

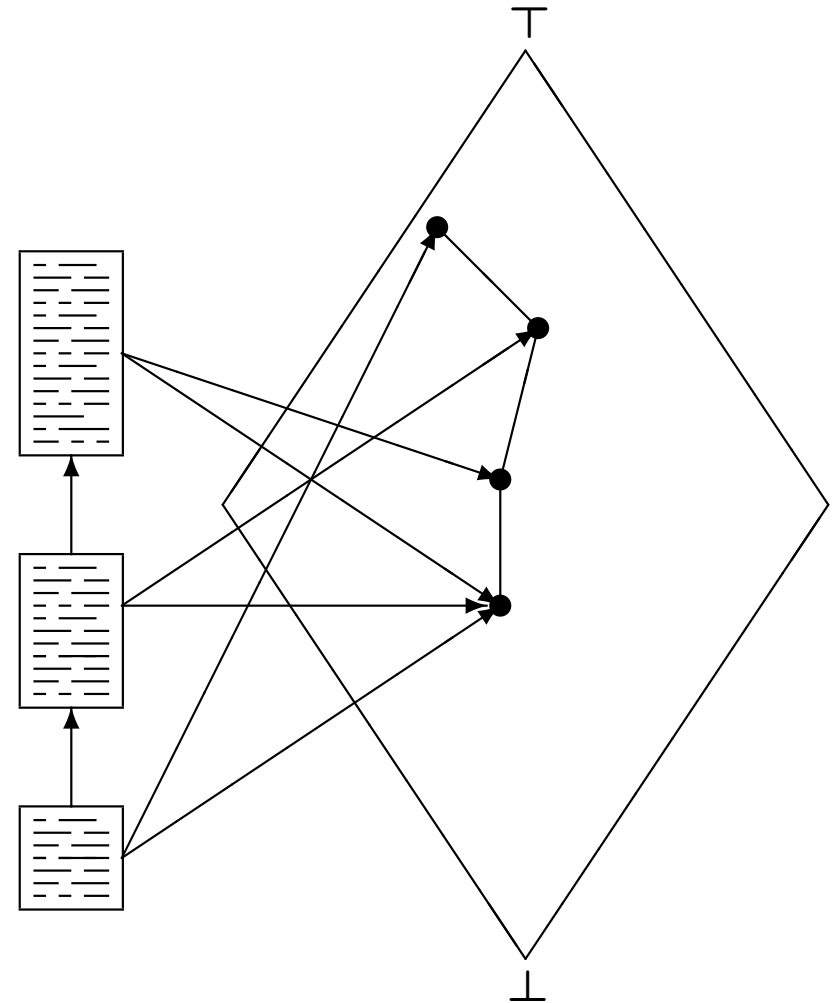
Static Analysis

COMP 520: Compiler Design (4 credits)

Alexander Krolik

`alexander.krolik@mail.mcgill.ca`

MWF 13:30-14:30, MD 279



Announcements (Monday, Apr 3)

- Milestone 4 due **Tuesday, April 4th 11:59PM** on GitHub
- Final Report due **Friday, April 7th 11:59PM** on GitHub
- Peephole Optimizer due **Friday, April 7th 11:59PM** on GitHub

Announcements (Wednesday, Apr 5)

- Final Report due **Monday, April 10th 11:59PM** on GitHub
- Peephole Optimizer due **Monday, April 10th 11:59PM** on GitHub
- Course evaluations! Please let us know what you think of the course!

Review classes:

- **Friday:** Scanner – bytecode generation
- **Monday:** Optimization – native code generation

Static analysis determines interesting properties of programs to enable some optimizations.

All interesting properties are actually undecidable, so the analysis computes a conservative approximation:

- if we say *yes*, then the property definitely holds;
- if we say *no*, then the property may or may not hold;
- only the *yes* answer will help us to perform the optimization;
- a trivial analysis will say *no* always; so
- the art is to say *yes* as often as possible.

Properties need not be simply *yes* or *no*, in which case the notion of *approximation* is more subtle.

Static analysis may take place:

- at the source code level;
- at some intermediate level; or
- at the machine code level.

Static analysis may look at:

- basic blocks only (called *local analysis*);
- an entire function (called *intraprocedural* or *global analysis*); or
- the whole program (called *whole program* or *interprocedural analysis*).

In each case, we are maximally pessimistic at the boundaries.

The precision and cost of an analysis rises as we include more information.

Simple static analysis:

- is merely advanced weeding;
- uses symbol and type information; and
- is recursive in the program syntax.

An example is the *definite assignment* requirement in Java and JOOS:

- local variables must be assigned before they are read;
- this is undecidable; but
- the language specification dictates a specific conservative approximation.

For each program point, compute a set of local variables that:

- contains only variables that have definitely been assigned;
- may be too small, since the analysis is conservative; and
- depends on the set computed for the previous program point.

