{W'}) \in \mathsf{H}.\ V = \emptyset$ or $W = W'$
  4. $\forall n_X \in \mathsf{is}.$

     $$\exists sel.\ (n_\emptyset, sel, n_X) \in \mathsf{is} \lor$$
     $$\exists (n_V, sel_1, n_X), (n_W, sel_2, n_X) \in \mathsf{H}.\ sel_1 \neq sel_2 \lor V \neq W$$

  5. $(n_V, sel_1, n_X), (n_W, sel_2, n_X) \in \mathsf{H}.$
     $((sel_1 \neq sel_2 \lor V \neq W) \land X \neq \emptyset) \to n_X \in \mathsf{is}$

- set of compatible shape graphs

$$SG = \{(\mathsf{S}, \mathsf{H}, \mathsf{is}) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \text{ is compatible}\}$$

- lattice 2^{SG} (finite)
- analysis Shape
  - forward
  - may

$$\begin{aligned} \mathsf{Shape}_\circ(l) &= \begin{cases} \iota & \text{if } l = init(S \\ \bigcup\{\mathsf{Shape}_\bullet(l') \mid (l', l) \in flow(S_*)\} & \text{otherwise} \end{cases} \\ \mathsf{Shape}_\bullet(l) &= f_l^{\mathsf{SA}}(\mathsf{Shape}_\circ(l))1 \end{aligned} \quad (29)$$

Example: list reversal

$[y := \text{nil}]^1$

`while` $[\text{not is}-\text{nil}(x)]^2$

`do` $(\; [z := y]^3$

$[y := x]^4$

$[x := x.\text{cdr}]^5$

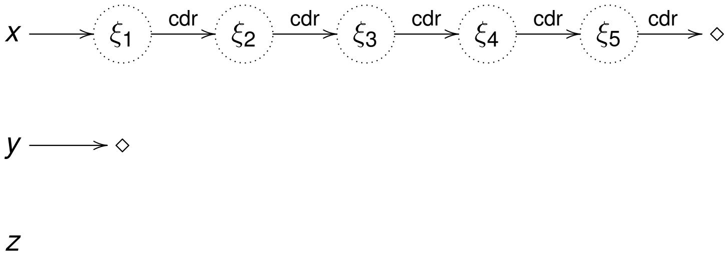
$[y.\text{cdr} := z]^6 \;);$

$[z := \text{nil}]^7$

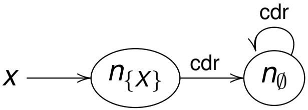
Example: list reversal

$$
\begin{array}{lclcl}
\mathrm{Shape}_\bullet(1) & = & f_1^{\mathrm{SA}}(\mathrm{Shape}_\circ(1)) & = & f_1^{\mathrm{SA}}(\iota) \\
\mathrm{Shape}_\bullet(2) & = & f_2^{\mathrm{SA}}(\mathrm{Shape}_\circ(2)) & = & f_2^{\mathrm{SA}}(\mathrm{Shape}_\bullet(1) \cup \mathrm{Shape}_\bullet(6)) \\
\mathrm{Shape}_\bullet(3) & = & f_3^{\mathrm{SA}}(\mathrm{Shape}_\circ(3)) & = & f_3^{\mathrm{SA}}(\mathrm{Shape}_\bullet(2)) \\
\mathrm{Shape}_\bullet(4) & = & f_4^{\mathrm{SA}}(\mathrm{Shape}_\circ(4)) & = & f_4^{\mathrm{SA}}(\mathrm{Shape}_\bullet(3)) \\
\mathrm{Shape}_\bullet(5) & = & f_5^{\mathrm{SA}}(\mathrm{Shape}_\circ(5)) & = & f_5^{\mathrm{SA}}(\mathrm{Shape}_\bullet(4)) \\
\mathrm{Shape}_\bullet(6) & = & f_6^{\mathrm{SA}}(\mathrm{Shape}_\circ(6)) & = & f_6^{\mathrm{SA}}(\mathrm{Shape}_\bullet(5)) \\
\mathrm{Shape}_\bullet(7) & = & f_7^{\mathrm{SA}}(\mathrm{Shape}_\circ(7)) & = & f_7^{\mathrm{SA}}(\mathrm{Shape}_\bullet(2))
\end{array}
$$

