Grisette: Symbolic Compilation as a Functional Programming Library

POPL 2023

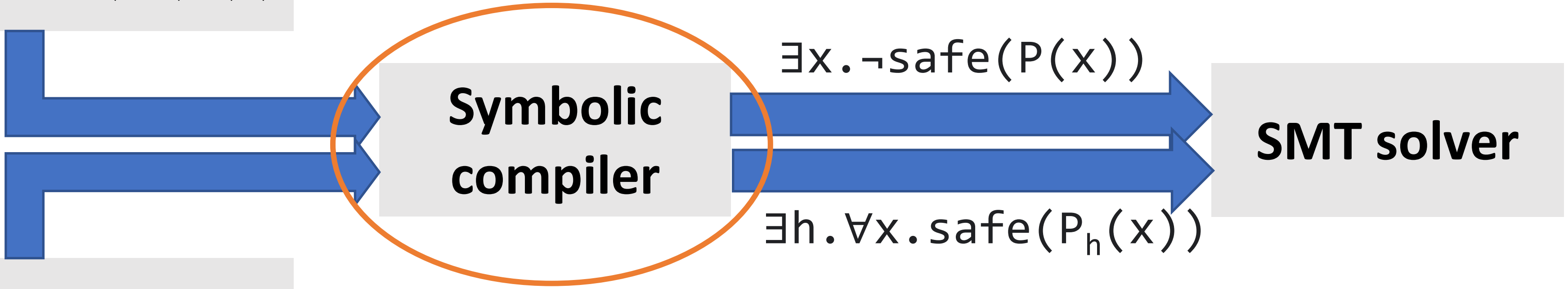
Sirui Lu (University of Washington)

Rastislav Bodík (Google Brain)

Symbolic compilation enables new tools

Verification
find a failing input

```
P(x) {
  ...
}
assert(safe(P(x)))
```



Synthesis
find code that meets the spec

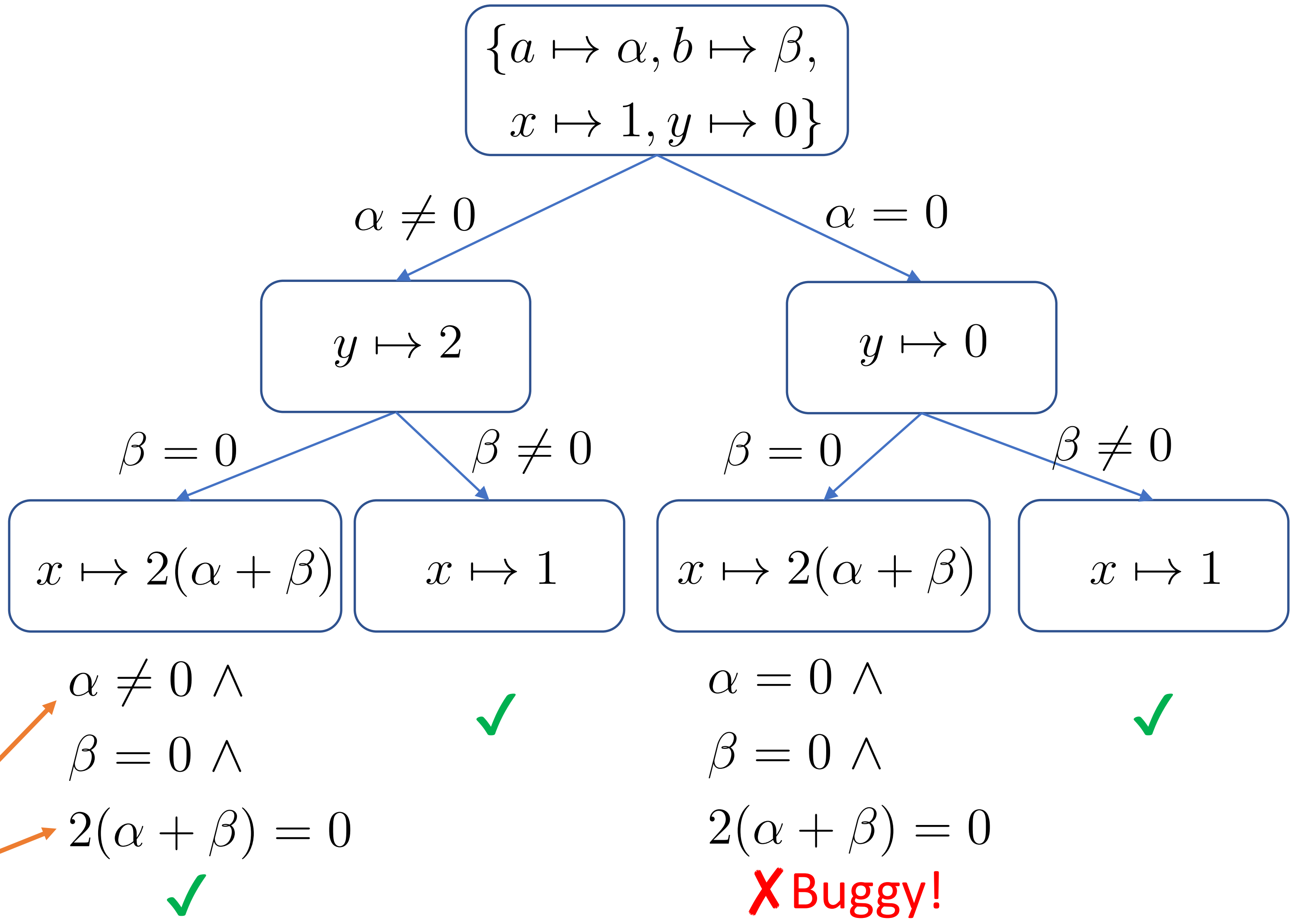
```
P(x) {
  v = h // hole
  ...
}
assert(safe(P(x)))
```

