Implicit Programming
through Automated Reasoning

presented by Viktor Kuncak

Swiss Federal Institute of Technology, Lausanne

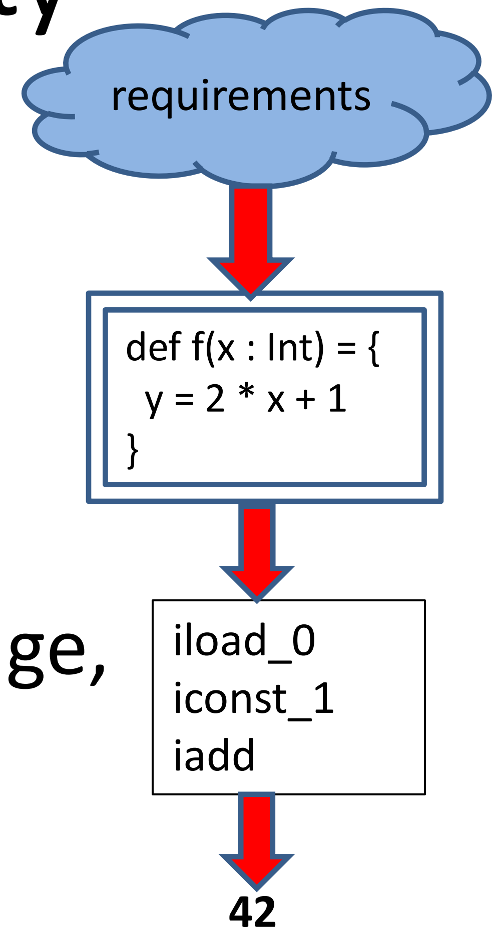
http://lara.epfl.ch/w/impro

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Programming Activity

Consider three related activities:

- Development within an IDE (Eclipse, Visual Studio, emacs, vim)
- Compilation and static checking (optimizing compiler for the language, static analyzer, contract checker)
- Execution on a (virtual) machine



More compute power available for each of these

→ use it to improve programmer productivity

Implicit Programming

- A high-level declarative programming model
- In addition to traditional recursive functions and loops, use relations,

  implicit specifications

  give property of result, not how to compute it
- More expressive, easier to argue correctness
- Challenge:
  - make it executable and efficient so it is useful
- Claim: automated reasoning is key technique

The **choose** Implicit Construct

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

3787 seconds $\longrightarrow$ 1 hour, 3 mins. and 7 secs.

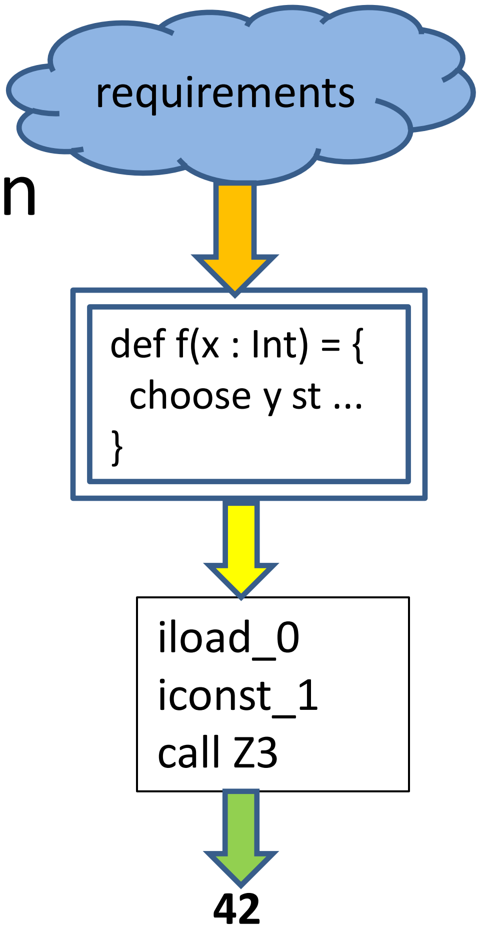
Notions Related to Implicit Programing

- Code completion
  - help programmer to interactively develop the program
- Synthesis – core part of our vision
  - key to compilation strategies for specification constructs
- Manual refinement from specs (Morgan, Back)
- Logic Programming
  - shares same vision, in particular CLP(*X*)
  - operational semantics design choices limit what systems can do (e.g. Prolog)
  - CLP solvers theories limited compared to SMT solvers
  - not on mainstream platforms, no curly braces ☺, SAT

Relationship to Verification

- Some functionality is best synthesized from specs
- Others are perhaps best implemented, then verified
- But currently, no choice - always must implement
  - so specifications viewed as overhead
- Goal: make specifications intrinsic part of program, with clear benefits to programmers – execution
- Expectation: this will help both
  - verifiability and
  - productivity
- example: state assertion, not how to establish it

Implicit Programming at All Levels



Opportunities for implicit programming in

- **Development** within an IDE
  - **isynth** tool
- **Compilation**
  - **Comfusy** and **RegSy** tools
- **Execution**
  - **Scala^Z3** and **UDITA** tools

