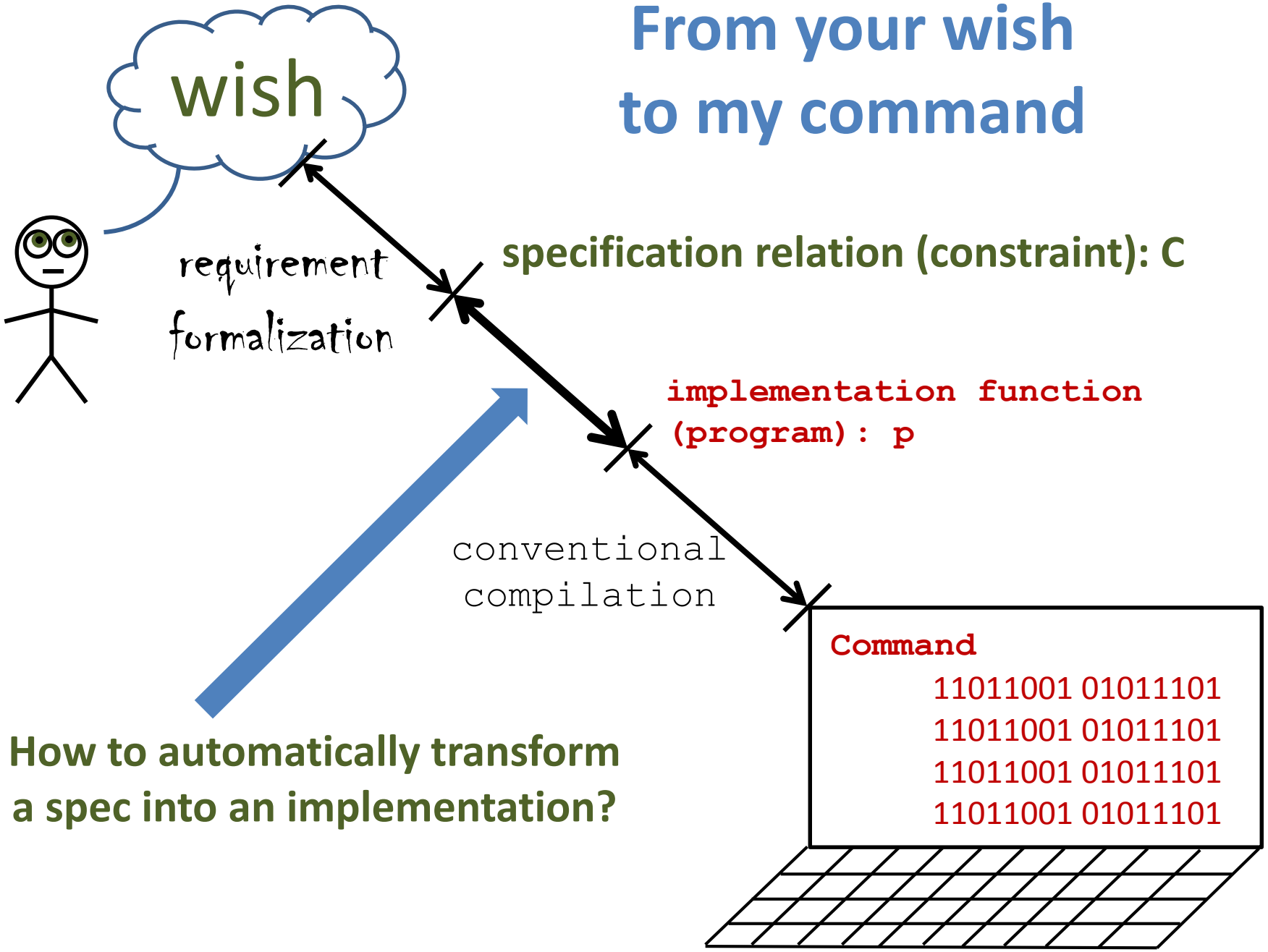
Synthesizing Computable Functions from Relations

Viktor Kuncak
EPFL
Laboratory for Automated Reasoning and Analysis

http://lara.epfl.ch http://leon.epfl.ch

From your wish to my command



How to automatically transform a spec into an implementation?

Sorting Specification as a Program

input

8900
6000
24140
2900

output

2900
6000
8900
24140

2900 < 6000

6000 < 8900

8900 < 24140

{8900, 6000, 24140,2900} = {2900, 6000, 8900, 24140}

Given a list of numbers, make **this list sorted**

(wish)

```
def sort_spec(input : List, output : List) : Boolean =
  content(output)==content(input)   &&  isSorted(output)
```

Specification (for us) is a *program* that checks, for a **given input**, whether the **given output** is acceptable

http://leon.epfl.ch

Etienne Kneuss
(to blame if demo breaks)

