Chapter 1

Introduction

Course "Static analysis and all that"

Martin Steffen

IN5440 / autum 2018

Chapter 1

Learning Targets of Chapter “Introduction”.

Apart from a motivational introduction, the chapter gives a high-level overview over larger topics covered in the lecture. They are treated hear just as a teaser and in less depth compared to later but there is already technical content.

Chapter 1

Outline of Chapter “Introduction”.

Motivation

Data flow analysis

Constraint-based analysis

Type and effect systems

Algorithms

Section

Motivation

Chapter 1 “Introduction”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Static analysis: why and what?

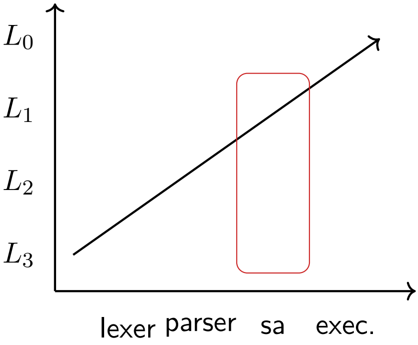
- what
  - *static*: at "compile time"
  - *analysis*: deduction of program properties
    - automatic/decidable
    - formally, based on semantics
- why
  - error catching
- catching common "stupid" errors without bothering the user much
- spotting errors early
- certain similarities to model checking
- examples: type checking, uninitialized variables, potential nil-pointer deref's, unused code
- optimization: based on analysis, transform the "code"[1], such the the result is "better"
  - examples: precalculation of results, optimized register allocation ...

[1] source code, intermediate code at various levels

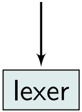
The nature of static analysis

- compiler with differerent *phases*
- corresponding to *Chomsky's hierarchy*
- static = in principle: before run-time, but in praxis, "*context-free*"
- since: run-time most often: undecidable

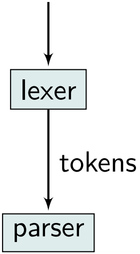
⇒ static analysis as approximation



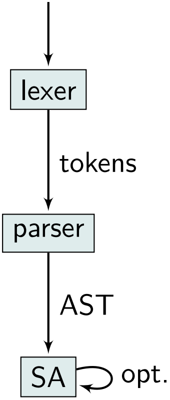
Phases



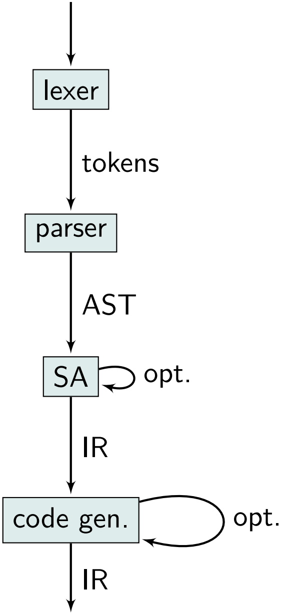
Phases



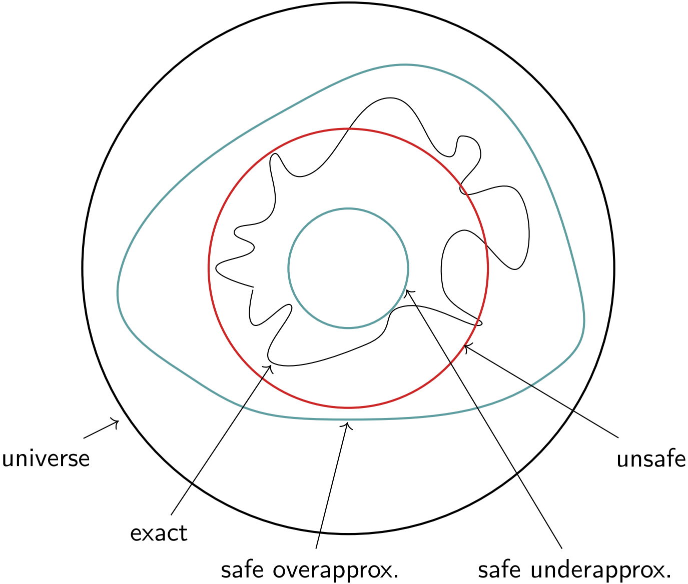
Phases



Phases



Static analysis as approximation



Optimal compiler?

Full employment theorem for compiler writers

It's a (mathematically proven!) fact that for any compiler, there exists another one which beats it.

- slightly more than *non-existence* of optimal compiler or *undecidability* of such a compiler
- theorem
  - just states that there room for improvement is always *guaranteed*
  - does not say *how!* Finding a better one: *undecidable*

Section

Data flow analysis

Chapter 1 “Introduction”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

While-language

- simple, prototypical imperative language
  - "untyped"
  - simple control structure: while, conditional, sequencing
  - simple data (numerals, booleans)
- abstract syntax $\neq$ concrete syntax
- disambiguation when needed: (...), or { ... } or begin ... end

Labelling

- associate *flow* information

⇒ labels

- *elementary block* = labelled item
- identify basic building blocks
- consistent/unique labelling

Abstract syntax

a	$::=$	$x \mid n \mid a \, \mathtt{op}_a \, a$	arithm. expressions
b	$::=$	$\mathsf{true} \mid \mathsf{false} \mid \mathsf{not}\ b \mid b \, \mathtt{op}_b \, b \mid a \, \mathtt{op}_r \, a$	boolean expr.
S	$::=$	$x := a \mid \mathsf{skip} \mid S_1; S_2$	statements
		$\mathtt{if}\, b \,\mathtt{then}\, S \,\mathtt{else}\, S \mid \mathtt{while}\, b \,\mathtt{do}\, S$	

Table: Abstract syntax

Abstract syntax

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

Table: Labelled abstract syntax

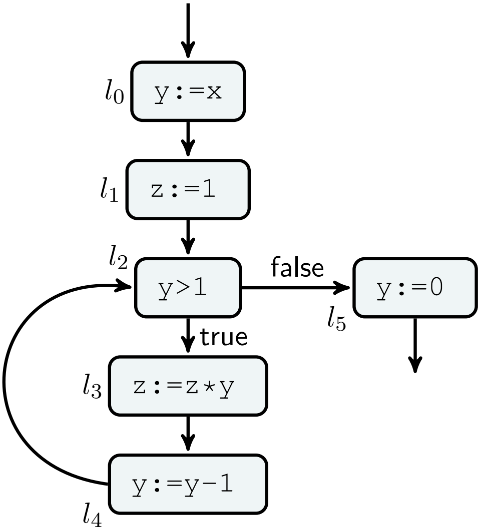
Example factorial

$$y := x; z := 1; \texttt{while}\ y > 1\, \texttt{do}(z := z * y; y := y - 1); y := 0$$

- input variable: x
- output variable: z

$$\begin{array}{l}[y := x]^0;\\ [z := 1]^1;\\ \ \texttt{while}\ [y > 1]^2\\ \ \texttt{do}([z := z * y]^3; [y := y - 1]^4);\\ \ [y := 0]^5\end{array} \tag{1}$$

CFG factorial



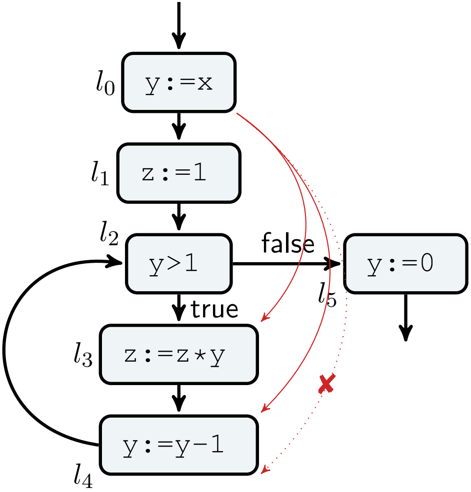
Factorial: reaching definitions analysis

- "definition" of x: assignment to x: $x := a$
- better name: reaching assignment analysis
- first, simple example of data flow analysis

Reaching def's

An *assignment* (= "definition") $[x := a]^l$ *may* reach a program point, if there *exists* an execution where x was *last assigned to* at l, when the mentioned program point is reached.

Factorial: reaching definitions



- data of interest: tuples of variable × label (or node)
- note: *distinguish* between *entry* and *exit* of a node.

Factorial: reaching assignments

- " points " in the program: *entry* and *exit* to elementary blocks/labels
- $?$: special label (not occurring otherwise), representing *entry* to the program, i.e., $(x, ?)$ represents initial (uninitialized) value of x
- full information: pair of "functions"

$$\mathsf{RD} = (\mathsf{RD}_{entry}, \mathsf{RD}_{exit}) \tag{2}$$

- tabular form (array): see next slide

Factorial: reaching assignments table

l	RD_{entry}	RD_{exit}
0	$(x,?),(y,?),(z,?)$	$(x,?),(y,0),(z,?)$
1	$(x,?),(y,0),(z,?)$	$(x,?),(y,0),(z,1)$
2	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,0),(y,4),(z,1),(z,3)$
3	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,0),(y,4),\quad (z,3)$
4	$(x,?),(y,0),(y,4),\quad (z,3)$	$(x,?),\quad (y,4),\quad (z,3)$
5	$(x,?),(y,0),(y,4),(z,1),(z,3)$	$(x,?),(y,5),\quad (z,1),(z,3)$

Reaching assignments: remarks

- *elementary* blocks of the form
  - $[b]^l$: entry/exit information coincides
  - $[x := a]^l$: entry/exit information (in general) different
- at program exit: $(x, ?)$, x is input variable
- table: *"best"* information = *smallest* sets:
  - additional pairs in the table: still *safe*
  - removing labels: *unsafe*
- note: still an approximation
  - no *real* (= run time) data, no real execution, only *data flow*
  - *approximate* since
    - in *concrete* runs: at each point *in that run*, there is exactly *one* last assignment, not a *set*
    - *label* represents (potentially infinitely many) runs
  - e.g.: at program exit in concrete run: *either* $(z, 1)$ or else $(z, 3)$

Data flow analysis

- standard: representation of program as control flow graph (aka flow graph)
  - nodes: elementary blocks with labels (or basic block)
  - edges: flow of control
- two approaches, both (especially here) quite similar
  - *equational* approach
  - *constraint-based* approach

From flow graphs to equations

- associate an equation system with the flow graph:
  - describing the "flow of information"
  - here:
    - the information related to reaching assignments
    - information imagined to flow forwards
- solutions of the equations
  - describe *safe* approximations
  - not unique, interest in the *least* (or *largest*) solution
  - here: give back RD of equation (2) on slide 22

Equations for RD and factorial: intra-block

first type: *local*, intra-block":

- flow through each individual block
- relating for each elementary block its exit with its entry

elementary block: $[y := x]^0$

$$\mathsf{RD}_{exit}(0) \quad = \quad \mathsf{RD}_{entry}(0) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,0)\} \tag{3}$$

Equations for RD and factorial: intra-block

first type: *local*, intra-block":

