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

$$\textbf{val}\ (\text{hours},\ \text{minutes},\ \text{seconds}) = \text{choose}((\text{h: Int},\ \text{m: Int},\ \text{s: Int}) \Rightarrow ($$
$$?\text{h} * 3600 + ?\text{m} * 60 + ?\text{s} == \underline{\text{totsec}}$$
$$\&\&\ 0 \leq ?\text{m}\ \&\&\ ?\text{m} \leq 60$$
$$\&\&\ 0 \leq ?\text{s}\ \&\&\ ?\text{s} \leq 60))$$

parameter - variable in scope

`choose: (A => Boolean) => A`

our synthesis procedure

$$\textbf{val}\ (\text{hours},\ \text{minutes},\ \text{seconds}) = \{$$
$$\quad \textbf{val}\ \text{loc1} = \text{totsec div } 3600$$
$$\quad \textbf{val}\ \text{num2} = \text{totsec} + ((-3600) * \text{loc1})$$
$$\quad \textbf{val}\ \text{loc2} = \text{min}(\text{num2 div } 60,\ 59)$$
$$\quad \textbf{val}\ \text{loc3} = \text{totsec} + ((-3600) * \text{loc1}) + (-60 * \text{loc2})$$
$$\quad (\text{loc1},\ \text{loc2},\ \text{loc3})$$
$$\}$$

`Warning: solution not unique for: totsec=60`

Synthesis for Pattern Matching

```
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)
}
```

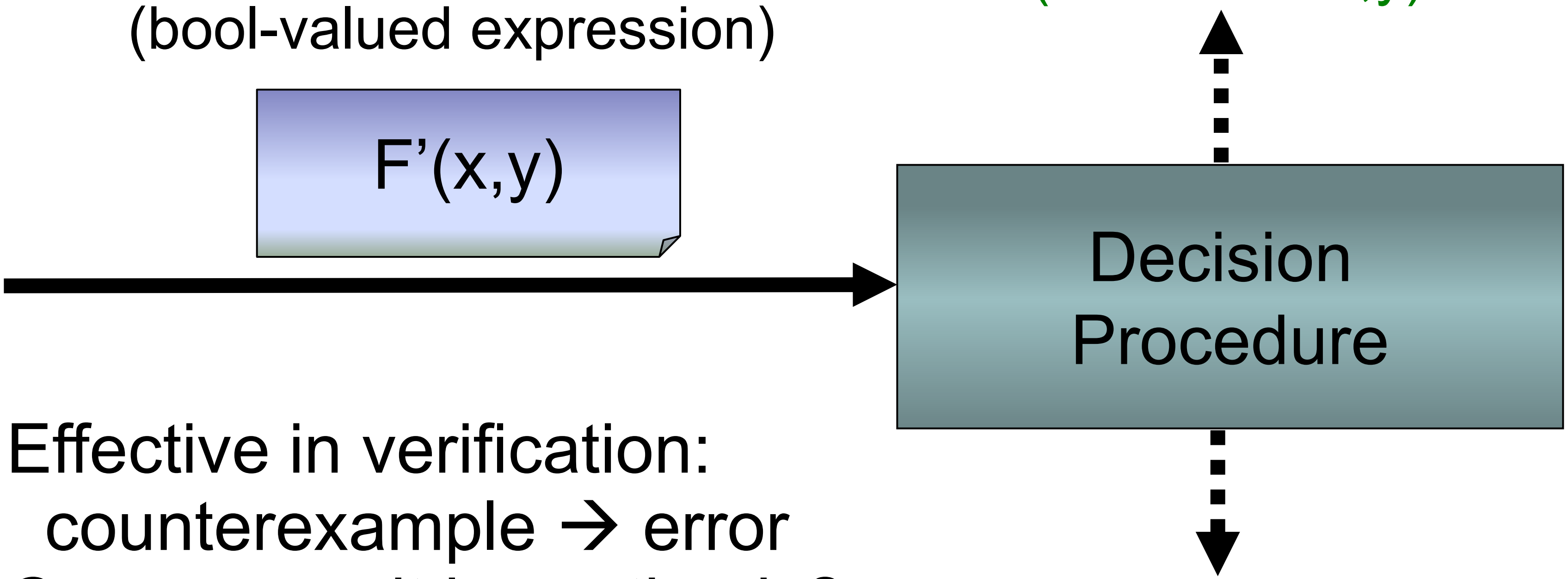
Our Scala compiler plugin:

- 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)

formula is valid
(true for all x,y)

formula
(bool-valued expression)

