Intermediate Languages from Birth to Execution

Viktor Kuncak
EPF Lausanne
http://lara.epfl.ch/w/impro
Birth of an Intermediate Language

Two difficult tasks:

- modeling *(obscure)* language features
- efficiently answering verification questions

Solve them modularly

- split a hard problem into two *(harder)* ones

Social aspects:

- focus efforts on one of the two sides of the problem
- can join even if you are ignorant about the other side
Rich Models

http://richmodels.org

Rich Model Toolkit

- translators from programming languages
- processing the intermediate code
  - VCG verifiers
  - model checking
  - abstract interpretation
  - synthesis
- provers that enable this
Next meeting: Turin, IT, October 3-4 then ETAPS’12, CADE’12

ICT COST Action IC0901

Rich-Model Toolkit - An Infrastructure for Reliable Computer Systems

Descriptions are provided by the Actions directly via e-COST.

The main objective is making automated reasoning techniques and tools applicable to a wider range of problems, as well as making them easier to use by researchers, software developers, hardware designers, and information system users and developers. The Action coordinates activities on developing infrastructures for automated reasoning about the new notion of Rich Models of computer systems. Rich Models have the expressive power of a large fragment of formalizable mathematics, enabling specification of software, hardware, embedded, and distributed systems. Rich Models support modeling at a wide range of abstraction levels, from knowledge bases and system architecture, to software source code and detailed hardware design. The Action contributes to the construction of Rich-Model Toolkit, a new unified infrastructure that precisely defines the meaning of Rich Models, introduces standardized representation formats, and incorporates a number of automated reasoning tools. Moreover, the Action develops and deploys new tools for automated reasoning that communicate using these standardized formats. The resulting tools will have a wide range of applicability and improved efficiency, helping system developers construct reliable systems through automated reasoning, analysis, and synthesis.
20 countries, 45+ researchers
Rich Model Toolkit:
Languages, Tools, and Competitions

Some of results and ongoing work

• Integration of SMT solvers and first-order provers as well as counterexample facilities in Isabelle (Blanchette, Böhme, Paulson)
• Hardware competition w/ standardized format (A.Biere)
• A format for transition systems (R.Iosif)
  – multiple paths to map C programs into it
  – infrastructure in many implementation languages
• A new SMT solver, OpenSMT (R. Bruttomesso, N.Sharygina,...)
• New techniques for analyzing linked structures in gcc compiler (T.Vojnar)
• Advances in the analysis of multi-threaded programs (A.Rybalchenko)
• Extensions of SMT-LIB 2 with collections (Kröning, Ruemmer)
• SMT language for transition systems (A.Cimatti, K.)
• SMT-like representation of a useful core of Isabelle theories (Nipkow, K.)
Benchmarks in Intermediate Language

- Programs as mathematical objects, given by their mathematical semantics
- One real-world program can have multiple mathematical models
  - ignore aspects of semantics (buffer or arith. overflows)
  - different representations of memory
  - different exceptional behavior
- All above choices made explicit in the model
- Model is as clear as a formula in Coq / Isabelle
  - as suitable for competition as SMT-LIB or TPTP
a suggestion to rename:

Intermediate Languages

arrow pointing down to

Central Languages

central for

- rigorous scientific study
- building systems
Imagine. A program diagnostic technique
– precisely treats arithmetic, aliasing, concurrency, exceptions, environment
– can process 100’000’000 lines a second
It exists. It is called testing. Most widely used
– works on programs
– translate program into intermediate language; does testing still work?
Key challenge: non-deterministic constructs
```plaintext
if (c) s1 else s2 \rightarrow (assume(c); s1) [] (assume(!c); s2)
```
havoc to e.g. approximate procedure executions
Also key feature: havoc+assume – specification statement
Executing Non-Deterministic Constructs

Ruzica Piskac  Philippe Suter  Tihomir Gvero  Mikaël Mayer  Ali Sinan Köksal  Sebastien Vasey

EPFL

Barbara Jobstmann  Jad Hamza
Verimag  ENS Cachan

Darko Marinov  Milos Gligoric  Vilas Jagannath
UIUC

Sarfraz Khurshid  
UT Austin
Several Tools Available

• UDITA extension of Java Pathfinder (w/ M.Gligoric,T. Gvero,V.Jagannath,D. Marinov,S.Khurshid)
  – make non-deterministic choice variables symbolic

• Java^Z3 (w/ P.Suter, R.Steiger)
  – invoke Z3 from Scala and vice versa with nice syntax

• Kaplan (w/ A.S.Köksal, P.Suter)
  – constraint solving for constraints over Z3+executable functions
  – solution enumeration through `for’ comprehensions
  – logical variables with global constraint store

• Comfusy (w/ M.Mayer,R.Piskac,P.Suter)
  – compile specifications using quantifier elimination

• RegSy (w/ J. Hamza, B. Jobstmann)
  – compile specifications into very fast automata
Idea of UDITA

• Executing programs with bounded non-determinism greatly benefits even from simple symbolic execution – lazy evaluation

• If we can execute non-deterministic programs, we can do modular testing
  – write precondition as non-deterministic programs
  – UDITA automatically generates tests that satisfy e.g. the precondition

• Programmer has control which properties established by construction, which to solve for
UDITA: Non-deterministic Language

```java
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];
        }
    }
}
```

We used to it to generate tests and find real bugs in javac, JPF itself, Eclipse, NetBeans refactoring.

