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
Notion of Implicit Programming
Calculus of numbers and functions
(studied for centuries in real analysis)

two tasks

- compute $y : \ F(x) = y$
  
  $5 \cdot 7^2 + 2 \cdot 7 + 3 = y$
  
  $\sin'(t) = y(t)$

- solve for $x : \ F(x?) = y$

  $5 \cdot x^2 + 2 \cdot x + 3 = 42$
  
  $f'(t) = \cos(t)$
Calculus of Programs

Compute with numbers and with:
lists, trees, sets, relations

Again two tasks

• compute the result \( y : F(x) = y? \)
  – today’s software (imperative and functional)

• solve for \( x : F(x?) = y \)
  – verification: find input that crashes program, or invariants that prove correctness
  – synthesis: find program that satisfies the spec tomorrow’s software – implicit computation
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

```python
def f(x : Int) = {
    y = 2 * x + 1
}
```

```assembly
iload_0
iconst_1
iadd
```

```
42
```
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;
        }
    }
}
```

Enter December 31, 2008

all music players of a major brand freeze
Date Conversion in Scala$^\text{Z3}$ System

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

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

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
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
Implicit (Programming Language) Constructs
A concrete language

“Scalable programming language”

- Blending of functional and object-oriented programming
- Runs on the Java Virtual Machine (and .NET)
- Rich type system (generics, type classes, implicit conversions, etc.)
- Now used by over 100’000 developers (incl. Twitter, UBS, LinkedIn)

Notation for functions in Scala

\[
\begin{align*}
  x & \mapsto x^2 & \forall x. x^2 & \text{squaring function } f, \text{ such that } f(x) = x^2 \\
  (x, y) & \mapsto x + 2y & \forall x. x + 2y & \text{example linear function of arguments } x \text{ and } y \\
  x & \mapsto (x > 0) & \forall x. x > 0 & \text{returning true iff the argument is positive} \\
  (x, y) & \mapsto x + y < 10 & & \text{true if sum of its arguments is less than 10} \\
  (x, y) & \mapsto x > 0 \land y > 0 & & \text{true if both arguments positive}
\end{align*}
\]

predicate: function of type \((A_1, \ldots, A_n) \mapsto \text{Boolean}\)
Algebraic Data Types and Pattern Matching

Objective Caml

```
type tree = Leaf
  | Node of tree*int*tree
```

```
let rec elem x t =
  match t with
  | Leaf -> false
  | Node(l,y,r) ->
    if x=y then true
    else if x<y then elem x l
    else elem x r
```

Scala

```
sealed abstract class Tree

case class Leaf() extends Tree

case class Node(left:Tree, data:Int, right:Tree)
  extends Tree

def elem(x:Int,t:Tree):Boolean=
  t match {
    case Leaf() => false
    case Node(l,y,r) =>
      if (x==y) true
      else if (x<y) elem(x,l)
      else elem(x,r)
  }
```
Option Types

abstract case class Option[T]
case class None() extends Option[T]
case class Some(x:T) extends Option[T]

Example use of option types:

lookup(x,myMap) match {
  case None() => print("Nothing found")
  case Some(v) => print("Found: "+v)
}
Constraints

Semantically, constraint is a predicate

Syntactically, we specify them (for now) using
1) variable binding and 2) boolean expression

\[ x \mapsto (x^2 = a) : \text{Int} \mapsto \text{Boolean} \]

Here ‘a’ is parameter, some valid value of the language

In general, constraint \( C \) is a function of form

\[ (x_1, \ldots, x_n) \mapsto P(x_1, \ldots, x_n, a_1, \ldots, a_n) : \text{Int}^n \mapsto \text{Boolean} \]

where \( P \) has type \( \text{Boolean} \)

Given values \( a_1, \ldots, a_n \), the set of solutions of \( C \) is

\[ S(C) = \{(x_1, \ldots, x_n) \mid P(x_1, \ldots, x_n, a_1, \ldots, a_n) \} \]
findAll and find

C - a constraint     S(C) - its solution set,     S(C) = { x | C(x) }
S(x => x > 5) = {5,6,7,8,...}

1) findAll(C) = S(C)     findAll: (T => Boolean) => Set[T]
We represent result of findAll as iterator and use a for loop
   for ((a,b) <- findAll((x,y)=> 0 < x && x < y && x + y < 10)) {
      println("a = " + a + "b = " + b)
   }

2) find(C) =     if S(C)={} then None
   else Some(x) where x \in S(C)
find: (T => Boolean) => Option[T]

Example of use:
   find(x => (x*x==a)) match {
      case Some(v) => println("Square root of a is" + v)
      case None() => println("a is not a square of an integer")
   }
find variations: choose and given

// choose : (T => Boolean) => T
def choose(C : T => Boolean) : T = find(C) match {
  case Some(v) => v
  case None() => throw new Exception("no solution")
}
Example: println(choose(x => (x*x==a)))    // print x such that x*x=a

// given one x such that P(x), compute q(x); if none exists, compute r

given (x => P(x)) have q(x) else r

find(x => P(x)) match {
  case Some(y) => q(y)
  case None() => r
}
Example: given (k=> (2*k==a)) have k else 3*a+1
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.
\[ x \mapsto (x+x < a) \]

\[ r = \{ (a, x) \mid x+x < a \} \]

\( r \)

(10, 4)
(10, 3)
(10, -3)
(5, 1)
(5, 0)
(5, 2)

\[ \forall a. \quad p(a, f(a)) \quad f(a)+f(a) < a \]

\[ f \subseteq r \]

f

(10, 4)
(5, 2)

Find largest partial function \( f \) such that \( f \subseteq r \)
What Pattern Matching Means