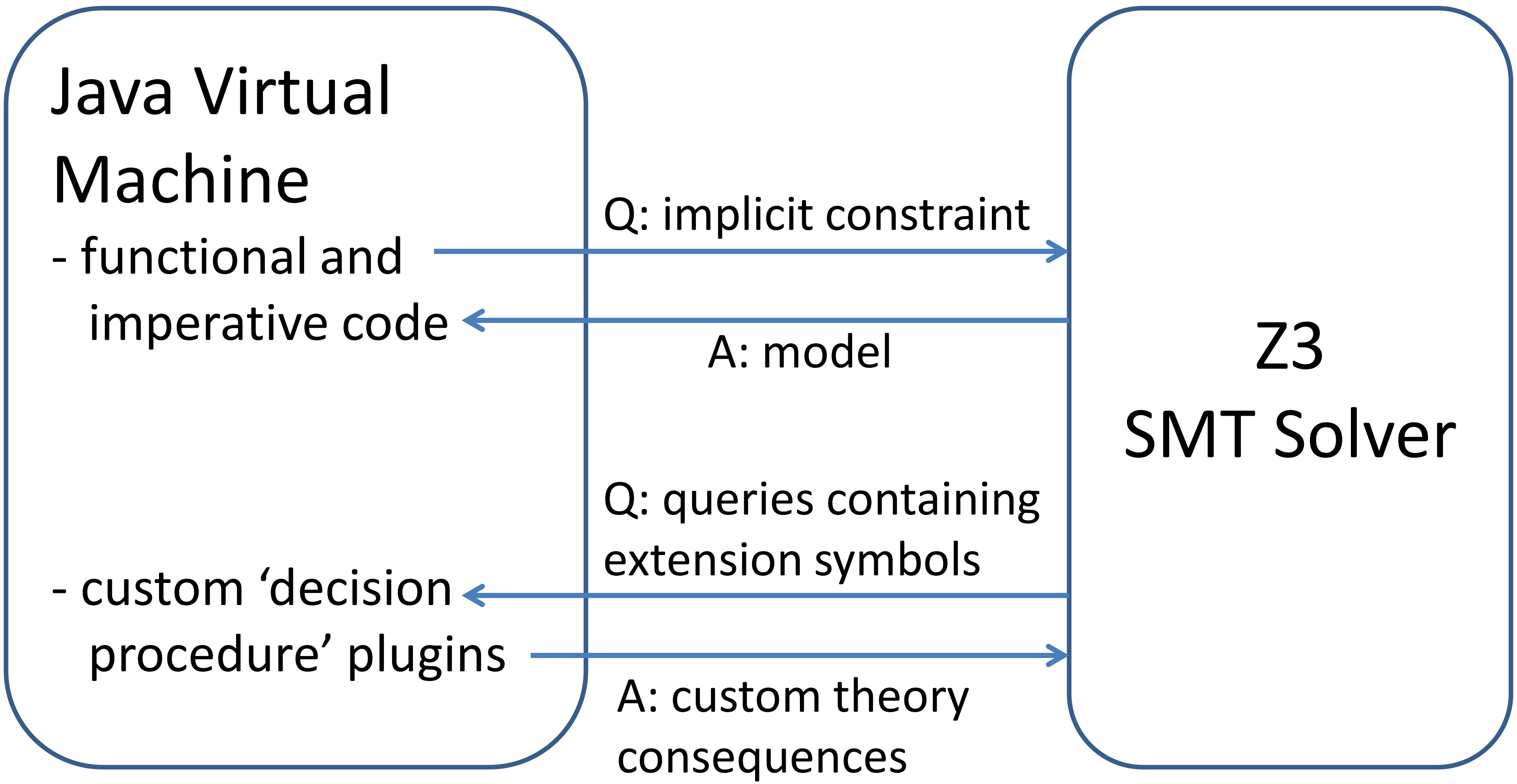
I next examine these tools, from last to first, focusing on Compilation

Execution of Implicit Constructs
- constraint programming

Scala^Z3,   UDITA

Scala^Z3

Invoking Constraint Solver at Run-Time



with: **Philippe Suter**, **Ali Sinan Köksal**, **Robin Steiger**

Executing **choose** using Z3

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

will be constant at run-time

syntax tree constructor

3787 seconds ⟶ 1 hour, 3 mins. and 7 secs.

It works, certainly for constraints within Z3's supported theories

Implemented as a library (jar + z3.so / dll) – no compiler extensions

Programming in Scala^Z3

find triples of integers x, y, z such that x > 0, y > x,
2x + 3y <= 40, x · z = 3y^2, and y is prime

model enumeration (currently: negate previous)

```
val results = for(
    (x,y) ← findAll((x: Var[Int], y: Var[Int]) ) => x > 0 && y > x && x * 2 + y * 3 <= 40);
     if isPrime(y);
    z ← findAll((z: Var[Int]) ) => x * z === 3 * y * y))
  yield (x, y, z)
```

user's Scala function

λ

Scala's existing mechanism for composing iterations
(reduces to standard higher order functions such as flatMap-s)

Use Scala syntax to construct Z3 syntax trees
- a type system prevents certain ill-typed Z3 trees

Obtain models as Scala values

Can also write own plugin decision procedures in Scala

UDITA: system for Test Generation

```
void generateDAG(IG ig) {
 for (int i = 0; i < ig.nodes.length; i++) {
  int num = chooseInt(0, i);
  ig.nodes[i].supertypes = new Node[num];
  for (int j = 0, k = −1; j < num; j++) {
   k = chooseInt(k + 1, i − (num − j));
   ig.nodes[i].supertypes[j] = ig.nodes[k];
  } } }
```

Java + choose
- integers
- (fresh) objects

We used to it to find real bugs in
javac, JPF itself, Eclipse, NetBeans refactoring

On top of Java Pathfinder's backtracking mechanism

Can enumerate all executions

Key: suspended execution of non-determinism

with: **M. Gligoric**, **T. Gvero, V. Jagannath, D. Marinov, S. Khurshid**

Implemented and released in official Java PathFinder

projects/jpf-delayed – Java P...

JPF .. the swiss army knife of Java™ verification

JPF-Wiki | Timeline | Roadmap | View Tic

wiki: projects / jpf-delayed

jpf-delayed

Milos Gligoric and Tihomir Gvero, {milos.gligoric, tihomir.gvero}@gmail.com, January 2010

Repository

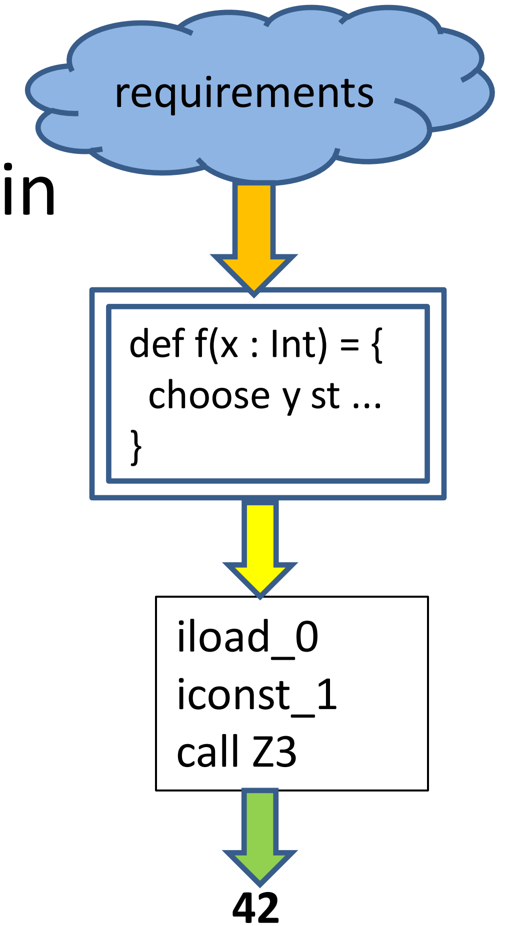
The repository for jpf-delayed is http://babelfish.arc.nasa.gov/hg/jpf/jpf-delayed.

