Software Construction using Executable Constraints

Presented by: Viktor Kuncak
joint work with:
Ali Sinan Köksal, Ruzica Piskac, and Philippe Suter

Swiss Federal Institute of Technology Lausanne (EPFL)

Arranging a Marriage of SMT with Programming Languages

A Research Program

Unify Programming Language and SMT technologies
For the benefits of both

- Programming Languages will finally be able to do what they always wanted to (constraint logic programming, synthesis)
- SMT technology will become even higher impact technique: essential to **run programs**, not only find errors or prove them

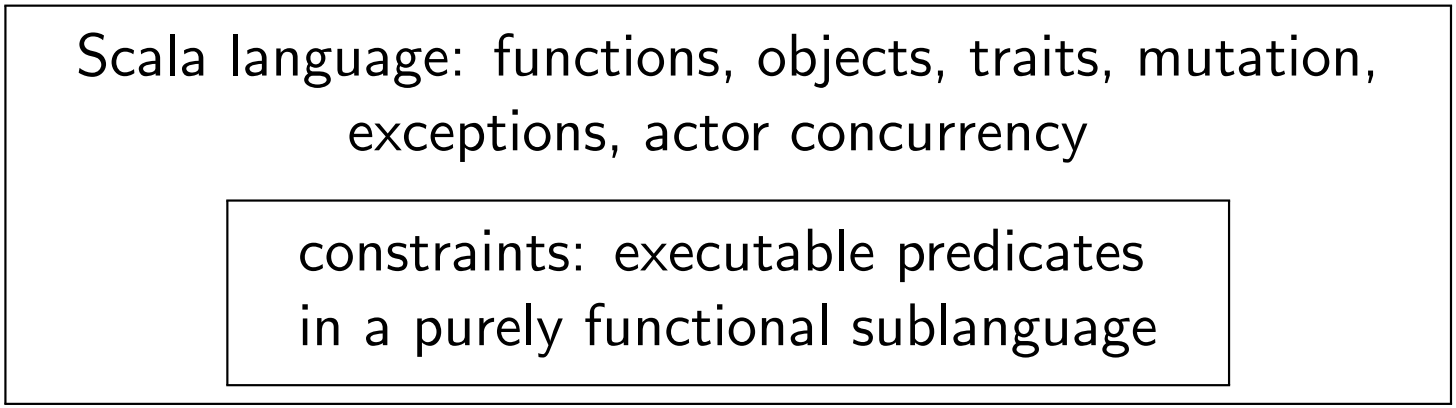
This agenda places new demands on SMT techniques:

- support for interfaces and functionality
  (model generation, standardized APIs, optimization)
- choice of theories to consider

I start by showing, through examples, a concrete system that we designed to explore this hypothesis

- then reflect on more technical results and directions

Constraints in Scala - Kaplan System (w/ Köksal, Suter)



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

Scala Language, http://www.scala-lang.org

```
sealed abstract class Tree
case class Leaf() extends Tree
case class Node(left : Tree, data : Int, right : Tree) extends Tree
def mirror(t : Tree) : Tree = t match {
  case Leaf() ⇒ Leaf()
  case Node(l, d, r) ⇒ Node(mirror(r), d, mirror(l)) }
val succ = ((x:Int) ⇒ x+1) // λx. x + 1
```

Scala is for scalability

- from concise scripting (concise syntax, local type inference, safe implicit conversions, read-eval-print loop)
- to efficient production systems (runs on JVM and .NET, benefits from their optimizations), Twitter switched to Scala
- scalable concurrency through actors

Interoperable with Java (use Java libraries, compile hybrid projects)
Excellent for embedding domain-specific languages

Basic Idea of Constraints

val c1: Constraint2[Int,Int] =
  ((x: Int, y: Int) $\Rightarrow$ 2*x + 3*y == 10 && x $\geq$ 0 && y $\geq$ 0)

scala>c1(2,1)
result: false
scala>c1(5,0)
result: true

scala>c1.solve
result: (5,0)
scala>c1.findAll
result: non−empty iterator

for(s $\leftarrow$ c1.findAll) println(s)
(5,0)
(2,2)

