http://lara.epfl.ch
Lab for Automated Reasoning and Analysis
Viktor Kuncak

a past project: http://JavaVerification.org
new group of project: http://lara.epfl.ch/w/impro
graduate class every Spring: http://lara.epfl.ch/sav10
large international effort: http://RichModels.org
Automated Reasoning

• Applied to
  – Software analysis and verification
    (proving programs correct automatically)
  – Synthesis of software from specifications
    • The result is guaranteed to be correct wrt. spec
  – Advanced programming language constructs

• Techniques:
  – Decision procedures, program analysis, verification, theorem proving, programming languages
A Costly Bug
(from a BBC article)

Cryosat, a satellite worth 135m euro
October 2007
Verification - of explicit programs

Tools such as: ESC/Java, VCC, Jahob and, in combination with model checking: SLAM, BLAST, ...

- program satisfies the properties
- proofs
- models
- error in program (or property)

program fragment

program properties

Program Verifier
**Synthesis**

LTL Synthesis in tools such as: Anzu, Lizzy, RATSY. Related goals: high-level compilation/generation tools (e.g. Simulink).

In this talk: Comfusy & RegSy.
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
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 higher – following design
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;
        }
    }
} else {
    days -= 366;
    Enter December 31, 2008
    all music players of a major brand freeze
```
Date Conversion in Scala\(^2\)Z3 System

Let origin denote 1980. Choose year and day such that the property holds, where leadYearsUpto is defined by the given expression.

\[
\begin{align*}
\text{val origin} & \text{ = 1980} \\
\text{val (year, day)} & \text{=} \text{choose((year:Val[Int], day:Val[Int])} \\
& \text{=} \{ \\
& \text{def leapYearsUpto(y : Tree[IntSort])} \\
& \text{=} \text{(y - 1) / 4 - (y - 1) / 100 + (y - 1) / 400} \\
& \text{totalDays} \text{ == (year - origin) * 365} \\
& \text{+ leapYearsUpto(year) - leapYearsUpto(origin)} \\
& \text{+ day \&\&} \\
& \text{day > 0 \&\& day <= 366} \\
& \text{)}
\end{align*}
\]

The implicit programming approach ensures both
- successful termination (when in decidable logic) and
- correctness with respect to stated properties.
Let origin denote 1980. Choose year and day such that the property holds, where leadYearsUpto is defined by the given expression.

```scala
val origin = 1980
@spec
def leapYearsUpto(y : Int) = // could also be recursive
  (y - 1) / 4 - (y - 1) / 100 + (y - 1) / 400

val (year, day)=choose( (year:Int, day:Int) => {
  totalDays == (year - origin) * 365
  + leapYearsUpto(year)-leapYearsUpto(origin)
  + day &&
  day > 0 && day <= 366
})
```
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**
  - *UDITA* (ICSE’10), *Scala^Z3* (CADE’11), Kaplan

```scala
def f(x : Int) = {
  choose y st ...
}
```

```python
iload_0
iconst_1
call Z3
```

42
Overview of Our Tools & Prototypes

**InSynth** (w/ T.Gvero, R. Piskac)
- synthesize code fragments interactively within IDE

**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

**UDITA** extension of Java Pathfinder
- make non-deterministic choice variables symbolic

**Scala^Z3** (w/ P. Suter, A.S. Köksal, 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
Complete functional synthesis
V. Kuncak, M. Mayer, R. Piskac, P. Suter
PLDI 2010
also Communications of the ACM (soon)
Example

...ordinary code...

```
val (hours, minutes, seconds) = choose((h: Int, m: Int, s: Int) ⇒ (h * 3600 + m * 60 + s == totsec && 0 ≤ m && m ≤ 60 && 0 ≤ s && s ≤ 60))
```

formula in Presburger arithmetic

...ordinary code...

