Lectures Outline

- 15/12 (today)
  - Static analysis
  - Basic principles of code analysis, Abstract Interpretation
  - Abstract state, state merging, fixpoints
- 16/12 (tomorrow)
  - Optimizations in dynamic languages within LLVM
  - Hands on with Rift (dead code/instruction elimination)

Static Analysis

MI-LCF

Static Analysis

Lecture Notes on Static Analysis

Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark
mis@brics.dk

Abstract

These notes present principles and applications of static analysis of programs. We cover type analysis, lattice theory, control flow graphs, dataflow analysis, fixed-point algorithms, narrowing and widening, interprocedural analysis, control flow analysis, and pointer analysis. A tiny imperative programming language with heap pointers and function pointers is subjected to numerous different static analyses illustrating the techniques that are presented.

The style of presentation is intended to be precise but not overly formal. The readers are assumed to be familiar with advanced programming language concepts and the basics of compiler construction.

Static Analysis

- (from the above:)

> Rice's theorem is a general result from 1953 that informally can be paraphrased as stating that all interesting questions about the behavior of programs are *undecidable*.

- Rice's theorem says that these "interesting" questions about program behavior are easily convertible to a halting problem. Consider:

```
x = 7;
if (halts(program p)) x = 8;
…
```

- Is x constant, or not?

However,

- The fact that interesting problems are undecidable **in general** does not mean that *all* their instances are undecidable
- The job of static analysis is to decide as many as possible, without producing wrong information

What to Analyze?

- Dead code elimination (DCE)
- Strength Reduction (CR)
- Loop Unrolling
- Function Inlining
- Constant Propagation (CP)
- Common Subexpression Elimination (CSE)
- Alias Analysis (AA)
- Reachability Analysis
- Global Value Numbering (GVN)
- Type Analysis

… and countless more

What to Analyze?

- **Dead code elimination** (DCE)
- **Strength Reduction** (CR)
- Loop Unrolling
- Function Inlining
- **Constant Propagation** (CP)
- Common Subexpression Elimination (CSE)
- Alias Analysis (AA)
- Reachability Analysis
- Global Value Numbering (GVN)
- **Type Analysis**

… and countless more

Dead Instruction Elimination

- A simple analysis
- “code that produces results which are never used does not have to be computed”

```
a = b + c
a = c * 2
```

Dead Instruction Elimination

- A simple analysis
- "code that produces results which are never used does not have to be computed"

```
~~a = b + c~~
a = c * 2
```

Dead Instruction Elimination

- A simple analysis
- "code that produces results which are never used does not have to be computed"

```
~~a = b + c~~
a = c * 2
```

- How about:

```
a = f(a, b)
a = 3
```

Dead Instruction Elimination

- A simple analysis
- “code that produces results which are never used does not have to be computed”

```
~~a = b + c~~
a = c * 2
```

- How about:

```
~~a = f(a, b)~~
a = 3
```

Dead Instruction Elimination

- A simple analysis
- "code that produces results which are never used does not have to be computed"

```
~~a = b + c~~
a = c * 2
```

- How about:

```
~~a = f(a, b)~~
a = 3
```

```
f = function(a, b) {
  print(a)
  print(b)
}
```

Dead Instruction Elimination

- A simple analysis
- “code that produces results which are never used does not have to be computed”

```
~~a = b + c~~
a = c * 2
```

- How about:

```
~~a = f(a, b)~~
a = 3
```

```
f = function(a, b) {
  print(a)
  print(b)
}
```

While f’s result is not used in the *analyzed* code, there is always a possibility it does something observable (sideeffects)

Constant Propagation

- any *pure* computation on constants will always return the same result, so we can “precompute” at compile-time

```
a = 3 + 5
b = a + 1
```

Constant Propagation

- any *pure* computation on constants will always return the same result, so we can "precompute" at compile-time

```
a = 3 + 5 8
b = a8 + 1 9
```

- How about examples that are not trivial?