Execution of declarative specifications

```
def sqrt(i : Int) : Int =
  ((res: Int) ⇒ res > 0 && res * res ≤ i &&
                          (res + 1) * (res + 1) > i).solve

def secondsToTime(totalSec : Int) : (Int,Int,Int) =
  ((h,m,s) : (Int,Int,Int) ⇒
    totalSec == 3600*h + 60*m + s &&
    0 ≤ h && 0 ≤ m && 0 ≤ s &&
    m ≤ 59 && s ≤ 59).solve
```

Constructs for checking satisfiability and enumeration solutions:

- **solve**: Contraint[A] $\longrightarrow$ A
- **find**: Contraint[A] $\longrightarrow$ Option[A]
- **findAll**: Constraint[A] $\longrightarrow$ Seq[A]

SAT Solver - Creating Constraints in Scala

Solving a CNF SAT instance in the standard DIMACS format

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

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

```
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> fromDimacs(p1).solve
scala> Some(Map(2 → true, 3 → false, 1 → false, 4 → false)))
scala> fromDimacs(Seq(Seq(1,2), Seq(-1), Seq(-2))).solve
scala> None
```

Creating and Solving Knapsack Problem Instances

```
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

```
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)
```

Implementing logical variables

- We introduce a boolean guard for each logical variable, denoting its *liveness*, i.e. whether its value has not been *forced*
- Constraints associated with logical variables are stored in a global context, in the form $live_i \Rightarrow p_i(l_i)$
- We check the consistency of the context by querying whether there is a model with all active liveness variables set to true.
- When a variable l_i is forced to have a concrete value, we remove its guard from the set of alive guards.

Wish List for SMT Solver Interfaces

- Efficient (and fair) model enumeration
- Reliable model extraction, faithful mapping to language values (possibly shared store with programming language)
- Witnesses for quantifier elimination
- Parametrized models for synthesis (compilation)
- UNSAT cores for guiding search built on top of solver
- Incrementality and assumption management; push/pop and beyond
- Standard APIs: currently only Z3 in our system; it would be significant effort to add new one

SMT-LIB standardization activities, also `http://richmodels.org`

Enumerating Trees - how to do this, using an SMT solver?

```
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())))
```

Satisfiability Modulo Recursive Functions

- We increase the expressive power of the logic by integrating: user-defined recursive functions and algebraic data types
- A satisfiability procedure for this logic is at the core of the Leon functional verification system [SKK11].

`http://lara.epfl.ch/leon/`

Leon by example

Let us consider the following definition of lists in Scala:

```
sealed abstract class List
case class Cons(head: Int, tail: List) extends List
case class Nil() extends List
```

We can define a method that appends one list to another as:

```
def append(l1 : List, l2 : List) : List = (l1 match {
  case Nil() ⇒ l2
  case Cons(x,xs) ⇒ Cons(x, append(xs, l2))
}) ensuring(content(_) == content(l1) ++ content(l2))
```

And then prove that it is associative:

```
@induct
def appendAssoc(xs : List, ys : List, zs : List) : Boolean =
  (append(append(xs, ys), zs) == append(xs, append(ys, zs))) holds
```

Data Types Supported

Leon accepts a purely functional subset of Scala. The allowed types can be inductively defined as follows:

$$\textbf{Int} \in \mathcal{T} \qquad \textbf{Boolean} \in \mathcal{T} \qquad \frac{T \in \mathcal{T}}{\textbf{Set}[T] \in \mathcal{T}} \qquad \frac{T_1 \in \mathcal{T} \qquad T_2 \in \mathcal{T}}{\textbf{Map}[T_1, T_2] \in \mathcal{T}}$$

$$\frac{T_1 \in \mathcal{T} \qquad T_2 \in \mathcal{T}}{T_1 \Rightarrow T_2 \in \mathcal{T}} \qquad \frac{\textbf{sealed abstract class } C}{C \in \mathcal{T}}$$

$$\frac{\textbf{case class } C(\overline{n} : \overline{D}) \textbf{ extends } E \qquad \overline{D} \in \mathcal{T} \qquad E \in \mathcal{T}}{C \in \mathcal{T}}$$

Leon verification system

- Integrates into the DPLL($\mathcal{T}$) loop of Z3 and performs demand-driven unfolding of function definitions
  - treat recursive functions as un-interpreted after some depth
  - if we block recursive branches and get SAT, then SAT
  - if we unblock and get UNSAT, the problem is UNSAT
  - else keep unfolding
- Checks the faithfulness of candidate examples using code execution
- Is complete for counterexamples, often [SDK10] also for proofs

Leon generates verification conditions to prove that:

- Function contracts are valid
- Preconditions are met at all function invocations
- Pattern matching expressions are exhaustive

Example: precondition for partial maps

To prove that no runtime error can occur due to map accesses, we define **isDefinedForAll** and use it as a precondition:

