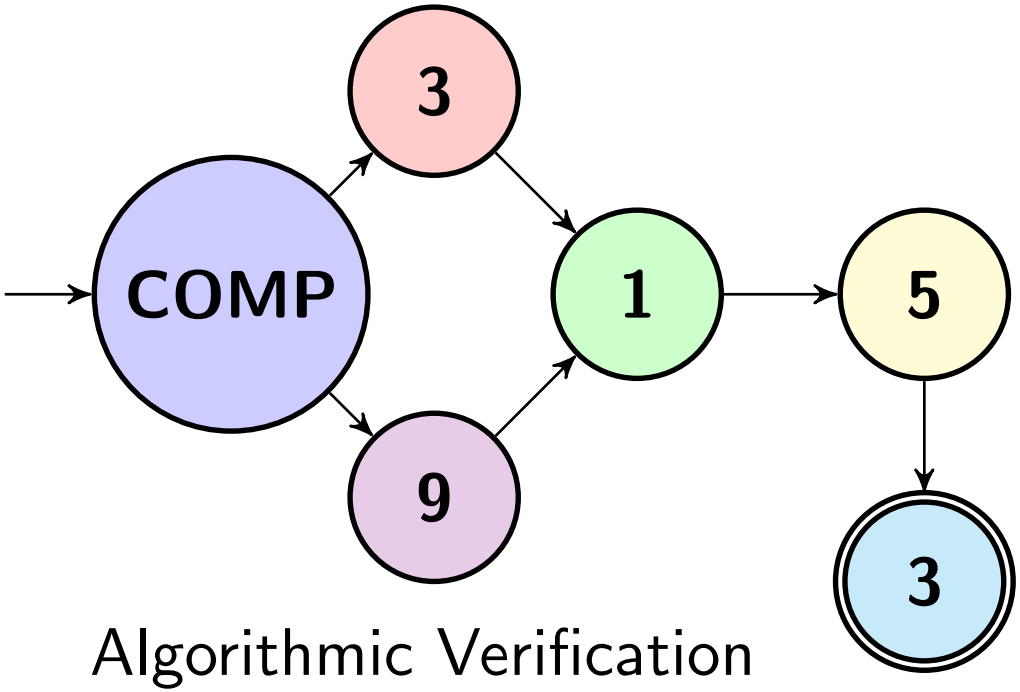


Algorithmic Verification

Abstract Interpretation and Static Analysis

Dr. Liam O'Connor
CSE, UNSW (for now)
Term 1 2020

Static Analysis

Static analysis is the automatic analysis of code without executing it. It has a variety of applications from security to performance optimisations.

Rice's Theorem

Any non-trivial property about the language recognized by a Turing machine is undecidable.

Our trick: Abstract away from the non-computable or intractable bits by approximating.

Caution

If we overapproximate, we may produce a lot of *false positives* (spurious errors), but our analysis will be *sound*. If we underapproximate, we may report that the program is fine when it isn't (*false negatives*), but our analysis will be *complete*.

Most tools aren't sound nor complete, because they're a mixture.

Math Recap: Lattices

Definition

A *lattice* $(L, \preceq)$ is a set L and a partial order $(\preceq) \subseteq L \times L$ where any set $X \subseteq L$ has both a *least upper bound* $\sup X \in L$ and a *greatest lower bound* $\inf X \in L$.
We define $\top = \sup L$ and $\bot = \inf L$

If I define a function $f : L \to L$ that is *monotone*, i.e:

$$x \preceq y \Rightarrow f(x) \preceq f(y)$$

I can prove that f has both a *least* and *greatest fixed point*, i.e.

Least The least fixed point μf of a function $f : L \to L$ is the smallest (wrt. $\preceq$) element $x \in L$ such that $f(x) = x$.

Greatest The greatest fixed point νf of a function $f : L \to L$ is the largest (wrt. $\preceq$) element $x \in L$ s.t. $f(x) = x$.

