Synthesis of Recursive ADT Transformations from Reusable Templates

Jeevana Inala
MIT

Nadia Polikarpova
Armando Solar-Lezama
(MIT)

Xiaokang Qiu
(Purdue)

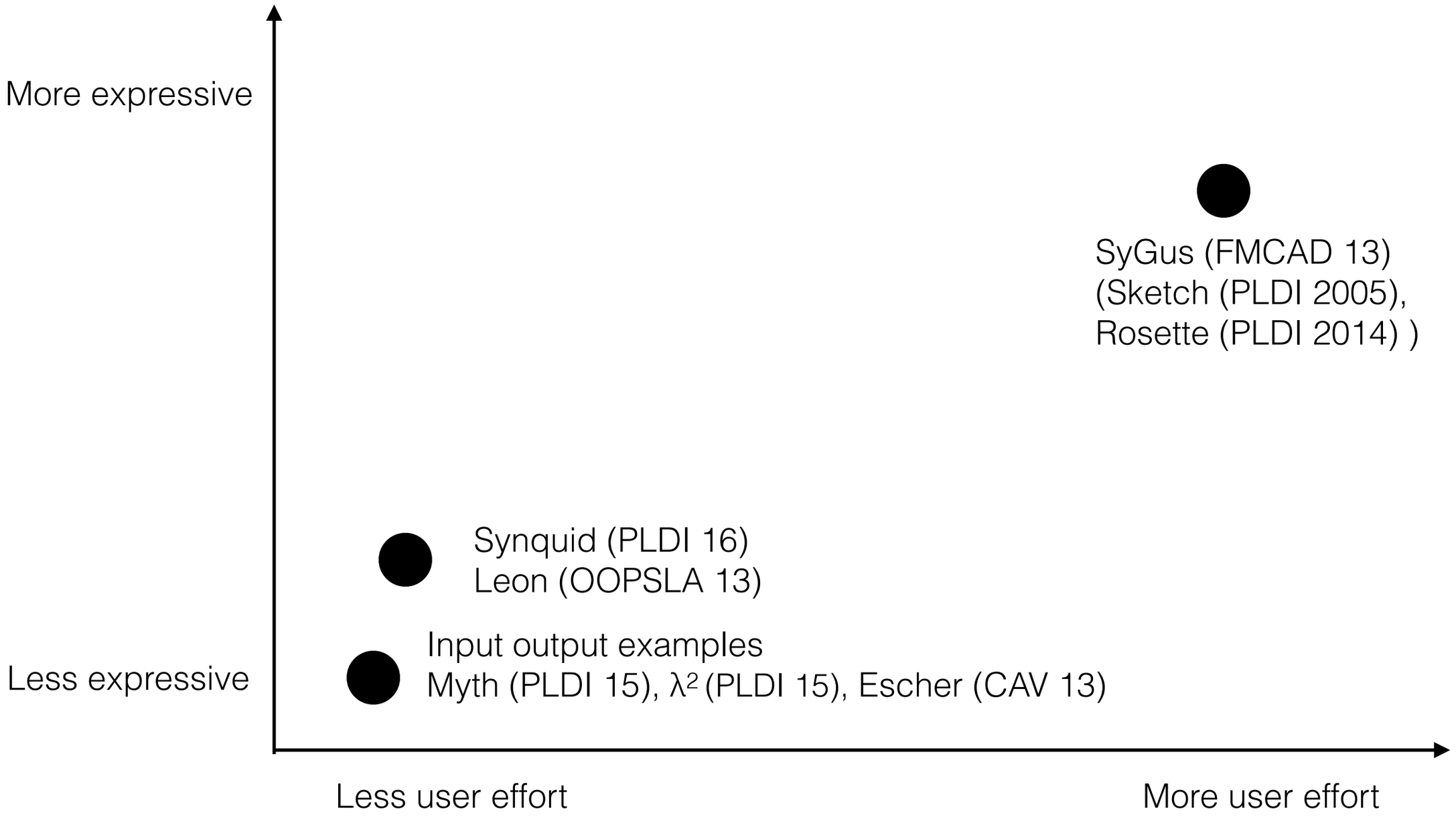
Ben Lerner
(Northeastern)

MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

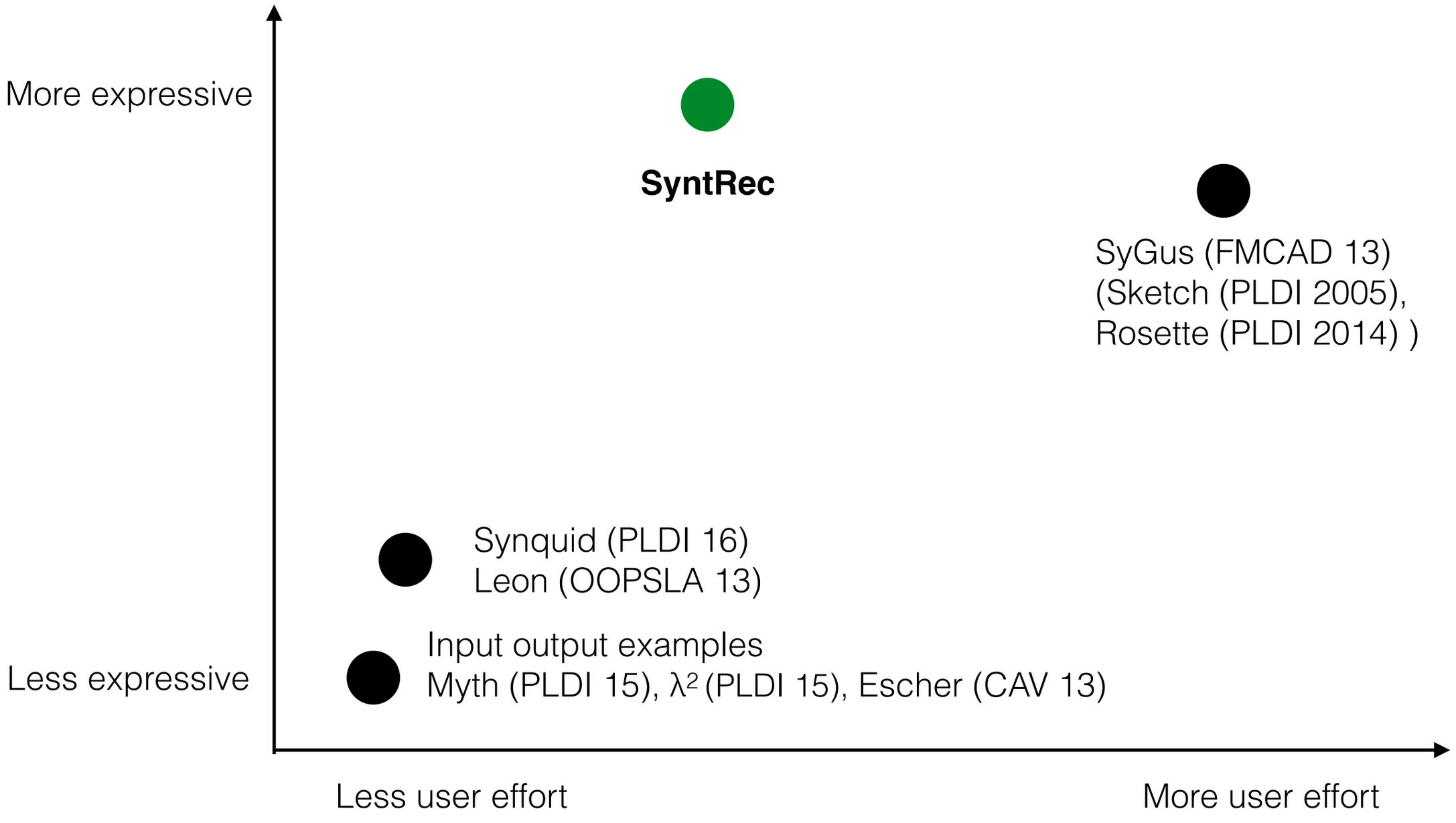
CSAIL

Synthesis Approaches

Synthesis Approaches



Synthesis Approaches



Recursive ADT Transformations

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
 }
```

```
adt dstAST {
  NumD { int v; }
  BoolD { bit v; }
  BinaryD { opcode op; dstAST a; dstAST b; }
}
```

Recursive ADT Transformations

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
 }
```

```
adt dstAST {
  NumD { int v; }
  BoolD { bit v; }
  BinaryD { opcode op; dstAST a; dstAST b; }
}
```

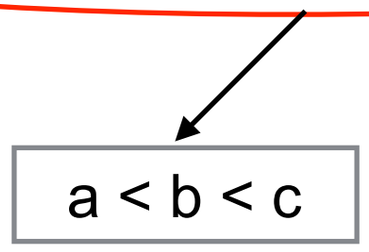
Synthesize

dstAST desugar(srcAST s) { …. }

Recursive ADT Transformations

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
}
```

```
adt dstAST {
  NumD { int v; }
  BoolD { bit v; }
  BinaryD { opcode op; dstAST a; dstAST b; }
}
```



Synthesize

dstAST desugar(srcAST s) { …. }

Specification

interpretS(s) == interpretD(desugar(s))

Template in Sketch

dstAST desugar (srcAST src)

Template in Sketch

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:

    case TrueS:

    case FalseS:

    case BinaryS:

    case BetweenS:
```

Template in Sketch

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:

    case TrueS:

    case FalseS:

    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)

    case BetweenS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      dstAST c = desugar(src.c)
```

Template in Sketch

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:

    case TrueS:

    case FalseS:

    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)

    case BetweenS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      dstAST c = desugar(src.c)
```

```
generator dstAST rcons(fun e)
  if (??)
    return e()
  if (??)
    int val = choose(e(), ??)
    return new NumD(v = val)
  if (??)
    bit val = choose(e(), ??)
    return new BoolD(v = val)
  if (??)
    dstAST a = rcons(e)
    dstAST b = rcons(e)
    opcode op = choose(e(), ??)
    return new BinaryD(op = op, a= a, b = b)
```

Template in Sketch

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
  case BinaryS:
    dstAST a = desugar(src.a)
    dstAST b = desugar(src.b)
    return rcons(()→choose(a, b, src .op))
  case BetweenS:
    dstAST a = desugar(src.a)
    dstAST b = desugar(src.b)
    dstAST c = desugar(src.c)
    return rcons(()→choose(a, b, c ))
```

```
generator dstAST rcons(fun e)
  if (??)
    return e()
  if (??)
    int val = choose(e(), ??)
    return new NumD(v = val)
  if (??)
    bit val = choose(e(), ??)
    return new BoolD(v = val)
  if (??)
    dstAST a = rcons(e)
    dstAST b = rcons(e)
    opcode op = choose(e(), ??)
    return new BinaryD(op = op, a= a, b = b)
```

Template in Sketch

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
case BinaryS:
  dstAST a = desugar(src.a)
  dstAST b = desugar(src.b)
  return rcons(()→choose(a, b, src .op))
case BetweenS:
  dstAST a = desugar(src.a)
  dstAST b = desugar(src.b)
  dstAST c = desugar(src.c)
  return rcons(()→choose(a, b, c ))
```

```
generator dstAST rcons(fun e)
  if (??)
    return e()
  if (??)
    int val = choose(e(), ??)
    return new NumD(v = val)
  if (??)
    bit val = choose(e(), ??)
    return new BoolD(v = val)
if (??)
  dstAST a = rcons(e)
  dstAST b = rcons(e)
  opcode op = choose(e(), ??)
  return new BinaryD(op = op, a= a, b = b)
```

No. of possible functions from template ~ 2^{110}

Template in Sketch

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      return rcons(()→choose
  case BetweenS:
    dstAST a = desugar(src.a)
    dstAST b = desugar(src.b)
    dstAST c = desugar(src.c)
    return rcons(()→choose(a, b, c ))
```

Verbose

Not reusable

```
generator dstAST rcons(fun e)
  if (??)
    return e()
  if (??)
    int val = choose(e(), ??)
    return new NumD(v = val)
  (??)
    it val = choose(e(), ??)
    eturn new BoolD(v = val)
  if (??)
    dstAST a = rcons(e)
    stAST b = rcons(e)
    pcode op = choose(e(), ??)
    return new BinaryD(op = op, a= a, b = b)
```

No. of possible functions from template ~ 2^{110}

Reusable Templates

- ***Polymorphic synthesis constructs*** to support general operations on ADTs

Reusable Templates

