Mechanics of Static Analysis

David Schmidt

Kansas State University

www.cis.ksu.edu/~schmidt

Outline

1. Small-step semantics: trace generation
2. State generation and collecting semantics
3. Data-flow analysis
4. Ensuring termination
5. Typing rules and big-step semantics
6. Interprocedural analysis

Static analysis

A *static analysis* of a program is a *sound, finite, and approximate* calculation of the program's execution semantics.

Approximate: not exact — computes properties or aspects of the execution semantics, such as pre- or post-conditions, invariants, data types, patterns of trace, or ranges-of-values.

Sound: consistent with the concrete, execution semantics — a sound *overapproximation* describes a superset of the program's executions (**safe descriptions**); a sound *underapproximation* describes a subset of the program's executions (**live descriptions**). We will focus on overapproximations.

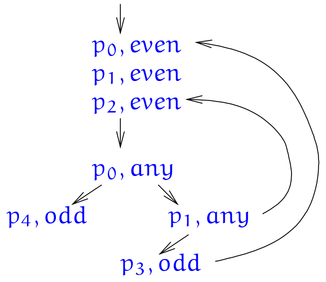
Finite: regardless of the program and its approximate semantics, the analysis terminates.

The most basic static analysis is *trace generation*

```
p0 : while (x != 1) {
  p1 : if Even(x)
    p2 : then   x = x div2;
    p3 : else   x = 3*x + 1;
  }
p4 : exit
```

Note: p_i, v abbreviates $p_i, \langle x : v \rangle$

Abstract overapproximating trace:



Two concrete traces:

$p_0, 4$	$p_0, 6$
$p_1, 4$	$p_1, 6$
$p_2, 4$	$p_2, 6$
$p_0, 2$	$p_0, 3$
$p_1, 2$	$p_1, 3$
$p_2, 2$	$p_2, 3$
$p_0, 1$	$p_0, 10$
$p_4, 1$	$\cdots$
	$p_4, 1$

The abstract tree *(abstract model)* is a static analysis of those concrete executions that use an even-valued input.

Each concrete transition, $p_i, s \rightarrow p_j, f_i(s)$, is reproduced by a corresponding abstract transition, $p_i, a \rightarrow p_j, f_i^\#(a)$, where $s \in \gamma(a)$. ($f_i^\# = \alpha \circ f_i \circ \gamma$.)

The traces embedded in the abstract trace tree simulate all the concrete traces, e.g., this concrete trace,

$$p_0, 4 \rightarrow p_1, 4 \rightarrow p_2, 4 \rightarrow p_0, 2 \rightarrow p_1, 2 \rightarrow p_2, 2 \rightarrow p_0, 1 \rightarrow p_4, 1$$

is simulated by this abstract trace, which is extracted from the abstract computation tree:

$$p_0, \mathit{even} \rightarrow p_1, \mathit{even} \rightarrow p_2, \mathit{even} \rightarrow p_0, \mathit{even} \rightarrow p_1, \mathit{even} \rightarrow p_2, \mathit{even} \rightarrow p_0, \mathit{odd} \rightarrow p_4, \mathit{odd}$$

because we used a Galois connection to justify the soundness of the transition steps in the abstract trace tree.

In this fashion, a static analysis can generate an *abstract test or abstract model*, which covers a range of concrete inputs.

State reachability and collecting semantics

If we are interested *only in the reachable states and not their orderings in the trace*, we compute the program's *collecting semantics* as a nondecreasing sequence of sets of program states. State-reachability semantics is an *abstraction* of trace-generation semantics.

State-reachability semantics, concrete and abstract:

$\{p_0, 4\}$	$\{p_0, \mathit{even}\}$
$\{p_0, 4;\ p_1, 4\}$	$\{p_0, \mathit{even};\ p_4, \mathit{even};\ p_1, \mathit{even}\}$
$\{p_0, 4;\ p_1, 4;\ p_2, 4\}$	$\{p_0, \mathit{even};\ p_4, \mathit{even};\ p_1, \mathit{even};\ p_2, \mathit{even}\}$
$\{p_0, 4;\ p_1, 4;\ p_2, 4;\ p_0, 2\}$	$\{p_0, \mathit{even};\ p_4, \mathit{even};\ p_1, \mathit{even};\ p_2, \mathit{even};\ p_0, \mathit{any}\}$
...	...
$\{p_0, 4;\ p_1, 4;\ p_2, 4;\ p_0, 2;\ p_1, 2;\ p_2, 2;\ p_0, 1;\ p_4, 1\}$	$\{p_0, \mathit{even};\ p_4, \mathit{even};\ p_1, \mathit{even};\ p_2, \mathit{even};\ p_0, \mathit{any};\ p_4, \mathit{any};\ p_1, \mathit{any};\ p_3, \mathit{odd}\}$

"Sticky" collecting semantics

A semantics of form, $\wp(\mathit{ProgramPoint} \times \mathit{AbsStore})$, is "attaching" $\mathit{AbsStore}$ values to each program point — the isomorphic representation, $\mathit{ProgramPoint} \rightarrow \wp(\mathit{AbsStore})$, is called the *(relational) "sticky" collecting semantics*:

$$[p_0 \mapsto \{\mathit{even}, \mathit{any}\};\ p_1 \mapsto \{\mathit{even}, \mathit{any}\};\ p_2 \mapsto \{\mathit{even}\};\ p_3 \mapsto \{\mathit{odd}\};\ p_4 \mapsto \{\mathit{even}, \mathit{any}\}]$$