Knaster-Tarski Theorem

Let's prove it for greatest fixed points, given a lattice L and a monotone function f:

- Define $D = \{x \in L \mid x \preceq f(x)\}$.
- We know that $\forall m.\ \bot \preceq m$, so we know $\bot \in D$,and, by monotonicity, $f(\bot) \in D$, $f(f(\bot)) \in D$ etc.
- Let $u = \sup D$, the least upper bound.
- Hence for all $x \in D, x \preceq u$, and by monotonicity $f(x) \preceq f(u)$.
- Thus $x \preceq f(x) \preceq f(u)$. So $f(u)$ is also an upper bound of D.
- u is the least upper bound of D, so $u \preceq f(u)$. Thus $u \in D$.
- By monotonicity, $f(u) \preceq f(f(u))$, so $f(u) \in D$
- Because u is the least upper bound of D, $f(u) \leq u$.
- Therefore $f(u) = u$, i.e. u is a fixed point.
- All fixed points are in D, therefore u is the greatest fixed point.

Fixed Point

How do we compute fixed points?
For *finite lattices*, we can compute the least fixed point by *iterating* f from $\bot$, and the greatest by iterating from $\top$:

Let $\iota \in \{\top, \bot\}$ depending on which fixed point we want:

$$
\begin{aligned}
&\mathit{prev} := \iota \\
&\mathit{curr} := f(\mathit{prev}) \\
&\textbf{while } \mathit{curr} \neq \mathit{prev} \textbf{ do} \\
&\quad \mathit{prev} := \mathit{curr} \\
&\quad \mathit{curr} := f(\mathit{curr}) \\
&\textbf{od}
\end{aligned}
$$

Why does this terminate?

Abstract Interpretation

A very common use-case for fixed point computations is in *abstract interpretation*, a type of static analysis.

Key Idea

1. Replace concrete variables with approximate abstractions in an *abstract domain*, which is a lattice.
2. Approximate the program's semantics using monotonic functions defined over that domain.
3. Compute the least fixed point of these functions.

We have seen this before. Predicate abstraction is an example of abstract interpretation.

The WHILE Language

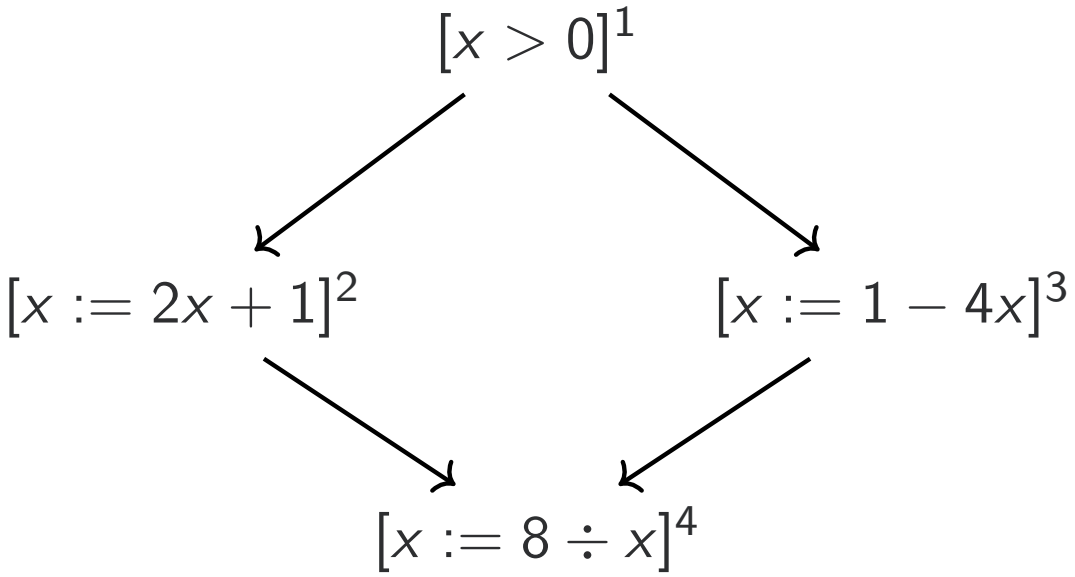
To make things easier, we will define a simple imperative programming language as follows:

$$
\begin{array}{lcl}
\mathcal{A} & ::= & \langle \textit{arithmetic expressions} \rangle \\
\mathcal{B} & ::= & \langle \textit{boolean expressions} \rangle \\
\mathcal{S} & ::= & [x := \mathcal{A}]^\ell \mid [\textbf{skip}]^\ell \mid \mathcal{S}_1; \mathcal{S}_2 \\
& \mid & \textbf{if } [\mathcal{B}]^\ell \textbf{ then } \mathcal{S} \textbf{ else } \mathcal{S} \\
& \mid & \textbf{while } [\mathcal{B}]^\ell \textbf{ do } \mathcal{S}
\end{array}
$$

