Verifying LLVM Optimizations in Coq

Steve Zdancewic
Oregon PL Summer School 2013
Thanks To

- Dmitri Garbuzov
  - developed the Vminus & hands-on part of the lectures

- Jianzhou Zhao
  - developed the Vellvm Coq framework

- Santosh Nagarakatte
- Milo Martin

- Xavier Leroy
  - some of the slides are modeled after his
Motivation: SoftBound/CETS

[Nagarakatte, et al. PLDI ’09, ISMM ’10]

- Buffer overflow vulnerabilities.
- Detect spatial/temporal memory safety violations in legacy C code.
- Implemented as an LLVM pass.
- What about correctness?

http://www.cis.upenn.edu/acg/softbound/
Motivation: Compiler Bugs

[Yang et al. PLDI 2011]

Random test-case generation

Source Programs

GCC

LLVM

64open

\{8 other C compilers\}

79 bugs: 25 critical

202 bugs

325 bugs in total
Motivation: Semantics

Are these two C programs equivalent?

```c
int Sum = (N & (N % 2 ? 0 : ~0)
| ( ((N & 2)>>1) ^ (N & 1) ) );
```

```c
int Sum = 0;
for (int i = 1; i < N; ++i)
{
    Sum = Sum ^ i;
}
```

(Yes!)
Motivation: OPLSS

• Demonstrate some applications of techniques from the summer school:
  – Formal Modeling in Coq
  – Operational Semantics
  – Preservation & Progress-style safety proofs
  – Simulation arguments

• Introduction to LLVM IR
  – Potentially useful target for PL implementations
Low-level Virtual Machine (LLVM)

[Battner et al.]

- Began in 2002 as Chris Lattner’s Masters Thesis
- Has since evolved into an industrial-strength compiler intermediate language
  - open source
  - used widely in academia
  - used extensively by Apple
  - very active community
- Key features:
  - Simple design: one IR for many analyses/optimizations
  - Single Static Assignment
  - Typed IR
- See: http://llvm.org
LLVM Compiler Infrastructure

[LLVM Compiler Infrastructure]

Front Ends

LLVM

Typed SSA IR

Optimizations/Transformations

Analysis

Code Gen/Jit

C++

C

Objective-C

Python

Scala

ARM

MIPS Technologies

Intel

SPARC

PowerPC
LLVM Compiler Infrastructure
[Lattner et al.]
The Vellvm Project
[Zhao et al. POPL 2012, CPP 2012, PLDI 2013]

- Formal semantics
- Facilities for creating simulation proofs
- Implemented in Coq
- Extract passes for use with LLVM compiler
- Example: verified memory safety instrumentation

Typed SSA IR
Optimizations/Transformations
Analysis
Vellvm Framework

Vellvm

Type System and SSA
Operational Semantics
Syntax
Memory Model
Proof Techniques & Metatheory

Coq

Extractor

OCaml Bindings
Parser
Printer

LLVM

C Source Code
LLVM IR
Transform
LLVM IR
Other Optimizations
Target
Vellvm Framework

Type System and SSA | Operational Semantics
---|---
Syntax | Memory Model
Proof Techniques & Metatheory

Coq

Extract

Verified Transform

OCaml Bindings

Parser

Printer

LLVM

C Source Code

LLVM IR

LLVM IR

Other Optimizations

Target
Plan

• Vminus: a highly simplified SSA IR based on LLVM
  – What is SSA?
• Verified Compilation of Imp to Vminus
  – What does it mean to “verify compilation”?
• Scaling up: Vellvm
  – Taste of the full LLVM IR
  – Operational Semantics
  – Metatheory + Proof Techniques
• Case studies:
  – SoftBound memory safety
  – mem2reg
• Conclusion:
  – challenges & research directions
define i32 @factorial(i32 %n) nounwind uwtable ssp {
entry:
  %1 = alloca i32, align 4
  %acc = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %acc, align 4
  br label %start

start: ; preds = %entry, %else
  %3 = load i32* %1, align 4
  %4 = icmp ugt i32 %3, 0
  br i1 %4, label %then, label %else

then: ; preds = %start
  %6 = load i32* %acc, align 4
  %7 = load i32* %1, align 4
  %8 = mul i32 %6, %7
  store i32 %8, i32* %acc, align 4
  %9 = load i32* %1, align 4
  %10 = sub i32 %9, 1
  store i32 %10, i32* %1, align 4
  br label %start

else: ; preds = %start
  %12 = load i32* %acc, align 4
  ret i32 %12
}
Distilling the LLVM

Documentation for January 2012 LLVM System Archives by thread

Messages sorted by: [subject] [author] [date]
More info on this list...

Starting: Sun Jan 1 12:44:27 CST 2012
Messages: 348

[LLVMdev] [PATCH] TLS support for Windows 32+64bit Kai
  [LLVMdev] [PATCH] TLS support for Windows 32+64bit Eli Friedman
  [LLVMdev] [PATCH] TLS support for Windows 32+64bit Kai
  [LLVMdev] [PATCH] TLS support for Windows 32+64bit Kai

[LLVMdev] _thaq_ Jiazhao Zhao
  [LLVMdev] Checking validity of metadata in an .ll file Seb
  [LLVMdev] Checking validity of metadata in an .ll file Devang Patel
  [LLVMdev] Checking validity of metadata in an .ll file Talin

[LLVMdev] Using llvm command line functions from within a plugin? Duncan Sands
  [LLVMdev] Using llvm command line functions from within a plugin? Talin

[LLVMdev] Comparison of Alias Analysis in LLVM Jiazhao Zhao
  [LLVMdev] Comparison of Alias Analysis in LLVM Chris Lattner
  [LLVMdev] Comparison of Alias Analysis in LLVM Jiazhao Zhao
  [LLVMdev] Comparison of Alias Analysis in LLVM Chris Lattner
  [LLVMdev] Comparison of Alias Analysis in LLVM Jiazhao Zhao

The LLVM Getting Started infrastructure. Everything from unpacking...
Distilling the LLVM
LLVM IR ⇒ Vminus

- Vastly Simplify! (For now...)

