Synthesizing an Instruction Selection Rule Library from Semantic Specifications

Sebastian Buchwald, Andreas Fried, Sebastian Hack
Instruction Selection

- Replace **IR pattern** with a single **goal instruction**
- No total ordering, no (virtual) register allocation yet
State of the Art

- Syntactic specification of patterns
- Code generation
- E.g. GCC machine description, LLVM TableGen
- Large rule libraries, growing larger
- Tedious manual maintenance
- Error-prone, especially missing patterns
Multiple Patterns per Goal

- Full support of new instruction needs 4 rules + commutativity
- Easier to specify **semantics** once
Existing Rulesets are Incomplete

- x86 has extensive addressing modes

$$r = \&a[x + 4*y + 42]; \quad r = *(p + x + x);$$

$$\Rightarrow$$

$$\text{leal } a+42(\%x,\%y,4), \%r \quad \text{movb } (\%p,\%x,2), \%r$$
Existing Rulesets are Incomplete

- x86 has extensive addressing modes
  
  \[
  r = \&a[x + 4*y + 42]; \quad r = *(p + x + x);
  \]
  
  \[
  \xrightarrow{\text{leal}} \quad \text{leal} a+42(%x,%y,4), %r
  \]
  
  \[
  \xrightarrow{\text{movb}} \quad \text{movb} (%p,%x,2), %r
  \]

- Rules are missing from GCC 7.3 (left) and Clang 6.0.0 (right)
  
  \[
  \text{leal} (%x,%y,4), %z
  \]
  
  \[
  \text{addl} \ %x, \ %p
  \]
  
  \[
  \text{addl} \ $a+42, %z
  \]
  
  \[
  \text{movb} (%x,%p), %r
  \]
Existing Rulesets are Incomplete

- x86 has extensive addressing modes
  \[
  r = \&a[x + 4*y + 42]; \quad \Rightarrow \quad \text{leal } a+42(%x,%y,4), \%r \\
  \text{movb } (%p,\%x,2), \%r
  \]

- Rules are missing from GCC 7.3 (left) and Clang 6.0.0 (right)
  \[
  \text{leal } (%x,%y,4), \%z \quad \text{addl } \%a+42, \%z \\
  \text{movb } (%p,\%x,2), \%r
  \]

- ...but susceptible to commutativity or associativity
  \[
  r = \&a[42 + x + 4*y]; \quad \Rightarrow \quad \text{leal } a+42(%x,%y,4), \%r \\
  \text{movb } (%p,\%x,2), \%r
  \]
New Approach

- **Semantic** specification of instructions
- Synthesize rule library
  - For each machine instruction $g$:
    - Find all smallest IR patterns equivalent to $g$
- **Correct and complete** rule libraries
- Push-button support for new ISAs or ISA extensions
Specifying Instructions
Gulwani et al., PLDI 2011

\[
\begin{align*}
&v_a[0] && v_a[1] \\
&\uparrow && \uparrow \\
&\text{And} && \\
&v_r[0] \\
&v_a[0] \quad v_a[0] \\
&\uparrow \\
&\text{Not} && \\
&v_r[0] \\
&v_a[0] && v_a[1] \\
&\uparrow && \uparrow \\
&\text{andn} && \\
&v_r[0]
\end{align*}
\]

Specification as SMT terms:
- Arguments \(v_a\) and results \(v_r\) are 32-bit bitvectors
- Semantics \(Q\) relate arguments to results:
  - \(Q_{\text{And}} = (v_r[0] = v_a[0] \land v_a[1])\)
  - \(Q_{\text{Not}} = (v_r[0] = \neg v_a[0])\)
  - \(Q_{\text{andn}} = (v_r[0] = \neg v_a[0] \land v_a[1])\)
Component-Based Synthesis
Gulwani et al., PLDI 2011

Provide IR instructions as components, machine instruction as goal
Component-Based Synthesis
Gulwani et al., PLDI 2011

- Provide IR instructions as components, machine instruction as goal
- SMT encoding of connections between components
Component-Based Synthesis
Gulwani et al., PLDI 2011