- flow through each individual block
- relating for each elementary block its exit with its entry

elementary block: $[y > 1]^2$

$$\begin{aligned} \mathsf{RD}_{exit}(0) &= \mathsf{RD}_{entry}(0) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,0)\} \\ \mathsf{RD}_{exit}(2) &= \mathsf{RD}_{entry}(2) \end{aligned} \tag{3}$$

Equations for RD and factorial: intra-block

first type: *local*, intra-block":

- flow through each individual block
- relating for each elementary block its exit with its entry

all equations with RD_{exit} as "left-hand side"

$$
\begin{array}{rcl}
\mathsf{RD}_{exit}(0) & = & \mathsf{RD}_{entry}(0) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,0)\} \\
\mathsf{RD}_{exit}(1) & = & \mathsf{RD}_{entry}(1) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \cup \{(z,1)\} \\
\mathsf{RD}_{exit}(2) & = & \mathsf{RD}_{entry}(2) \\
\mathsf{RD}_{exit}(3) & = & \mathsf{RD}_{entry}(3) \setminus \{(z,l) \mid l \in \mathbf{Lab}\} \cup \{(z,3)\} \\
\mathsf{RD}_{exit}(4) & = & \mathsf{RD}_{entry}(4) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,4)\} \\
\mathsf{RD}_{exit}(5) & = & \mathsf{RD}_{entry}(5) \setminus \{(y,l) \mid l \in \mathbf{Lab}\} \cup \{(y,5)\}
\end{array}
\tag{3}
$$

Inter-block flow

second type: *global*, inter-block

- flow *between* the elementary blocks, following the control-flow edges
- relating the *entry* of each block with the *exits* of *other* blocks, that are connected via an *edge* (exception: the initial block has no incoming edge)
- *initial* block: mark variables as *uninitialized*

$$\mathsf{RD}_{entry}(1) = \mathsf{RD}_{exit}(0) \tag{4}$$

$$\begin{aligned}
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(2)
\end{aligned}$$

Inter-block flow

second type: *global*, inter-block

- flow *between* the elementary blocks, following the control-flow edges
- relating the *entry* of each block with the *exits* of *other* blocks, that are connected via an *edge* (exception: the initial block has no incoming edge)
- *initial* block: mark variables as *uninitialized*

$$
\begin{aligned}
\mathsf{RD}_{entry}(1) &= \mathsf{RD}_{exit}(0) \\
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(2)
\end{aligned} \tag{4}
$$

Inter-block flow

second type: *global*, inter-block

- flow *between* the elementary blocks, following the control-flow edges
- relating the *entry* of each block with the *exits* of *other* blocks, that are connected via an *edge* (exception: the initial block has no incoming edge)
- *initial* block: mark variables as *uninitialized*

$$
\begin{aligned}
\mathsf{RD}_{entry}(1) &= \mathsf{RD}_{exit}(0) \\
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(2)
\end{aligned}
\tag{4}
$$

$$
\mathsf{RD}_{entry}(0) = \{(x, ?), (y, ?), (z, ?)\}
$$

General scheme (for RD)

Intra

- for assignments $[x := a]^l$

$$\mathsf{RD}_{exit}(l) = \mathsf{RD}_{entry}(l) \setminus \{(x, l') \mid l' \in \mathbf{Lab}\} \cup \{(x, l)\} \tag{5}$$

- for other blocks $[b]^l$ (side-effect free)

$$\mathsf{RD}_{exit}(l) = \mathsf{RD}_{entry}(l) \tag{6}$$

Inter

$$\mathsf{RD}_{entry}(l) = \bigcup_{l' \to l} \mathsf{RD}_{exit}(l') \tag{7}$$

Initial l: label of the initial block (isolated entry)

$$\mathsf{RD}_{entry}(l) = \{(x, ?) \mid x \text{ is a program variable}\} \tag{8}$$

The equation system as fix point

- RD example: solution to the equation system = *12 sets*

$$\mathsf{RD}_{entry}(0), \ldots, \mathsf{RD}_{exit}(5)$$

- i.e., the $\mathsf{RD}_{entry}(l), \mathsf{RD}_{exit}(l)$ are the *variables* of the equation system, of *type*: sets of pairs of the form (x, l)
- *domain* of the equation system:
- $\vec{\mathsf{RD}}$: the mentioned twelve-tuple of variables

⇒ equation system understood as function F

Equations

$$\vec{\mathsf{RD}} = F(\vec{\mathsf{RD}})$$

The least solution

- $\mathbf{Var}_* =$ variables "of interest" (i.e., occurring), $\mathbf{Lab}_*$: labels of interest
- here $\mathbf{Var}_* = \{x, y, z\}$, $\mathbf{Lab}_* = \{?, 1, \ldots, 6\}$

$$F : (2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12} \to (2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12} \qquad (9)$$

- domain $(2^{\mathbf{Var}_* \times \mathbf{Lab}_*})^{12}$: *partially ordered* pointwise:

$$\vec{\mathsf{RD}} \sqsubseteq \vec{\mathsf{RD}}' \text{ iff } \forall i.\ \mathsf{RD}_i \subseteq \mathsf{RD}'_i \qquad (10)$$

$\Rightarrow$ complete lattice

Constraint-based approach

- next, for DFA: a simple *variant* of the equational approach
- *rearrangement* of the entry-exit relationships
- instead of equations: *inequations* (sub-set instead of set-equality)
- in more complex settings: constraints become more complex, no split in exit- and entry-constraints

Factorial program: intra-block constraints

elementary block: $[y := x]^0$

$$\mathsf{RD}_{exit}(0) \supseteq \mathsf{RD}_{entry}(0) \setminus \{(y, l) \mid l \in \mathbf{Lab}\}$$
$$\mathsf{RD}_{exit}(0) \supseteq \{(y, 0)\}$$

Factorial program: intra-block constraints

elementary block: $[y > 1]^2$

$$\mathsf{RD}_{exit}(2) \supseteq \mathsf{RD}_{entry}(2)$$

Factorial program: intra-block constraints

all equations with RD_{exit} as left-hand side

$$
\begin{aligned}
&\mathsf{RD}_{exit}(0) \supseteq \mathsf{RD}_{entry}(0) \setminus \{(y, l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(0) \supseteq \{(y, 0)\}\\
&\mathsf{RD}_{exit}(1) \supseteq \mathsf{RD}_{entry}(1) \setminus \{(z, l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(1) \supseteq \{(z, 1)\}\\
&\mathsf{RD}_{exit}(2) \supseteq \mathsf{RD}_{entry}(2)\\
&\mathsf{RD}_{exit}(3) \supseteq \mathsf{RD}_{entry}(3) \setminus \{(z, l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(3) \supseteq \{(z, 3)\}\\
&\mathsf{RD}_{exit}(4) \supseteq \mathsf{RD}_{entry}(4) \setminus \{(y, l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(4) \supseteq \{(y, 4)\}\\
&\mathsf{RD}_{exit}(5) \supseteq \mathsf{RD}_{entry}(5) \setminus \{(y, l) \mid l \in \mathbf{Lab}\}\\
&\mathsf{RD}_{exit}(5) \supseteq \{(y, 5)\}
\end{aligned}
$$

Factorial program: inter-block constraints

cf. slide 30 ff.: inter-block equations:

$$
\begin{aligned}
\mathsf{RD}_{entry}(1) &= \mathsf{RD}_{exit}(0) \\
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(2)
\end{aligned}
$$

$$
\mathsf{RD}_{entry}(0) = \{(x, ?), (y, ?), (z, ?)\}
$$

Factorial program: inter-block constraints

splitting of composed right-hand sides + using $\supseteq$ instead of $=$:

$$
\begin{array}{rcl}
\mathsf{RD}_{entry}(1) & \supseteq & \mathsf{RD}_{exit}(0) \\
\mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(4) & \supseteq & \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) & \supseteq & \mathsf{RD}_{exit}(2)
\end{array}
$$

$$
\mathsf{RD}_{entry}(1) \supseteq \{(x,?),(y,?),(z,?)\}
$$

Least solution revisited

instead of $F(\vec{\mathsf{RD}}) = \vec{\mathsf{RD}}$

- clear: solution to the equation system $\Rightarrow$ solution to the constraint system
- important: least solutions *coincides*!

Pre-fixpoint

$$F(\vec{\mathsf{RD}}) \sqsubseteq \vec{\mathsf{RD}} \tag{11}$$

Section

Constraint-based analysis

Chapter 1 “Introduction”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Control-flow analysis

Goal CFA

which elem. blocks lead to which other elem. blocks

- for while-language: immediate (labelled elem. blocks, resp., graph)
- complex for: more *advanced* features, *higher-order* languages, oo languages ...
- here: prototypical higher-order functional language λ-calculus
- formulated as constraint-based analysis

Simple example

```
let f = fn x => x 1;
    g = fn y => y + 2;
    h = fn z => z + 3;
in (f g) + (f h)
```

- *higher-order* function f
- for simplicity: untyped
- local definitions via let-in
- interesting *above:* $x\ 1$

Goal (more specifically)

For each function application, which function may be applied.

Labelling

- more complex language ⇒ more *complex* labelling
- "elem. blocks" can be *nested*
- *all* syntactic constructs (expressions) are labelled
- consider:

Unlabelled abstract syntax

$$(\mathtt{fn}\, x \Rightarrow x)\ (\mathtt{fn}\, y \Rightarrow y)$$

- functional language: side-effect free

⇒ *no* need to distinguish *entry* and *exit* of labelled blocks.

Labelling

- more complex language $\Rightarrow$ more *complex* labelling
- "elem. blocks" can be *nested*
- *all* syntactic constructs (expressions) are labelled
- consider:

Full labelling

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

- functional language: side-effect free

$\Rightarrow$ *no* need to distinguish *entry* and *exit* of labelled blocks.

