Viktor Kuncak

Implicit Programming

http://lara.epfl.ch/w/impro
How to obtain reliable software?

software verification:

specification → verifier → formulas → theorem prover

software synthesis:

specification (formula) → constructive theorem prover → code

Program verification: No specification (formula) results in a no output from a constructive theorem prover.
In 1958 it meant FORTRAN (FORmula TRANslation)

What would automatic programming mean today?
Programming Activities

Consider three related activities:

- **Development** within an IDE (Eclipse, Visual Studio, emacs)
- **Compilation** and static checking (optimizing compiler, static analyzer, contract checker)
- **Execution** on a (virtual) machine

More compute power available for each

→ *use it to improve programmer productivity*
Implicit Programming Agenda

Advancing
• **Development** within an IDE
• **Compilation** and static checking
• **Execution** on a (virtual) machine

Automated reasoning is key enabling technology

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

```
ild_0
iconst_1
iadd
```

```
42
```
Example: Date Conversion in C

Knowing number of days since 1980, find current year and day

```c
BOOL ConvertDays(UINT32 days) {
    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) froze in the boot sequence.
Implicit Programming for Date Conversion

Knowing number of **days** since 1980, find current **year** and **day**

```
val origin = 1980 // beginning of the universe
@spec def leapsTill(y : Int) = (y-1)/4 - (y-1)/100 + (y-1)/400

val (year, day)=choose( (year:Int, day:Int) => {
  days == (year-origin)*365 + leapsTill(year)-leapsTill(origin) + day &&
  0 < day && day <= 366
}) // Choose year and day such that the property holds.
```

- We did not write how to compute **year** and **day**
- Instead, we gave a **constraint** they should satisfy
- We defined them *implicitly*, though this constraint
  - More freedom (can still do it the old way, if needed)
  - Correctness, termination simpler than with loop
Solving Implicit Constraints – 2 ways

Solve each time, knowing concrete parameter values:

\[
\text{leapsTill}(y) = \frac{y-1}{4} - \frac{y-1}{100} + \frac{y-1}{400}
\]

\[
\text{days} = (\text{year}-1980) \times 365 + \text{leapsTill}() - \text{leapsTill}(1980) + \text{day} \land \land \text{day} \leq 366
\]

Compile (synthesize) constraint into explicit program

\[
\text{leapsTill}(y) = \frac{y-1}{4} - \frac{y-1}{100} + \frac{y-1}{400}
\]

\[
\text{days} = (\text{year}-1980) \times 365 + \text{leapsTill}() - \text{leapsTill}(1980) + \text{day} \land \land \text{day} \leq 366
\]
Key Technology: Automated Reasoning

Theorem proving: semi-decidable problems:
given a logical formula F in some language
  – find a proof that F is valid (¬F is unsatisfiable), or
  – find a counterexample (¬F has satisfying assignment)

Decision procedures for decidable logics
  – always terminate (detect a proof or formula exists)
  – gives back a model – constraint solver for infinite domains
  – each logic a different story, a new algorithm

Quantifier elimination: solve parametric problems – compilation

SAT – amazing progress in the last decade (despite P ≠ NP)
  – problems in practice exhibit structure that reduces search space
  – efficient propagation, clause learning, restarts
    (Sharad Malik - SuRI 2011, Sakallah, ...)

SMT = SAT + decision procedureS
  – Satisfiability Modulo Theory Solvers, Recent talk by Nikolaj Bjorner
Implicit Programming Agenda

Advancing

3) • **Development** within an IDE

2) • **Compilation** and static checking

1) • **Execution** on a (virtual) machine

Automated reasoning is key enabling technology

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

```
iload_0
iconst_1
iadd
```

```42```

```
lload_0
iconst_1
iadd
```
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

Leon constraint solver using Z3 (Bjorner, de Moura)

Suter, Köksal, Kuncak - SAS’11, CADE’11, POPL 2012
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))