Note that we *label* all statements and conditions (usually with a number). These labelled terms correspond to nodes on the control flow graph.

Examples

if $[x > 0]^1$ **then**
$[x := 2x + 1]^2$
else
$[x := 1 - 4x]^3$
$[x := 8 \div x]^4$



What abstract domain should we use to detect divide by zero?

Intervals

Let's define intervals n, m as either:

- $\emptyset$, the empty interval, or
- $[n, m]$ where $n, m \in \mathbb{Z} \cup \{+\infty, -\infty\}$.

We can define interval intersection $n \cap m$ as the interval where n and m overlap.
Likewise, define union $n \cup m$ as the smallest interval containing both n and m.

Observation

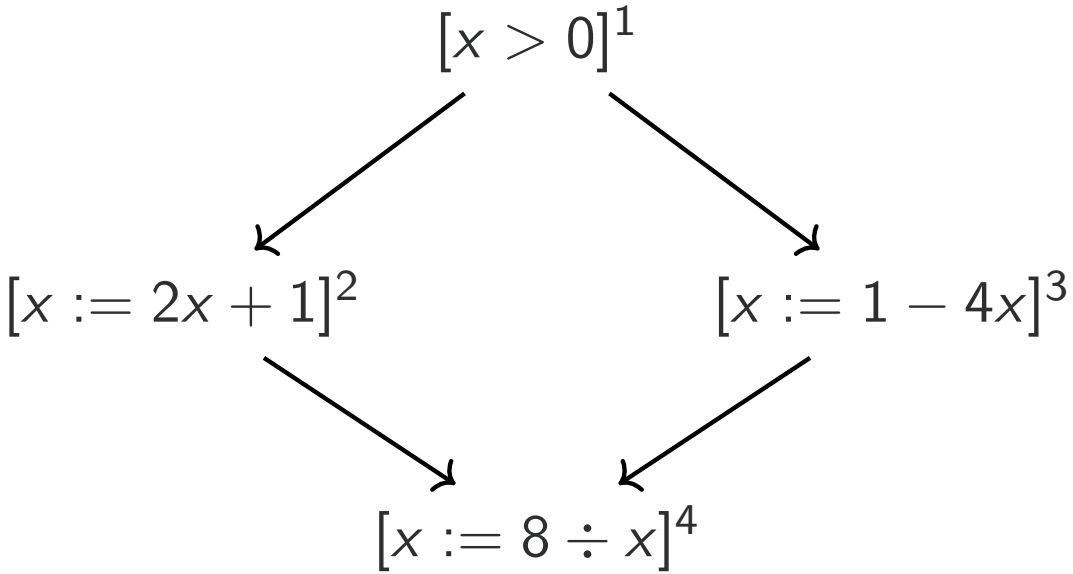
Define $\inf S = \bigcap S$ and $\sup S = \bigcup S$, then intervals form a lattice: $\bot = \emptyset$ and $\top = [-\infty, +\infty]$. The ordering $\preceq$ here is interval inclusion.

We can "lift" arithmetic operators to the interval level, where they apply to both bounds. Similarly define e.g. $\hat{3} = [3, 3]$.

Equation Systems

We define a series of *entry* interval equations x_i, and a series of *exit* equations x'_i describing the possible values of x before and after the statement i.

$$
\begin{array}{lcl}
x_1 & = & [-\infty, +\infty] \\
x'_1 & = & x_1 \\
x_2 & = & x'_1 \cap [1, +\infty] \\
x'_2 & = & \hat{2} \times x_2 + \hat{1} \\
x_3 & = & x'_1 \cap [-\infty, 0] \\
x'_3 & = & \hat{1} - \hat{4} \times x_3 \\
x_4 & = & x'_2 \cup x'_3 \\
x'_4 & = & \hat{8} \div x_4
\end{array}
$$



Observe that all operations used are monotone. How can we compute what the intervals are? Iterate to the least fixed point !

