Synthesizing
Computable Functions from Relations

Viktor Kuncak
EPFL
Laboratory for Automated Reasoning and Analysis

http://lara.epfl.ch  http://leon.epfl.ch
How to automatically transform a spec into an implementation?
Given a list of numbers, make this list sorted (wish)

```python
def sort_spec(input : List, output : List) : Boolean = 
content(output)==content(input)   &&  isSorted(output)
```

Specification (for us) is a program that checks, for a given input, whether the given output is acceptable
http://leon.epfl.ch

Etienne Kneuss
(to blame if demo breaks)
Tasks of Interest (i: input, o: output)

a) Check assertion while program runs: \( C(i, p(i)) \)

b) Verify whether program always meets the spec: \( \forall i. C(i, p(i)) \)

c) Constraint programming: once \( i \) is known, find \( o \) to satisfy a given constraint: find \( o \) such that \( C(i, o) \)

d) Synthesis: solve \( C \) symbolically to obtain program \( p \) that is correct by construction, for all inputs: find \( p \) such that \( \forall i. C(i, p(i)) \) i.e. \( p \subseteq C \)
Runtime Assertion Checking

a) Check assertion while program $p$ runs: $C(i,p(i))$

```scala
def p(i : List) : List = {
  sort i using a sorting algorithm and return the result
} ensuring (o ⇒ content(i)==content(o) && isSorted(o))
```

```scala
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, Cons(y, ys)) ⇒
    x < y && isSorted(Cons(y, ys))
}
```

Already works in Scala!

Key design decision:
- constraints are programs

Runtime checks are expensive
⇒ memoization of checks

Runtime Verification’14

Emmanouil Koukoutos
b) Verify that program always meets spec: $\forall i. C(i, p(i))$

```scala
def p(i : List) : List = {
  // sort i using a sorting algorithm and return the result
  \}

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, Cons(y, ys)) ⇒
    x < y && isSorted(Cons(y, ys))
}
```

Type in a Scala program and spec, see it verified

input $i$ such that $\not C(i, p(i))$

proof of $\forall i. C(i, p(i))$

timeout
def sortedIns(e: Int, l: List): List = {
  require(isSorted(l))
  l match {
    case Nil() => Cons(e, Nil())
    case Cons(x, xs) =>
      if (x <= e) Cons(x, sortedIns(e, xs)) else Cons(e, l)
  }
} ensuring(res => contents(res) == contents(l) ++ Set(e)
  && isSorted(res)
  && size(res) == size(l) + 1
} /* Insertion sort yields a sorted list of
  same size and content as the input list */

def sort(l: List): List = (l match {
  case Nil() => Nil()
  case Cons(x, xs) => sortedIns(x, sort(xs))
}) ensuring(res => contents(res) == contents(l)
  && isSorted(res)
  && size(res) == size(l))
Reported Counterexample in Case of a Bug

```haskell
def sortedIns(e: Int, l: List): List = {
    require(isSorted(l))
    1 match {
        case Nil() => Cons(e, Nil())
        case Cons(x, xs) => Cons(x, sortedIns(e, xs))
    }
}
```

Verification

Leon verifies the validity all the verification conditions found in the selected function.

Invalid!

<table>
<thead>
<tr>
<th>Function</th>
<th>Kind</th>
<th>Result</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sortedIns</td>
<td>precond.</td>
<td>✓ valid</td>
<td>0.012</td>
</tr>
<tr>
<td>sortedIns</td>
<td>postcond.</td>
<td>✗ invalid</td>
<td>0.051</td>
</tr>
</tbody>
</table>

The following valuation violates the VC:

```
l := Cons(8946, Nil())
e := 8945
```
Approaches and Their Guarantees

- **a)** Check assertion while program \( p \) runs: \( C(i, p(i)) \)
- **b)** Verify that program always meets spec: \( \forall i. C(i, p(i)) \)
- **c)** Constraint programming
  once \( i \) is known, find \( o \) to satisfy a given constraint:
  \( \text{find } o \text{ such that } C(i, o) \) run-time
- **d)** Synthesis: solve \( C \) symbolically to obtain program \( p \) that is correct by construction, for all inputs:
  \( \forall i. C(i, p(i)) \) \( \text{ i.e. } \ p \subseteq C \) compile-time
def insert(x : Int, t : Tree) = \textbf{choose}((t1:Tree) =>
   isRBT(t1) && content(t1) = content(t) ++ Set(x))