The above can be abstracted to a function in $\mathit{ProgramPoint} \rightarrow \mathit{AbsStore}$, the *independent-attribute* semantics:

$$[p_0 \mapsto \mathit{any};\ p_1 \mapsto \mathit{any};\ p_2 \mapsto \mathit{even};\ p_3 \mapsto \mathit{odd};\ p_4 \mapsto \mathit{any}]$$

which is based on this abstraction mapping:

$$\alpha : \wp(\mathit{AbsStore}) \rightarrow \mathit{AbsStore}$$
$$\alpha(S) = \langle i : \textstyle\bigsqcup_{s \in S} s(i) \rangle_{i \in \mathit{Identifier}}$$

Notice that the independent-attribute semantics is less precise than its relational ancestor; for example, variables `x` and `y` might have these values at program point p_i:

$$[...p_i \mapsto \{\langle x : \mathit{even},\ y : \mathit{even}\rangle,\ \langle x : \mathit{odd},\ y : \mathit{odd}\rangle\}...]$$

meaning that `x + y` computes to even at p_i.

But the independent-attribute abstraction,

$$[...p_i \mapsto \langle x : \mathit{any},\ y : \mathit{any}\rangle...]$$

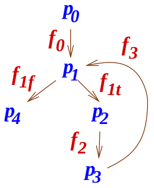
makes `x + y` compute to any, losing precision.

Note also that we could define a collecting version of a trace-generation semantics, which generates an analysis of form $\mathit{ProgramPoint} \rightarrow \wp(\mathit{Trace})$.

Formalizing the "small steps": *transfer functions*

A trace's transitions, $pp_i, s \longrightarrow pp_i, s'$, are computed with a *control-flow graph* annotated with *transfer functions*.

```
p0 :  y = 1;
p1 :  while Even(x) {
        p2 : y = y * x;
        p3 : x = x div2;
      }
p4 :  exit
```



Concrete transfer functions: $\langle u, v\rangle$ abbreviates $\langle x : u, y : v\rangle$

$f_0\langle u, v\rangle = \langle u, 1\rangle$

$f_{1t}(s) = \begin{cases} s & \text{if } s = \langle 2u, v\rangle \\ \perp & \text{otherwise} \end{cases}$ $\quad f_{1f}(s) = \begin{cases} s & \text{if } s = \langle 2u+1, v\rangle \\ \perp & \text{otherwise} \end{cases}$

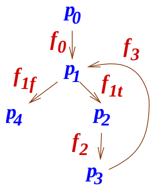
$f_2\langle u, v\rangle = \langle u, v * u\rangle$

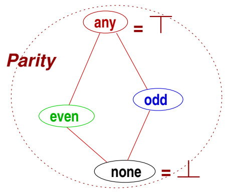
$f_3\langle u, v\rangle = \langle u/2, v\rangle$

Important: *configurations of form, $p_i, \perp$, cannot appear in a trace.*

The *abstract transfer functions* are derived as $f^\# = \alpha \circ f \circ \gamma$

```
p0 :  y = 1;
p1 :  while Even(x) {
        p2 : y = y * x;
        p3 : x = x div2;
      }
p4 :  exit
```





As usual, $\langle u, v\rangle$ abbreviates $\langle x : u, y : v\rangle$

Note: all $f^\#$ are *totally strict:* $f^\#\langle u, \perp\rangle = f^\#\langle \perp, v\rangle = \langle \perp, \perp\rangle$

$f_0^\#\langle u, v\rangle = \langle u, odd\rangle$

$f_{1t}^\# s = s \sqcap \langle even, \top\rangle \qquad f_{1f}^\# s = s \sqcap \langle odd, \top\rangle$

$f_2^\#\langle u, v\rangle = \langle u, w\rangle$, where $w = \begin{cases} even & \text{if } u = even \text{ or } v = even, \text{ else} \\ odd & \text{if } u = odd \text{ and } v = odd, \text{ else} \\ \top & \end{cases}$

$f_3^\#\langle u, v\rangle = \langle \top, v\rangle$

Note: $\langle a, b\rangle \sqcap \langle a', b'\rangle = \langle a \sqcap a', b \sqcap b'\rangle$.

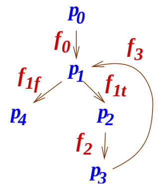
Flow equations calculate the (sticky, collecting) independent-attribute semantics

The value "attached" to program point p_i is defined by the equational pattern,

$$p_i\mathrm{Store} = \bigsqcup_{p_j \in pred(p_i)} f_j^\#(p_j\mathrm{Store})$$

The collecting semantics of p_i is the join of the answers computed by p_i's predecessor transfer functions.