Tasks of Interest (i: input, o: output)

both specification **C** and program **p** are given:

a) Check assertion while program **p** runs: **C(i,p(i))**

b) Verify whether program always meets the spec: **∀i. C(i,p(i))**

only specification **C** is given:

c) Constraint programming: once **i** is known, find **o** to satisfy a given constraint: **find o such that C(i,o)**

d) Synthesis: solve **C** symbolically to obtain program **p** that is correct by construction, for all inputs: **find p such that ∀i.C(i,p(i))** **i.e.** **p ⊆ C**

run-time

compile-time

Runtime Assertion Checking

a) **Check assertion** while program **p** runs: **C(i,p(i))**

```
def p(i : List) : List = {
   sort i using a sorting algorithm and return the result
} ensuring (o ⇒ content(i)==content(o)  &&  isSorted(o))
```

```
def content(lst : List) = lst match {
  case Nil ⇒ Set.empty
  case Cons(x, xs) ⇒ Set(x) ++ content(xs)
}
def isSorted(lst : List) = lst match {
  case Nil                ⇒ true
  case Cons(_, Nil)  ⇒ true
  case Cons(x, Cons(y, ys)) ⇒
     x < y && isSorted(Cons(y, ys))
}
```

Already works in Scala!

Key design decision:

constraints are programs

Runtime checks are expensive

⇒ memoization of checks

Runtime Verification'14

Emmanouil Koukoutos

Static Verification in Leon - Same Input

b) Verify that program always meets spec: **∀i. C(i,p(i))**

```
def p(i : List) : List = {
   sort i using a sorting algorithm and return the result
} ensuring (o ⇒ content(i)==content(o)  &&  isSorted(o))
```

```
def content(lst : List) = lst match {
  case Nil ⇒ Set.empty
  case Cons(x, xs) ⇒ Set(x) ++ content(xs)
}
def isSorted(lst : List) = lst match {
  case Nil                ⇒ true
  case Cons(_, Nil)  ⇒ true
  case Cons(x, Cons(y, ys)) ⇒
     x < y && isSorted(Cons(y, ys))
}
```

Type in a Scala program and spec, see it verified

timeout

proof of **∀i. C(i,p(i))**

input **i** such that **not C (i,p(i))**

Insertion Sort Verified as You Type It

Leon Save Undo Redo Link Menu Console Params

```
31  def sortedIns(e: Int, l: List): List = {
32    require(isSorted(l))
33    l match {
34      case Nil() => Cons(e,Nil())
35      case Cons(x,xs) =>
36        if (x <= e) Cons(x,sortedIns(e, xs)) else Cons(e, l)
37    }
38  } ensuring(res => contents(res) == contents(l) ++ Set(e)
39                    && isSorted(res)
40                    && size(res) == size(l) + 1
41                  )
42  /* Insertion sort yields a sorted list of
43     same size and content as the input list */
44  def sort(l: List): List = (l match {
45    case Nil() => Nil()
46    case Cons(x,xs) => sortedIns(x, sort(xs))
47  }) ensuring(res => contents(res) == contents(l)
48                    && isSorted(res)
49                    && size(res) == size(l))
```

Load a Program:

-- Select a Program --

Analysis Results

Function	Verif.
size	✓
contents	✓
isSorted	✓
sortedIns	✓
sort	✓

☺

Reported Counterexample in Case of a Bug

Leon Save Undo Redo Link Menu Console Params

```
31  def sortedIns(e: Int, l: List): List = {
32    require(isSorted(l))
33    l match {
34      case Nil() => Cons(e,Nil())
35      case Cons(x,xs) => Cons(x,sortedIns(e, xs))
36    }
37  } ensuring(res => contents(res) == contents(l) ++ Set(e)
38                    && isSorted(res)
39                    && size(res) == s
40          )
41  /* Insertion sort yields a sorted l
42      same size and content as the in
43  def sort(l: List): List = (l match
44    case Nil() => Nil()
45    case Cons(x,xs) => sortedIns(x, s
46  }) ensuring(res => contents(res) ==
47                    && isSorted(res)
48                    && size(res) ==
49 }
```

Load a Program:

-- Select a Program --

Analysis Results

Verification

Leon verifies the validity all the verification conditions found in the selected function.

Invalid!

Function	Kind	Result	Time
sortedIns	precond.	✔ valid	0.012
sortedIns	postcond.	✖ invalid	0.051

The following valuation violates the VC:

```
l := Cons(8946, Nil())
e := 8945
```

Approaches and Their Guarantees

both specification **C** and program **p** are given:

a) Check assertion while program **p** runs: **C(i,p(i))**

b) Verify that program always meets spec: **∀i. C(i,p(i))**

only specification **C** is given:

c) Constraint programming
once **i** is known, find **o** to satisfy a given constraint:
find o such that C(i,o)

run-time

d) Synthesis: solve **C** symbolically to obtain program **p** that is correct by construction, for all inputs: **find p such that ∀i.C(i,p(i))** **i.e.** **p ⊆ C**

compile-time

Insertion into Red-Black Tree

```
def insert(x : Int, t : Tree) = choose((t1:Tree) =>
  isRBT(t1)  &&  content(t1) = content(t) ++ Set(x))
```

Remarks:

- no more effort than implementation - wrote functions for invs.
- these invariants is what drives data structure design
- this is how things are explained in a textbook
- using properties promotes reuse - combine them using &&
- functional programming alone favors *deterministic* complete implementations and does *not* automate process invariant -> code