- Throw out:
  - types, complex & structured data
  - local storage allocation, complex pointers
  - functions
  - undefined values & nondeterminism

- What’s left?
  - basic arithmetic
  - control flow
  - global, preallocated state (a la Imp)
Vminus by Example

entry:

Control-flow Graphs:
+ Labeled blocks

loop:

exit:
Vminus by Example

**entry:**
\[ r_0 = \ldots \]
\[ r_1 = \ldots \]
\[ r_2 = \ldots \]

**Control-flow Graphs:**
+ Labeled blocks
+ Binary Operations

**loop:**
\[ r_3 = \ldots \]
\[ r_4 = r_1 \times r_2 \]
\[ r_5 = r_3 + r_4 \]
\[ r_6 = r_5 \geq 100 \]

**exit:**
\[ r_7 = \ldots \]
\[ r_8 = r_1 \times r_2 \]
\[ r_9 = r_7 + r_8 \]
Vminus by Example

**Control-flow Graphs:**
- Labeled blocks
- Binary Operations
- Branches/Return

**entry:**
\[
\begin{align*}
  r_0 & = \ldots \\
  r_1 & = \ldots \\
  r_2 & = \ldots \\
\end{align*}
\]

\textbf{br} \ r_0 \ \textbf{loop exit}

**loop:**
\[
\begin{align*}
  r_3 & = \ldots \\
  r_4 & = r_1 \times r_2 \\
  r_5 & = r_3 + r_4 \\
  r_6 & = r_5 \geq 100 \\
\end{align*}
\]

\textbf{br} \ r_6 \ \textbf{loop exit}

**exit:**
\[
\begin{align*}
  r_7 & = \ldots \\
  r_8 & = r_1 \times r_2 \\
  r_9 & = r_7 + r_8 \\
  \textbf{ret} \ r_9
\end{align*}
\]
Vminus by Example

Control-flow Graphs:
+ Labeled blocks
+ Binary Operations
+ Branches/Return
+ Static Single Assignment

(each *local identifier* assigned only *once*, statically)

local identifier a.k.a. uid or SSA variable
Vminus by Example

Control-flow Graphs:
+ Labeled blocks
+ Binary Operations
+ Branches/Return
+ Static Single Assignment
+ $\phi$ nodes

entry:
\[
\begin{align*}
  r_0 &= \ldots \\
  r_1 &= \ldots \\
  r_2 &= \ldots \\
  \text{br } r_0 \ \text{loop exit}
\end{align*}
\]

loop:
\[
\begin{align*}
  r_3 &= \phi[0;\text{entry}][r_5;\text{loop}] \\
  r_4 &= r_1 \times r_2 \\
  r_5 &= r_3 + r_4 \\
  r_6 &= r_5 \geq 100 \\
  \text{br } r_6 \ \text{loop exit}
\end{align*}
\]

exit:
\[
\begin{align*}
  r_7 &= \phi[0;\text{entry}][r_5;\text{loop}] \\
  r_8 &= r_1 \times r_2 \\
  r_9 &= r_7 + r_8 \\
  \text{ret } r_9
\end{align*}
\]
Vminus by Example

Control-flow Graphs:
+ Labeled blocks
+ Binary Operations
+ Branches/Return
+ Static Single Assignment
+ $\phi$ nodes

(choose values based on predecessor blocks)
Static Single Assignment (SSA)

• Compiler intermediate representation developed in the late 1980’s early 1990’s:
  – Global Value Numbers and Redundant Computations [Rosen, Wegman, Zadeck 1988]
  – An Efficient Method of Computing Static Single Assignment Form [Cytron, Ferrante, +RWZ, 1989]
  – Efficiently Computing Static Single Assignment Form and the Control Dependence Graph [Cytron, et. al, TOPLAS 1991]

• Makes optimizing imperative programming languages clean and efficient.
  – Used in gcc, clang, intel, Jikes, HotSpot, Open64, ...
SSA IR’s in Practice

• SSA simplifies register allocation:
  – The left-hand sides of SSA assignments can be thought of as “registers”
  – Renaming corresponds to “live range splitting” (decouples false dependencies)
  – register allocation is (arguably) the most important optimization for performance on modern processors
Critical Optimization in LLVM

O1 speeds up the program by 101%. mem2reg speeds it up by 81%
SSA Construction by Example

I := 0;;
J := 0;;
WHILE J < 100 DO
  IF I < 10 THEN
    I := I + 1;;
    J := J + I
  ELSE
    I := I + 2;;
    J := J + 1;
  FI
END;;
RETURN J
SSA Construction by Example

Step 1: Convert to a control-flow graph.

I := 0;;
J := 0;;
WHILE J < 100 DO
  IF I < 10 THEN
    I := I + 1;;
    J := J + I
  ELSE
    I := I + 2;;
    J := J + 1;
  FI
END;;
RETURN J
Step 2: Rename variables to satisfy single assignment.
SSA Construction by Example

I := 0;;
J := 0;;
WHILE J < 100 DO
  IF I < 10 THEN
    I := I + 1;;
    J := J + I
  ELSE
    I := I + 2;;
    J := J + 1;
  FI
END;;
RETURN J

Step 3: Insert “ϕ” functions that capture control dependence.
SSA IR’s in Practice (2)

• SSA yields an efficient representation
  – Simplifies Def-Use information needed in dataflow analysis
  – Imperative data structure to map a definition to its uses

• However: Real SSA IRs still retain mutable state
  – SSA uid’s don’t have addresses...
  – memory operations: explicit pointer manipulation, allocation
  – example (in C):

    ```c
    int foo() {
      int x;
      init(&x);    // pointer escapes
      return x;
    }
    ```

  – suggests the idea of “promoting” some imperative variables to SSA-style (those whose addresses don’t “escape”)


Vminus.Vminus.v

Up to the CFG module
Vminus Operational Semantics

• Only 5 kinds of instructions:
  – Binary arithmetic
  – Memory Load
  – Memory Store
  – Terminators
  – Phi nodes

• What is the state of a Vminus program?
Subtlety of Phi Nodes

- Phi-Nodes admit “cyclic” dependencies:

\[
\begin{align*}
\text{pred:} & \\
& \cdots \\
& \text{br loop}
\end{align*}
\]

\[
\begin{align*}
\text{loop:} & \\
& \%x = \phi[0;\text{pred}][y;\text{loop}] \\
& \%y = \phi[1;\text{pred}][x;\text{loop}] \\
& \%b = \%x \leq \%y \\
& \text{br } \%b \text{ loop exit}
\end{align*}
\]
Semantics of Phi Nodes

• The value of the RHS of a phi-defined uid is relative to the state at the entry to the block.

• Option 1:
  – Require all phi nodes to be at the beginning of the block
  – Execute them “atomically, in parallel”
  – (Original Vellvm followed this model)

• Option 2:
  – Keep track of the state upon entry to the block
  – Calculate the RHS of phi nodes relative to the entry state
  – (Vminus follows this model)
Vminus.Vminus.v

Opsem module
End of Part I
Recap

• Yesterday:
  – Defined a simple language called Vminus.
  – Five types of instructions:
    • binary arithmetic / load / store / phi nodes / terminators
  – Static Single Assignment
  – Operational semantics
    • Small step, relational

• Today: Static Semantics for Vminus
  – Scoping for SSA variables
Key SSA Invariant

**entry:**
- \( r_0 = \ldots \)
- \( r_1 = \ldots \)
- \( r_2 = \ldots \)

\[ \text{br } r_0 \text{ loop exit} \]

**loop:**
- \( r_3 = \phi[0;\text{entry}][r_3;\text{loop}] \)
- \( r_4 = r_1 \times r_2 \)
- \( r_5 = r_3 + r_4 \)
- \( r_6 = r_5 \geq 100 \)
- \[ \text{br } r_6 \text{ loop exit} \]