(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_4)
```

```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> 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

```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
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)
```
sealed abstract class List

case class Cons(head : Int, tail : List) extends List

case class Nil() extends List

def content(lst : List) = lst match {
  case Nil() ⇒ Set.empty
  case Cons(x, xs) ⇒ Set(x) ++ content(xs)
}

def isSorted(lst : List) = lst match {
  case Nil() ⇒ true
  case Cons(_, Nil()) ⇒ true
  case Cons(x, xs @ Cons(y, ys)) ⇒ x < y && isSorted(xs)
}

val s = Set(0, 1, -3)
((l : List) ⇒ isSorted(lst) && content(lst) == s).solve

> Cons(-3, Cons(0, Cons(1, Nil())))
Red-Black Trees

12 Binary Search Trees  
12.1 What is a binary search tree? 286
12.2 Querying a binary search tree 289
12.3 Insertion and deletion 294
12.4 Randomly built binary search trees 299

13 Red-Black Trees 308
13.1 Properties of red-black trees 308
13.2 Rotations 312
13.3 Insertion 315
13.4 Deletion 323

14 Augmenting Data Structures 339
14.1 Dynamic order statistics 339
14.2 How to augment a data structure 345
14.3 Interval trees 348
Chapter 12 showed that a binary search tree of height \( h \) can support any of the basic dynamic-set operations—such as \texttt{SEARCH}, \texttt{PREDECESSOR}, \texttt{SUCCESSOR}, \texttt{MINIMUM}, \texttt{MAXIMUM}, \texttt{INSERT}, and \texttt{DELETE}—in \( O(h) \) time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take \( O(\lg n) \) time in the worst case.

### 13.1 Properties of red-black trees

A red-black tree is a binary search tree with one extra bit of storage per node: its \texttt{color}, which can be either \texttt{RED} or \texttt{BLACK}. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately \texttt{balanced}.

Each node of the tree now contains the attributes \texttt{color}, \texttt{key}, \texttt{left}, \texttt{right}, and \texttt{p}. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value \texttt{NIL}. We shall regard these \texttt{NIL}s as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following red-black \textbf{properties}:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (\texttt{NIL}) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

\textbf{Implementation: next 30 pages}

\textbf{Is there a simpler way?}
Formalize Invariants (in Scala)

```scala
sealed abstract class Tree

case class Empty() extends Tree

case class Node(color: Color, left: Tree, value: Int, right: Tree) extends Tree

def blackBalanced(t : Tree) : Boolean = t match {
  case Node(_,l,_,r) => blackBalanced(l) && blackBalanced(r) &&
    blackHeight(l) == blackHeight(r)
  case Empty() => true
}

def blackHeight(t : Tree) : Int = t match {
  case Empty() => 1
  case Node(Black(), l, _, _) => blackHeight(l) + 1
  case Node(Red(), l, _, _) => blackHeight(l)
}

def rb(t: Tree) : Boolean = t match {
  case Empty() => true
  case Node(Black(), l, _, r) => rb(l) && rb(r)
  case Node(Red(), l, _, r) => isBlack(l) && isBlack(r) && rb(l) && rb(r)
}

... def isSorted(t:Tree) = ...
```
Define Abstraction as ‘tree fold’

```python
def content(t: Tree) : Set[Int] = t match {
  case Empty() => Set.empty
  case Node(_, l, v, r) => content(l) ++ Set(v) ++ content(r)
}
```

```
{ 2, 4, 5, 7, 9 }
```
We can now define insertion

def insert(x : Int, t : Tree) = choose(t1:Tree =>
   isRBT(t1) && content(t1) = content(t) ++ Set(x))
Evolving the Program

Suppose we have a red-black tree implementation.

We only implemented ‘insert’ and ‘lookup’.