Evolving the Program

Suppose we have a red-black tree implementation

We only implemented ‘insert’ and ‘lookup’

Now we also need to implement ‘remove’

```
void RBDelete(rb_red_blk_tree* tree, rb_red_blk_node* z){
  rb_red_blk_node* y;
  rb_red_blk_node* x;
  rb_red_blk_node* nil=tree->nil;
  rb_red_blk_node* root=tree->root;
  y= ((z->left == nil) || (z->right == nil)) ? z : TreeSuccessor(tree,z);
  x= (y->left == nil) ? y->right : y->left;
  if (root == (x->parent = y->parent)) { /* assignment of y->p to x->p is intentional */
    root->left=x;
  } else {
    if (y == y->parent->left) {
      y->parent->left=x;
    } else {
      y->parent->right=x;
    }
  }
  if (y != z) { /* y should not be nil in this case */
#ifdef DEBUG_ASSERT
    Assert( (y!=tree->nil),"y is nil in RBDelete\n");
#endif
    /* y is the node to splice out and x is its child */
    if (!(y->red)) RBDeleteFixUp(tree,x);
    tree->DestroyKey(z->key);
    tree->DestroyInfo(z->info);
    y->left=z->left;
    y->right=z->right;
    y->parent=z->parent;
    y->red=z->red;
 z->left->parent=z->right->parent=y;
    if (z == z->parent->left) {
      z->parent->left=y;
    } else {
      z->parent->right=y;
    }
    free(z);
  } else {
    tree->DestroyKey(y->key);
    tree->DestroyInfo(y->info);
    if (!(y->red)) RBDeleteFixUp(tree,x);
    free(y);
  }
#ifdef DEBUG_ASSERT
  Assert(!tree->nil->red,"nil not black in RBDelete");
#endif
}
```

140 lines of tricky C, even reusing existing functions

```
void RBDeleteFixUp(rb_red_blk_tree* tree, rb_red_blk_node* x) {
  rb_red_blk_node* root=tree->root->left;
  rb_red_blk_node* w;
  while( (!x->red) && (root != x)) {
    if (x == x->parent->left) {
      w=x->parent->right;
      if (w->red) {
            w->red=0;
            x->parent->red=1;
            LeftRotate(tree,x->parent);
            w=x->parent->right;
      }
      if ( (!w->right->red) && (!w->left->red) ) {
            w->red=1;
            x=x->parent;
      } else {
            if (!w->right->red) {
              w->left->red=0;
              w->red=1;
              RightRotate(tree,w);
              w=x->parent->right;
            }
            w->red=x->parent->red;
            x->parent->red=0;
            w->right->red=0;
            LeftRotate(tree,x->parent);
            x=root; /* this is to exit while loop */
      }
    } else { /* the code below is has left and right switched from above */
      w=x->parent->left;
      if (w->red) {
            w->red=0;
            x->parent->red=1;
            RightRotate(tree,x->parent);
            w=x->parent->left;
      }
      if ( (!w->right->red) && (!w->left->red) ) {
            w->red=1;
            x=x->parent;
      } else {
            if (!w->left->red) {
              w->right->red=0;
              w->red=1;
              LeftRotate(tree,w);
              w=x->parent->left;
            }
            w->red=x->parent->red;
            x->parent->red=0;
            w->left->red=0;
            RightRotate(tree,x->parent);
            x=root; /* this is to exit while loop */
      }
    }
  }
  x->red=0;
#ifdef DEBUG_ASSERT
```

remove using specifications: 2 lines

```
def remove(x : Int, t : Tree) = choose((t1:Tree) =>
  isRBT(t1) && content(t1)=content(t) – Set(x))
```

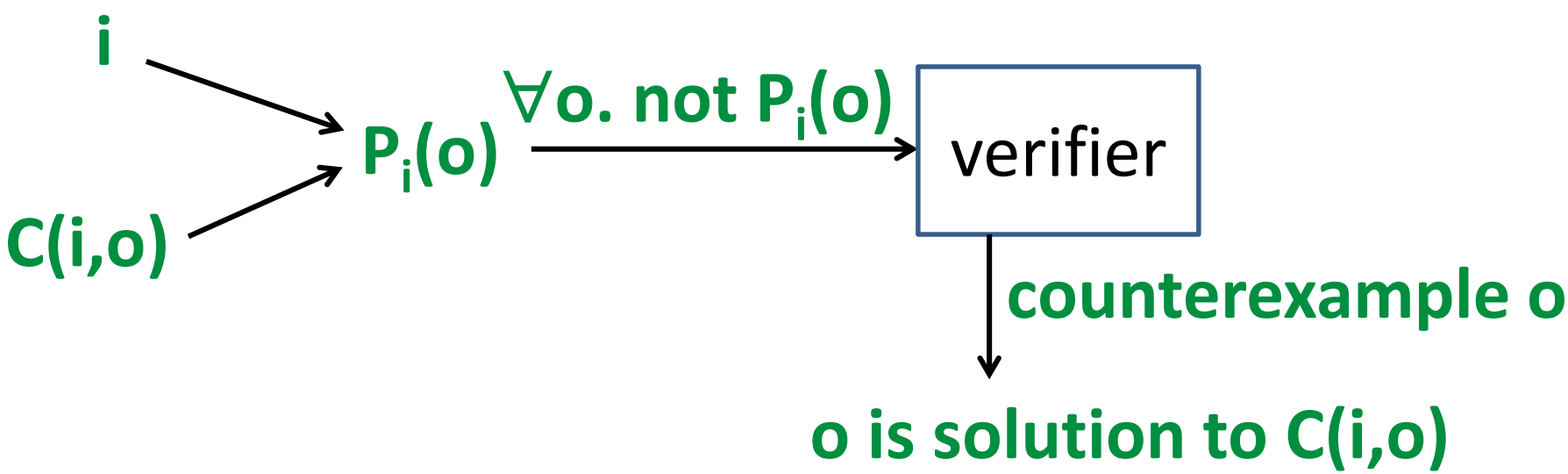
The biggest expected payoff:

properties are more reusable

Using Verifier to Solve Constraints

Goal: find a value that satisfies a given constraint:
find o such that C(i,o) (once **i** is known)

Method: use verification technology: try to prove that no such **o** exists, report counter-examples!



Approaches and Their Guarantees

both specification **C** and program **p** are given:

a) Check assertion while program **p** runs: **C(i,p(i))**

b) Verify that program always meets spec: **∀i. C(i,p(i))**

only specification **C** is given:

c) Constraint programming: once **i** is known, find **o** to satisfy a given constraint: **find o such that C(i,o)**

run-time

d) Synthesis: solve **C** symbolically to obtain program **p** that is correct by construction, for all inputs: **find p such that ∀i.C(i,p(i))** **i.e.** **p ⊆ C**

compile-time

Synthesis for Theories

assert(i % 2 == 1)

$3\,i + 2\,o = 13$ → $o = (13 - 3\,i)/2$

- Wanted: "equation solving" for programs
  - for linear integer equations: extended Euclid's algorithm
  - need to handle disjunctions, negations, more data types
- For every formula in e.g. Presburger arithmetic
  - synthesis algorithm terminates
  - produces the most general precondition (assertion characterizing when the result exists)
  - generated code always terminates and gives correct result
- If there are multiple or no solutions for some input parameters, the algorithm identifies those inputs
- Works not only for arithmetic but also for e.g. sets with sizes and for trees
- Goal: lift everything done for SMT solvers to synthesizers

Time Conversion Example

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
  choose((h: Int, m: Int, s: Int) ⇒ (
        h * 3600 + m * 60 + s == totalSeconds
    && h ≥ 0
    && m ≥ 0 && m < 60
    && s ≥ 0 && s < 60    ))
```

close to a wish

could infer from types

automated synthesis

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
  val t1 = totalSeconds div 3600
  val t2 = totalSeconds -3600 * t1
  val t3 = t2 div 60
  val t4 = totalSeconds - 3600 * t1 - 60 * t3
  (t1, t3, t4)
```