**exit:**
- \( r_7 = \phi[0;\text{entry}][r_5;\text{loop}] \)
- \( r_8 = r_1 \times r_2 \)
- \( r_9 = r_7 + r_8 \)
- \[ \text{ret } r_9 \]

---

**Definition of \( r_2 \).**

**Uses of \( r_2 \).**
Key SSA Invariant

entry:
\[
\begin{align*}
  r_0 &= \ldots \\
  r_1 &= \ldots \\
  r_2 &= \ldots 
\end{align*}
\]

br r_0 loop exit

loop:
\[
\begin{align*}
  r_3 &= \phi[0;\text{entry}][r_3;\text{loop}] \\
  r_4 &= r_1 \times r_2 \\
  r_5 &= r_3 + r_4 \\
  r_6 &= r_5 \geq 100 \\
  \text{br} r_6 \text{ loop exit}
\end{align*}
\]

exit:
\[
\begin{align*}
  r_7 &= \phi[0;\text{entry}][r_5;\text{loop}] \\
  r_8 &= r_1 \times r_2 \\
  r_9 &= r_7 + r_8 \\
  \text{ret} r_9
\end{align*}
\]

Definition of \( r_2 \).

Uses of \( r_2 \).

The definition of a variable must dominate its uses.
Defining SSA Variable Scope

**Graph**: g corresponds to a “fine grained” CFG

**Nodes**: program points (maybe more than one per block)

**Edges**: “fallthroughs”, jump and branch instructions

**Distinguished entry**
Paths

- Paths:
  \[ \text{Path } g \ a \ d \ [a;b;d] \]
Reachability

- Paths:
  Path g a d [a;b;d]
- Reachability:
  Reachable g x

iff

∃ vs. Path g e x vs

Reachable

Unreachable
Domination

- **Paths:**
  
  Path: g → a → d → [a;b;d]

- **Reachability:**
  
  Reachable: g → x

- **Domination:**
  
  Dom: g → b → c

iff every path from e to c goes through b.
Domination

- **Paths:**
  
  Path: \( a \rightarrow b \rightarrow d \)

- **Reachability:**
  
  Reachable: \( g \rightarrow x \)

- **Domination:**
  
  Domination: \( g \rightarrow b \rightarrow c \)

iff every path from \( e \) to \( c \) goes through \( b \).
Domination

- Paths:
  Path \( g \cdot a \cdot d \) \([a;b;d]\)
- Reachability:
  Reachable \( g \cdot x \)
- Domination:
  \( \text{Dom} \ g \cdot b \cdot c \)

iff every path from \( e \) to \( c \) goes through \( b \).
Domination

- Paths:
  - Path: g a d [a;b;d]
- Reachability:
  - Reachable: g x
- Domination:
  - Dom: g b c

Nodes dominated by b.
Strict Domination

- Paths:
  Path \( g \rightarrow a \rightarrow d \) \( [a;b;d] \)
- Reachability:
  Reachable \( g \rightarrow x \)
- Domination:
  \( \text{Dom} \ g \rightarrow b \rightarrow c \)
- **Strict Domination:**
  \( \text{SDom} \ g \rightarrow b \rightarrow c \)

Nodes strictly dominated by \( b \).
Domination Tree

• Order the reachable nodes by (immediate) dominators, and you get a tree:

Entry

• This is an inductive data structure (unlike CFG) ⇒ better for certain proofs. (e.g. those that have to do with scoping).
Vminus.Dom.v

Coq
Dominator Algorithm Tradeoffs

- **Cooper-Harvey-Kennedy (CHK)**
  - Extended from AC
  - Nearly as fast as LT in common cases

- **Lengauer-Tarjan (LT) (LLVM and GCC)**
  - Based on tricky graph theory
  - $O(E \times \log(N))$

- **Allen-Cocke (AC)**
  - Based on Kildall's algorithm
  - Large asymptotic complexity

“Although proving its correctness and verifying its running time require rather complicated analysis, the algorithm is quite simple to program...” [LT’79]
Dominator Algorithm Tradeoffs

Cooper-Harvey-Kennedy (CHK)
- ✓ Extended from AC
- ✓ Nearly as fast as LT in common cases

Allen-Cocke (AC)
- ✓ Based on Kildall’s algorithm
- ✗ Large asymptotic complexity

Lengauer-Tarjan (LT) (LLVM and GCC)
- ✗ Based on tricky graph theory
- ✓ O(E x log(N))

Vellvm implements both.
Safety Properties

• A well-formed program never accesses undefined variables.

\[
\text{If } \vdash f \text{ and } f \vdash \sigma_0 \rightarrow^* \sigma \text{ then } \sigma \text{ is not stuck.}
\]

$\vdash f$ program $f$ is well formed
$
\sigma$
program state
$f \vdash \sigma \rightarrow^* \sigma$
evaluation of $f$

• \textit{Initialization}:

\[
\text{If } \vdash f \text{ then } wf(f, \sigma_0).
\]

• \textit{Preservation}:

\[
\text{If } \vdash f \text{ and } f \vdash \sigma \rightarrow \sigma' \text{ and } wf(f, \sigma) \text{ then } wf(f, \sigma')
\]

• \textit{Progress}:

\[
\text{If } \vdash f \text{ and } wf(f, \sigma) \text{ then } f \vdash \sigma \rightarrow \sigma'
\]
Safety Properties

- A well-formed program never accesses undefined variables.

\[ \text{If } \vdash f \text{ and } f \vdash \sigma_0 \xrightarrow{\ast} \sigma \text{ then } \sigma \text{ is not stuck.} \]

- **Initialization:**
  
  \[ \text{If } \vdash f \text{ then } \text{wf}(f, \sigma_0) \]

- **Preservation:**
  
  \[ \text{If } \vdash f \text{ and } f \vdash \sigma \xrightarrow{\ast} \sigma' \text{ and } \text{wf}(f, \sigma) \text{ then } \text{wf}(f, \sigma') \]

- **Progress:**
  
  \[ \text{If } \vdash f \text{ and } \text{wf}(f, \sigma) \text{ then } \text{done}(f, \sigma) \text{ or } \text{stuck}(f, \sigma) \text{ or } f \vdash \sigma \xrightarrow{\ast} \sigma' \]
Well-formed States

entry:
\[ r_0 = \ldots \]
\[ r_1 = \ldots \]
\[ r_2 = \ldots \]
\[ \text{br } r_0 \text{ loop exit} \]

loop:
\[ r_3 = \phi[0;\text{entry}][r_5;\text{loop}] \]
\[ r_4 = r_1 \times r_2 \]
\[ r_5 = r_3 + r_4 \]
\[ r_6 = r_5 \geq 100 \]
\[ \text{br } r_6 \text{ loop exit} \]

exit:
\[ r_7 = \phi[0;\text{entry}][r_5;\text{loop}] \]
\[ r_8 = r_1 \times r_2 \]
\[ r_9 = r_7 + r_8 \]
\[ \text{ret } r_9 \]

State \( \sigma \) is:
\[ \text{pc} = \text{program counter} \]
\[ \delta = \text{local values} \]
Well-formed States (Roughly)

entry:

- \( r_0 = \ldots \)
- \( r_1 = \ldots \)
- \( r_2 = \ldots \)
- \( \text{br } r_0 \text{ loop exit} \)

State \( \sigma \) is:

- \( \text{pc} = \text{program counter} \)
- \( \delta = \text{local values} \)

- \( \text{sdom}(f, \text{pc}) = \text{variable defns. that strictly dominate } \text{pc} \).

loop:

- \( r_3 = \phi[0; \text{entry}][r_5; \text{loop}] \)
- \( r_4 = r_1 \times r_2 \)
- \( r_5 = r_3 + r_4 \)
- \( r_6 = r_5 \geq 100 \)
- \( \text{br } r_6 \text{ loop exit} \)

exit:

- \( r_7 = \phi[0; \text{entry}][r_5; \text{loop}] \)
- \( r_8 = r_1 \times r_2 \)
- \( r_9 = r_7 + r_8 \)
- \( \text{ret } r_9 \)
Well-formed States (Roughly)

**State** \( \sigma \) contains:
- \( \text{pc} = \) program counter
- \( \delta = \) local values

\( \text{sdom}(f, \text{pc}) = \) variable defns. that *strictly dominate* \( \text{pc} \).

\[ \text{wf}(f, \sigma) = \forall r \in \text{sdom}(f, \text{pc}). \exists v. \delta(r) = \lfloor v \rfloor \]

“All variables in scope are initialized.”
Vminus.Vminus.v

Typing
Compiler Verification

• 1967: Correctness of a Compiler for Arithmetic Expressions [McCarthy, Painter]
• 1972: Proving Compiler Correctness in a Mechanized Logic [Milner, Weyhrauch]
• ... many interesting developments

See: Compiler Verification, A Bibliography [Dave, 2003]

• 2006-present: CompCert [Leroy, et al.]
  – (Nearly!) fully verified compiler from C to Power PC, ARM, etc.
  – Randomized compiler testing found no bugs (in the verified components – the original, unverified parser had a bug)
Vminus.Imp.v

Coq
Execution Models

• Interpretation:
  – program represented by abstract syntax
  – tree traversed by interpreter

•Compilation to native code:
  – program translated to machine instructions
  – executed by hardware

•Compilation to virtual machine code:
  – program translated to “virtual machine” instructions
  – interpreted (efficiently)
  – further translated to machine code
  – just-in-time compiled to machine code
Correct Execution?

• What does it mean for an Imp program to be executed correctly?

• Even at the interpreter level we could show *equivalence* between the small-step and the large-step operational semantics:

\[
\text{cmd} / \text{st} \leftrightarrow \overset{*}{\longrightarrow} \text{SKIP} / \text{st}'
\]

iff

\[
\text{cmd} / \text{st} \downarrow \text{st}'
\]
Compiler Correctness?

• We have to relate the source and target language semantics across the compilation function $\mathcal{C}[-] : \text{source} \rightarrow \text{target}$.

  \[
  \text{cmd / st} \quad \xleftarrow{s} \quad * \quad \text{SKIP / st'}
  \]

  iff

  \[
  \mathcal{C}[\text{cmd}] / \mathcal{C}[\text{st}] \quad \xleftarrow{t} \quad * \quad \mathcal{C}[\text{st'}]
  \]

• Is this enough?
• What if cmd goes into an infinite loop?
Comparing Behaviors

• Consider two programs P and P’ possibly in different languages.
  – e.g. P is an Imp program, P’ is its compilation to Vminus

• The semantics of the languages associate to each program a set of observable behaviors:

  \[ \mathcal{B}(P) \text{ and } \mathcal{B}(P') \]

• Note: \[ |\mathcal{B}(P)| = 1 \text{ if } P \text{ is deterministic, } > 1 \text{ otherwise} \]
What is Observable?

- For Imp-like languages:

  observable behavior ::=  
  | terminates(st)    (i.e. observe the final state)  
  | diverges          
  | goeswrong

- For pure functional languages:

  observable behavior ::=  
  | terminates(v)     (i.e. observe the final value)  
  | diverges          
  | goeswrong
What about I/O?

- Add a trace of input-output events performed:

  \[
  t ::= [] \mid e :: t \quad \text{(finite traces)} \\
  \text{coind. } T ::= [] \mid e :: T \quad \text{(finite and infinite traces)}
  \]

  observable behavior ::= 
  \[
  | \text{terminates}(t, st) \quad \text{(end in state st after trace t)} \\
  | \text{diverges}(T) \quad \text{(loop, producing trace T)} \\
  | \text{goeswrong}(t)
  \]
Examples

• P1:
  print(1); /st ⇒ terminates(out(1)::[],st)

• P2:
  print(1); print(2); /st
  ⇒ terminates(out(1)::out(2)::[],st)

• P3:
  WHILE true DO print(1) END /st
  ⇒ diverges(out(1)::out(1)::...)

• So  $B(P1) \neq B(P2) \neq B(P3)$
Bisimulation

- Two programs P1 and P2 are bisimilar whenever:

\[ \mathcal{B}(P1) = \mathcal{B}(P2) \]

- The two programs are completely indistinguishable.

- But... this is often too strong in practice.
Compilation Reduces Nondeterminism

• Some languages (like C) have underspecified behaviors:
  – Example: order of evaluation of expressions $f() + g()$

• Concurrent programs often permit nondeterminism
  – Classic optimizations can reduce this nondeterminism
  – Example:
    
    ```
    a := x + 1; b := x + 1  ||  x := x+1
    ```

    vs.

    ```
    a := x + 1; b := a  ||  x := x+1
    ```

• As we’ll see, LLVM explicitly allows nondeterminism.
Backward Simulation

• Program P2 can exhibit fewer behaviors than P1:

\[ \mathcal{B}(P1) \supseteq \mathcal{B}(P2) \]

• All of the behaviors of P2 are permitted by P1, though some of them may have been eliminated.
• Also called refinement.
What about goes wrong?

- Compilers often translate away bad behaviors.

\[
x := \frac{1}{y} ; x := 42 \quad \text{vs.} \quad x := 42
\]

(divide by 0 error) \hspace{1cm} (always terminates)

- Justifications:
  - Compiled program does not “go wrong” because the program type checks or is otherwise formally verified
  - Or just “garbage in/garbage out”
Safe Backwards Simulation

- Only require the compiled program’s behaviors to agree if the source program could not go wrong:

\[
goes\text{wrong}(t) \notin \mathcal{B}(P1) \implies \mathcal{B}(P1) \supseteq \mathcal{B}(P2)
\]

- Idea: let \( S \) be the functional specification of the program:
  A set of behaviors not containing \( \text{goes\text{wrong}}(t) \).
  - A program \( P \) satisfies the spec if \( \mathcal{B}(P) \subseteq S \)

- Lemma: If \( P2 \) is a safe backwards simulation of \( P1 \) and \( P1 \) satisfies the spec, then \( P2 \) does too.
Building Backward Simulations

Source:

\[ \mathcal{C}[-] \]

Target:

\[ \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \cdots \]

\[ \mathcal{C}[-] \]

**Idea:** The event trace along a (target) sequence of steps originating from a compiled program must correspond to some source sequence.