Flow equations for previous example:



$p_0\mathrm{Store} = \langle x : \top, y : \top\rangle$

$p_1\mathrm{Store} = f_0^\#(p_0\mathrm{Store}) \sqcup f_3^\#(p_3\mathrm{Store})$

$p_2\mathrm{Store} = f_{1t}^\#(p_1\mathrm{Store})$

$p_3\mathrm{Store} = f_2^\#(p_2\mathrm{Store})$

$p_4\mathrm{Store} = f_{1f}^\#(p_1\mathrm{Store})$

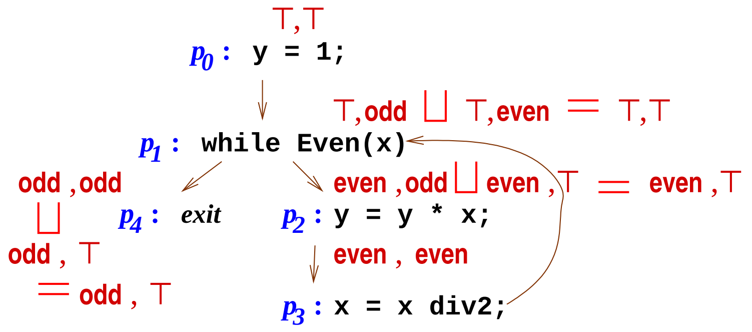
We *solve* the flow equations by calculating approximate solutions in stages until *the least fixed point* is reached.

Note: u, v abbreviates $\langle x : u, y : v\rangle$.

stage	$p_0\mathrm{Store}$	$p_1\mathrm{Store}$	$p_2\mathrm{Store}$	$p_3\mathrm{Store}$	$p_4\mathrm{Store}$
0	$\perp, \perp$	$\perp, \perp$	$\perp, \perp$	$\perp, \perp$	$\perp, \perp$
1	$\top, \top$	$\perp, \perp$	$\perp, \perp$	$\perp, \perp$	$\perp, \perp$
2	$\top, \top$	$\top, odd$	$\perp, \perp$	$\perp, \perp$	$\perp, \perp$
3	$\top, \top$	$\top, odd$	$even, odd$	$\perp, \perp$	odd, odd
4	$\top, \top$	$\top, odd$	$even, odd$	$even, even$	odd, odd
...					
8	$\top, \top$	$\top, \top$	$even, \top$	$even, even$	$odd, \top$
9	$\top, \top$	$\top, \top$	$even, \top$	$even, even$	$odd, \top$

A faster algorithm uses a *worklist* that remembers exactly which equations should be recalculated at each stage.

To summarize, we annotate the control-flow graph with the non-$\perp$ values that arrive at the program points:



The analysis approximates the stores that arrive at the program points.

The equational format is called *data-flow analysis*. It is the most popular static analysis format.

Variants of data-flow analysis

We might vary whether the "data flow" goes forwards or backwards; we might also vary whether information is "joined" ($\sqcup$) or "met" ($\sqcap$):

Forwards-possibly:

$p_i Store = \sqcup_{p_j \in pred(p_i)} f_j(p_j Store)$

Forwards-necessarily:

$p_i Store = \sqcap_{p_j \in pred(p_i)} f_j(p_j Store)$

Backwards-possibly:

$p_i Store = f_i^{-1}(\cup_{p_j \in succ(p_i)} p_j Store)$

Backwards-necessarily:

$p_i Store = f_i^{-1}(\cap_{p_j \in succ(p_i)} p_j Store)$

The backwards analyses almost always compute sets of values, hence the use of $\cup$ and $\cap$.

A *forwards* analysis computes "histories" that arrive at a point:

forwards analysis = postcondition semantics

$p_i Store = a$ approximates the set of traces of the form $p_0, s_0 \rightarrow p_1, s_1 \rightarrow \cdots \rightarrow p_i, s_i$ (where $s_i \in \gamma(a)$)

A *backwards* analysis computes the "futures" from a program point:

backwards analysis = precondition semantics

$p_i Store = a$ approximates the set of traces of the form $p_i, s_i \rightarrow \cdots \rightarrow p_{exit}, s_{final}$ (where $s_i \in \gamma(a)$)

A *possibly* analysis predicts a "superset" of the actual computations: if $p_i Store = a$, then *for all* concrete values, $c \sqsubseteq_C \gamma(a)$, that arrive at p_i, we have $c \sqsubseteq_C \gamma(a)$ — all possibilities are predicted.

A *necessarily* analysis predicts a "subset" of the actual computations: if $p_i Store = a$, then *there exists* some $c \sqsubseteq_C \gamma(a)$, that arrives at p_i.

The data-flow example developed earlier in this Lecture computed answers of the form,

$$p_i Store = a$$

which asserted, if store s arrives at program point p_i, then $s \in \gamma(a)$.

But there are data-flow analyses where $p_i Store = a$ means that all *execution traces* that arrive at p_i contain some *pattern* of program points and stores, described by a.

We will develop the Galois-connection formalities in the next Lecture, but just now we study two examples, used by compilers for improving register allocation in target code. *These examples compute sets of program phrases that describe patterns within execution traces.*

The examples show variations of the forwards/backwards and possibly/necessarily forms of data-flow analysis.