- Provide IR instructions as components, machine instruction as goal
- SMT encoding of connections between components
- Produce \textit{pattern semantics} $Q^+$ from connections

\[
Q^+ = Q_{\text{Xor}}([a, b], [c]) \land Q_{\text{And}}([d, e], [f]) \land (a = x) \land (b = y) \land (d = c) \land (e = y) \land (\text{result} = f)
\]
Component-Based Synthesis
Gulwani et al., PLDI 2011

- Provide IR instructions as components, machine instruction as goal
- SMT encoding of connections between components
- Produce pattern semantics $Q^+$ from connections
- SMT solver finds connections with correct semantics

$$Q^+ = Q_{Xor}([a, b], [c]) \land Q_{And}([d, e], [f]) \land$$
$$\quad (a = x) \land (b = y) \land (d = c) \land (e = y) \land (result = f)$$
IR graph includes memory dependencies (→ HotSpot)
- Actually use notional SSA value for memory state $m : M$
- Store: update, Load: query
Theory “ArraysEx” provides maps, $M = \text{Array} (\text{Pointer, Value})$

Problem: $\forall m : M \ldots$ and $\not\exists m : M \ldots: 2^{235}$ possibilities
SMT Representation

- Theory “ArraysEx” provides maps, $M = \text{Array}(\text{Pointer}, \text{Value})$
- Problem: $\forall m : M \ldots$ and $\not\exists m : M \ldots$: $2^{35}$ possibilities

- But most addresses are irrelevant

⇒ Extract **symbolic** addresses from goal’s semantics
Only model those

| addr $\mapsto$ | \( (*\text{addr} + x)_{0...7} \) |
| addr + 1 $\mapsto$ | \( (*\text{addr} + x)_{8...15} \) |
| addr + 2 $\mapsto$ | \( (*\text{addr} + x)_{16...23} \) |
| addr + 3 $\mapsto$ | \( (*\text{addr} + x)_{24...31} \) |
Synthesis Task

\( \exists p : \text{Pattern. } \forall v_a : \text{Args. } \forall v_r : \text{Results. } Q^+(p, v_a, v_r) \iff Q(\text{goal}, v_a, v_r) \)

Unfortunately intractable as-is:

- \( \forall \) quantifiers
  - \( \Rightarrow \) Counterexample-guided inductive synthesis (CEGIS)
- Too many different components
  - Gulwani’s technique: Assumes right components already selected
  - Enumeration: Search space too large
- \( \Rightarrow \) Need a compromise
Iterative CEGIS

\[ \exists p : \text{Pattern. } \forall v_a : \text{Args. } \forall v_r : \text{Results. } Q^+(p, v_a, v_r) \iff Q(\text{goal}, v_a, v_r) \]

- Gulwani’s algorithm has problems with extraneous components
  - IRs provide \( > 20 \) instructions
  - Each pattern needs few, but some multiple times

Solution:
- Iterate over \textbf{sub-multisets} of IR in increasing size
  - Run synthesis for each

\[ IR = \{ \text{Add, Load, Store} \} \]

\[
\{ \text{Add} \} \\
\{ \text{Load} \} \\
\{ \text{Store} \} \\
\{ \text{Add, Add} \} \\
\{ \text{Add, Load} \} \\
\{ \text{Add, Store} \} \\
\{ \text{Load, Load} \} \\
\{ \text{Load, Store} \} \\
\{ \text{Store, Store} \} \\
\{ \text{Add, Add, Add} \} \\
\{ \text{Add, Add, Load} \} \ldots \]
## Synthesis Results

- IR: 22 simple operations
- Machine instructions: IA32 32-bit integer subset
  - Basic group: RISC-like, no addressing mode
- One eight-core desktop workstation

<table>
<thead>
<tr>
<th>Group</th>
<th>#Goals</th>
<th>#Patterns</th>
<th>Max. Size</th>
<th>Synthesis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>39</td>
<td>575</td>
<td>4</td>
<td>3:25</td>
</tr>
<tr>
<td>Load/Store</td>
<td>35</td>
<td>607</td>
<td>4</td>
<td>5:45</td>
</tr>
<tr>
<td>Unary arithmetic</td>
<td>70</td>
<td>2106</td>
<td>7</td>
<td>18:10:58</td>
</tr>
<tr>
<td>Binary arithmetic</td>
<td>260</td>
<td>6316</td>
<td>6</td>
<td>10:27:06</td>
</tr>
<tr>
<td><code>cmp/test; jcc</code></td>
<td>265</td>
<td>145441</td>
<td>7</td>
<td>3:00:07:05</td>
</tr>
<tr>
<td>Total</td>
<td>630</td>
<td>154470</td>
<td>7</td>
<td>4:04:50:54</td>
</tr>
</tbody>
</table>
Application: Instruction Selection