Data of the analysis

Data of the analysis:

Pairs $(\hat{C}, \hat{\rho})$ of mappings:

abstract cache: $\hat{C}(l)$: set of values/function abstractions, the subexpression labelled l may evaluate to

abstract env.: $\hat{\rho}$: values, x may be bound to

The constraint system

- ignoring "`let`" here: *three* syntactic constructs ⇒ *three* kinds of constraints
- relating $\hat{C}$, $\hat{\rho}$, and the program in form of subset constraints (subsets, order-relation)

The constraint system

- ignoring "`let`" here: *three* syntactic constructs $\Rightarrow$ *three* kinds of constraints
- relating $\hat{C}$, $\hat{\rho}$, and the program in form of subset constraints (subsets, order-relation)

3 syntactic classes

- function abstraction: $[\mathtt{fn}\, x \Rightarrow x]^l$
- variables: $[x]^l$
- application: $[f\ g]^l$

Constraint system for the small example

Labelled example

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

Constraint system for the small example

Labelled example

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

- function abstractions

$$\begin{array}{rcl} \{\mathtt{fn}\, x \Rightarrow [x]^1\} & \subseteq & \hat{C}(2) \\ \{\mathtt{fn}\, y \Rightarrow [y]^3\} & \subseteq & \hat{C}(4) \end{array}$$

Constraint system for the small example

Labelled example

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

- variables (occurrences of use)

$$\begin{array}{lcl} \hat{\rho}(x) & \subseteq & \hat{C}(1) \\ \hat{\rho}(y) & \subseteq & \hat{C}(3) \end{array}$$

Constraint system for the small example

Labelled example

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

- application: connecting function entry and (body) exit with the argument

$$\begin{aligned} \hat{C}(4) &\subseteq \hat{\rho}(x) \\ \hat{C}(1) &\subseteq \hat{C}(5) \end{aligned}$$

Constraint system for the small example

Labelled example

$$[\ [\mathtt{fn}\,x \Rightarrow [x]^1]^2\ [\mathtt{fn}\,y \Rightarrow [y]^3]^4\ ]^5$$

- application: connecting function entry and (body) exit with the argument but:
- also $[\mathtt{fn}\,y \Rightarrow [y]^3]^4$ is a candidate at 2! (according to $\hat{C}(2)$)

$$\begin{array}{rcl}
\hat{C}(4) & \subseteq & \hat{\rho}(x) \\
\hat{C}(1) & \subseteq & \hat{C}(5) \\
\hat{\mathbf{C}}(\mathbf{4}) & \subseteq & \hat{\rho}(y) \\
\hat{C}(3) & \subseteq & \hat{\mathbf{C}}(\mathbf{5})
\end{array}$$

Constraint system for the small example

Labelled example

$$[\ [\mathtt{fn}\, x \Rightarrow [x]^1]^2\ [\mathtt{fn}\, y \Rightarrow [y]^3]^4\ ]^5$$

$$\begin{array}{rcccc}
\{\mathtt{fn}\, x \Rightarrow [x]^1\} \subseteq \hat{C}(2) & \Rightarrow & \hat{C}(4) & \subseteq & \hat{\rho}(x) \\
\{\mathtt{fn}\, x \Rightarrow [x]^1\} \subseteq \hat{C}(2) & \Rightarrow & \hat{C}(1) & \subseteq & \hat{C}(5) \\
\{\mathtt{fn}\, y \Rightarrow [y]^3\} \subseteq \hat{C}(2) & \Rightarrow & \hat{\mathbf{C}}(\mathbf{4}) & \subseteq & \hat{\rho}(y) \\
\{\mathtt{fn}\, y \Rightarrow [y]^3\} \subseteq \hat{C}(2) & \Rightarrow & \hat{C}(3) & \subseteq & \hat{\mathbf{C}}(\mathbf{5})
\end{array}$$

The least (= best) solution

$$
\begin{array}{rcl}
\hat{C}(1) & = & \{\texttt{fn}\, y \Rightarrow [y]^3\} \\
\hat{C}(2) & = & \{\texttt{fn}\, x \Rightarrow [x]^1\} \\
\hat{C}(3) & = & \emptyset \\
\hat{C}(4) & = & \{\texttt{fn}\, y \Rightarrow [y]^3\} \\
\hat{C}(5) & = & \{\texttt{fn}\, y \Rightarrow [y]^3\} \\
\hline
\hat{\rho}(x) & = & \{\texttt{fn}\, y \Rightarrow [y]^3\} \\
\hat{\rho}(y) & = & \emptyset
\end{array}
$$

One interesting bit here in the solution is: $\hat{\rho}(y) = \emptyset$: that means, the variable y never evaluated, i.e., the function is not applied at all.

Section

Type and effect systems

Chapter 1 “Introduction”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Effects: Intro

- type system: "classical" static analysis:

$$t : T$$

- *judgment*: "term or program phrase has type T"
- in general: *context-sensitive* judgments (remember Chomsky ...)

Judgement :

$$\Gamma \vdash t : \tau$$

- Γ: *assumption* or *context*
- here: *"non-standard"* type systems: effects and annotations
- natural setting: typed languages, here: *trivial!* setting (while-language)

"Trival" type system

- setting: while-language
- each statement maps: state to states
- Σ: type of *states*

judgement

$$\vdash S : \Sigma \to \Sigma \tag{12}$$

- specified as a *derivation* system
- note: *partial* correctness assertion

"Trival" type system: rules

$$\vdash [x := a]^l : \Sigma \to \Sigma \qquad \textsc{Ass}$$

$$[\mathsf{skip}]^l : \Sigma \to \Sigma \qquad \textsc{Skip}$$

$$\frac{\vdash S_1 : \Sigma \to \Sigma \qquad S_2 : \Sigma \to \Sigma}{\vdash S_1; S_2 : \Sigma \to \Sigma} \textsc{Seq}$$

‘

$$\frac{\vdash S : \Sigma \to \Sigma}{\vdash \mathtt{while}[b]^l\,\mathtt{do}\,S : \Sigma \to \Sigma} \textsc{While}$$

$$\frac{\vdash S_1 : \Sigma \to \Sigma \qquad \vdash S_2 : \Sigma \to \Sigma}{\vdash \mathtt{if}[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2 : \Sigma \to \Sigma} \textsc{Cond}$$

Types, effects, and annotations

annot. type system

$$\vdash S : \Sigma_1 \to \Sigma_2 \quad (13)$$

effect system

$$\vdash S : \Sigma \xrightarrow{\varphi} \Sigma \quad (14)$$

type and effect system (TES)

- *effect* system + *annotated* type system
- borderline fuzzy
- annotated type system
  - Σ_i: property of state ("$\Sigma_i \subseteq \Sigma$")
  - "abstract" properties: invariants, a variable is positive, etc.
- effect system
  - "statement S maps state to state, with (potential ...) effect φ"
  - *effect* φ: e.g.: errors, exceptions, file/resource access, ...

Annotated type systems

- example again: *reaching definitions* for while-language
- 2 flavors
  1. annotated base types: $S : \mathsf{RD}_1 \rightarrow \mathsf{RD}_2$
  2. annotated type constructors: $S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$

RD with annotated base types

judgement

$$\vdash S : \mathsf{RD}_1 \to \mathsf{RD}_2 \tag{15}$$

- $\mathsf{RD} \subseteq 2^{\mathbf{Var} \times \mathbf{Lab}}$
- auxiliary functions
  - note: every S has one "initial" elementary block, potentially more than one "at the end"
  - $init(S)$: the (unique) label at the entry of S
  - $final(S)$: the set of labels at the exits of S

"meaning" of judgment $\vdash S : \mathsf{RD}_1 \to \mathsf{RD}_2$

"RD_1 is the set of var/label reaching the entry of S and RD_2 the corresponding set at the exit(s) of S":

$$\begin{array}{rcl} \mathsf{RD}_1 & = & \mathsf{RD}_{entry}(init(S)) \\ \mathsf{RD}_2 & = & \bigcup\{\mathsf{RD}_{exit}(l) \mid l \in final(S)\} \end{array}$$

$$\vdash [x := a]^{l'} : \mathsf{RD} \to \mathsf{RD} \setminus \{(x, l) \mid l \in \mathbf{Lab}\} \cup \{(x, l')\} \qquad \text{ASS}$$

$$\vdash [\mathsf{skip}]^{l} : \mathsf{RD} \to \mathsf{RD} \qquad \text{SKIP}$$

$$\frac{\vdash S_1 : \mathsf{RD}_1 \to \mathsf{RD}_2 \qquad \vdash S_2 : \mathsf{RD}_2 \to \mathsf{RD}_3}{\vdash S_1; S_2 : \mathsf{RD}_1 \to \mathsf{RD}_3} \ \textsc{Seq}$$

$$\frac{\vdash S_1 : \mathsf{RD}_1 \to \mathsf{RD}_2 \qquad \vdash S_2 : \mathsf{RD}_1 \to \mathsf{RD}_2}{\vdash \mathtt{if}[b]^{l}\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2 : \mathsf{RD}_1 \to \mathsf{RD}_2} \ \textsc{If}$$

$$\frac{\vdash S : \mathsf{RD} \to \mathsf{RD}}{\vdash \mathtt{while}[b]^{l}\,\mathtt{do}\,S : \mathsf{RD} \to \mathsf{RD}} \ \textsc{While}$$

$$\frac{\vdash S : \mathsf{RD}_1' \to \mathsf{RD}_2' \qquad \mathsf{RD}_1 \subseteq \mathsf{RD}_1' \qquad \mathsf{RD}_2' \subseteq \mathsf{RD}_2}{\vdash S : \mathsf{RD}_1 \to \mathsf{RD}_2} \ \textsc{Sub}$$

Meaning of annotated judgments

"Meaning" of judgment $S : \mathsf{RD}_1 \rightarrow \mathsf{RD}_2$:

"RD_1 is *the* set of var/label reaching the entry of S and RD_2 the corresponding set at the exit(s) of S":

$$\begin{array}{rcl} \mathsf{RD}_1 & = & \mathsf{RD}_{entry}(init(S)) \\ \mathsf{RD}_2 & = & \bigcup\{\mathsf{RD}_{exit} l \mid l \in final(S)\} \end{array}$$

- Be careful:

$$\texttt{if}\,[b]^l\,\texttt{then}\,S_1\,\texttt{else}\,S_2$$

- more concretely

$$\texttt{if}\,[b]^l\,\texttt{then}\ [x := y]^{l_1}\,\texttt{else}\ [y := x]^{l_2}$$

Meaning of annotated judgments

Once again: "Meaning" of judgment $S : \mathsf{RD}_1 \rightarrow \mathsf{RD}_2$:

"if RD_1 is a set of var/label reaching the entry of S, then RD_2 is a corresponding set at the exit(s) of S":

$$\begin{array}{rrcl} & \text{if} \quad \mathsf{RD}_1 & \subseteq & \mathsf{RD}_{entry}(init(S)) \\ \text{then} & \forall l \in \mathit{final}(S).\ \mathsf{RD}_{exit}(l) & \subseteq & \mathsf{RD}_2 \end{array}$$

Derivation

$$\dfrac{[y := x]^0 : \mathsf{RD}_0 \to \{?_x, 0, ?_z\} \qquad \dfrac{[z := 1]^1 : \{?_x, 0, ?_z\} \to \{?_x, 0, 1\} \quad f_3 : \{?_x, 0, 1\} \to \mathsf{RD}_{\mathit{final}}}{f_2 : \{?_x, 0, ?_z\} \to \mathsf{RD}_{\mathit{final}}}}{f : \mathsf{RD}_0 \to \mathsf{RD}_{\mathit{final}}}$$

$$\mathsf{RD}_0 = \{?_x, ?_y, ?_z\} \quad \mathsf{RD}_{\mathit{final}} = \{?_x, 5, 1, 3\}$$

type sub-derivation for the rest $f_3 = \texttt{while} \ldots; [y := 0]^5$
loop invariant

$$\mathsf{RD}_{\mathit{body}} = \{?_x, 0, 4, 1, 3\}$$

Derivation

$$[z := \_]^3 : \mathsf{RD}_{body} \to \{?_x, 0, 4, 3, \not{1}\}$$

$$[y := \_]^4 : \{?_x, 0, 4, 3\} \to \{?_x, 4, 3\}$$

$$f_{body} : \mathsf{RD}_{body} \to \{?_x, 4, 3\}$$

--- Sub

$$f_{body} : \mathsf{RD}_{body} \to \mathsf{RD}_{body}$$

$$f_{while} : \mathsf{RD}_{body} \to \mathsf{RD}_{body}$$

--- Sub

$$f_{while} : \{?_x, 0, 1\} \to \mathsf{RD}_{body} \qquad [y := 0]^5 : \mathsf{RD}_{body} \to \mathsf{RD}_{fi}$$

$$f_3 : \{?_x, 0, 1\} \to \mathsf{RD}_{final}$$

Annotated type constructors

- alternative approach of annotated type systems
- arrow constructor itself *annotated*
- annotion of $\rightarrow$: flavor of effect system
- judgment

$$S : \Sigma \underset{\mathsf{RD}}{\longrightarrow} \Sigma$$

- annotation with RD (corresponding to the post-condition from above) alone is *not enough*