Delayed Choice

The basic ***delayed choice*** postpones non-deterministic choice of values until they are used, reducing the size of the search tree. The technique works with both int and boolean, i.e., with Verify.getInt and Verify.getBoolean methods. Additionally, we speed up the basic ***delayed choice*** by introducing copy propagation that keeps non-deterministic values symbolic even if they are copied through memory locations. We also implement a special class for linked structures, called ObjectPool, which has the following methods for non-deterministic assignments of objects:

```
public final class ObjectPool<T> implements Iterable<T> {
  public ObjectPool(Class<?> clz, int size, boolean includeNull) {...}
  public T getAny() {...}
  public T getNew() {...}
  public Iterator<T> iterator() {...}
}
```

Implicit Programming at All Levels

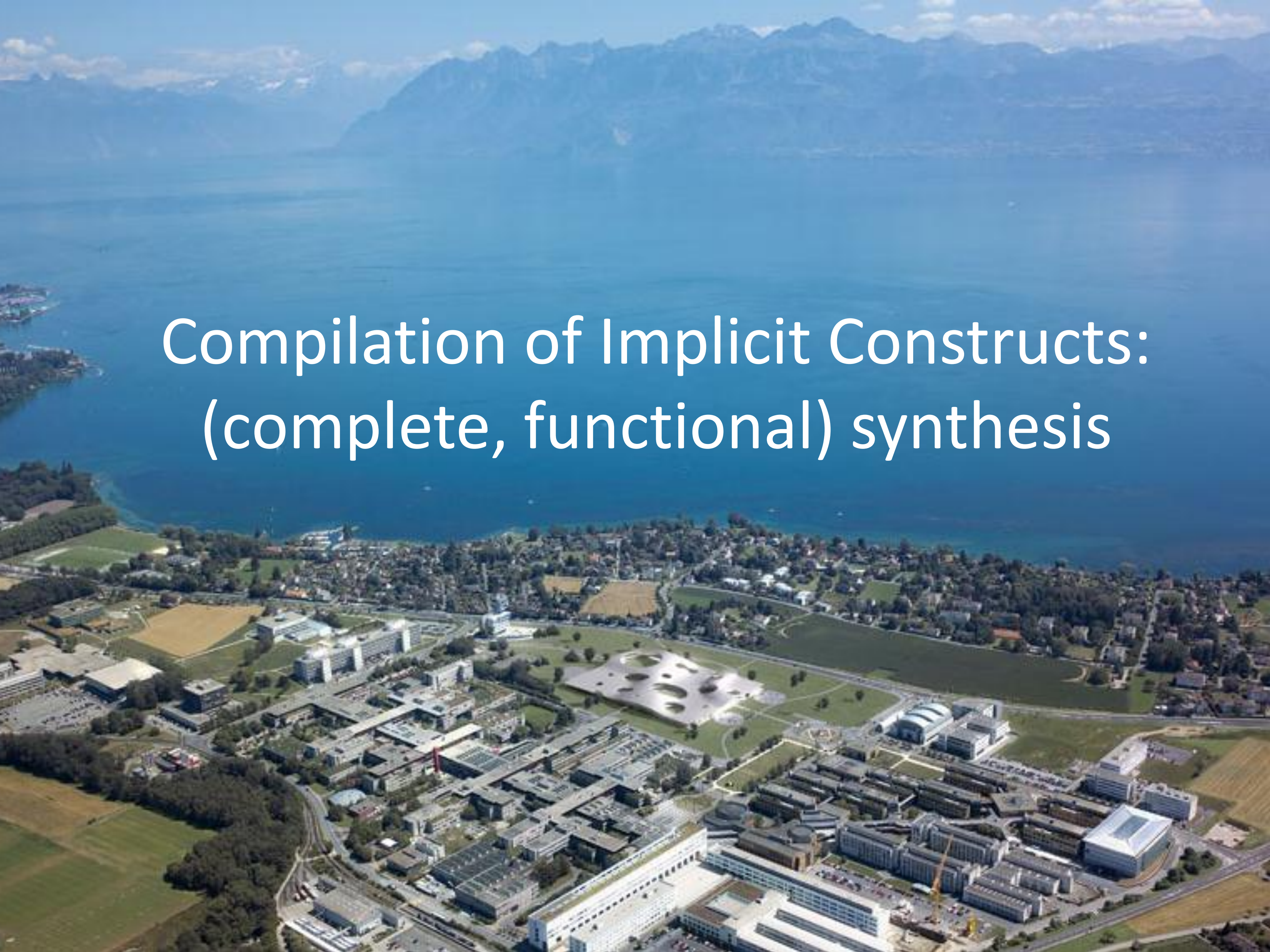


Opportunities for implicit programming in

- **Development** within an IDE
  - **isynth** tool
- **Compilation**
  - **Comfusy** and **RegSy** tools
- √ **Execution**
  - **Scala^Z3** and **UDITA** tools

I next examine these tools, from last to first, focusing on Compilation

Compilation of Implicit Constructs: (complete, functional) synthesis



An 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    ))
```

3787 seconds ⟶ 1 hour, 3 mins. and 7 secs.

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
  val t1 = totalSeconds div 3600
  val t2 = totalSeconds + ((-3600) * t1)
  val t3 = min(t2 div 60, 59)
  val t4 = totalSeconds + ((-3600) * t1) + (-60 * t3)
  (t1, t3, t4)
```

Comparing with runtime invocation

Pros of runtime invocation

- Conceptually simpler
- Can use off-the-shelf solver
- for now can be more expressive and even faster
- but:

```
val times =
  for (secs ← timeStats)
    yield secondsToTime(secs)
```

Pros of synthesis

- Change in complexity: time is spent at compile time
- Solving most of the problem only once
- *Partial evaluation*: we get a specialized decision procedure
- No need to ship a decision procedure with the program

Our approach

- Synthesis as programming language construct

```
...
val x = readInteger() + 4

val r = choose(y ⇒ 5*x + 7*y = 31)

println("r^2: " + r*r)
...
```

standard code

implicit code

- Like compilation, synthesis should *always succeed*

Turn *decision procedures* into *synthesis procedures*

Decision vs. synthesis procedures

for a well-defined class of formulas

Decision procedure (model-generating)

a theorem prover that always succeeds

- **Takes**: a formula
- **Makes**: a model of the formula

$5*x + 7*y = 31$

↓

$x := 2$

$y := 3$

Synthesis procedure

a synthesizer that always succeeds

- **Takes**: a formula, with *input (params)* and *output* variables
- **Makes**: a program to compute *output* values from *input* values

Inputs: { x } outputs: { y }

$5*x + 7*y = 31$

↓

$y := (31 – 5*x) / 7$

Complete Functional Synthesis

- **Synthesis**: our procedures start from an implicit specification.
- **Functional**: computes a function that satisfies a given input/output relation.
- **Complete**: guaranteed to work for all specification expressions from a well-defined class.