```
val (hours, minutes, seconds) = {
  val loc1 = totsec div 3600
  val num2 = totsec + ((−3600) * loc1)
  val loc2 = min(num2 div 60, 59)
  val loc3 = totsec + ((−3600) * loc1) + (−60 * loc2)
  (loc1, loc2, loc3)
}
```

witness-term-generating quantifier elimination

Standard, executable code

Implemented as extension of programming language Scala
How does it work in general?
Basis: Q.E. for integer linear arithmetic

• Well studied:
  – [Presburger, 1929]
  – [Cooper, 1972]
  – [Pugh, 1992]
  – [Weispfenning, 1997]

• Our algorithm for integers:
  – Works on disjunctive normal form
  – Efficient in handling equalities (possible in DNF)
  – Handling of inequalities similar as in [Pugh 1992]
  – Computation of **witness terms**
Decision Procedures

formula in decidable logic

\[
\neg \text{next0}^*(\text{root0}, n1) \land \\
x \notin \{\text{data0}(n) \mid \text{next0}^*(\text{root0}, n)\} \land \\
\text{next} = \text{next0}[n1 := \text{root0}] \land \\
\text{data} = \text{data0}[n1 := x] \rightarrow \\
|\{\text{data}(n) \mid \text{next}^*(n1, n)\}| = \\
|\{\text{data0}(n) \mid \text{next0}^*(\text{root0}, n)\}| + 1
\]

procedure that always terminates

Decision Procedure

formula is valid

formula has a counterexample
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:
x = 3a
y = -2a

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
ceil(5a/12) ≤ z

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 = ceil(5*31/12) = 13
x = -7*13 + 3*31 = 2
y = 5*13 - 2*31 = 3
choose\(((x, y)\Rightarrow 5\cdot x + 7\cdot y == a && x \leq y && x \geq 0)\)

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

\[
x = -7z + 3a \\
y = 5z - 2a
\]

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

ceil(\(5a/12\)) \(\leq z\)

Rewrite \(x \geq 0\):

\(z \leq floor(3a/7)\)

Precondition on a:

ceil(\(5a/12\)) \(\leq floor(3a/7)\) (exact precondition)

Obtain synthesized program:

\[
\text{assert}\(\text{ceil}(5\cdot a/12) \leq \text{floor}(3\cdot a/7)\)\\
\text{val}\ z = \text{ceil}(5\cdot a/12)\\
\text{val}\ x = -7\cdot z + 3\cdot a\\
\text{val}\ y = 5\cdot z - 2\cdot a
\]
Checks for *Pure* Presburger Arithmetic

```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
```

Such checks not possible with multiplication by parameters
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 = 0
Solution 1: h = 0, m = 0, s = 0
Solution 2: h = -1, m = 59, s = 60
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
))
## Synthesis Times (seconds)

<table>
<thead>
<tr>
<th></th>
<th>scalac</th>
<th>with plugin</th>
<th>with checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecondsToTime</td>
<td>3.05</td>
<td>3.2</td>
<td>3.25</td>
</tr>
<tr>
<td>FastExponentiation</td>
<td>3.1</td>
<td>3.15</td>
<td>3.25</td>
</tr>
<tr>
<td>ScaleWeights</td>
<td>3.1</td>
<td>3.4</td>
<td>3.5</td>
</tr>
<tr>
<td>PrimeHeuristic</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
</tr>
<tr>
<td>SetConstraints</td>
<td>3.3</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>SplitBalanced</td>
<td>3.3</td>
<td>3.9</td>
<td>4.0</td>
</tr>
<tr>
<td>Coordinates</td>
<td>3.2</td>
<td>4.2</td>
<td>--</td>
</tr>
<tr>
<td>All</td>
<td>5.75</td>
<td>6.35</td>
<td>6.75</td>
</tr>
</tbody>
</table>

Extra time required to synthesis code is modest
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 gives time-inefficient code
  - Cannot do bitwise operations (not in PA)
Synthesis for regular specifications over unbounded domains

J. Hamza, B. Jobstmann, V. Kuncak
FMCAD 2010
Example: Beam Balance

Fig. 5. Beam balance with three weights

- Synthesize function that, given \( w \), computes values for \( l_i \) and \( r_i \) such that  
  \[
  F(w,l_1,l_3,l_9,r_1,r_3,r_9)
  \]
  \[
  w + l_1 + 3l_3 + 9l_9 = r_1 + 3r_3 + 9r_9
  \]
  \[
  l_1 + r_1 \leq 1 \quad l_3 + r_3 \leq 1 \quad l_9 + r_9 \leq 1
  \]
- Can additionally require, for linear \( f \):  
  \[
  F(w,y) \land \forall y'. F(w,y') \Rightarrow f(y) \leq f(y')
  \]
Experiments using MONA

- Linear scaling observed in length of input 😊
- In 3 seconds solve constraint, minimizing the output; inputs and outputs are of order $2^{4000}$
- Multiplication by constants can be problematic

**Table I**

| No | Example      | MONA (ms) | Synthesis (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      | 8’291     | 1’306          | 58   | 177  | 608  | 1312  | 2391  | 4930  |
| 4  | parity       | 346       | 108            | 4    | 5    | 336  | 670   | 1310  | 2572  |
| 5  | mod-6-test   | 341       | 242            | 23   | 27   | 460  | 917   | 1765  | 3567  |
| 6  | 3-weights-min| 26’963    | 640            | 22   | 13   | 438  | 875   | 1688  |       |
| 7  | 4-weights    | 2’707     | 1’537          | 55   | 19   | 458  | 903   | 1781  | 3605  |
| 8  | smooth-4bits | 51’578    | 1’950          | 1781 | 955  | 637  | 1271  | 2505  | 4942  |
| 9  | smooth-f-2bits | 569    | 331            | 73   | 67   | 531  | 989   | 1990  | 3905  |
| 10 | smooth-b-2bits | 569    | 1’241          | 73   | 342  | 169  | 347   | 628   | 1304  |
| 11 | forward-6-3n+1 | 834   | 1’007          | 233  | 79   | 556  | 953   | 1882  | 4022  |
Most Recent LARA Photo

http://lara.epfl.ch