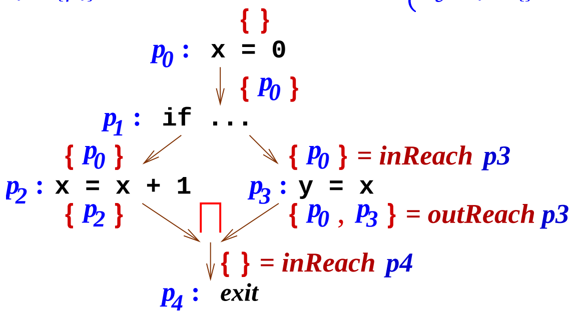
Forwards-necessarily-reaching definitions: which assignments *must* reach their successors

$$\mathrm{inReach}p_i = \bigcap_{p_j \in pred(p_i)} \mathrm{outReach}p_j$$
$$\mathrm{outReach}p_i = f_i^{\#}(\mathrm{inReach}p_i) = (\mathrm{inReach}p_i - \mathrm{kill}_i) \cup \mathrm{gen}_i$$

(the transfer function computes a set of assignment statements)

for $p_i : x = e$, $\begin{cases} \mathrm{kill}_i = \{p_j \mid p_j : x = ...\} \\ \mathrm{gen}_i = \{p_i\} \end{cases}$ for p_i : `if e`, $\begin{cases} \mathrm{kill}_i = \{\} \\ \mathrm{gen}_i = \{\} \end{cases}$

Sample analysis:



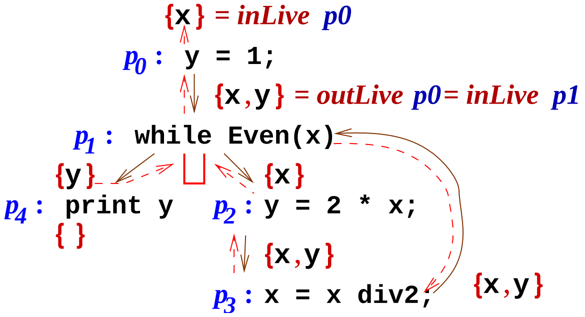
Backwards-possibly-live variables: which variables *might* be referenced in the future

$$\mathrm{outLive}p_i = \bigcup_{p_j \in succ(p_i)} \mathrm{inLive}p_j$$
$$\mathrm{inLive}p_i = f_i^{\#}(\mathrm{outLive}p_i) = (\mathrm{outLive}p_i - \mathrm{kill}_i) \cup \mathrm{gen}_i$$

(the transfer function computes a set of variable names)

for $p_i : x = e$ $\begin{cases} \mathrm{kill}_i = \{x\} \\ \mathrm{gen}_i = \{v \mid v \text{ in } e\} \end{cases}$ for `print e` / p_i : `while e` $\begin{cases} \mathrm{kill}_i = \{\} \\ \mathrm{gen}_i = \{v \mid v \text{ in } e\} \end{cases}$

Sample analysis:



Explanation:

If $p' \in \mathrm{inReach}p_i$, where p' labels the assignment, $p' : v = e$, then *all* traces from p_0 to p_i *must* possess the pattern,

$$p_0 \to \cdots \to p' \to \cdots \to p_i$$

and no assignment, $v = e'$, occurs between p' and p_i in the trace.

If $p' \in \mathrm{inReach}p_i$ holds, then the assignment at p' should save its right-hand-side value in a register for quick access by p_i.

Explanation:

If there is a concrete execution trace containing the pattern,

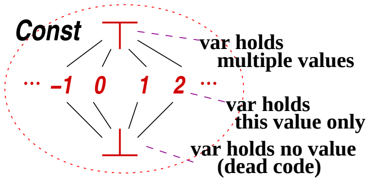
$$p_i \to \cdots \to p' \to \cdots \to p_{exit}$$

such that p' references variable v and no assignment to v appears between p_i and p', then $v \in \mathrm{outLive}p_i$.

If $v \notin \mathrm{outLive}p_i$ holds, then v's value should be removed from all registers upon completion of p_i's execution — v is a *"dead variable"* after p_i.

Termination: Constant propagation reviewed

```
p0 :  x = 1; y = 2;
p1 :  while (x < y + z)
        p2 :  x = x + 1;
      }
p3 :  exit
```



where $m + n$ is interpreted

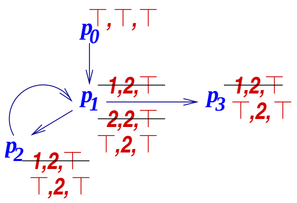
$$k_1 + k_2 \longrightarrow \mathrm{sum}(k_1, k_2),$$
$$\top \neq k_i \neq \perp, i \in 1..2$$
$$\top + k \longrightarrow \top$$
$$k + \top \longrightarrow \top$$

The naive trace does not terminate.

Abstract trace:

$p_0, \langle \top, \top, \top \rangle$
$p_1, \langle 1, 2, \top \rangle$ → $p_3, \langle 1, 2, \top \rangle$
$p_2, \langle 1, 2, \top \rangle$
$p_1, \langle 2, 2, \top \rangle$ → $p_3, \langle 2, 2, \top \rangle$
$p_2, \langle 2, 2, \top \rangle$
$p_1, \langle 3, 2, \top \rangle$ → ...
...