Ideally, we translate **all** program paths to a single formula

Credit to "Solver-Aided Programming for All," ICFP'19, Emina Torlak

Symbolic execution: path explosion but easy to solve

```
void f(int a, int b) {
  int x = 1, y = 0;
  if (a != 0) {
    y = x + 1;
  }
  if (b == 0) {
    x = 2 * (a + b);
  }
  assert (x != y);
}
```



Path condition

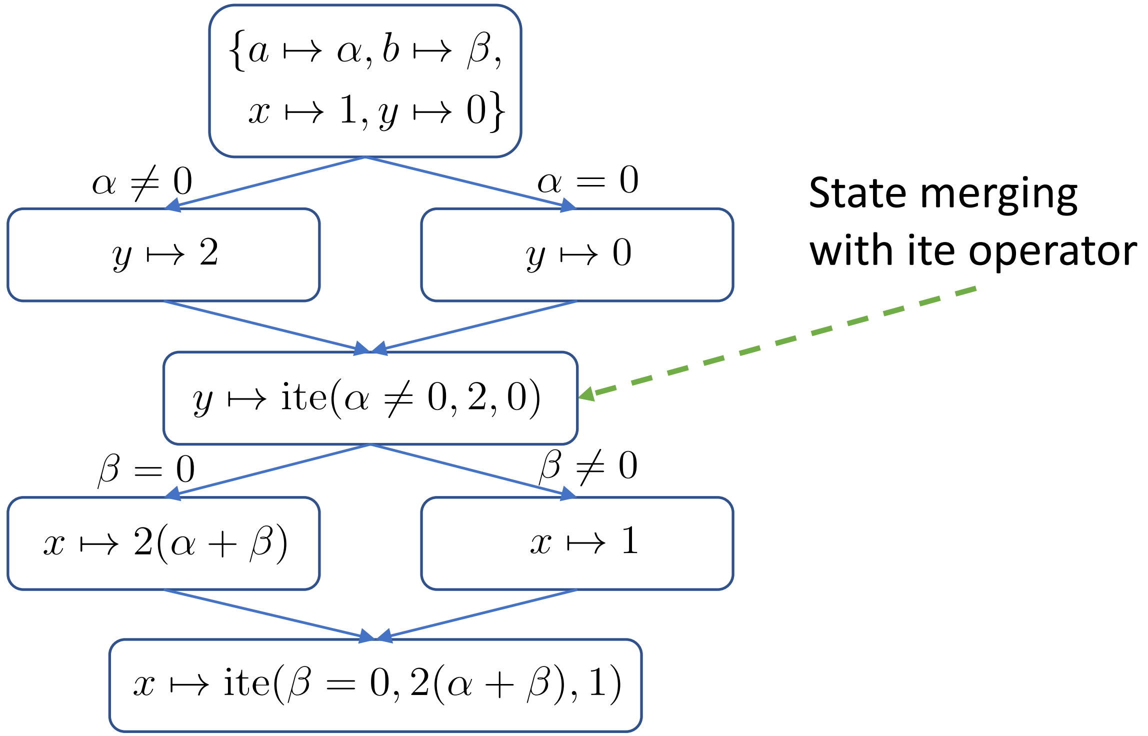
(negated) assertion

King-style symbolic execution (CACM 1976), Klee (OSDI'08), etc

Unsat means no bug

Bounded model checking: compact but harder to solve

```
void f(int a, int b) {
  int x = 1, y = 0;
  if (a != 0) {
    y = x + 1;
  }
  if (b == 0) {
    x = 2 * (a + b);
  }
  assert (x != y);
}
```



$$\text{ite}(\alpha \neq 0, 2, 0) = \text{ite}(\beta = 0, 2(\alpha + \beta), 1)$$

Satisfiable with $\{\alpha = 0, \beta = 0\}$

Bounded model checking,
CBMC (TACAS'04), Sketch (ASPLOS'04), etc

No path explosion, but results can be harder to solve (Kuznetsov et al., PLDI'12)