On top of Java Pathfinder’s backtracking mechanism can enumerate all executions.

Key technique: suspended execution of non-determinism with: M. Gligoric, T. Gvero, V. Jagannath, D. Marinov, S. Khurshid

Java + choose
- integers
- (fresh) objects
Implemented and released in official Java PathFinder

JPF .. the swiss army knife of Java™ verification

jenkins

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:

```java
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() {...}
}
```
Non-determinism is very useful in programming
Implicit Programming

• A high-level programming model
• In addition to traditional constructs, use **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 a key technique
Explicit Design

Explicit = written down, machine readable
Implicit = omitted, to be (re)discovered

• Current practice:
  – explicit program code
  – implicit design (key invariants, properties)

• Goal:
  – explicit design
  – implicit program

Total work not increased, moreover
  – can be decreased for certain types of specifications
  – confidence in correctness is increased
Example: Date Conversion

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

```c
BOOL ConvertDays(UINT32 days, SYSTEMTIME* lpTime)
{
    ...; 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 a major brand freeze
The `choose` Implicit Construct

```scala
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 → 1 hour, 3 mins. and 7 secs.
```
How to execute **choose** by invoking SMT solver at run time

```python
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))
```

3787 seconds → exec (Z3 “h * 3600 + m * 60 + s == 3787 && ...”)

```
sat
model: h=1, m=3, s=7
```

This approach works for constraints in theories for which SMT solver is **complete** and provides **model generation**.
Scala\(^{\text{Z3}}\)
Invoking Constraint Solver at Run-Time

Java Virtual Machine
- functional and imperative code
- custom ‘decision procedure’ plugins

Z3 SMT Solver

Q: implicit constraint
A: model
Q: queries containing extension symbols
A: custom theory consequences

with: Philippe Suter, Ali Sinan Köksal (this Wednesday)
Programming in Scala^Z3: Enumeration

Find triples of integers $x$, $y$, $z$ such that $x > 0$, $y > x$, $2x + 3y \leq 40$, $x \cdot z = 3y^2$, and $y$ is prime.

```scala
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)
```

Model enumeration (currently: negate previous)

User's Scala function

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

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
Recursive Functions

Limitation of Current SMT Provers:
– no built-in support for recursive functions
– encoding using quantifiers is brittle

P. Suter, M. Dotta, V. Kuncak, POPL’10:
*If a function with a simple recursive schema maps many elements to one (is sufficiently surjective), then adding it preserves decidability.*
Leon verification system

- Integrates into the DPLL(\(T\)) loop of Z3 and performs demand-driven unfolding of function definitions
  - treat recursive functions as un-interpreted after some depth
  - if we block recursive branches and get SAT, then SAT
  - if we unblock and get UNSAT, the problem is UNSAT
  - else keep unfolding
- Checks the faithfulness of candidate examples using code execution
  - Is complete for counterexamples, often [SDK10] also for proofs

Leon generates verification conditions to prove that:
- Function contracts are valid
- Preconditions are met at all function invocations
- Pattern matching expressions are exhaustive
Leon as a Verifier

Verification of data structure implementations and syntax tree manipulations:

Benchmark ($LOC$, $time$ in seconds):

- List (107, 0.78s): tail-recursive size, associativity of append, distributivity of content over append
- Associative list (50 LOC, 0.24s): content of union, read-over-write
- Insertion sort (99 LOC, 0.43s): sortedness, size and content of output
- Red-black tree (117 LOC, 3.75s): set interface, black-balancedness, “red nodes have black children”
- Propositional logic (86 LOC, 2.37s): idempotence of NNF and simplification
- Sum and max (46 LOC, 0.29s): sum less than size times maximum
- Search linked list (48 LOC, 0.17s): position of first zero
- Amortized queue (124 LOC, 3.4s): abstract queue interface, list size invariant
→ use Leon as constraint solver for executable predicates

Pure recursive functions: very useful for specification and intermediate languages

KAPLAN
Scala language: functions, objects, traits, mutation, exceptions, actor concurrency

constraints: executable predicates in a purely functional sublanguage

general computations **create** first-class constraints
- as Boolean-valued Scala expressions with pure functions
- manipulated as well-typed trees (computed at run-time)

**solving** first-class constraints **controls** general computation
- constraint solver creates stream of solutions to constraint
- a **for** loop construct iterates over solutions and supplied them to general computation
- logical variables and global constraint store enable optimization across multiple **for** loops
Enumerating Trees - how to do this, using an SMT solver?