Finite-height and $\bigsqcup$ give termination



stage	p_0Store	p_1Store	p_2Store	p_3Store
1	$\top,\top,\top$	$\perp,\perp,\perp$	$\perp,\perp,\perp$	$\perp,\perp,\perp$
2	$\top,\top,\top$	$1,2,\top$	$\perp,\perp,\perp$	$\perp,\perp,\perp$
3	$\top,\top,\top$	$1,2,\top$	$1,2,\top$	$1,2,\top$
4	$\top,\top,\top$	$\top,2,\top$	$1,2,\top$	$1,2,\top$
5	$\top,\top,\top$	$\top,2,\top$	$\top,2,\top$	$\top,2,\top$
6	$\top,\top,\top$	$\top,2,\top$	$\top,2,\top$	$\top,2,\top$

Termination is *guaranteed* because the transfer functions and $\bigsqcup$ are monotonic (each stage has values not smaller than its predecessors) and the abstract domain, **Const**, has *finite height* — there are no infinitely ascending sequences (the stages cannot increase forever).

(Indeed, the longest seqence in **Const** goes: $\perp \sqsubseteq k \sqsubseteq \top$.)

Termination: Array-bounds checking reviewed

Integer variables receive values from the *interval domain*,

$$I = \{[i, j] \mid i, j \in Int \cup \{-\infty, +\infty\}\}.$$

We define $[a, b] \sqcup [a', b'] = [\min(a, a'), \max(b, b')]$.

```
int a = new int[10];           - - - i = [0,0]
i = 0;
while (i < 10)  {         p1   i = [0,0] ⊓ [-∞,9]  = [0,0]
    ... a[i] ...               i = [0,0] ⊔ [1,1] ⊓ [-∞,9] = [0,1]
                               ...
    i = i + 1;            p2   i = [1,1]
}                              i = [1,1] ⊔ [2,2] = [1,2]
                               ...
```

This example terminates: i's ranges are

at p_1 : $[0..9]$

at p_2 : $[1..10]$

at loop exit : $[1..10] \sqcap [10, +\infty] = [10, 10]$

But others might not, because the domain is not finite height:

```
i = 0;               - - - i = [0,0]
while true {               i = [0,0] ⊔ [1,1] ⊔ [2, 2] ...
   ...                     infinite limit is [ 0, + ∞ ]
   i = i + 1;
}
                     - - - i = [ ]   (dead code)
```

The analysis generates the infinite sequence of stages, $[0, 0], [0, 1], ..., [0, i], ...$ as i's value in the loop's body.

The domain of intervals, where $[i, j] \sqsubseteq [i', j']$ *iff* $i \leq j$ *and* $j \leq j'$*, has infinitely ascending chains.*

To forcefully terminate the analysis, we can replace the $\sqcup$ operation by ∇, called a *widening operator*:

$$[\,]\nabla[i, j] = [i, j] \qquad [i, j]\nabla[i', j'] = \begin{array}{l}[\text{if } i' < i \text{ then } -\infty \text{ else } i,\\ \text{if } j' > j \text{ then } +\infty \text{ else } j]\end{array}$$

The widening operator, which guarantees finite convergence for all increasing sequences on the interval domain, quickly terminates the example:

```
i = 0;         <---- i = [0,0]
while true {
   ...         <--- i = [0,0] ∇ [1,1] = [0,+∞]
  i = i + 1;
}
             <----------- i = [ ]  (dead code)
```

but in general, it can lose much precision:

```
int a = new int[10];
i = 0;              <-------- i = [0,0]
while (i < 10)  {
    ... a[i]        <--- i = [0,0] ∇ [1,1] = [0,+∞]
    i = i + 1;
}
              <------------- i = [10,+∞]
```


For this reason, a complementary operation, $\triangle$, called a *narrowing operation*, can be used after ∇ gives convergence to recover some precision and retain a fixed-point solution.

We will not develop $\triangle$ here, but for the interval domain, a suitable $\triangle$ tries to reduce $-\infty$ and $+\infty$ to finite values. For the last example, the convergent value, $[0, +\infty]$, in the loop body would be narrowed to $[0, 10]$, making i's value on loop exit $[10, 10]$.

Another approach is to use multiple "thresholds" for widening, e.g. $-\infty$, $(2^{-31} - 1)$, 0, etc. for lower limits, and $(2^{31} - 1)$ and $+\infty$ for upper limits.

Structured (*big-step*) static analysis

Given a block of statements, B, we might wish to calculate the values that "enter" and "exit" from B. If B is coded in a structured language, we can define the static analysis to compute a *structured transfer function* for B:

$$C \ ::= \ p : x \ = E \ | \ C_1; C_2 \ | \ \texttt{if}\ E\ C_1\ C_2 \ | \ \texttt{while}\ E\ C$$

A sample structured analysis that ignores tests: $[\![C]\!] : A_{in} \to A_{out}$

$[\![p : x \ = E]\!]in = f^{\#}_p(in)$ (the transfer function for p)