Constant Propagation

- any *pure* computation on constants will always return the same result, so we can "precompute" at compile-time

```
a = ~~3 + 5~~ 8
b = ~~a8 + 1~~ 9
```

- How about examples that are not trivial?

```
a = 3, b = 1
while (b < 10) {
    if (b == 67)
        a = 7
    ++b
}
// is a constant here?
```

Precise x Conservative

- Analysis is said to be **precise** if it finds all possible answers
- Analysis is said to be **conservative** if all answers it finds are valid answers

- We already know that we can't be *precise* in general, but in compilers, we **must** be *conservative*
  - Non-conservative analysis might lead to semantics changing optimizations

  - Can you think of settings in which being non-conservative might still be acceptable?

Analysis Scope

- Local
  - Optimizes only within a single basic block
- Intra-procedural
  - Optimizes only within a single function, ignoring all other functions
- Inter-procedural (global, whole-program analysis)
  - Optimizes with knowledge of all functions in the program

Abstract Interpretation

MI-LCF

Motivation

- Remember the constant propagation example

```
a = 3, b = 1
while (b < 10) {
    if (b == 67)
        a = 7
    ++b
}
// is a constant here?
```

- This is not nearly as bad as the halting problem that makes it unsolvable
- But still, analyzing cases like this does not seem to be easy

Abstract Interpretation

ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

1. Introduction

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text $-1515 * 17$ may be understood to denote computations on the abstract universe $\{(+), (-), (\pm)\}$ where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution $-1515 * 17$ $\Longrightarrow -(+) * (+) \Longrightarrow (-) * (+) \Longrightarrow (-)$, proves that $-1515 * 17$ is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. $-1515 + 17 \Longrightarrow -(+) + (+) \Longrightarrow$

Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski [55]. The abstraction process is defined in section 6. It is shown that the program properties obtained by an abstract interpretation of a program are consistent with those obtained by a more refined interpretation of that program. In particular, an abstract interpretation may be shown to be consistent with the formal semantics of the language. Levels of abstraction are formalized by showing that consistent abstract interpretations form a lattice (section 7). Section 8 gives a constructive definition of abstract properties of programs based on constructive definitions of fixpoints. It shows that various classical algorithms such as Kildall [73], Wegbreit[75] compute program properties as limits of finite Kleene[52]'s sequences. Section

Abstract Interpretation

- The **concrete semantics** of a program models all of the possible execution paths the program may take
  - (depending on its input)
- As we have already said, this is undecidable

- **Abstract semantics** of a program is a superset of the concrete semantics
  - i.e. it covers *all* possible paths the program may take (sound)
  - + often some more (imprecise)
  - Unlike concrete semantics, is decidable

Abstract Interpretation

- Deciding about the abstract semantics means to interpret the program in the abstract semantics
  - We already know it is by definition *decidable*
- Required properties of abstract semantics
  - Sound (so that no *possible* errors will be reported)
  - Precise **enough** (so that no *impossible* errors are reported)
  - As simple (abstract) as possible so that it runs fast
- While this describes checking, each analysis can be thought of as checking some properties

Abstract Semantics

- Abstract values
  - Mapping from concrete values to abstract ones
  - Often multiple concrete values map to single abstract value
- Abstract operations
  - Each concrete operation must have its abstract counterpart defined that performs the same operation on abstract value
- Abstract state
  - Abstract state is represented in the same way as concrete state, only abstract values are used
  - (so for example abstract state is variable names mapped to abstract values)

Example – Simple CP

- Example programs consist only of +,-, == and integer variables
- We want to know which variables hold constants at which program points

Example – Simple CP

- Example programs consist only of +,-, == and integer variables
- We want to know which variables hold constants at which program points

```
NOT_CONST, IS_CONST
```

Example – Simple CP

- Example programs consist only of +,-, == and integer variables
- We want to know which variables hold constants at which program points

```
NOT_CONST, IS_CONST
```

- While this is woking abstract value, it does not provide the real information we need