Our contributions

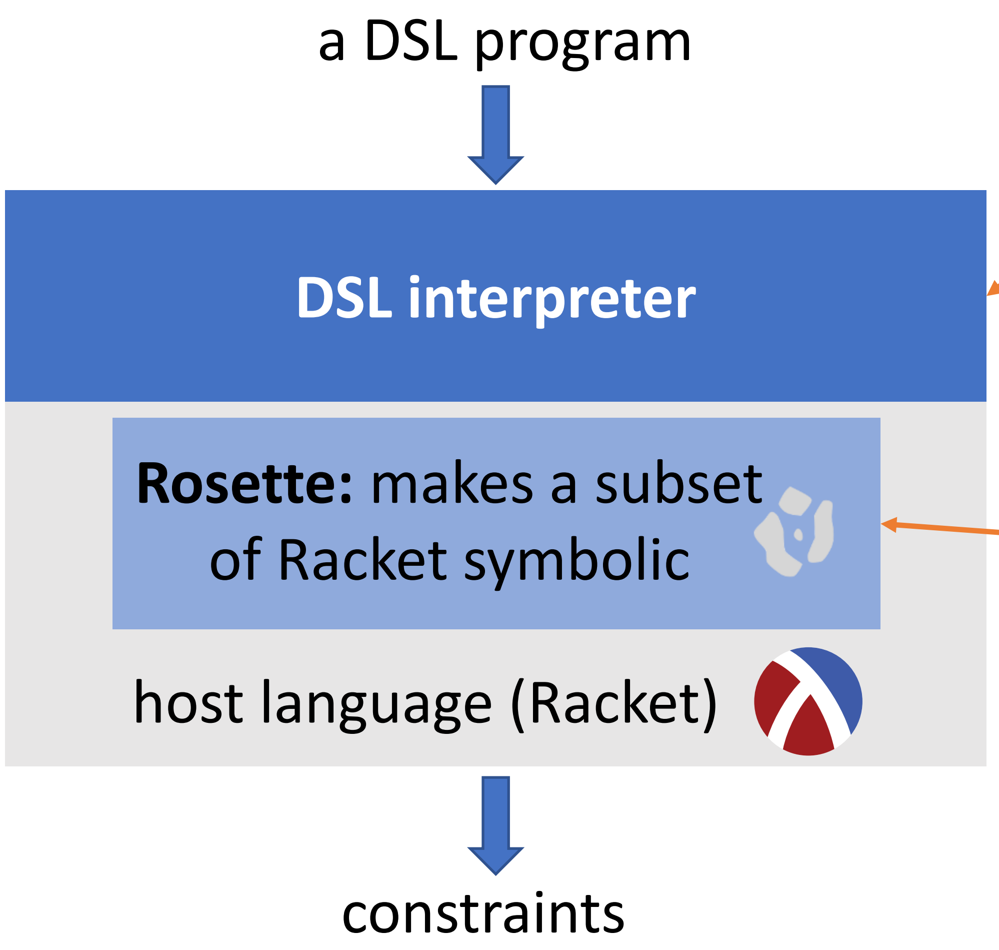
A new representation of symbolic values:

- Smaller formula.
- **3.7x** average speedup over the state of the art.
- Verified in Coq.

Symbolic compilation as a typed, purely functional library:

- A **reusable** symbolic compiler named Grisette.
- Open-source Haskell implementation.

Reusable symbolic compilers



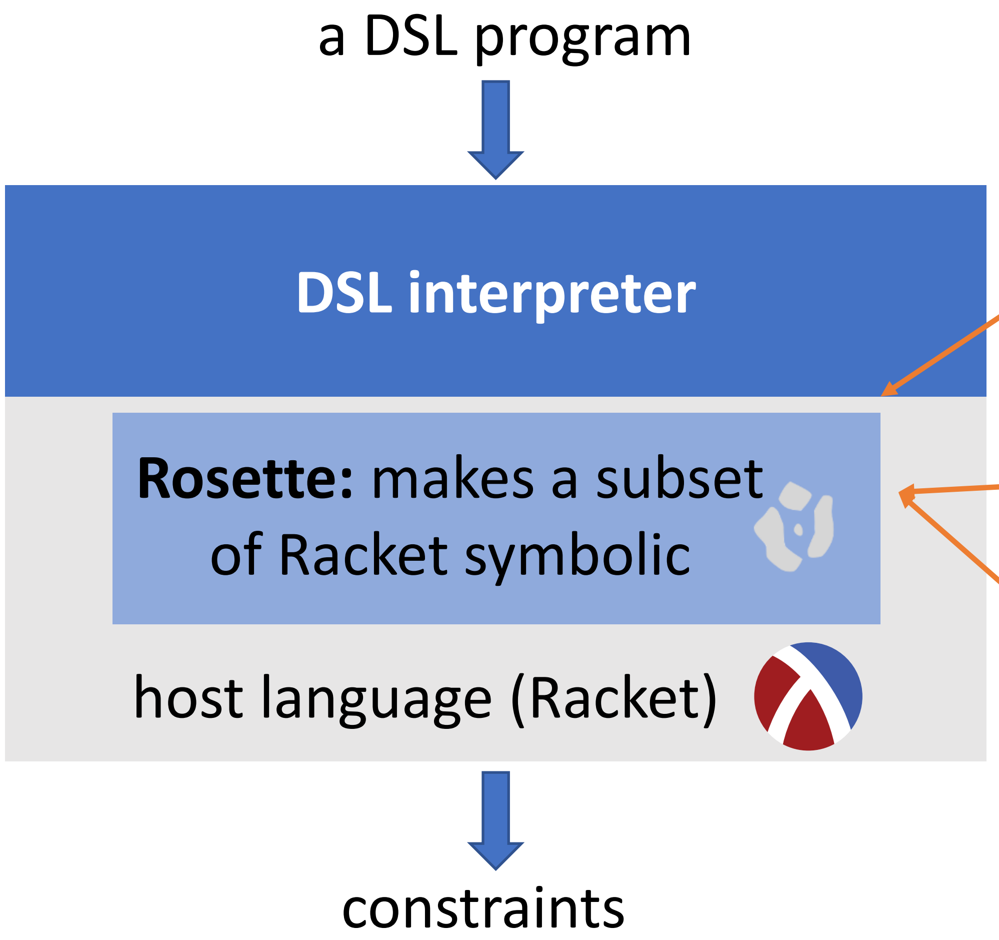
Rosette system (Onward!'13, PLDI'14, POPL'22)