```
def definedForAll(m: Map[Int,Int], l: List): Boolean = l match {
  case Nil() ⇒ true
  case Cons(x, xs) ⇒ m.isDefinedAt(x) && definedForAll(m, xs)
}

def map(m : Map[Int,Int], l : List) : List = {
  require(definedForAll(m, l))
  l match {
    case Nil() ⇒ Nil()
    case Cons(x, xs) ⇒ Cons(m(x), map(m, xs))
  }
}
```

Supporting function types

- We further extend the language with function types
- A Scala function of type **A** $\Rightarrow$ **B** is represented directly by a Z3 array of domain **A** and range **B**
- We can therefore prove properties of higher-order functions.

For instance, we prove that **append** and **map** operations commute:

```
@induct
def appendMapCommute(l1: List, l2: List, f: Int ⇒ Int): Boolean = {
  map(f, append(l1, l2)) == append(map(f, l1), map(f, l2))
} holds
```

Leon as a Verifier

Verification of data structure implementations and syntax tree manipulations:

Benchmark *(LOC, time in seconds)*:

- List *(107, 0.78s)*: tail-recursive size, associativity of append, distributivity of content over append
- Associative list *(50 LOC, 0.24s)*: content of union, *read-over-write*
- Insertion sort *(99 LOC, 0.43s)*: sortedness, size and content of output
- Red-black tree *(117 LOC, 3.75s)*: set interface, black-balancedness, "red nodes have black children"
- Propositional logic *(86 LOC, 2.37s)*: idempotence of NNF and simplification
- Sum and max *(46 LOC, 0.29s)*: sum less than size times maximum
- Search linked list *(48 LOC, 0.17s)*: position of first zero
- Amortized queue *(124 LOC, 3.4s)*: abstract queue interface, list size invariant

Enumerating Red-Black Trees with Leon's Help