```
dstAST desugar (srcAST src)
    return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer)
```

Polymorphic Generator

Reusable Templates

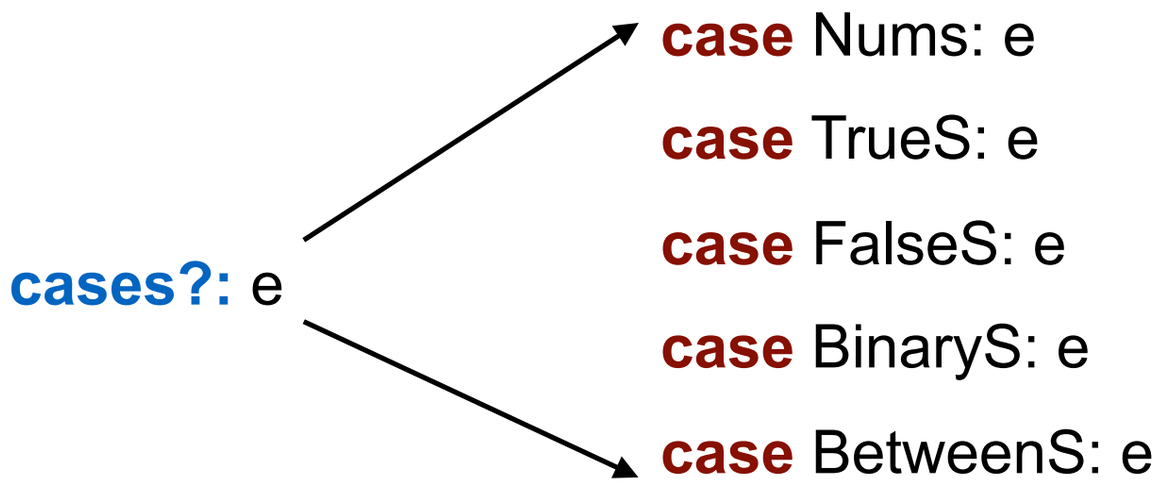
```
dstAST desugar (srcAST src)
  return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer)
  switch(src)
    cases?:
```

Flexible pattern matching

src :: srcAST



Reusable Templates

```
dstAST desugar (srcAST src)
  return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer)
  switch(src)
    cases?:
      P[ ] a = map(src.fields?, replacer )
```

Fields list

```
case BinaryS:
  dstAST[ ] a = map(src.fields?, desugar )
```

src :: BinaryS{opcode op;
srcAST a; srcAST b;}

src.fields? ⟶ { src.a, src.b }

src.fields? :: srcAST[]

Reusable Templates

```
dstAST desugar (srcAST src)
  return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer)
  switch(src)
    cases?:
      P[ ] a = map(src.fields?, replacer )
```

Fields list

```
case NumS:
  dstAST[ ] a = map(src.fields?, desugar )
```

src :: NumS{ int v; }

src.fields? :: srcAST[]

src.fields? ⟶ { }

Reusable Templates

```
dstAST desugar (srcAST src)
  return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer)
  switch(src)
  cases?:
    P[ ] a = map(src.fields?, replacer )
    return rcons(()→choose(a[??], field ( src )))
```

Reusable Templates

```
dstAST desugar (srcAST src)
  return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer)
  switch(src)
    cases?:
      P[ ] a = map(src.fields?, replacer )
      return rcons(()→choose(a[??], field ( src )))

generator T field <T,S> (S e)
  return (e.fields?)[??];
```

Reusable Templates

```
dstAST desugar (srcAST src)
  return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator T rcons <T> (fun e)
  if (??)
    return e();
  else
    return new cons?(rcons(e));
```

Unknown Constructor

T = dstAST

```
cons?(rcons(e)) ⟶
```

```
choose( new NumD(v = rcons(e)),
        ….,
        new BinaryD(op = rcons(e),
          a = rcons(e), b = rcons(e)))
```

Polymorphic Synthesis Constructs (PSCs)

- Polymorphic generators
- Flexible pattern matching

  switch (e_1) **case?** e_2

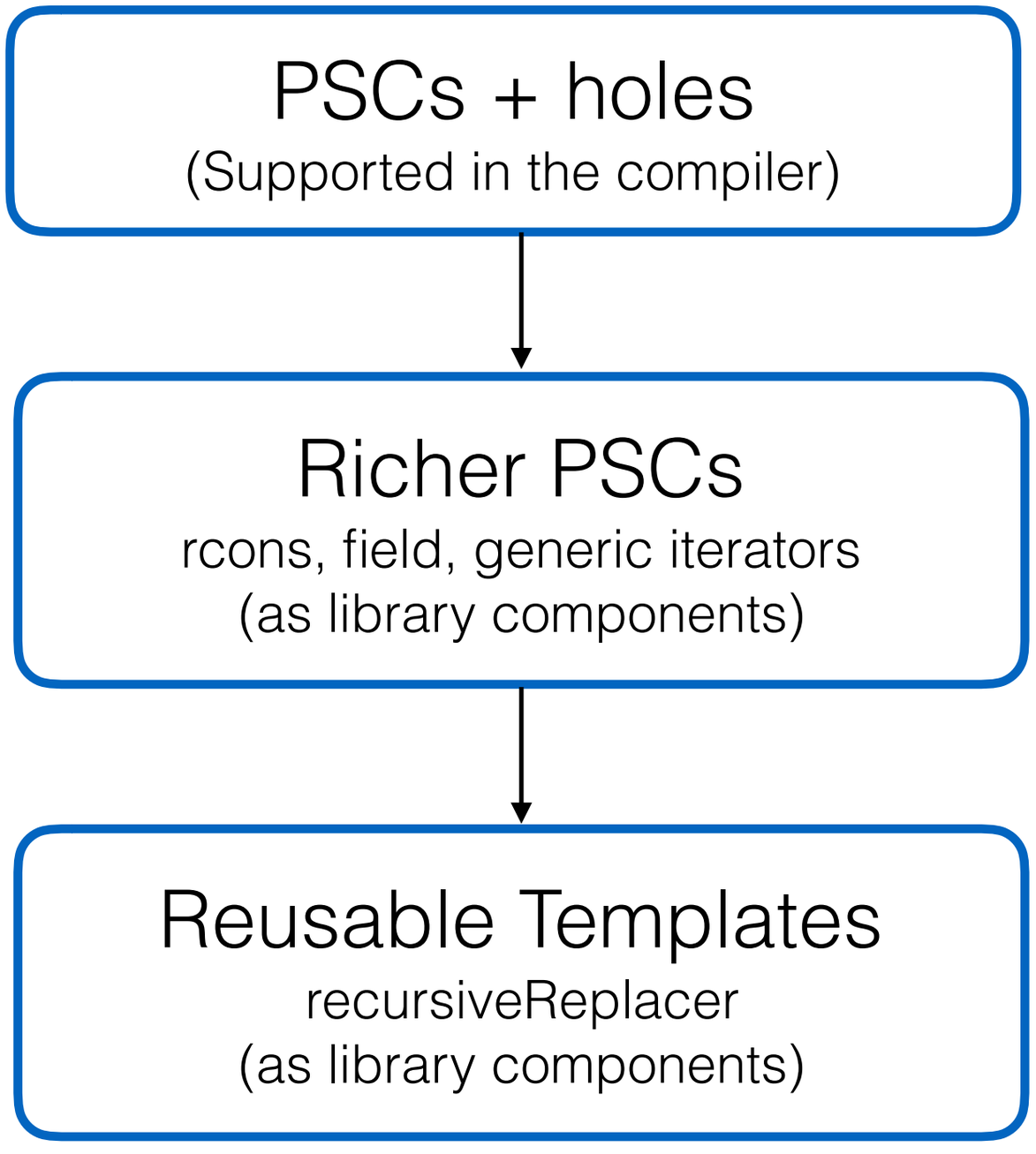
- Fields list

  e.**fields?**

- Unknown constructor

  cons?(e_1, e_2, … e_k)

Combinations of PSCs



Reusable Templates

```
dstAST desugar (srcAST src)
  return recursiveReplacer ( src , desugar)
```

P = dstAST
Q = srcAST
replacer = desugar

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer)
  switch(src)
  cases?:
    P[ ] a = map(src.fields?, replacer )
    return rcons(()→choose(a[??], field ( src )))
```

Concise

Reusable

Reusable Templates

Synthesized program for desugar

```
dstAST desugar (srcAST src)
  switch (src)
     case NumS: return new NumD(v = src.v)
     case TrueS: return new BoolD(v = 1)
     case FalseS: return new BoolD(v = 0)
     case BinaryS:
     dstAST[2] a = {desugar(src.a), desugar(src .b)}
     return new BinaryD(op = src.op, a = a[1],  b = a [2])
  case BetweenS:
     dstAST[3] a = {desugar(src.a), desugar(src .b), desugar(src .c)}
     return new BinaryD(op = new AndOp(),
           a = new BinaryD(op = new LtOp(), a = a[0],  b = a[1]),
           b = new BinaryD(op = new LtOp(), a = a[1],  b = a [2]))
```

Synthesis Time: 8s

Reusable Templates

Synthesized program for desugar

```
dstAST desugar (srcAST src)
  switch (src)
     case NumS: return new NumD(v = src.v)
     case TrueS: return new BoolD(v = 1)
     case FalseS: return new BoolD(v = 0)
     case BinaryS:
     dstAST[2] a = {desugar(src.a), desugar(src .b)}
     return new BinaryD(op = src.op, a = a[1],  b = a [2])
  case BetweenS:
     dstAST[3] a = {desugar(src.a), desugar(src .b), desugar(src .c)}
     return new BinaryD(op = new AndOp(),
           a = new BinaryD(op = new LtOp(), a = a[0],  b = a[1]),
           b = new BinaryD(op = new LtOp(), a = a[1],  b = a [2]))
```

Synthesis Time: 8s

Reusable Templates

Church encoding for Boolean operations

```
adt Bool {
  True { }
  False { }
  And { Bool a; Bool b; }
  Or { Bool a; Bool b; }
  Not { Bool a; }
}
```

```
adt E {
  Var { int v; }
  Abs { Var v; E a  }
  App { E a; E b; }
}
```

```
E desugar (Bool src)
    return recursiveReplacer ( src , desugar)
```

Reusable Templates

Synthesized program for Church encodings for Boolean operations

```
E desugar (Bool src)
  switch (src)
    case True: return new Abs(new Var(0), new Abs(new Var(1), new
   Var(0)))
    case False: return new Abs(new Var(0), new Abs(new Var(1), new
   Var(1)))
 case And:
    E[2] a = { desugar(src.a),  desugar(src.b) }
    return new App(new App(a[0], a[1]), a[0])
 case Or:
    E[2] a = { desugar(src.a), desugar(src.b) }
    return new App(new App(a[0], a[0]), a[1])
 case Not:
    E[1] a = { desugar(src.a) }
    return new App(new App(a[0], new Abs(new Var(0), new Abs(new
    Var(1), new Var(0)))), new Abs(new Var(0), new Abs(new Var(1),
    new Var(1))))
```

Synthesis Time: 43s

Reusable Templates

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer, fun condGen, fun fieldGen)
  switch(src)
    cases?:
      P[ ] a = map(src.fields?, replacer )
      repeat
       if (condGen( field(src) )
        return rcons(()→choose(a[??], field ( src ), fieldGen()))
```

Reusable Templates

```
generator P recursiveReplacer <P, Q> (Q src, fun replacer, fun condGen, fun fieldGen)
  switch(src)
    cases?:
      P[ ] a = map(src.fields?, replacer )
      repeat
        if (condGen( field(src) )
          return rcons(()→choose(a[??], field ( src ), fieldGen()))
```

```
dstAST desugar (srcAST src)
   return recursiveReplacer ( src , desugar, ()→true, ()→0)
```

```
Btree insert (Btree tree, int x)
   return recursiveReplacer ( tree , (t)->insert(t, x),
                                     (v)→choose(v <= x, v > x, true),
                                     ()->x)
```

Reusable Templates

Synthesized program for binary tree insertion