```scala
sealed abstract class Tree
case class Leaf() extends Tree
case class Node(left: Tree, data: Int, right: Tree) extends Tree

t match {
  case Leaf() => q
  case Node(l, y, r) => r(l, y, r)
}

⇒ ∃l, y, r. t == Node(l, y, r)

if (t == Leaf()) q
else if (t.isInstanceOf[Node]) r(t.left, t.data, t.right)
else throw new Exception("unhandled match case")

⇒

given _ => t == Leaf() have q
else given (l, y, r) => t == Node(l, y, r) have r(l, y, r)
else throw new Exception("unhandled match case")
```
Generalized pattern matching by translation to \textit{given}

```
t match {
  case _ : Leaf() => q
  case l,y,r : Node(l,y,r) => r(l,y,r)
}
```

bound variables indicated explicitly, not by convention

... case \(v_1,\ldots,v_n\) : \(p_i(v_1,\ldots,v_n)\) => \(q_i(v_1,\ldots,v_n)\) ...

→ \(\text{... else given } v_1,\ldots,v_n \Rightarrow t==p_i(v_1,\ldots,v_n) \text{ have } q_i(v_1,\ldots,v_n) \text{ ...} \)

```
t match {
  case _ : Leaf() => ...
  case l,k,r : Node(l,2*k,r) => ...
  case l,k,r : Node(l,2*k+1,Node(l’,k+1,r’)) => ...
}
```
Halving Functions using `given`

```scala
def halfToZero(x: Int) : Int =
  given k=> x==2*k have k else
given k=> x==2*k+1 && k >= 0 have k else
given k=> x==2*k - 1 && k < 0 have k else error
}
def halfToMinusInf(x: Int) : Int =
  given k=> x==2*k have k else
given k=> x==2*k+1 && k >= 0 have k else
given k=> x==2*k+1 && k < 0 have k else error
}
```

Note: error can never happen in these examples
More
Implicit Programming Constructs
SAT Solver - Creating Constraints in Scala

Solving a CNF SAT instance in the standard DIMACS format

```scala
val p1 = Seq(Seq(1, -2, -3), Seq(2, 3, 4), Seq(-1, -4))

(x₁ ∨ ¬x₂ ∨ ¬x₃) ∧ (x₂ ∨ x₃ ∨ x₄) ∧ (¬x₁ ∨ ¬x₄)
```

```scala
def fromDimacs(problem : Seq[Seq[Int]]) : Constraint1[Map[Int, Boolean]] =
  problem.map(clause => clause.map(literal => {
    val id = abs(literal)
    val isPos = literal > 0
    ((m : Map[Int, Boolean]) => m(id) == isPos).c
  }).reduceLeft(_ || _)
    .reduceLeft(_ && _)
```