Example: list reversal, initial value



Example: list reversal, initial value



Transfer function

- $f_l^{\mathrm{SA}} : 2^{\mathrm{SG}} \to 2^{\mathrm{SG}}$
- defined *pointwise:*

$$f_l^{\mathrm{SA}}(\mathrm{SG}) = \tag{30}$$

Transfer function

- $f_l^{\mathsf{SA}} : 2^{\mathsf{SG}} \rightarrow 2^{\mathsf{SG}}$
- defined *pointwise:*

$$f_l^{\mathsf{SA}}(\mathsf{SG}) = \bigcup \{\Phi_l^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \in SG\} \qquad (30)$$

with

$$\Phi_l^{\mathsf{SA}} : \mathsf{SG} \rightarrow 2^{\mathsf{SG}} \qquad (31)$$

Side-effect free commands

- for $[b]^l$ and $[\text{skip}]^l$

Side-effect free commands

- for $[b]^l$ and $[\text{skip}]^l$
- trivial

$$\Phi_l^{SA}((S, H, is)) = (S, H, is)$$

Assignment (1)

- assignment of value to variable

$$[x := a]^l \quad \text{where } a \text{ is } n,\ a_1 \underset{a}{\mathrm{op}} a_2,\ \text{nil}$$

- "renaming" of locations

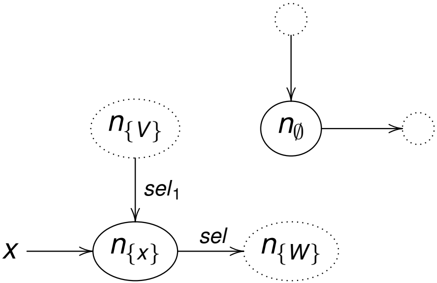
$$k_x(n_Z) = n_{Z \setminus \{x\}}$$

$$\Phi_l^{SA}((\mathrm{S}, \mathrm{H}, \mathrm{is})) = \{kill_x((\mathrm{S}, \mathrm{H}, \mathrm{is}))\}$$

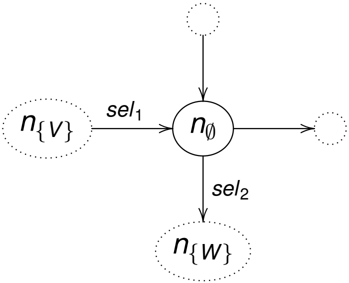
$kill_x((\mathrm{S}, \mathrm{H}, \mathrm{is})) = ((\acute{\mathrm{S}}, \acute{\mathrm{H}}, \acute{\mathrm{is}}))$:

$$\begin{array}{lcl} \acute{\mathrm{S}} & = & \{(z, k_x(n_Z)) \mid (z, n_Z) \in \mathrm{S} \quad z \neq x\} \\ \acute{\mathrm{H}} & = & \{(k_x(n_V), sel, k_k(n_W)) \mid (n_v, sel, n_W) \in \mathrm{H}\} \\ \acute{\mathrm{is}} & = & \{k_x(n_X) \mid n_X \in \mathrm{is}\} \end{array}$$

Assignment (1)



Assignment (1)



Assignment (2)

- assignment of variable to variable

$$x := y \quad \text{where } x \neq y$$

- the overriding for x: with the $kill_x$ as before

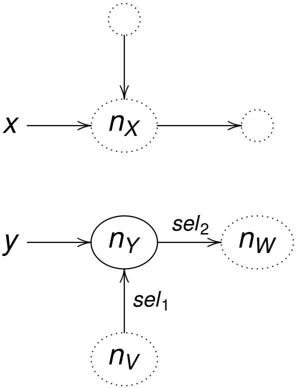
$$g_x^y(n_Z) \quad = \quad \begin{cases} n_{Z \cup \{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases}$$

$$\Phi_I^{SA}((\mathrm{S}, \mathrm{H}, \mathrm{is})) = \{\mathrm{S}'', \mathrm{H}'', \mathrm{is}''\}$$

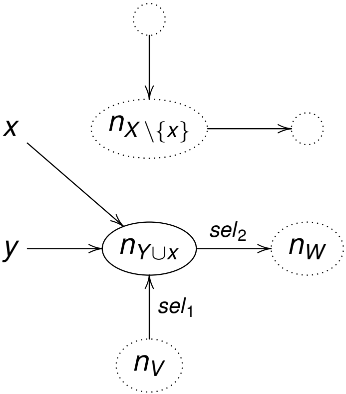
where $(\mathrm{S}', \mathrm{H}', \mathrm{is}') = kill_x((\mathrm{S}, \mathrm{H}, \mathrm{is}))$ and

$$\begin{aligned} \mathrm{S}'' &= \{(z, g_x^y(n_Z)) \mid (z, n_Z) \in \mathrm{S}'\} \\ &\quad \cup \{(x, g_x^y(n_Y)) \mid (y', n_Y) \in \mathrm{S}', y' = y\} \\ \mathrm{H}'' &= \{(g_x^y(n_V), sel, g_x^y(n_W)) \mid n_V, sel, n_W \in \mathrm{H}'\} \\ \mathrm{is}'' &= \{g_x^y(n_Z) \mid n_Z \in \mathrm{is}'\} \end{aligned}$$