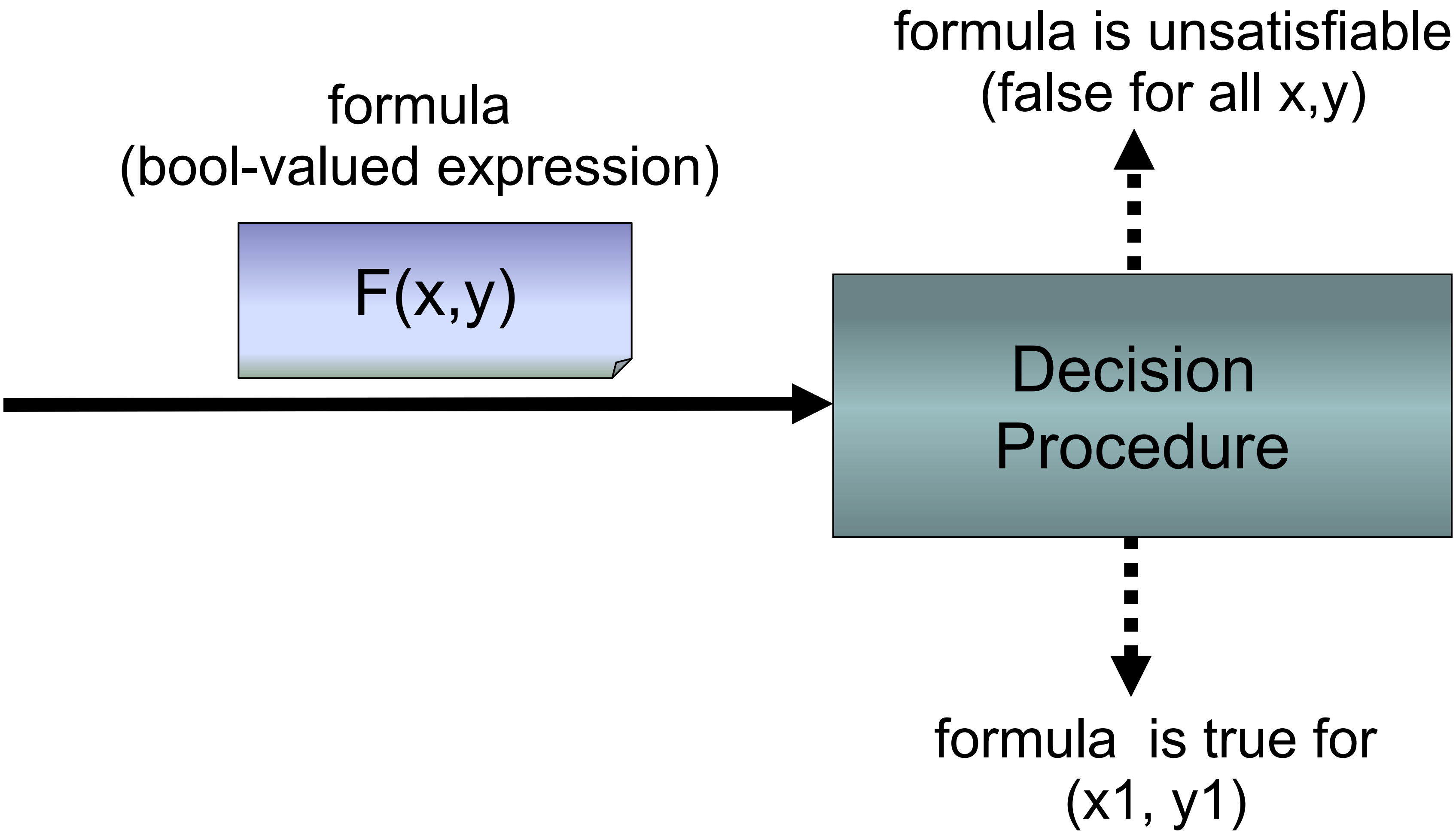


formula has a
counterexample
(x1, y1)
for which it is false

Effective in verification:
counterexample → error
Can we use it in synthesis?