Annotated type constructors

- alternative approach of annotated type systems
- arrow constructor itself *annotated*
- annotion of $\rightarrow$: flavor of effect system
- judgment

$$S : \Sigma \xrightarrow[\text{RD}]{X} \Sigma$$

- annotation with RD (corresponding to the post-condition from above) alone is *not enough*
- also needed: the *variables "being" changed*

Intended meaning

"S maps states to states, where RD is the set of reaching definitions, S may produce and X the set of var's S must (= unavoidably) assign.

$$[x := a]^l : \Sigma \xrightarrow[\{(x,l)\}]{\{x\}} \Sigma \qquad \textsc{Ass} \qquad\qquad [\mathsf{skip}]^l : \Sigma \xrightarrow[\emptyset]{\emptyset} \Sigma \qquad \textsc{Skip}$$

$$\frac{S_1 : \Sigma \xrightarrow[\mathsf{RD}_1]{X_1} \Sigma \qquad S_2 : \Sigma \xrightarrow[\mathsf{RD}_2]{X_2} \Sigma}{S_1; S_2 : \Sigma \xrightarrow[\mathsf{RD}_1 \setminus X_2 \cup \mathsf{RD}_2]{X_1 \cup X_2} \Sigma} \textsc{Seq}$$

$$\frac{S_1 : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma \qquad S_2 : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma}{\mathtt{if}\,[b]^l\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2 : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma} \textsc{If}$$

$$\frac{S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma}{\mathtt{while}[b]^l\,\mathtt{do}\,S : \Sigma \xrightarrow[\mathsf{RD}]{\emptyset} \Sigma} \textsc{While}$$

$$\frac{S : \Sigma \xrightarrow[\mathsf{RD}']{X'} \Sigma \qquad X \subseteq X' \qquad \mathsf{RD}' \subseteq \mathsf{RD}}{S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma} \textsc{Sub}$$

Effect systems

- this time: back to the *functional* language
- starting point: simple type system
- *judgment*:

$$\Gamma \vdash e : \tau$$

- Γ: *type environment* (or context), "mapping" from variable to types
- types: bool, int, and $\tau \rightarrow \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_{\pi} x \Rightarrow e : \tau_1 \to \tau_2}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

Effects: Call tracking analysis

Call tracking analysis:

Determine: for each subexpression: which function abstractions may be applied, i.e., called, during the subexpression's evaluation.

$\Rightarrow$ set of function names

annotate: function type with latent effect

$\Rightarrow$ *annotated* types: $\hat{\tau}$: base types as before, arrow types:

$$\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \tag{16}$$

- functions from τ_1 to τ_2, where in the execution, functions from set φ are called.

Judgment

$$\hat{\Gamma} \vdash e : \hat{\tau} :: \varphi \tag{17}$$

$$\frac{\hat{\Gamma}(x) = \hat{\tau}}{\hat{\Gamma} \vdash x : \hat{\tau} :: \emptyset}\ \textsc{Var}$$

$$\frac{\Gamma, x{:}\hat{\tau}_1 \vdash e : \hat{\tau}_2 :: \ \varphi}{\Gamma \vdash \mathtt{fn}_\pi x \Rightarrow e : \hat{\tau}_1 \overset{\varphi \cup \{\pi\}}{\to} \hat{\tau}_2 :: \emptyset}\ \textsc{Abs}$$

$$\frac{\hat{\Gamma} \vdash e_1 : \hat{\tau}_1 \overset{\varphi}{\to} \hat{\tau}_2 :: \varphi_1 \qquad \hat{\Gamma} \vdash e_2 : \hat{\tau}_1 :: \varphi_2}{\hat{\Gamma} \vdash e_1\ e_2 : \hat{\tau}_2 :: \varphi \cup \varphi_1 \cup \varphi_2}\ \textsc{App}$$

Call tracking: example

$$
\dfrac{\dfrac{x{:}\mathsf{int} \xrightarrow{\{Y\}} \mathsf{int} \vdash x{:}\mathsf{int} \xrightarrow{\{Y\}} \mathsf{int} :: \emptyset}{\vdash (\mathtt{fn}_X\, x \Rightarrow x) : (\mathsf{int} \xrightarrow{\{Y\}} \mathsf{int}) \xrightarrow{\{X\}} (\mathsf{int} \xrightarrow{\{Y\}} \mathsf{int}) :: \emptyset} \qquad \vdash (\mathtt{fn}_Y\, y \Rightarrow y) : \mathsf{int} \xrightarrow{\{Y\}} \mathsf{int} :: \emptyset}{\vdash (\mathtt{fn}_X\, x \Rightarrow x)\ (\mathtt{fn}_Y\, y \Rightarrow y) : \mathsf{int} \xrightarrow{\{Y\}} \mathsf{int} :: \{X\}}
$$

Section

Algorithms

Chapter 1 “Introduction”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Chaotic iteration

- back to data flow/reaching def's
- goal: solve

$$\vec{RD} = F(\mathsf{RD}) \quad \text{or} \quad \vec{RD} \sqsubseteq F(\vec{\mathsf{RD}})$$

- F: monotone, finite domain

straightforward approach

init $\vec{\mathsf{RD}}_0 = F^0(\emptyset)$

iterate $\vec{\mathsf{RD}}_{n+1} = F(\vec{\mathsf{RD}}_n) = F^{n+1}(\emptyset)$ until stabilization

- approach to implement that: chaotic iteration
- non-deterministic stategy
- abbreviate:

$$\vec{\mathsf{RD}} = (\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$$

Chaotic iteration (for RD)

```
Input:     equations for reaching defs
           for the given program
Output:    least solution: RD⃗ = (RD_1, ..., RD_12)
```

Input: equations for reaching defs for the given program

Output: least solution: $\vec{\mathsf{RD}} = (\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$

Initialization:

$\mathsf{RD}_1 := \emptyset; \ldots ; \mathsf{RD}_{12} := \emptyset$

Iteration:

while $\mathsf{RD}_j \neq F_j(\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$ for some j
do

$\mathsf{RD}_j := F_j(\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$

Chapter 2

Data flow analysis

Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Chapter 2

Learning Targets of Chapter “Data flow analysis”.

various DFAs

monotone frameworks

operational semantics

foundations

special topics (SSA, context-sensitive analysis ...)

Chapter 2

Outline of Chapter "Data flow analysis".

Section

Intraprocedural analysis

Chapter 2 “Data flow analysis”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

While language and control flow graph

- starting point: while language from the intro
- *labelled* syntax (unique labels)
- labels = *nodes* of the cfg
- initial and final labels
- edges of a cfg: given by function flow

3 functions (definition see script / book)

1. $\mathit{init} : \mathbf{Stmt} \to \mathbf{Lab}$
2. $\mathit{final} : \mathbf{Stmt} \to 2^{\mathbf{Lab}}$
3. $\mathit{flow} : \mathbf{Stmt} \to 2^{\mathbf{Lab} \times \mathbf{Lab}}$

Flow and reverse flow

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

- data flow analysis can be *forward* (like RD) or backward
- $flow$: for forward analyses
- for backward analyses: *reverse* flow $flow^R$, simply invert the edges

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

$$[x := a + b]^0; [y := a * b]^1; \quad \texttt{while} \quad [y > a + b]^2$$
$$\texttt{do} \quad ([a := a + 1]^3; [x := a + b]^4)$$

- usage: avoid *re-computation*

Avoid recomputation: Available expressions

$$[x := a + b]^0; [y := a * b]^1; \quad \texttt{while} \quad [y > a + b]^2$$
$$\texttt{do} \quad ([a := a + 1]^3; [x := a + b]^4)$$

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 $\subseteq$-constraints, but not too crucial, as we know already)
- uniform representation of *effect of basic blocks* (= *intra-block flow*)

2 ingredients of intra-block flow

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

- later analyses: presented similarly
- different analyses $\Rightarrow$ different kind of flow information + different kill- and generate-functions

Available expressions: types

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

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

- analysis: pair of functions

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

Intra-block flow specification: Kill and generate

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

Intra-block flow specification: Kill and generate

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

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

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

split into

nodes: intra-block equations, using *kill* and *generate*

edges: inter-block equations, using *flow*

Flow equations for AE

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

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

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

- note the *"order"* of kill and generate

Available expressions

- *forward* analysis (as RD)
- interest in *largest* solution (unlike RD)

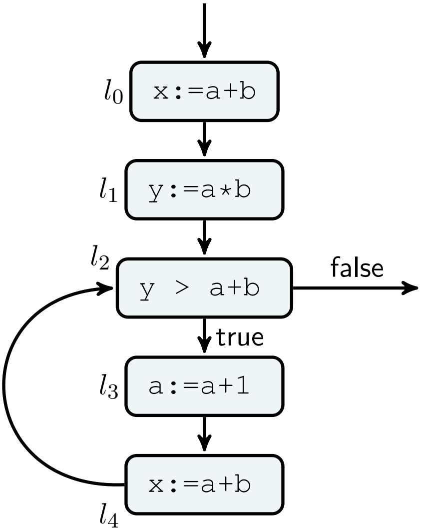
⇒ must analysis (as opposed to *may*)

- expression is available: if *no path kills it*
- remember: informal description of AE: expression available on *all paths* (i.e., not killed on any)
- illustration

Example AE

$$
\begin{array}{lll}
[x := a+b]^0; [y := a*b]^1; & \texttt{while} & [y > a+b]^2 \\
 & \texttt{do} & ([a := a+1]^3; [x := a+b]^4)
\end{array}
$$

Example AE



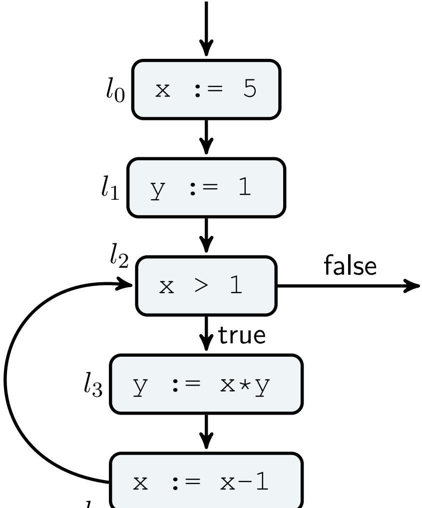
Reaching definitions

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

$$[x := 5]^0; [y := 1]^1; \mathtt{while}[x > 1]^2\,\mathtt{do}([y := x*y]^3; [x := x-1]^4)$$

Reaching definitions

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



Reaching definitions: types

- interest in *sets of tuples of var's and program points i.e., labels*:

$$2^{\mathbf{Var}_* \times \mathbf{Lab}_*^?} \quad \text{where} \quad \mathbf{Lab}_*^? = \mathbf{Lab}_* + \{?\}$$

- generation and killing:

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

- analysis: pair of mappings

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

Reaching defs: kill and generate

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

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

Reaching defs: kill and generate

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

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

Flow equations: $\mathsf{RD}^{=}$

split into

- *intra*-block equations, using *kill* and *generate*
- *inter*-block equations, using *flow*

Flow equations for RD

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

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

- same order of kill/generate

Flow equations: $\mathsf{RD}^{=}$

split into

- *intra*-block equations, using *kill* and *generate*
- *inter*-block equations, using *flow*

Flow equations for RD

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

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

- same order of kill/generate

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 expression 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).

