Karlsruhe Institute of Technology

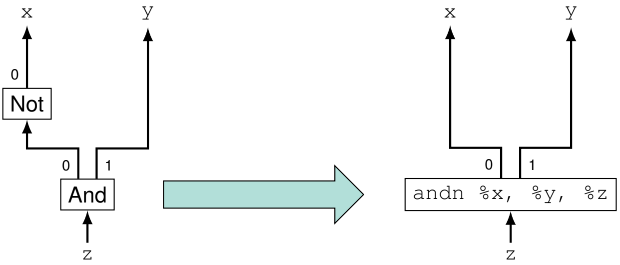
SAARLAND UNIVERSITY
COMPUTER SCIENCE

Synthesizing an Instruction Selection Rule Library from Semantic Specifications

Sebastian Buchwald, Andreas Fried, Sebastian Hack

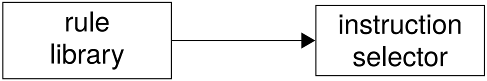
Programming Paradigms Chair, IPD Snelting   Compiler Design Lab

Instruction Selection



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

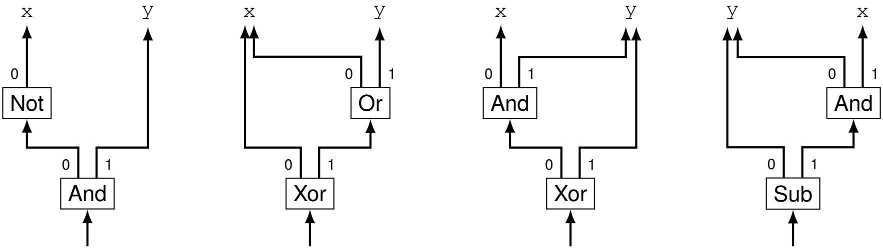
State of the Art



- 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];`	`r = *(p + x + x);`
$\Longrightarrow$	$\Longrightarrow$
`leal a+42(%x,%y,4), %r`	`movb (%p,%x,2), %r`

Existing Rulesets are Incomplete

- x86 has extensive addressing modes

`r = &a[x + 4*y + 42];`	`r = *(p + x + x);`
$\Longrightarrow$	$\Longrightarrow$
`leal a+42(%x,%y,4), %r`	`movb (%p,%x,2), %r`

- Rules are missing from GCC 7.3 (left) and Clang 6.0.0 (right)

`leal (%x,%y,4), %z`	`addl %x, %p`
`addl $a+42, %z`	`movb (%x,%p), %r`

Existing Rulesets are Incomplete

- x86 has extensive addressing modes

`r = &a[x + 4*y` **`+ 42`**`];`	`r = *(`**`(`**`p + x`**`)`**` + x);`
⟹	⟹
`leal a+42(%x,%y,4), %r`	`movb (%p,%x,2), %r`

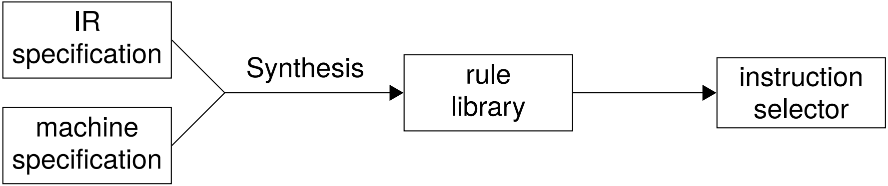
- Rules are missing from GCC 7.3 (left) and Clang 6.0.0 (right)

`leal (%x,%y,4), %z`	`addl %x, %p`
`addl $a+42, %z`	`movb (%x,%p), %r`

- ...but susceptible to commutativity or associativity

`r = &a[`**`42 +`**` x + 4*y];`	`r = *(p + `**`(`**`x + x`**`)`**`));`
⟹	⟹
`leal a+42(%x,%y,4), %r`	`movb (%p,%x,2), %r`