```
sealed abstract class Tree
case class Leaf() extends Tree
case class Node(c : Color, l : Tree, v : Int, r : Tree) extends Tree
def size(tree : Tree) : Int = (tree match {
  case Leaf() ⇒ 0
  case Node(_, l, _, r) ⇒ 1 + size(l) + size(r)
}) ensuring(result ⇒ result ≥ 0)
def orderedKeys(t : Tree) : Boolean = ...
def validColoring(t : Tree) : Boolean = ...
def validTree(t : Tree) = orderedKeys(t) && validColoring(t)
def valuesWithin(t : Tree, min : Int, max : Int) : Boolean = ...
scala>(for(i ← (0 to 7)) yield ((t : Tree) ⇒ validTree(t) &&
    valuesWithin(i) && size(t) == i ).findAll.size).toList
result: List(1, 1, 2, 2, 4, 8, 16, 33)
```

Using recursive functions to specify the desired value

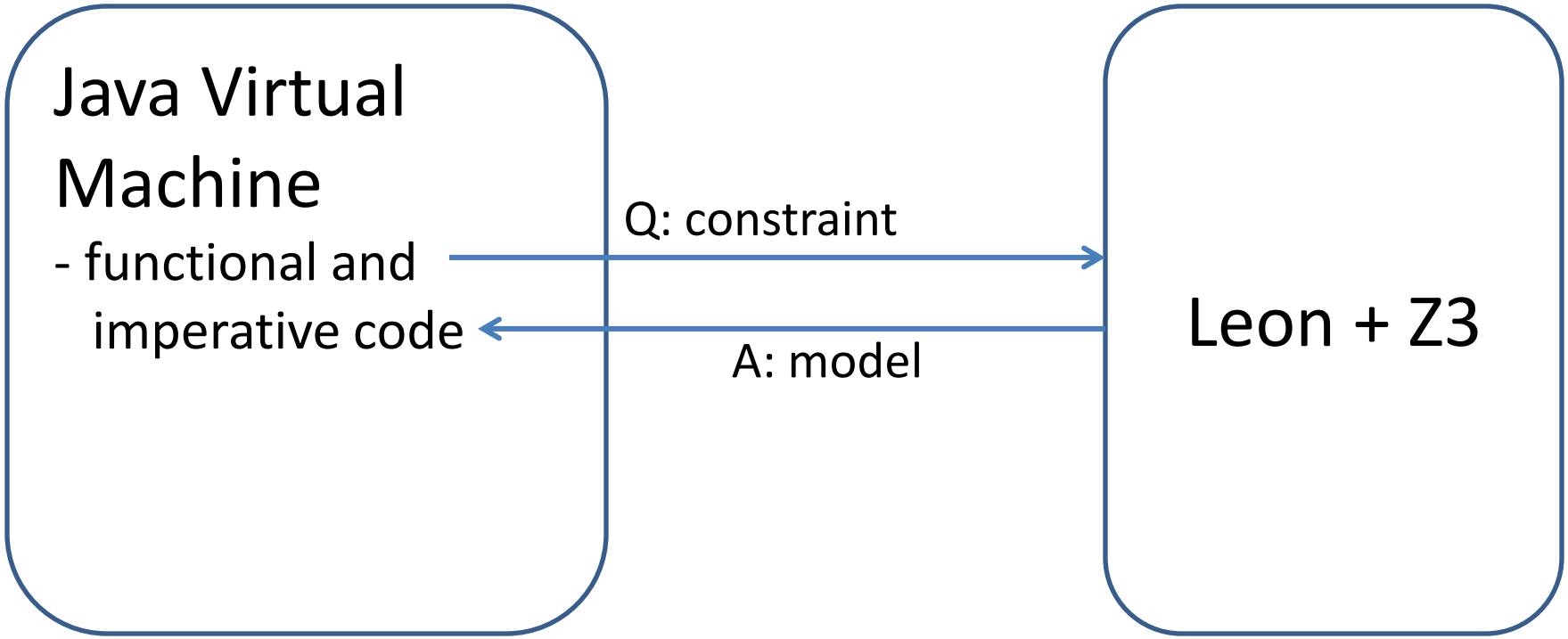
```
def append(l1 : List, l2 : List) : List = l1 match {
  case Nil() ⇒ l2
  case Cons(x, xs) ⇒ Cons(x, append(xs, l2)) }
def snoc(lst : List) : (List,Int) =
  ((res,e) : (List,Int)) ⇒ lst == append(res, Cons(e, Nil()))).solve

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
```

References


Philippe Suter, Mirco Dotta, and Viktor Kuncak.
Decision procedures for algebraic data types with abstractions.
In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010.


Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak.
Satisfiability modulo recursive programs.
In *SAS*, 2011.

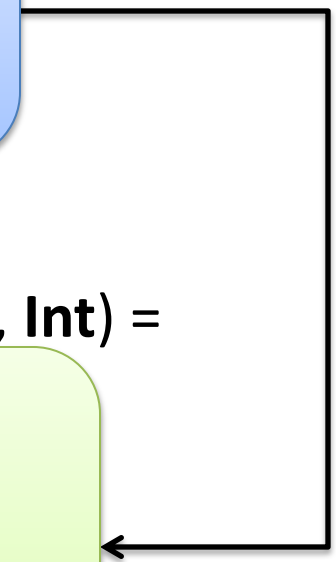
Invoking Constraint Solver at Run-Time



What would ideal code look like?

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
  ((h: Int, m: Int, s: Int) ⇒ (
        h * 3600 + m * 60 + s == totalSeconds
    && h ≥ 0
    && m ≥ 0 && m < 60
    && s ≥ 0 && s < 60   )).solve
```

synthesis



```
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)
```

Comparing with runtime invocation

Pros of runtime invocation

- Conceptually simpler
- Can use off-the-shelf solver
- for now can be more expressive and even faster
- but:

```
val times =
  for (secs ← timeStats)
    yield secondsToTime(secs)
```