Now we also need to implement ‘remove’.
void RBDelete(rb_red_blk_tree* tree, rb_red_blk_node* z) {
    rb_red_blk_node* y;
    rb_red_blk_node* x;
    rb_red_blk_node* nil=tree->nil;
    rb_red_blk_node* root=tree->root;
    y=(z->left == nil) || (z->right == nil) ? TreeSuccessor(tree,z);
    x=*y;
    y=y->parent;
    if (!(y->left == nil) ? y->right : y->left)
        x=x->left;
    if (root == (x->parent = y->parent)) { /* assignment of y->p to x->p is intentional */
        root->left=x;
    } else {
        if (y == y->parent->left) {
        y->parent->left=x;
        } else {
        y->parent->right=x;
        }
    }
    if (y != z) { /* y should not be nil in this case */
        if (!y->red) RBDeleteFixUp(tree,x);
        tree->DestroyKey(z->key);
        tree->DestroyInfo(z->info);
        y->left=z->left;
        y->right=z->right;
        y->parent=z->parent;
        y->red=z->red;
        z->left->parent=z->right->parent=y;
        if (z == z->parent->left) {
            z->parent->left=y;
        } else {
            z->parent->right=y;
        }
    }
    free(z);
    } else {
    tree->DestroyKey(y->key);
    tree->DestroyInfo(y->info);
    if (!y->red)) RBDeleteFixUp(tree,x);
    free(y);
    }
    #ifdef DEBUG_ASSERT
    Assert((y!=tree->nil),"y is nil in RBDelete\n");
    #endif
    /* y is the node to splice out and x is its child */
    if (!((y->red)) RBDeleteFixUp(tree,x);
    tree->DestroyKey(z->key);
    tree->DestroyInfo(z->info);
    y->left=z->left;
    y->right=z->right;
    y->parent=z->parent;
    y->red=z->red;
    z->left->parent=z->right->parent=y;
    if (z == z->parent->left) {
        z->parent->left=y;
    } else {
        z->parent->right=y;
    }
    free(z);
} else {
    if (!y->left->red) {
        w->red=0;
        x->parent->red=1;
        LeftRotate(tree,x->parent);
        w=x->parent->right;
    }
    if (w->red) {
        if (w->right) {
            x=x->parent->red;
            x->parent->red=0;
            w=x->parent->right;
            RightRotate(tree,x->parent);
            x=root; /* this is to exit while loop */
        } else if (!w->left->red) {
            w->red=1;
            x=x->parent;
        } else {
            w->red=x->parent->red;
            x->parent->red=0;
            w->right->red=0;
            LeftRotate(tree,x->parent);
            x=root; /* this is to exit while loop */
        }
    } else { /* the code below is has left and right switched from above */
    w=x->parent->left;
    if (w->red) {
        w->red=0;
        x->parent->red=1;
        RightRotate(tree,w);
        w=x->parent->right;
    }
    w->red=w->parent->red;
    x->parent->red=0;
    w->right->red=0;
    LeftRotate(tree,x->parent);
    x=root; /* this is to exit while loop */
    }
    
140 lines of tricky C, reusing existing functions
Unreadable without pictures
(from Emin Martinian)
def remove(x : Int, t : Tree) = choose(t1:Tree =>
  isRBT(t1) && content(t1)=content(t) – Set(x))

The biggest expected payoff:

declarative knowledge is more reusable
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())))

Can use it to test even code that is not in Scala or on JVM
Recursive Functions in Constraints

If a function with a simple recursive schema maps many elements to one (e.g. content, size, ...), then adding it to logic preserves decidability

(w/ Philippe Suter, POPL’10)

→ extend the power of many decidable theories

A verifier for functional Scala code,

– complete for all counterexamples
– often finds proofs – detects that counterexamples do not exist
– semi algorithm which decides above class

(w/ Suter and Koeksal: SAS’11)

→ http://lara.epfl.ch/leon/

Use this semi algorithm not just for counter-example search, but for actual computation

(w/ Suter and Koeksal: POPL’12)

→ tool being released soon
Implicit Programming Agenda

Advancing

3) **Development** within an IDE

2) **Compilation** and static checking

1) **Execution** on a (virtual) machine

Automated reasoning is key enabling technology

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

```
iload_0
iconst_1
iadd
```

```
42
```
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 ))

def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =
val t1 =
val t2 =
val t3 =
val t4 =
(t1, t3, t4)
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 (QE) problem

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

where \( \vec{a} \) are parameters

• Witness terms from QE are the synthesized code
Methodology QE $\rightarrow$ Synthesis