Take negation of F'…

Starting point: counterexample-generating decision procedures (satisfiability)



Example: integer linear arithmetic

formula F with integer variables

$10 < y \wedge x < 6 \wedge y < 3*x$

Decision Procedure

true for
$x=4, \; y=11$

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

Synthesis procedure for integers

formula F with integer variables

$10 < y \wedge \mathbf{x} < 6 \wedge y < 3*x$

Synthesis Procedure

function g on integers
$g_x(y)=(y+1) \text{ div } 3$

precondition
P on y
$10 < y < 14$

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

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

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

$\bigvee_{i=1}^{n} F_i(t_i(y),y)$

We can use it to generate a program:

x = **if** $F_1(t_1(y),y)$ **then** $t_1(y)$
else if $F_2(t_2(y),y)$ **then** $t_2(y)$
…
else if $F_n(t_n(y),y)$ **then** $t_n(y)$
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

$T ::= k \mid C \mid T_1 + T_2 \mid T_1 - T_2 \mid C{\cdot}T$

$A ::= T_1 = T_2 \mid T_1 < T_2$

$F ::= A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists k.F$

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:

$$\bigwedge_{i=1}^{L} a_i < x \wedge \bigwedge_{j=1}^{U} x < b_j \wedge \bigwedge_{i=1}^{D} K_i \mid (x + t_i)$$

Parameterized Presburger arithmetic

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

```
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

$$\mathbf{S} ::= \mathbf{V} \mid \mathbf{S_1} \cup \mathbf{S_2} \mid \mathbf{S_1} \cap \mathbf{S_2} \mid \mathbf{S_1} \setminus \mathbf{S_2}$$

$$\mathbf{T} ::= \mathbf{k} \mid \mathbf{C} \mid \mathbf{T_1} + \mathbf{T_2} \mid \mathbf{T_1} - \mathbf{T_2} \mid \mathbf{C} \cdot \mathbf{T} \mid \mathbf{card(S)}$$

$$\mathbf{A} ::= \mathbf{S_1} = \mathbf{S_2} \mid \mathbf{S_1} \subseteq \mathbf{S_2} \mid \mathbf{T_1} = \mathbf{T_2} \mid \mathbf{T_1} < \mathbf{T_2}$$

$$\mathbf{F} ::= \mathbf{A} \mid \mathbf{F_1} \wedge \mathbf{F_2} \mid \mathbf{F_1} \vee \mathbf{F_2} \mid \neg \mathbf{F} \mid \exists \mathbf{S.F} \mid \exists \mathbf{k.F}$$

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

```
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

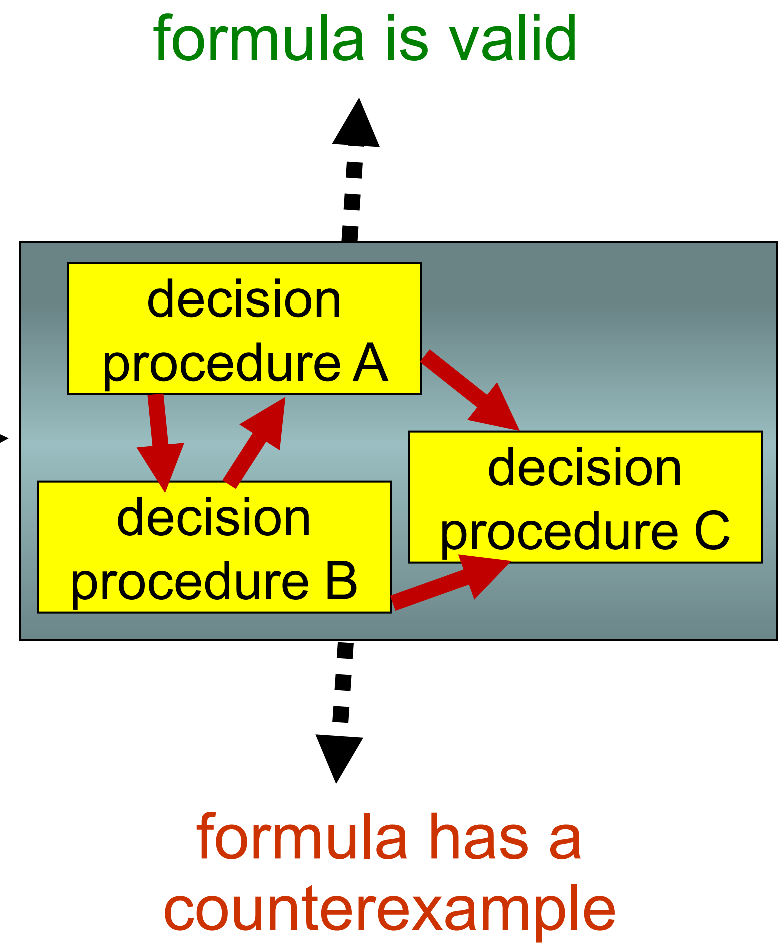
→

new synthesis algorithms

Combining decision procedures

formula in an expressive decidable logic

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



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		decision procedure for function symbols
$x < y+1$ $y < x+1$ $x'=y'+1$	$x=y$ → ← $x'=y'$	$x'=f(x)$ $y'=f(y)$

$0=1$

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