Assignment (2)



Assignment (2)



Assignment (3.a)

- Assignment of "selector" to variable

$$[x := y.sel]^{l} \quad \text{where } y = x$$

equivalent to

$$[t := y.sel]^{l_1}, [x := t]^{l_2}; [t := \text{nil}]^{l_3}$$

Assignment (3.b)

- Assignment of ”selector” to variable

$$[x := y.sel]^l \quad \text{where } y \neq x$$

1. first step: $(\mathrm{S}', \mathrm{H}', \mathrm{is}') = kill_x((\mathrm{S}, \mathrm{H}, \mathrm{is}))$
2. “rename” abstract location appropriately
   1. y or $y.sel$ is an integer, undefined, or nil
   2. $y.sel$ defined and pointed at by some other variable (U)
   3. $y.sel$ defined but not pointed at by some other variable

Assignment (3.b.1)

- either:
  1. no abstract location n_Y s.t. $(y, n_Y) \in \mathsf{S}'$ or
  2. there is an n_Y s.t. $(y, n_Y) \in \mathsf{S}'$ but no n s.t. $(n_y, sel, n) \in \mathsf{H}'$.
- case 1: nothing changes:

$$\Phi_l^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is}) = \{kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))\}$$

$$[x := y.sel]^l \quad \text{where } y \neq x$$

- conditions

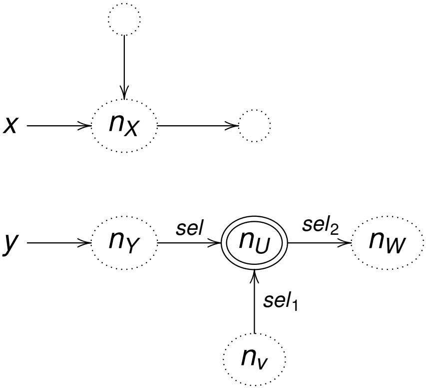
$$(y, n_Y) \in \mathsf{S}' \quad \text{and} \quad (n_Y, sel, n_U) \in \mathsf{H}'$$

$$h_x^U(n_Z) = \begin{cases} n_{U \cup \{x\}} & \text{if } Z = U \\ n_z & \text{otherwise} \end{cases}$$

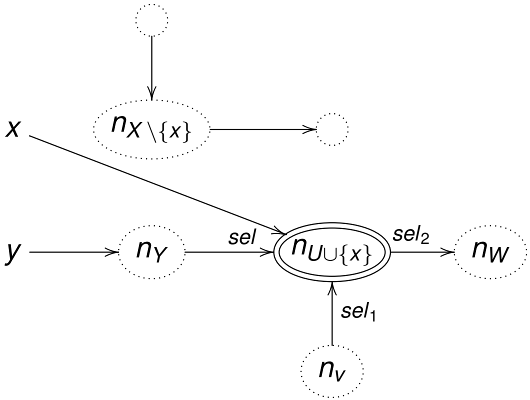
$$\Phi_l^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')\}$$

$$\begin{aligned} \mathsf{S}'' &= \{(z, \mathbf{h}_x^U(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \cup \{(x, \mathbf{h}_x^U(n_U))\} \\ \mathsf{H}' &= \{(\mathbf{h}_x^U(n_V), sel', h_x^U(n_W)) \mid (n_V, sel', n_W) \in \mathsf{H}'\} \\ \mathsf{is}' &= \{\mathbf{h}_x^U(n_Z) \mid n_Z \in \mathsf{is}'\} \end{aligned}$$

Assignment (3.b.2)



Assignment (3.b.2)



Assignment (3.b.3)

$[x := y.sel]^l$ where $y \neq x$

- conditions

$$(y, n_Y) \in \mathsf{S}' \quad \text{and} \quad (n_Y, sel, n_\emptyset) \in \mathsf{H}'$$