```
BTree insert (BTree tree, int x)
  switch (src)
      case Branch:
          BTree[2] a = {insert(tree.left, x), insert(tree.right, x)}
          if (tree.val <= x)
            return new BTree(tree.val, a[0], tree.right)
      if (tree.val > x)
        return new BTree(tree.val, tree.left, a[1])
case Leaf:
      if (tree.val <= x)
        return new BTree(x, new Leaf(tree.v), new Empty())
      if (tree.val > x)
        return new BTree(x, new Empty(), new Leaf(tree.v))
case Empty:
      return new Leaf(x)
```

Synthesis Time: 20s

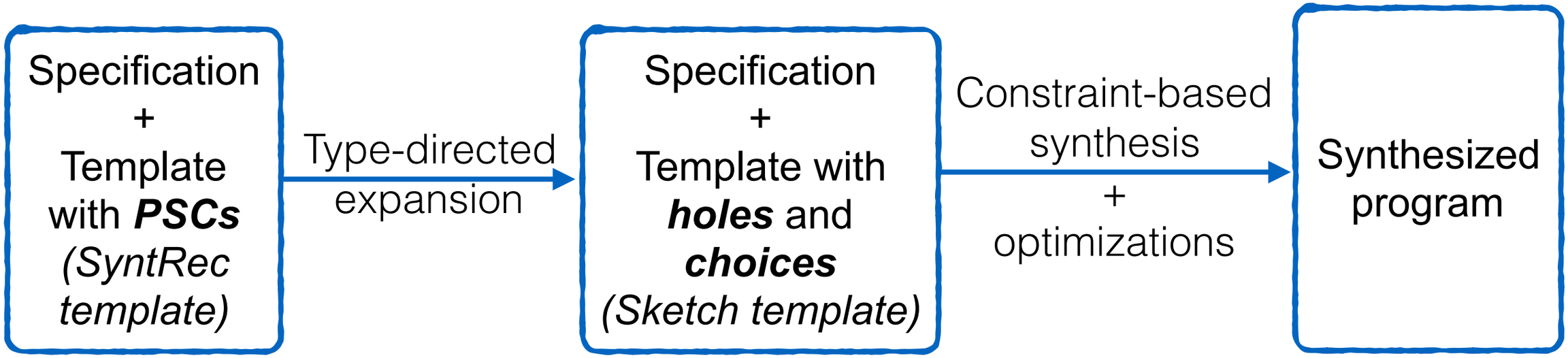
Reusable Templates

Synthesized program for binary tree insertion

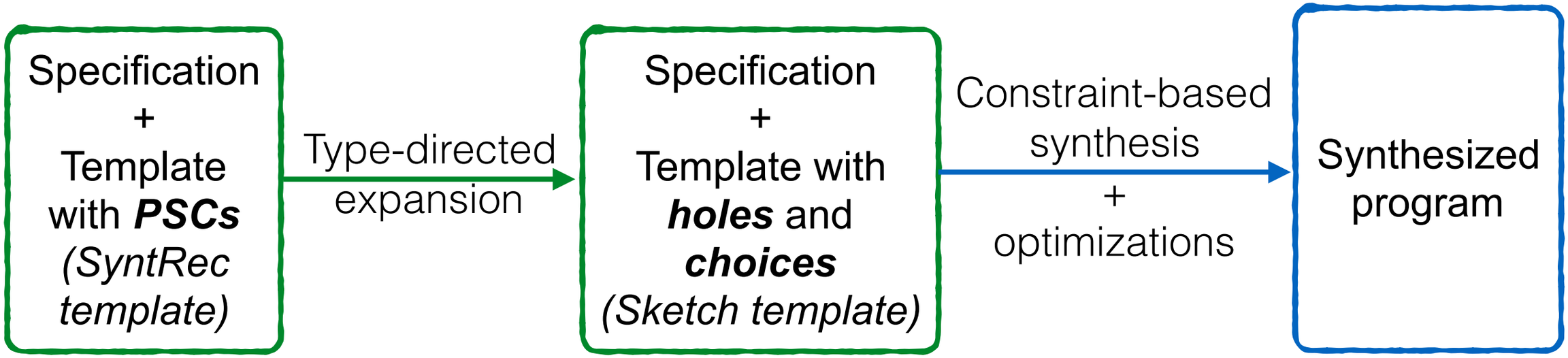
```
BTree insert (BTree tree, int x)
  switch (src)
      case Branch:
          BTree[2] a = {insert(tree.left, x), insert(tree.right, x)}
          if (tree.val <= x)
            return new BTree(tree.val, a[0], tree.right)
          if (tree.val > x)
            return new BTree(tree.val, tree.left, a[1])
  case Leaf:
      if (tree.val <= x)
        return new BTree(x, new Leaf(tree.v), new Empty())
      if (tree.val > x)
        return new BTree(x, new Empty(), new Leaf(tree.v))
  case Empty:
      return new Leaf(x)
```

Synthesis Time: 20s

Synthesis Approach

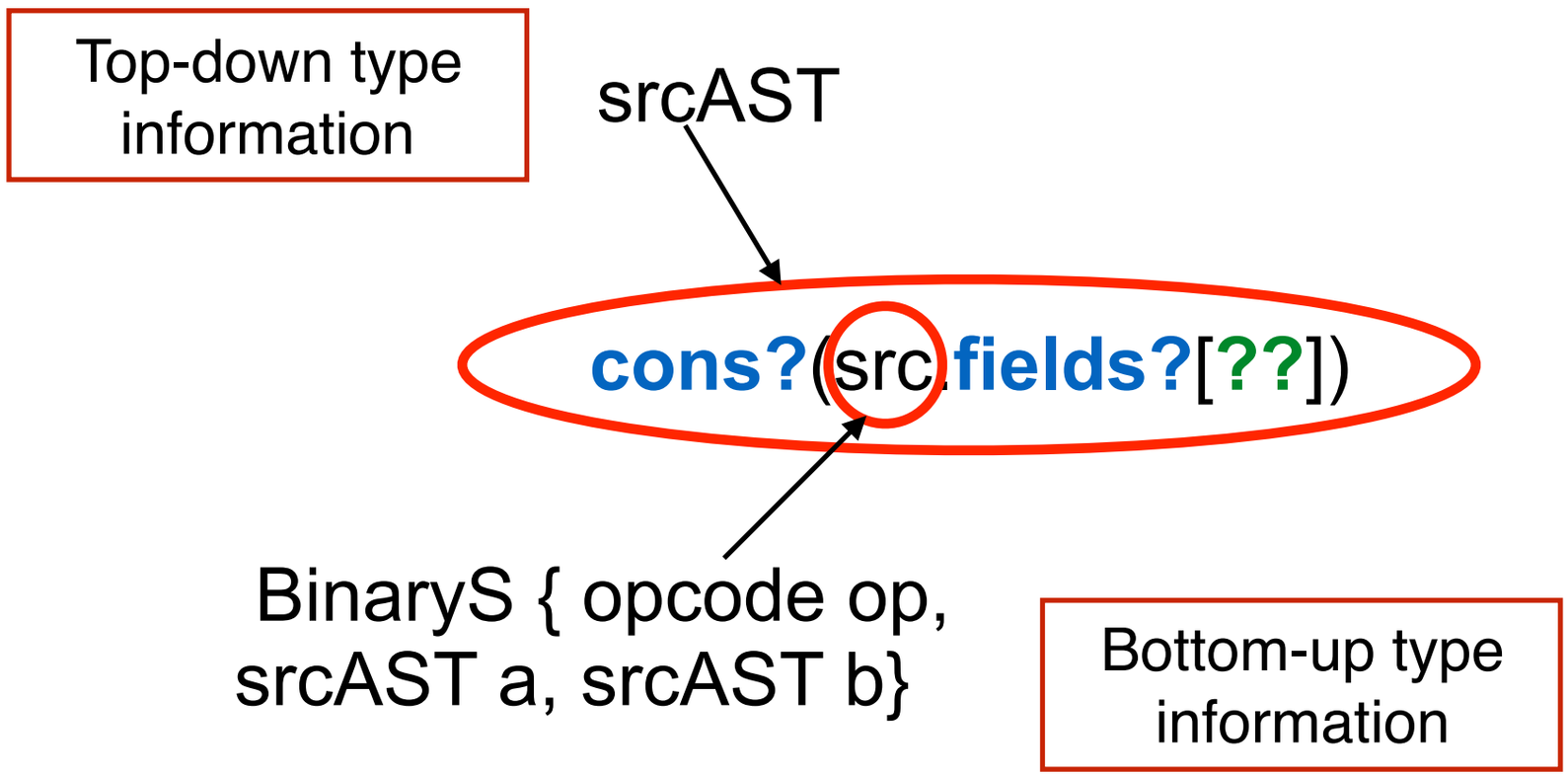


Synthesis Approach



Type-directed expansion

Requires propagating type information both top-down and bottom-up



Type-directed expansion

- Bi-directional rules of Pierce and Turner, 2000
- Also used in Myth (PLDI 2015) and Synquid (PLDI 2016)

Type Inference rule

$$\Gamma \vdash e : \theta$$

Expansion rule

$$\Gamma \vdash e \xrightarrow{\theta} e'$$

Type-directed expansion

cons?(src.**fields?**[**??**]) srcAST →

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
 }
```

Type-directed expansion

cons?(src.**fields?**[**??**]) —srcAST→ **choose** (**new** NumS(…), **new** BinaryS(…), …)

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
 }
```

Type-directed expansion

cons?(src.**fields?**[**??**]) —srcAST→ **choose** (**new** NumS(…), **new** BinaryS(a = src.**fields?**[**??**], …), …)

src.**fields?**[**??**] —srcAST→

src.**fields?** —srcAST[ ]→

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
 }
```

Type-directed expansion

cons?(src.**fields?**[**??**]) —srcAST→ **choose** (**new** NumS(…), **new** BinaryS(a = src.**fields?**[**??**], …), …)

src.**fields?**[**??**] —srcAST→

src.**fields?** —srcAST[ ]→

src —BinaryS→ src

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
}
```

Type-directed expansion

cons?(src.**fields?**[**??**]) —srcAST→ **choose** (**new** NumS(…), **new** BinaryS(a = src.**fields?**[**??**], …), …)

src.**fields?**[**??**] —srcAST→ **choose**(src.a , src.b)

src.**fields?** —srcAST[ ]→ { src.a , src.b}

src —BinaryS→ src

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
 }
```

Type-directed expansion

cons?(src.**fields?**[**??**]) —srcAST→ **choose** (**new** NumS(…), **new** BinaryS(a = **choose**(src.a , src.b) …), …)

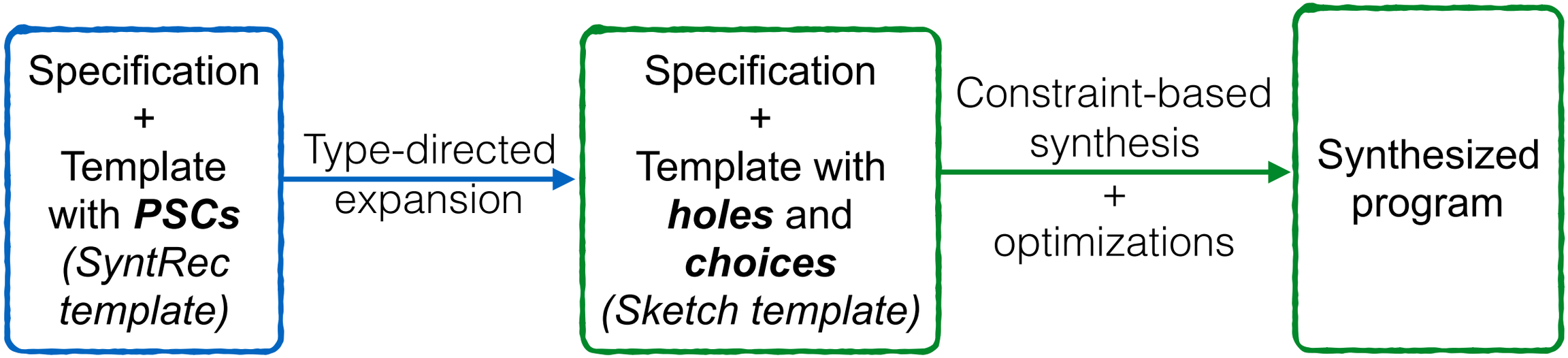
src.**fields?**[**??**] —srcAST→ **choose**(src.a , src.b)