Tool: **Comfusy**

Mikaël Mayer, **Ruzica Piskac, Philippe Suter,** in PLDI 2010, CAV 2010

Possible starting point: quantifier elimination

- A specification statement of the form

$$\vec{r} = \boldsymbol{choose}(\vec{x} \Rightarrow F(\vec{a}, \vec{x}))$$

"let r *be* x *such that* F(a, x) *holds"*

- Corresponds to constructively solving the **quantifier elimination** problem

$$\exists \vec{x}\,.\, F(\vec{a}, \vec{x})$$

where $\vec{a}$ is a parameter

Quantifier elimination

- Converts a formula into an equivalent one with no quantified variables

Observation: we can obtain **witness terms** for the eliminated variables

- Witness terms become the instructions of the synthesized program

- Prominent application of Q.E.:
  integer linear arithmetic

Q.E. for integer linear arithmetic

- Problem of great interest:
  - [Presburger, 1929], [Cooper, 1972]
  - [Pugh, 1992],
  - [Weispfenning, 1997]
  - Nipkow: elegant and verified, runs within Isabelle
- Our algorithm for integers:
  - Works on disjunctive normal form
  - Handling of inequalities as in [Pugh 1992]
  - Efficient in handling equalities (solves integer systems)
  - Computes **witness terms** , builds a program from them

choose((x, y) ⇒ 5 * x + 7 * y == a && x ≤ y)

Corresponding quantifier elimination problem:

$\exists x \exists y . 5x + 7y = a \wedge x \leq y$

Use extended Euclid's algorithm to find particular solution to $5x + 7y = a$:

$x = 3a$
$y = -2a$

(5,7 are mutually prime, else we get divisibility pre.)

Express general solution of *equations* for x, y using a new variable z:

$x = -7z + 3a$
$y = 5z - 2a$

Rewrite *inequations* x ≤ y in terms of z:

$5a \leq 12z$

$\rightarrow$ $z \geq \mathrm{ceil}(5a/12)$

Obtain synthesized program:

```
val z = ceil(5*a/12)
val x = -7*z + 3*a
val y = 5*z + -2*a
```

For a = 31:

$z = \mathrm{ceil}(5*31/12) = 13$
$x = -7*13 + 3*31 = 2$
$y = 5*13 – 2*31 = 3$

choose((x, y) ⇒ 5 * x + 7 * y == a && x ≤ y && x ≥ 0)

Express general solution of *equations* for x, y using a new variable z:

$x = -7z + 3a$

$y = 5z - 2a$

Rewrite *inequations* x ≤ y in terms of z:

$z \geq \mathrm{ceil}(5a/12)$

Rewrite x ≥ 0:

$z \leq \mathrm{floor}(3a/7)$

Precondition on a:

$\mathrm{ceil}(5a/12) \leq \mathrm{floor}(3a/7)$

(exact precondition)

Obtain synthesized program:

```
assert(ceil(5*a/12) ≤ floor(3*a/7))
val z = ceil(5*a/12)
val x = -7*z + 3*a
val y = 5*z + -2*a
```

With more inequalities we may generate a for loop

NP-Hard Constructs

- Disjunctions
  - Synthesis of a formula computes program and exact precondition of when output exists
  - Given disjunctive normal form, use preconditions to generate if-then-else expressions (try one by one)
- Divisibility combined with inequalities:
  - corresponding to big disjunction in q.e. , we will generate a for loop with constant bounds (could be expanded if we wish)

General Form of Synthesized Functions for Presburger Arithmetic

choose x **such that** F(x,a) → x = t(a)

Result t(a) is expressed in terms of

+, -, C*, /C, if

Need arithmetic for solving equations

Need conditionals for

- disjunctions in input formula
- divisibility and inequalities (find a witness meeting bounds and divisibility by constants)

t(a) = if P_1(a) t_1(a) elseif … elseif P_n(a) t_n(a)
else error("No solution exists for input",a)

When do we have witness generating quantifier elimination?

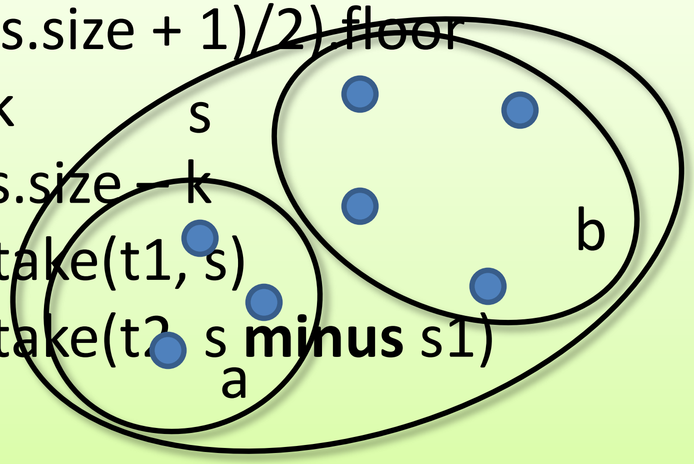
- Suppose we have
  - class of specification formulas **S**
  - decision procedure for formulas in class **D** that produces satisfying assignments
  - function (e.g. substitution) that, given concrete values of parameters *a* and formula F in **S**, computes F(x,a) that belongs to **D**
- Then we have synthesis procedure for **S** (proof: invoke decision procedure at run-time)

If have decidability → also have computable witness-generating QE in the language extended w/ computable functions

Synthesis for sets

```
def splitBalanced[T](s: Set[T]) : (Set[T], Set[T]) =
  choose((a: Set[T], b: Set[T]) ⇒ (
          a union b == s && a intersect b == empty
     && a.size – b.size ≤ 1
     && b.size – a.size ≤ 1
  ))
```

```
def splitBalanced[T](s: Set[T]) : (Set[T], Set[T]) =
  val k = ((s.size + 1)/2).floor
  val t1 = k
  val t2 = s.size – k
  val s1 = take(t1, s)
  val s2 = take(t2, s minus s1)
  (s1, s2)
```



Synthesis for non-linear arithmetic

```
def decomposeOffset(offset: Int, dimension: Int) : (Int, Int) =
  choose((x: Int, y: Int) ⇒ (
    offset == x + dimension * y && 0 ≤ x && x < dimension
  ))
```

- The predicate becomes linear *at run-time*
- Synthesized program must do case analysis on the sign of the input variables
- Some coefficients are computed at run-time

Compile-time warnings

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
  choose((h: Int, m: Int, s: Int) ⇒ (
        h * 3600 + m * 60 + s == totalSeconds
     && h ≥ 0 && h < 24
     && m ≥ 0 && m < 60
     && s ≥ 0 && s < 60
  ))
```

```
Warning: Synthesis predicate is not
satisfiable for variable assignment:
  totalSeconds = 86400
```

Compile-time warnings