Turn patterns into instruction selection rules

- Greedy DAG matcher (≈ LLVM)
- Integrated in FIRM research compiler
  - Synthesized matcher goes first
  - Handwritten matcher used as fallback
- Synthesized matcher covers 75.7% of SPEC CINT2000
SPEC CINT2000 Performance Results

Overhead

-2.5 %
-0 %
+5 %
+10 %
+11.56 %
+15 %

253.perlbmk +30.67%
186.crafty
255.vortex

256.bzip2
176.gcc
175.vpr
197.parser

254.gap
181.mcf
300.twolf
164.gzip

254.gap
256.bzip2
253.perlbmk
176.gcc
300.twolf
181.mcf
164.gzip

255.vortex
186.crafty

Basic Handwritten
Full Handwritten
Rule Libraries of Other Compilers

- Turn patterns into compiler test cases

```
char a[4242];
char *ia32_Lea(int x, int y)
{
    return &a[x + 4 * y + 42];
}
```

- Compile and check for goal instruction `leal a+42(\%x, \%y, 4), \%r`
Results

- GCC 7.3 supports 31 400 / 63 012 rules (50 %)
- Clang 6.0.0 RC3 supports 26 647 / 63 012 rules (42 %)

More information on our website: [http://libfirm.org/selgen](http://libfirm.org/selgen)
- Full tables of unsupported patterns
- Links to examples in Godbolt’s Compiler Explorer

⇒ Instruction selection patterns still missing in production compilers
Further Work

Waiting for SMT solver progress
- Floating point
- Division

The to-do list
- Synthesis techniques for larger patterns
- Vector instructions, loops
- Multiple bit widths

Might be a good idea
- Completeness vs. synthesis performance
- Function calls
Conclusion

Contributions

- Automatic synthesis of instruction rule libraries
  - Memory encoding for synthesis
  - Iterative CEGIS
- Generated instruction selector
  - On par with handwritten counterpart
- Instruction selector testing
  - Manual rule libraries are incomplete

Artifact

- Synthesis tool, research compiler libFIRM, compiler testing scripts
- Freely available under GPL

http://libfirm.org/selgen
END
SMT →
\[ \exists x : \text{BitVec}_{32}. \exists y : \text{BitVec}_{32}. x > 0 \land y > 0 \land x \times x + y \times y = 0 \]

- SAT + first-order quantifiers + Theories

- Solver produces model for outer \( \exists \) quantifiers
  - No other quantifiers: “quantifier-free” → better performance
\[ \exists x : \text{BitVec}_{32}. \exists y : \text{BitVec}_{32}. x > 0 \land y > 0 \land x \times x + y \times y = 0 \]

- SAT + first-order quantifiers + **Theories**
  - “FixedSizeBitVectors” implements two’s-complement arithmetic
- Solver produces model for outer \( \exists \) quantifiers
  - No other quantifiers: “quantifier-free” \( \rightarrow \) better performance
\[ \exists x : \text{BitVec}_{32}. \exists y : \text{BitVec}_{32}. x > 0 \land y > 0 \land x \times x + y \times y = 0 \]

- **SAT + first-order quantifiers + Theories**
  - “FixedSizeBitVectors” implements two’s-complement arithmetic

- **Solver produces model for outer \( \exists \) quantifiers**
  - No other quantifiers: “quantifier-free” → better performance

- **Model:** \( x = 16382 \times 2^{16}, y = 32766 \times 2^{16} \)
Mem →
Memory Access

- Store: update, Load: query
Store: update, Load: query
Keep the antidependencies!
Remembering Loads

Remember loads with **access flag**.

\[ M = \text{Pointer} \rightarrow (\text{Bool} \times \text{Value}) \]