src.**fields?** —srcAST[ ]→ { src.a , src.b}

src —BinaryS→ src

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
 }
```

Synthesis Approach

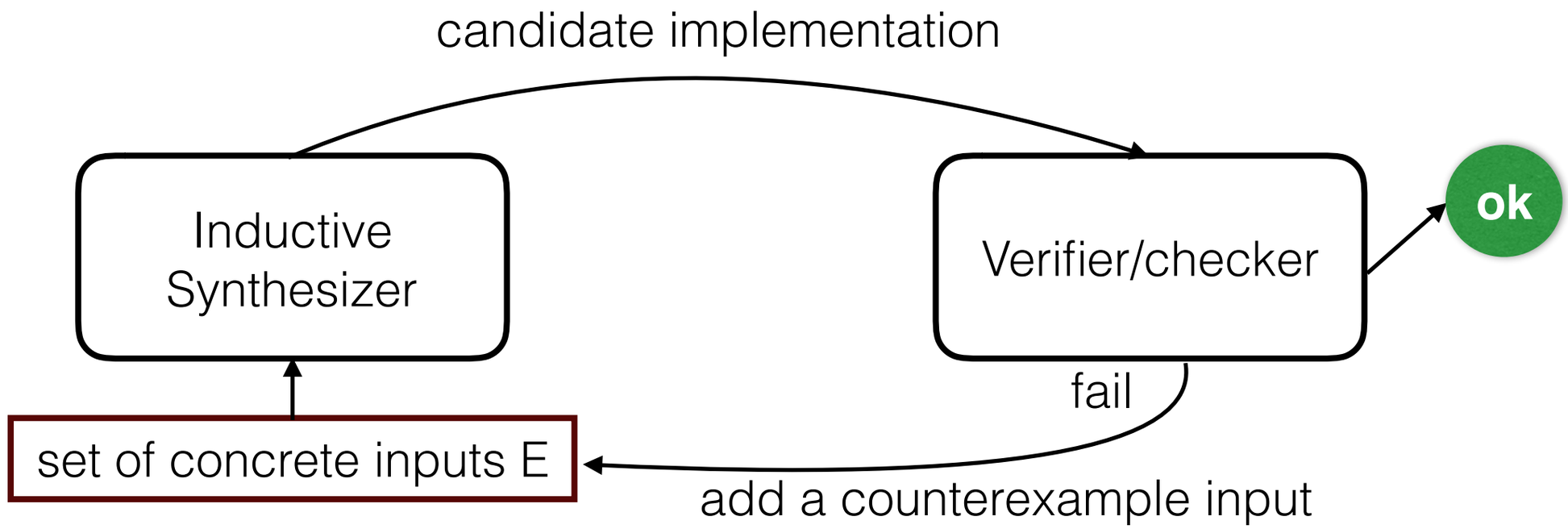


Constraint-based synthesis

- Inlines function calls and unrolls loops

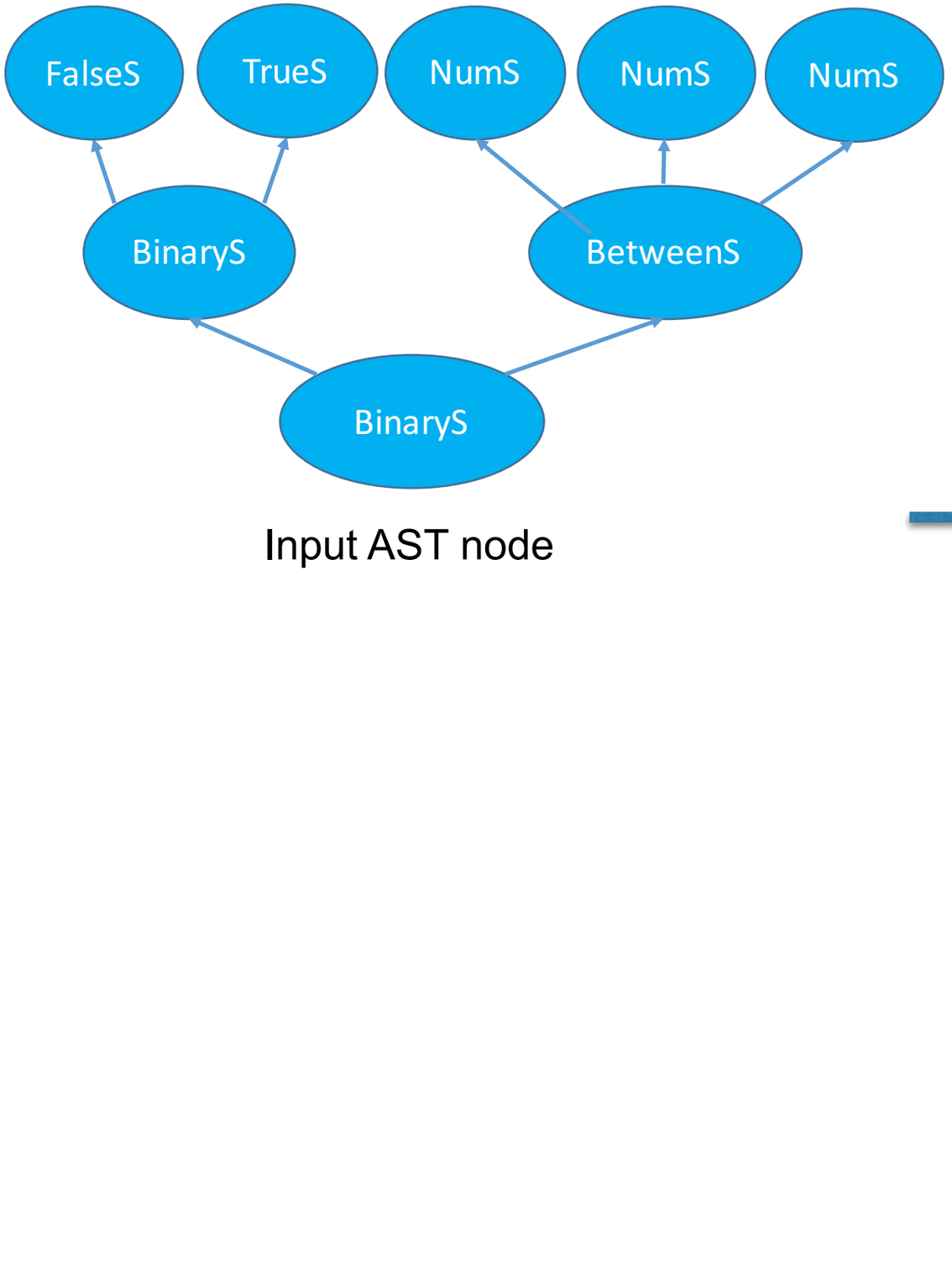
$$\exists c.\ \forall in.\ Q(in, c)$$

- Uses Counter Example Guided Inductive Synthesis (CEGIS)



[1] A. Solar-Lezama. Program Synthesis By Sketching, 2008

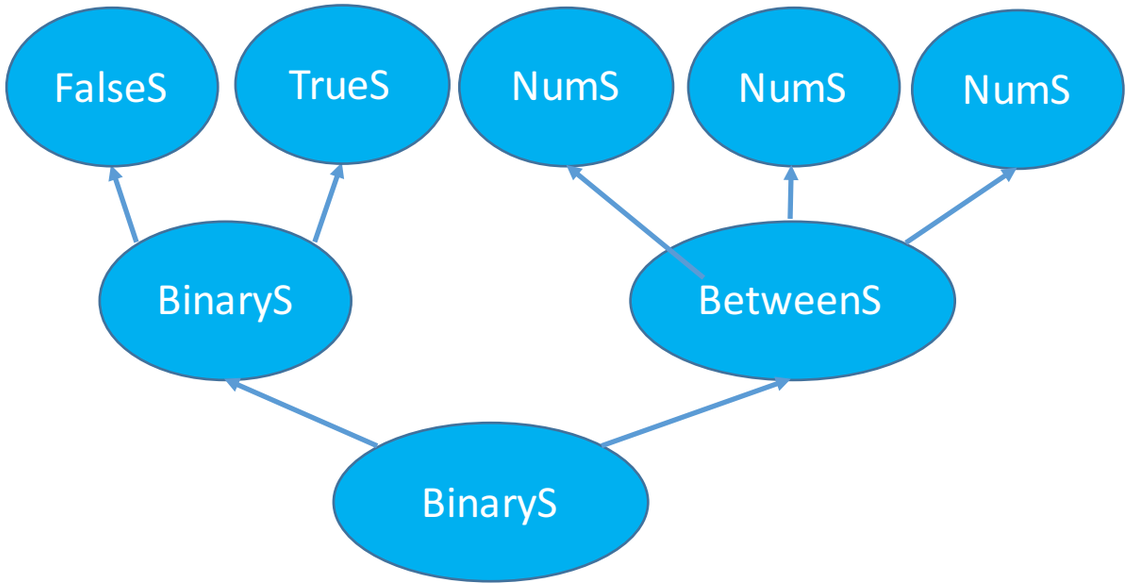
Constraint-based synthesis

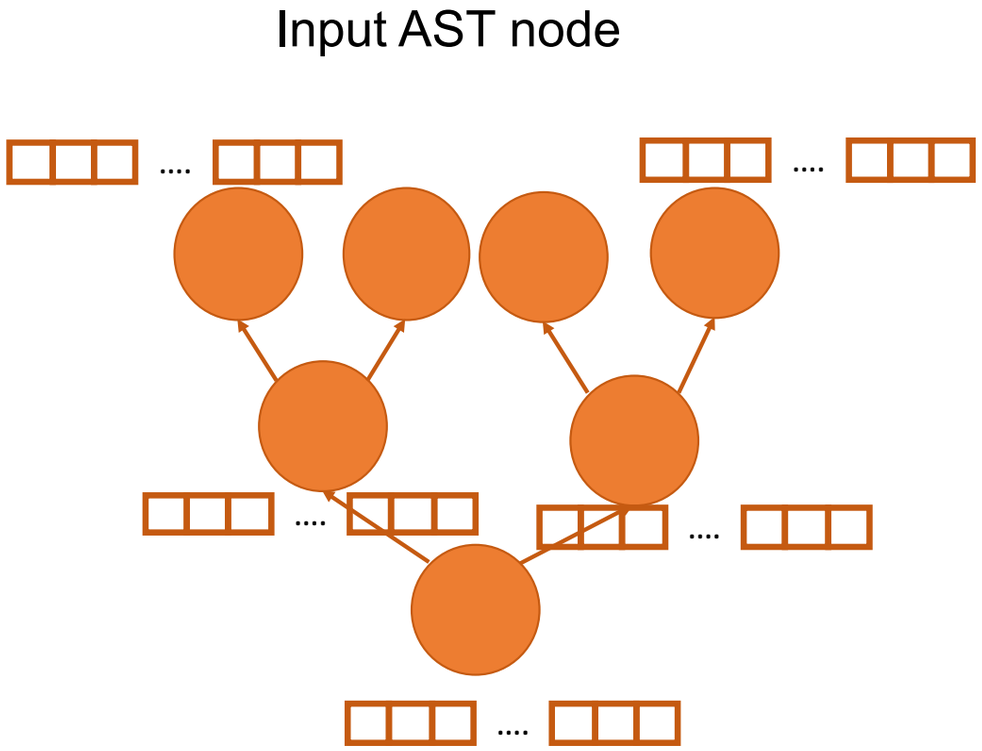


Input AST node

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      return rcons(()→choose(a, b, src .op))
    case BetweenS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      dstAST c = desugar(src.c)
      return rcons(()→choose(a, b, c ))
```