New Approach

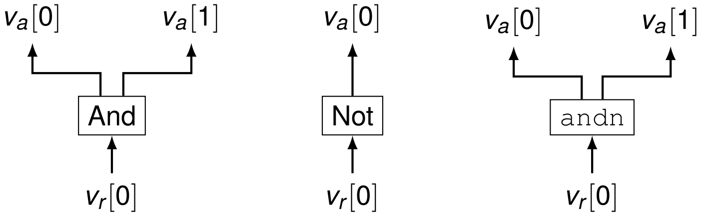


- **Semantic** specification of instructions
- Synthesize rule library
  - For each machine instruction *g*:
    Find all smallest IR patterns equivalent to *g*

- **Correct and complete** rule libraries
- Push-button support for new ISAs or ISA extensions

Specifying Instructions

Gulwani et al., PLDI 2011

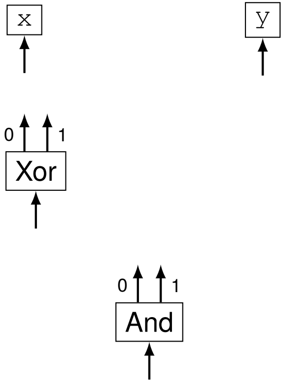


Specification as SMT terms:

- Arguments v_a and results v_r are 32-bit bitvectors
- Semantics Q relate arguments to results:
  - $Q_{And} = (v_r[0] = v_a[0] \wedge v_a[1])$
  - $Q_{Not} = (v_r[0] = \neg v_a[0])$
  - $Q_{\texttt{andn}} = (v_r[0] = \neg v_a[0] \wedge v_a[1])$

Component-Based Synthesis

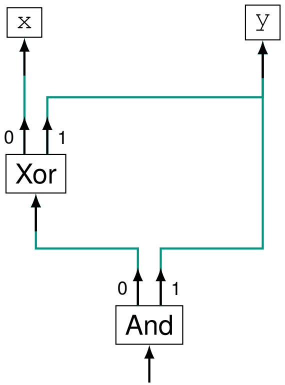
Gulwani et al., PLDI 2011



- Provide IR instructions as components, machine instruction as goal

Component-Based Synthesis

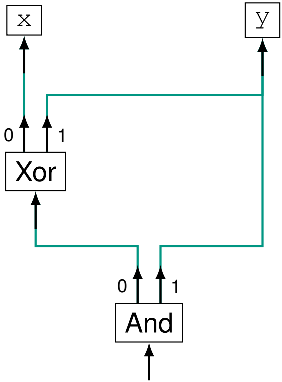
Gulwani et al., PLDI 2011



- Provide IR instructions as components, machine instruction as goal
- SMT encoding of connections between components

Component-Based Synthesis

Gulwani et al., PLDI 2011

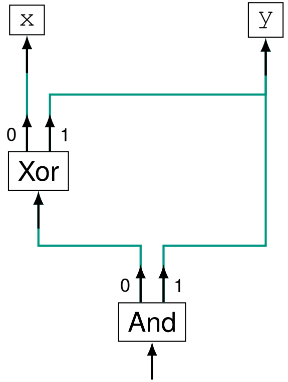


- Provide IR instructions as components, machine instruction as goal
- SMT encoding of connections between components
- Produce **pattern semantics** Q^+ from connections

$$Q^+ = Q_{Xor}([a,b],[c]) \wedge Q_{And}([d,e],[f]) \wedge$$
$$(a = \mathrm{x}) \wedge (b = \mathrm{y}) \wedge (d = c) \wedge (e = \mathrm{y}) \wedge (result = f)$$

