

Implicit Programming

Viktor Kunčák

<http://lara.epfl.ch>



GOAL

Help people construct software
that does what they expect.

Personal approach:

- embrace modern programming languages
- focus on algorithms and tools

Problem

Programming is hard, because computation is given *explicitly* (*how*)

Claim:

We can make it easier, if we support *implicit* computation (*what*)

human intentions



Implicit Programming

GAP



Existing Technology



human intentions



IMPRO

2) Empowering Users

specifications

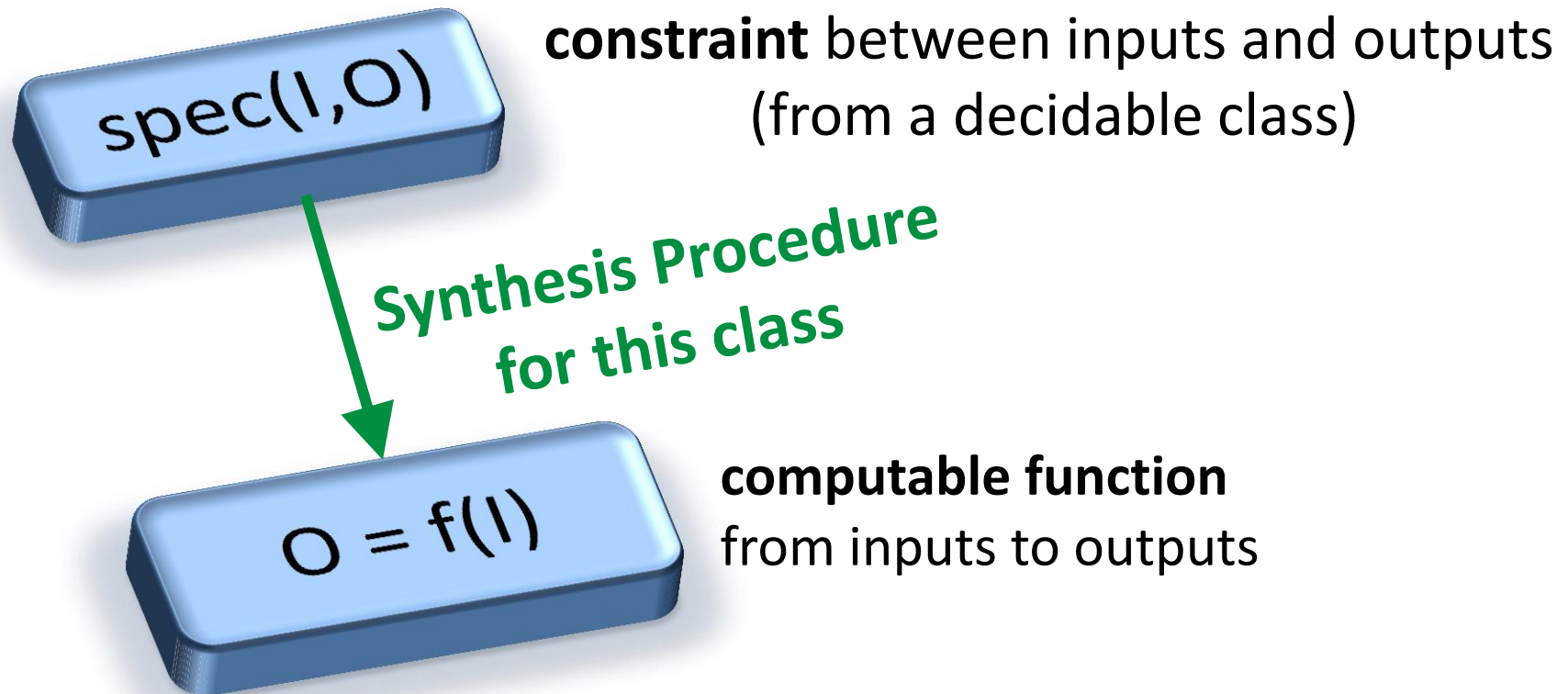
1) Synthesis Procedures

functional languages

computational
realizations

1) Synthesis Procedures

= compiler for specifications

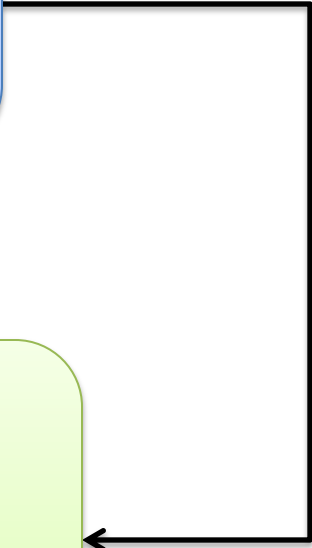


Numeric domains: **linear integers, reals** (PLDI'10, OOPSLA'11)

Symbolic domains: **Calculus of Data Structures** (VMCAI'10, CSL'10)

Synthesis for Arithmetic

choose ((h, m, s) \Rightarrow
 $h * 3600 + m * 60 + s == \text{totalSeconds}$
 $\wedge h \geq 0$
 $\wedge m \geq 0 \wedge m < 60$
 $\wedge s \geq 0 \wedge s < 60$)



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)

Implemented as an extension of the Scala compiler.

Properties of Synthesis Algorithm

- For every formula in linear integer arithmetic
 - synthesis algorithm terminates
 - produces the most general precondition (assertion saying when result exists)
 - generated code gives correct values whenever correct values exist
- If there are multiple or no solutions for some parameters, we get a warning
- Extended to arithmetic pattern matching
- Extended to sets with cardinalities
- Handling bitwise operations (FMCAD'10, IJCAR'12)

Extensions to Data Structures

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

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

The biggest expected payoff:

declarative knowledge is more reusable

Technology: Constraint Solving

- **Run-time checking:** very useful: $C(t)$
- **Verification** that functions meet contracts or algebraic statements: $\forall x. C(x)$
- **Falsification:** produce a counterexample when the verification fails: **find x such that $\neg C(x)$**
- **Computation:** compute any (best) value that satisfies a given constraint: **find x such that $C(x)$**
- **Test generation:** enumerate inputs that satisfy given precondition: **find all x such that $C(x)$**
- **Synthesis:** specialize constraint solver for a given constraint: **find f such that $\forall x. C(x, f(x))$**

Two Constraint Solving Techniques

1) Constraints with recursive functions:

Leon algorithm for solving
computable constraints
(POPL'10, SAS'11, POPL'12)

2) Constraints on bags, sets, and sizes: optimal complexity through sparse encoding into integer linear programs (CADE'07, VMCAI'08, CAV'08)

General Form of Recursive Functions

Extend logic of recursive data structures (q.f. term algebra with):