Pros of synthesis

- 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

Possible starting point: quantifier elimination

- A specification statement of the form

$$\vec{r} = \textbf{\textit{choose}}(\vec{x} \Rightarrow F(\vec{a}, \vec{x}))$$

"*let* r *be* x *such that* F(a, x) *holds*"

- Corresponds to constructively solving the **quantifier elimination** problem

$$\exists \vec{x}.\ F(\vec{a}, \vec{x})$$

where a is a parameter

- Witness terms from QE are the generated program!

choose((x, y) ⇒ 5 * x + 7 * y == a && x ≤ y)

Corresponding quantifier elimination problem:

$$\exists x \exists y \,.\, 5x + 7y = a \wedge x \leq y$$

Use extended Euclid's algorithm to find particular solution to 5x + 7y = a:

x = 3a
y = -2a

(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 ≤ y in terms of z:

5a ≤ 12z

→ z ≥ ceil(5a/12)

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) ⇒ 5 * x + 7 * y == a && x ≤ y && x ≥ 0)

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:

z ≥ ceil(5a/12)

Rewrite x ≥ 0:

z ≤ floor(3a/7)

Precondition on a:

ceil(5a/12) ≤ floor(3a/7)

(exact precondition)

Obtain synthesized program:

```
assert(ceil(5*a/12) ≤ floor(3*a/7))
val z = ceil(5*a/12)
val x = -7*z + 3*a
val y = 5*z + -2*a
```

With more inequalities and divisibility: generate ‘for’ loop

NP-Hard Constructs

- Disjunctions
  - Synthesis of a formula computes program and exact precondition of when output exists
  - Given disjunctive normal form, use preconditions to generate if-then-else expressions (try one by one)
- Divisibility combined with inequalities:
  - corresponding to big disjunction in q.e. , we will generate a for loop with constant bounds (could be expanded if we wish)

Compile-time warnings

```
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
```

Compile-time warnings

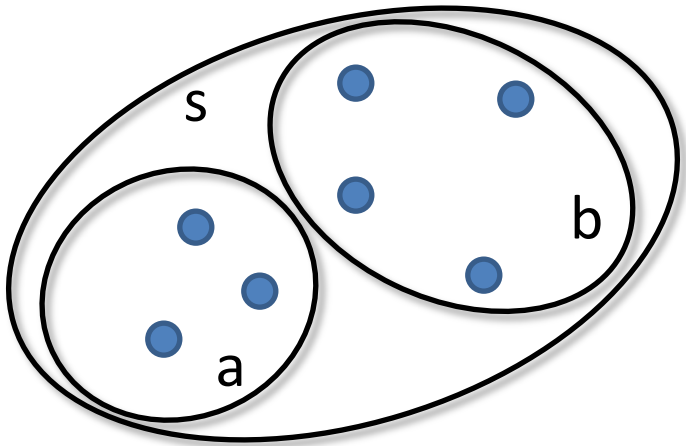
```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
  choose((h: Int, m: Int, s: Int) ⇒ (
        h * 3600 + m * 60 + s == totalSeconds
    && h ≥ 0
    && m ≥ 0 && m ≤ 60
    && s ≥ 0 && s < 60
  ))
```

```
Warning: Synthesis predicate has multiple
solutions for variable assignment:
  totalSeconds = 60
Solution 1: h = 0, m = 0, s = 60
Solution 2: h = 0, m = 1, s = 0
```

Synthesis for sets

```
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
  ))
```