Component-Based Synthesis

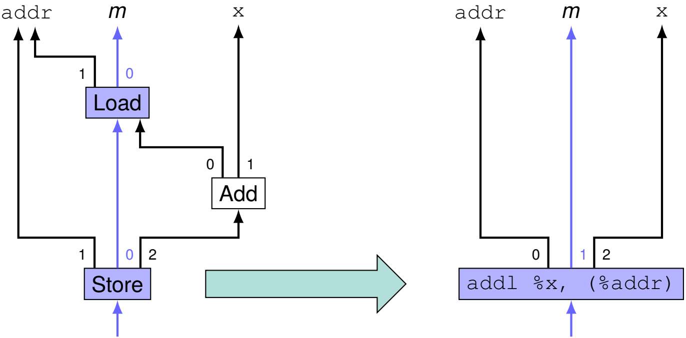
Gulwani et al., PLDI 2011



- 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

$$Q^+ = Q_{Xor}([a,b],[c]) \wedge Q_{And}([d,e],[f]) \wedge \\ (a = \mathrm{x}) \wedge (b = \mathrm{y}) \wedge (d = c) \wedge (e = \mathrm{y}) \wedge (result = f)$$

Memory Access



- IR graph includes memory dependencies (→ HotSpot)
- Actually use notional SSA value for memory state $m : M$
- Store: update, Load: query

SMT Representation

- Theory "ArraysEx" provides maps, $M = Array(Pointer, Value)$
- Problem: $\forall m : M \ldots$ and $\nexists m : M \ldots$: $2^{2^{35}}$ possibilities

SMT Representation

Karlsruhe Institute of Technology

- Theory “ArraysEx” provides maps, $M = \mathit{Array}(\mathit{Pointer}, \mathit{Value})$
- Problem: $\forall m : M \ldots$ and $\nexists m : M \ldots$: $2^{2^{35}}$ possibilities

- But most addresses are irrelevant

$\Rightarrow$ Extract **symbolic** addresses from goal’s semantics
Only model those

`addr` $\mapsto$	$($`*addr + x`$)_{0 \ldots 7}$
`addr + 1` $\mapsto$	$($`*addr + x`$)_{8 \ldots 15}$
`addr + 2` $\mapsto$	$($`*addr + x`$)_{16 \ldots 23}$
`addr + 3` $\mapsto$	$($`*addr + x`$)_{24 \ldots 31}$

Synthesis Task

Karlsruhe Institute of Technology

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

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

Karlsruhe Institute of Technology

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

- Gulwani's algorithm has problems with extraneous components
  - IRs provide $>$ 20 instructions
  - Each pattern needs few, but some multiple times

Solution:

- Iterate over **sub-multisets** of IR in increasing size
  - Run synthesis for each

$IR = \{\text{Add}, \text{Load}, \text{Store}\}$

$\{\text{Add}\}$
$\{\text{Load}\}$
$\{\text{Store}\}$

$\{\text{Add}, \text{Add}\}$
$\{\text{Add}, \text{Load}\}$
$\{\text{Add}, \text{Store}\}$
$\{\text{Load}, \text{Load}\}$
$\{\text{Load}, \text{Store}\}$
$\{\text{Store}, \text{Store}\}$

$\{\text{Add}, \text{Add}, \text{Add}\}$
$\{\text{Add}, \text{Add}, \text{Load}\}$ …

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

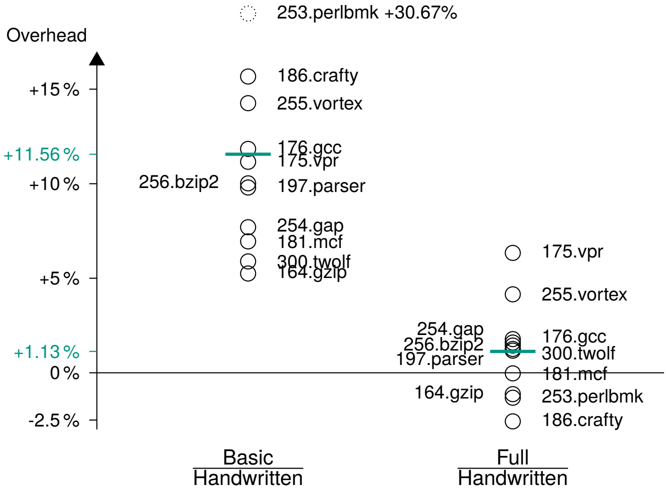
Group	#Goals	#Patterns	Max. Size	Synthesis Time
Basic	39	575	4	3:25
Load/Store	35	607	4	5:45
Unary arithmetic	70	2106	7	18:10:58
Binary arithmetic	260	6316	6	10:27:06
`cmp/test; jcc`	265	145441	7	3:00:07:05
Total	630	154470	7	4:04:50:54