$\alpha : \text{Tree} \rightarrow C$

$\alpha(\text{Nil}) = \mathbf{empty}$

$\mathbf{empty} : C$

$\alpha(\text{NODE}(l, v, r)) = \mathbf{combine}(\alpha(l), v, \alpha(r))$

$\mathbf{combine} : (C, E, C) \rightarrow C$

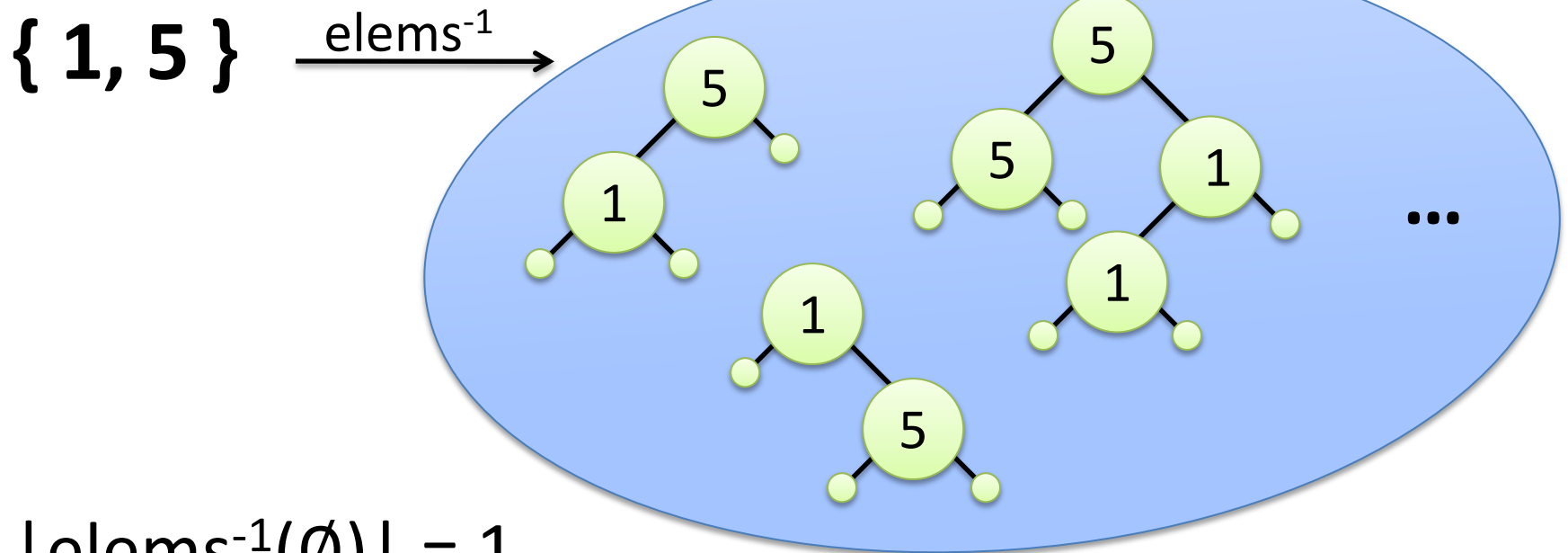
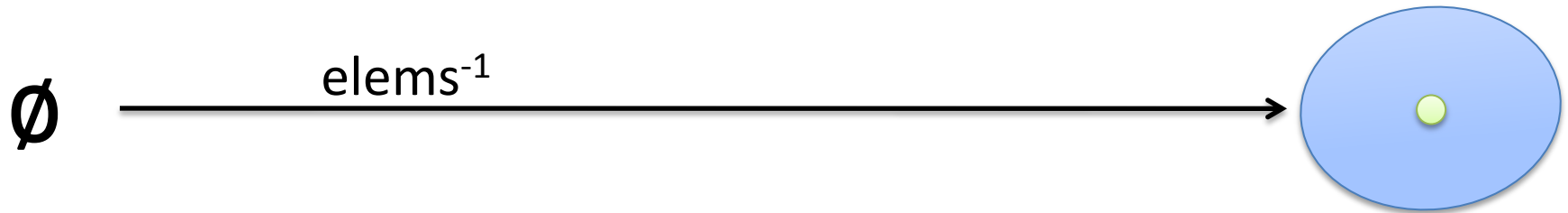
e.g. $\alpha(l) \cup \{v\} \cup \alpha(r)$

Many operations have this form:

- elements, size, height, max element, is-sorted, free variables in a syntax tree
- moreover, they are **sufficiently surjective**

Example: **elems** function that computes the set of elements stored in the tree.