**Tricky parts:**

- Must consider all possible target steps
- If the compiler uses many target steps for once source step, we have invent some way of relating the intermediate states to the source.
- the compilation function goes the wrong way to help!
End of Part 2
Safe Backwards Simulation

• Only require the compiled program’s behaviors to agree if the source program could not go wrong:

\[
goes\text{wrong}(t) \notin \mathcal{B}(P1) \implies \mathcal{B}(P1) \supseteq \mathcal{B}(P2)
\]

• Idea: let \( S \) be the functional specification of the program:
A set of behaviors not containing \( \text{goes\text{wrong}}(t) \).
  – A program \( P \) satisfies the spec if \( \mathcal{B}(P) \subseteq S \)

• Lemma: If \( P2 \) is a safe backwards simulation of \( P1 \) and \( P1 \) satisfies the spec, then \( P2 \) does too.
Safe Forwards Simulation

• Source program’s behaviors are a subset of the target’s:

\[
\text{goeswrong}(t) \notin \mathcal{B}(P1) \implies \mathcal{B}(P1) \subseteq \mathcal{B}(P2)
\]

• P2 captures all the good behaviors of P1, but could exhibit more (possibly bad) behaviors.

• But: Forward simulation is significantly easier to prove:
  – Only need to show the existence of a compatible target trace.
Determinism!

- Lemma: If P2 is deterministic then forward simulation implies backward simulation.

- Proof: \( \emptyset \subseteq B(P1) \subseteq B(P2) = \{b\} \) so \( B(P1) = \{b\} \).

- Corollary: safe forward simulation implies safe backward simulation if P2 is deterministic.
Forward Simulations

Idea: Show that every transition in the source program:
- is simulated by some sequence of transitions in the target
- while preserving a relation $\sim$ between the states
Imp: A Refresher

id := X | Y | Z | ...  \hspace{1cm} Variables

aexp := n | id | aexp + aexp | aexp - aexp | aexp * aexp \hspace{1cm} Arithmetic Expressions

bexp := true | false | aexp = aexp | !bexp | bexp && bexp \hspace{1cm} Boolean Expressions

cmd :=
    | SKIP \hspace{1cm} Do nothing
    | id ::= aexp \hspace{1cm} Assignment
    | cmd ;; cmd \hspace{1cm} Sequence
    | IFB bexp THEN cmd ELSE cmd FI \hspace{1cm} Conditional
    | WHILE bexp DO cmd END \hspace{1cm} Loop

See Vminus/Imp.v for the Coq formalism
Vminus.ComplImmp.v

Coq
A single source-program step is simulated by a single target step.

(Solid = assumptions, Dashed = must be shown)
“Plus”-step Forward Simulation

Source: \[ \sigma_1 \rightarrow \sigma_2 \]

Target: \[ C[\sigma_1] \rightarrow \tau_0 \rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \]

A single source-program step is simulated by \textit{one or more} target steps. (But only finitely many!)

(Solid = assumptions, Dashed = must be shown)
A single source-program step is simulated by zero steps in the target.
Problem with “Infinite Stuttering”

An infinite sequence of source transitions can be “simulated” by 0 transitions in the target!

(This simulation doesn’t preserve nontermination.)
Solution: Disallow such “trivial” simulations

Equip the source language with a measure $|\sigma|$ and require that $|\sigma_2| < |\sigma_1|$.

The measure can’t decrease indefinitely, so the target program must either take a step or the source must terminate.

The target diverges if the source program does.
Vminus.CompilImp.v

Coq
Is Backward Simulation Hopeless?

• Suppose the source & target languages are the same.
  – So they share the same definition of program state.
• Further suppose that the steps are very “small”.
  – Abstract machine (i.e. no “complex” instructions).
• Further suppose that “compilation” is only a very minor change.
  – add or remove a single instruction
  – substitute a value for a variable

• Then: backward simulation is more achievable
  – it’s easier to invent the “decompilation” function because the
    “compilation” function is close to trivial

• Happily: This is the situation for LLVM optimizations
Lock-Step Backward Simulation

\[ S_1 \xrightarrow{o} S_2 \]

\[ T_1 \xrightarrow{o} T_2 \]

\( o \) is either an “observable event” or a “silent event”

\( o ::= e \mid \varepsilon \)

Example use: proving variable substitution correct.
Right-Option Backward Simulation

• Either:
  – the source and target are in lock-step simulation.

Or
  – the source takes a silent transition to a smaller state

Example use: removing an instruction in the target.
Right-Option Backward Simulation

- Either:
  - the source and target are in lock-step simulation.
  - the source takes a silent transition to a smaller state

Example use: removing an instruction in the target.
Left-Option Backward Simulation

- Either:
  - the source and target are in lock-step simulation.
  - the target takes a silent transition to a smaller state

Example use: adding an instruction to the target.
Generalizing Safety

• Definition of \( wf \):

\[
wf(f,(pc, \delta)) = \forall r \in sdom(f,pc). \exists v. \delta(r) = [v]
\]

• Generalize like this:

\[
wf(f,(pc, \delta)) = P f (\delta \mid_{sdom(f,pc)})
\]

where \( P : \text{Program} \rightarrow \text{Locals} \rightarrow \text{Prop} \)

• Methodology: for a given \( P \) prove the

  * Initialization(\( P \))
  * Preservation(\( P \))
  * Progress(\( P \))

Consider only variables in scope \( \Rightarrow P \) defined relative to the dominator tree of the CFG.
Instantiating

- For usual safety:
  \[ P_{\text{safety}} \ f \ \delta \ = \ \forall r \in \text{dom}(\delta). \ \exists v. \ \delta(r) = [v] \]

- For semantic properties:
  \[ P_{\text{sem}} \ f \ \delta \ = \ \forall r. \ f[r] = [\text{rhs}] \implies \delta(r) = [\text{rhs}^\delta] \]

- Useful for creating the simulation relation for correctness of:
  - code motion, dead variable elimination, common expression elimination, etc.
End of Part 3
Strategy for Proving Optimizations

• Decompose the program transformation into a sequence of “micro” transformations
  – e.g. code motion =
    1. insert “redundant” instruction
    2. substitute equivalent definitions
    3. remove the “dead” instruction

• Use the backward simulations to show each “micro” transformation correct.
  – Often uses a generalization of the Vminus safety property

• Compose the individual proofs of correctness
mem2reg in LLVM

- Promote stack allocas to temporaries
- Insert minimal φ-nodes

Front-ends w/o SSA construction → The LLVM IR w/o φ-nodes → mem2reg → The LLVM IR in the minimal SSA form → Backends

• imperative variables ⇒ stack allocas
• no φ-nodes
• trivially in SSA form
int x = 0;
if (y > 0)
  x = 1;
return x;

The LLVM IR in the trivial SSA form
**mem2reg Example**

```c
int x = 0;
if (y > 0)
    x = 1;
return x;
```