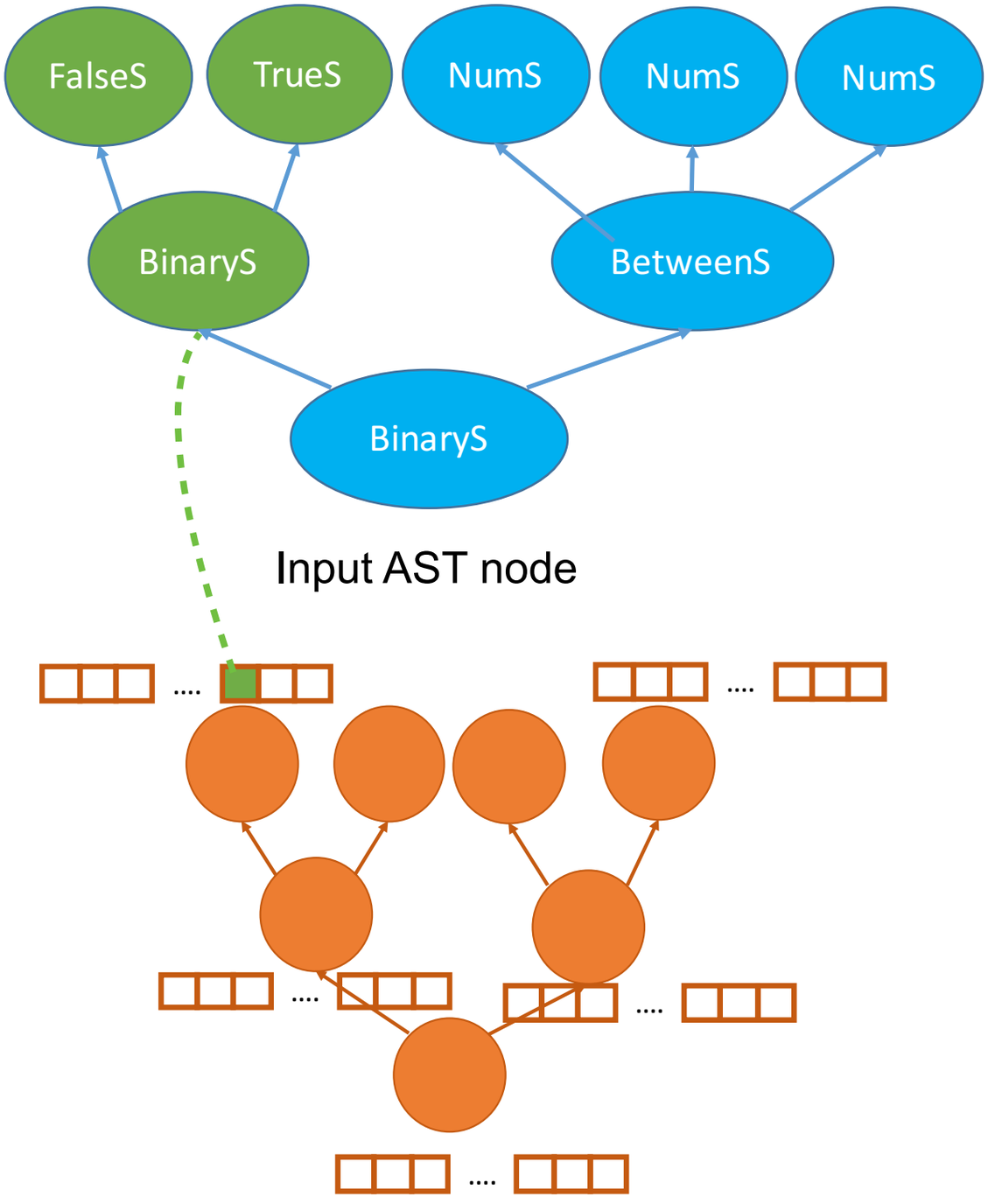
Constraint-based synthesis





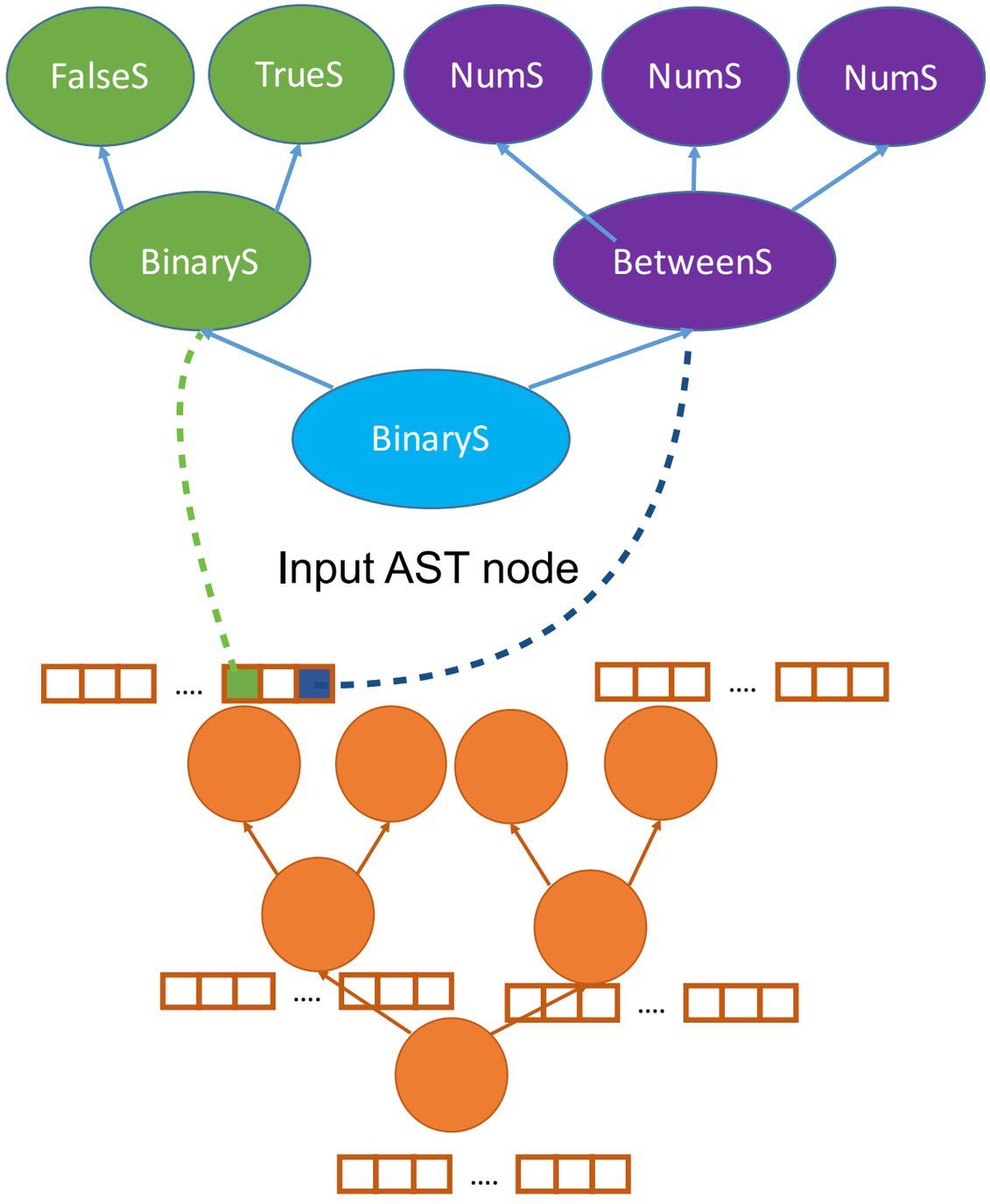

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      return rcons(()→choose(a, b, src .op))
    case BetweenS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      dstAST c = desugar(src.c)
      return rcons(()→choose(a, b, c ))
```

Constraint-based synthesis



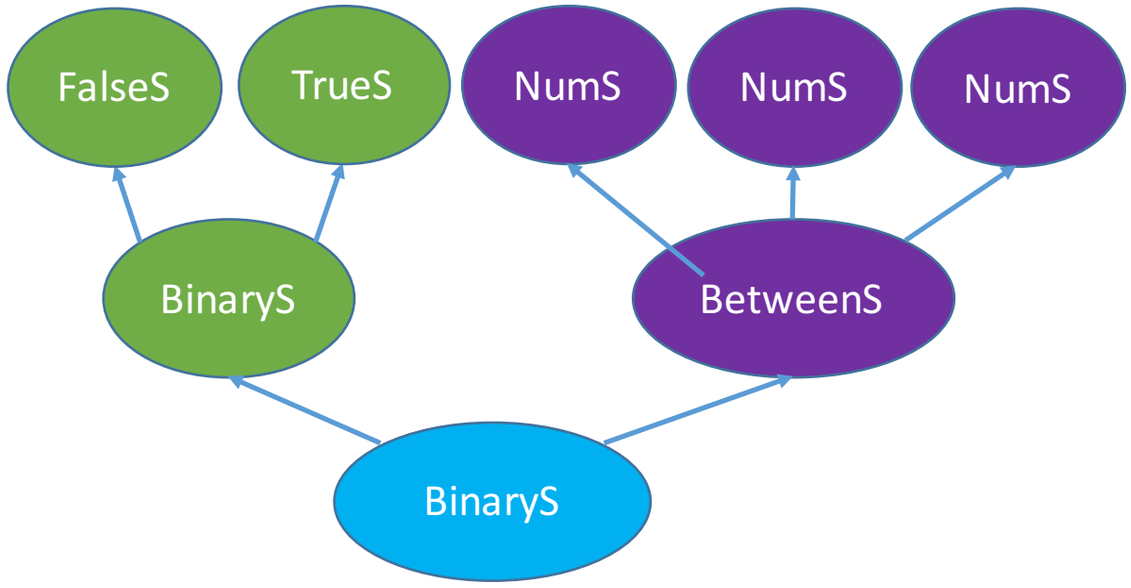

```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      return rcons(()→choose(a, b, src .op))
    case BetweenS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      dstAST c = desugar(src.c)
      return rcons(()→choose(a, b, c ))
```

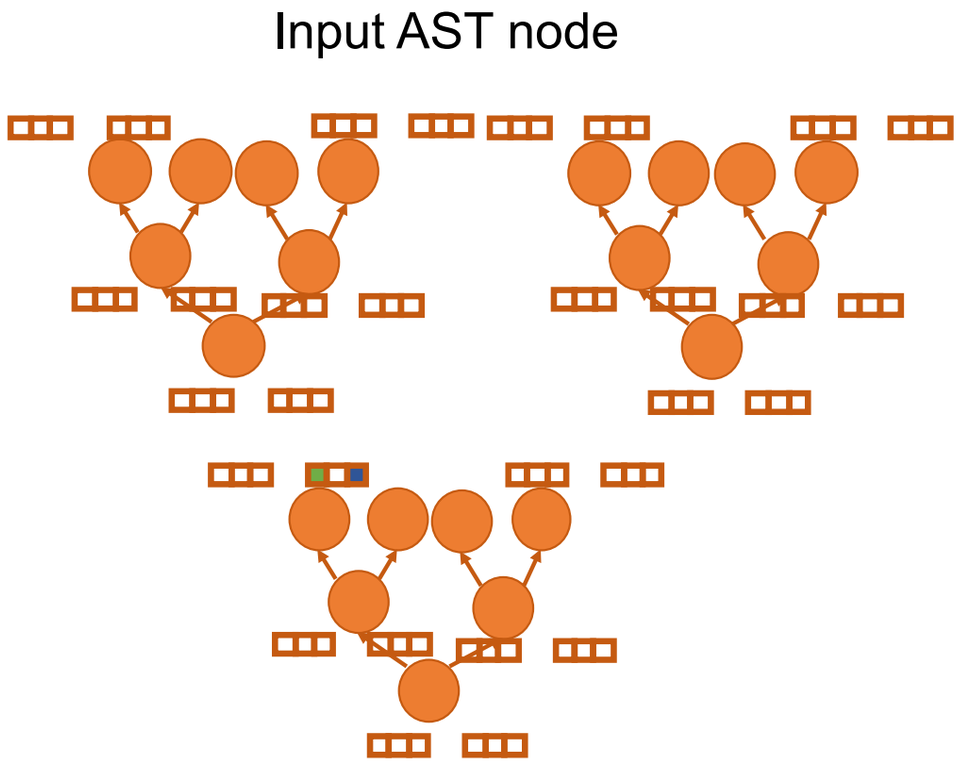
Constraint-based synthesis




```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      return rcons(()→choose(a, b, src .op))
    case BetweenS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      dstAST c = desugar(src.c)
      return rcons(()→choose(a, b, c ))
```

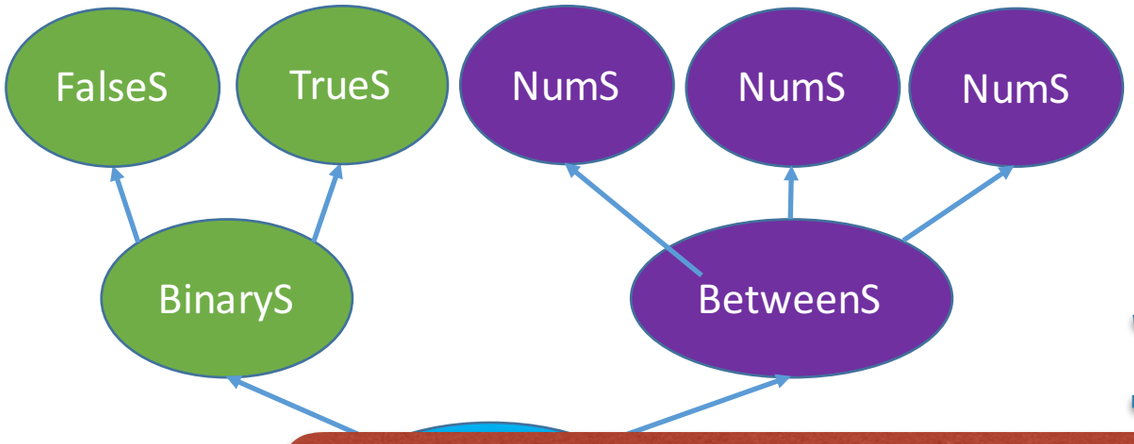
Constraint-based synthesis






```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
    case FalseS:
      return rcons(()→ 0 )
    case BinaryS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      return rcons(()→choose(a, b, src .op))
    case BetweenS:
      dstAST a = desugar(src.a)
      dstAST b = desugar(src.b)
      dstAST c = desugar(src.c)
      return rcons(()→choose(a, b, c ))
```

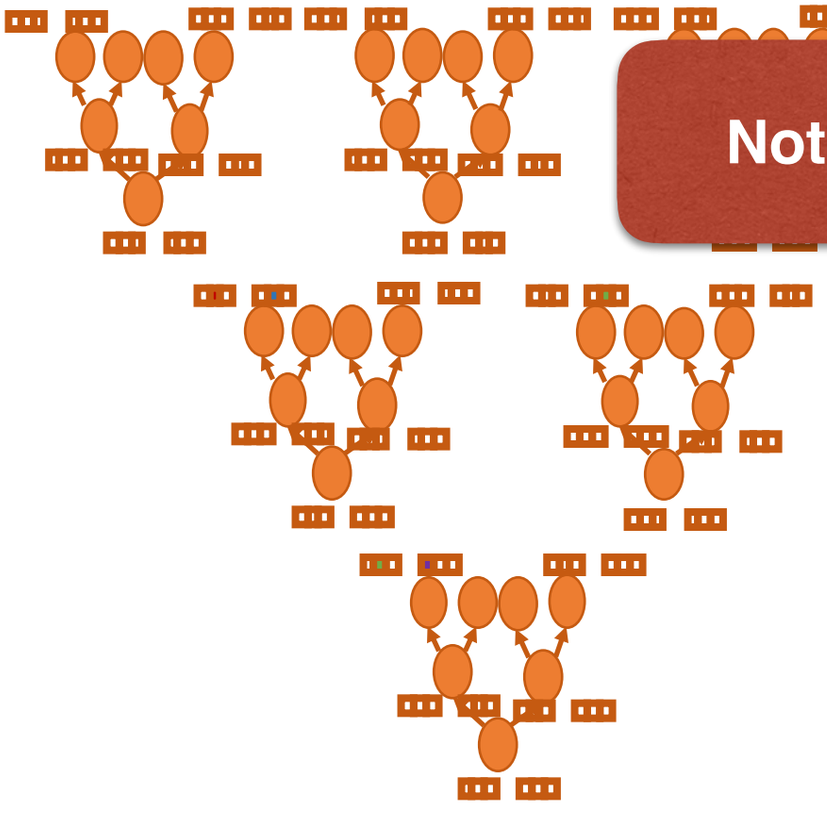
Constraint-based synthesis