Example – Simple CP

- Example programs consist only of +,-, == and integer variables
- We want to know which variables hold **what** constants at which program points

```
NOT_CONST, IS_CONST<k>
```

Example – Simple CP

- Example programs consist only of +,-, == and integer variables
- We want to know which variables hold **what** constants at which program points

```
NOT_CONST, IS_CONST<k>
```

- Abstract state is simple, mapping from variables to abstract values defined above
- Abstract operations are not too hard either

Example – Simple CP

X + Y	NO_CONST	IS_CONST <k>
NO_CONST	NO_CONST	NO_CONST
IS_CONST <l>	NO_CONST	IS_CONST <k+l>

Example – Simple CP

X * Y	NO_CONST	IS_CONST <k>	IS_CONST <0>
NO_CONST	NO_CONST	NO_CONST	IS_CONST <0>
IS_CONST <l>	NO_CONST	IS_CONST <k+l>	IS_CONST <0>
IS_CONST <0>	IS_CONST <0>	IS_CONST <0>	IS_CONST <0>

Example – Simple CP

```
a = 3, b = 1

while (b < 10) {

    if (b == 67)

        a = 7

    ++b

}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1

while (b < 10) {

    if (b == 67)

        a = 7

    ++b

}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) {

    if (b == 67)

        a = 7

    ++b

}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<1>}
    if (b == 67)

        a = 7

    ++b

}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<1>}
    if (b == 67) NOT TAKEN

        a = 7

    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<12>}
    if (b == 67) NOT TAKEN

        a = 7

    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<…9>}
    if (b == 67) NOT TAKEN

        a = 7

    ++b
    {a = IS_CONST<3>, b = IS_CONST<…10>}
}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN NOT TAKEN
    {a = IS_CONST<3>, b = IS_CONST<…9>}
    if (b == 67) NOT TAKEN

        a = 7

    ++b
    {a = IS_CONST<3>, b = IS_CONST<…10>}
}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN NOT TAKEN
    {a = IS_CONST<3>, b = IS_CONST<…9>}
    if (b == 67) NOT TAKEN

        a = 7

    ++b
    {a = IS_CONST<3>, b = IS_CONST<…10>}
}
{a = IS_CONST<3>, b = IS_CONST<10>}
// is a constant here? - YES!!
```

Example – Simple CP

- We have answered the question!

Example – Simple CP

- We have answered the question!
- But:
  - It took as much time as would be necessary to interpret the concrete program
  - Also, it does not seem quite robust, still a lot of black magic

Example – Simple CP

- We have answered the question!
- But:
  - It took **as much time** as would be necessary to interpret the concrete program
  - Also, it does not seem quite robust, still a lot of black magic

Example – Simple CP

- We have answered the question!
- But:
  - It took **as much time** as would be necessary to interpret the concrete program
  - Also, it does not seem quite robust, still a lot of black magic

```
a = 1
while (true) {
  …
}
// is a constant here?
```


WAIT, WHAT?!
WE DID THIS ALREADY!

Didn’t we say that abstract semantics is always decidable?

Termination

- Yes, we did, but you must *make* it always decidable
- The key question is **when** is it ok to terminate

- And there really aren't that much options
  - Let's terminate when the abstract state no longer changes

Termination

- Yes, we did, but you must *make* it always decidable
- The key question is **when** is it ok to terminate
- And there really aren't that much options
  - Let's terminate when the abstract state no longer changes (fixpoint)
- What if it never does?
  - We *must* make it
  - i.e. the abstract state must converge, and must do so in finite amount of time

Abstract Values Revisited

- Abstract values must form a **lattice**

Lattice is a partially ordered set wrt to operation <=, where for each two elements A, B, there exists:

- their supremum S such that A <= S and B <= S

- their infimum I such that I <= A and I <= B

- By extension there must be supremum and infimum of *all* lattice elements
  - Infimum of all will be called bottom (⊥)
  - Supremum of all elements will be called top (T)

Merging vs Updates

- So far we have only seen state updates
  - Deterministic, state after update is exactly as the update specifies

```
{a = ?}
a = 1
{a = IS_CONST<1>}
```