It accepts:

```
int k;  
if (flag) k = 3; else k = 4;  
System.out.println(k);
```

but rejects:

```
int k;  
if (flag) k = 3;  
if (!flag) k = 4;  
System.out.println(k);
```

JOOS code for statements:

```
ASNSET *defasnSTATEMENT (STATEMENT *s, ASNSET *before) {
    if (s!=NULL) {
        switch (s->kind) {
            case skipK:
                return before;
            case expK:
                return defasnEXP (s->val.expS, before);
            case returnK:
                if (s->val.returnS!=NULL)
                    (void) defasnEXP (s->val.returnS, before);
                return setUniversal();
            case sequenceK:
                return defasnSTATEMENT (s->val.sequenceS.second,
                    defasnSTATEMENT (s->val.sequenceS.first,
                    before)
                );
        }
    }
    [...]
}
```



```
ASNSET *defasnSTATEMENT (STATEMENT *s, ASNSET *before) {
    [...]
    case ifelseK:
        return
            setIntersect (
                defasnSTATEMENT (s->val.ifelseS.thenpart,
                    defasnEXPassume (s->val.ifelseS.condition,
                        before, 1)
                ),
                defasnSTATEMENT (s->val.ifelseS.elsepart,
                    defasnEXPassume (s->val.ifelseS.condition,
                        before, 0)
                )
            );
    [...]
}
```

To make the analysis more precise, it considers boolean expressions in more detail.

The procedure `def asnEXP assume (. . . , b)` assumes the expression evaluates to `b`.

This refinement handles a case like:

```
int k;  
if (a > 0 && (k = b) > 0) {  
    System.out.println(k);  
}
```

which would otherwise be rejected.

In general, a static analysis becomes more precise when it may make further assumptions about the context.

The definite assignment analysis is particularly simple:

there are no recursive dependencies between the computed sets.

This allows a simple implementation: a top-down traversal of the parse tree.

For more sophisticated analyses, we generate equations and compute the solution as a *fixed point*.

For the JIT compiler, we want to optimize the use of registers:

```
mov 1, R3       $\implies$   mov 1, R1  
mov R3, R1
```

This requires knowledge about the future uses of registers:

The optimization is only sound if the value of `R3` is not used later on.

For basic block register allocation, which variables need to be written back to memory?

The naïve scheme:

- must write all those variables that are *only* in registers.

A better scheme:

- write all those variables that are only in registers *and* whose values might be used later on.

This could avoid many useless spills.

In both examples, we need to know if some R_i might be used later on. If so, it is called *live*; otherwise, it is called *dead*.

A static analysis can conservatively approximate liveness at each program point.

Exact liveness is of course undecidable.

A trivial analysis will call everything live, which precludes all optimizations.

A superior analysis will identify more dead variables.

Liveness analysis for VirtualRISC:

- build a control flow graph (goto graph);
- define dataflow equations for each node;
- compute the least solution of these equations.

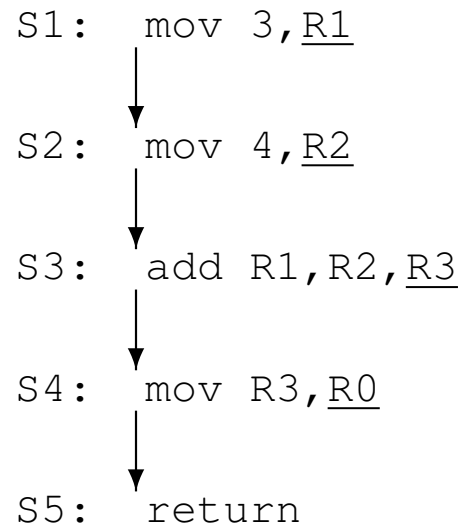
For basic blocks the computation is trivial.

For intraprocedural analysis we must compute a minimal fixed point in a lattice.

Consider a simple basic block:

```
mov 3, R1  
mov 4, R2  
add R1, R2, R3  
mov R3, R0  
return
```

The underlined registers are written (defined), the others are merely read (used).

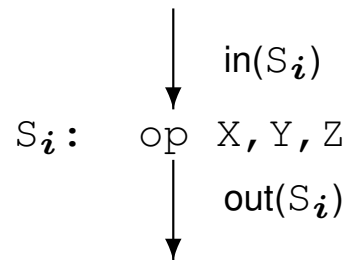
The control flow graph is:

Each instruction uses some registers and defines some registers:

	$uses(S_i)$	$defines(S_i)$
S1: <code>mov 3, <u>R1</u></code>	{ }	{ R1 }
↓		
S2: <code>mov 4, <u>R2</u></code>	{ }	{ R2 }
↓		
S3: <code>add R1, R2, <u>R3</u></code>	{ R1, R2 }	{ R3 }
↓		
S4: <code>mov R3, <u>R0</u></code>	{ R3 }	{ R0 }
↓		
S5: <code>return</code>	{ R0 }	{ }

The register R0 is implicitly used for the return value.

Let $\text{out}(S_i)$ be the variables that are live just *after* S_i and $\text{in}(S_i)$ those that are live just *before* S_i :



Then we have the dataflow equation:

$$\text{in}(S_i) = \text{uses}(S_i) \cup (\text{out}(S_i) - \text{defines}(S_i))$$

We add those registers that are used in the current instruction and delete those that are defined here.