```
dstAST desugar (srcAST src)
  switch (src)
    case NumS:
      return rcons(()→src .v )
    case TrueS:
      return rcons(()→ 0 )
```

Synthesizer needs to reason about all variants together

Input AST node



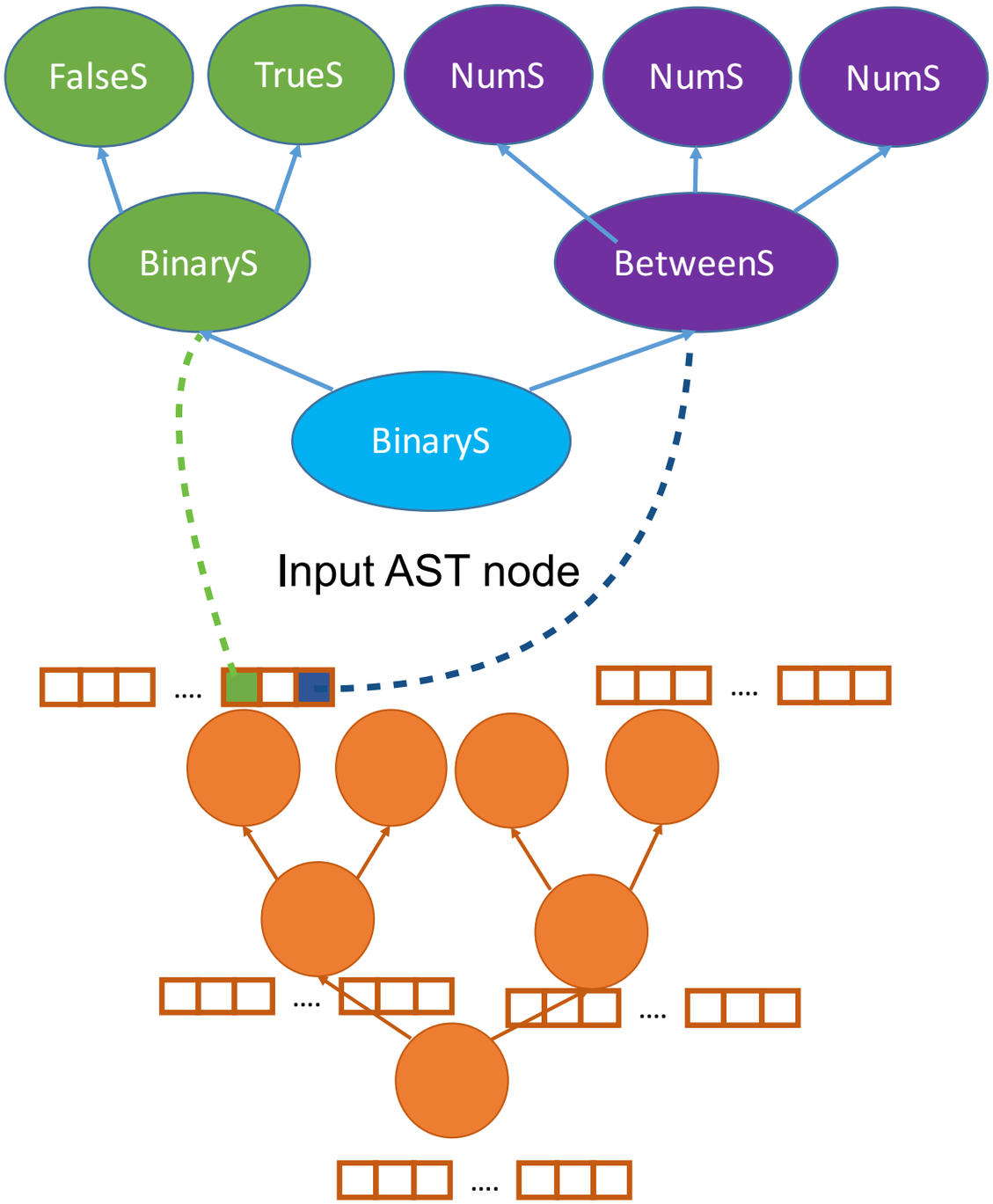
Not scalable for ADTs with many cases

```
dstAST a = desugar(src.a)
dstAST b = desugar(src.b)
      e(a, b, src .op))
      .a)
dstAST b = desugar(src.b)
dstAST c = desugar(src.c)
return rcons(()→choose(a, b, c ))
```

Inductive Decomposition

- Use specification as an invariant to abstract recursive calls

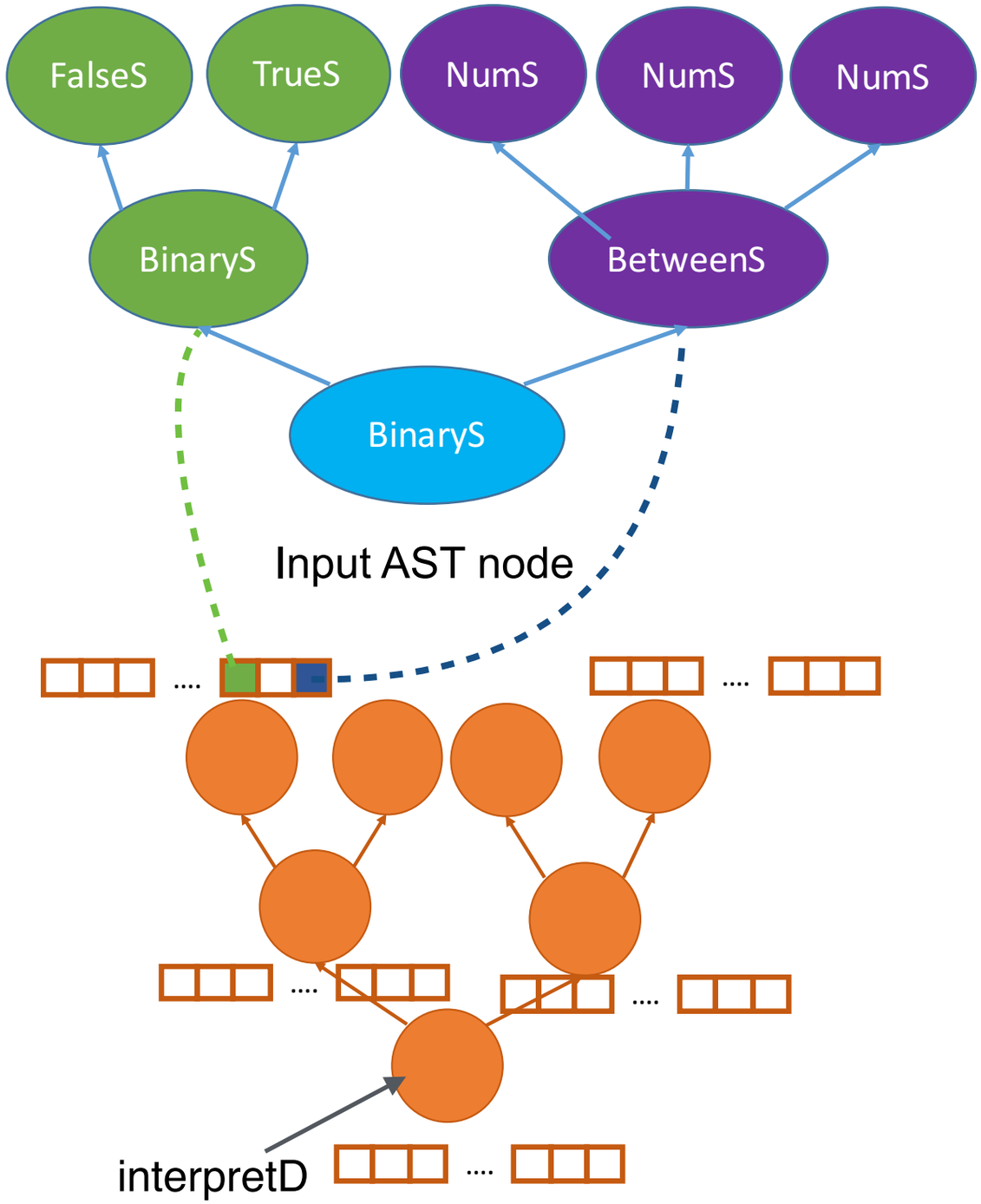
Inductive Decomposition




```
int interpretD (dstAST dst)
  switch (dst)
    case NumD:
      return dst.v
    case BoolD:
      return dst.v
    case Binary:
      base a = interpretD(dst.a)
      base b = interpretD(dst.b)
      return compute(dst.op, a, b)
```

interpretS(s) == interpretD(desugar(s))

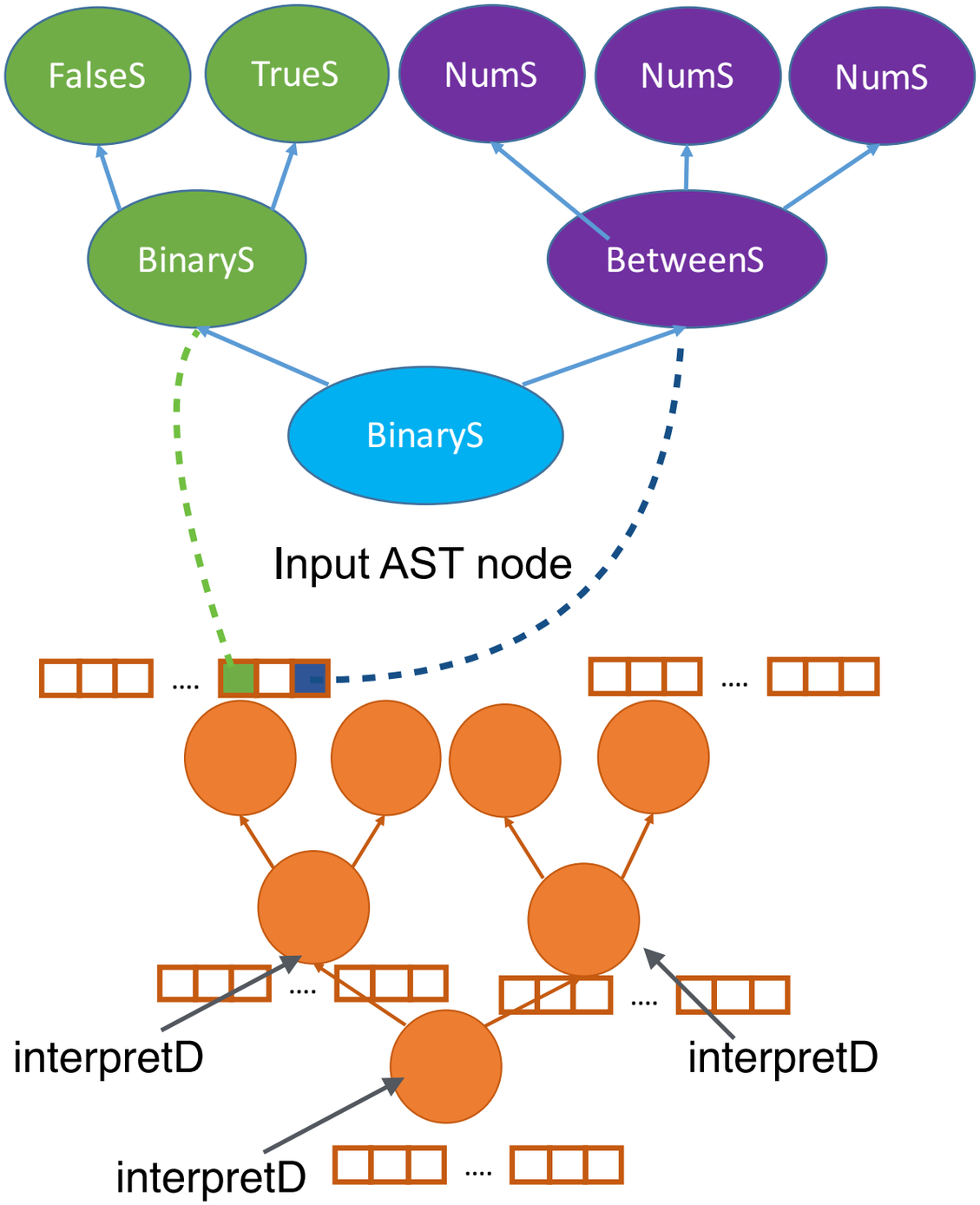
Inductive Decomposition