Least Fixed Points for Equation Systems

We start initialising all equations to $\bot$, and then iterate until the results stop changing. We can choose the equations in any (fair) order, and we will always reach a fixed point eventually. Some ways are faster than others.

$$
\begin{array}{lcll}
x_1 & = & \emptyset = [-\infty, +\infty] & = [-\infty, +\infty] \\
x_1' & = & \emptyset = [-\infty, +\infty] & = x_1 \\
x_2 & = & \emptyset = [1, +\infty] & = x_1' \cap [1, +\infty] \\
x_2' & = & \emptyset = [3, +\infty] & = \hat{2} \times x_2 + \hat{1} \\
x_3 & = & \emptyset = [-\infty, 0] & = x_1' \cap [-\infty, 0] \\
x_3' & = & \emptyset = [1, +\infty] & = \hat{1} - \hat{4} \times x_3 \\
x_4 & = & \emptyset = [1, +\infty] & = x_2' \cup x_3' \\
x_4' & = & \emptyset = [0, 8] & = \hat{8} \div x_4
\end{array}
$$

Seeing as $0 \notin x_4$, we know divide by zero is impossible.

Slow Convergence

Because the previous example had no loops, all equations converged after one step. Compare to this example:

$$
\begin{array}{l}
[n := 1]^1 \\
\textbf{while } [n < 1000]^2 \textbf{ do} \\
\quad [n := n + 1]^3 \\
[\textbf{skip}]^4
\end{array}
$$

Slightly simplified equations for presentation:

$$
\begin{array}{lcll}
n_1' & = & \emptyset = [1,1] & = [1,1] \\
n_2 & = & \emptyset = [1,1] = [1,2] = [1,3] = \cdots & = n_1' \cup n_3' \\
n_3 & = & \emptyset = [1,1] = [1,2] = [1,3] = \cdots & = n_2 \cap [-\infty, 999] \\
n_3' & = & \emptyset = [2,2] = [2,3] = [2,4] = \cdots & = n_3 + 1 \\
n_4 & = & \emptyset & = n_3' \cap [1000, +\infty]
\end{array}
$$

This is going to take a long time to converge! (1000 steps)

Widening

Our interval abstraction is too detailed, making our loop iterations take ages.

Solution

Let n be the value we are updating and m be the result of the next iteration. Then, we update with $n \triangledown m$ instead of m:

$$\begin{array}{lcl} \emptyset \triangledown m & = & m \\ n \triangledown \emptyset & = & n \\ [\ell_0, u_0] \triangledown [\ell_1, u_1] & = & [\textbf{if } \ell_1 < \ell_0 \textbf{ then } -\infty \textbf{ else } \ell_1 \\ & & , \textbf{if } u_1 > u_0 \textbf{ then } +\infty \textbf{ else } u_1] \end{array}$$

In other words, if we ever try to loosen a bound, we just extrapolate all the way to infinity.

This is an overapproximation, but it converges much faster than the normal iterative sequence does.

Loops with Widening

$$
\begin{array}{lcll}
n_1' & = & \emptyset = [1,1] & = [1,1] \\
n_2 & = & \emptyset = [1,1] = [1,+\infty] = [1,1000] & = n_1' \cup n_3' \\
n_3 & = & \emptyset = [1,1] = [1,999] = [1,999] & = n_2 \cap [-\infty, 999] \\
n_3' & = & \emptyset = [2,2] = [2,1000] = [2,1000] & = n_3 + 1 \\
n_4 & = & \emptyset = [1000,1000] & = n_3' \cap [1000, +\infty]
\end{array}
$$

Beyond Interval Anaylsis

Interval analysis is very effective but not very accurate, because it doesn't express the relationships between variables.
Predicate abstraction and *polyhedral models* do better in many cases, but are more complicated.
All are based on the same principle of least-fixed point of a system of equations.