- Merging happens when "superposition" of states must be interpreted at the same time
  - When merging abstract values are replaced with their **supremums**

```
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) {
    …
    {a = IS_CONST<3>, b = IS_CONST<2>}
```

Merging vs Updates

```
{a = IS_CONST<3>,
 b = IS_CONST<1>}
```

```
{a = IS_CONST<3>,
 b = IS_CONST<2>}
```

```
{a = sup(IS_CONST<3>, IS_CONST<3>),
 b = sup(IS_CONST<1>, IS_CONST<2>)}
```

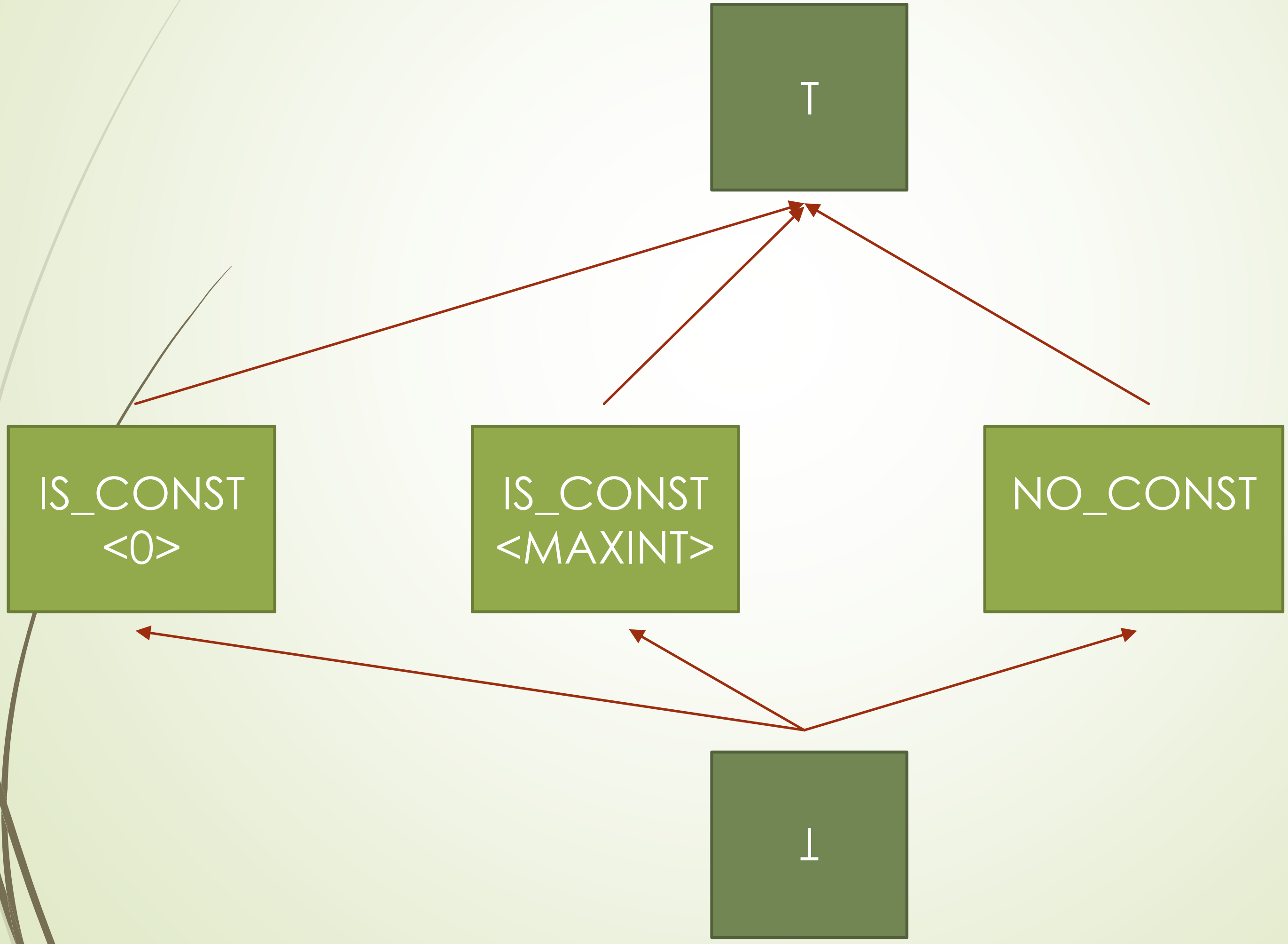
CP Revisited

CP Revisited

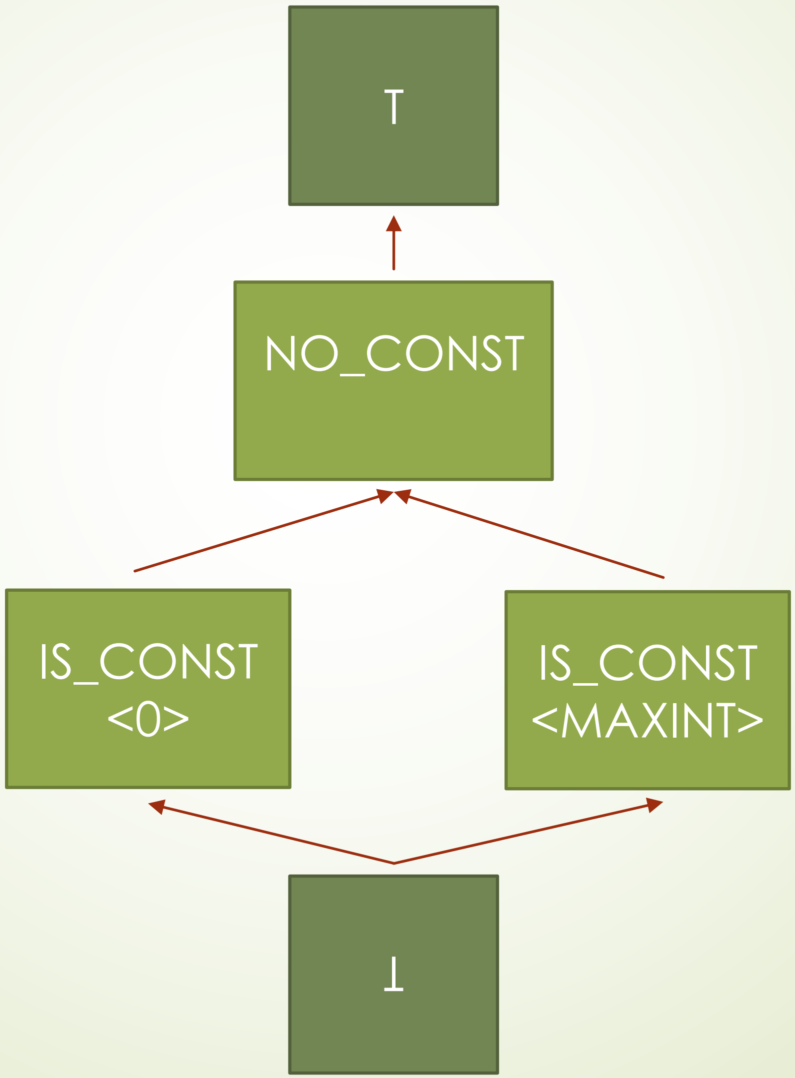
$\top$

$\bot$

CP Revisited



CP Revisited



Merging vs Updates

```
{a = IS_CONST<3>,
 b = IS_CONST<1>}
```

```
{a = IS_CONST<3>,
 b = IS_CONST<2>}
```

```
{a = IS_CONST<3>,
 b = NO_CONST}
```

Ok. Great.
Why does it not terminate now?

Ok. Great.
Why does it terminate now?