```
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
  ))
```

```
Warning: Synthesis predicate has multiple
solutions for variable assignment:
  totalSeconds = 60
Solution 1: h = 0, m = 0, s = 60
Solution 2: h = 0, m = 1, s = 0
```

Arithmetic pattern matching

```
def fastExponentiation(base: Int, power: Int) : Int = {
  def fp(m: Int, b: Int, i: Int): Int = i match {
    case 0 ⇒ m
    case 2 * j ⇒ fp(m, b*b, j)
    case 2 * j + 1 ⇒ fp(m*b, b*b, j)
  }
  fp(1, base, p)
}
```

- Goes beyond Haskell's *(n+k)* patterns
- Compiler checks that all patterns are reachable and whether the matching is exhaustive

Experience with Comfusy ☺

- Works well for examples we encountered
  - Needed: synthesis for more expressive logics, to handle more examples
  - seems ideal for domain-specific languages
- Efficient for conjunctions of equations (could be made polynomial)
- **Extends to synthesis with parametric coefficients**
- Extends to logics that reduce to Presburger arithmetic (implemented for BAPA)

Comfusy for Arithmetic ☹

- Limitations of Comfusy for arithmetic:
  - Naïve handling of disjunctions
  - Blowup in elimination, divisibility constraints
  - *Complexity of running synthesized code (from QE): doubly exponential in formula size*
  - Not tested on modular arithmetic, or on synthesis with optimization objectives
  - Arbitrary-precision arithmetic with multiple operations generates time-inefficient code
  - Cannot do bitwise operations (not in PA)

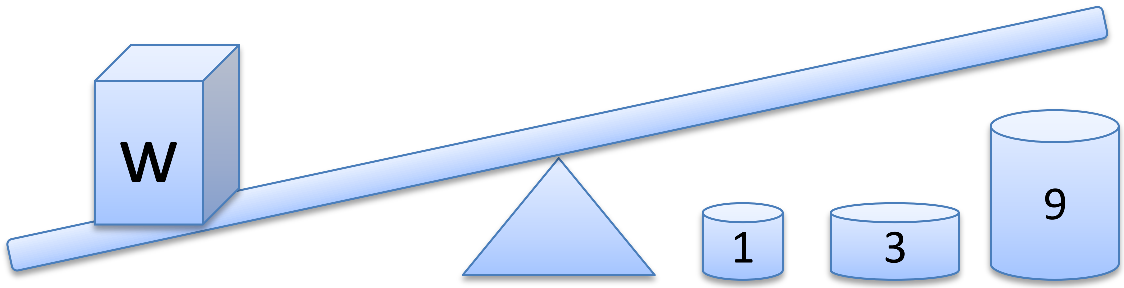
RegSy

Synthesis for regular specifications over unbounded domains

J. Hamza, B. Jobstmann, V. Kuncak

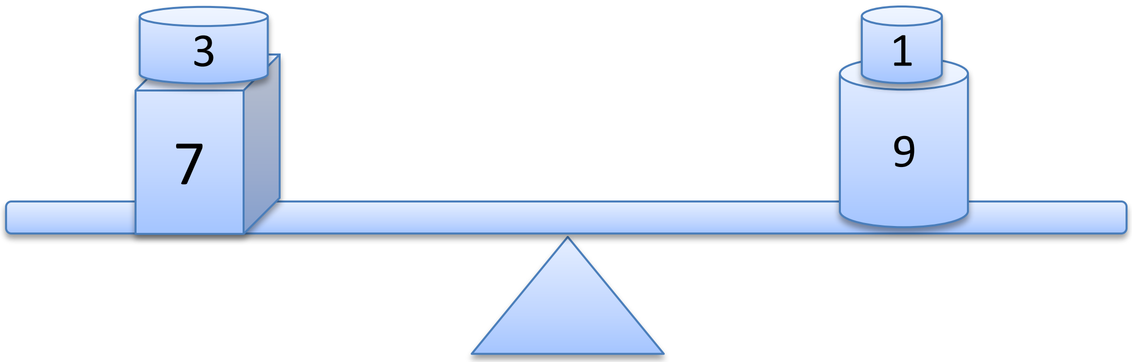
FMCAD 2010

Synthesize Functions over Integers



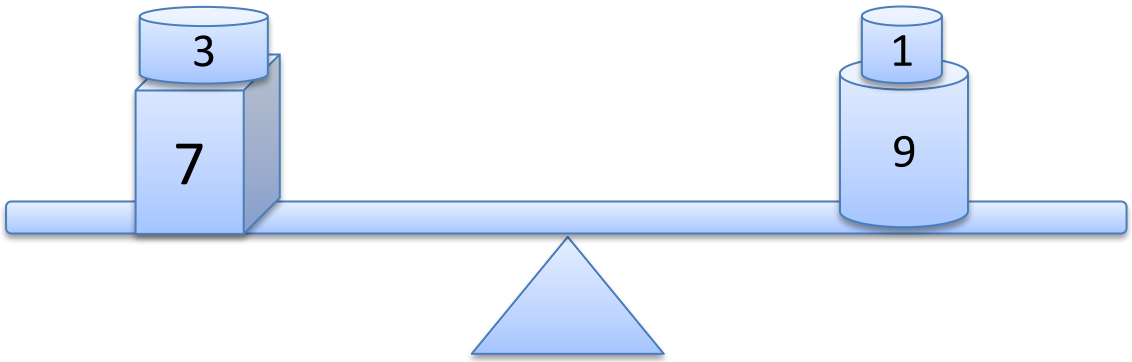
- Given weight w, balance beam using weights 1kg, 3kg, and 9kg
- Where to put weights if w=7kg?

Synthesize Functions over Integers



- Given weight w, balance beam using weights 1kg, 3kg, and 9kg
- Where to put weights if w=7kg?
- Program that computes correct positions of 1kg, 3kg, and 9kg for any w (if possible)?

Synthesize Functions over Integers



Synthesize function that, given weight w,

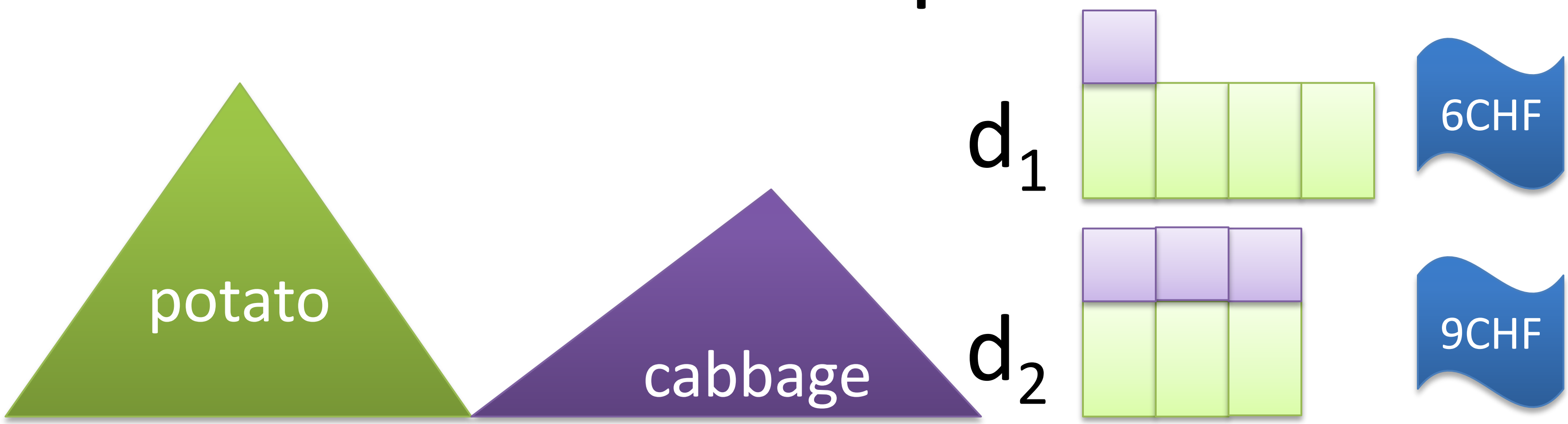