```scala
cscala> fromDimacs(p1).solve
```
```scala
cscala> Some(Map(2 -> true, 3 -> false, 1 -> false, 4 -> false))
```
```scala
cscala> fromDimacs(Seq(Seq(1, 2), Seq(-1), Seq(-2))).solve
```
```scala
cscala> None
```
sealed abstract class Tree

case class Leaf() extends Tree

case class Node(left : Tree, data : Int, right : Tree) extends Tree

def isSorted(t : Tree) : Boolean = {
  ...
}

def content(t : Tree) : Set[Int] = t match {
  case Leaf() ⇒ Set()
  case Node(l, d, r) ⇒ content(l) ++ Set(d) ++ content(r) }

def printTreesContaining(s : Set[Int]) = {
  for (t ← ((t : Tree) ⇒ isSorted(t) && content(t)===s).findAll)
    println(t) // replace with e.g. testUnitWithInput(t)
}

scala> printTreesContaining(Set(5,2,9))
Node(Node(Node(Leaf(),2,Leaf()),5,Leaf()),9,Leaf())
Node(Node(Leaf(),2,Node(Leaf(),5,Leaf())),9,Leaf())
Node(Node(Leaf(),2,Leaf()),5,Node(Leaf(),9,Leaf()))
Node(Leaf(),2,Node(Node(Leaf(),5,Leaf())),9,Leaf()))
Node(Leaf(),2,Node(Leaf(),5,Node(Leaf(),9,Leaf())))
Creating and Solving Knapsack Problem Instances

```scala
def solveKnapsack(vals : List[Int], weights : List[Int], max : Int) = {
  def conditionalSumTerm(vs : List[Int]) = {
    vs.zipWithIndex.map(pair => {
      val (v,i) = pair
      (m : Map[Int, Boolean]) => (if (m(i)) v else 0).i
    }).reduceLeft(_ + _)
  }

  val valueTerm = conditionalSumTerm(vals)
  val weightTerm = conditionalSumTerm(weights)
  val answer = ((x : Int) => x <= max).compose0(weightTerm)
               .maximizing(valueTerm)
               .solve
}

scala> val vals : List[Int] = List(4, 2, 2, 1, 10)
scala> val weights : List[Int] = List(12, 1, 2, 1, 4)
scala> val max : Int = 15
scala> solveKnapsack(vals, weights, max)
result:  Map(0 → false, 1 → true, 2 → true, 3 → true, 4 → true)
```
Lazy (Logical) Variables

Constraint Logic Programming
CLP(R), CLP(X)

```scala
val c1: Constraint2[Int,Int] =
  ((x: Int, y: Int) => 2*x + 3*y == 10 && x >= 0 && y >= 0)
scala> val (x,y) = c1.lazySolve; println((x,y))
result: (L(?),L(?))

scala>x.value
result: 5
scala>println((x,y))
result: (L(5),L(?))

scala> for ((x,y) <- c1.lazyFindAll if (x == y)) println((x,y))
(2,2)
```
About Holes

Program with a hole is a function taking a logical variable (lvar)
  – occurrence of the same lvar denotes same value
  – multiple lvars denote possibly different values

User-programmable synthesis:
  – finding values of logical variables to satisfy given correctness properties

Ongoing work:
  – retrofit holes into standard notions of non-determinism and staging
Interpretation:
Some Systems and Comparison to Compilation
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

Java + choose
- integers
- (fresh) objects

ignoref (....)
asume (!)
(Test) generators in UDITA are non-deterministic programs

UDITA’s delayed non-deterministic choice mechanism makes generation more efficient

Special support for generating bounded graphs (data structures) while avoiding isomorphic ones
JPF: a 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.

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() {...}
}
```
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.
Some Decidable Logics

Integer linear arithmetic (Presburger arithmetic)
Mixed arithmetic over reals and integers
Arithmetic with multiplication but no addition
Algebraic data types
Q.F. Uninterpreted Functions
Executable constraints over bounded domains (including bitvectors, bounded strings)
Monadic Second-Order Logic of Trees
Universal Class, Description Logics, Guarded Logics, Two-variable Logics with Counting
• Efficient SMT solver from Microsoft Research
• Supports many theories through DPLL\((T)\) and Nelson-Oppen combination
• SMT-LiB standard input format, as well as C, .NET, OCaml, and Python bindings

Scala

“Scalable programming language”