Application: Instruction Selection

Turn patterns into instruction selection rules

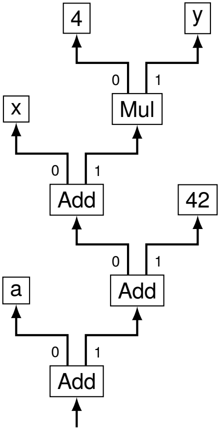
- Greedy DAG matcher ($\approx$ LLVM)
- Integrated in FIRM research compiler
  - Synthesized matcher goes first
  - Handwritten matcher used as fallback
- Synthesized matcher covers 75.7 % of SPEC CINT2000

SPEC CINT2000 Performance Results



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`

- Full tables of unsupported patterns
- Links to examples in Godbolt's Compiler Explorer

⇒ Instruction selection patterns still missing in production compilers

Further Work

Karlsruhe Institute of Technology

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→

SMT Solving

Karlsruhe Institute of Technology

$$\exists x : \text{BitVec}_{32}. \exists y : \text{BitVec}_{32}. x > 0 \land y > 0 \land x * x + y * y = 0$$

- SAT + first-order quantifiers + **Theories**

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

SMT Solving

Karlsruhe Institute of Technology

$$\exists x : \text{BitVec}_{32}. \exists y : \text{BitVec}_{32}. x > 0 \land y > 0 \land x * x + y * y = 0$$

- 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

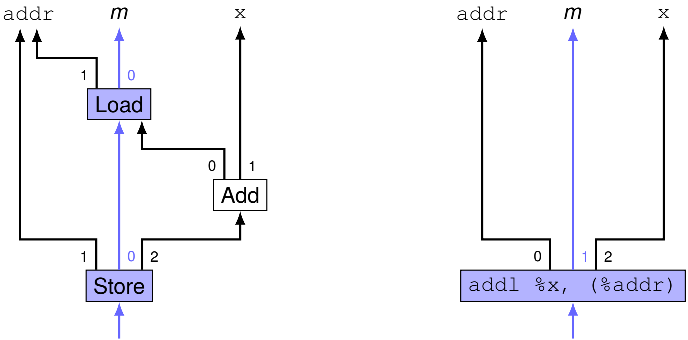
SMT Solving

$$\exists x : \mathrm{BitVec}_{32}. \exists y : \mathrm{BitVec}_{32}. x > 0 \land y > 0 \land x * x + y * y = 0$$

- 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
- Model: $x = 16382 * 2^{16}$, $y = 32766 * 2^{16}$

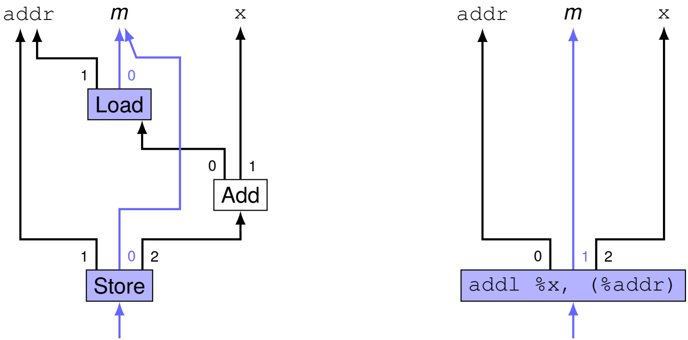
Mem→

Memory Access



- Store: update, Load: query

Memory Access



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

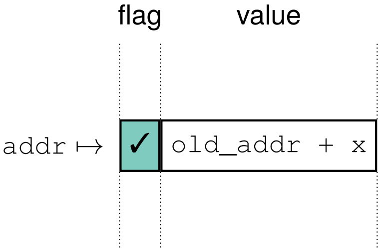
Remembering Loads

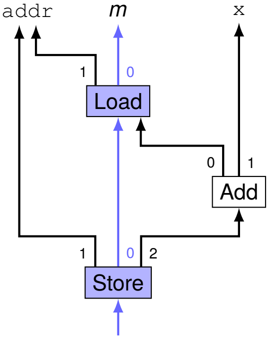
Remember loads with **access flag**.