Reusable: verification/synthesis tools for free with an interpreter, with Rosette's symbolic compiler reused

Efficiency: Novel symbolic representation balancing evaluation efficiency and formula quality.

Tools with Rosette: Cosette (CIDR'17), Ferrite (PLDI'17), Bonsai (POPL'18), Jitterbug (OSDI'20), …

Reusable symbolic compilers



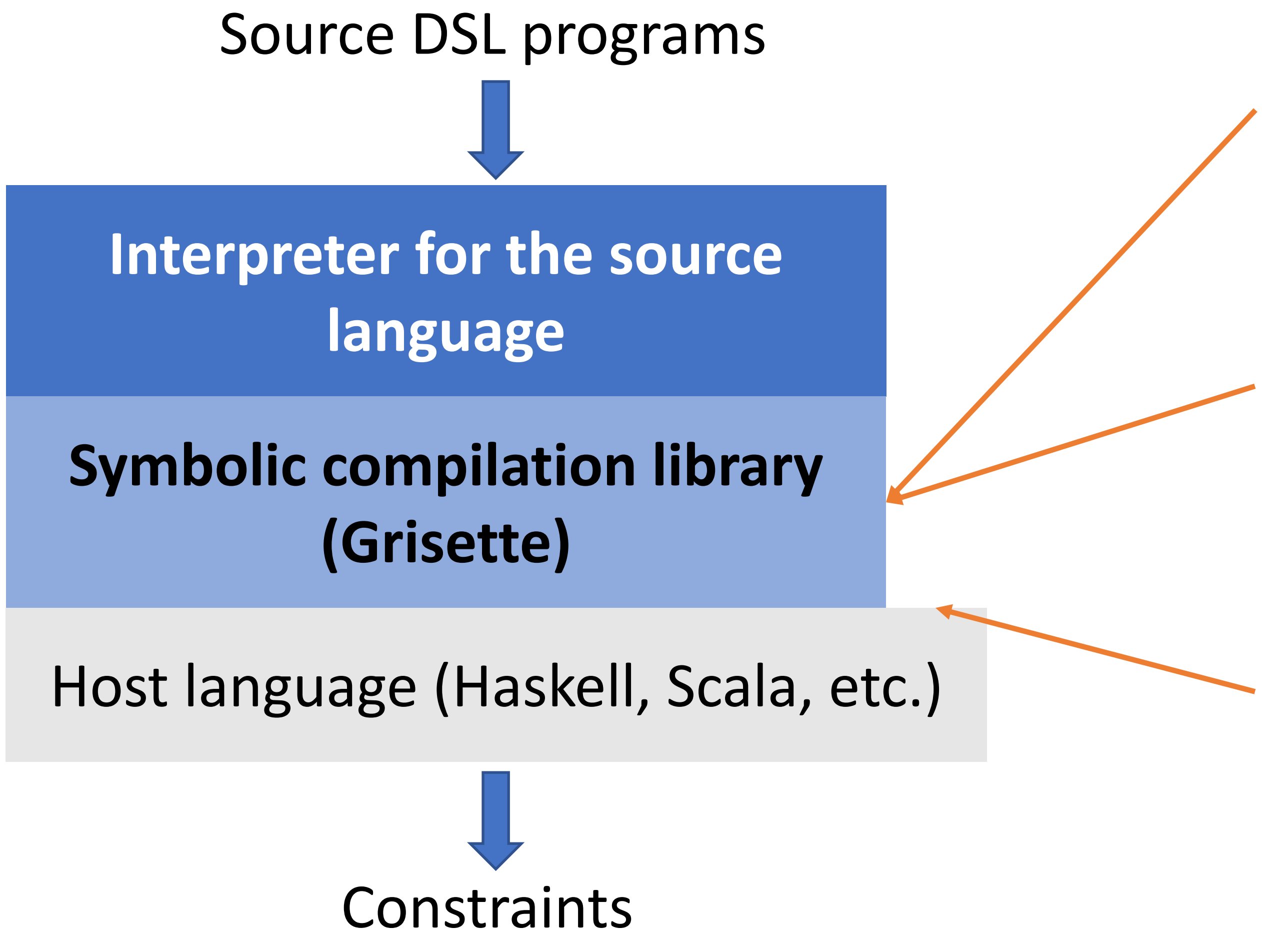
Confusion: What constructs accept symbolic values? Hard to debug.
(previous attempt: typed rosette, POPL'18)

Not easily portable: Rosette is a Racket-specific implementation.