Quantifier Elimination: $\exists x. S(x,a) \Leftrightarrow P(a)$

Synthesis: $\exists x. S(x,a) \Leftrightarrow S(t(a),a)$

For all quantifier elimination procedures we looked at we were able to find the corresponding witness terms $t$

- one-point rule immediately gives a term
  $\exists x. (x = t(a) \&\& S(x,a)) \Leftrightarrow S(t(a),a)$
- change variables, using a computable function
- strengthen formula while preserving realizability
- recursively eliminate variables one-by-one

Example:

$\exists x. (a1 < x \& x < a2) \Leftrightarrow a1 + 1 < a2 \quad t(a1,a2)=a1+1$
Synthesis Procedure: Equalities

Process equalities first:
• compute parametric description of solution set
• replace n variables with n-1

\[ h \times 3600 + m \times 60 + s = \text{totalSeconds} \]

\[ s = \text{totalSeconds} - h \times 3600 - m \times 60 \]

In general we obtain divisibility constraints
  – use Extended Euclid’s Algorithm,
    matrix pseudo inverse in \( \mathbb{Z} \)
Synthesis Procedure: Inequalities

• Solve for one by one variable:
  – separate inequalities depending on polarity of $x$:
    $$A_i \leq \alpha_i x$$
    $$\beta_j x \leq B_j$$
  – define values $a = \max_i [A_i/\alpha_i]$ and $b = \min_j [B_j/ \beta_j]$
• If $b$ is defined, return $x = b$ else return $x = a$
• Further continue with the conjunction of all formulas $[A_i/\alpha_i] \leq [B_j/\beta_j]$
• Similar to Fourier-Motzkin elimination (remove floor and ceiling using divisibility)
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 )
  )

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)

Implemented as an extension of the Scala compiler.
Properties of Synthesis Algorithm

• For every formula in linear integer arithmetic
  – synthesis algorithm terminates
  – produces the most general precondition
    (assertion saying when result exists)
  – generated code gives correct values whenever correct values exist

• If there are multiple or no solutions for some parameters, we get a warning
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
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
Arithmetic pattern matching

```python
def fastExponentiation(base: Int, power: Int) : Int = {
    def fp(m: Int, b: Int, i: Int): 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)
}
```

• Goes beyond Haskell’s $(n+k)$ patterns
• Compiler checks that all patterns are reachable and whether the matching is exhaustive
Synthesis for non-linear arithmetic

```python
def decomposeOffset(offset: Int, dimension: Int) : (Int, Int) =
choose((x: Int, y: Int) ⇒ (  
    offset == x + dimension * y && 0 ≤ x && x < dimension  
))
```

- The predicate becomes linear at run-time
- Synthesized program must do case analysis on the sign of the input variables
- Some coefficients are computed at run-time
Alternative: Automata-Based Synthesis

- Disjunctions can be handled more efficiently
- Result not depends on the syntax of input formula
- Complexity of running synthesized code: *nearly linear in the number of bits (log in values!)*
- Modular arithmetic and bitwise operators: can synthesize bit manipulations for
  - unbounded number of bits, uniformly
  - without skeletons
- Supports quantified constraints
  - including optimization constraints
- Based on WS1S. Improvements with Andrej Spielmann

Joint work with Jad Hamza and Barbara Jobstmann (FMCAD 2010)
## Experiments with Automata Synthesis