Kuncak, Mayer, Piskac, Suter: PLDI'10, STTT'12, CACM'12

Date Conversion in C

Knowing number of days since 1980, find current year and day

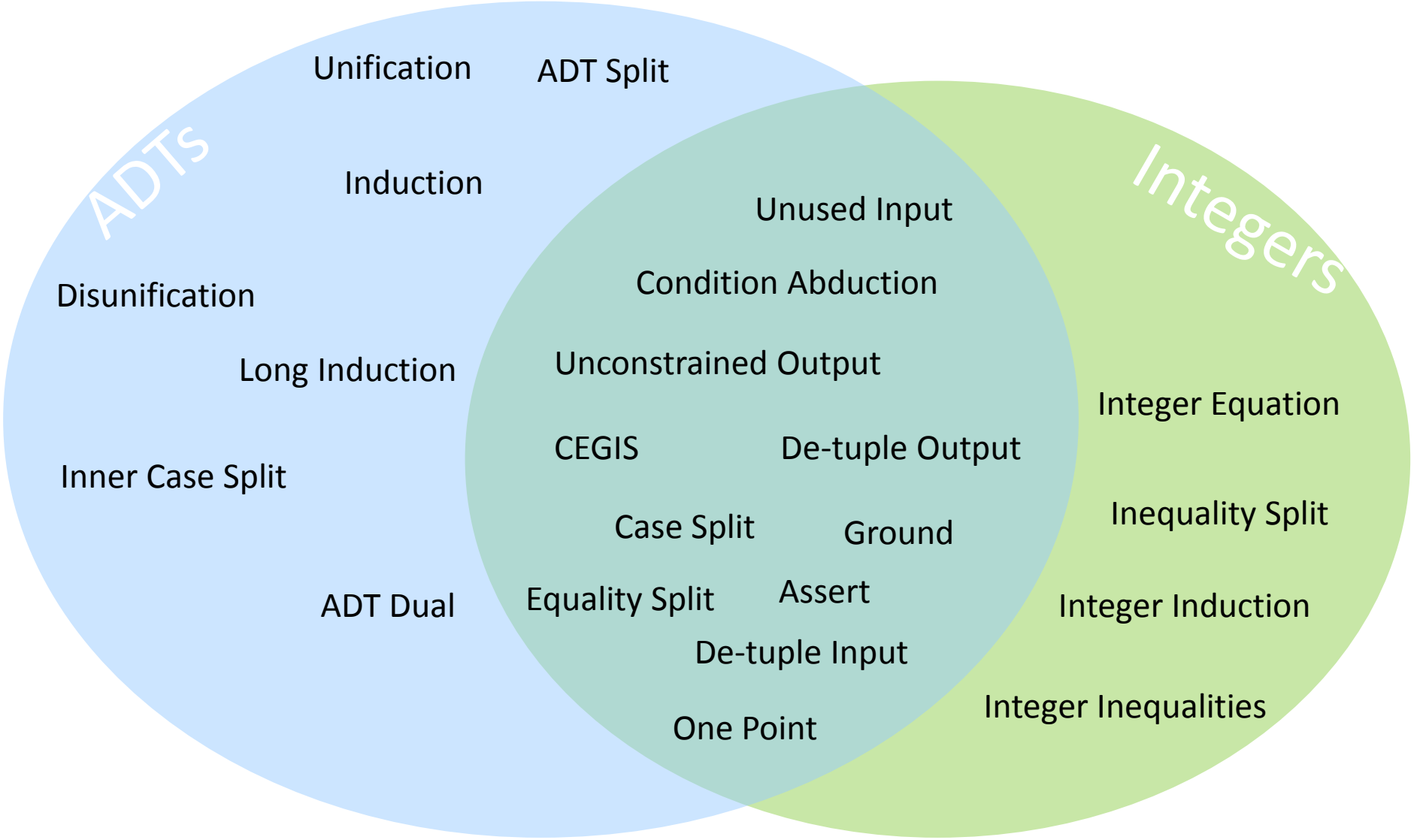
```
BOOL ConvertDays(UINT32 days) {
 year = 1980;
 while (days > 365) {
  if (IsLeapYear(year)) {
   if (days > 366) {
    days -= 366;
    year += 1;
   }
  } else {
   days -= 365;
   year += 1;
  } ...
 }
```

Enter December 31, 2008

All music players (of this brand) froze in the boot sequence.

Synthesis can generate terminating and correct solution from a spec.

Deduction Rules within Framework



Recursion Schemas + STE in Action

```
def delete(in1: List, v: Int) = choose {
  (out: List) => content(out) == content(in1) -- Set(v)
}
def delete(in1: List,  v: Int)  = {
  def rec(in: List, v: Int): List = in match {
    case Cons(h,t) =>
      val r = rec(t,v)
      if (h == v) {
CEGIS   r
      } else {
CEGIS   Cons(h, r)
      }
    case Nil =>
CEGIS   Nil
  } ensuring { content(_) == content(in1) -- Set(v) }
  rec(in1, v)
}
```

ADT Induction

EQ Split

Synthesis of Data Structure Manipulation

Operation	Size	Calls	Proof	sec.
`List.Insert`	3	0	√	0.6
`List.Delete`	19	1	√	1.8
`List.Union`	12	1	√	2.1
`List.Diff`	12	2	√	7.6
`List.Split`	27	1	√	9.3
`SortedList.Insert`	34	1		9.9
`SortedList.InsertAlways`	36	1	√	7.2
`SortedList.Delete`	23	1	√	4.1
`SortedList.Union`	19	2	√	4.5
`SortedList.Diff`	13	2	√	4.0
`SortedList.InsertionSort`	10	2	√	4.2
`SortedList.MergeSort`	17	4	√	14.3
`StrictSortedList.Insert`	34	1	√	14.1
`StrictSortedList.Delete`	21	1		15.1
`StrictSortedList.Union`	19	2	√	3
`UnaryNumerals.Add`	11	1	√	1.3
`UnaryNumerals.Distinct`	12	0	√	1.1
`UnaryNumerals.Mult`	12	1	√	2.7
`BatchedQueue.Checkf`	14	4	√	7.4
`BatchedQueue.Snoc`	7	2	√	3.7
`AddressBook.Make`	50	14		8.8
`AddressBook.MakeHelpers`	21	5		4.9
`AddressBook.Merge`	11	3		8.9

Techniques used:

- symbolic exploration of the space of programs
- synthesis based on type inhabitation
- fast falsification by executing candidates on test inputs
- Leon's verification capabilities
- synthesis for theory of trees and integers
- recursion schemas
- case splitting
- learning conditional expressions
- cost-based search over possible synthesis steps