$M = \mathit{Pointer} \to (\mathit{Bool} \times \mathit{Value})$

Load Set access flag, extract data

Store Update data, leave access flag untouched





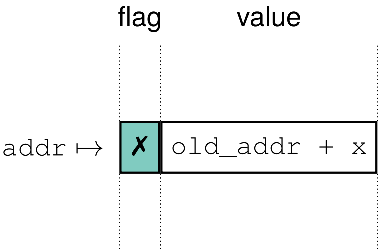
Remembering Loads

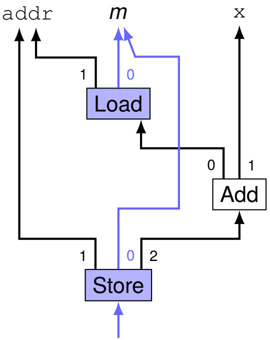
Remember loads with **access flag**.

$$M = \mathit{Pointer} \to (\mathit{Bool} \times \mathit{Value})$$

Load Set access flag, extract data

Store Update data, leave access flag untouched





SMT Representation

- Theory “ArraysEx” provides maps, $M = \mathit{Array}(\mathit{Pointer}, (\mathit{Bool} \times \mathit{Value}))$
- Problem: $\forall m : M \ldots$ and $\nexists m : M \ldots$: $2^{2^{35}}$ possibilities

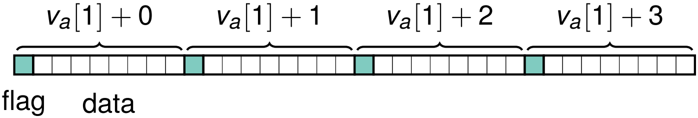
SMT Representation

- Theory “ArraysEx” provides maps, $M = Array(Pointer, (Bool \times Value))$
- Problem: $\forall m : M \ldots$ and $\nexists m : M \ldots$: $2^{2^{35}}$ possibilities