computes values for $l_1, l_3, l_9, r_1, r_3, r_9$ such that

$$w + l_1 + 3l_3 + 9l_9 = r_1 + 3r_3 + 9r_9$$

$$l_1 + r_1 \leq 1,\ l_3 + r_3 \leq 1,\ l_9 + r_9 \leq 1$$

Assumption: Integers are non-negative

Parameterized Optimization



Parametric linear constraints over integers p, c, d_1, d_2, e.g.,

$$R(p, c, d_1, d_2) := (4d_1 + 3d_2 \leq p) \wedge (d_1 + 3d_2 \leq c)$$

Synthesize function $(d_1, d_2) := f(p, c)$ that

(i) satisfies constraints, i.e., $R(p, c, f(p, c))$

(ii) maximizes profit $(d_1, d_2) := 6d_1 + 9d_2$

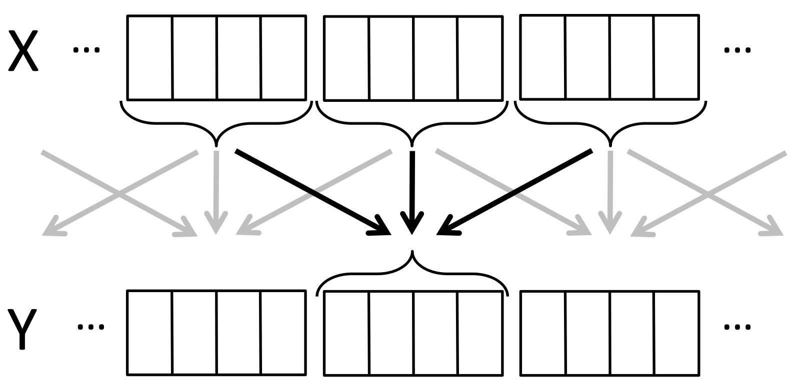
Note: integers have unbounded number of bits

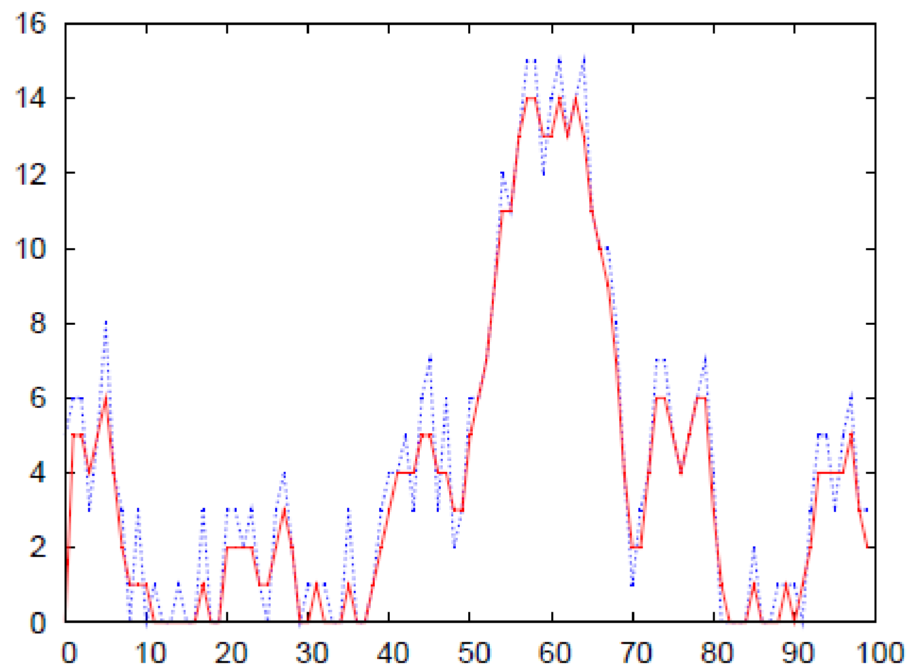
Synthesize Functions over bit-Streams

Smoothing a function:

- Given sequence X of 4-bit numbers
- Compute its average sequence Y

$$4Y[n..n+3] = X[n-4...n-1] + 2X[n..n+3] + X[n+4..n+7]$$





Expressiveness of Spec Language

- Non-negative integer constants and variables
- Boolean operators ($\wedge, \vee, \neg$)
- Linear arithmetic operator ($+$, $c \cdot x$)
- Bitwise operators (|, &, !)
- Quantifiers over numbers and bit positions

PAbit = Presburger arithmetic with bitwise operators
WS1S= weak monadic second-order logic of one successor

Problem

Given

- relation R over bit-stream (integer) variables in WS1S (PAbit)
- partition of variables into inputs and outputs

Constructs program that, given inputs, computes correct output values, whenever they exist.

Basic Idea of Regular Synthesis

- View integers as finite (unbounded) bit-streams (binary representation starting with LSB)
- Specification in WS1S (PAbit)
- Synthesis approach:
  - Step 1: Compile specification to automaton over combined input/output alphabet (automaton specifying relation)
  - Step 2: Use automaton to generate efficient function from inputs to outputs realizing relation

Example: Parity of Input Bits

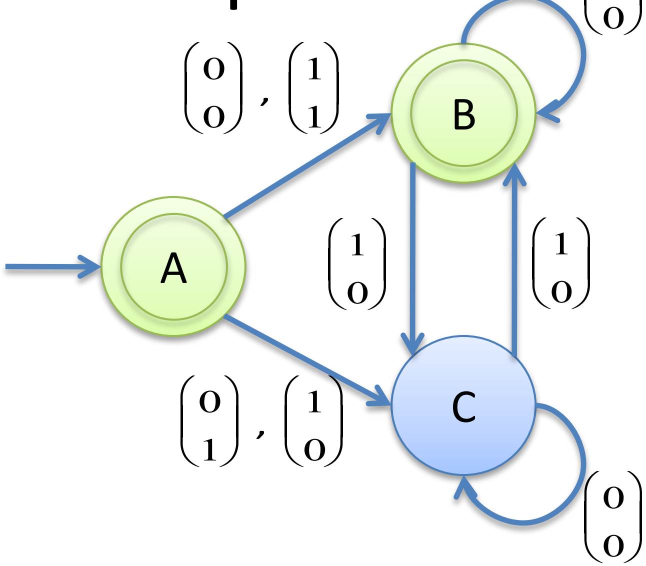
- Input x and output y are bit-streams
- Spec: output variable y indicates parity of non-zero bits in input variable x
  - y=00^* if number of 1-bits in x is even, otherwise y=10^*
  - Examples:

$$\begin{array}{llll} x: & 0 & 1 & 1 \\ y: & 0 & 0 & 0 \end{array}$$