The LLVM IR in the trivial SSA form

Minimal SSA after mem2reg
mem2reg Algorithm

• Two main operations
  – Phi placement (Lengauer-Tarjan algorithm)
  – Renaming of the variables

• Intermediate stage breaks SSA invariant
  – Defining semantics & well formedness non-trivial
vmem2reg Algorithm

- Incremental algorithm
- Pipeline of micro-transformations
  - Preserves SSA semantics
  - Preserves well-formedness

- Inspired by Aycock & Horspool 2002.
Example of vmem2reg Algorithm

l₁: %p = alloc i32
    store %p
    %b = %p > 0
    br %b, %l₂, %l₃

l₂:
    store 1, %p
    br
    LAS/LAA

l₃:
    %x = load %p
    ret %x
    DAE
    elim φs
Example of vmem2reg Algorithm

\[
\begin{align*}
    & l_1: \%p = \text{alloca } \text{i32} \\
    & \quad \text{store } 0, \%p \\
    & \quad \%b = \%y > 0 \\
    & \quad \text{br } \%b, \%l_2, \%l_3 \\
    & l_2: \\
    & \quad \text{store } 1, \%p \\
    & \quad \text{br } \%l_3 \\
    & l_3: \\
    & \quad \%x = \text{load } \%p \\
    & \quad \text{ret } \%x
\end{align*}
\]

- How to place phi nodes without breaking SSA?

Find alloca

max \(\phi\)s

LAS/
LAA

DSE

DAE

elim \(\phi\)s
Example of vmem2reg Algorithm

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

```
l_1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2:
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3:
    %x = load %p
    ret %x
```

- Find alloca
- max φs
- LAS/LAA
- DSE
- DAE
- elim φs
Example of vmem2reg Algorithm

l₁: %p = alloca i32  
  store 0, %p  
  %b = %y > 0  
  %x₁ = load %p  
  br %b, %l₂, %l₃

Find alloca  
max φs  
LAS/LAA  
DSE  
DAE  
elim φs

• How to place phi nodes without breaking SSA?

l₂: %x₃ = φ[%x₁, %l₁]

store 1, %p  
%x₂ = load %p  
br %l₃

• Insert
  – Loads at the end of each block
  – Insert φ-nodes at each block

l₃: %x₄ = φ[%x₁; %l₁, %x₂:%l₂]

%x = load %p  
ret %x
Example of vmem2reg Algorithm

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block
  - Insert stores after φ-nodes

```c
l_1: \%p = alloca i32
    store 0, \%p
    \%b = \%y > 0
    \%x_1 = load \%p
    br \%b, \%l_2, \%l_3

l_2: \%x_3 = \phi[\%x_1, \%l_1]
    store \%x_3, \%p
    store 1, \%p
    \%x_2 = load \%p
    br \%l_3

l_3: \%x_4 = \phi[\%x_1, \%l_1, \%x_2, \%l_2]
    store \%x_4, \%p
    \%x = load \%p
    ret \%x
```
Example of vmem2reg Algorithm

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

1. l₁: %p = alloca i32
   store 0, %p
   %b = %y > 0
   %x₁ = load %p
   br %b, %l₂, %l₃

2. l₂: %x₃ = φ[%x₁, %l₁]
   store %x₃, %p
   store 1, %p
   %x₂ = load %p
   br %l₃

3. l₃: %x₄ = φ[%x₁, %l₁, %x₂, %l₂]
   store %x₄, %p
   %x = load %p
   ret %x

Find alloca
max φs
LAS/LAA
DSE
DAE
elim φs
Example of vmem2reg Algorithm

\[ l_1: \% p = \text{alloca i32} \]
\[ \text{store 0, } \% p \]
\[ \% b = \% y > 0 \]
\[ \% x_1 = \text{load } \% p \]
\[ \text{br } \% b, \% l_2, \% l_3 \]

\[ l_2: \% x_3 = \phi[\% x_1, \% l_1] \]
\[ \text{store } \% x_3, \% p \]
\[ \text{store 1, } \% p \]
\[ \% x_2 = \text{load } \% p \]
\[ \text{br } \% l_3 \]

\[ l_3: \% x_4 = \phi[\% x_1, \% l_1, \% x_2: \% l_2] \]
\[ \text{store } \% x_4, \% p \]
\[ \% x = \text{load } \% p \]
\[ \text{ret } \% x \]

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load
Example of vmem2reg Algorithm

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```
l1: %p = alloca i32
store 0, %p
%b = %p > 0
%x1 = load %p
br %b, %l2, %l3

l2: %x3 = φ[0, %l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[0; %l1, %x2 : %l2]
    store %x4, %p
    %x = load %p
    ret %x
```
Example of vmem2reg Algorithm

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

\[ l_1: \%p = \text{alloca } \text{i32} \]
\[ \text{store } 0, \%p \]
\[ \%b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ l_2: \%x_3 = \phi[0, \%l_1] \]
\[ \text{store } \%x_3, \%p \]
\[ \boxed{\text{store } 1, \%p} \]
\[ \%x_2 = \text{load } \%p \]
\[ \text{br } \%l_3 \]

\[ l_3: \%x_4 = \phi[0; \%l_1, \%x_2, \%l_2] \]
\[ \text{store } \%x_4, \%p \]
\[ \%x = \text{load } \%p \]
\[ \text{ret } \%x \]
Example of vmem2reg Algorithm

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```
l1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    br %b, %l2, %l3

l2: %x3 = φ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```
Example of vmem2reg Algorithm

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load
Example of vmem2reg Algorithm

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load

```
l_1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0,%l_1]
    store %x_3, %p
    store 1, %p
    br %l_3

l_3: %x_4 = phi[0:%l_1, 1:%l_2]
    store %x_4, %p
    %x = load %p
    ret %x_4
```
Example of vmem2reg Algorithm

- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.

- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

```assembly
l1:  %p = alloca i32
     store 0, %p
     %b = %y > 0
     br %b, %l2, %l3

l2:  %x3 = phi[0,%l1]
     store %x3, %p
     store 1, %p
     br %l3

l3:  %x4 = phi[0;%l1, 1:%l2]
     store %x4, %p
     ret %x4
```

Find alloca
max φs
LAS/LAA
DSE
DAE
elim φs
Example of vmem2reg Algorithm

- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.

- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