$[\![C_1; C_2]\!]in = [\![C_2]\!]([\![C_1]\!]in)$

$[\![\texttt{if}\ E\ C_1\ C_2]\!]in = [\![C_1]\!]in \ \sqcup \ [\![C_2]\!]in$

$[\![\texttt{while}\ E\ C]\!]in = in \sqcup out_C$,

where $out_C = \bigsqcup_{i \geq 0} out_i$,

and $out_0 = \perp_A$ and $out_{i+1} = [\![C]\!](in \sqcup out_i)$

We can annotate a syntax tree with the in-and out-data — here is a *forwards-possibly* reaching definitions analysis, which computes sets of assignments that might reach future program points:

$[\![p : x \ = E]\!]in = in - kill_x \cup \{p\}$

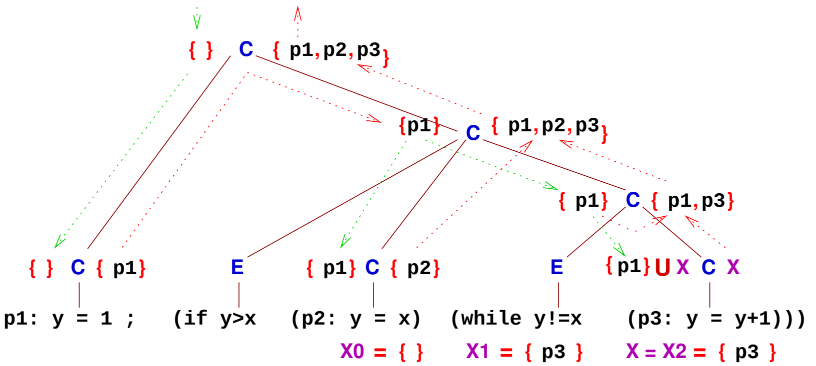
$[\![C_1; C_2]\!]in = [\![C_2]\!]([\![C_1]\!]in)$

$[\![\texttt{if}\ E\ C_1\ C_2]\!]in = [\![C_1]\!]in \ \cup \ [\![C_2]\!]in$

$[\![\texttt{while}\ E\ C]\!]in = in \cup \bigcup_{i \geq 0} out_i$,

where $out_0 = \{\}$

and $out_{i+1} = [\![C]\!](in \cup out_i)$



The analysis calculates a “local” least-fixed point at each while-loop, in contrast to a data-flow analysis, which calculates a single “global” least-fixed point for the entire program. (It is straightforward to prove that both techniques compute the same answer.)

The structured, equational style is based on *denotational semantics*.

The structured analysis is no more precise than the iterative, data-flow analysis (that is, a sticky, collecting semantics); indeed, $[\![C]\!] : A_{in} \to A_{out}$ is an abstraction of the data-flow analysis of C in the sense that $[\![C]\!]$ “forgets” the flow information of C’s subphrases and returns only C’s output.

Structured analysis in inference-rule format

The style of the previous example suggests that a structured analysis pairs each phrase, C, with its input and outputs, in_c and out_c.

We might write relational “assertions” in the formats

$$in_c \; C \; out_c \qquad \text{or} \qquad C : in_c \to out_c.$$

The first format is used in Hoare-logics, the second in data-typing.

The semantics equations inspire us to write these inference rules:

$$\vdash p : \mathtt{x} = E : in \to f_p^{\#}(in) \qquad \frac{\vdash C_1 : in \to out \quad \vdash C_2 : out \to out'}{\vdash C_1; C_2 : in \to out'}$$

$$\frac{\vdash C_1 : in \to out_1 \quad \vdash C_2 : in \to out_2}{\vdash \mathtt{if}\; E\; C_1\; C_2 : in \to out_1 \sqcup out_2} \qquad \frac{\vdash C : in \sqcup out \to out}{\vdash \mathtt{while}\; E\; C : in \to out}$$

We use the rules to derive a program’s analysis as a “proof.”

Reaching definitions, repeated:

$$in \; p : \mathtt{x} = E \; in - kill_x \cup \{p\} \qquad \frac{in \; C_1 \; out \quad out \; C_2 \; out'}{in \; C_1; C_2 \; out'}$$

$$\frac{in \; C_1 \; out_1 \quad in \; C_2 \; out_2}{in \; \mathtt{if}\; E\; C_1\; C_2 \; out_1 \cup out_2} \qquad \frac{in \cup out \; C \; out}{in \; \mathtt{while}\; E\; C \; out}$$

The (inverted) proof resembles the annotated syntax tree:

```
        { } p1: y = 1 ; if y>x ... { p1,p2,p3}
-----------------------------------------------------
{ } p1: y = 1 { p1}        { p1} if y>x ...{ p1,p2,p3}
                    -----------------------------------------------------
                    { p1} p2: y = x { p2}  { p1} while y!=x ... { p1,p3}
                                           ----------------------------
                                            { p1,p3} p3: y = y+1 { p3}
```

Unlike the denotational-semantics version, the `while`-rule does *not* calculate a least-fixed point: A “guess” or “inference” of an *invariant assertion* is made to obtain a proof of $in \cup out C out$ that is used to prove in `while` $E \; C out$. *The program analysis is done in one pass*.