- usage: expression "hoisting"

Goal

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

Very busy expressions: types

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

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

- analysis: pair of mappings

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

Very busy expr.: kill and generate

core of the intra-block flow specification

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

Very busy expr.: kill and generate

core of the intra-block flow specification

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

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

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

split into

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

however: everything works backwards now

Flow equations: VB

$$\mathsf{VB}_{exit}(l) =$$

$$\mathsf{VB}_{entry}(l) =$$

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

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

split into

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

however: everything works backwards now

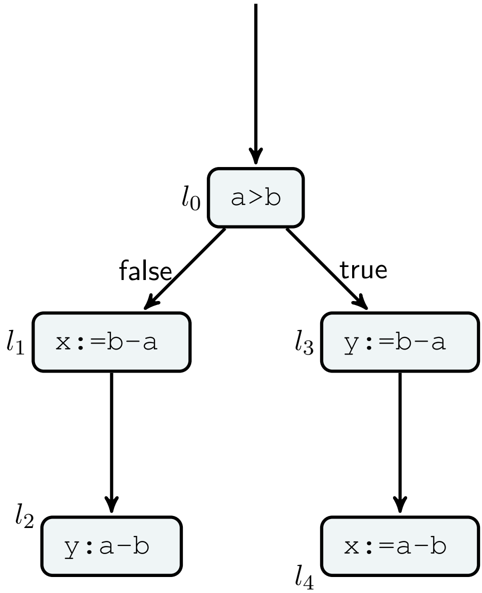
Flow equations: VB

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

$$\mathsf{VB}_{entry}(l) = \mathsf{VB}_{exit}(l) \setminus \mathit{kill}_{\mathsf{VB}}(B^l) \cup \mathit{gen}_{\mathsf{VB}}(B^l)$$

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

Example



When can var's be "recycled": Live variable analysis

$$[x := 2]^0; [y := 4]^1; [x := 1]^2;$$
$$(\texttt{if}[y > x]^3 \,\texttt{then}\, [z := y]^4 \,\texttt{else}\, [z := y * y]^5); [x := z]^6$$

Goal therefore

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

- usage: *register allocation*

When can var's be "recycled": Live variable analysis

$$[x := 2]^0; [y := 4]^1; [x := 1]^2;$$
$$(\texttt{if}[y > x]^3 \, \texttt{then} \, [z := y]^4 \, \texttt{else} \, [z := y * y]^5); [x := z]^6$$

Live variable

A variable is live (at the exit of a label) if 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)

Goal therefore

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

- usage: *register allocation*

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

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

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

$$\mathsf{LV}_{exit}(l) \quad = $$

$$\mathsf{LV}_{entry}(l) \quad = $$

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

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

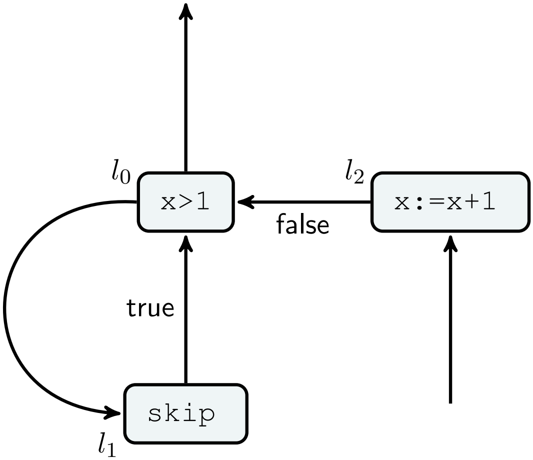
$$\mathsf{LV}_{entry}(l) = \mathsf{LV}_{exit}(l) \setminus \mathit{kill}_{\mathsf{LV}}(B^l) \cup \mathit{gen}_{\mathsf{LV}}(B^l)$$

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

Example

$$(\texttt{while}\ [x > 1]^{l_0}\ \texttt{do}\ [\mathsf{skip}]^{l_1}); [x := x + 1]^{l_2}$$

Looping example



Section

Theoretical properties and semantics

Chapter 2 “Data flow analysis”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Relating programs with analyses

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

States, configs, and transitions

fixing some data types

- *state* $\sigma : \mathbf{State} = \mathbf{Var} \rightarrow \mathbf{Z}$
- *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

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

simplifying assumption: no errors

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

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

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

$$\langle [x := a]^l, \sigma \rangle \rightarrow \sigma[x \mapsto [\![a]\!]^{\mathcal{A}}_{\sigma}] \qquad \text{ASS} \qquad \langle [\mathsf{skip}]^l, \sigma \rangle \rightarrow \sigma \qquad \text{SKIP}$$

$$\frac{\langle S_1, \sigma \rangle \rightarrow \langle \acute{S}_1, \acute{\sigma} \rangle}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle \acute{S}_1; S_2, \acute{\sigma} \rangle} \text{SEQ}_1 \qquad \frac{\langle S_1, \sigma \rangle \rightarrow \acute{\sigma}}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \acute{\sigma} \rangle} \text{SEQ}_2$$

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

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

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

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
- CFG for the "rest" of the program only gets "smaller" when running:

Lemma

1. $\langle S, \sigma \rangle \rightarrow \sigma'$, *then* $\mathit{final}(S) = \{\mathit{init}(S)\}$
2. *Assume* $\langle S, \sigma \rangle \rightarrow \langle \acute{S}, \acute{\sigma} \rangle$, *then*
   - 2.1 $\mathit{final}(S) \supseteq \{\mathit{final}(\acute{S})\}$
   - 2.2 $\mathit{flow}(S) \supseteq \{\mathit{flow}(\acute{S})\}$
   - 2.3 $\mathit{blocks}(S) \supseteq \mathit{blocks}(\acute{S})$; *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

$$
\begin{array}{rcll}
\mathsf{LV}_{exit}(l) & \supseteq & \begin{cases} \emptyset & l \in \mathit{final}(S_*) \\ \bigcup\{\mathsf{LV}_{entry}(l') \mid (l',l) \in \mathit{flow}^R(S_*)\} & \text{otherwise} \end{cases} \\
\mathsf{LV}_{entry}(l) & \supseteq & \mathsf{LV}_{exit}(l) \setminus \mathit{kill}_{\mathsf{LV}}(B^l) \cup \mathit{gen}_{\mathsf{LV}}(B^l)
\end{array}
$$

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

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

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

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

Equational vs. constraint analysis

Lemma

1. *If* $live \models \mathsf{LV}^{=}$*, then* $live \models \mathsf{LV}^{\subseteq}$
2. *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 $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top)$: a partially ordered set where meets and joins exist for *all subsets*, furthermore $\top = \bigsqcap \emptyset$ and $\bot = \bigsqcup \emptyset$.

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: $f(l) \sqsupseteq l$:

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

Define "lfp" / "gfp"

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

Tarski's theorem

Core

Perhaps core insight of the whole lattice/fixpoint business: not only does the $\bigsqcap$ of all pre-fixpoints uniquely exist (that's what the lattice is for), but —and that's the trick— *it's a pre-fixpoint* itself (ultimately due to montonicity of f).

Theorem

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

$$\begin{aligned} lfp(f) &\triangleq \bigsqcap Red(f) \in Fix(f) \\ gfp(f) &\triangleq \bigsqcup Ext(f) \in Fix(f) \end{aligned} \tag{18}$$

- Note: lfp (despite the name) is *defined* as glb of all pre-fixpoints
- The theorem (more or less directly) implies lfp is the *least* fixpoint

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)

$$\bot \sqsubseteq f^n(\bot) \sqsubseteq \bigsqcup_n f^n(\bot) \quad \sqsubseteq \quad lfp(f)$$
$$gfp(f) \quad \sqsubseteq \bigsqcap_n f^n(\top) \sqsubseteq f^n(\top) \sqsubseteq (\top)$$

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

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

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

Basic preservation results

Lemma ("Smaller" graph → less constraints)

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

Corollary ("subject reduction")

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

Lemma (Flow)

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

$\Rightarrow$

Correctness relation on states:

Given V = set of variables:

$$\sigma_1 \sim_V \sigma_2 \text{ iff } \forall x \in V.\sigma_1(x) = \sigma_2(x) \tag{19}$$

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

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

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

- *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* $\acute{\sigma}_1 \sim_{X(init(S))} \acute{\sigma}_2$.

$$
\begin{array}{ccc}
\langle S, \sigma_1 \rangle & \longrightarrow & \langle \acute{S}, \acute{\sigma}_1 \rangle \\
\Big| \sim_{N(init(S))} & & \vdots \sim_{N(init(\acute{S}))} \\
\langle S, \sigma_2 \rangle & \cdots\cdots\rightarrow & \langle \acute{S}, \acute{\sigma}_2 \rangle
\end{array}
\qquad
\begin{array}{ccc}
\langle S, \sigma_1 \rangle & \longrightarrow & \acute{\sigma}_1 \\
\Big| \sim_{N(init(S))} & & \vdots \sim_{X(init(S))} \\
\langle S, \sigma_2 \rangle & \cdots\cdots\rightarrow & \acute{\sigma}_2
\end{array}
$$

Correctness (many steps)

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

- 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 $l \in final(S)$.