Data-flow Analysis

Data-flow analysis is a type of static analysis used extensively in compilers.

Example

- *Available Expressions Analysis* – Compute what expressions **must** have already been computed (and don't need to be recomputed).
- *Live Variables Analysis* – Compute which variables **may** be read before next being written to (and thus hold important values).

Data-flow analyses may be *forwards* or *backwards*, and *may* or *must*.

AEA is a forwards must analysis. LVA is a backwards may analysis.

Step 1: Gen and Kill

Each location in the CFG has an associated gen set, of generated information, and kill set, of information that is no longer accurate.

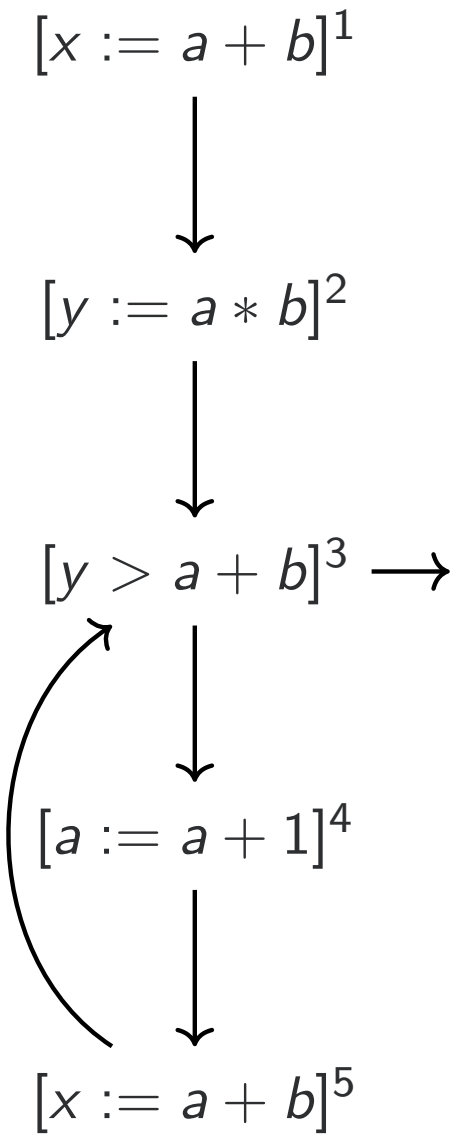
Example (AEA)

In AEA, $\text{gen}_{AE}(\ell)$ is the expressions evaluated (and not updated) in ℓ and $\text{kill}_{AE}(\ell)$ is those expressions updated by ℓ.
For example, $x := a + b$ would generate $\{a + b\}$, but kill any expression involving x.
Note: $a := a + 1$ would kill $a + 1$, not generate it. Why?

Example (LVA)

In LVA, $\text{gen}_{LV}(\ell)$ is the variables read (and not written to) in ℓ and $\text{kill}_{LV}(\ell)$ would be the variables written to in ℓ.
For example, $x := a + b$ would generate $\{a, b\}$ and kill $\{x\}$.

Available Expressions Example



ℓ	gen_{AE}	kill_{AE}
1	$\{a + b\}$	$\emptyset$
2	$\{a * b\}$	$\emptyset$
3	$\{a + b\}$	$\emptyset$
4	$\emptyset$	$\{a + b, a * b, a + 1\}$
5	$\{a + b\}$	$\emptyset$

What to do now?

Specify an equation system that relates these, then find the fixed point!

Forwards Must Analysis

The set of available expressions as we enter a location ℓ is the intersection of all available expressions from the predecessors to ℓ:

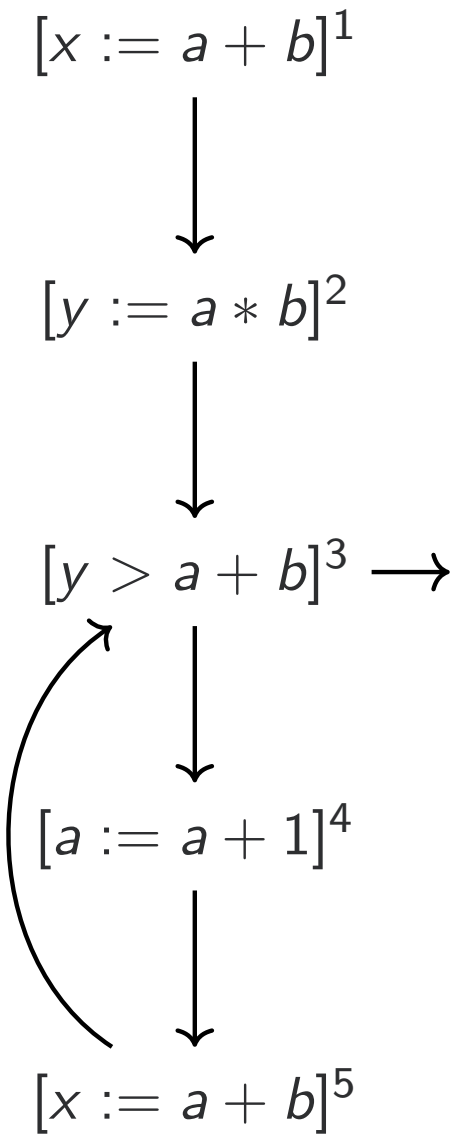
$$AE_{\text{entry}}(\ell) = \begin{cases} \emptyset & \text{if } \ell \in I_{\text{CFG}} \\ \bigcap \{ AE_{\text{exit}}(\ell') \mid (\ell', \ell) \in \delta_{\text{CFG}} \} & \text{otherwise} \end{cases}$$

We choose intersection because we want expressions that are **definitely** available no matter what route we took to get to ℓ (a **must** analysis).

The set of available expressions as we exit a location ℓ is the set that we entered with, minus anything we kill, plus anything we generate:

$$AE_{\text{exit}}(\ell) = (AE_{\text{entry}}(\ell) \setminus \text{kill}_{\text{AE}}(\ell)) \cup \text{gen}_{\text{AE}}(\ell)$$

Example for AEA

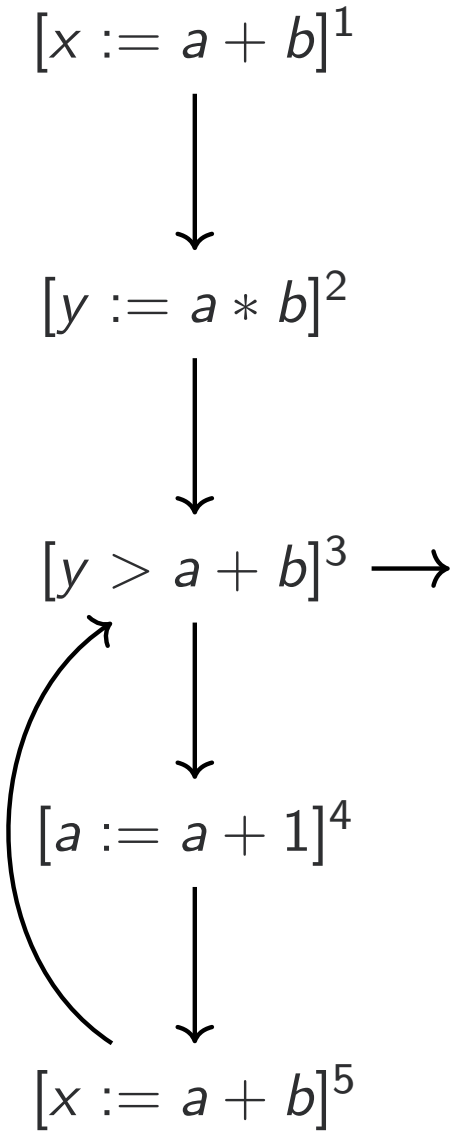


ℓ	gen_{AE}	kill_{AE}
1	$\{a + b\}$	$\emptyset$
2	$\{a * b\}$	$\emptyset$
3	$\{a + b\}$	$\emptyset$
4	$\emptyset$	$\{a + b, a * b, a + 1\}$
5	$\{a + b\}$	

