Towards Implicit Programming

Viktor Kuncak

EPF Lausanne

http://lara.epfl.ch/w/impro
Joint work with

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
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 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
The **choose** Implicit Construct

```python
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.

Are such examples realistic?

• 1/2010: all music players of a major brand froze on New Year’s day
• reason: **a subtly incorrect loop, trying to do a version of this**, for date and leap years (we can handle this example)
• our approach ensures both termination and correctness
Relationship to Verification

- **Some** functionality is best *synthesized* from specs
  - **other**: better *implemented, verified* (and put into library)
- Currently, no choice – must always implement
  - specifications and verification are viewed as **overhead**
- Goal: make specifications intrinsic part of programs, with clear benefits to programmers – **they run!**
- Example: write an assertion, not how to establish it
- This will help both
  - verifiability: document, not reverse-engineer the invariants
  - productivity: avoid writing messy implementation
Implicit Programming at All Levels

Opportunities for implicit programming in

• **Development** within an IDE
  – *InSynth* tool (CAV’11)

• **Compilation**
  – *Comfusy* (PLDI’10) and *RegSy* (FMCAD’10)

• **Execution**
  – *Scala^Z3* (CADE’11) and *UDITA* (ICSE’10)

I next examine these tools, from last to first, focusing on Compilation
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\(^Z3\)

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
Programming in Scala^Z3: Enumeration

find triples of integers x, y, z such that x > 0, y > x, 2x + 3y <= 40, x · z = 3y², and y is prime

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
Recursive functions within constraints

• If a function with a simple recursive schema maps many elements to one, then adding it preserves decidability! (w/ P. Suter, POPL’10)
  \[ \text{content(balance(tree))} = \text{content(tree)} \]

• A verifier for functional code, complete for counterexamples (and often finds proofs)

• We can also use it to solve constraints with user-defined functions
  – ongoing work to make it more efficient (P.Suter, A.S.Koeksal)
  – observation: SMT solvers can be brittle
  – counterexample generation made originally for diagnostics
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
Implemented and released in official Java PathFinder

**JPF** .. the *swiss army knife* of Java™ verification

**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](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() { ... }
}
```
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
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    ))

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)
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:

```haskell
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

z ≥ 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

For a = 31:

z = ceil(5*31/12) = 13
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 inequalities \( x ≤ y \) 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:

\[ \text{assert}(\text{ceil}(5a/12) ≤ \text{floor}(3a/7)) \]
\[ \text{val } z = \text{ceil}(5a/12) \]
\[ \text{val } x = -7z + 3a \]
\[ \text{val } y = 5z - 2a \]

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
General Form of Synthesized Functions for Presburger Arithmetic

choose \( x \) such that \( F(x,a) \rightarrow x = t(a) \)

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)

\( t(a) = \text{if } P_1(a) \text{ t}_1(a) \text{ elseif } ... \text{ elseif } P_n(a) \text{ t}_n(a) \text{ else error(“No solution exists for input”),a) } \)
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

```python
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
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

```python
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
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): \(\text{doubly exponential in formula size}\)
  – 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 $w=7$kg?
Synthesize Functions over Integers

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

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

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

Assumption: Integers are non-negative
Expressiveness of Spec Language

• Non-negative integer constants and variables
• Boolean operators ($\land, \lor, \lnot$)
• 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 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.

Unlike Church synthesis problem, we do not require causality (but if spec has it, it helps us)
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
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 | 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 $2^{4000}$
More Compact Encoding

- Instead of using MONA, use circuits
- Use PAbit as specification language
  - obtain elementary complexity for synthesis
  - same definable constraints
- Technical report at EPFL
More on Compilation

• Essence: partial evaluation

• Recent work: partially evaluate Simplex
  – repeatedly solve same formula
    (with different parameters)
  – obtaining orders of magnitude speedups

• Work by Sebastien Vasey
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.
**isynth** - Interactive Synthesis of Code Snippets

```scala
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?
Monomorphic: decidable. Polymorphic: undecidable

joint work with: **Tihomir Gvero, Ruzica Piskac**
Code Synthesis inside IDE

- Find all visible symbols
- Create clauses:
  - encode in FOL
  - assign weights
- Resolution algorithm with weights
- Code snippets
Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th># Loaded Declarations</th>
<th># Methods in Synthesized Snippets</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileReader</td>
<td>6</td>
<td>4</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>Map</td>
<td>4</td>
<td>4</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>FileManager</td>
<td>3</td>
<td>3</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>Calendar</td>
<td>7</td>
<td>3</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>FileWriter</td>
<td>320</td>
<td>6</td>
<td>0.093</td>
</tr>
<tr>
<td>SwingBorder</td>
<td>161</td>
<td>2</td>
<td>0.016</td>
</tr>
<tr>
<td>TcpService</td>
<td>89</td>
<td>2</td>
<td>&lt; 0.001</td>
</tr>
</tbody>
</table>

On 72 out of 83 code samples from web finds the expected code fragment
As the platform for our tools we use the Scala language developed at EPFL by Martin Odersky

http://www.scala-lang.org/

“Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive.”

→ one can write both ML-like and Java-like code
→ used in industry, academia, NASA
→ Scala non-profit, and a company http://typesafe.com
Notions Related to Implicit Programming

• Code completion
  – help programmer to interactively develop the program

• Synthesis for code fragments
  – arises as compilation for specification constructs

• Manual refinement from specs (Morgan, Back)

• Logic Programming
  – shares similar vision, in particular CLP(X)
  – operational semantics choice limits what specifications can be executed (e.g. bad recursion in Prolog, cut)
  – CLP solver theories limited (compared to SMT solvers)
  – not on mainstream platforms, not fully utilizing SAT
  – CLP compilation not using synthesis techniques
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

Future: linked structures and distributed systems

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