```scala
sealed abstract class Tree
case class Leaf() extends Tree
case class Node(left : Tree, data : Int, right : Tree) extends Tree
def isSorted(t : Tree) : Boolean = { ... }
def content(t : Tree) : Set[Int] = t match {
  case Leaf() ⇒ Set()
  case Node(l, d, r) ⇒ content(l) ++ Set(d) ++ content(r) }
def printTreesContaining(s : Set[Int]) = {
  for (t ← ((t : Tree) ⇒ isSorted(t) && content(t)==s).findAll)
    println(t) // replace with e.g. testUnitWithInput(t)
}
scala> printTreesContaining(Set(5,2,9))
Node(Node(Node(Leaf(),2,Leaf()),5,Leaf()),9,Leaf())
Node(Node(Leaf(),2,Node(Leaf(),5,Leaf())),9,Leaf())
Node(Node(Leaf(),2,Leaf()),5,Node(Leaf(),9,Leaf()))
Node(Leaf(),2,Node(Node(Leaf(),5,Leaf())),9,Leaf()))
Node(Leaf(),2,Node(Leaf(),5,Node(Leaf(),9,Leaf())))
Node(Leaf(),2,Node(Leaf(),5,Node(Leaf(),9,Leaf())))
```
Using recursive functions to specify the desired value

```python
def append(l1 : List, l2 : List) : List = l1 match {
  case Nil() => l2
  case Cons(x, xs) => Cons(x, append(xs, l2))
}
def snoc(lst : List) : (List, Int) =
  ((res, e) : (List, Int)) => lst == append(res, Cons(e, Nil())).solve

def addDeclarative(x : Int, tree : Tree) : Tree =
  ((t : Tree) => isRedBlackTree(t) &&
   content(t) == content(tree) ++ Set(x)).solve

def removeDeclarative(x : Int, tree : Tree) : Tree =
  ((t : Tree) => isRedBlackTree(t) &&
   content(t) == content(tree) -- Set(x)).solve
```
SAT Solver - Creating Constraints in Scala

Solving a CNF SAT instance in the standard DIMACS format

```scala
val p1 = Seq(Seq(1,-2,-3), Seq(2,3,4), Seq(-1,-4))

(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_4)
```

```scala
def fromDimacs(problem : Seq[Seq[Int]]): Constraint1[Map[Int,Boolean]] = 
    problem.map(clause => clause.map(literal => {
        val id = abs(literal)
        val isPos = literal > 0
        ((m : Map[Int,Boolean]) ⇒ m(id) == isPos).coun
        .reduceLeft(_ || _))

scala> fromDimacs(p1).solve
scala> Some(Map(2 → true, 3 → false, 1 → false, 4 → false))
scala> fromDimacs(Seq(Seq(1,2), Seq(-1), Seq(-2))).solve
scala> None
```
Creating and Solving Knapsack Problem Instances

```scala
def solveKnapsack(vals : List[Int], weights : List[Int], max : Int) = {
  def conditionalSumTerm(vs : List[Int]) = {
    vs.zipWithIndex.map(pair ⇒ {
      val (v,i) = pair
      ((m : Map[Int,Boolean]) ⇒ (if(m(i)) v else 0)).i
    }).reduceLeft(_ + _)
  }
  val valueTerm = conditionalSumTerm(vals)
  val weightTerm = conditionalSumTerm(weights)
  val answer = (((x : Int) ⇒ x ≤ max).compose0(weightTerm)
    .maximizing(valueTerm)
    .solve
  }
}
scala> val vals : List[Int] = List(4, 2, 2, 1, 10)
scala> val weights : List[Int] = List(12, 1, 2, 1, 4)
scala> val max : Int = 15
scala> solveKnapsack(vals, weights, max)
result:  Map(0 → false, 1 → true, 2 → true, 3 → true, 4 → true)
```
Several Tools Available

• **UDITA** extension of Java Pathfinder (w/ M.Gligoric, T. Gvero, V. Jagannath, D. Marinov, S. Khurshid)
  - make non-deterministic choice variables symbolic

• **Java^Z3** (w/ P. Suter, R. Steiger)
  - invoke Z3 from Scala and vice versa with nice syntax

• **Kaplan** (w/ A.S. Köksal, P. Suter)
  - constraint solving for constraints over Z3+executable functions
  - solution enumeration through `for` comprehensions
  - logical variables with global constraint store

• **Comfusy** (w/ M. Mayer, R. Piskac, P. Suter)
  - compile specifications using quantifier elimination

• **RegSy** (w/ J. Hamza, B. Jobstmann)
  - compile specifications into very fast automata
What would ideal code look like?