Since $\text{out}(S5) = \{ \}$, it follows that:

$$\text{in}(S5) = \text{uses}(S5) = \{R0\}$$

We can continue to unravel the equations:

$$\text{out}(S4) = \text{in}(S5) = \{R0\}$$

$$\begin{aligned}\text{in}(S4) &= \text{uses}(S4) \cup (\text{out}(S4) - \text{defines}(S4)) \\ &= \{R3\} \cup (\{R0\} - \{R0\}) \\ &= \{R3\}\end{aligned}$$

$$\text{out}(S3) = \text{in}(S4) = \{R3\}$$

$$\begin{aligned}\text{in}(S3) &= \text{uses}(S3) \cup (\text{out}(S3) - \text{defines}(S3)) \\ &= \{R1, R2\} \cup (\{R3\} - \{R3\}) \\ &= \{R1, R2\}\end{aligned}$$

and so on:

		$uses(S_i)$	$defines(S_i)$	$in(S_i)$
S1:	mov 3, <u>R1</u>	{}	{R1}	{}
	↓			
S2:	mov 4, <u>R2</u>	{}	{R2}	{R1}
	↓			
S3:	add R1, R2, <u>R3</u>	{R1, R2}	{R3}	{R1, R2}
	↓			
S4:	mov R3, <u>R0</u>	{R3}	{R0}	{R3}
	↓			
S5:	return	{R0}	{}	{R0}

In basic blocks we use the equation:

$$\text{out}(S_i) = \text{in}(S_{i+1})$$

If we have branches, then a node in the control flow graph may have several successors.

In this case, we must use the equation:

$$\text{out}(S_i) = \bigcup_{x \in \text{succ}(S_i)} \text{in}(x)$$

But now the equations are cyclic and cannot simply be unraveled.

Consider the small piece of C code:

```

int i, sum_even, sum_odd, sum;
i = 1;
sum_even = 0;
sum_odd = 0;
sum = 0;
while (i < 10) {
    if (i%2 == 0) sum_even = sum_even + i;
    else sum_odd = sum_odd + i;
    sum = sum + i;
    i++;
}

```

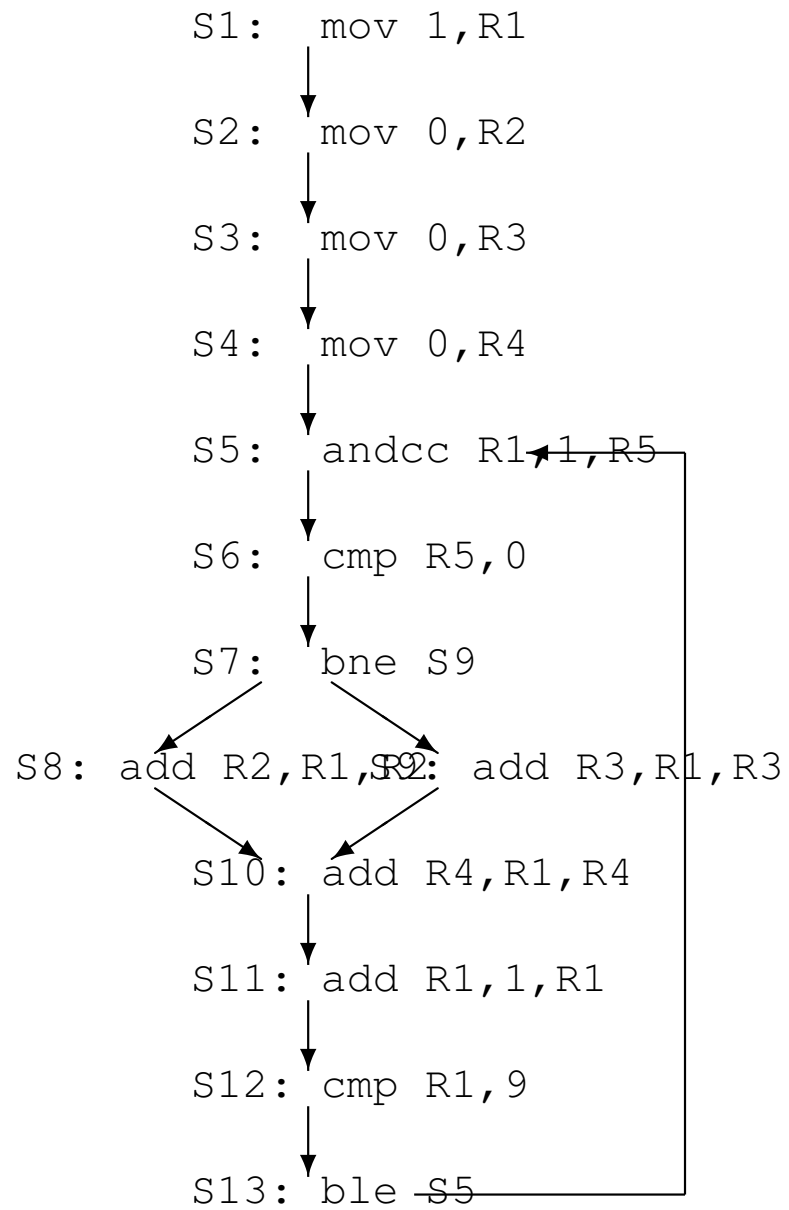
It yields the following VirtualRISC code:

```

mov 1,R1           // R1 is i
mov 0,R2           // R2 is sum_even
mov 0,R3           // R3 is sum_odd
mov 0,R4           // R4 is sum
loop:
andcc R1,1,R5      // R5 = R1 & 1
cmp R5,0
bne else           // if R5 != 0 goto else
add R2,R1,R2       // R2 = R2 + R1; even case
b endif
else:
add R3,R1,R3       // R3 = R3 + R1; odd case
endif:
add R4,R1,R4       // R4 = R4 + R1; update sum
add R1,1,R1        // R1 = R1 + 1; increment i
cmp R1,9
ble loop           // if i <= 9 goto loop