Section

Monotone frameworks

Chapter 2 "Data flow analysis"
Course "Static analysis and all that"
Martin Steffen
IN5440 / autum 2018

Monotone framework: general pattern

$$
\begin{array}{rcll}
Analysis_\circ(l) & = & \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{Analysis_\bullet(l') \mid (l',l) \in F\} & \text{otherwise} \end{cases} \\
Analysis_\bullet(l) & = & f_l(Analysis_\circ(l))
\end{array}
\tag{20}
$$

- $\bigsqcup$: 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

$$
\begin{aligned}
Analysis_\circ(l) &= \iota^l_E \sqcup \bigsqcup\{Analysis_\bullet(l') \mid (l', l) \in F\} \\
&\quad \text{where } \iota^l_E = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases} \\
Analysis_\bullet(l) &= f_l(Analysis_\circ(l))
\end{aligned}
\tag{21}
$$

where $l \sqcup \bot = l$

Basic definitions: property space

- *property space* L, often *complete lattice*
- *combination* operator: $\bigsqcup : 2^L \to L$, $\sqcup$: 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*
- requirement: *monotone*
- $\mathcal{F}$: monotone functions over L:
  - containing all *transfer functions*
  - containing *identity*
  - *closed under composition*

Summary

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

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

The 4 classical examples

- for a label consistent program S_*, all are *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
  - *distributivity*: assured by using the kill- and generate-formulation

Overview over the 4 examples

	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 $[B]^l \in blocks(S_*)$			

Section

Equation solving

Chapter 2 “Data flow analysis”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

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
  - MFP: "maximal fix point"
  - 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/set) of pairs
- related to *chaotic iteration*

Chaotic iteration

```
Input:      equations for reaching defs
            for the given program
Output:     least solution: RD⃗ = (RD_1, ..., RD_12)
```

$$\text{Output: least solution: } \vec{\mathsf{RD}} = (\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$$

Initialization:

$$\mathsf{RD}_1 := \emptyset; \ldots; \mathsf{RD}_{12} := \emptyset$$

Iteration:

while $\mathsf{RD}_j \neq F_j(\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$ for some j
do

$$\mathsf{RD}_j := F_j(\mathsf{RD}_1, \ldots, \mathsf{RD}_{12})$$

Worklist algorithms

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

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

- for more details (different traversal strategies): see Chap. 6 from [**?**]

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])
```

ML Code

```
let rec solve (wl1 : edge list) : unit =
  match wl1 with
  |    [] -> ()                                    (* wl done *)
  |    (l,l')::wl' ->
       let ana_pre    : var list = lookx (ana,l) (* extract ``states *)
       and  ana_post : var list = lookx (ana,l')
       in let ana_exitpre : var list = f_trans(ana_pre,l)
       in
       if not (subset (ana_exitpre,ana_post))
       then
         (enter (ana,l',union(ana_post,ana_exitpre));
          let (new_edges : edge list) =
            (let (preds : node list) = Flow.Graph.pred (l')
            in List.map (fun n -> (l',n)) preds)
          in solve (new_edges @ wl')
         )
       else                                  (* Nothing to do here. *)
         (solve (wl'))
  in
  solve wl_init;
  fun (x: node) -> lookx (ana, x)
;;
```

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

$\Rightarrow$

$$O(e \cdot h) \tag{22}$$

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

Section

Interpreocedural analysis

Chapter 2 “Data flow analysis”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

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
- complications:
  - calls/return (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

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

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

- 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
- mutual recursion possible
- assumption: unique labelling, only declared procedures are called, all procedures have different names.

Example: Fibonacci

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

Block, labels, etc.

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

Block, labels, etc.

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

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

- two *new* kinds of flows (written slightly different(!)): *calling* and *returning*
- *static* dispatch only

For procedure declaration

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

For procedure declaration

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

“Standard” flow of complete program

not yet interprocedural flow (IF)

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

side remark: S_*: notation for complete program “of interest”

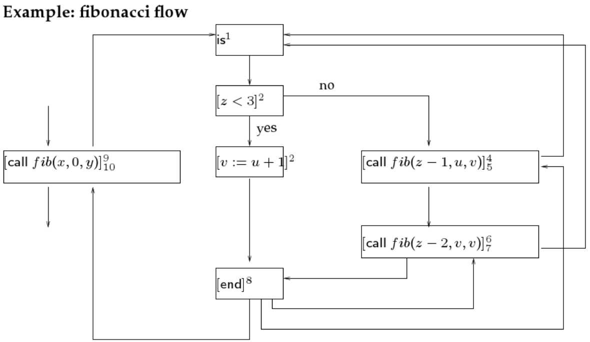
New kind of edges: Interprocedural flow (IF)

- inter-procedural: from call-site to procedure, and back: $(l_c; l_n)$ and $(l_x; l_r)$.
- more *precise* ($=$ better) capture of flow
- abbreviation: IF for $inter\text{-}flow_*$ or $inter\text{-}flow_*^R$

IF

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

Example: fibonacci flow



Semantics: stores, locations,...

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

Representation of "memory"

ξ	$\in$	$\mathbf{Loc}$	locations
ρ	$\in$	$\mathbf{Env} = \mathbf{Var}_* \to \mathbf{Loc}$	environment
ς	$\in$	$\mathbf{Store} = \mathbf{Loc} \to_{fin} \mathbf{Z}$	store

- $\sigma = \varsigma \circ \rho$: total $\Rightarrow ran(\rho) \subseteq dom(\varsigma)$
- top-level environment: ρ_*: all var's are mapped to unique locations (no aliasing !!!!)

SOS steps

- steps *relative* to *environment* ρ

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

or

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

- old rules needs to be adapted
- "global" environment ρ_* (for global vars)

Call-rule

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

Call-rule

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

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)[2]

[2]Not covered in the lecture.

Bind-construct

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

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

- bind-syntax: "runtime syntax"

⇒ formulation of correctness must be adapted, too (Chap. 3)[2]

[2]Not covered in the lecture.

Transfer function: 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* $(L, \mathcal{F}, F, E, \iota, f)$

Naive

- treat IF edges $(l_c; l_n)$ and $(l_x; l_r)$ as ordinary flow edges (l_1, l_2)
- *ignore* parameter passing: *transfer* functions for proc. calls and proc definitions are *identity*

Equation system (“naive” version”)

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

with

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

- analysis: *safe*
- unnecessarily imprecise, too abstract

Paths

- remember: "MFP"
- historically: MOP stands for meet over all paths
- here: dually mosty *joins*
- 2 "versions" of a path:
  - path to entry of a block: blocks traversed from the "extremal block" of the program, but not including it
  - path to exit of a block

Paths

$$
\begin{aligned}
path_{\circ}(l) &= \{[l_1, \ldots \mathbf{l_{n-1}}] \mid l_i \rightarrow_{flow} l_{i+1} \wedge l_n = l \wedge l_1 \in E\} \\
path_{\bullet}(l) &= \{[l_1, \ldots \mathbf{l_n}] \mid l_i \rightarrow_{flow} l_{i+1} \wedge l_n = l \wedge l_1 \in E\}
\end{aligned}
$$

- transfer function for paths $\vec{l}$

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

Meet over all paths

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

MOP

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

MOP vs. MFP

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

Lemma

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet \tag{23}$$

In case of a distributive framework

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet \tag{24}$$

MVP

- take calls and returns (IF) serious
- restrict attention to valid ("possible") paths

⇒ capture the nesting structure

- from MOP to MVP: "meet over all valid paths"
- *complete* path:
  - appropriate call-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) (we assume forward analysis here)
- basically a context-free grammar

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

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

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

Example: Fibonacci

- concrete 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 paths (context-free grammar)

Valid path (generated from non-terminal VP_*):

- start at extremal node (E),
- all proc *exits* have matching *entries*

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

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

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

MVP

- adapt the definition of paths

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

- MVP solution:

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

- but still: “meets over paths” is *impractical*

Fixpoint calculations

next: how to reconcile the path approach with MFP

Contexts

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

⇒ instead of MVP: "embellish" MFP

$$\delta \in \Delta \tag{25}$$

- δ: context information
- for instance: representing/recording of the *path* taken

⇒ "embellishment": adding contexts

embellished monotone framework

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

- intra-procedural (no change of embellishment Δ)
- inter-procedural

Intra-procedural: basically unchanged

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

$$\hat{f}_l(\hat{l})(\delta) = f_l(\hat{l}(\delta)) \tag{26}$$

- flow equations: "unchanged" for intra-proc. part

$$\begin{aligned} A_\bullet(l) &= \hat{f}_l(A_\circ(l)) \\ A_\circ(l) &= \bigsqcup\{A_\bullet(l') \mid (l', l) \in F \text{ or } (l'; l) \in F)\} \sqcup \iota_E^{\hat{l}} \end{aligned} \tag{27}$$

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

Sign analysis (unembellished)

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

Transfer function for $[x := a]^l$

$$f_l^{sign}(Y) = \bigcup\{\phi_l^{sign}(\sigma^{sign}) \mid \sigma^{sign} \in Y\} \tag{28}$$

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

$$\phi_l^{sign}(\sigma^{sign}) = \{\sigma^{sign}[x \mapsto s] \mid s \in [\![a]\!]_{\sigma^{sign}}^{\mathcal{A}_{sign}}\} \tag{29}$$

Sign analysis: embellished

$$\hat{L}_{sign} = \Delta \to L_{sign} \tag{30}$$
$$= \Delta \to 2^{\mathbf{Var}_* \to \mathbf{Sign}} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}$$

Transfer function for $[x := a]^l$

$$\hat{f}_l^{sign}(Z) = \bigcup \{\{\delta\} \times \phi_l^{sign}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z\} \tag{31}$$

Inter-procedural

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

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

- 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 $(l_c, l_n, l_x, l_r) \in IF$

2 transfer functions

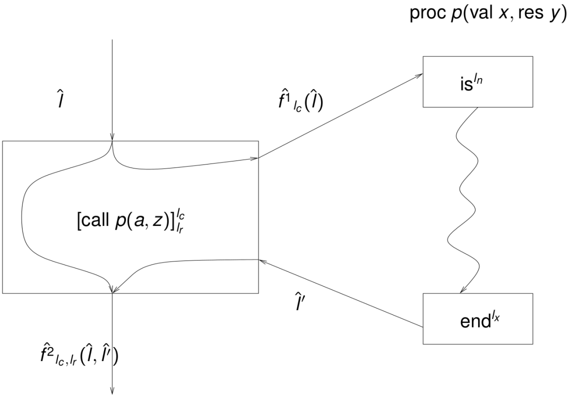
1. for calls: $\hat{f}^1{}_{l_c} : (\Delta \to L) \to (\Delta \to L)$

$$A_\bullet(l_c) = \hat{f}^1{}_{l_c}(A_\circ(l_c)) \tag{32}$$

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

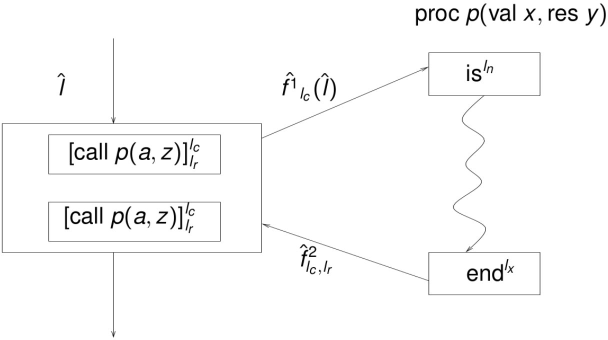
$$A_\bullet(l_r) = \hat{f}^2{}_{l_c,l_r}(A_\circ(l_c), A_\circ(l_r))) \tag{33}$$

Procedure call



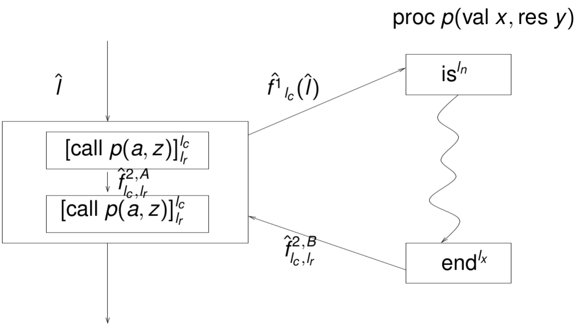
Ignoring the call context

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



Merging call contexts

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



Context sensitivity

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

⇒ context *sensitive* analysis ⇒ more precision + more effort

In the following: 2 *specializations*:

1. control (“call strings”)
2. data

(combinations of course possible)

Call strings

- context = *path*
- call-string = sequence of currently "active" calls
- concentrating on calls: flow-edges (l_c, l_n), where just l_c is recorded

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

- *extremal* value (from $\hat{L} = \Delta \to L$)

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

Call strings

- context = *path*
- call-string = sequence of currently "active" calls
- concentrating on calls: flow-edges (l_c, l_n), where just l_c is recorded

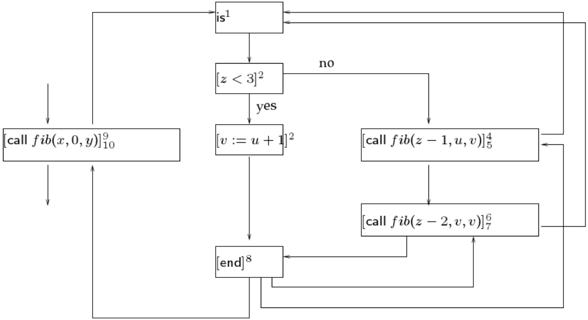
$$\Delta = \mathbf{Lab}^* \qquad \text{call strings}$$

- *extremal* value (from $\hat{L} = \Delta \to L$)

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

Fibonacci flow

Example: fibonacci flow



Fibonacci call strings

some call strings:

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

similar, but not same as valid paths

Transfer functions for call strings

- here: forward analysis
- 2 cases: define $\hat{f}^1_{l_c}$ and $\hat{f}^2_{l_c,l_r}$

Transfer functions

- calls (basically: check that the path ends with l_c):

$$\begin{aligned} \hat{f}^1_{l_c}(\hat{l})([\delta, l_c]) &= f^1_{l_c}(\hat{l}(\delta)) \\ \hat{f}^1_{l_c}(\_) &= \bot \end{aligned} \quad (34)$$

- returns (basically: match return with (a same-level) call)

$$\hat{f}^2_{l_c,l_r}(\hat{l}, \hat{l}')(\delta) = f^2_{l_c,l_r}(\hat{l}(\delta), \hat{l}'([\delta, l_c])) \quad (35)$$

- rather "higher-order" way of connecting the flows, using the call-strings as contexts
- *connection* between the arguments (via δ) of f_{l_c,l_r}
- given: underlying $f^1_{l_c}$ and $f^2_{l_c,l_r}$.
- Notation $[\delta, l_c]$ ...

Sign analysis (continued)

- so far: “unconcrete”, i.e.,
- given some underlying analysis: how to make it context-sensitive
- call-strings as context
- now: apply to some simple case: signs
- remember: $\hat{L} \simeq 2^{\Delta \times (\mathbf{Var}_* \to \mathbf{Sign})}$ (see Eq. (30))
- before: standard embellished $\hat{f}_l^{\mathbf{Sign}}$ (with the help of $\phi_l^{\mathbf{Sign}}$)
- now: *inter-procedural*

Sign analysis: aux. functions ϕ

still unembellished

calls: abstract parameter-passing

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

returns (analogously)

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

(formal params: x, y, where y is the *result parameter*, actual parameter z)

- non-det "assignment" to y
- remember: operational semantics,

Sign analysis: aux. functions ϕ

still unembellished

calls: abstract parameter-passing

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

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, where y is the *result parameter*, actual parameter z)

- non-det "assignment" to y
- remember: operational semantics,

Sign analysis

calls: abstract parameter-passing + glueing calls-returns

$$\hat{f}^{sign1}_{l_c}(Z) \quad = \quad \bigcup\{\{\delta'\} \times \phi^{sign1}_{l_c}(\sigma^{sign}) \mid (\delta', \sigma^{sign}) \in Z, \delta' = \quad)\}$$

Returns: analogously

$$\hat{f}^{sign2}_{l_c,l_r}(Z, Z') \quad = \quad \bigcup\{\{\delta\} \times \phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) \mid \quad (\delta, \sigma^{sign}_1) \in Z \quad \}$$

(formal params: x, y, actual parameter z)

Sign analysis

calls: abstract parameter-passing + glueing calls-returns

$$\hat{f}^{sign1}_{l_c}(Z) \;=\; \bigcup\{\{\delta'\} \times \phi^{sign1}_{l_c}(\sigma^{sign}) \mid (\delta', \sigma^{sign}) \in Z, \delta' = [\delta, l_c])\}$$

Returns: analogously

$$\hat{f}^{sign2}_{l_c,l_r}(Z, Z') \;=\; \bigcup\{\{\delta\} \times \phi^{sign2}_{l_c,l_r}(\sigma^{sign}_1, \sigma^{sign}_2) \mid \begin{array}{l} (\delta, \sigma^{sign}_1) \in Z \\ (\delta', \sigma^{sign}_2) \in Z' \\ \delta' = [\delta, l_c] \end{array} \}$$

(formal params: x, y, actual parameter 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: lattice

$L = 2^D$

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

restrict to only the last call

dependency on data only $\Rightarrow$

(large) assumption set context

$$\Delta = 2^D$$

- $\hat{\iota} = \{(\{\iota\}, \iota)\}$ extremal value

Transfer functions

- calls

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

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

- note: new context δ' for the procedure body
- "caller-callee" connection via the context (= data) δ
- 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

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

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

- note: new context δ' for the procedure body
- "caller-callee" connection via the context (= data) δ
- return

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

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

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

- return

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

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

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

global variables that may be assigned to (also indirectly) when p is called

- two aux. definitions (straightforwardly defined, obviously flow-insensitive)
  - $\mathsf{AV}(S)$: assigned variables in S
  - $\mathsf{CP}(S)$: called procedures in S

$$\mathsf{IAV}(p) = (\mathsf{AV}(S) \setminus \{x\}) \cup \bigcup\{\mathsf{IAV}(p') \mid p' \in CP(S)\} \quad (36)$$

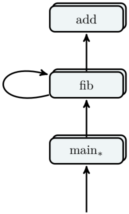
where $\mathtt{proc}\, p(\mathtt{val}\, x, \mathtt{res}\, y)\, \mathtt{is}^{l_n}\, S\, \mathtt{end}^{l_x} \in D_*$

- CP ⇒ procedure call graph (which procedure calls which one; see example)

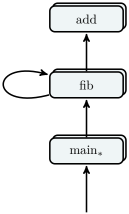
Example