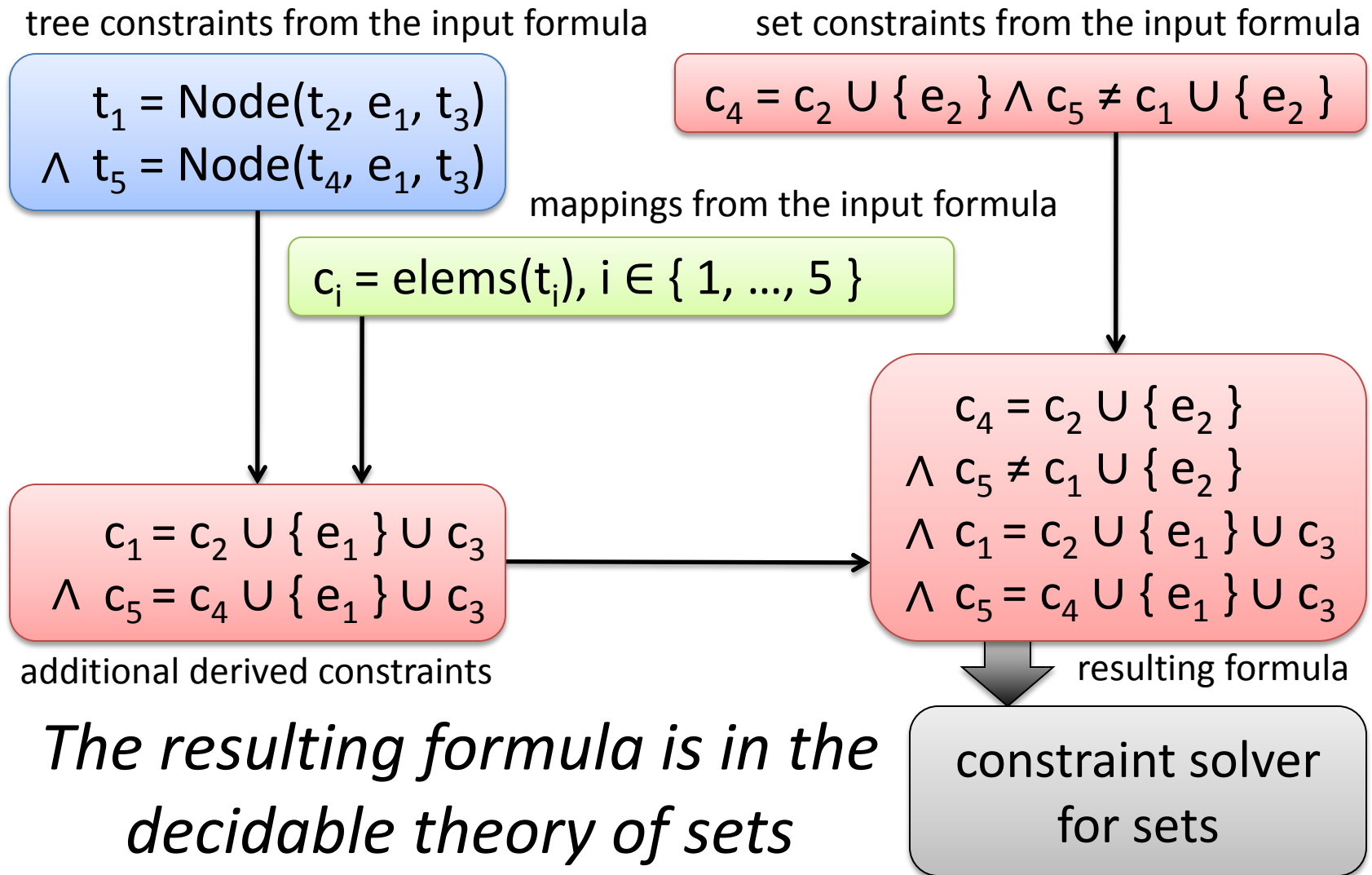
'Surjectivity' of Set Abstraction



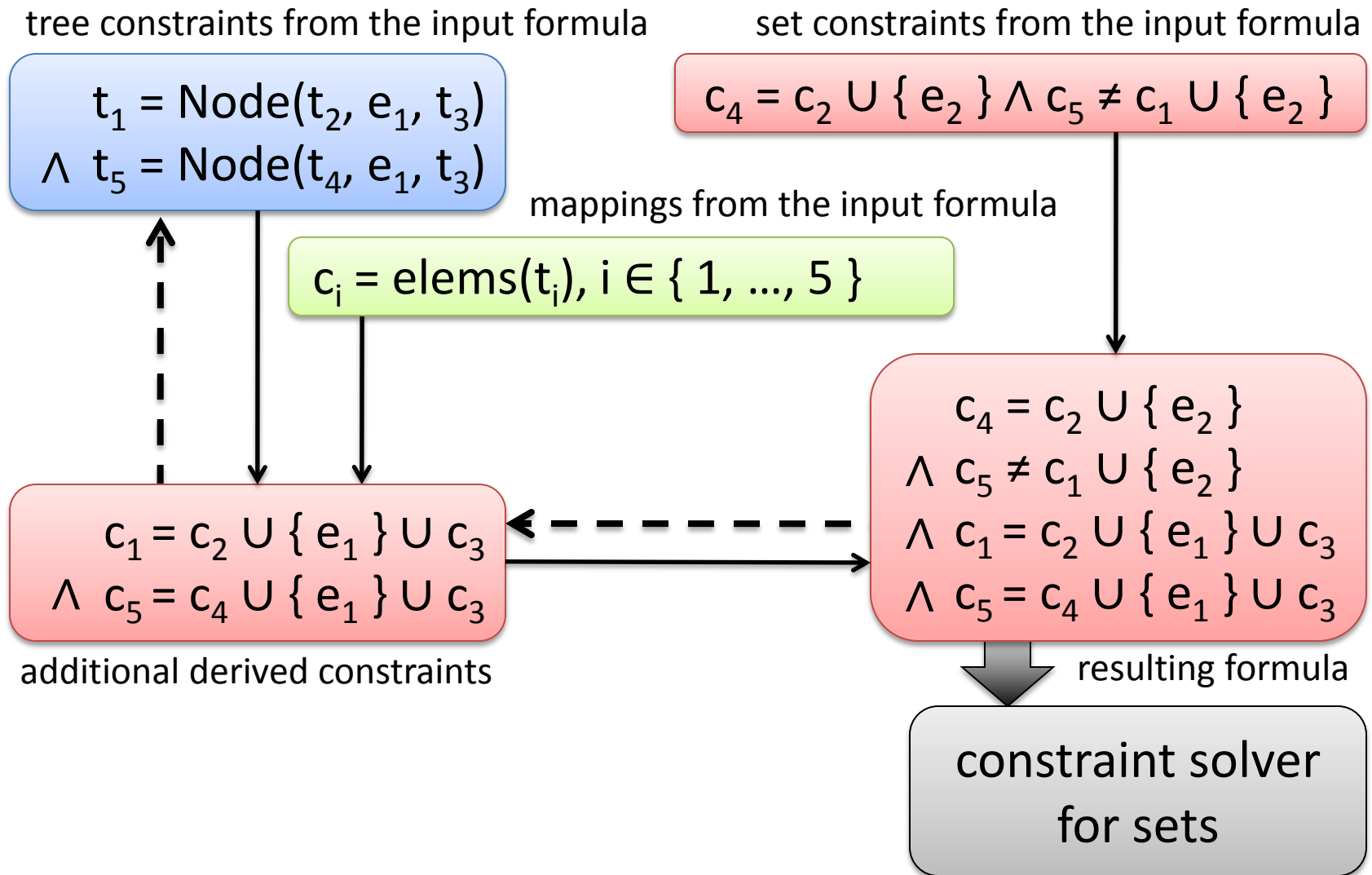
$$|\text{elems}^{-1}(\emptyset)| = 1$$

$$|\text{elems}^{-1}(\{1, 5\})| = \infty$$

Propagating Constraints to Sets



Mapping Solutions Back



human intentions



2) Empowering Users

specifications

1) Synthesis Procedures ✓

functional languages

Interactive Synthesis within an IDE

<http://lara.epfl.ch/w/insynth>

