Announcements

• HW 7: Optimization & Experiments
  – Post your benchmark programs early (i.e. tonight!)
  – Due: Tomorrow April 29\textsuperscript{th}

• Final Exam:
  – Thursday, May 7\textsuperscript{th}
  – 9:00AM
  – Moore 216
Vellvm

VERIFYING COMPILER TRANSFORMATIONS
• Define a transition relation:
  \[ f \vdash \sigma_1 \leftrightarrow \sigma_2 \]
  - \( f \) is the program
  - \( \sigma \) is the program state: \( pc \), \( \text{locals}(\delta) \), stack, heap

• Nondeterministic
  - \( \delta \) maps local \%uids to sets.
  - Step relation is nondeterministic

• Mostly straightforward (given the heap model)
  - Another 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>LLVM&lt;sub&gt;ND&lt;/sub&gt;</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>∪</td>
</tr>
<tr>
<td>Deterministic</td>
<td>LLVM_{D}</td>
<td></td>
</tr>
</tbody>
</table>

Instantiate ‘undef’ with default value (0 or null) ⇒ deterministic.
<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>( \text{LLVM}<em>{\text{Interp}} ) (\approx) ( \text{LLVM}</em>{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>(\text{LLVM}_{ND})</td>
<td>(\Uparrow)</td>
</tr>
<tr>
<td>Deterministic</td>
<td>(\text{LLVM}<em>{Interp}) (\approx) (\text{LLVM}</em>{D}) (\supseteq) (\text{LLVM}^<em>_{DFn}) (\supseteq) (\text{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

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 safety property
  – Safety: establish an invariant of the execution of the program

• Compose the individual proofs of correctness
Safety Properties

• A well-formed program never accesses undefined variables.

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

\[
\vdash f \quad \text{program } f \text{ is well formed}
\]

\[
\sigma \quad \text{program state}
\]

\[
f \vdash \sigma \xrightarrow{\ast} \sigma \quad \text{evaluation of } f
\]

• Initialization:

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

• Preservation:

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

• Progress:

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

- A well-formed program never accesses undefined variables.

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

\[
\begin{align*}
\vdash f & \quad \text{program } f \text{ is well formed} \\
\sigma & \quad \text{program state} \\
f \vdash \sigma \xrightarrow{\ast} \sigma & \quad \text{evaluation of } f
\end{align*}
\]

- **Initialization:**

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

- **Preservation:**

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

- **Progress:**

\[
\text{If } \vdash f \quad \text{and} \quad \text{wf}(f, \sigma) \quad \text{then} \quad f \vdash \sigma \xrightarrow{} \sigma'
\]
Well-formed States

State \( \sigma \) is:

\[ pc = \text{program counter} \]
\[ \delta = \text{local values} \]

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

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

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

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

State \( \sigma \) is:
\[
\begin{align*}
    pc &= \text{program counter} \\
    \delta &= \text{local values}
\end{align*}
\]

\[\text{sdom}(f, pc) = \text{variable defns. that strictly dominate } pc.\]
Well-formed States (Roughly)

State \( \sigma \) contains:

- \( pc = \) program counter
- \( \delta = \) local values

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

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

“All variables in scope are initialized.”
mem2reg in LLVM (part of SROA)

- Promote stack allocas to temporaries
- Insert minimal \( \phi \)-nodes

Front-ends w/o SSA construction → The LLVM IR w/o \( \phi \)-nodes → mem2reg → The LLVM IR in the minimal SSA form → Backends

- imperative variables \( \Rightarrow \) stack allocas
- no \( \phi \)-nodes
- trivially in SSA form
The LLVM IR in the trivial SSA form
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.
How to Establish Correctness?