```
int interpretD (dstAST dst)
  switch (dst)
    case NumD:
      return dst.v
    case BoolD:
      return dst.v
    case Binary:
      base a = interpretD(dst.a)
      base b = interpretD(dst.b)
      return compute(dst.op, a, b)
```

interpretS(s) == interpretD(desugar(s))

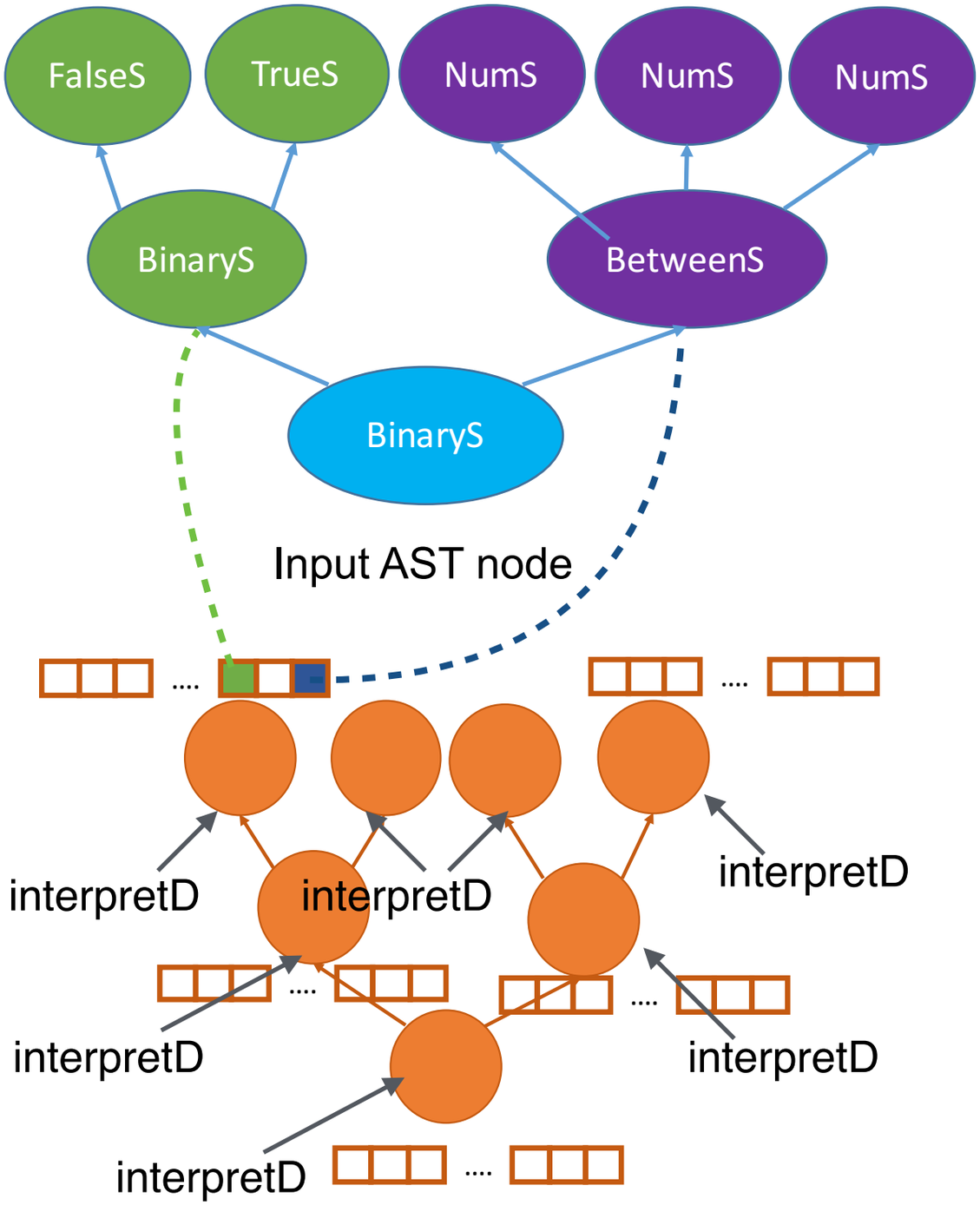
Inductive Decomposition




```
int interpretD (dstAST dst)
  switch (dst)
    case NumD:
      return dst.v
    case BoolD:
      return dst.v
    case Binary:
      base a = interpretD(dst.a)
      base b = interpretD(dst.b)
      return compute(dst.op, a, b)
```

interpretS(s) == interpretD(desugar(s))

Inductive Decomposition



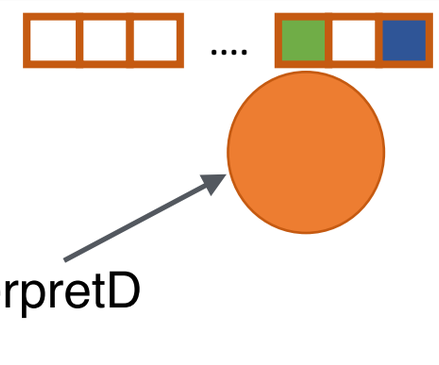