```
import java.io._

object Main {
  def main(args:Array[String]) = {

    var body = "email.txt"
    var sig = "signature.txt"

    var inStream:SequenceInputStream =

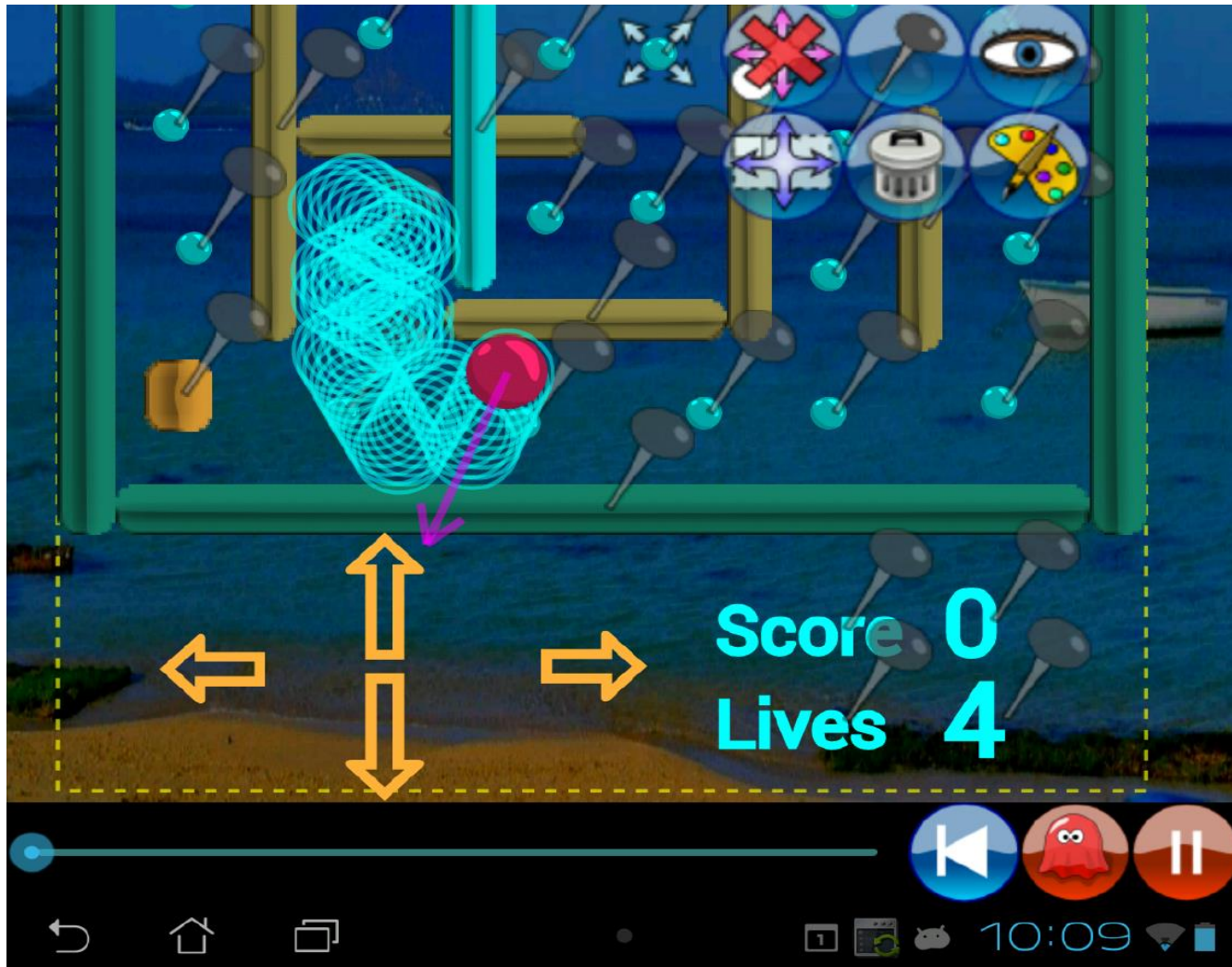
    var eof:Boolean = false;
    var byteCount:Int = 0;
    while (!eof) {
      var c:Int = inStream.read()
      if (c == -1)
        eof = true;
      else {
        System.out.print(c.toChar);
        byteCount+=1;
      }
    }
    System.out.println(byteCount + " bytes were read");
    inStream.close();
  }
}
```

```
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(sig))
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(body))
new SequenceInputStream(new FileInputStream(body), new FileInputStream(sig))
new SequenceInputStream(new FileInputStream(body), new FileInputStream(body))
new SequenceInputStream(new FileInputStream(sig), System.in)
```