OOPSLA 2013

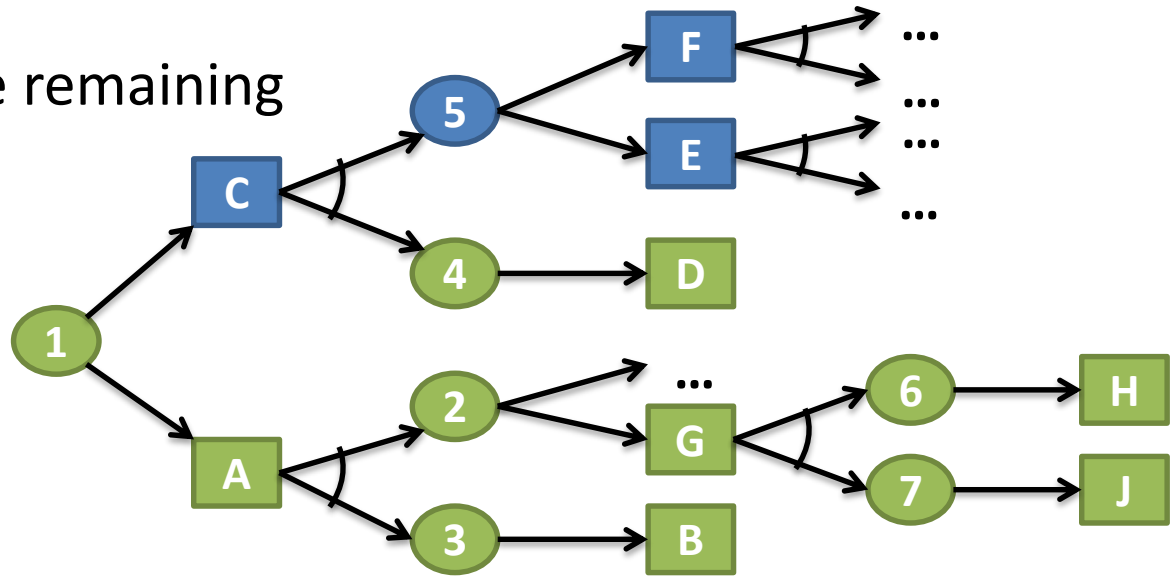
Synthesis and Constraint Solving

If we did not find an expression that solves it in all cases, we emit a runtime call to solver

Result: solver invoked only in some cases

- for some components of result
- for some conditions on inputs

after timeout, close the remaining branches by inserting a runtime solver call



Recent work: Automated Repair

```
def desugar(e : Trees.Expr) : SimpleE = { e match {
  case Trees.Plus (lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
  case Trees.Minus(lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
  case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
  case Trees.And  (lhs, rhs) => Ite(desugar(lhs), desugar(rhs), Literal(1))
  case Trees.Or   (lhs, rhs) => Ite(desugar(lhs), Literal(1), desugar(rhs))
  case Trees.Not(e) => Ite(desugar(e), Literal(0), Literal(1))
  case Trees.Eq(lhs, rhs) => Eq(desugar(lhs), desugar(rhs))
  case Trees.Ite(cond, thn, els) => Ite(desugar(cond), desugar(thn), desugar(els))
  case Trees.IntLiteral(v)  => Literal(v)
  case Trees.BoolLiteral(b) => Literal(b2i(b))
 }} ensuring { res =>
  sem(res) == Semantics.semUntyped(e)
 }
```

Equivalence-preserving transformation of a program into simpler form

Recent work: Automated Repair

```
def desugar(e : Trees.Expr) : SimpleE = { e match {
  case Trees.Plus (lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
  case Trees.Minus(lhs, rhs) => Plus(desugar(lhs), Neg(desugar(rhs)))
  case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
  case Trees.And  (lhs, rhs) => Ite(desugar(lhs), desugar(rhs), Literal(1))
  case Trees.Or   (lhs, rhs) => Ite(desugar(lhs), Literal(1), desugar(rhs))
  case Trees.Not(e) => Ite(desugar(e), Literal(0), Literal(1))
  case Trees.Eq(lhs, rhs) => Eq(desugar(lhs), desugar(rhs))
  case Trees.Ite(cond, thn, els) => Ite(desugar(cond), desugar(thn), desugar(els))
  case Trees.IntLiteral(v)  => Literal(v)
  case Trees.BoolLiteral(b) => Literal(b2i(b))
 }} ensuring { res =>
  sem(res) == Semantics.semUntyped(e)
 }
```

Equivalence-preserving transformation of a program into simpler form

Recent work: Automated Repair

```
def desugar(e : Trees.Expr) : SimpleE = { e match {
  case Trees.Plus (lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
  case Trees.Minus(lhs, rhs) => Plus(desugar(lhs), Neg(desugar(rhs)))
  case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
  case Trees.And  (lhs, rhs) => Ite(desugar(lhs), desugar(rhs), Literal(0))
  case Trees.Or   (lhs, rhs) => Ite(desugar(lhs), Literal(1), desugar(rhs))
  case Trees.Not(e) => Ite(desugar(e), Literal(0), Literal(1))
  case Trees.Eq(lhs, rhs) => Eq(desugar(lhs), desugar(rhs))
  case Trees.Ite(cond, thn, els) => Ite(desugar(cond), desugar(thn), desugar(els))
  case Trees.IntLiteral(v)  => Literal(v)
  case Trees.BoolLiteral(b) => Literal(b2i(b))
 }} ensuring { res =>
  sem(res) == Semantics.semUntyped(e)
 }
```