$$\begin{array}{lllll} x: & 1 & 0 & 1 & 1 \\ y: & 1 & 0 & 0 & 0 \end{array}$$

Example: Parity of Input Bits

- Step 1: construct automaton for spec over joint alphabet



Spec: y=00* if number of 1-bits in x is even, otherwise y=10*

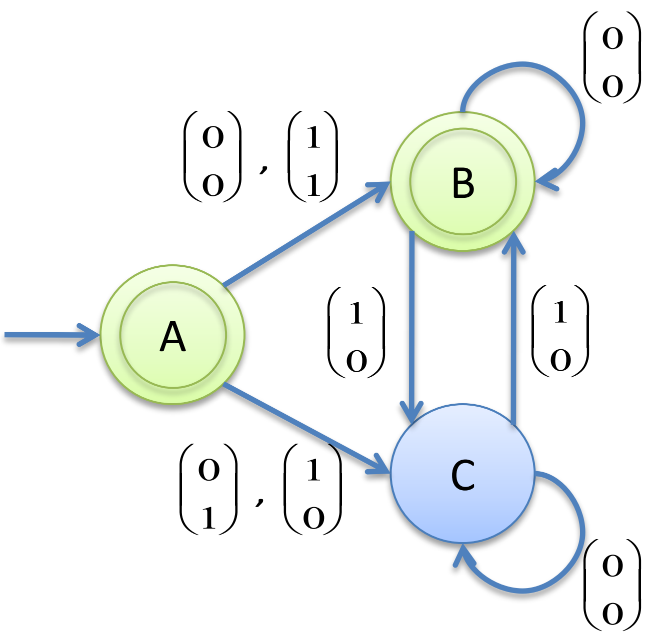
Accepting states are green

$$\begin{array}{cccc} x: & 0 & 1 & 1 \\ y: & 0 & 0 & 0 \end{array}$$

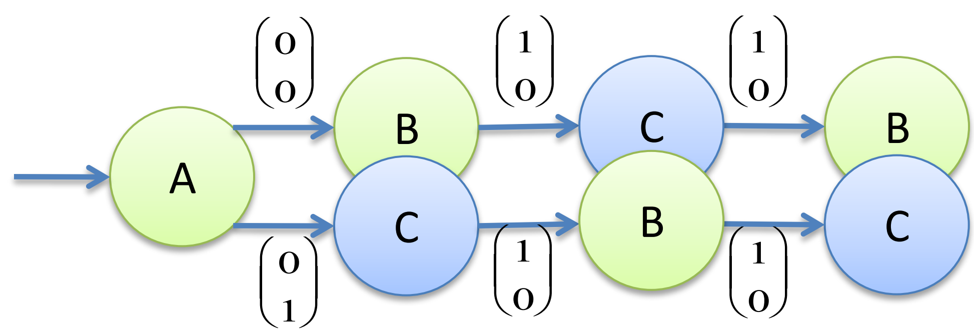
Note: must read entire input to know first output bit (non-causal)

Idea

- Run automaton on input, collect states for all possible outputs (subset construction)
- From accepting state compute backwards to initial state and output corresponding value



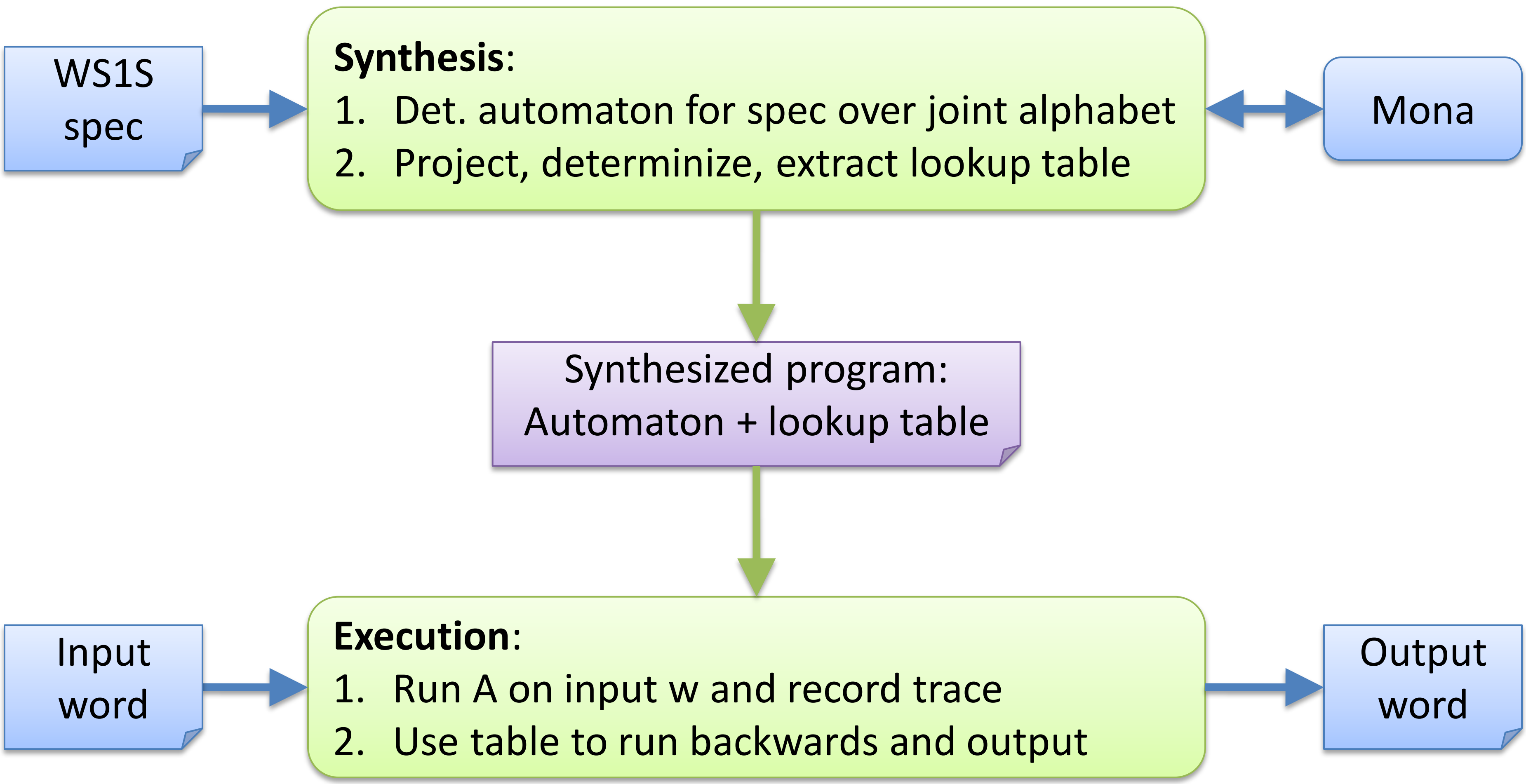
$$\begin{array}{cccc} x: & 0 & 1 & 1 \\ y: & ? & ? & ? \end{array}$$



Automaton has all needed information but subset construction for every input

Our Approach: Precompute

without losing backward information



Time and Space

- Automata may be large, but if table lookup is constant time, then forth-back run is linear in input (in number of bits)
- Also linear space
  - can trade space for extra time by doing logarithmic check-pointing of state gives O(log(n)) space, O(n log(n)) time

Prototype and Experiments

- RegSy is implemented in Scala
- Uses MONA to construct automaton on joint alphabet from specification
- Build input-deterministic automaton and lookup table using a set of automata constructions
- Run on several bit-stream and parametric linear programming examples

Experiments

No	Example	MONA (ms)	Syn (ms)	|A|	|A'|	512b	1024b	2048b	4096b
1	addition	318	132	4	9	509	995	1967	3978
2	approx	719	670	27	35	470	932	1821	3641
3	company	8291	1306	58	177	608	1312	2391	4930
4	parity	346	108	4	5	336	670	1310	2572
5	mod-6	341	242	23	27	460	917	1765	3567
6	3-weights-min	26963	640	22	13	438	875	1688	3391
7	4-weights	2707	1537	55	19	458	903	1781	3605
8	smooth-4b	51578	1950	1781	955	637	1271	2505	4942
9	smooth-f-2b	569	331	73	67	531	989	1990	3905
10	smooth-b-2b	569	1241	73	342	169	347	628	1304
11	6-3n+1	834	1007	233	79	556	953	1882	4022

In 3 seconds solve constraint, minimizing the output; inputs and outputs are of order 2^{4000}

Summary of RegSy