| No | Example     | MONA (ms) | Syn (ms) | |A| | |A'| | 512b | 1024b | 2048b | 4096b |
|----|-------------|-----------|---------|---|---|---|---|---|---|---|---|---|
| 1  | addition    | 318       | 132     | 4 | 9 | 509 | 995 | 1967 | 3978 |
| 2  | approx      | 719       | 670     | 27 | 35 | 470 | 932 | 1821 | 3641 |
| 3  | company     | 8291      | 1306    | 58 | 177 | 608 | 1312 | 2391 | 4930 |
| 4  | parity      | 346       | 108     | 4 | 5 | 336 | 670 | 1310 | 2572 |
| 5  | mod-6       | 341       | 242     | 23 | 27 | 460 | 917 | 1765 | 3567 |
| 6  | 3-weights-min | 26963 | 640     | 22 | 13 | 438 | 875 | 1688 | 3391 |
| 7  | 4-weights   | 2707      | 1537    | 55 | 19 | 458 | 903 | 1781 | 3605 |
| 8  | smooth-4b   | 51578     | 1950    | 1781 | 955 | 637 | 1271 | 2505 | 4942 |
| 9  | smooth-f-2b | 569       | 331     | 73 | 67 | 531 | 989 | 1990 | 3905 |
| 10 | smooth-b-2b | 569       | 1241    | 73 | 342 | 169 | 347 | 628 | 1304 |
| 11 | 6-3n+1      | 834       | 1007    | 233 | 79 | 556 | 953 | 1882 | 4022 |

In 3 seconds solve constraint, minimizing the output; Inputs and outputs are of order $2^{4000}$.
Synthesis for (multi)sets (BAPA)

```python
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)
```
Decision Procedure for Multisets

Operations and Relations on Multisets:
• Plus: \((m_1 \cup m_2)(e) = m_1(e) + m_2(e)\)
• Subset: \(m_1 \subseteq m_2 \iff \forall e. m_1(e) \leq m_2(e)\)
• \(\forall e. F(m_1(e), \ldots, m_k(e)),\)
  \(F - \text{linear integer arithmetic formula}\)
• Cardinality: \(|m| = \sum_{e \in E} m(e)\)
• arbitrary use of \& \&, ||, !, +, -, \leq

Given an expression with these operations, is there an algorithm that finds a value for which expression is \textbf{true}?
\(\rightarrow\) synthesize code from such specifications

Previously: algorithms \textsc{NEXPTIME} or worse

Result: an \textsc{NP}-complete (problem is \textsc{NP} complete)

\((Piskac, Kuncak: VMCAI’08, CSL’08, CAV’08, VMCAI’10)\)
Implicit Programming Agenda

Advancing

3) • Development within an IDE

2) • Compilation and static checking  
√

1) • Execution on a (virtual) machine  
√

Automated reasoning is key enabling technology

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

iadd 0
iconst 1
iadd
Type-Driven Synthesis within an IDE

class Main {
  def main(args: Array[String]) {
    var outputStream: CharArrayWriter = new CharArrayWriter()
    var s: String = "This is a test."
    for (i <- 0 until s.length())
      outputStream.write(s.charAt(i));
    var inStream: CharArrayReader =
""
Type-Driven Synthesis within an IDE

class Main {
    def main(args: Array[String]) {
        var outStream: CharArrayWriter = new CharArrayWriter()
        var s: String = "This is a test."
        for (i <- 0 until s.length())
            outStream.write(s.charAt(i));
        var inStream: CharArrayReader
        = new CharArrayReader(outStream.toCharArray())
        new CharArrayReader(new CharArrayWriter().toCharArray())
        new CharArrayReader(new CharArrayWriter(outStream.size()).toCharArray())
        new CharArrayReader(new CharArrayWriter(new CharArrayWriter().size()))
        new CharArrayReader(new CharArrayWriter(new CharArrayWriter().size()))
Synthesis Approach Outline

1. Source code in editor
   - Program point (cursor)
   - Pre-computed weights

2. Extract:
   - Visible symbols
   - Expected type

3. Create type environment:
   - Encode subtypes
   - Assign initial weights

4. Search algorithm with weights
   (generates intermediate expressions and their types)

5. Ranking

5 suggested expressions

w/ Tihomir Gvero and Ruzica Piskac, CAV’11
Search Algorithm

• Semi-decidable problem
• Related to proof search in intuitionistic logic
• Weights are an important additional aspect:
  – in our application we find many solutions quickly
  – must select interesting ones
  – interesting = small weight
  – algorithm with weights preserves completeness
• Decidable (even polynomial) cases, in the absence of generics and subtyping
Assign weights to expressions and their types