After automated repair, it verifies with respect to a very strong spec

Equivalence-preserving transformation of a program into simpler form

Further Contributors to Leon

Regis Blanc
EPFL

Philippe Suter
IBM Research
(Leon founder)

Ali Sinan Koeksal
Berkeley

Ivan Kuraj
MIT

Eva Darulova

Ravichandhran Madhavan

Nicolas Voirol

Synthesis of Invariants

Given templates, find inductive invariants in Leon

Combination of

- Farkas lemma for encoding forall-exists into exists for non-linear problems
- Encoding of algebraic data types
- Incremental, on-demand handling of disjuncts and recursive functions
- Specialized techniques for resource bounds (such as sequential and parallel execution time)

CAV'14

Ravichandhran Madhavan

Synthesis of Real Computation in Real Software

```
def turbine(v: Real, w: Real, r: Real): Real = {
  3 + 2/(r*r) - 0.125*(3-2*v)*(w*w*r*r)/(1-v) - 4.5
}
```

Real compiler

```
def turbine(v: Int, w: Int, r: Int): Int = {
    val tmp1 = (r * r) >> 15
    val tmp2 = (16384 << 11) / tmp1
    val tmp3 = ((24576 << 2) + tmp2) >> 2
    val tmp4 = (16384 * v) >> 14
    val tmp5 = (24576 - (tmp4 << 2)) >> 2
    val tmp6 = (16384 * tmp5) >> 14
    val tmp7 = (w * w) >> 15
    val tmp8 = (tmp7 * r) >> 15
    val tmp9 = (tmp8 * r) >> 15
    val tmp10 = (tmp6 * tmp9) >> 15
    val tmp11 = (16384 - (v << 2)) >> 2
    val tmp12 = (tmp10 << 14) / tmp11
    val tmp13 = (tmp3 - (tmp12 << 3)) >> 2
    val tmp14 = ((tmp13 << 1) - 18432) >> 2
    tmp14
  }
```

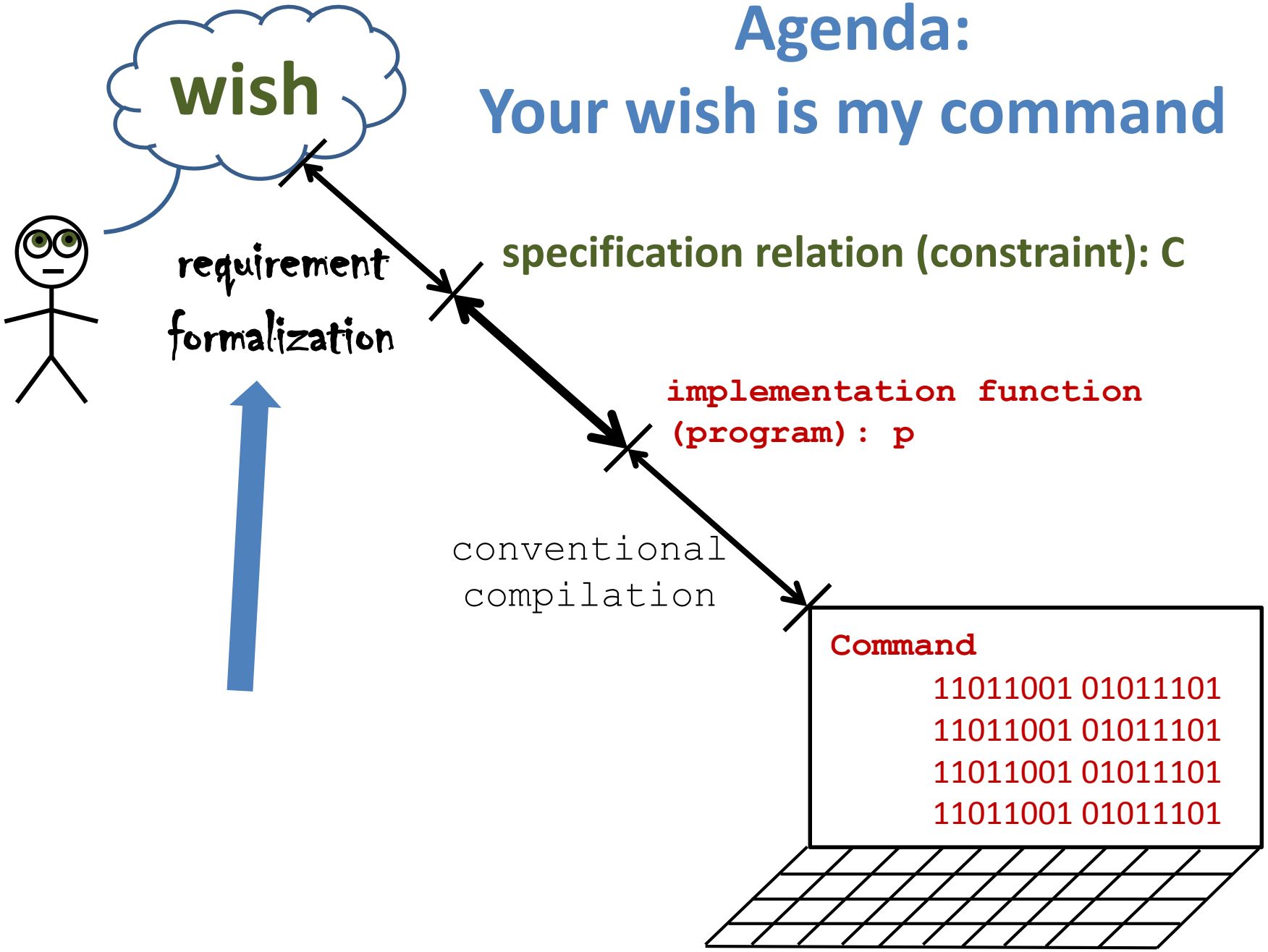
```
def turbine(v: Double, w: Double,
            r: Double): Double = {
  3 + 2/(r*r) -
   0.125*(3-2*v)*(w*w*r*r)/(1-v) - 4.5
}
```