```

The control flow graph:



To unravel the liveness equations, we should start with:

$$\text{out}(S13) = \text{in}(S5)$$

... but we have not computed $\text{in}(S5)$ yet, so this will not work!

If $\text{in}(S1), \dots, \text{in}(S13)$ are known, then we can unravel the code as before and obtain the sets $\text{in}(S1), \dots, \text{in}(S13)$ once again.

But this means that unraveling is a function:

$$f : \mathcal{P}(R)^{13} \rightarrow \mathcal{P}(R)^{13}$$

where $R = \{R1, R2, \dots, R5\}$.

A solution is a fixed point, and we want the minimal one.

Two fundamental observations:

- the set $D = \mathcal{P}(R)^{13}$ is a finite *lattice*:

$$\forall x, y \in D : x \sqcap y \in D \wedge x \sqcup y \in D$$

where \sqsubseteq is point-wise set inclusion; and

- the unraveling function f is *monotonic*:

$$\forall x, y \in D : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

since $g(x) = A \cup (x - B)$ is monotonic.

The fixed point theorem:

Any monotonic function f on a finite lattice D has the unique minimal fixed point:

$$\bigsqcup_i f^i(\perp)$$

which is always obtained after finitely many iterations.

For $D = \mathcal{P}(R)^{13}$ we have that:

$$\perp = (\emptyset, \emptyset, \dots, \emptyset)$$

so we start with the sets $\text{in}(S_i) = \{ \}$ and keep unraveling until they no longer change.

Note that:

$$\top = (R, R, \dots, R)$$

is always a safe answer, but clearly useless and pessimistic.

Observe that the maximal fixed-point:

$$\sqcap_i f^i(\top)$$

may in general be smaller than \top .

Computing the minimal fixed point:

	uses	defs	succ	\perp	$f(\perp)$	$f^2(\perp)$
S1		R1	S2	{}	{}	{}
S2		R2	S3	{}	{}	{}
S3		R3	S4	{}	{}	{}
S4		R4	S5	{}	{}	{R1}
S5	R1	R5	S6	{}	{R1}	{R1}
S6	R5		S7	{}	{R5}	{R5}
S7			S8, S9	{}	{}	{R1, R2, R3}
S8	R1, R2	R2	S10	{}	{R1, R2}	{R1, R2, R4}
S9	R1, R3	R3	S10	{}	{R1, R3}	{R1, R3, R4}
S10	R1, R4	R4	S11	{}	{R1, R4}	{R1, R4}
S11	R1	R1	S12	{}	{R1}	{R1}
S12	R1		S13	{}	{R1}	{R1}
S13			S5	{}	{}	{R1}

The function is:

$$f(X_1, X_2, \dots, X_{13}) = (Y_1, Y_2, \dots, Y_{13})$$

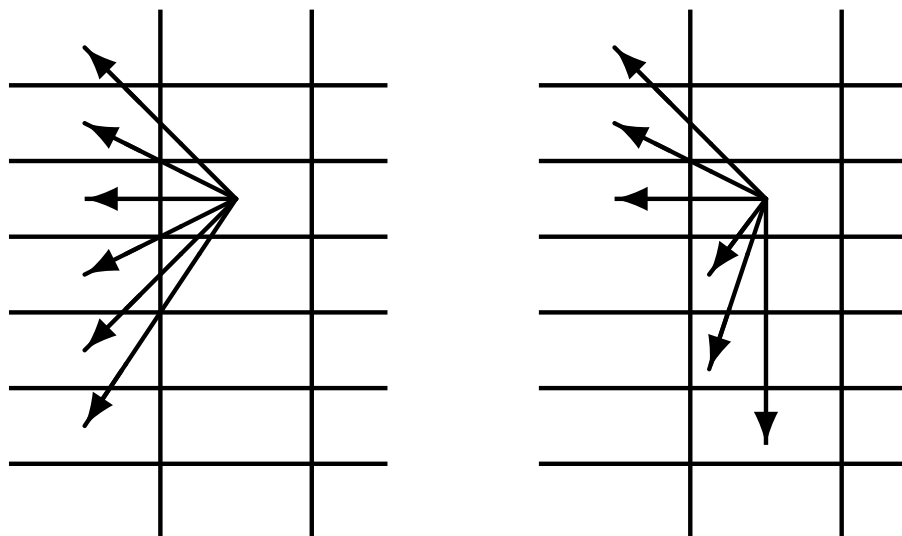
where:

$$Y_i = \text{uses}(S_i) \cup \left(\bigcup_{S_j \in \text{succ}(S_i)} X_j - \text{defs}(S_i) \right)$$