- Synthesize function over bit-stream (integer) variables
- Specification: WS1S or
  PA with bit-wise operators (including quantifiers)
- Linear complexity of running synthesized code (linear in number of input bits)
- Synthesize specialized solvers to e.g. disjunctive parametric linear programming problems
- Recent work:
  replace MONA with different construction

Alonzo Church

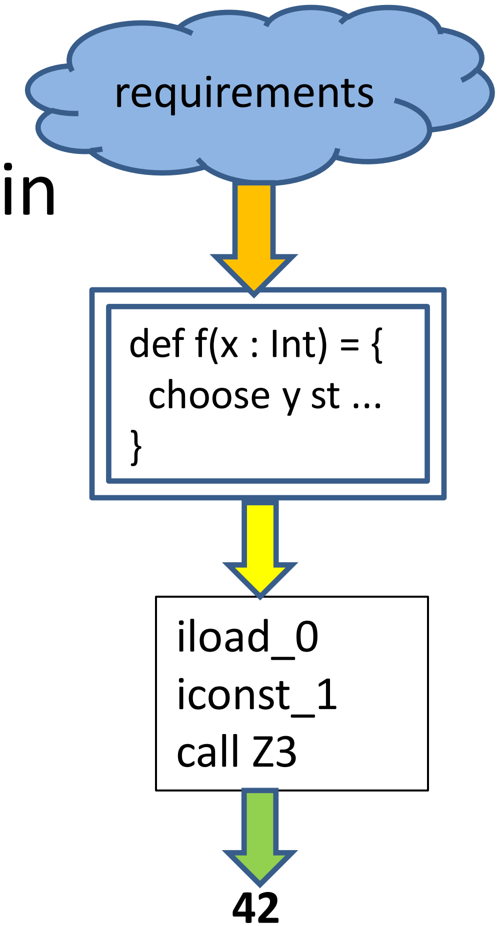
- Lambda calculus (1936)
  Foundation of modern functional programming languages

- Church synthesis problem (1957)
  Synthesis as foundation of future programming languages/systems

Implicit Programming at All Levels

Opportunities for implicit programming in

- **Development** within an IDE
  - **isynth** tool
- √ **Compilation**
  - **Comfusy** and **RegSy** tools
- √ **Execution**
  - **Scala^Z3** and **UDITA** tools



I next examine these tools, from last to first, focusing on Compilation

Implicit Constructs in IDEs:
code completion using automated reasoning

joint work with: **Tihomir Gvero**, **Ruzica Piskac**

isynth - Interactive Synthesis of Code Snippets

```
def map[A,B](f:A => B, l:List[A]): List[B] = { ... }
def stringConcat(lst : List[String]): String = { ... }
...
def printInts(intList:List[Int], prn: Int => String): String = □
```

Returned value:
`stringConcat(map[Int, String](prn, intList))`

Is there a term of given type in given environment?

Monorphic: decidable. Polymorphic: undecidable

Solution: use first-order resolution

isynth tool:

- based on first-order resolution – combines forward and backward reasoning
- supports method combinations, type polymorphism, user preferences
- ranking of multiple returned solutions
  - using a system of weights
  - preserving completeness
- further enhancements under way

Conclusion: Implicit Programming

Development within an IDE

- **isynth** tool – FOL resolution as code completion

Compilation

- **Comfusy** : decision procedure → synthesis procedure Scala implementation for integer arithmetic, BAPA
- **RegSy** : solving WS1S constraints

Execution

- **Scala^Z3** : constraint programming
- **UDITA**: Java + choice as test generation language

http://lara.epfl.ch/w/impro