ℓ	$AE_{\mathsf{entry}}(\ell)$	$AE_{\mathsf{exit}}(\ell)$
1	$\emptyset$	$AE_{\mathsf{entry}}(1) \cup \{a + b\}$
2	$AE_{\mathsf{exit}}(1)$	$AE_{\mathsf{entry}}(2) \cup \{a * b\}$
3	$AE_{\mathsf{exit}}(2) \cap AE_{\mathsf{exit}}(5)$	$AE_{\mathsf{entry}}(3) \cup \{a + b\}$
4	$AE_{\mathsf{exit}}(3)$	$AE_{\mathsf{entry}}(4) \setminus \mathsf{kill}_{AE}(4)$
5	$AE_{\mathsf{exit}}(4)$	$AE_{\mathsf{entry}}(5) \cup \{a + b\}$

Liam: Compute the LFP on the board

Results



ℓ	gen_{AE}	kill_{AE}
1	$\{a + b\}$	$\emptyset$
2	$\{a * b\}$	$\emptyset$
3	$\{a + b\}$	$\emptyset$
4	$\emptyset$	$\{a + b, a * b, a + 1\}$
5	$\{a + b\}$	

ℓ	$AE_{\text{entry}}(\ell)$	$AE_{\text{exit}}(\ell)$
1	$\emptyset$	$\{a + b\}$
2	$\{a + b\}$	$\{a + b, a * b\}$
3	$\{a + b\}$	$\{a + b\}$
4	$\{a + b\}$	$\emptyset$
5	$\emptyset$	$\{a + b\}$

Note $\ell = 3$ can be optimised, as $a + b$ is already computed!

Backwards May Analysis

The set of live variables as we exit a location ℓ is the union of live variables from the successors to ℓ:

$$LV_{\text{exit}}(\ell) = \begin{cases} \emptyset & \text{if } \ell \in F_{\text{CFG}} \\ \bigcup \{ LV_{\text{exit}}(\ell') \mid (\ell, \ell') \in \delta_{\text{CFG}} \} & \text{otherwise} \end{cases}$$

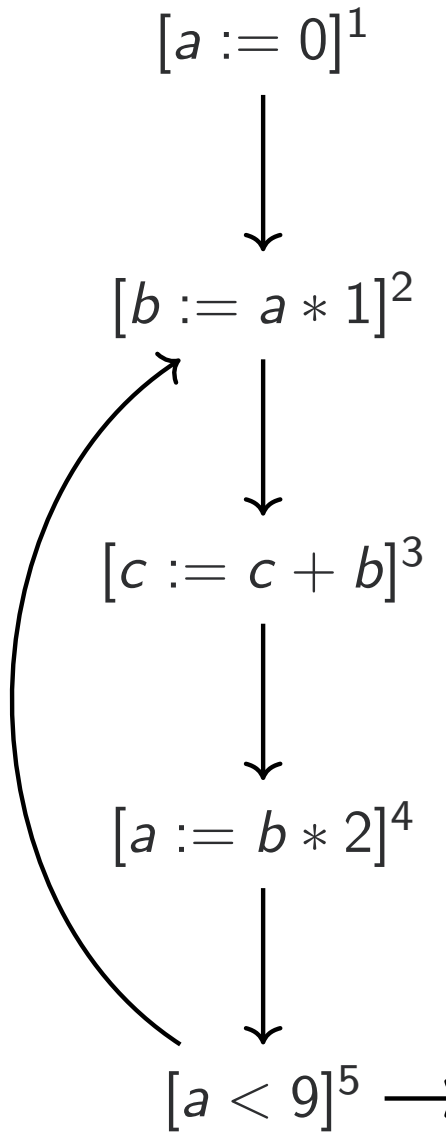
We choose union because we want any variables that **might be** used in a successor to be marked live in ℓ (a **may** analysis).

The set of live variables as we enter a location ℓ is the set of live variables at the exit, minus anything we kill (write to), plus anything we generate (read from):

$$LV_{\text{entry}}(\ell) = (LV_{\text{exit}}(\ell) \setminus \text{kill}_{\text{LV}}(\ell)) \cup \text{gen}_{\text{LV}}(\ell)$$

The entry and exit equations are flipped because this is a backwards analysis.