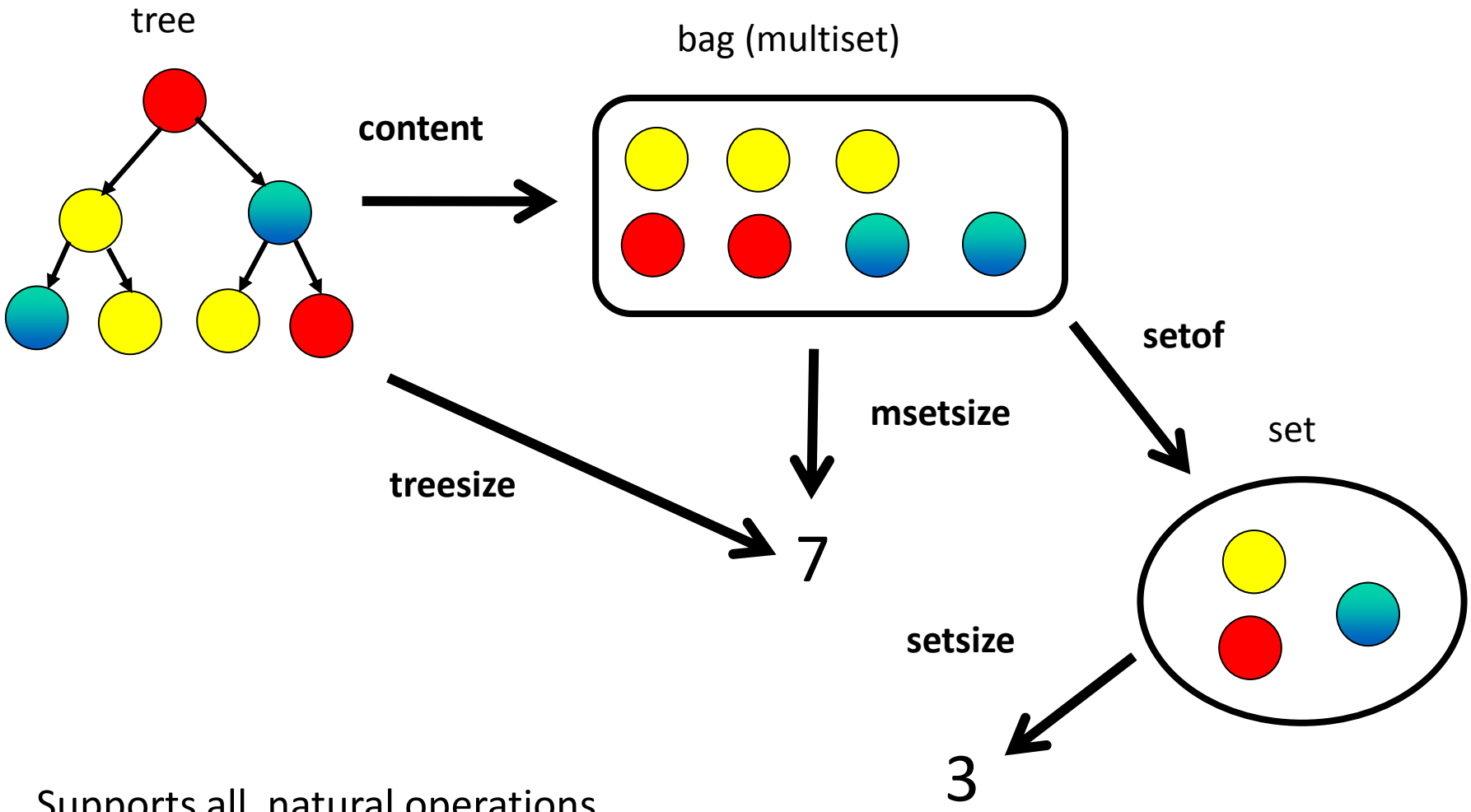
```
def splitBalanced[T](s: Set[T]) : (Set[T], Set[T]) =
  val k = ((s.size + 1)/2).floor
  val t1 = k
  val t2 = s.size – k
  val s1 = take(t1, s)
  val s2 = take(t2, s minus s1)
  (s1, s2)
```



