Integrating high-level constructs into programming languages

Language extensions to make programming more productive

Underspecified programs
  – give assertions, get code that enforces them
  – simplify programming, reasoning, testing

Pattern matching
  – widely used construct in functional programs
  – synthesis can make it more expressive
Synthesis as Scala-compiler plugin

Given number of seconds, break it into hours, minutes, leftover

```scala
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))
```

choose:(A => Boolean) => A

our synthesis procedure

```scala
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)
}
```

Warning: solution not unique for: totsec=60
Synthesis for Pattern Matching

```scala
def pow(base : Int, p : Int) = {
  def fp(m : Int, b : Int, i : 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)
}
```

Our Scala compiler plugin:
• generates code that does division and testing of reminder
• checks that all cases are covered
• can use any integer linear arithmetic expressions
Starting point: counterexample-generating decision procedures (validity)

Effective in verification: counterexample $\rightarrow$ error
Can we use it in synthesis?

Take negation of $F'$...
Starting point: counterexample-generating decision procedures (satisfiability)

- Formula is unsatisfiable (false for all x, y)
- Formula is true for (x1, y1)

Diagram:
- Formula (bool-valued expression): F(x, y)
Example: integer linear arithmetic

formula F with integer variables

\[10 < y \land x < 6 \land y < 3x\]

No a-priori bounds on integers (add e.g. \(0 \leq y < 2^{64}\) if needed)

true for \(x=4, \ y=11\)
function $g$ on integers

$g(x)(y) = (y + 1) \text{ div } 3$

Two kinds of variables:
- inputs – here $y$
- outputs – here $x$

$10 < y \land x < 6 \land y < 3 \times x$

- $P$ describes precisely when solution exists.
- $(g_x(y), y)$ is solution whenever $P(y)$

Synthesis Procedure

precondition $P$ on $y$

$10 < y < 14$

function $g$ on integers

$g_x(y) = (y + 1) \text{ div } 3$
How does it work?
Quantifier elimination

Take formula of the form
   \( \exists x. F(x,y) \)
replace it with an equivalent formula
   \( G(y) \)
without introducing new variables
Repeat this process to eliminate all variables

Algorithms for quantifier elimination (QE) exist for:
   – Presburger arithmetic (integer linear arithmetic)
   – set algebra
   – algebraic data types (term algebras)
   – polynomials over real/complex numbers
   – sequences of elements from structures with QE
Example: test-set method for QE (e.g. Weispfenning’97)

Take formula of the form
\[ \exists x. F(x, y) \]
replace it with an equivalent formula

\[ \forall i=1^n \ F_i(t_i(y), y) \]

We can use it to generate a program:

\[
x = \text{if } F_1(t_1(y), y) \text{ then } t_1(y) \\
\quad \text{else if } F_2(t_2(y), y) \text{ then } t_2(y) \\
\quad \ldots \\
\quad \text{else if } F_n(t_n(y), y) \text{ then } t_n(y) \\
\text{else throw new Exception(“No solution exists”)}
\]

Can do it more efficiently – generalizing decision procedures and quantifier-elimination algorithms (use div, %, …)
Example: Omega-test for Presburger arithmetic – Pugh’92
Presburger Arithmetic

\[ T ::= k \mid C \mid T_1 + T_2 \mid T_1 - T_2 \mid C \cdot T \]
\[ A ::= T_1 = T_2 \mid T_1 < T_2 \]
\[ F ::= A \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \mid \exists k.F \]

Presburger showed quantifier elimination for PA in 1929
  • requires introducing divisibility predicates
  • Tarski said this was not enough for a PhD thesis

Normal form for quantifier elimination step:

\[ \bigwedge_{i=1}^{L} a_i < x \land \bigwedge_{j=1}^{U} x < b_j \land \bigwedge_{i=1}^{D} K_i \mid (x + t_i) \]
Parameterized Presburger arithmetic

Given a base, and number convert a number into this base

```scala
val base = read(...)  
val x = read(...)  
val (d2, d1, d0) = choose((x2, x1, x0) =>
    x0 + base * (x1 + base * x2) == x &&
    0 <= x0 < base &&
    0 <= x1 < base)
```

This also works, using a similar algorithm

- This time essential to have ‘for’ loops
- ‘for’ loops are useful even for simple PA case
  - reduce code size, preserve efficiency
Beyond numbers
Synthesizing sets

Partition a set into two parts of almost-equal size

val s = ...
val (a1,a2) = choose((a1:Set[O],a2:Set[O])) ⇒
    a1 union a2 == s &&
    a1 intersect a2 == empty &&
    abs(a1.size - a2.size) ≤ 1)
Boolean Algebra with Presburger Arithmetic

\[ S ::= V \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2 \]

\[ T ::= k \mid C \mid T_1 + T_2 \mid T_1 - T_2 \mid C \cdot T \mid \text{card}(S) \]

\[ A ::= S_1 = S_2 \mid S_1 \subseteq S_2 \mid T_1 = T_2 \mid T_1 < T_2 \]

\[ F ::= A \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \mid \exists S.F \mid \exists k.F \]

Our results related to BAPA
- complexity for full BAPA (like PA, has QE)
- polynomial-time fragments
- complexity for Q.F.BAPA
- generalized to multisets
- combined with function images
- used as a glue to combine expressive logics
- synthesize sets of objects from specifications
Computational benefits of synthesis

Example: propositional formula $F$

```scala
var p = read(...); var q = read(...)
val (p0, q0) = choose((p, q) => F(p, q, u, v))
```

– SAT is **NP-hard**

– generate BDD circuit over input variables
  • for leaf nodes compute one output, if exists

– running through this BDD is **polynomial**

Reduced NP problem to polynomial one

Also works for linear rational arithmetic
  (build decision tree with comparisons)
new decision procedures
→
new synthesis algorithms
Combining decision procedures

formula in an expressive decidable logic

\[
\neg \text{next}^0(\text{root}^0, n1) \land x \not\in \{\text{data}^0(n) \mid \text{next}^0(\text{root}^0, n)\} \land \\
\text{next} = \text{next}^0[n1 := \text{root}^0] \land \\
\text{data} = \text{data}^0[n1 := x] \Rightarrow \\
|\{\text{data}(n) \mid \text{next}^*(n1, n)\}| = \\
|\{\text{data}^0(n) \mid \text{next}^0(\text{root}^0, n)\}| + 1
\]

formula is valid

formula has a counterexample
Combining formulas with disjoint signatures (current tools)

\[ x < y+1 \& y < x+1 \& x' = f(x) \& y' = f(y) \& x' = y' + 1 \]

- **decision procedure for integer arithmetic**
  - \( x < y+1 \)
  - \( y < x+1 \)
  - \( x' = y' + 1 \)
  - \( 0 = 1 \)

- **decision procedure for function symbols**
  - \( x = y \)
  - \( x' = f(x) \)
  - \( y' = f(y) \)
  - \( x' = y' \)
Some research directions of LARA (Lab for Automated Reasoning & Analysis)

- **Program verification** and analysis tools both language-independent techniques, and translations from Scala, Java, PHP to logical models
- **Decision procedures** for reasoning about: algebraic data types, multisets, sets, graphs
- Techniques to **combine** decision procedures
- **Program synthesis** from specifications
- **Dynamically deployed** analysis, synthesis
- Specification-based **systematic testing**
- Collaboration on such activities within Europe

http://RichModels.org