	$f^3(\perp)$	$f^4(\perp)$	$f^5(\perp)$
S1	{}	{}	{}
S2	{}	{R1}	{R1}
S3	{R1}	{R1}	{R1}
S4	{R1}	{R1}	{R1, R2, R3}
S5	{R1}	{R1, R2, R3}	{R1, R2, R3, R4}
S6	{R1, R2, R3, R5}	{R1, R2, R3, R4, R5}	{R1, R2, R3, R4, R5}
S7	{R1, R2, R3, R4}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S8	{R1, R2, R4}	{R1, R2, R4}	{R1, R2, R4}
S9	{R1, R3, R4}	{R1, R3, R4}	{R1, R3, R4}
S10	{R1, R4}	{R1, R4}	{R1, R4}
S11	{R1}	{R1}	{R1}
S12	{R1}	{R1}	{R1, R2, R3}
S13	{R1}	{R1, R2, R3}	{R1, R2, R3, R4}

	$f^6(\perp)$	$f^7(\perp)$	$f^8(\perp)$
S1	{}	{}	{}
S2	{R1}	{R1}	{R1}
S3	{R1, R2}	{R1, R2}	{R1, R2}
S4	{R1, R2, R3}	{R1, R2, R3}	{R1, R2, R3}
S5	{R1, R2, R3, R4}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S6	{R1, R2, R3, R4, R5}	{R1, R2, R3, R4, R5}	{R1, R2, R3, R4, R5}
S7	{R1, R2, R3, R4}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S8	{R1, R2, R4}	{R1, R2, R4}	{R1, R2, R3, R4}
S9	{R1, R3, R4}	{R1, R3, R4}	{R1, R2, R3, R4}
S10	{R1, R4}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S11	{R1, R2, R3}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S12	{R1, R2, R3, R4}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S13	{R1, R2, R3, R4}	{R1, R2, R3, R4}	{R1, R2, R3, R4}

A turbo fixed point technique:



The improved function is:

$$f_{\Delta}(X_1, X_2, \dots, X_{13}) = (Y_1, Y_2, \dots, Y_{13})$$

where:

$$Y_i = \text{uses}(S_i) \cup \left(\bigcup_{S_j \in \text{succ}(S_i)} Z_j - \text{defs}(S_i) \right)$$

$$Z_j = \begin{cases} Y_j & \text{if } j > i \\ X_j & \text{otherwise} \end{cases}$$

Improved fixed point computation:

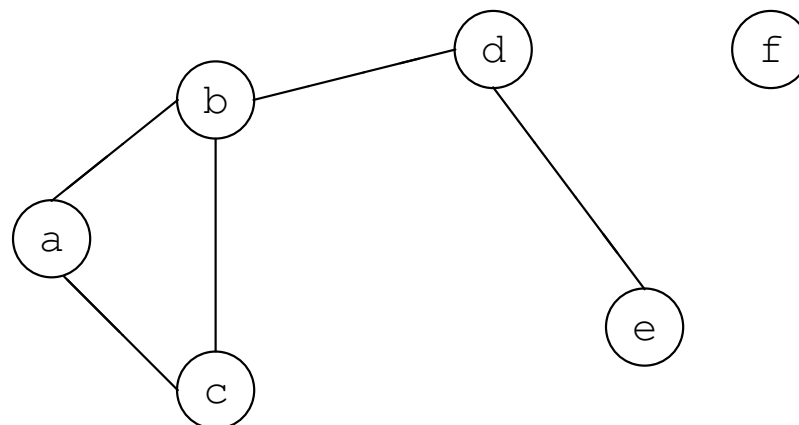
	\perp	$f_{\Delta}(\perp)$	$f_{\Delta}^2(\perp)$	$f_{\Delta}^3(\perp)$
S1	{}	{}	{}	{}
S2	{}	{}	{R1}	{R1}
S3	{}	{}	{R1, R2}	{R1, R2}
S4	{}	{}	{R1, R2, R3}	{R1, R2, R3}
S5	{}	{R1}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S6	{}	{R5}	{R1, R2, R3, R4, R5}	{R1, R2, R3, R4, R5}
S7	{}	{}	{R1, R2, R3, R4}	{R1, R2, R3, R4}
S8	{}	{R1, R2}	{R1, R2, R4}	{R1, R2, R3, R4}
S9	{}	{R1, R3}	{R1, R3, R4}	{R1, R2, R3, R4}
S10	{}	{R1, R4}	{R1, R4}	{R1, R2, R3, R4}
S11	{}	{R1}	{R1}	{R1, R2, R3, R4}
S12	{}	{R1}	{R1}	{R1, R2, R3, R4}
S13	{}	{}	{R1}	{R1, R2, R3, R4}

Number of iterations is down from 8 to 3.