Key Ingredient to Compilation

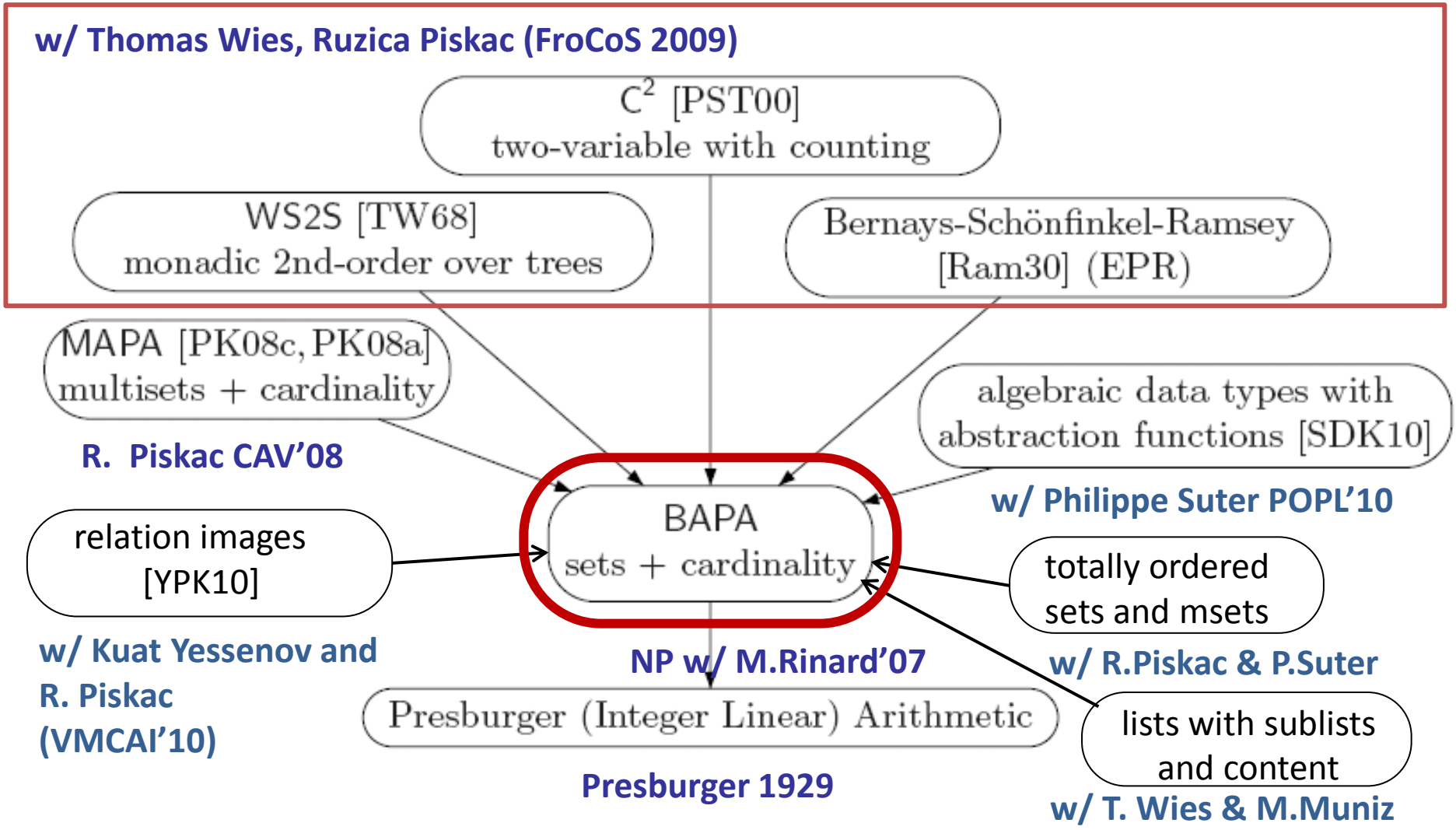
- Parameterized decision procedure
  - solves satisfiability for formulas with parameters
  - quantifier elimination procedures are an example
  - in general, key question is how much we can save by specialization
- In any case, we need
  - decision procedure for more data types used in programming

Decision Procedures Inspired by Functional Programs



Supports all natural operations
on trees, multisets, sets, and homomorphisms between them

BAPA: a "SAT" of combinations of rich logics



Relationship to Verification

- **Some** functionality is best **synthesized** from specs
  - **other**: better **implemented, verified** (and put into library)
- Currently, no choice –must always implement
  - specifications and verification are viewed as **overhead**
- Goal: make specifications intrinsic part of programs, with clear benefits to programmers – **they run!**
- Example: write an assertion, not how to establish it
- This will help both
  - verifiability: document, not reverse-engineer the invariants
  - productivity: avoid writing messy implementation

