Lecture 26

CIS 341: COMPILERS
Announcements

- HW 7: Optimization & Experiments
  - Due: April 29th

- Final Exam:
  - Thursday, May 7th
  - 9:00AM
  - Moore 216

- Visitor: Yaron Minsky of Jane St. Capital
  - Monday, April 27th
  - Lunch: noon – 1:15 (Raisler Lounge) sign-up sheet on Piazza
  - Talk: 2:00 – 3:00 (Raisler Lounge)
    From Theory into Practice: the story of Incremental
COMPILER VERIFICATION
Compiler Correctness?

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

\[
\text{cmd} / \text{st} \xrightarrow{s} \ast \text{SKIP} / \text{st}'
\]

iff

\[
C[\text{cmd}] / C[\text{st}] \xrightarrow{t} \ast C[\text{st'}]
\]

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

- Consider two programs $P_1$ and $P_2$ possibly in different languages.
  - e.g. $P_1$ is an Oat program, $P_2$ is its compilation to LL

- The semantics of the languages associate to each program a set of observable behaviors:
  
  \[ B(P) \text{ and } B(P') \]

- Note: $|B(P)| = 1$ if $P$ is deterministic, $> 1$ otherwise
What is Observable?

- For C-like languages:

  \[
  \text{observable behavior ::=}
  \begin{array}{l}
  \mid \text{terminates}(st) \quad \text{(i.e. observe the final state)} \\
  \mid \text{diverges} \\
  \mid \text{goeswrong}
  \end{array}
  \]

- For pure functional languages:

  \[
  \text{observable behavior ::=}
  \begin{array}{l}
  \mid \text{terminates}(v) \quad \text{(i.e. observe the final value)} \\
  \mid \text{diverges} \\
  \mid \text{goeswrong}
  \end{array}
  \]
What about I/O?

• Add a *trace* of input-output events performed:

\[
\begin{align*}
t & ::= \emptyset \mid e :: t \quad \text{(finite traces)} \\
\text{coind. } T & ::= \emptyset \mid e :: T \quad \text{(finite and infinite traces)}
\end{align*}
\]

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

• P1:
  \[
  \text{print}(1); / \text{st} \quad \Rightarrow \quad \text{terminates(out}(1)::[], \text{st})
  \]

• P2:
  \[
  \text{print}(1); \text{ print}(2); / \text{st} \quad \Rightarrow \quad \text{terminates(out}(1)::\text{out}(2)::[], \text{st})
  \]

• P3:
  \[
  \text{WHILE true DO print}(1) \text{ END / st} \quad \Rightarrow \quad \text{diverges(out}(1)::\text{out}(1)::…)
  \]

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

• Two programs $P_1$ and $P_2$ are bisimilar whenever:

$$\mathcal{B}(P_1) = \mathcal{B}(P_2)$$

• The two programs are completely indistinguishable.

• But… this is often too strong in practice.
• 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.

    $a := x + 1; b := a \ || \ x := x+1$

• LLVM explicitly allows nondeterminism:
  – undef values (not part of LLVM lite)
  – see the discussion later
Backward Simulation

• Program P2 can exhibit fewer behaviors than P1:

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

• 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.

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

(divide by 0 error) (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:

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

• Idea: let \( S \) be the \textit{functional specification} of the program:
  A set of behaviors not containing \( \text{goes\text{wrong}}(t) \).
  \( \quad \) 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.
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!
Safe Forwards Simulation

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

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

• 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: $\emptyset \subseteq \mathcal{B}(P1) \subseteq \mathcal{B}(P2) = \{b\} \text{ so } \mathcal{B}(P1) = \{b\}$.

- Corollary: safe forward simulation implies safe backward simulation if P2 is deterministic.
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
A single source-program step is simulated by a single target step.

(Solid = assumptions, Dashed = must be shown)
A single source-program step is simulated by *one or more* target steps. (But only finitely many!)

(Solid = assumptions, Dashed = must be shown)
Optional Forward Simulation

Source: $\sigma_1 \rightarrow \sigma_2$

Target: $\mathcal{C}[\sigma_1]$

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.
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 many LLVM optimizations
\o\ is either an “observable event” or a “silent event”
\o\ ::= e | \epsilon

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

Or

– the target takes a silent transition to a smaller state

Example use: adding an instruction to the target.
Verifying optimizations at the LLVM level of abstraction.

**EXAMPLE: VELLVM**
Step 1: Define LLVM IR Semantics

- Essentially: define an interpreter for LLVM IR code

- But: more complex than the LLVMlite we use in class
  - Aggregate / Structured data
  - Undefined behaviors
  - Nondeterminism

- So: can’t be just an interpreter
  - Semantics is given by a relation
### Other Parts of the LLVM IR

| 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 | ::= |
| \(φ[\text{op1};\text{lbl1}]…[\text{opn};\text{lbln}]\) |

| terminator | ::= |
| *ret* %ty op |
| *br* op *label* %lbl1, *label* %lbl2 |
| *br* *label* %lbl |
Sources of Undefined Behavior

Target-dependent Results

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

• 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

Target-dependent Results

• Uninitialized variables:
  \%
  \texttt{v} = \texttt{add i32 \%x, undef}

• Uninitialized memory:
  \%
  \texttt{ptr} = \texttt{alloca i32}
  \%
  \texttt{v} = \texttt{load (i32*) \%ptr}

• Ill-typed memory usage

Defined by a predicate on the program configuration.

A program configuration is \textit{stuck} if there is no transition it can make.

\[
\text{Stuck}(f, \sigma) = \text{BadFree}(f, \sigma) \vee \text{BadLoad}(f, \sigma) \vee \text{BadStore}(f, \sigma) \vee \ldots \vee \text{...}
\]
LLVM’s memory model

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

- Manipulate structured types.

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

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

• 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]
• What is the value of %y after running the following?

```assembly
% x = or i8 undef, 1
% y = 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:

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

\[
\begin{align*}
[\text{i8 undef}] &= \{0, \ldots, 255\} \\
[\text{i8 1}] &= \{1\}
\end{align*}
\]

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

11:
...
...
...
\texttt{br \textbf{undef} 12 13}

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

- Define a transition relation:
  \[ f \vdash \sigma_1 \rightarrow \sigma_2 \]
  - \( f \) is the program
  - \( \sigma \) is the program state: pc, locals(\( \delta \)), stack, heap
- Nondeterministic
  - \( \delta \) maps local \$_{\text{uids}}$ to sets.
  - Step relation is nondeterministic
- Mostly straightforward (given the heap model)
  - Another wrinkle: phi-nodes executed atomically
Need for Atomic Phi-node Updates

\begin{verbatim}
blk:
  %x = phi i32 [ %z, %blk ], [ 0, %pred ]
  %z = phi i32 [ %x, %blk ], [ 1, %pred ]
  %b = icmp leq %x %z
  br %b %blk %succ
\end{verbatim}
## 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>(\cup)</td>
</tr>
<tr>
<td>Deterministic</td>
<td>LLVM(_D)</td>
<td></td>
</tr>
</tbody>
</table>

Instantiate ‘undefined’ 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} \approx 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>( \text{LLVM}^*_\text{ND} )</td>
<td></td>
</tr>
<tr>
<td>Deterministic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{LLVM}_{\text{Interp}} ) ≈ ( \text{LLVM}_D )</td>
<td>( \text{LLVM}^<em>_\text{DFn} ) ( \supseteq ) ( \text{LLVM}^</em>_\text{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