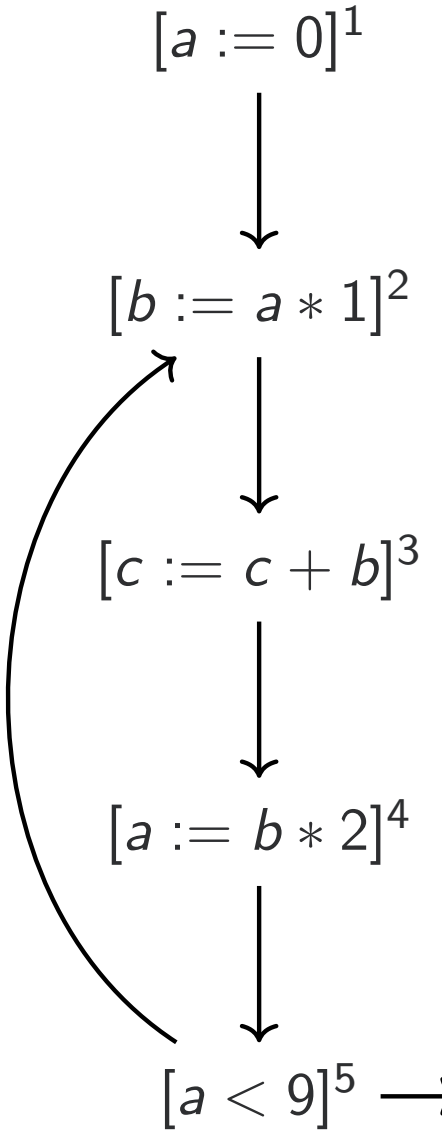
Example for LVA



ℓ	gen_{LV}	kill_{LV}
1	$\emptyset$	$\{a\}$
2	$\{a\}$	$\{b\}$
3	$\{b\}$	$\{c\}$
4	$\{b\}$	$\{a\}$
5	$\{a\}$	$\emptyset$

ℓ	$LV_{\text{entry}}(\ell)$	$LV_{\text{exit}}(\ell)$
1	$LV_{\text{exit}}(1) \setminus \{a\}$	$LV_{\text{entry}}(2)$
2	$LV_{\text{exit}}(2) \setminus \{b\} \cup \{a\}$	$LV_{\text{entry}}(3)$
3	$LV_{\text{exit}}(3) \setminus \{c\} \cup \{b\}$	$LV_{\text{entry}}(4)$
4	$LV_{\text{exit}}(4) \setminus \{a\} \cup \{b\}$	$LV_{\text{entry}}(5)$
5	$LV_{\text{exit}}(5) \cup \{a\}$	$LV_{\text{entry}}(2) \cup LV_{\text{entry}}(6)$

Results



ℓ	gen_{LV}	kill_{LV}
1	$\emptyset$	$\{a\}$
2	$\{a\}$	$\{b\}$
3	$\{b\}$	$\{c\}$
4	$\{b\}$	$\{a\}$
5	$\{a\}$	$\emptyset$

ℓ	$LV_{\text{entry}}(\ell)$	$LV_{\text{exit}}(\ell)$
1	$\emptyset$	$\{a\}$
2	$\{a\}$	$\{b\}$
3	$\{b\}$	$\{b, c\}$
4	$\{b, c\}$	$\{a, c\}$
5	$\{a, c\}$	$\{a, c\}$

Note: b and a are never simultaneously live!

Existence of Solutions

Solutions **always exist** to our data flow equations.
Why? Because $(2^{\mathcal{A}}, \subseteq)$ (for AEA) and $(2^{\mathsf{Var}}, \subseteq)$ (for LVA) are both lattices and all our functions are monotone. So the Knaster-Tarski theorem applies.

Bibliography

- F. Neilson: Principles of Program Analysis, Chapters 2 and 4, Springer 1999
- P. Cousot, A Tutorial on Abstract Interpretation, VMCAI 2005.
- Aho, Lam, Sethi, Ullman: Compilers: Principles Techniques and Tools (the Dragon Book), Second Edition.