- First: if there are no loops it must always terminate
- Second: the merge operation moves always upwards in the lattice
  - After some merges it must reach T, T is supremum of *all* values, so no future merges can change the value
  - As long as height of the lattice is finite (!!)
- It is ok for the lattice to have infinite number of elements (as were the constants)
- It is the height of the lattice we want to minimize!

Let's try again!

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) {

    if (b == 67)

        a = 7

    ++b

}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<1>}
    if (b == 67) NOT TAKEN

        a = 7
    {a = IS_CONST<3>, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<1>}
    if (b == 67) NOT TAKEN

        a = 7
    {a = IS_CONST<3>, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Update

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<1>}
    if (b == 67) NOT TAKEN

        a = 7
    {a = IS_CONST<3>, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_CONST<1>}
    if (b == 67) NOT TAKEN

        a = 7
    {a = IS_CONST<3>, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Merge

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = NO_CONST}
    if (b == 67) NOT TAKEN

        a = 7
    {a = IS_CONST<3>, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Merge

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = NO_CONST}
    if (b == 67) NOT TAKEN TAKEN
        a = 7
        {a = IS_CONST<7>, b = NO_CONST}
    {a = IS_CONST<3>, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = NO_CONST}
    if (b == 67) NOT TAKEN TAKEN
        a = 7
        {a = IS_CONST<7>, b = NO_CONST}
    {a = IS_CONST<3>, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Merge!!!

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = NO_CONST}
    if (b == 67) NOT TAKEN TAKEN
        a = 7
        {a = IS_CONST<7>, b = NO_CONST}
    {a = NO_CONST, b = IS_CONST<1>}
    ++b
    {a = IS_CONST<3>, b = IS_CONST<2>}
}

// is a constant here?
```

Merge!!!

Example – Simple CP

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = NO_CONST, b = NO_CONST}
    if (b == 67) TAKEN
        a = 7
        {a = IS_CONST<7>, b = NO_CONST}
    {a = NO_CONST, b = NO_CONST}
    ++b
    {a = NO_CONST, b = NO_CONST}
}

// is a constant here?
```