```
l_1: %p = alloca i32
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0,%l_1]
    store %x_3, %p
    store 1, %p
    br %l_3

l_3: %x_4 = phi[0;%l_1, 1;%l_2]
    store %x_4, %p
    ret %x_4
```
Example of vmem2reg Algorithm

$l_1$: 
\[
\% b = \% y > 0 \\
br \% b, \% l_2, \% l_3
\]

$l_2$: 
\[
\% x_3 = \phi[0,\% l_1] \\
br \% l_3
\]

$l_3$: 
\[
\% x_4 = \phi[0;\% l_1, 1:\% l_2] \\
ret \% x_4
\]

• Eliminate $\phi$ nodes:
  – Singletons
  – With identical values from each predecessor
  – See Aycock & Horspool, 2002
Example of vmem2reg Algorithm

\[ l_1: \]
\[ \%b = \%y > 0 \]
\[ \text{br \ %b, \ %l_2, \ %l_3} \]

\[ l_2: \ \%x_3 = \phi[0,\%l_1] \]
\[ \text{br \ %l_3} \]

\[ l_3: \ \%x_4 = \phi[0;\%l_1, \ 1:\%l_2] \]
\[ \text{ret \ %x_4} \]

- Eliminate \( \phi \) nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

Find alloca
max \( \phi \)s
LAS/LAA
DSE
DAE
elim \( \phi \)s
Example of vmem2reg Algorithm

\[ l_1: \]
\[ %b = %y > 0 \]
\[ br %b, %l_2, %l_3 \]

\[ l_2: \]
\[ br %l_3 \]

\[ l_3: %x_4 = \phi[0;%l_1, 1:%l_2] \]
\[ ret %x_4 \]

\[ \text{Find alloca} \]
\[ \text{max } \phi \text{s} \]
\[ \text{LAS/LAA} \]
\[ \text{DSE} \]
\[ \text{DAE} \]
\[ \text{elim } \phi \]

• Done!
How to Establish Correctness?

1. Find `alloca`
2. `max φs`
3. `LAS/LAA`
4. `DSE`
5. `DAE`
6. `elim φ`
How to Establish Correctness?

1. Simple aliasing properties (e.g. to determine promotability)

2. Instantiate proof technique for
   - Substitution
   - Dead Instruction Elimination

\[ P_{DIE} = \ldots \]

   Initialize(\( P_{DIE} \))

   Preservation(\( P_{DIE} \))

   Progress(\( P_{DIE} \))

3. Put it all together to prove composition of “pipeline” correct.
Theorem: The vmem2reg algorithm preserves the semantics of the source program.

Proof:

Composition of simulation relations from the “mini” transformations, each built using instances of the sdom proof technique.

(See Coq Vellvm development.) □
Runtime overhead of verified mem2reg

Vmem2reg: 77%  LLVM’s mem2reg: 81%
(LLVM’s mem2reg promotes allocas used by intrinsics)
Plan

• Vminus: a highly simplified SSA IR based on LLVM
  – What is SSA?

• Verified Compilation of Imp to Vminus
  – What does it mean to “verify compilation”?

• Scaling up: Vellvm
  – Taste of the full LLVM IR
  – Operational Semantics
  – Metatheory + Proof Techniques

• Case studies:
  – SoftBound memory safety

• Conclusion:
  – challenges & research directions
Other Parts of the LLVM IR

\begin{verbatim}

op  ::= %uid | constant | undef  

bop ::= add | sub | mul | shl | ...  

cmpop ::= eq | ne | slt | sle | ...  

insn ::= 
  | %uid = alloca ty  
  | %uid = load ty op1  
  | store ty op1, op2  
  | %uid = getelementptr ty op1 ...  
  | %uid = call rt fun(...args...)  
  | ...  

phi ::= 
  | \phi[op1;lbl1]...[opn;lbln]  

terminator ::= 
  | ret %ty op  
  | br op label %lbl1, label %lbl2  
  | br label %lbl

\end{verbatim}

Operands
Operations
Comparison
Stack Allocation
Load
Store
Address Calculation
Function Calls
Structured Data in LLVM

- LLVM’s IR is uses types to describe the structure of data.

```
ty ::=  
| i1 | i8 | i32 |... | N-bit integers  
| [<#elts> x t] | arrays  
| r (ty₁, ty₂, ... , tyₙ) | function types  
| {ty₁, ty₂, ... , tyₙ} | structures  
| ty* | pointers  
| %Tident | named (identified) type
```

- `<#elts>` is an integer constant `>= 0`
- (Recursive) Structure types can be named at the top level:

```
%T₁ = type {ty₁, ty₂, ... , tyₙ}
```
Example LLVM Types

• An array of 341 integers: \([ 341 \times i32 ]\)

• A 2D array of integers: \([ 3 \times [ 4 \times i32 ] ]\)

• C-style linked lists:
  \[
  \%Node = \text{type } \{ \text{i32, } \%Node^* \}\]

• Structs:
  \[
  \%Rect = \{ \%Point, \%Point, \%Point, \%Point \}
  \%Point = \{ \text{i32, i32} \}\]
GetElementPtr

• LLVM provides the getelementptr instruction to compute pointer values
  – Given a pointer and a “path” through the structured data pointed to by that pointer, getelementptr computes an address
  – This is the abstract analog of the X86 LEA (load effective address). It does not access memory.
  – It is a “type indexed” operation, since the size computations involved depend on the type

\[
\text{insn ::= } \ldots \\
| \text{uid} = \text{getelementptr } t*, \%val, t1 \text{ idx1}, t2 \text{ idx2 }, \ldots
\]
Example