The example suggests that data-typing systems defined as “inference-rule sets” are *one-pass, structured static analyses*:

$$\pi \vdash x : \pi(x) \qquad \frac{\pi \oplus [x \mapsto \tau_1] \vdash E : \tau_2}{\pi \vdash \lambda x.E : \tau_1 \to \tau_2} \qquad \frac{\pi \vdash E_1 : \tau_1 \to \tau_2 \quad \pi \vdash E_2 : \tau_1}{\pi \vdash E_1\; E_2 : \tau_2}$$

These data-typing rules, which underlie the ML languages, analyze a program in one pass and predict the range of values *(data type)* that the program’s phrases will produce when executed:

$$\gamma(bool) = Bool = \{true, false\}$$
$$\gamma(int) = Int = \{\ldots -1, 0, 1, \ldots\}$$
$$\gamma(\tau_1 \to \tau_2) = \{f : Val \mid \text{for all } a \in \gamma(\tau_1),\; f(a) \in \gamma(\tau_2)\}$$

where $Val = \bigcup_{i \geq 0} V_i$, such that $V_0 = \{\}$ and $V_{i+1} = Bool \cup Int \cup (V_i \to V_i)$.

A guess is needed for τ_1 in the hypotheis of the second typing rule. A typical implementation of the rules uses first-order unification to calculate an intelligent guess.

Big-step (natural) semantics is a multi-pass analysis

$$\sigma \vdash p : x \;=\; E \Downarrow f_p(\sigma)$$

$$\frac{\sigma \vdash C_1 \Downarrow \sigma_1 \quad \sigma_1 \vdash C_2 \Downarrow \sigma_2}{\sigma \vdash C_1;\, C_2 \Downarrow \sigma_2} \qquad \frac{f_{Et}(\sigma) \vdash C_1 \Downarrow \sigma_1 \quad f_{Ef}(\sigma) \vdash C_2 \Downarrow \sigma_2}{\sigma \vdash \texttt{if}\ E\ C_1\ C_2 \Downarrow \sigma_1 \sqcup \sigma_2}$$

$$\frac{f_{Et}(\sigma) \vdash C \Downarrow \sigma' \quad \sigma' \vdash \texttt{while}\ E\ C \Downarrow \sigma''}{\sigma \vdash \texttt{while}\ E\ C \Downarrow f_{Ef}(\sigma) \sqcup \sigma''} \qquad \bot \vdash C \Downarrow \bot$$

Recall that f_p is a transfer function and that f_{Et} and f_{Ef} "filter" the store, e.g.,

$f_{x>2t}\langle x:4, y:3\rangle = \langle x:4, y:3\rangle$, whereas $f_{x>2t}\langle x:0, y:3\rangle = \bot$.

An example: `if Even(x) (x=0) (while x≠3 (x = x+1))`

$\langle x{:}1\rangle \vdash \texttt{if Even(x)}\ (x=0)\ (\texttt{while}\ x \neq 3\ (x = x+1)) \Downarrow \bot \sqcup \langle x{:}3\rangle = \langle x{:}3\rangle$

$\bot \vdash x = 0 \Downarrow \bot$ $\quad$ $\langle x{:}1\rangle \vdash \texttt{while}\ x \neq 3 \ldots \Downarrow \bot \sqcup \langle x{:}3\rangle$

$\langle x{:}1\rangle \vdash x = x+1 \Downarrow \langle x{:}2\rangle$ $\quad$ $\langle x{:}2\rangle \vdash \texttt{while}\ x \neq 3 \ldots \Downarrow \bot \sqcup \langle x{:}3\rangle$

$\langle x{:}2\rangle \vdash x = x+1 \Downarrow \langle x{:}3\rangle$ $\quad$ $\langle x{:}3\rangle \vdash \texttt{while}\ x \neq 3 \ldots \Downarrow \langle x{:}3\rangle \sqcup \bot = \langle x{:}3\rangle$

$\bot \vdash x = x+1 \Downarrow \bot$ $\quad$ $\bot \vdash \texttt{while}\ x \neq 3 \ldots \Downarrow \bot$

An abstract big-step tree: using the same inference rules but with abstract transfer functions for **Parity** $= \{\bot, even, odd, \top\}$, we generate an abstract tree that is *infinite* but *regular*:

$\langle x{:}odd\rangle \vdash \texttt{if Even(x)}\ (x=0)\ (\texttt{while}\ x \neq 3\ (x = x+1)) \Downarrow \bot \sqcup X$

$\bot \vdash x = 0 \Downarrow \bot$ $\quad$ $\langle x{:}odd\rangle \vdash \texttt{while}\ x \neq 3 \ldots \Downarrow \langle x{:}odd\rangle \sqcup X = X$

$\langle x{:}odd\rangle \vdash x = x+1 \Downarrow \langle x{:}even\rangle$ $\quad$ $\langle x{:}even\rangle \vdash \texttt{while}\ x \neq 3 \ldots \Downarrow \bot \sqcup X$

$\langle x{:}even\rangle \vdash x = x+1 \Downarrow \langle x{:}odd\rangle$ $\quad$ $\langle x{:}odd\rangle \vdash \texttt{while}\ x \neq 3 \ldots \Downarrow X$

$\ldots$

Variable X denotes the answer from the repeated loop subderivation:

$$X = \langle x{:}odd\rangle \sqcup X$$

The least solution sets $X = \langle x{:}odd\rangle$.