Merge

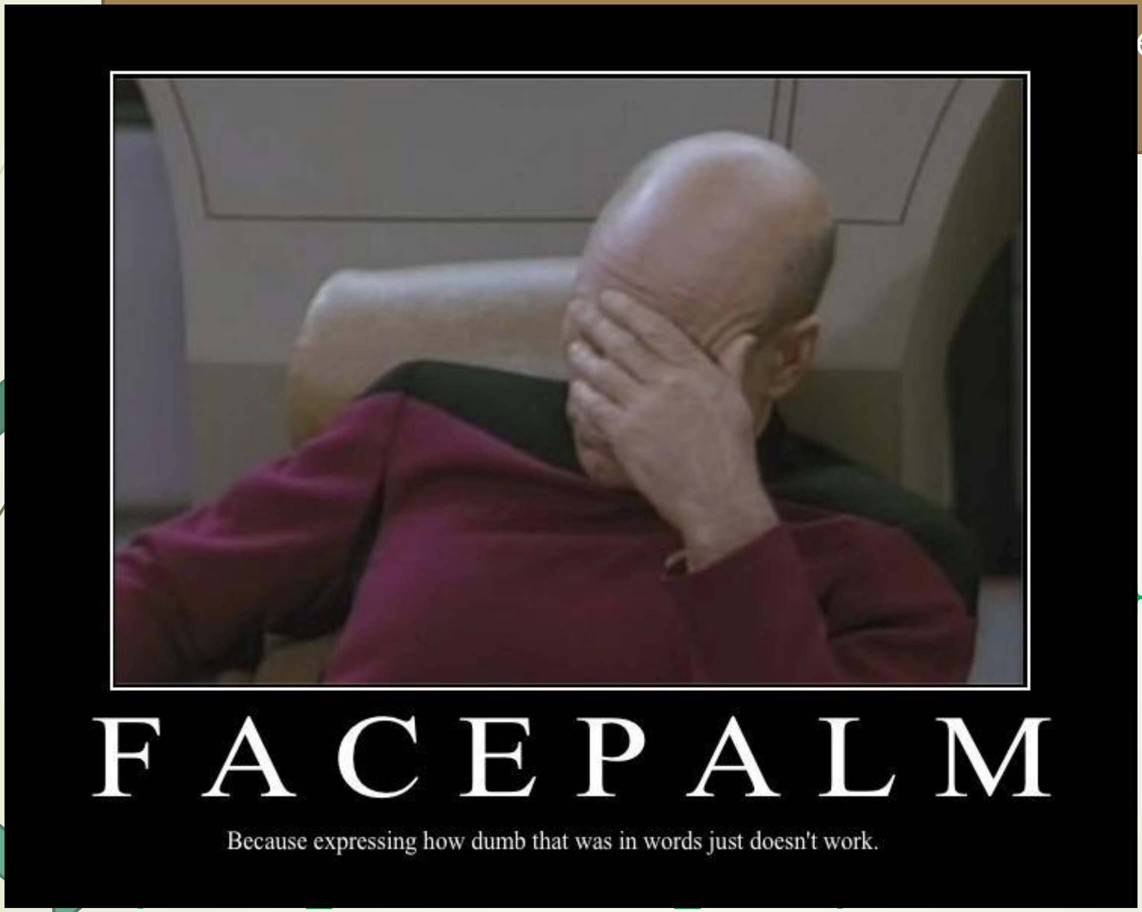
Example – Simple CP

after next iteration nothing changes, no need to do loop body again

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN NOT TAKEN
    {a = NO_CONST, b = NO_CONST}
    if (b == 67) TAKEN
        a = 7
        {a = IS_CONST<7>, b = NO_CONST}
    {a = NO_CONST, b = NO_CONST}
    ++b
    {a = NO_CONST, b = NO_CONST}
}
{a = NO_CONST, b = NO_CONST}
// is a constant here? NO!!!
```

Example – Simple CP

eed to do




```
// is a constant here? NO!!!!!
```

Let's Change the lattice

- Assume we introduce ranges:
  - IS_CONST<k>
  - IS_RANGE<a,b>
  - NO_CONST == effectively IS_RANGE<min,max>
- We must update the lattice merging for supremum:
  - sup(const<a>, const<b>) = range(min(a, b), max(a, b))
  - sup(const<c>, range<a,b>) = range(min(a, c), max(b, c))
  - Sup(range<a,b>, range<c,d>=range(min(a,c), mac(b,d))

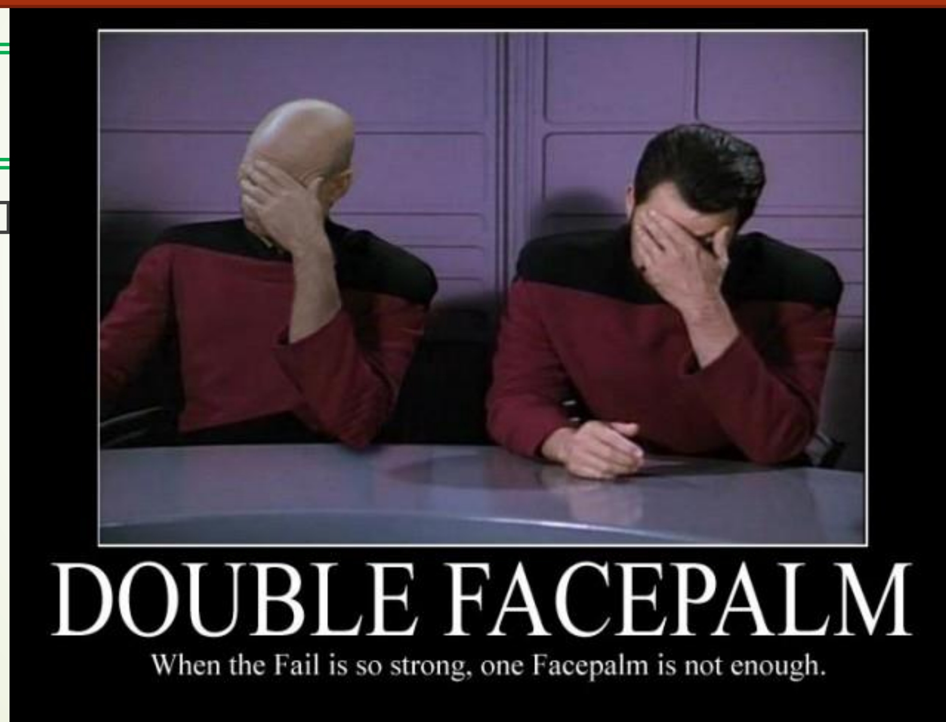
Example – Simple CP

when it terminates

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_RANGE<1,9>}
    if (b == 67) NOT TAKEN
        a = 7
        {a = IS_CONST<7>, b = IS_RANGE<1,9>}
    {a = IS_CONST<3>, b = IS_RANGE<1,9>}
    ++b
    {a = IS<CONST<3>, b = IS_RANGE<2,10>}
}
{a = IS_CONST<3>, b = IS_RANGE<2,10>}
// is a constant here? YES!!!
```