```
int interpretD (dstAST dst)
  switch (dst)
    case NumD:
      return dst.v
    case BoolD:
      return dst.v
    case Binary:
      base a = interpretD(dst.a)
      base b = interpretD(dst.b)
      return compute(dst.op, a, b)
```

interpretS(s) == interpretD(desugar(s))

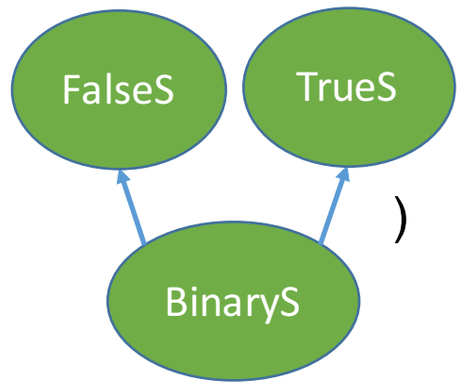
Inductive Decomposition

interpretS(s) == interpretD(desugar(s))

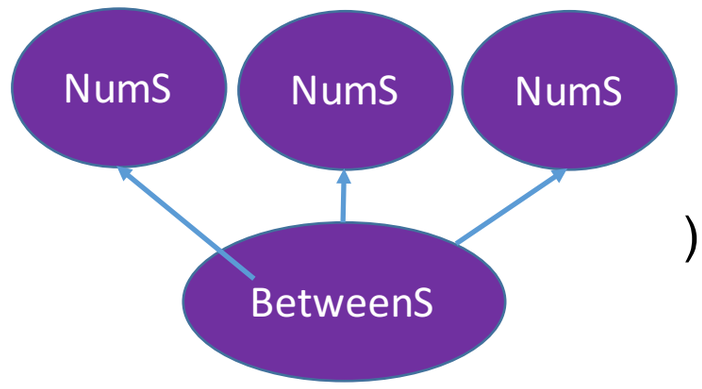
Case 1: Node flows directly into interpretD



interpretD ( ) == interpretS ( )



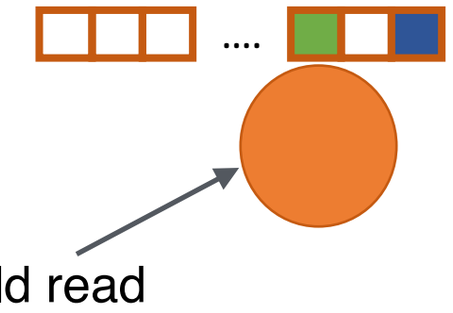
interpretD ( ) == interpretS ( )



Inductive Decomposition

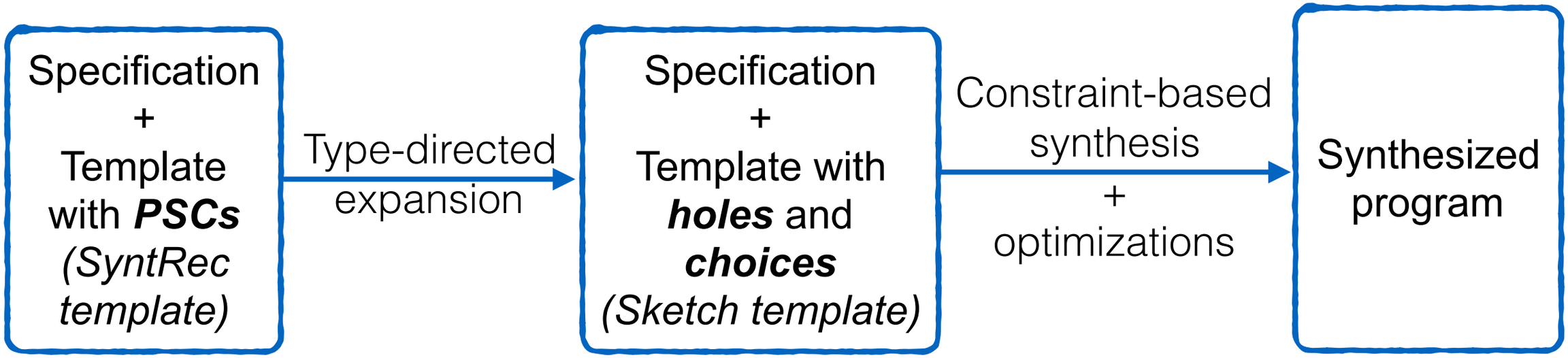
interpretS(s) == interpretD(desugar(s))

Case 2: Node flows into other nodes



Revert back to inlining the desugar calls

Synthesis Approach



Evaluation

- 23 benchmarks across various domains

Desugaring

List

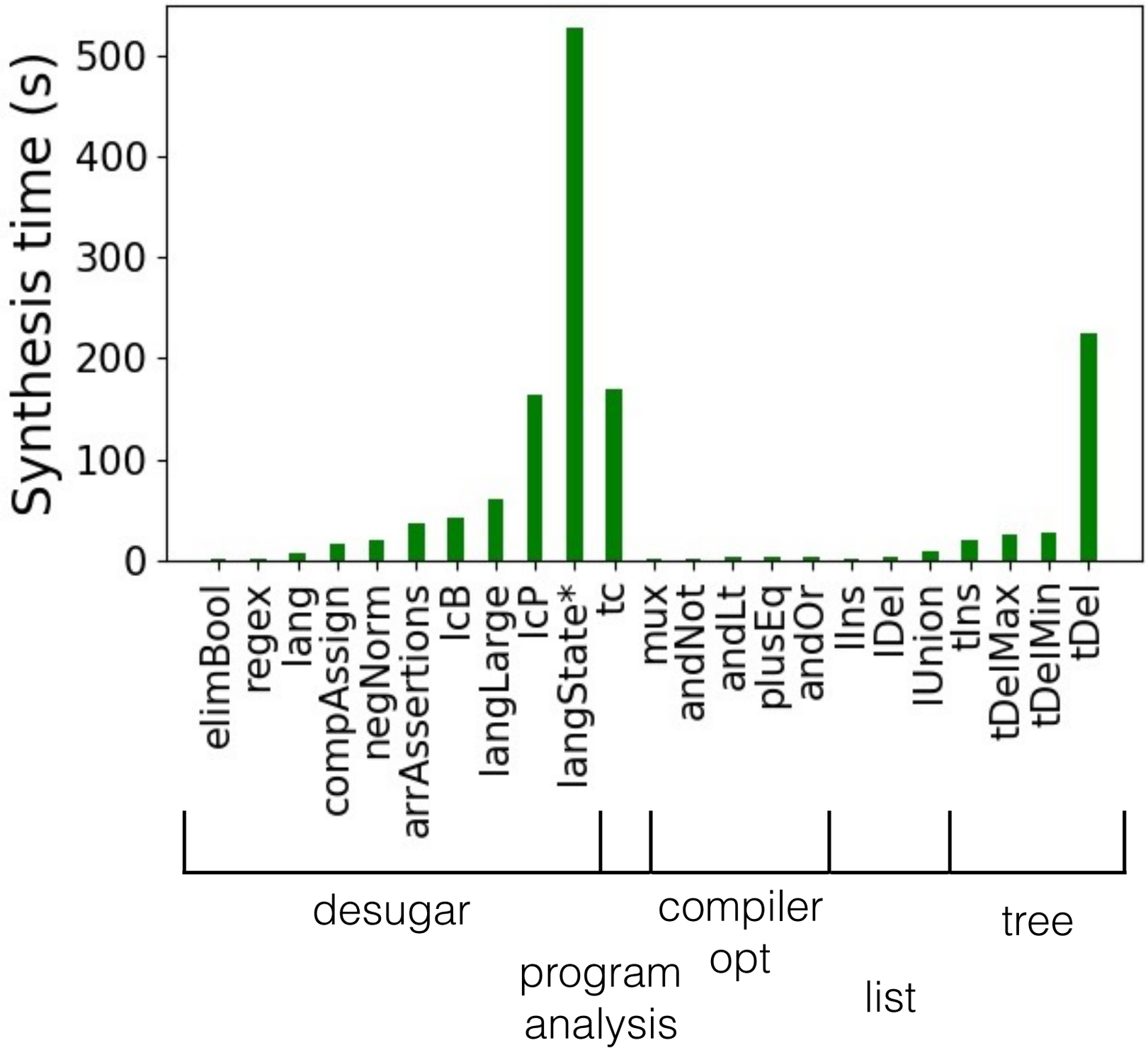
Tree

Program Analysis

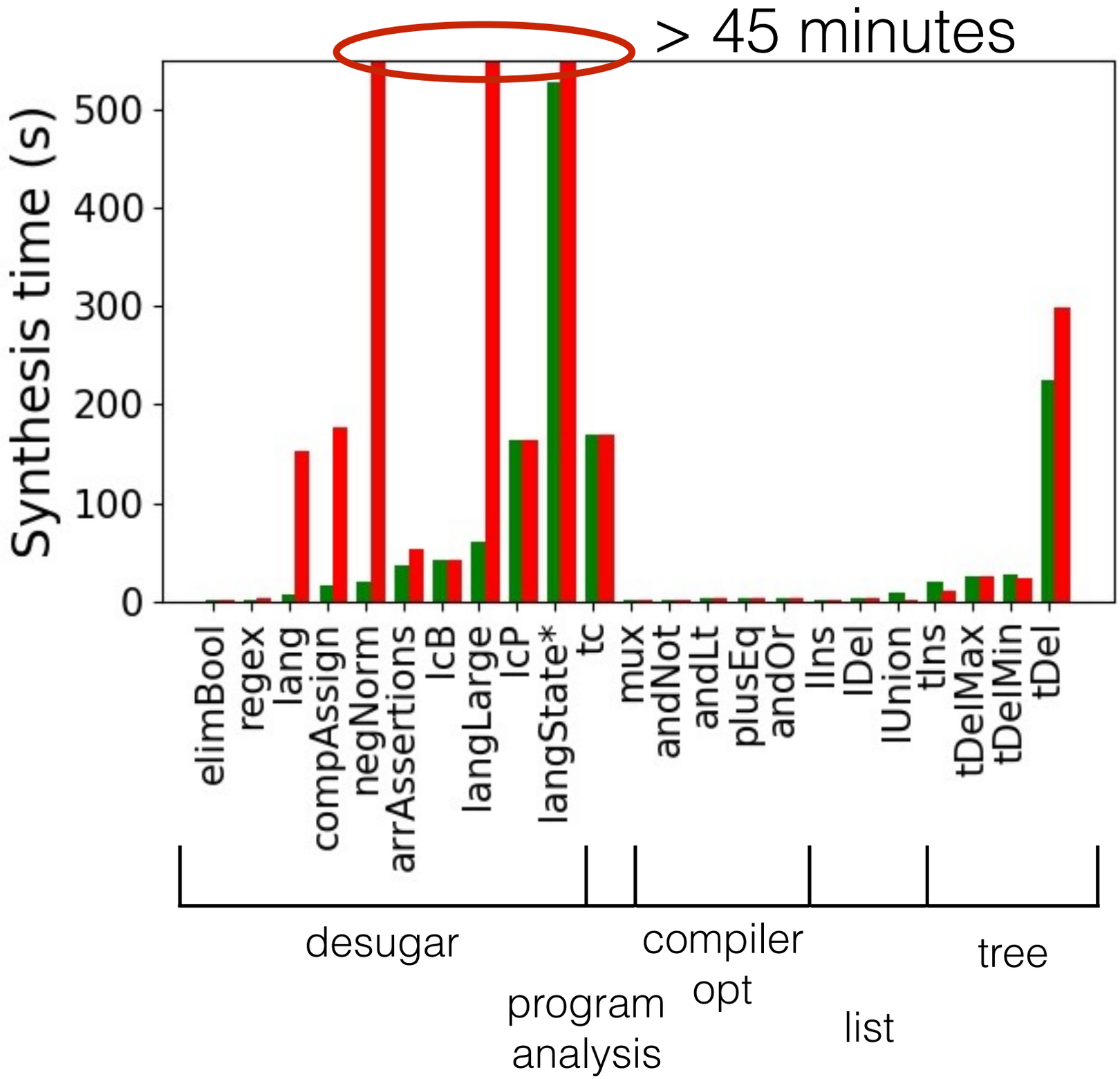
Compiler optimizations

From only 4 generic library templates

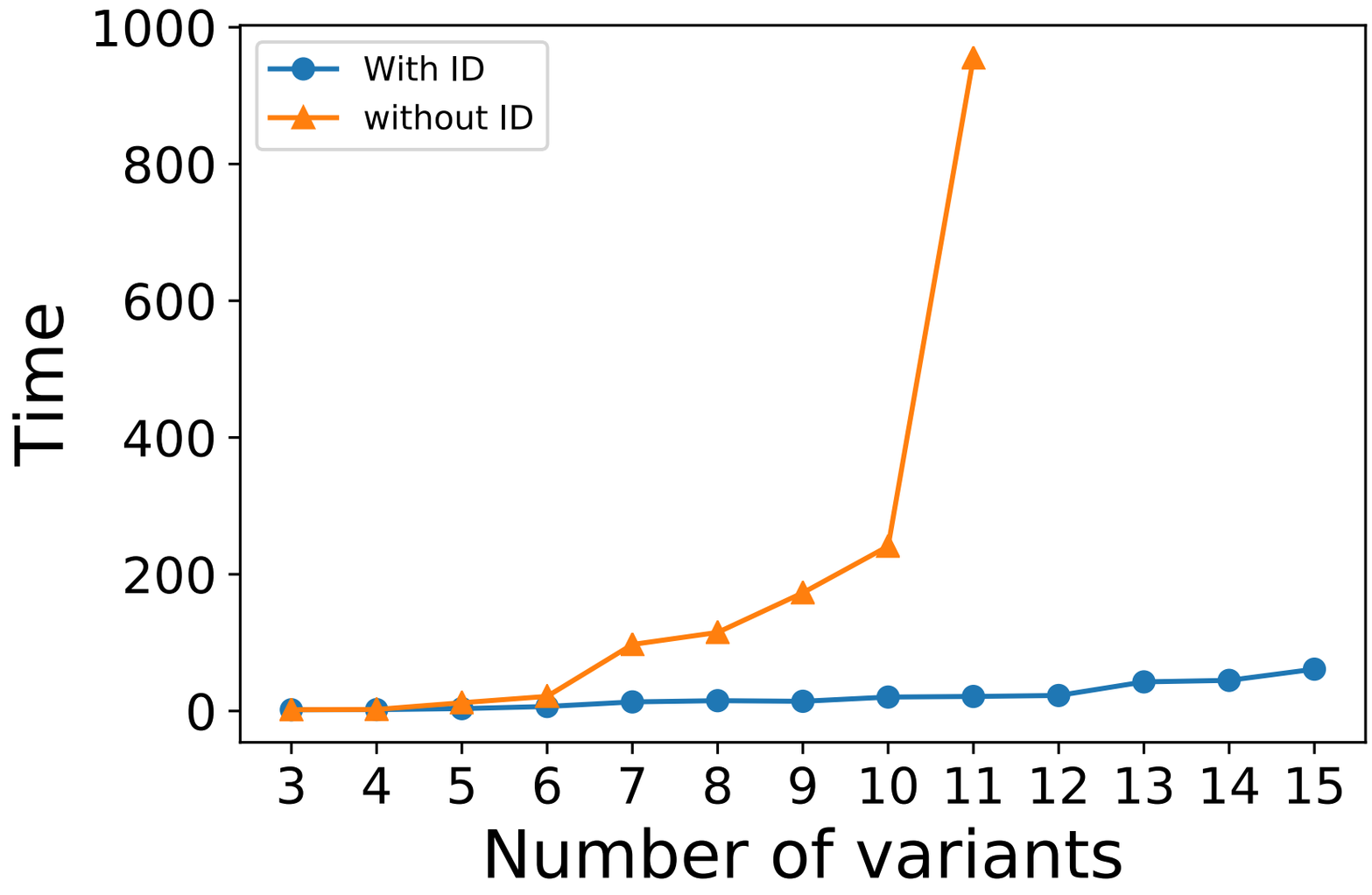
Evaluation


Synthesis time (s)
0
100
200
300
400
500
elimBool
regex
lang
compAssign
negNorm
arrAssertions
lcB
langLarge
lcP
langState*
tc
mux
andNot
andLt
plusEq
andOr
llns
lDel
lUnion
tlns
tDelMax
tDelMin
tDel
desugar
program analysis
compiler opt
list
tree

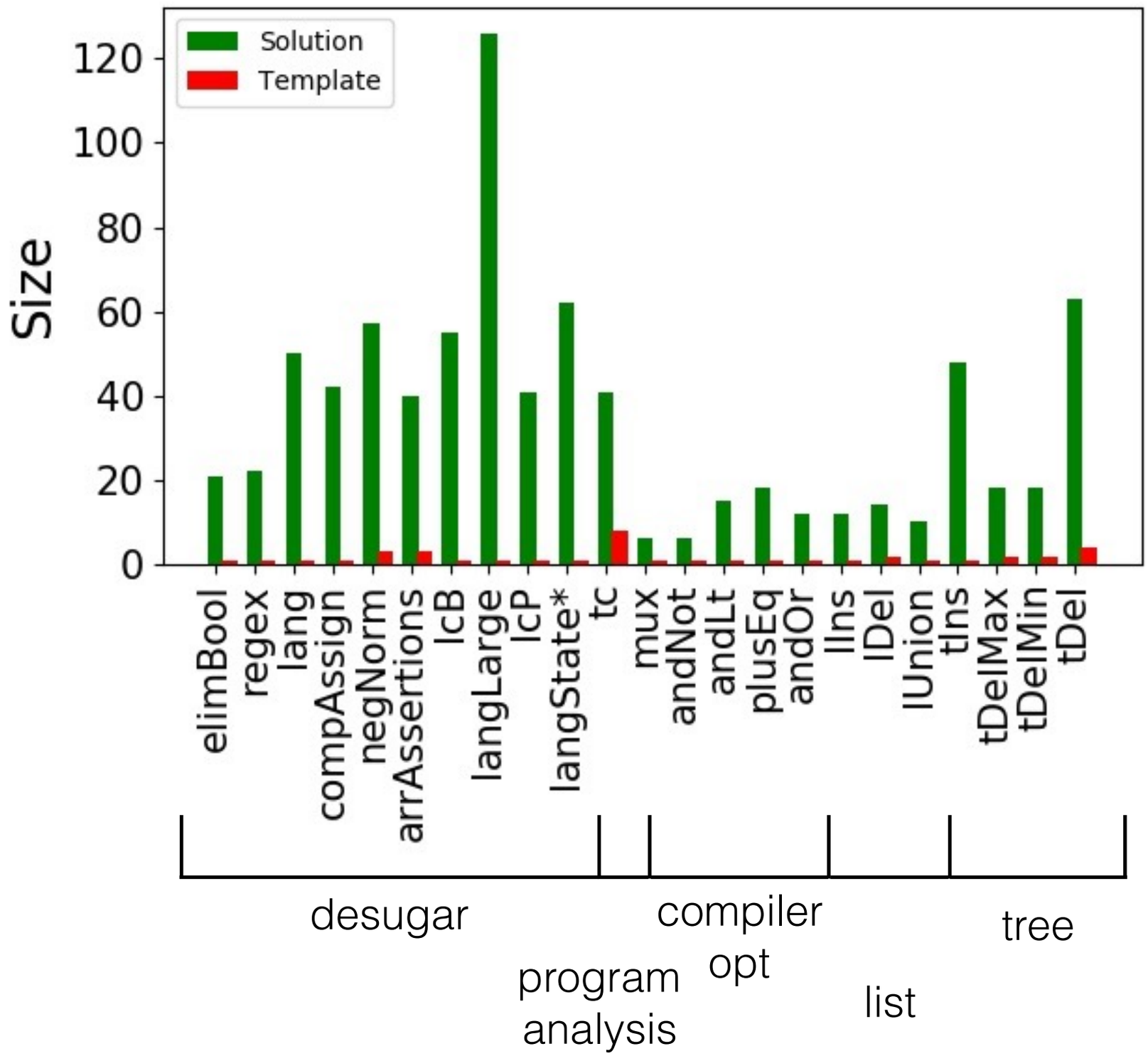
Evaluation


> 45 minutes
Synthesis time (s)
0
100
200
300
400
500
elimBool
regex
lang
compAssign
negNorm
arrAssertions
lcB
langLarge
lcP
langState*
tc
mux
andNot
andLt
plusEq
andOr
llns
lDel
lUnion
tlns
tDelMax
tDelMin
tDel
desugar
compiler opt
tree
program analysis
list

Evaluation


With ID
without ID
Time
1000
800
600
400
200
0
3
4
5
6
7
8
9
10
11
12
13
14
15
Number of variants

Evaluation


Solution
Template
Size
120
100
80
60
40
20
0
elimBool
regex
lang
compAssign
negNorm
arrAssertions
lcB
langLarge
lcP
langState*
tc
mux
andNot
andLt
plusEq
andOr
lIns
lDel
lUnion
tIns
tDelMax
tDelMin
tDel
desugar
program analysis
compiler opt
list
tree

Summary

- A system to synthesize transformations on Algebraic Data Types from high level ***reusable templates***
- Uses a combination of type inference and constraint solving
- ***Inductive decomposition*** optimization to improve the scalability of synthesis of recursive functions
- Can synthesize complex functions like desugaring