```
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



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

$\Rightarrow$ smallest solution

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

Chapter 3

Types and effect systems

Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Chapter 3

Learning Targets of Chapter "Types and effect systems".

type systems

effects

functional languages

type inference and unification

Chapter 3

Outline of Chapter “Types and effect systems”.

Type checking

Type inference

Section

Type checking

Chapter 3 “Types and effect systems”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Introduction

- now: working with a
  - *typed* language
  - functional language Fun
- cf. the corresponding intro-section (annotated types)
- here: control-flow analysis (perhaps more). Remember also the constraint based analysis/CFA in the intro

Syntax

$$e \;::=\; c \mid x \mid \mathtt{fn}_\pi x \Rightarrow e \mid \mathtt{fun}_\pi f\ x \Rightarrow e \mid e\ e \qquad \text{terms}$$
$$\mid\; \mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e \mid \mathtt{let}\, x = e\, \mathtt{in}\, e \mid e\, \mathtt{op}\, e$$

Table: Abstract syntax

π	$\in$	**Pnt**	program points
e	$\in$	**Expr**	expressions
c	$\in$	**Const**	constants
op	$\in$	**Op**	operators
f, x	$\in$	**Var**	variables

Examples

Example (Application)

$$(\mathtt{fn}_X\, x \Rightarrow x)\ (\mathtt{fn}_Y\, y \Rightarrow y)$$

Example

$$\begin{array}{ll} \mathtt{let}\, g & = (\mathtt{fun}_F\, f\ x \Rightarrow f(\mathtt{fn}_Y\, y \Rightarrow y)) \\ \mathtt{in} & g\ (\mathtt{fn}_Z\, x \Rightarrow x) \end{array}$$

Types

- *Curry*-style typing

$$\begin{array}{llll} \tau & \in & \mathbf{Type} & \text{types} \\ \Gamma & \in & \mathbf{TEnv} & \text{type environment} \end{array}$$

Types

$$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \tau \to \tau$$

- base types:
  - bool and int
  - standard constants and operators assumed ($\mathsf{true}, 5, +, \leq, \ldots$)
  - each constant has a base type τ_c
- type environments (finite mappings)

$$\Gamma ::= [] \mid \Gamma, x{:}\tau$$

Judgments and derivation system

Type judgments

$$\Gamma \vdash_{\mathsf{UL}} e : \tau \tag{37}$$

- derivation system:
  - Curry-style formulation
  - $\Rightarrow$ *non-deterministic*
  - nonetheless: *monomorphic* let
- type reconstruction/type inference

$$\Gamma \vdash c : \tau_c \quad \textsc{Con} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \textsc{Var}$$

$$\frac{\Gamma \vdash e_1 : \tau^1_{\mathsf{op}} \qquad \Gamma \vdash e_2 : \tau^2_{\mathsf{op}}}{\Gamma \vdash e_1 \,\mathsf{op}\, e_2 : \tau_{\mathsf{op}}} \textsc{Op}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \tau_1 \rightarrow \tau_2} \textsc{Fn} \qquad \frac{\Gamma, x{:}\tau_1, f{:}\tau_1 \rightarrow \tau_2 \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fun}_\pi x \Rightarrow e : \tau_1 \rightarrow \tau_2} \textsc{Fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2} \textsc{App}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \ \mathtt{if}\, e_0 \,\mathtt{then}\, e_1 \,\mathtt{else}\, e_2 : \tau} \textsc{If}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \ \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 : \tau_2} \textsc{Let}$$

Section

Type inference

Chapter 3 “Types and effect systems”
Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

Inference algorithms

- take care of *terminology*
- so far: no *algorithm*! (price of laxness)
- *foresight* needed
- guessing wrong ⇒ backtracking (and we seriously don't want that)

⇒ required: mechanism to make
  - tentative *guesses*
  - *refine* guesses

- we start first: with the *underlying system*

Augmented types

fancy name for: "we have added type variables"

$$\begin{array}{lll} \tau & \in & \mathbf{AType} \quad \text{augmented types} \\ \alpha & \in & \mathbf{TVar} \quad \text{type variables} \end{array}$$

$$\begin{array}{lll} \tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \tau \to \tau \mid \alpha \\ \alpha & ::= & '\mathtt{a} \mid '\mathtt{b} \mid \ldots \end{array}$$