```python
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)

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)
```

Mikaël Mayer  Ruzica Piskac  Philippe Suter,  PLDI’10
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:

```plaintext
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
Possible starting point: quantifier elimination

- A specification statement of the form

\[ \vec{r} = \text{choose}(\vec{x} \Rightarrow F(\vec{a}, \vec{x})) \]

"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 \( a \) is a parameter

- Witness terms from QE are the generated program!
choose((x, y) ⇒ 5 * x + 7 * y == a && x ≤ y)

Corresponding quantifier elimination problem:

∃ x ∃ y . 5x + 7y = a ∧ x ≤ y

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

(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:

\[
5a ≤ 12z \Rightarrow z ≥ \text{ceil}(5a/12)
\]

Obtain synthesized program:

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

For a = 31:

\[
x = -7*13 + 3*31 = 2 \\
y = 5*13 - 2*31 = 3
\]
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:

\[
\begin{align*}
    x &= -7z + 3a \\
    y &= 5z - 2a
\end{align*}
\]

Rewrite inequations \(x ≤ y\) in terms of z:

\[z ≥ \text{ceil}(5a/12)\]

Rewrite \(x ≥ 0\):

\[z ≤ \text{floor}(3a/7)\]

Precondition on a:

\[\text{ceil}(5a/12) ≤ \text{floor}(3a/7)\]  
(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 and divisibility: generate ‘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)
Methodology QE $\rightarrow$ Synthesis

• Each quantifier elimination ‘trick’ we found corresponds to a synthesis trick
• Find the corresponding terms
• Key techniques:
  – one point rule immediately gives a term
  – change variables, using a computable function
  – strengthen formula while preserving realizability
  – recursively eliminate variables one-by one
• Example use
  – transform formula into disjunction
  – strengthen each disjunct using equality
  – apply one-point rule
Compile-time warnings

```python
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
```
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
Synthesis for sets

```python
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
    ))
```

```python
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
“Extreme Compilation”

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 $w=7$kg?
Synthesize Functions over Integers

• Given weight w, balance beam using weights 1kg, 3kg, and 9kg
• Where to put weights if w=7kg?
• Synthesize program that computes correct positions of 1kg, 3kg, and 9kg for any w?
Synthesize function that, given weight \( w \), computes (minimal) values for \( l_1, l_3, l_9, r_1, r_3, r_9 \) such that

\[
    w + l_1 + 3l_3 + 9l_9 = r_1 + 3r_3 + 9r_9
\]

\[
    l_1 + r_1 \leq 1, \quad l_3 + r_3 \leq 1, \quad l_9 + r_9 \leq 1
\]

Assumption: Integers are non-negative
Expressiveness of Spec Language

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

\[ \text{PAbit} = \text{Presburger arithmetic with bitwise operators} \]
\[ \text{WS1S} = \text{weak monadic second-order logic of one successor} \]
Problem Formulation

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

• 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
3. Find $x$.

$1_2 + x = 11000_2$

unbounded bitwidth serial adder:

\[ \begin{align*}
\ldots & 0 0 0 0 0 0 1 \\
+ & \ldots 0 1 0 1 1 1 \\
\ldots & 0 1 1 0 0 0
\end{align*} \]

can check if $x$ was correct

to Find $x$
1) compute tree
2) go back from accepting leaf
Inefficient but Correct Method

• Run a deterministic automaton over inputs and outputs
• Becomes run of a non-deterministic automaton, if we look only at outputs
• Idea:
  – Simulate all branches until the end of input
  – Successful branches indicate outputs that work
Our Approach: Precompute
without losing backward information

**Synthesis:**
1. Det. automaton for spec over joint alphabet
2. Project, determinize, extract lookup table

**Synthesized program:**
Automaton + lookup table

**Execution:**
1. Run A on input w and record trace
2. Use table to run backwards and output
## 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| 
| 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 $2^{4000}$
Birth and Execution of Intermediate Languages

Rich Model Toolkit initiative: http://richmodels.org
  – one of the goals: standardize languages for programs and formulas

Execution of non-deterministic constructs
  – UDITA: Java + choice as a test generation language
  – Scala^Z3: tool demo at this CADE
  – Kaplan: recursive functions in specifications, logical vars

Compilation
  – Comfusy: decision procedure → synthesis procedure
    Scala implementation for integer arithmetic, BAPA
  – RegSy: compilation of WS1S constraints using automata

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