Functional but not pure: assertions are compiled using a global state.
Want a small, purely functional core

Rosette system (Onward!'13, PLDI'14, POPL'22)

Reusable symbolic compiler as a library



Purely functional: Compose with widely available FP constructs for rich features, such as using Either for assertions

Portable: symbolic representation as a generic data structure.

Safe: Unsafe use of host language constructs are prohibited with types.

Challenges: to merge multiple paths functionally, we need a new symbolic representation.
(A pure version of Rosette's representation runs 14x slower than Rosette on one benchmark)

Design goals

We want a system that has good

- Efficiency (speed of compilation)
- Effectiveness (solvability of formulas)
- Usability (programming experience)

Outline

- The representation of symbolic values

  How good are the formulas created with a purely functional symbolic value?

- Empirical evaluation

- The programming interface

  What is the programming experience with a purely functional symbolic value?

Outline

- The representation of symbolic values
- Empirical evaluation
- The programming interface

An example program

```
a = if c1 then [x1] else [x2,x3]
b = if c2 then [x4] else [x5,x6]
d = a ++ b
e = head d
```

Symbolic values are in orange

Complex values, e.g., lists

Operations on lists

MEG (Mutually Exclusive Guards)

```
a = if c1 then [x1] else [x2,x3]
b = if c2 then [x4] else [x5,x6]
d = a ++ b
e = head d
```

7 new nodes created

$$\mathbf{d_{MEG}} = \begin{cases} [x1,x4] & \text{if } c1 \wedge c2 \\ [\text{ite}(\text{cond1}, x1, x2), \cdots] & \text{if } (c1 \wedge \neg c2) \vee (c2 \wedge \neg c1) \\ [x2,x3,x5,x6] & \text{if } \neg c1 \wedge \neg c2 \end{cases}$$

Problem 1:
mutual exclusiveness duplicates conditions

head can be applied to the branches

MEG is the key design for Rosette and MultiSE (FSE'15) to support advanced features easily

ORG (ORdered Guards)

```
a = if c1 then [x1] else [x2,x3]
b = if c2 then [x4] else [x5,x6]
d = a ++ b
e = head d
```

7 new nodes created

$$d_{MEG} = \begin{cases} [x1,x4] & \text{if } c1 \wedge c2 \\ [\text{ite}(\text{cond1}, x1, x2), \cdots] & \text{if } (c1 \wedge \neg c2) \vee (c2 \wedge \neg c1) \\ [x2,x3,x5,x6] & \text{if } \neg c1 \wedge \neg c2 \end{cases}$$

Problem 1:
mutual exclusiveness duplicates conditions

$$d_{ORG} = \begin{cases} [x1,x4] & \text{if } c1 \wedge c2 \\ [\text{ite}(\text{cond1}, x1, x2), \cdots] & \text{else if } c1 \vee c2 \\ [x2,x3,x5,x6] & \text{otherwise} \end{cases}$$

If/else if/otherwise

Insight 1: guards can be ordered and implicitly mutually exclusive with smaller terms

Sources of duplications in MEG vs. ORG

`if c4 then 4 else if c3 then 3 else if c2 then 2 else 1`

MEG | **ORG**

Original

MEG:

$$\begin{cases} 4 & \text{if} \quad \mathtt{c4} \\ 3 & \text{if} \quad \mathtt{c3} \wedge \neg \mathtt{c4} \\ 2 & \text{if} \quad \mathtt{c2} \wedge \neg \mathtt{c3} \wedge \neg \mathtt{c4} \\ 1 & \text{if} \quad \neg \mathtt{c2} \wedge \neg \mathtt{c3} \wedge \neg \mathtt{c4} \end{cases}$$

ORG:

$$\begin{cases} 4 & \text{if} & \mathtt{c4} \\ 3 & \text{else if} & \mathtt{c3} \\ 2 & \text{else if} & \mathtt{c2} \\ 1 & \text{otherwise} & \end{cases}$$

MEG: duplication when making guards **disjoint**

Reordered

MEG:

$$\begin{cases} 1 & \text{if} \quad \neg \mathtt{c2} \wedge \neg \mathtt{c3} \wedge \neg \mathtt{c4} \\ 2 & \text{if} \quad \mathtt{c2} \wedge \neg \mathtt{c3} \wedge \neg \mathtt{c4} \\ 3 & \text{if} \quad \mathtt{c3} \wedge \neg \mathtt{c4} \\ 4 & \text{if} \quad \mathtt{c4} \end{cases}$$

ORG:

$$\begin{cases} 1 & \text{if} & \neg \mathtt{c2} \wedge \neg \mathtt{c3} \wedge \neg \mathtt{c4} \\ 2 & \text{else if} & \neg \mathtt{c3} \wedge \neg \mathtt{c4} \\ 3 & \text{else if} & \neg \mathtt{c4} \\ 4 & \text{otherwise} & \end{cases}$$

ORG: duplication caused by **reordering**

Merging two ORG containers

`if cond then a else b`

How to merge?