```c
struct RT {
    int A;
    int B[10][20];
    int C;
}
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
}
int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}
```
LLVM’s memory model

%ST = type {i10, [10 x i8*]}

- Manipulate structured types.

%val = load %ST* %ptr
...
store %ST* %ptr, %new
LLVM’s memory model

- Manipulate structured types.
  - `%val = load %ST* %ptr
  ...`  
  - `store %ST* %ptr, %new`

- Semantics is given in terms of byte-oriented low-level memory.
  - padding & alignment
  - physical subtyping
Adapting CompCert’s Memory Model

- Data lives in blocks
- Represent pointers abstractly
  - block + offset
- Deallocate by invalidating blocks
- Allocate by creating new blocks
  - infinite memory available
Dynamic Physical Subtyping

[Nita, et al. POPL '08]
Sources of Undefined Behavior

Target-dependent Results

- Uninitialized variables:
  \[
  \%v = \text{add i32 } \%x, \text{ undef}
  \]

- Uninitialized memory:
  \[
  \%ptr = \text{alloca i32}
  \%v = \text{load (i32*) } \%ptr
  \]

- Ill-typed memory usage

Fatal Errors

- Out-of-bounds accesses
- Access dangling pointers
- Free invalid pointers
- Invalid indirect calls

Nondeterminism

Stuck States
Sources of Undefined Behavior

- **Uninitialized variables:**
  \[
  \%v = \text{add} \ %x, \text{undef}
  \]

- **Uninitialized memory:**
  \[
  \%ptr = \text{alloca} \ i32
  
  \%v = \text{load} \ (i32*) \ %ptr
  \]

- **Ill-typed memory usage**

Defined by a predicate on the program configuration.

\[
\text{Stuck}(f, \sigma) = \text{BadFree}(f, \sigma) \\
\lor \text{BadLoad}(f, \sigma) \\
\lor \text{BadStore}(f, \sigma) \\
\lor \ldots
\]

Nondeterminism

Stuck States
• What is the value of \( \%y \) after running the following?

\[
\%x = \text{or i8 undef, 1} \\
\%y = \text{xor i8 } \%x \%x
\]

• One plausible answer: 0

• Not LLVM’s semantics!

  (LLVM is more liberal to permit more aggressive optimizations)
• Partially defined values are interpreted *nondeterministically* as sets of possible values:

\[
\begin{align*}
\%x &= \text{or } i8 \text{ undefined, } 1 \\
\%y &= \text{xor } i8 \%x \%x \\
\end{align*}
\]

\[
\begin{align*}
[i8 \text{ undefined}] &= \{0, \ldots, 255\} \\
[i8 \ 1] &= \{1\} \\
[\%x] &= \{a \text{ or } b \mid a \in [i8 \text{ undefined}], \ b \in [1]\} \\
&= \{1, 3, 5, \ldots, 255\} \\
[\%y] &= \{a \text{ xor } b \mid a \in [\%x], \ b \in [\%x]\} \\
&= \{0, 2, 4, \ldots, 254\}
\end{align*}
\]
Nondeterministic Branches

11:
  ...
  ...
  ...
  br undefined 12 13

? 

12:
  ...
  ...
  ...

12:
  ...
  ...
  ...
LLVM_{ND} Operational Semantics

- Define a transition relation:
  \[ f \leftarrow \sigma_1 \rightarrow \sigma_2 \]
  - \( f \) is the program
  - \( \sigma \) is the program state: pc, locals(\( \delta \)), stack, heap
- Nondeterministic
  - \( \delta \) maps local \%u\_ids to sets.
  - Step relation is nondeterministic
- Mostly straightforward (given the heap model)
  - One wrinkle: phi-nodes executed atomically
# Operational Semantics

<table>
<thead>
<tr>
<th></th>
<th>Small Step</th>
<th>Big Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nondeterministic</td>
<td>( \text{LLVM}_{ND} )</td>
<td></td>
</tr>
<tr>
<td>Deterministic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Deterministic Refinement

<table>
<thead>
<tr>
<th></th>
<th>Small Step</th>
<th>Big Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nondeterministic</td>
<td>$LLVM_{ND}$</td>
<td>$\cup$</td>
</tr>
<tr>
<td>Deterministic</td>
<td>$LLVM_D$</td>
<td></td>
</tr>
</tbody>
</table>

Instantiate ‘undef’ with default value (0 or null) $\Rightarrow$ deterministic.
# Big-step Deterministic Refinements

<table>
<thead>
<tr>
<th></th>
<th>Small Step</th>
<th>Big Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nondeterministic</td>
<td>LLVM_{ND}</td>
<td></td>
</tr>
<tr>
<td>Deterministic</td>
<td>LLVM_{Interp} ≈ LLVM_{D}</td>
<td></td>
</tr>
</tbody>
</table>

Bisimulation up to “observable events”:

- external function calls
# Big-step Deterministic Refinements

<table>
<thead>
<tr>
<th></th>
<th>Small Step</th>
<th>Big Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nondeterministic</td>
<td>LLVM(_{ND})</td>
<td></td>
</tr>
<tr>
<td>Deterministic</td>
<td>LLVM(<em>{Interp}) ⪅ LLVM(</em>{D}) ⪅ LLVM(<em>{DFn}) ⪅ LLVM(</em>{DB})</td>
<td></td>
</tr>
</tbody>
</table>

Simulation up to “observable events”:
- useful for encapsulating behavior of function calls
- large step evaluation of basic blocks

[Tristan, et al. POPL ’08, Tristan, et al. PLDI ’09]
SoftBound

- Implemented as an LLVM pass.
- Detect spatial/temporal memory safety violations in legacy C code.
- Good test case:
  - Safety Critical $\Rightarrow$ Proof cost warranted
  - Non-trivial Memory transformation
SoPBound

Maintain base and bound for all pointers

Propagate metadata on assignment

Check that a pointer is within its bounds when being accessed

C Source Code → LLVM IR → SoftBound → LLVM IR → Other Optimizations → Target

%p = call malloc [10 x i8]
%q = gep %p, i32 0, i32 255

%p = call malloc [10 x i8]
%p_base = gep %p, i32 0
%p_bound = gep %p, i32 0, i32 10
%q = gep %p, i32 0, i32 255
%q_base = %p_base
%q_bound = %p_bound

assert %q_base <= %q
\ / %q+1 < %q_bound
store i8 0, %q

store i8 0, %q
Disjoint Metadata

- Maintain pointer bounds in a separate memory space.
- Key Invariant: Metadata cannot be corrupted by bounds violation.
Proving SoftBound Correct

1. Define $\text{SoftBound}(f, \sigma) = (f_s, \sigma_s)$
   - Transformation pass implemented in Coq.
2. Define predicate: $\text{MemoryViolation}(f, \sigma)$
3. Construct a non-standard operational semantics:
   $$f \vdash \sigma \xrightarrow{\text{SB}} \sigma'$$
   - Builds in safety invariants “by construction”
   $$f \vdash \sigma \xrightarrow{\text{SB}[*]} \sigma' \implies \neg \text{MemoryViolation}(f, \sigma')$$
4. Show that the instrumented code simulates the “correct” code:
   $$\text{SoftBound}(f, \sigma) = (f_s, \sigma_s) \implies [f \vdash \sigma \xrightarrow{\text{SB}[*]} \sigma'] \approx [f_s \vdash \sigma_s \xrightarrow{\text{SB}[*]} \sigma'_s]$$
Memory Simulation Relation

(MM, M) \approx^o M'

 Globals

 Allocated

 Memory simulation

Frame simulation

(\Delta, \mu) \approx^o \Delta'

p_1 \ b_1 \ e_1
v_2
p_3 \ b_3 \ e_3
v_4

Where V_i \approx^o V'_i
Lessons About SoftBound

• Found several bugs in our C++ implementation

• Simulation proofs suggested a redesign of SoftBound’s handling of stack pointers.
  – Use a “shadow stack”
  – Simplify the design/implementation
  – Significantly more robust (e.g. varargs)
The performance of extracted SoftBound is competitive with the non-verified original.
Related Work

- **CompCert** [Leroy et al.]
- **CompCertSSA** [Barthe, Demange et al. ESOP 2012]
  - Translation validate the SSA construction
- **Verified Software Toolchain** [Appel et. al]
- **Verifiable SSA Representation** [Menon et al. POPL 2006]
  - Identify the well-formedness safety predicate for SSA
- **Specification of SSA**
  - Temporal checking & model checking for proving SSA transforms [Mansky et al, ITP 2010]
  - Matrix representation of $\phi$ nodes [Yakobowski, INRIA]
  - Type system equivalent to SSA [Matsuno et al]
Conclusions

• Proof techniques for verifying SSA transformations
  – Generalize the SSA scoping predicate
  – Preservation/progress + simulations.
  – Simulation proofs

• Verified:
  – Softbound & vmem2reg
  – Similar performance to native implementations

• See the papers/coq sources for details!

• Future:
  – Clean up + make more accessible
  – Alias analysis? Concurrency?
  – Applications to more LLVM-SSA optimizations

http://www.cis.upenn.edu/~stevez/vellvm/