Press 'Ctrl+Space' to show Default Proposals

Programming by Demonstration

Graphical editing of not only terrain but also behavior
Indicate/correct the desired actions by demonstrating them



Results for Fully Automatic Verification

Benchmark	LoC	#Funs.	#VCs.	Time (s)
ListOperations	107	15	27	0.76
AssociativeList	50	5	11	0.23
InsertionSort	99	6	15	0.42
RedBlackTrees	117	11	24	3.73
PropositionalLogic	86	9	23	2.36
AmortizedQueue	124	14	32	3.37
	677	71	155	11.66

Functional correctness properties of data structures: red-black trees implement a set and maintain height invariants, associative list has read-over-write property, insertion sort returns a sorted list of identical content, amortized queue implements a balanced queue, etc.

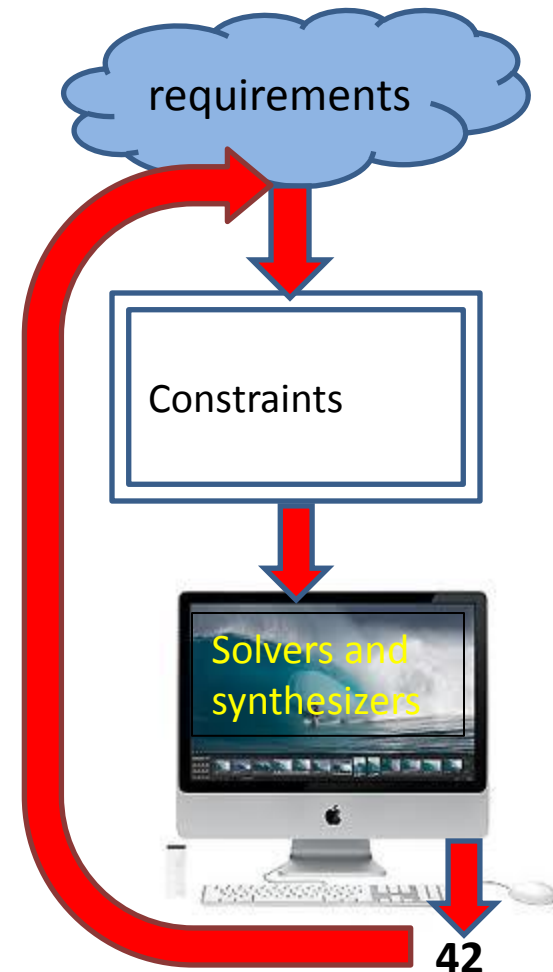
Conclusions

Project: Implicit Programming

- Compiling Specifications
- Helping People Construct Specifications

Expertise: advancing constraint solving theory and practice to enable

- **Verification**
- **Falsification**
- **Computation**
- **Test generation**
- **Synthesis**
- **Development within IDE**
- **End-user programming**



Thank you.

Viktor Kuncak
lara.epfl.ch
ic.epfl.ch