$$\begin{cases} 1 & \textbf{if} \quad \texttt{cond1} \\ 3 & \textbf{else if} \quad \texttt{cond2} \\ 2 & \textbf{otherwise} \end{cases} \qquad \begin{cases} 2 & \textbf{if} \quad \texttt{cond3} \\ 1 & \textbf{else if} \quad \texttt{cond4} \\ 3 & \textbf{otherwise} \end{cases}$$

reorder to align values

$$\begin{cases} 1 & \textbf{if} \quad \texttt{cond1} \\ 3 & \textbf{else if} \quad \texttt{cond2} \\ 2 & \textbf{otherwise} \end{cases} \qquad \begin{cases} 1 & \textbf{if} \quad \neg\texttt{cond3} \land \texttt{cond4} \\ 3 & \textbf{else if} \quad \neg\texttt{cond4} \\ 2 & \textbf{otherwise} \end{cases}$$

merged result =

$$\begin{cases} 1 & \textbf{if} \quad \mathrm{ite}(\texttt{cond}, \texttt{cond1}, \neg\texttt{cond3} \land \texttt{cond4}) \\ 3 & \textbf{else if} \quad \mathrm{ite}(\texttt{cond}, \texttt{cond2}, \neg\texttt{cond4}) \\ 2 & \textbf{otherwise} \end{cases}$$

Merging two sorted ORG container

Problem 2: we need to reorder and create complex conditions every time we merge
Insight 2: we can keep ORG containers sorted, and reduce the need for reordering

Sortedness is a representation invariant => Further merging avoids reordering

Merging complex types in ORG

Problem 3: merging is inefficient when ORG containers are big
Insight 3: use hierarchical encoding to allow sub-containers to be treated atomically when the values are complex

Direct generalization

$$\begin{cases} \texttt{(1,1,u)} & \text{if} & \texttt{c1} \\ \texttt{(1,2,v)} & \text{else if} & \texttt{c2} \\ \texttt{(2,3,w)} & \text{else if} & \texttt{c3} \\ \texttt{(2,4,x)} & \text{else if} & \texttt{c4} \\ \texttt{(9,2,y)} & \text{otherwise} & \end{cases}$$

Hierarchical encoding

$$\begin{cases} t_1 & \text{if} & \texttt{c1} \\ t_2 & \text{else if} & \texttt{c2} \\ \texttt{(9,2,y)} & \text{otherwise} & \end{cases}$$

$$t_1 = \begin{cases} \texttt{(1,1,u)} & \text{if} & \texttt{c11'} \\ \texttt{(1,2,v)} & \text{otherwise} & \end{cases}$$

$$t_2 = \begin{cases} \texttt{(2,3,w)} & \text{if} & \texttt{c21'} \\ \texttt{(2,4,x)} & \text{otherwise} & \end{cases}$$

Preserves worst-case linear-time in # of symbolic values in ORG (proven with Coq)

Outline

- The representation of symbolic values
- Empirical evaluation
- The programming interface

Empirical evaluation

RQ1: is Grisette more efficient than the state-of-the-art?

- evaluation time (symbolic compilation)
- solving time

RQ2: why do Grisette's constraints solve faster?

Evaluation settings

Four symbolic compilation systems:

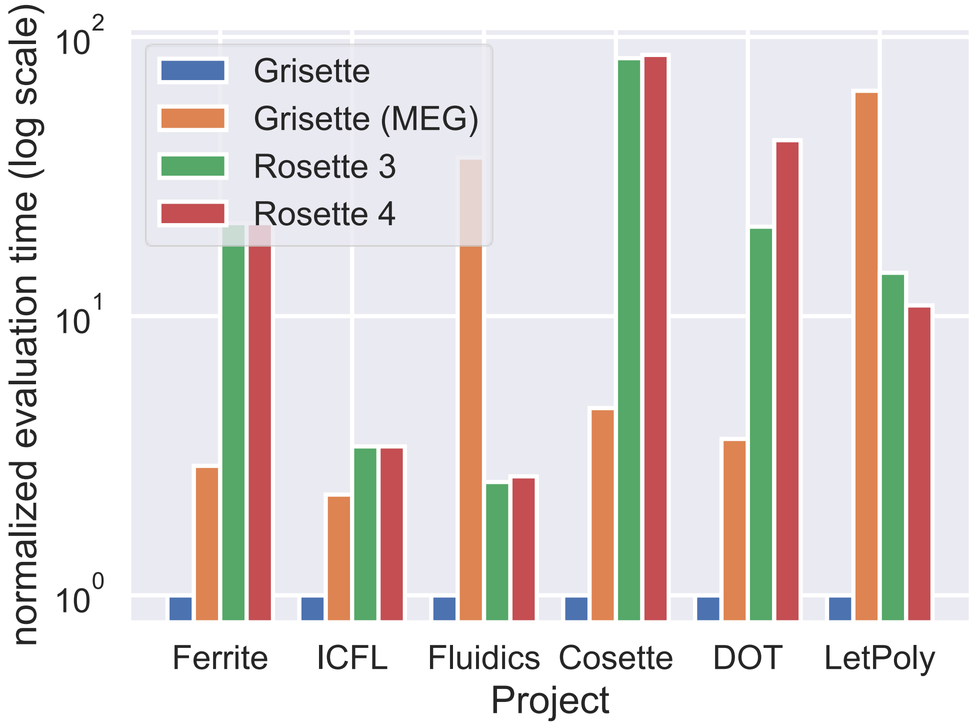
- Grisette with **ORG**
- Grisette with functional **MEG** (i.e., assertions propagated, not in global state)
- Rosette 3 (pre-POPL'22)
- Rosette 4 (post-POPL'22)