More Tomorrow in PSY

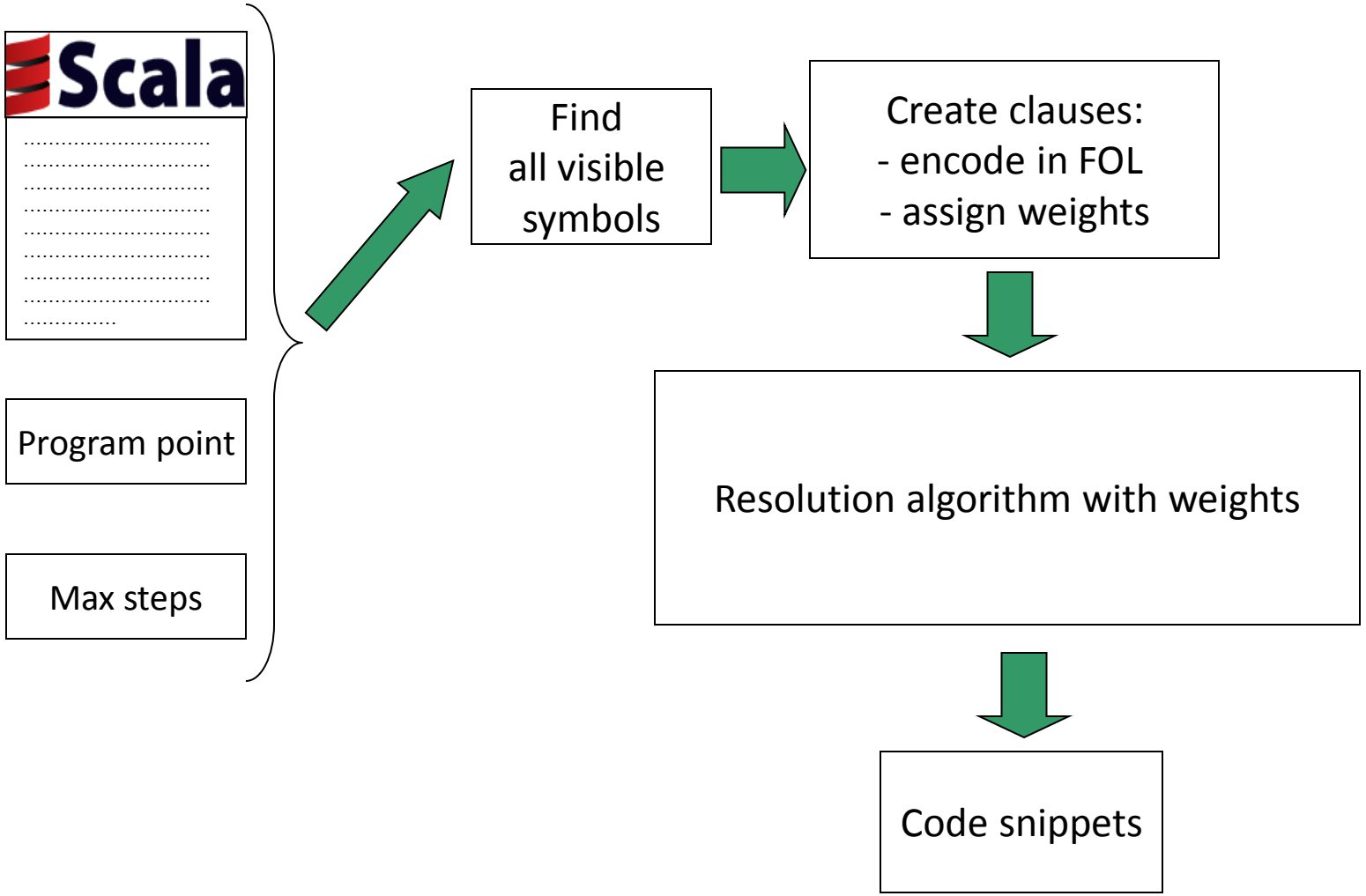
Compilation

- **RegSy** : solving WS1S constraints

Development within an IDE

- **isynth** tool – theorem proving as code completion

Interactive Synthesis of Code Snippets (Monday Demo)



Summary

- Expressive and reasonably efficient general-purpose constraint programming
  - functional language of first-class constraints
  - Z3 to handle recursion-free fragment
  - unfolding to handle recursion
- Model enumeration, APIs are important
- Compilation approach:
  - example based on quantifier elimination (Comfusy)
  - more decidable fragments by reduction to BAPA

http://lara.epfl.ch/w/impro