Liveness analysis is used for register allocation in optimizing compilers.

In the basic block case, reduce spills to those variables that are only in registers *and* live.

In the intraprocedural case, construct a graph whose nodes are variables:



and where edges connect nodes that are live at the same time.

Register allocation is now reduced to finding a minimal graph coloring:

$$\{ \{a, d, f\}, \{b, e\}, \{c\} \}$$

and assigning a register to each color.

Liveness analysis is a *backwards* analysis, since we unravel from the future towards the past.

An example of a *forwards* analysis is constant propagation:

S1:	mov 3, R1	{ (R0, ?) , (R1, ?) , (R2, ?) , (R3, ?) }
	↓	
S2:	mov 4, R2	{ (R0, ?) , (R1, 3) , (R2, ?) , (R3, ?) }
	↓	
S3:	add R1, R2, R3	{ (R0, ?) , (R1, 3) , (R2, 4) , (R3, ?) }
	↓	
S4:	mov R3, R0	{ (R0, ?) , (R1, 3) , (R2, 4) , (R3, 7) }
	↓	
S5:	return	{ (R0, 7) , (R1, 3) , (R2, 4) , (R3, 7) }

A basic static analysis of JOOS and other object-oriented languages is *type inference*.

Given an expression, what are the possible classes of the objects to which it may evaluate?

The exact answer is undecidable, so we must conservatively approximate:

- we will accept a set that is too large;
- we want it as small as possible; and
- a trivial answer includes all classes.

This analysis is interprocedural and requires access to the whole program.

Possible uses of type inference:

- inline methods when there is only one possible receiver;
- eliminate run-time checks that can be decided statically;
- remove code that is never executed; and
- approximate the control flow graph to enable other static analyses.

In each case, smaller inferred sets will give better results.

The constraint technique:

- assign a variable $\llbracket E \rrbracket$ to each occurrence of an expression E ;
- assign a variable $\llbracket m \rrbracket$ to each occurrence of a method m ;
- the variables range over the set of all classes $C = \{C_1, C_2, \dots, C_n\}$;
- each parse tree node generates a local constraint on the variables; and
- the global minimal solution of these constraints is finally computed.

Again, we must compute a minimal fixed point in a finite lattice.

Each constraint models the flow of objects:

- the assignment “`i = E`” yields: $\llbracket E \rrbracket \subseteq \llbracket i \rrbracket$;
- the creation “`new C()`” yields: $\{C\} \subseteq \llbracket \text{new } C() \rrbracket$;
- the cast “`C(E)`” yields:
 $\{C\} \subseteq \llbracket C(E) \rrbracket$;
- the constant “`this`” yields: $\{C\} \subseteq \llbracket \text{this} \rrbracket$,
where C is the surrounding class; and
- the statement “`return E`” yields: $\llbracket E \rrbracket \subseteq \llbracket m \rrbracket$,
where m is the surrounding method.

The method invocation:

$$E.m(E_1, E_2, \dots, E_k)$$

yields the *conditional* constraints:

$$C_i \in \llbracket E \rrbracket \Rightarrow \left\{ \begin{array}{l} \llbracket E_1 \rrbracket \subseteq \llbracket x_1 \rrbracket \\ \llbracket E_2 \rrbracket \subseteq \llbracket x_2 \rrbracket \\ \vdots \\ \llbracket E_k \rrbracket \subseteq \llbracket x_k \rrbracket \\ \llbracket m \rrbracket \subseteq \llbracket E.m(E_1, E_2, \dots, E_k) \rrbracket \end{array} \right.$$

whenever the class C_i implements a method named m which accepts k arguments named x_1, x_2, \dots, x_k .

Since the constraint:

$$v \subseteq w$$

holds if and only if the equality:

$$w = v \cup w$$

does, we can rewrite a set of constraints into a function:

$$f : \mathcal{P}(C)^k \rightarrow \mathcal{P}(C)^k$$

such that fixed-points of f correspond to solutions to the constraints.

For the example constraints:

$$v_1 \subseteq v_2$$

$$c_3 \in v_2 \Rightarrow v_3 \subseteq v_1$$

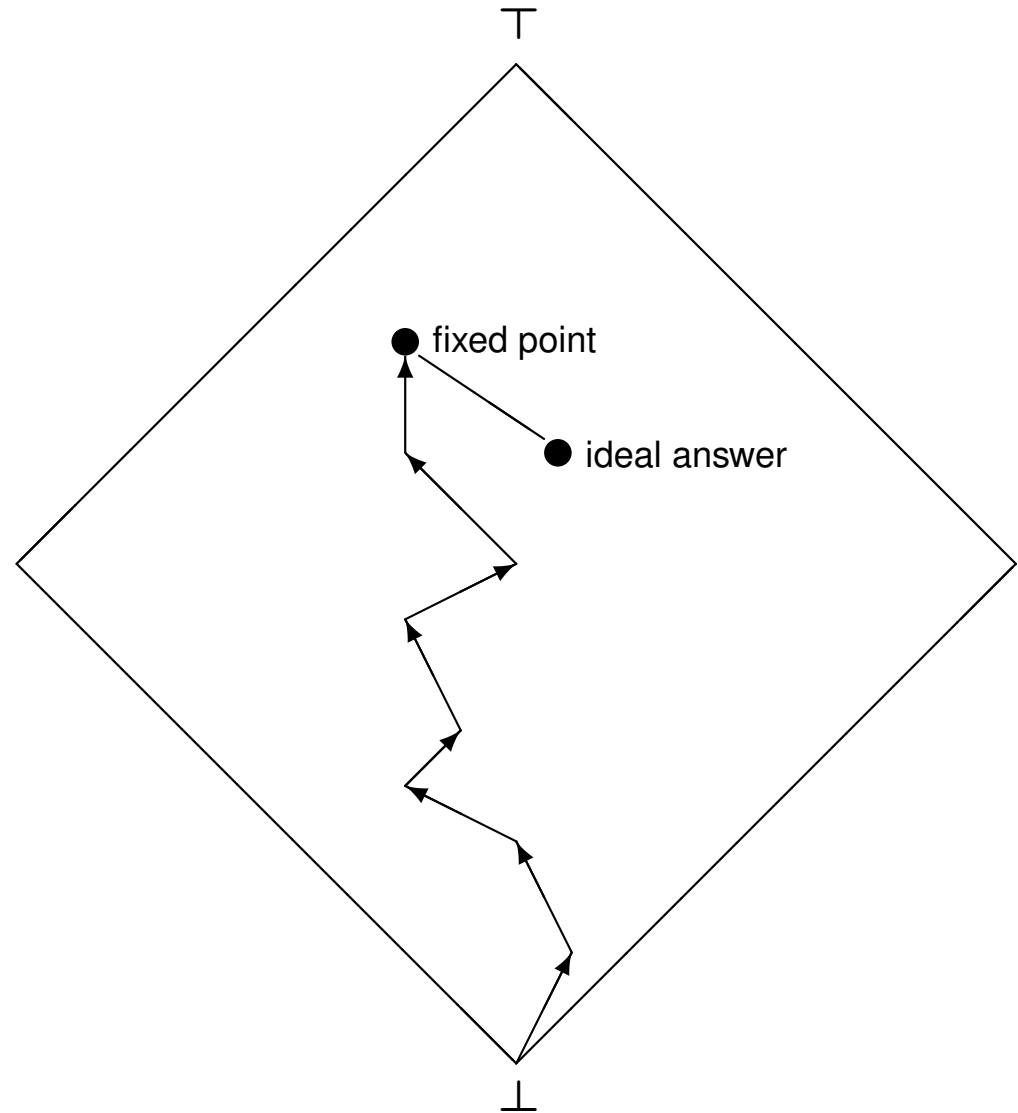
$$\{c_7\} \subseteq v_3$$

we get the function:

$$f(X_1, X_2, X_3) = \begin{cases} (X_1 \cup X_3, X_1 \cup X_2, \{c_7\} \cup X_3) & \text{if } c_3 \in X_2 \\ (X_1, X_1 \cup X_2, \{c_7\} \cup X_3) & \text{otherwise} \end{cases}$$

Solving the constraints:

- $\mathcal{P}(C)^k$ is a finite lattice;
- each function f is monotonic; and
- the least fixed point of f is the unique smallest solution of the constraints.



A tiny JOOS sketch:

```

public class A {
    public A() { super(); }
    public A id(A x) { return x; }
}

public class B extends A {
    public B() { super(); }
    public B me() { return (B)(new A()).id(this); }
}

```

The generated constraints are:

$$\begin{aligned}
 \llbracket x \rrbracket_A &\subseteq \llbracket id \rrbracket_A \\
 \llbracket x \rrbracket_B &\subseteq \llbracket id \rrbracket_B \\
 \llbracket (B)(new A()).id(this) \rrbracket &\subseteq \llbracket me \rrbracket \\
 \{B\} &\subseteq \llbracket (B)(new A()).id(this) \rrbracket \\
 \{B\} &\subseteq \llbracket this \rrbracket \\
 \{A\} &\subseteq \llbracket new A() \rrbracket \\
 A \in \llbracket new A() \rrbracket &\Rightarrow \llbracket this \rrbracket \subseteq \llbracket x \rrbracket_A \\
 A \in \llbracket new A() \rrbracket &\Rightarrow \llbracket id \rrbracket_A \subseteq \llbracket (new A()).id(this) \rrbracket \\
 B \in \llbracket new A() \rrbracket &\Rightarrow \llbracket this \rrbracket \subseteq \llbracket x \rrbracket_B \\
 B \in \llbracket new A() \rrbracket &\Rightarrow \llbracket id \rrbracket_B \subseteq \llbracket (new A()).id(this) \rrbracket
 \end{aligned}$$

The minimal solution is:

$$\begin{aligned}
 \llbracket new A() \rrbracket &= \{A\} \\
 \llbracket x \rrbracket_A &= \llbracket id \rrbracket_A = \llbracket this \rrbracket = \llbracket (new A()).id(this) \rrbracket = \{B\} \\
 \llbracket (B)(new A()).id(this) \rrbracket &= \{B\} \\
 \llbracket x \rrbracket_B &= \llbracket id \rrbracket_B = \{\}
 \end{aligned}$$

The generated code for the `me` method is:

```
.method public me()LB;
.limit locals 1
.limit stack 2
new A
dup
invokenonvirtual A/<init>()V
aload_0
invokevirtual A/id(LA;)LA;
checkcast B
areturn
.end method
```

The information $[[\text{new } A() .id(\text{this})]] = \{B\}$ eliminates the `checkcast` instruction.