- affine arithmetic
- non-linear solvers
- optimization
- symbolic algebra

POPL 2014

Eva Darulova

Agenda:
Your wish is my command



Synthesizing Code from Free-Form Queries

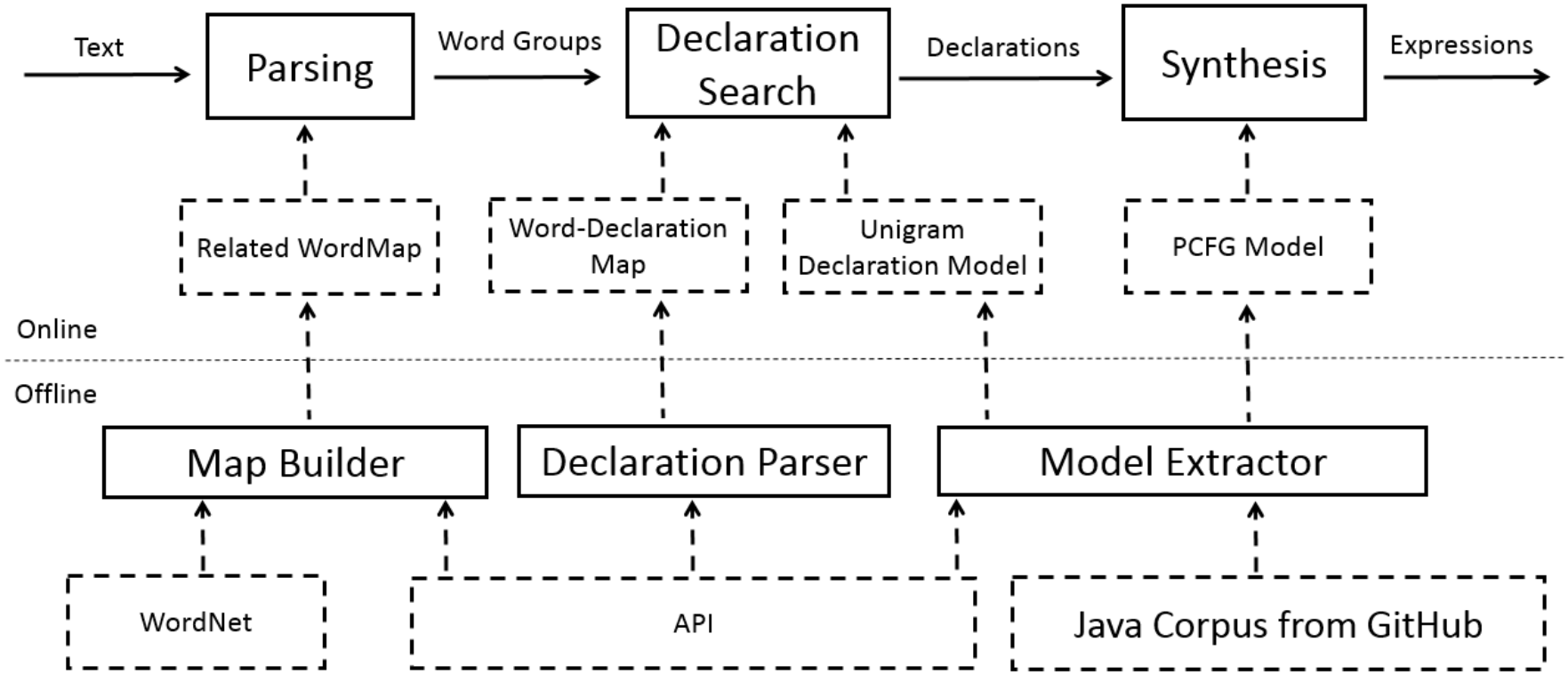
```
public class Utils {
    public void backupFile(String fname) {
        String bname = fname+".bak";

    }
}
```

copy file fname to bname

- FileUtils.copyFile(new File(fname), new File(bname))
- FileUtils.copyFile(new File(bname), new File(fname))
- FileUtils.copyFileToDirectory(new File(fname), new File(bname))
- FileUtils.copyFileToDirectory(new File(bname), new File(fname))
- FileUtils.copyFile(<arg>, new File(fname))

Tihomir Gvero



"Your wish my command" agenda