Big-step semantics naturally supports interprocedural analysis

$$\frac{\texttt{func f(x) local y;}\ C. \qquad [x \mapsto [\![E]\!]\sigma][y \mapsto \bot] \vdash C \Downarrow \sigma'}{\sigma \vdash z \;=\; f(E) \Downarrow \sigma[z \mapsto \sigma'(y)]}$$

where $[\![E]\!]\sigma$ denotes E's value with σ, and $x \mapsto v$ assigns v to x.

Example:

```
func g(x) local z; z = x+1.
a = g(2); b = g(a); a = a*b
```

$\langle a{:}\bot, b{:}\bot\rangle \vdash a = g(2);\ b = g(a);\ a = a*b \Downarrow \langle a{:}even, b{:}even\rangle$

$\langle a{:}\bot, b{:}\bot\rangle \vdash a = g(2) \Downarrow \langle a{:}odd, b{:}\bot\rangle$ $\quad$ $\langle a{:}odd, b{:}\bot\rangle \vdash b = g(a);\ a = a*b \Downarrow \langle a{:}even, b{:}even\rangle$

$\langle x{:}even, z{:}\bot\rangle \vdash z = x+1 \Downarrow \langle x{:}even, z{:}odd\rangle$ $\quad$ $\langle a{:}odd, b{:}even\rangle \vdash a = a*b \Downarrow \langle a{:}even, b{:}even\rangle$

$\langle a{:}odd, b{:}\bot\rangle \vdash b = g(a); \Downarrow \langle a{:}odd, b{:}even\rangle$

$\langle x{:}odd, z{:}\bot\rangle \vdash z = x+1 \Downarrow \langle x{:}odd, z{:}even\rangle$

The derivation tree naturally separates the calling contexts.

Recursions (*) force accelerated termination (!):

`func fac(a) local b; if a = 0 (b = 1) (b = fac(a − 1); b = a * b)`
`c = fac(3)`

$\langle c:\bot\rangle \vdash c = \texttt{fac}(3) \Downarrow \langle c:\top\rangle$

$*\ \langle 3, \bot\rangle \vdash \texttt{if}\ a = 0\ (b = 1)(b = \texttt{fac}(a-1);\ b = a*b) \Downarrow \bot \sqcup \langle \top, \top\rangle = \langle \top, \top\rangle$

$\bot \vdash b = 1 \Downarrow \bot$ $\quad$ $\langle 3, \bot\rangle \vdash b = \texttt{fac}(a-1);\ b = a*b \Downarrow \langle \top, \top\rangle$

$\langle 3, \bot\rangle \vdash b = \texttt{fac}(a-1) \Downarrow \langle 3, \top\rangle$ $\quad$ $3, \top \vdash b = a*b \Downarrow \top, \top$

$*\ \langle 3, \bot\rangle \sqcup \langle 2, \bot\rangle = \langle \top, \bot\rangle \vdash \texttt{if}\ a = 0 \ldots \Downarrow \langle 0, 1\rangle \sqcup \langle \top, \top * X.b\rangle = X = \langle \top, \top\rangle$

$\langle 0, \bot\rangle \vdash b = 1 \Downarrow \langle 0, 1\rangle$ $\quad$ $\langle \top, \bot\rangle \vdash b = \texttt{fac}(a-1);\ b = a*b \Downarrow \langle \top, \top * X.b\rangle$

$\langle \top, \bot\rangle \vdash b = \texttt{fac}(a-1) \Downarrow \langle \top, X.b\rangle$ $\quad$ $\langle \top, X.b\rangle \vdash b = a*b \Downarrow \langle \top, \top * X.b\rangle$

$*!\ \langle \top, \bot\rangle \vdash \texttt{if}\ a = 0 \ldots \Downarrow X$

$X = \langle 0, 1\rangle \sqcup \langle \top, \top * X.b\rangle$ **The least solution sets** $X = \langle \top, \top\rangle$.

The traditional data-flow implementation uses *call strings*:

- Each procedure has its own control-flow graph, as does the main program. Each procedure invocation and return is drawn as a "goto" arc in the graph for the entire program.
- When the program is analyzed, the store is accompanied by a calling history, called the *call string*. (E.g., `main` calls `p` — the call string is `"main::p"`.)
- A finite bound, k, is placed on the call string's length — only the the k most recent invocations are remembered.
- Say that the call string is S, execution is in p, and p calls q. The call string is revised to $S' = (S :: p) \downarrow k$, and q's activation record labelled S' is used to execute q. At conclusion, control returns to $S.last$ — p — and the call string is shortened. (If the call string is empty, then the return "gos to" all possible return points!)

References

- A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, Chapter 10. Addison Wesley, 1986.
- P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. ACM POPL 1977.
- P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. PLILP 1992, Springer LNCS 631.
- Neil Jones and Flemming Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science, Vol. 4*, Oxford University Press, 1994.
- H.R. Nielson, F. Nielson. *Semantics with Applications.* Wiley, 1992. Available from `www.imm.dtu.dk/~riis`.
- H.R. Nielson, F. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
- D. Schmidt. Trace-Based Abstract Interpretation of Operational Semantics. J. Lisp and Symbolic Computation 10 (1998).