- But most addresses are irrelevant

$\Rightarrow$ Extract relevant addresses from goal’s semantics
Only model those

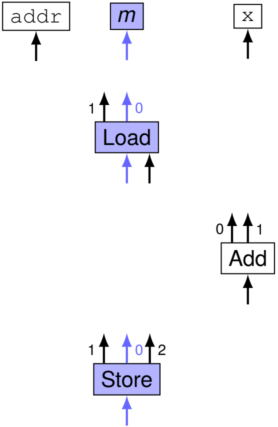
- Bit-vectors for efficiency



Gulwani→

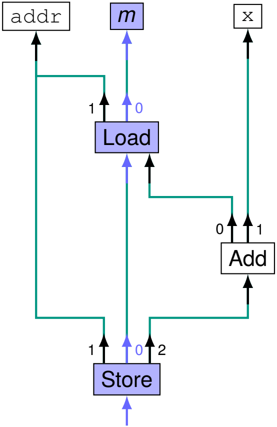
Component-Based Synthesis

Gulwani et al., PLDI 2011



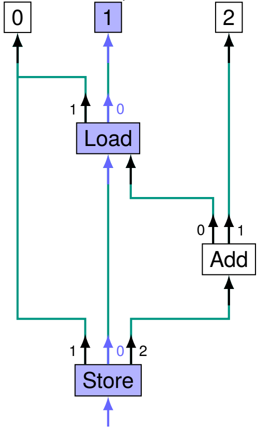
Component-Based Synthesis

Gulwani et al., PLDI 2011



Component-Based Synthesis

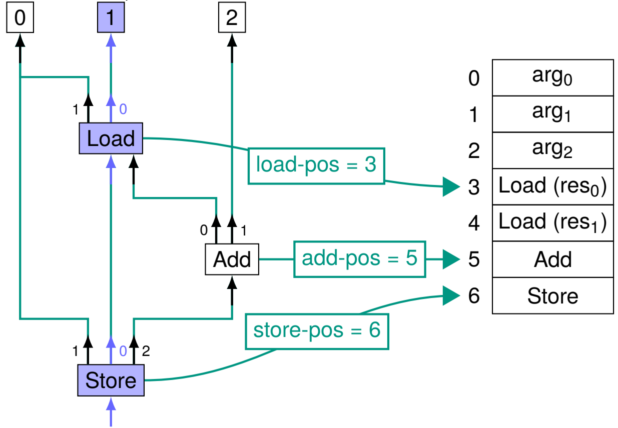
Gulwani et al., PLDI 2011



0	arg_0
1	arg_1
2	arg_2
3	
4	
5	
6	

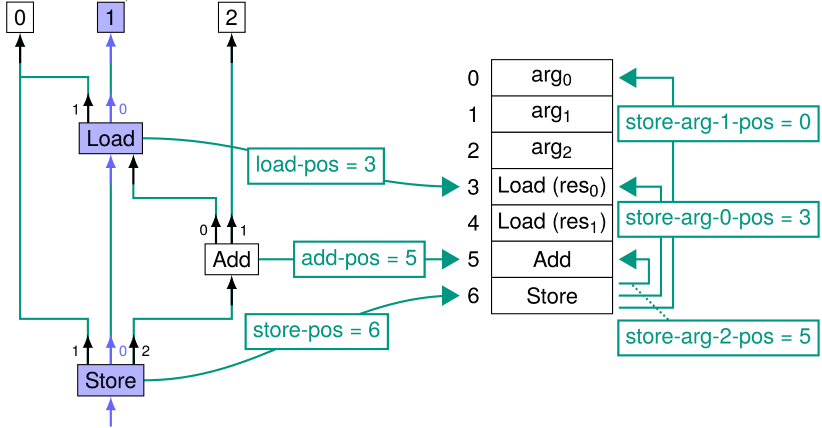
Component-Based Synthesis

Gulwani et al., PLDI 2011



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

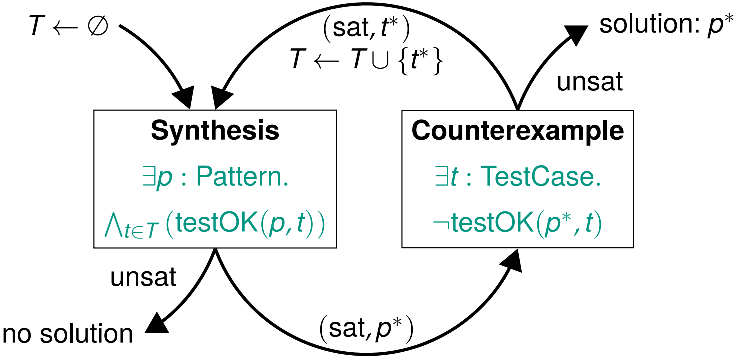
CEGIS →

Counterexample-Guided Inductive Synthesis

a. k. a. CEGIS

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

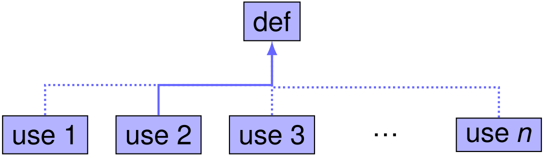
- Small set of test-cases T usually enough



linear→

Linear Type Encoding

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



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

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

opts→

Further Optimizations

- Load/Store/both necessary?

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

- $\geq d$ uses of a sort with d defs?
- Source (def without use) for all uses?

COV →

Instruction Coverage

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 %