**Load** Set access flag, extract data

**Store** Update data, leave access flag untouched

\[
\begin{align*}
\text{addr} & \mapsto \text{old_addr} + x
\end{align*}
\]
Remembering Loads

Remember loads with access flag.

\[ M = \text{Pointer} \rightarrow (\text{Bool} \times \text{Value}) \]

**Load**  Set access flag, extract data

**Store**  Update data, leave access flag untouched

\[
\begin{array}{c|c}
\text{addr} & \text{old_addr} + x \\
\hline
\text{flag} & \text{value} \\
\end{array}
\]
Theory “ArraysEx” provides maps, $M = Array(Pointer, (Bool \times Value))$

Problem: $\forall m : M \ldots$ and $\not\exists m : M \ldots$: $2^{2^{35}}$ possibilities
Theory “ArraysEx” provides maps, $M = \text{Array}(\text{Pointer}, (\text{Bool} \times \text{Value}))$

Problem: $\forall m : M \ldots$ and $\not\exists m : M \ldots$: $2^{35}$ possibilities

But most addresses are irrelevant

⇒ Extract relevant addresses from goal’s semantics
Only model those

Bit-vectors for efficiency

\[
v_a[1] + 0 \quad v_a[1] + 1 \quad v_a[1] + 2 \quad v_a[1] + 3
\]

flag data
Gulwani
Component-Based Synthesis

Gulwani et al., PLDI 2011

Constraints ensure well-formedness

Derive pattern semantics

Q + (p, va, vr) from assignment to *-pos
Component-Based Synthesis
Gulwani et al., PLDI 2011

Constraints ensure well-formedness
Derive pattern semantics

Q + (p, v\textsubscript{a}, v\textsubscript{r}) from assignment to *-pos
Component-Based Synthesis
Gulwani et al., PLDI 2011

Constraints ensure well-formedness
Derive pattern semantics

Q + (p, va, vr) from assignment to *-pos
Component-Based Synthesis
Gulwani et al., PLDI 2011

Constraints ensure well-formedness
Derive pattern semantics

Q(p, v_a, v_r) from assignment to *-pos

Load
load-pos = 3

Add
add-pos = 5

Store
store-pos = 6
Constraints ensure well-formedness

Derive pattern semantics $Q^+ (p, v_a, v_r)$ from assignment to *-pos
CEGIS →
Counterexample-Guided Inductive Synthesis
a. k. a. CEGIS

$$\exists p : \text{Pattern. } \forall v_a : \text{Args. } \forall v_r : \text{Results. } Q^+(p, v_a, v_r) \iff Q(\text{goal}, v_a, v_r)$$

- Small set of test-cases $T$ usually enough

$$T \leftarrow \emptyset$$

$$T \leftarrow T \cup \{t^*\}$$

Synthesis

$$\exists p : \text{Pattern. } \forall t \in T (\text{testOK}(p, t))$$

Counterexample

$$\exists t : \text{TestCase. } \neg \text{testOK}(p^*, t)$$

$$\text{unsat}$$

$$\text{unsat}$$

no solution
linear →
Linear Type Encoding

- Alternative to the access flag
- Add SMT constraints to ensure linear type property (i.e. exactly one use per def)

\[ \sum_i (\text{use-}i\text{-arg-}0\text{-pos} = \text{def-pos}) = 1 \]

- **Pseudo-boolean** constraint, supported by Z3 but not SMT-LIB
- Other optimization relies on access flag
opts →
Further Optimizations

- Load/Store/both necessary?

$$\exists m_{\text{before}} : M. \exists m_{\text{after}} : M. Q(\text{goal}, [\ldots m_{\text{before}} \ldots], [\ldots m_{\text{after}} \ldots])$$

$$\land m_{\text{before}} \neq m_{\text{after}}$$

- $\geq d$ uses of a sort with $d$ defs?
- Source (def without use) for all uses?
COV →
Frequency of unsupported instructions:

- Phi/Sync 35.7%
- Conditional 20.0%
- Call 18.5%
- Internal 11.1%
- Load/Store 7.8%
- Cast 4.8%
- Arithmetic 1.5%
- Div/Mod 0.4%
- Builtin 0.1%