Weight of a term is computed based on
- precomputed weights of API functions
  • determined by mining frequencies from a corpus of Scala code
- proximity to the program point where the tool is invoked
### Result Highlights

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Length</th>
<th>#Initial</th>
<th>#Derived</th>
<th>#Snip.Gen.</th>
<th>Rank</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteArrayInputStreambytebufintoffsetintlength</td>
<td>4</td>
<td>22</td>
<td>4049</td>
<td>102</td>
<td>3</td>
<td>546</td>
</tr>
<tr>
<td>CharArrayReadercharbuf</td>
<td>3</td>
<td>26</td>
<td>782</td>
<td>343</td>
<td>1</td>
<td>546</td>
</tr>
<tr>
<td>HashSetiterator</td>
<td>2</td>
<td>60</td>
<td>1832</td>
<td>201</td>
<td>1</td>
<td>546</td>
</tr>
<tr>
<td>Hashtableelements</td>
<td>2</td>
<td>32</td>
<td>869</td>
<td>445</td>
<td>1</td>
<td>546</td>
</tr>
<tr>
<td>HashtableentrySet</td>
<td>2</td>
<td>31</td>
<td>874</td>
<td>441</td>
<td>1</td>
<td>546</td>
</tr>
<tr>
<td>HashtablekeySet</td>
<td>2</td>
<td>32</td>
<td>968</td>
<td>492</td>
<td>3</td>
<td>546</td>
</tr>
<tr>
<td>Hashtablekeys</td>
<td>2</td>
<td>30</td>
<td>818</td>
<td>477</td>
<td>2</td>
<td>515</td>
</tr>
<tr>
<td>PriorityQueuepoll</td>
<td>2</td>
<td>27</td>
<td>1208</td>
<td>363</td>
<td>1</td>
<td>562</td>
</tr>
</tbody>
</table>

Examples demonstrating API usage
Remove 1-2 lines; ask tool to synthesize it
Time budget for search: 0.5 seconds
Generates upto 1000s of well-typed expressions that compile at this point
Displays top 5 of them
The one that was removed appeared as rank 1, 2, or 3
Similarly good behavior in over 50% of the 120 examples evaluated
Implicit Programming Agenda

Advancing

3) • **Development** within an IDE

2) • **Compilation** and static checking

1) • **Execution** on a (virtual) machine

Automated reasoning is key enabling technology
**Implicit Programming, Explicit Design**

Explicit = written down, machine readable  
Implicit = omitted, to be (re)discovered  

**Current practice:**  
- explicit program code  
- implicit design (key invariants, properties, contracts)  

A lot of hard work exists on verification and analysis: checking design against given code and recovering (reverse engineering) implicit invariants.

**Goal:**  
- explicit design  
- implicit program  

Total work of developer is not increased. Moreover:  
- can be decreased for certain types of specifications  
- confidence in correctness higher – program is spec
Theoretical results and algorithms


More in Ruzica Piskac’s public thesis defense on December 6
Joint work with

Ruzica Piskac  Philippe Suter  Tihomir Gvero  Mikaël Mayer  Ali Sinan Köksal
EPFL

Barbara Jobstmann  Jad Hamza
Verimag  ENS Cachan

Swen Jacobs, EPFL
Thanks also to group members

Eva Darulova
trustworthy numerical computation
OOPSLA’11

Hossein Hojjat
model checking concurrent programs

Etienne Kneuss
static analysis for PHP, Scala RV 2011, FSE 2011

Giuliano Losa
verifying distributed algorithms (w/ Rachid Guerraoui)

Andrej Spielmann
automated synthesis of algorithms for storage (w/ Christoph Koch)
and to Programming Methods Lab

Prof. Martin Odersky

Adriaan Moors
Philipp Haller
Miguel Garcia

Ingo Maier
Hubert Plociniczak
Aleksandar Prokopec
Tiark Rompf
Lukas Rytz
Conclusions

Advancing

3) Development within an IDE
synthesize entire expressions

2) Compilation and static checking
transform spec into program

1) Execution on a (virtual) machine
generalize CLP(X), use SMT solvers

http://lara.epfl.ch