Five Rosette-based tools (six benchmarks) ported to Grisette:

- **Ferrite (ASPLOS'16):** file system crash model verifier and sync call synthesizer
- **IFCL (PLDI'14):** information flow control verification and synthesizer
- **Fluidics (ASPLOS'19):** microfluidics manipulation program synthesizer
- **Cosette (CIDR'17):** SQL equivalence checker
- **Bonsai (POPL'17) for DOT (scala) & LetPoly:** type system soundness checker

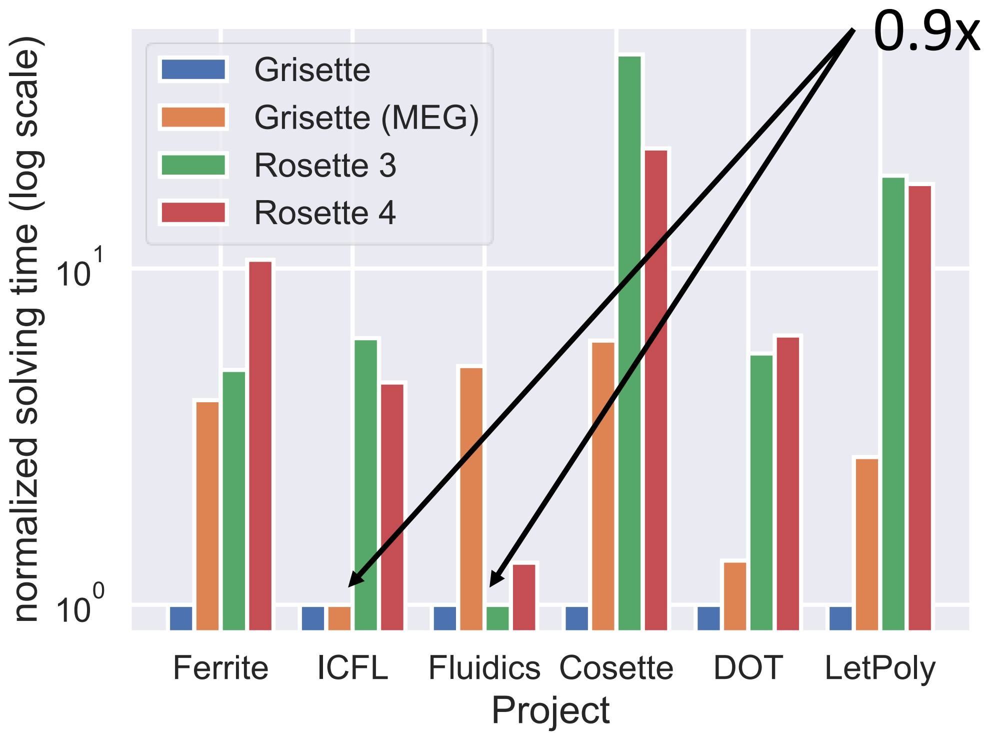
RQ1: Grisette is more efficient than SOTA, in both compilation and solving time