Find alloca
→ max φ s
→ LAS/ LAA
→ DSE
→ DAE
→ elim φ
1. Simple aliasing properties (e.g. to determine promotability)

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

\[ P_{DIE} = \ldots \]

\[ \text{Initialize}(P_{DIE}) \]
\[ \text{Preservation}(P_{DIE}) \]
\[ \text{Progress}(P_{DIE}) \]

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)
**SoftBound**

- Implemented as an LLVM pass.
- Detect spatial/temporal memory safety violations in legacy C code.
- Good test case:
  - Safety Critical ⇒ Proof cost warranted
  - Non-trivial Memory transformation
%p = call malloc [10 x i8]

Maintain base and bound for all pointers

%q = gep %p, i32 0, i32 255

Propagate metadata on assignment

%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

C Source Code

LLVM IR

SoftBound

LLVM IR

Other Optimizations

Target
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:

\[
\text{SoftBound}(f, \sigma) = (f_s, \sigma_s) \implies [f \vdash^S \sigma \xrightarrow{\ast} \sigma'] \succeq [f_s \vdash \sigma_s \xrightarrow{\ast} \sigma_s']
\]
Memory Simulation Relation

Memory simulation

Frame simulation

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

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.
Final Exam

• Will cover material since the midterm almost exclusively
  – Starting from Lecture 14
  – Objects, inheritance, types, implementation of dynamic dispatch
  – Basic optimizations
  – Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    • Liveness
  – Control flow analysis
    • Loops, dominator trees
  – SSA
  – Graph-coloring Register Allocation

• Will focus more on the theory side of things

• Format will be similar to the midterm
  – Simple answer, computation, multiple choice, etc.
  – Sample exam from last time is on the web
What have we learned?
Where else is it applicable?
What next?
Why CIS 341?

• You will learn:
  – Practical applications of theory
  – Parsing
  – How high-level languages are implemented in machine language
  – (A subset of) Intel x86 architecture
  – A deeper understanding of code
  – A little about programming language semantics
  – Functional programming in OCaml
  – How to manipulate complex data structures
  – How to be a better programmer

• Did we meet these goals?
Stuff we didn’t Cover

• We skipped stuff at every level…
• Concrete syntax/parsing:
  – Much more to the theory of parsing…
  – Good syntax is art not science!
• Source language features:
  – Exceptions, recursive data types (easy!), advanced type systems, type inference, concurrency
• Intermediate languages:
  – Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
• Compilation:
  – Continuation-passing transformation, efficient representations, scalability
• Optimization:
  – Scientific computing, cache optimization, instruction selection/optimization
Course Work

• 72% Projects: *The Quaker OAT Compiler*

• 12% Midterm
• 16% Final exam

• Expect this to be a challenging, implementation-oriented course.

I think we met this goal...
Related Courses: Fall 2013

• CIS 500: Software Foundations
  – Dr. Pierce
  – Theoretical course about functional programming, proving program properties, type systems, lambda calculus. Uses the theorem prover Coq.

• CIS 501: Computer Architecture
  – Dr. Devietti
  – 371++: pipelining, caches, VM, superscalar, multicore,…

• CIS 552: Advanced Programming
  – Dr. Weirich
  – Advanced functional programming in Haskell, including generic programming, metaprogramming, embedded languages, cool tricks with fancy type systems

• CIS 670: Special topics in programming languages
  – TBA
Where to go from here?

• Conferences (proceedings available on the web):
  – Programming Language Design and Implementation (PLDI)
  – Principles of Programming Languages (POPL)
  – Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  – International Conference on Functional Programming (ICFP)
  – European Symposium on Programming (ESOP)
  – …

• Technologies / Open Source Projects
  – Yacc, lex, bison, flex, …
  – LLVM – low level virtual machine
  – Java virtual machine (JVM), Microsoft’s Common Language Runtime (CLR)
  – Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?
Where else is this stuff applicable?

- **General programming**
  - In C/C++, better understanding of how the compiler works can help you generate better code.
  - Ability to read assembly output from compiler
  - Experience with functional programming can give you different ways to think about how to solve a problem

- **Writing domain specific languages**
  - lex/yacc very useful for little utilities
  - understanding abstract syntax and interpretation

- **Understanding hardware/software interface**
  - Different devices have different instruction sets, programming models
Thanks!

• To the TAs: Dmitri, Rohan, and Mitchell
  – for doing an amazing job putting together the projects for the course.

• To you for taking the class!

• How can I improve the course?
  – Feedback survey posted to Piazza