Substitutions

Substitution (in general)

mapping from variables to "terms"

- "syntactic mapping" here:
  - "terms" are (augmented) types
  - variables: type variables

$$\theta : \mathbf{TVar} \rightarrow_{\mathit{fin}} \mathbf{AType}$$

- considered as finite functions: we write $dom(\theta)$.
- ground substitution: mapping to *ordinary* types (no variables)
- substitutions: *lifted* to types in the standard manner
- composition of substitutions: $\theta_1 \circ \theta_2$ (or just $\theta_2\theta_1$)

Algorithm: basic idea

- instead of guessing type *now* ⇒ *postpone* the decision

⇒ use of type variables
replace:

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \tau_1 \to \tau_2}\ \textsc{Fn}$$

by

Algorithm: basic idea

- instead of guessing type *now* $\Rightarrow$ *postpone* the decision

$\Rightarrow$ use of type variables

$$\frac{\Gamma, x{:}\alpha \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \alpha \to \tau_2} \ \textsc{Fn}$$

Algorithm: basic idea

- instead of guessing type *now* $\Rightarrow$ *postpone* the decision

$\Rightarrow$ use of type variables

$$\frac{\Gamma, x{:}\alpha \vdash e : \tau_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \alpha \rightarrow \tau_2} \ \text{FN}$$

- $x{:}\alpha$ when α is fresh (otherwise unused) means: type of x is completely arbitrary.
- syntax-directed now?
- τ_1: meta-variable for concrete types
- α: (still meta variable for) type variables

Algorithm: basic idea

- instead of guessing type *now* $\Rightarrow$ *postpone* the decision

$\Rightarrow$ use of type variables
α's completely arbitrary?
Consider body

$$e = x\ g$$

for $\mathtt{fn}_\pi x \Rightarrow e$
$\Rightarrow$

Restriction on α here

- a function type: $\alpha = \beta \to \gamma$
- fit together with type of g $\Rightarrow$ condition or constraint on β

Algorithm: basic idea

- instead of guessing type *now* $\Rightarrow$ *postpone* the decision

$\Rightarrow$ use of type variables

  - judments "give back" not just the type, but also "restrictions" on type variables.
  - represented as constraint[3]
  - $\Rightarrow$

$$\Gamma \vdash e : (\tau, C)$$

Under the assumptions Γ (which might "assign" to (program) variables: type variables), program e possesses type τ (again potentially containing type variables) *and* imposes the restrictions "embodied" by C on the type variables.

[3] In the book, what is given back is a substitution instead.

Constraints

- generally:
  - constraint(s) is a formula with free variables
  - solving a constraint set: finding values for the variables such that here formula becomes true (satisfiability)

  . set of constraints = interpreted as $\wedge$ (conjuction)
- more precisely here: (term) unification constraints
- notation $\tau_1 =^? \tau_2$
- many other forms of "constraints" systems exists with specialized solving techniques
- here: term *unification*

Constraint generation

$$\frac{}{\Gamma \vdash c : (\tau_c, \emptyset)}\ \textsc{T-Const} \qquad \frac{}{\Gamma \vdash x : (\Gamma(x), id)}\ \textsc{T-Var}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x{:}\alpha \vdash e_0 : (\tau_0, C_0)}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e_0 : (\alpha \to \tau_0, C_0)}\ \textsc{T-Fn}$$

$$\frac{\alpha, \alpha_0 \text{ fresh} \quad \Gamma, f{:}\alpha \to \alpha_0, x{:}\alpha \vdash e_0 : (\tau_0, C_0) \quad C_1 = \{\tau_0 =^? \alpha\}}{\Gamma \vdash \mathtt{fun}_\pi f\ x \Rightarrow e_0 : (\alpha \to \tau_0, C_0, C_1)}\ \textsc{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, C_1) \quad \Gamma \vdash e_2 : (\tau_2, C_2) \quad \alpha \text{ fresh} \quad C_3 = \{\tau_1 =^? (\tau_2 \to \alpha)\}}{\Gamma \vdash e_1\ e_2 : (\alpha, C_1, C_2, C_3)}\ \textsc{T-App}$$

Constraint generation

$$\frac{\Gamma \vdash e_0 : (\tau_0, C_0) \qquad \Gamma \vdash e_1 : (\tau_1, C_1) \qquad \Gamma \vdash e_2 : (\tau_2, C_2) \qquad C_4 = \tau_0 =^? \textsf{bool} \qquad C_5 = \tau_1 =^? \tau_2}{\Gamma \vdash \texttt{if}\, e_0 \,\texttt{then}\, e_1 \,\texttt{else}\, e_2 : (\tau_2, C_1, C_2, C_3, C_4, C_5)} \textsc{If}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, C_1) \qquad \Gamma, x{:}\tau_1 \vdash e_2 : (\tau_2, C_2)}{\Gamma \vdash \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 : (\tau_2, C_1, C_1)} \textsc{Let}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, C_1) \qquad \Gamma \vdash e_2 : (\tau_2, C_2) \qquad C = \{\tau_1 =^? \tau^1_{\texttt{op}}, \tau_2 =^? \tau^2_{\texttt{op}}\}}{\Gamma \vdash e_1 \,\texttt{op}\, e_2 : (\tau_{\texttt{op}}, C_1, C_2, C)} \textsc{Op}$$

Unification

- "classical" algorithm ([1])
- many applications (theorem proving, Prolog etc.)
- definition: substitution

Unifier

A unifier of two types τ_1 and τ_2: a *substitution* θ such that

$$\theta(\tau_1) = \theta(\tau_2)$$

- *unfication problem* given τ_1 and τ_2, determine a *unifier* for them, if it exists

Ordering substitution (and unifiers)

- better formulation of unfication problem: given τ_1 and τ_2, determine *the best* $=$ *most general unifier* for them (if they are unifiable).
- solve unification constraint $\tau_1 =^? \tau_2$
- easy generalizable to constraints: $\theta \models C$

Ordering: "less general", "more specific"

$\theta_1 \lesssim \theta_2$ if $\theta_1 = \theta\theta_2 \quad$ (for some θ)

- most-general-unifier of two types $=$ "the" least upper bound of all unifiers

Unification algorithm for underlying types

$$\begin{array}{rcl}
\mathcal{U}(\mathsf{int}, \mathsf{int})) & = & id \\
\mathcal{U}(\mathsf{bool}, \mathsf{bool})) & = & id \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') & = & \begin{array}{ll} \texttt{let} & \theta_1 = \mathcal{U}(\tau_1, \tau_1') \\ & \theta_2 = \mathcal{U}(\theta_1\tau_2, \theta_1\tau_2') \\ \texttt{in} & \theta_2 \circ \theta_1 \end{array} \\
\mathcal{U}(\tau, \alpha) & = & \begin{cases} [\alpha \mapsto \tau] & \text{if } \alpha \text{ does not occur in } \tau \\ & \text{or if } \alpha = \tau \\ \mathsf{fail} & \text{else} \end{cases} \\
\mathcal{U}(\alpha, \tau) & = & \text{symmetrically} \\
\mathcal{U}(\tau_1, \tau_2) & = & \mathsf{fail} \qquad \text{in all other cases}
\end{array}$$

1-phase Type inference algorithm

- formulated here as *rule system*
- immediate correspondence to a *recursive* function:

$$\mathcal{W}(\Gamma, e) = (\tau, \theta)$$

instead of

$$\Gamma \vdash e : (\tau, \theta)$$

- not 2-phase, giving back a set of unification constraints C

$$\frac{}{\Gamma \vdash c : (\tau_c, id)}\ \textsc{T-Const} \qquad \frac{}{\Gamma \vdash x : (\Gamma(x), id)}\ \textsc{T-Var}$$

$$\frac{\alpha \textsf{ fresh} \qquad \Gamma, x{:}\alpha \vdash e_0 : (\tau_0, \theta_0)}{\Gamma \vdash \ \texttt{fn}_\pi x \Rightarrow e_0 : (\theta_0\alpha \rightarrow \tau_0, \theta_0)}\ \textsc{T-Fn}$$

$$\frac{\alpha, \alpha_0 \textsf{ fresh} \quad \Gamma, f{:}\alpha \rightarrow \alpha_0, x{:}\alpha \vdash e_0 : (\tau_0, \theta_0) \quad \theta_1 = \mathcal{U}(\tau_0, \theta_0\alpha_0)}{\Gamma \vdash \texttt{fun}_\pi f\ x \Rightarrow e_0 : (\theta_1\theta_0\alpha \rightarrow \theta_1(\tau_0), \theta_1 \circ \theta_0)}\ \textsc{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \theta_1) \quad \theta_1\Gamma \vdash e_2 : (\tau_2, \theta_2) \quad \alpha \textsf{ fresh} \quad \theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \rightarrow \alpha)}{\Gamma \vdash e_1\ e_2 : (\theta_3\alpha, \theta_3\theta_2\theta_1)}\ \textsc{T-App}$$

$$\frac{\Gamma \vdash e_0 : (\tau_0, \theta_0) \qquad \theta_0\Gamma \vdash e_1 : (\tau_1, \theta_1) \qquad \theta_1\theta_0\Gamma \vdash e_2 : (\tau_2, \theta_2) \\ \theta_3 = \mathcal{U}(\theta_2\theta_0\tau_0, \mathsf{bool}) \qquad \theta_4 = \mathcal{U}(\theta_3\tau_2, \theta_3\theta_2\tau_1)}{\Gamma \vdash \texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : (\theta_4\theta_3\tau_2, \theta_4\theta_3\theta_2\theta_1\theta_0)} \text{ IF}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \theta_1) \qquad \theta_1\Gamma, x{:}\tau_1 \vdash e_2 : (\tau_2, \theta_2)}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : (\tau_2, \theta_2\theta_1)} \text{ LET}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \theta_1) \qquad \theta_2\Gamma \vdash e_2 : (\tau_2, \theta_2) \\ \theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_{\texttt{op}}^1) \qquad \theta_3 = \mathcal{U}(\theta_3\tau_2, \tau_{\texttt{op}}^2)}{\Gamma \vdash e_1\, \texttt{op}\, e_2 : (\tau_{\texttt{op}}, \theta_4\theta_3\theta_2\theta_1)} \text{ OP}$$

"Classic" type inference

- we did not look at the *full* well-known Hindley-Damas-Milner type inference algorithm
- missing here: polymorphic let
- monomoprhic let: "almost useless" polymorphism
- Note the fine line
  - polymorphic let: yes
  - polymorphic functions as function arguments: no!

the classical type "inference" algo

- higher-order functions,
- polymorphic functions,
- but *no "higher-order polymorphic functions"*

- dropping the last restriction: type inference *undecidable*
- no type variables in the underlying type system (the "specification"), the type inference algo does
- types (with variables) and *type schemes* $\forall\alpha.\tau$

Chapter 4

References

Course “Static analysis and all that”
Martin Steffen
IN5440 / autum 2018

References I

Bibliography

[1] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41.