Compilation



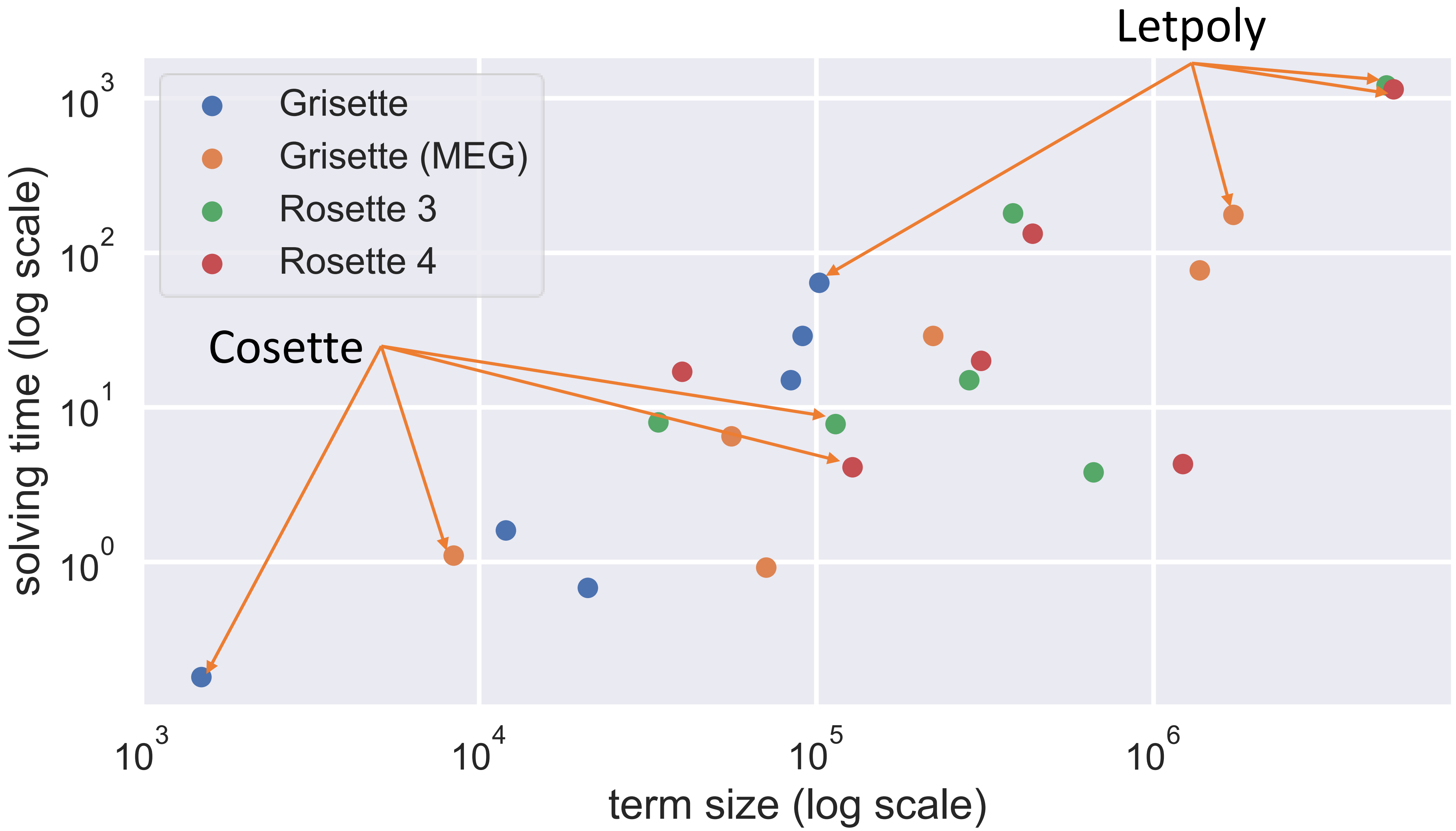
6.1x over Grisette (MEG)
13.0x over Rosette 3
14.1x over Rosette 4

Solving



2.4x over Grisette (MEG)
5.5x over Rosette 3
5.7x over Rosette 4

RQ2: faster solving can be a result of smaller terms



Outline

- The representation of symbolic values
- Empirical evaluation
- The programming interface

A minimal synthesizer

Program space: synthesize a function `\x -> x + ` ***c***

Some example programs in the space: `\x -> x + 1`,
`\x -> x + 2`

Specification: I/O pair (2, 5)

Expect result: `\x -> x + 3`

A minimal synthesizer

UnionM is an ORG container representing a symbolic set of expressions

```
data SymExpr -- Symbolic candidate program space
  = SIntValue SymInteger
  | SAdd (UnionM SymExpr) (UnionM SymExpr)
  | SMul (UnionM SymExpr) (UnionM SymExpr)
  deriving ...
```

Define DSL syntax

```
interpret :: SProgram -> SymInteger
interpret (SIntValue c) = c
interpret (SAdd x y) = interpretU x + interpretU y
interpret (SMul x y) = interpretU x * interpretU y

interpretU :: UnionM Sprogram -> SymInteger
interpretU = onUnion interpret
```

DSL interpreter.
Interprets simultaneously all ASTs in the space.

onUnion lifts 'interpret' to ORGs of ASTs

A minimal synthesizer

A symbolic integer variable to be solved

```
programSpace :: SymInteger -> SymExpr
programSpace x = SAdd (return x) (return "c")
```

Define the program space \x -> x + c

```
executableProgramSpace :: Integer -> SymInteger
executableProgramSpace = interpret . programSpace . toSym
```

Make the program space executable

```
quickExample :: IO ()
quickExample = do
  let constraint = executableProgramSpace 2 ==~ 5
  Right model <- solve solverConfig constraint
```

Call the solver with I/O pair (2,5)

```
  print $ evaluateSym False model (programSpace "x")
  -- SMul {SIntValue x} {SIntValue 3}
```

Print the *synthesized* program \x -> x + 3

```
  let synthesizedProgram :: Integer -> Integer =
        evaluateSymToCon model . executableProgramSpace
  print $ synthesizedProgram 20 -- 60
```

Get *concrete* synthesized program

Discussion 1: Stateful programming with Grisette

Monadic ORG container

UnionM operations are generalized with type classes.

```
StateT state UnionM val
```

Example:

StateT requires **~30** lines of code.

Want an imperative DSL? Write a StateT-based interpreter.

Free monad + combinators and functors for trampolines (~250 lines of code)

mtl transformers (mostly <30 lines of code each)

Want coroutines? Use trampolines or delimited continuations.

Discussion 2: Additional benefits

Static types:

- constrain the symbolic representation for performance tuning
- ex: improved Cosette performance for an additional 8.7x speedup

Purely functional:

- memoization (1.2 – 7.5x compilation speed up on 4 projects)
- parallelization seems also possible

See the paper for more details

Grisette: Symbolic Compilation as a Functional Programming Library

https://github.com/lsrcz/grisette

Thanks!

https://hackage.haskell.org/package/grisette

Sirui Lu
Rastislav Bodík