That $[[\text{new } A()]] = \{A\}$ is a singleton further allows inlining of the `id` method:

```
.method public me()LB;
.limit locals 1
.limit stack 1
aload_0
areturn
.end method
```

With type inference, many little methods become almost free.

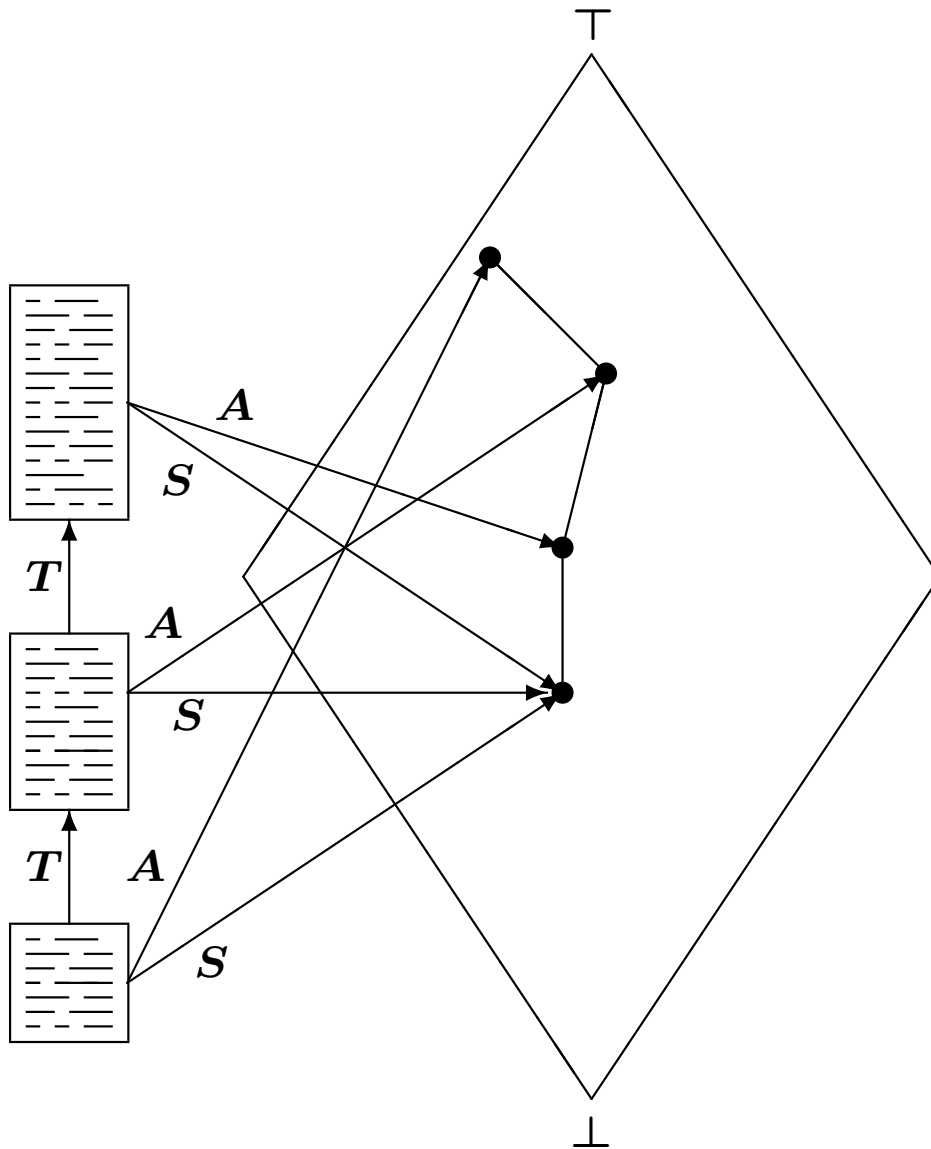
Improving analyses by transformations:

- let P be our set of programs;
- let $S : P \rightarrow D$ be an ideal static analysis (uncomputable); and
- let $T : P \rightarrow P$ be a program transformation that preserves the semantics.

Since S gives the ideal information, clearly $S(T(p)) = S(p)$ for all $p \in P$.

However, if $A : P \rightarrow D$ is a conservative approximation to S , then $A(T(p))$ may be different from $A(p)$, perhaps even better.

Transformations boost analyses:



The transformation T :

- may unfold the program to make it more explicit; or
- may itself be an optimization.