• Blending of functional and object-oriented programming
• Runs on the Java Virtual Machine (and .NET)
• Rich type system (generics, type classes, implicit conversions, etc.)
• Now used by over 100’000 developers (incl. Twitter, UBS, LinkedIn)
Example using Scala\(^{\text{\textsuperscript{Z}3}}\)

find triples of integers \(x, y, z\) such that \(x > 0, y > x, 2x + 3y \leq 40, x \cdot z = 3y^2,\) and \(y\) is prime

```scala
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
Scala^Z3
Invoking Constraint Solver at Run-Time

Java Virtual Machine
- functional and imperative code
- custom ‘decision procedure’ plugins

Q: implicit constraint
A: model
Q: queries containing extension symbols
A: custom theory consequences

Z3 SMT Solver

with: Philippe Suter, Ali Sinan Köksal
Constraints in Scala - Kaplan System (w/ Köksal, Suter)

Scala language: functions, objects, traits, mutation, exceptions, actor concurrency

- constraints: executable predicates in a purely functional sublanguage

general computations **create** first-class constraints

- as Boolean-valued Scala expressions with pure functions
- manipulated as well-typed trees (computed at run-time)

**solving** first-class constraints **controls** general computation

- constraint solver creates stream of solutions to constraint
- a **for** loop construct iterates over solutions and supplied them to general computation
- logical variables and global constraint store enable optimization across multiple **for** loops
Using recursive functions to specify the desired value

```python
def append(l1 : List, l2 : List) : List = l1 match {
case Nil() ⇒ l2
case Cons(x, xs) ⇒ Cons(x, append(xs, l2)) }
def snoc(lst : List) : (List,Int) =
((res,e) : (List,Int)) ⇒ lst == append(res, Cons(e, Nil())).solve

def addDeclarative(x: Int, tree: Tree) : Tree =
((t: Tree) ⇒ isRedBlackTree(t) &&
content(t) == content(tree) ++ Set(x)).solve
def removeDeclarative(x: Int, tree: Tree) : Tree =
((t: Tree) ⇒ isRedBlackTree(t) &&
content(t) == content(tree) -- Set(x)).solve
```
Comparison of 3 Runtime Approaches

• Scala\textsuperscript{Z3} is a library approach to using a constraint solving
  – efficient solving of given formulas
  – nice interface to Z3 from Scala
• UDITA is an extension of JavaPathfinder, a non-deterministic JVM (a few o.m. slower)
• Recent work: Kaplan
  – integrates non-determinism through for comprehensions (iterate over solutions)
  – supports symbolic logical variables (postpone solving)
  – has first class constraints, so subsumes Scala\textsuperscript{Z3}
This was interpretation...

```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)
```

This approach works for constraints in theories for which SMT solver is complete and provides model generation.
An Alternative: Compilation

```python
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 ))
```

```python
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
    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)
```
AUTOMATIC PROGRAMMING
PROPERTIES AND PERFORMANCE OF
FORTRAN SYSTEMS I AND II

by

J. W. BACKUS

To be presented at a Symposium on
The Mechanization of Thought Processes,
which will be held at the National Physical
Laboratory, Teddington, Middlesex, from 24th-
27th November 1958. The papers and the discussions
are to be published by H.M.S.O. in the Proceedings
of the Symposium. This paper should not be repro-
duced without the permission of the author and of
the Secretary, National Physical Laboratory.
constraint (relation) between inputs and outputs

function from inputs to outputs
Interpretation versus Compilation

Pros of interpretation
- 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 compilation
- 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
Compilation using Variable Elimination for Linear Integer Arithmetic
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

\[ (\exists \vec{x}. F(\vec{a}, \vec{x})) \iff \text{pre}(\vec{a}) \]

where \( a \) is a parameter

• Witness terms from QE are the generated program!
Quantifier Elimination for Linear Integer Arithmetic

• Problem of great interest:
  – [Presburger, 1929], [Cooper, 1972]
  – [Pugh, 1992],
  – [Weispfenning, 1997]
  – [Nipkow 2008] – verified in Isabelle

• Our algorithm for integers:
  – Efficient handling equalities
  – Handling of inequalities as in [Pugh 1992]
  – Computes **witness terms**, builds a program from them
Complete Functional Synthesis

Definition (Synthesis Procedure)
A synthesis procedure takes as input formula $F(x, a)$ and outputs:

1. a precondition formula $pre(a)$
2. list of terms $\Psi$

such that the following holds:

$$\exists x. F(x, a) \Leftrightarrow pre(a) \Leftrightarrow F[x := \Psi]$$

Note: $pre(a)$ is the “best” possible, range of relation
choose((x, y) \Rightarrow 5 \times x + 7 \times y == a \&\& x \leq y)

Corresponding quantifier elimination problem:

∃ x ∃ y . 5x + 7y = a \land x \leq 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 \leq y in terms of z:

5a \leq 12z

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