- required: *new* abstact location for x: "split" $n_\emptyset$

Assignment (3.b.3)

consider conceptually

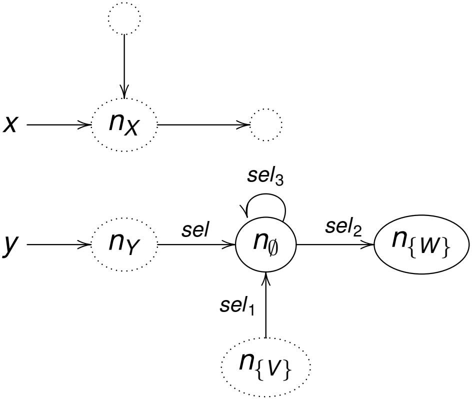
$$x := \text{nil}; [x := y.sel]^l; x := \text{nil}$$

$$\begin{array}{rl}\Phi_l^{SA}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \mid & (\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \quad \text{is compatible}, \\ & kill_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')) = (\mathsf{S}', \mathsf{H}', \mathsf{is}'), \\ & (x, n_{\{x\}}) \in \mathsf{S}'', \\ & (n_Y, sel, n_{\{x\}}) \in \mathsf{H}''\}\end{array}$$

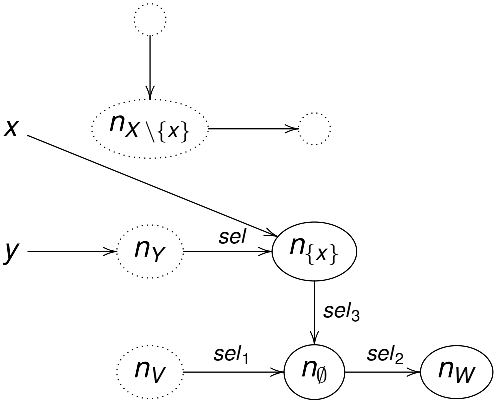
$$(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

Start configs

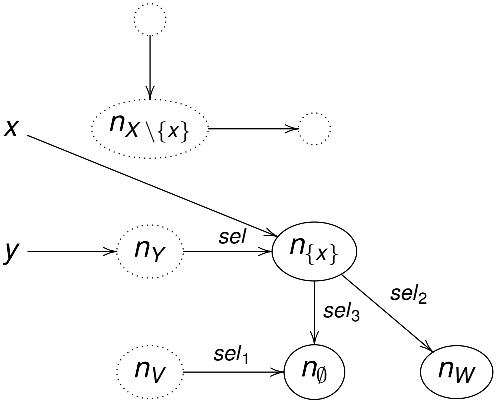
note in the example: $n_\emptyset$ and n_W are not shared!



