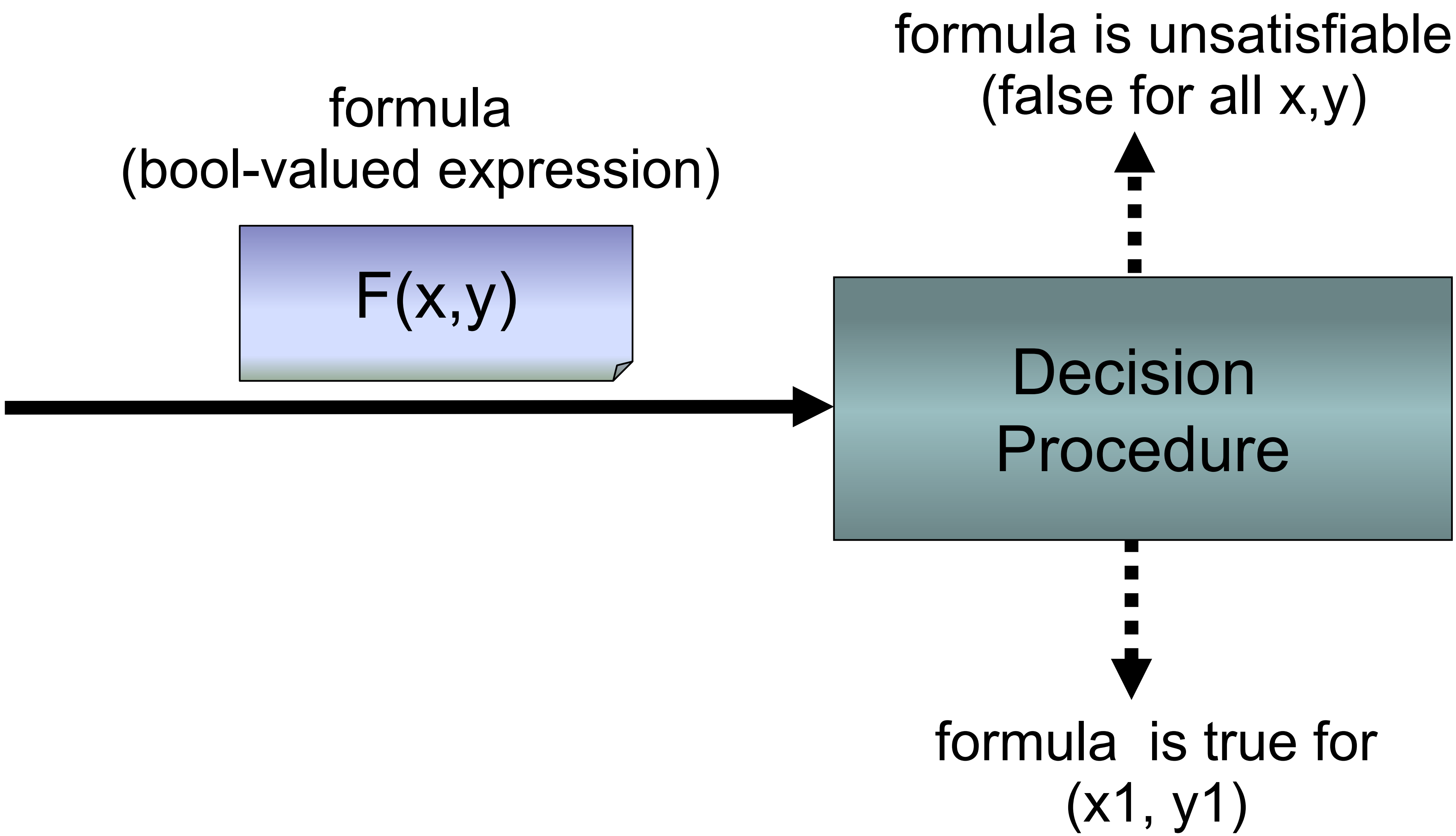
Starting point: counterexample-generating decision procedures (satisfiability)



Example: integer linear arithmetic

formula F with integer variables

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

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

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

function g on integers
$g_x(y)=(y+1)$ div 3

precondition
P on y
$10 < y < 14$

- 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

Synthesis as Scala-compiler plugin

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

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

parameter - variable in scope

our synthesis procedure

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

`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

Beyond numbers

Boolean Algebra with Presburger Arithmetic

$$S ::= V \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2$$
$$T ::= k \mid C \mid T_1 + T_2 \mid T_1 - T_2 \mid C \cdot T \mid \mathrm{card}(S)$$
$$A ::= S_1 = S_2 \mid S_1 \subseteq S_2 \mid T_1 = T_2 \mid T_1 < T_2$$
$$F ::= A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists S.F \mid \exists k.F$$

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**

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)
```

http://lara.epfl.ch/dokuwiki/comfusy

Complete Functional Synthesis

Scala progrmaming language – developed in Martin Odersky's group at EPFL

Time improvements 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)