Example – Simple CP

But it took us as long as running the program…




```
{a =
a =
{a =
whil
                                        >}

                                        1,10>}
                                        >}

                                        >}
}
{a = IS_CONST<3>, b = IS_RANGE<2,11>}
// is a constant here? YES!!!
```

Example – Simple CP

But it took us as long as running the program…

```
{a = ?, b = ?}
a = 3, b = 1
{a = IS_CONST<3>, b = IS_CONST<1>}
while (b < 10) { TAKEN
    {a = IS_CONST<3>, b = IS_RANGE<1,10>}
    if (b == 67) NOT TAKEN
        a = 7
        {a = IS_CONST<7>, b = IS_RANGE<1,10>}
    {a = IS_CONST<3>, b = IS_RANGE<1,10>}
    ++b
    {a = IS<CONST<3>, b = IS_RANGE<2,11>}
}
{a = IS_CONST<3>, b = IS_RANGE<2,11>}
// is a constant here? YES!!!
```

Also:
Why
<2..10>?

Can we do better?

- Yes!
  - For instance, abstract semantics of x<y in the condition may set x's state to range<x,y-1>
  - This way we won't need to do all loop executions, while knowing that b is never 67 in the loop
- The design of abstract semantics is really important
  - If too specific, the running time will be horrible
  - If too general, not enough information will be captured
  - It is always a tradeoff


But wait!
There's more!
ICANHASCHEEZBURGER.COM

How to actually implement this?

- Remember basic blocks? They are handy:
- We only need to merge at the beginnings of basic blocks
  - Once first instruction in basic block gets executed, all must
- Keep a queue of basic blocks to analyze, start by pushing first basic block of the function in
- Have merge function on abstract state tell if the state changed in the merge, only schedule basic block if it does

Algorithm

0) Push first BB into the queue, input state of the first BB is initial state

1) While queue not empty take BB out of queue and its incoming state

2) Abstract interpret all instructions in the BB

we end up with abstract state *after* the terminating instruction of the BB

3) For each successor of current BB, do:

merge its incoming state with current state, if it changes, put the basic block in the queue (if it is not there already)

4) Goto 1)

There are variations…

- You can merge upon entering the basic block
- Not all basic blocks need their states to be remembered
  - Only those with more than one predecessor
- You do not need to remember entire state

- !! Note that unlike the algorithm we used in lectures, this one always takes all possible branches
  - This is fine, albeit often not necessary

Analyzing RIFT in LLVM

- Treat runtime functions and selected LLVM functions as operations
- Define abstract values, operations and state
- The state must track
  - Values in the environment
  - LLVM local variables (registers)
- More on this tomorrow…
- HW: Think about a lattice for type analysis in Rift

Q & A

OK, that's really it.

MI-LCF