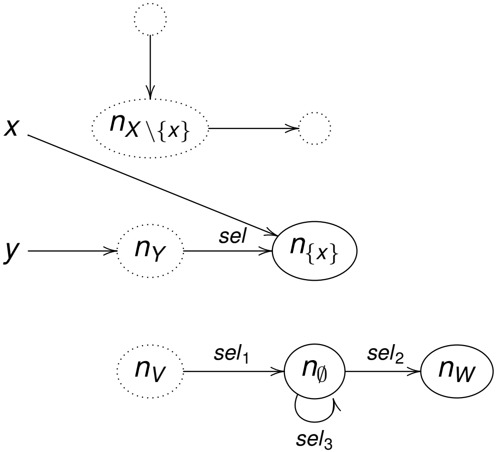
Result configs



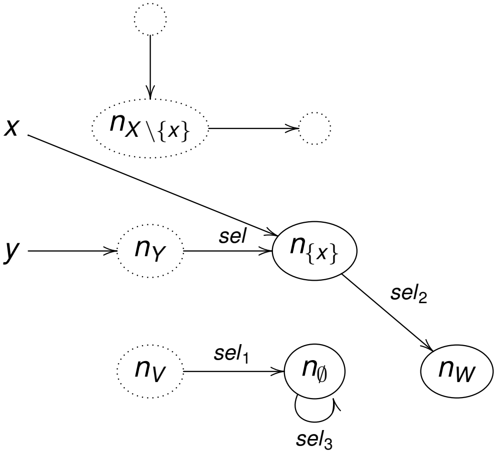
Result configs



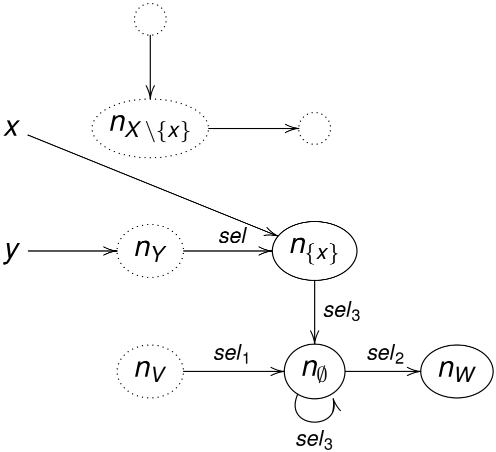
Result configs



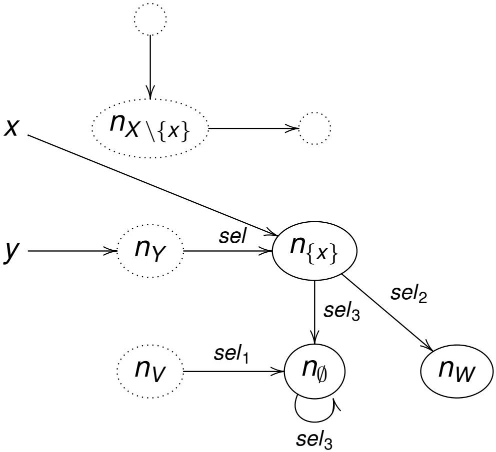
Result configs



Result configs



Result configs



- assignment of value to selector

$$[x.sel := a]^l \quad \text{where } a \text{ is } n,\ a_1 \mathop{\mathrm{op}}_a a_2,\ \text{nil}$$

Assume:

$$(x, n_X) \in \mathsf{S} \quad \text{and} \quad (n_X, sel, n_U) \in \mathsf{H}$$

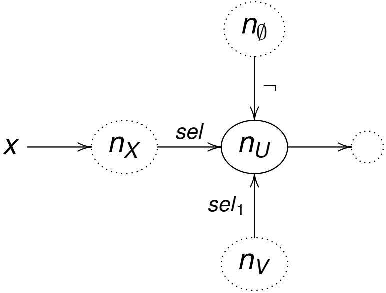
$$\Phi_l^{SA}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}(\mathsf{S}, \mathsf{H}, \mathsf{is})\} = \{(\mathsf{S}', \mathsf{H}', \mathsf{is}')\}$$

$$\mathsf{S}' = \mathsf{S}$$

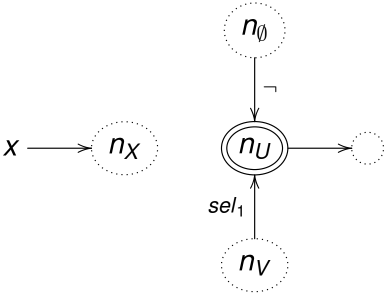
$$\mathsf{H}' = \{(n_V, sel', n_W) \mid (n_V, sel', n_W) \in \mathsf{H}, \neg(X = V \wedge sel = sel')\}$$

$$\mathsf{is}' = \begin{cases} \mathsf{is} \setminus \{n_U\} & \text{if } n_U \in \mathsf{is},\ |into(n_U, \mathsf{H}')| \leq 1,\quad n_U \in \mathsf{is}, \\ & \qquad \neg\exists sel'.\ (n_\emptyset, sel', n_U) \in \mathsf{H}' \\ \mathsf{is} & \text{otherwise} \end{cases}$$

Assignment 4



Assignment 4

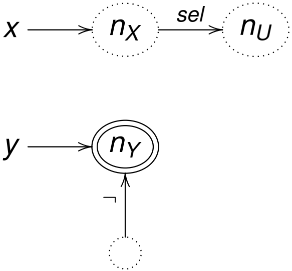


Assignment 5

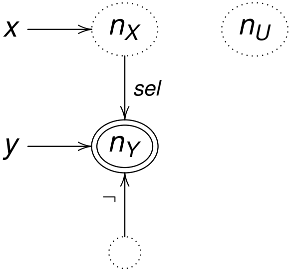
- assignment of value to selector

$$[x.sel := y]^l$$

Assignment 5



Assignment 5



Assignment 6

- assignment of selector to selector

$$[x.sel := y.sel']^{l}$$

- decompose into

$$[t := y.sel']^{l_1}; [x.sel := t]^{l_2}; [t := \text{nil}]^{l_3}$$

Malloc

- `malloc` x

$$\Phi_l^{SA}((S,H,is)) = \{(S' \cup \{(x, n_{\{x\}})\}), H', is'\} \quad \text{and} \quad (S', H', is') = k$$

References I

[1] A. W. Appel.
Modern Compiler Implementation in ML.
Cambridge University Press, 1998.

[2] F. Nielson, H.-R. Nielson, and C. L. Hankin.
Principles of Program Analysis.
Springer-Verlag, 1999.