Remarks:
• no more effort than implementation - wrote functions for invs.
• these invariants is what drives data structure design
• this is how things are explained in a textbook
• using properties promotes reuse - combine them using \&\
• functional programming alone favors \textit{deterministic} complete implementations and does \textit{not} automate process \ inv\ invariant -> code
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->left == nil) ? y->right : y->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 */
            Assert( (y!=tree->nil && y->left), "y is nil in RBDelete\n");
        } else {
            if (!y->left) {
                tree->DestroyKey(y->key);
                tree->DestroyInfo(y->info);
                (!y->red) RBDeleteFixUp(tree,y);
                free(y);
            } else {
                tree->DestroyKey(y->key);
                tree->DestroyInfo(y->info);
                (!y->red) RBDeleteFixUp(tree,y);
                free(y);
            } #ifdef DEBUG_ASSERT
            Assert(!tree->nil, "nil not black in RBDelete");
        } #endif
    } #ifdef DEBUG_ASSERT
    Assert(!tree->nil, "y is nil in RBDelete\n");
    #endif
}
remove using specifications: 2 lines

def remove(x : Int, t : Tree) = choose((t1:Tree) =>
isRBT(t1) && content(t1)=content(t) – Set(x))

The biggest expected payoff:

properties are more reusable
Using Verifier to Solve Constraints

**Goal:** find a value that satisfies a given constraint: find $o$ such that $C(i,o)$ (once $i$ is known)

**Method:** use verification technology: try to prove that no such $o$ exists, report counter-examples!

$C(i,o)$

$\forall o. \neg P_i(o)$

Verifier

Counterexample $o$

$o$ is solution to $C(i,o)$
Approaches and Their Guarantees

both specification $C$ and program $p$ are given:

a) Check assertion while program $p$ runs: $C(i, p(i))$

b) Verify that program always meets spec: $\forall i. C(i, p(i))$

donly specification $C$ is given:

c) Constraint programming: once $i$ is known, find $o$ to satisfy a given constraint: find $o$ such that $C(i, o)$

d) Synthesis: solve $C$ symbolically to obtain program $p$ that is correct by construction, for all inputs: find $p$ such that $\forall i. C(i, p(i))$ i.e. $p \subseteq C$

compile-time

run-time
Synthesis for Theories

\[ 3i + 2o = 13 \quad \Rightarrow \quad o = \frac{(13 - 3i)}{2} \]

- Wanted: "equation solving" for programs
  - for linear integer equations: extended Euclid’s algorithm
  - need to handle disjunctions, negations, more data types
- For every formula in e.g. Presburger arithmetic
  - synthesis algorithm terminates
  - produces the most general precondition
    (assertion characterizing when the result exists)
  - generated code always terminates and gives correct result
- If there are multiple or no solutions for some input parameters, the algorithm identifies those inputs
- Works not only for arithmetic but also for e.g. sets with sizes and for trees
- Goal: lift everything done for SMT solvers to synthesizers
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)

Kuncak, Mayer, Piskac, Suter: PLDI'10, STTT’12, CACM'12
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;
        }
    } else {
        days -= 365;
        year += 1;
    }
    ...
}
```

Enter December 31, 2008
All music players (of this brand) froze in the boot sequence.

Synthesis can generate terminating and correct solution from a spec.
Deduction Rules within Framework

ADTs
- Unification
- ADT Split
- Induction
- Long Induction
- Inner Case Split
- ADT Dual
- Disunification

Integers
- Unused Input
- Condition Abduction
- Unconstrained Output
- CEGIS
- Case Split
- Equality Split
- One Point
- De-tuple Input
- Ground
- Assert
- Integer Equation
- Integer Induction
- Integer Inequalities
- Inequality Split

CEGIS
Recursion Schemas + STE in Action

```scala
def delete(in1: List, v: Int) = choose {
  (out: List) => content(out) == content(in1) -- Set(v)
}

def delete(in1: List, v: Int) = {
  def rec(in: List, v: Int): List = in match {
    case Cons(h,t) =>
      val r = rec(t,v)
      if (h == v) {
        CEGIS r
      } else {
        CEGIS Cons(h, r)
      }
    case Nil =>
      CEGIS Nil
  } ensuring { content(_) == content(in1) -- Set(v) }
  rec(in1, v)
}
```
Synthesis of Data Structure Manipulation

Techniques used:

- symbolic exploration of the space of programs
- synthesis based on type inhabitation
- fast falsification by executing candidates on test inputs
- Leon’s verification capabilities
- synthesis for theory of trees and integers
- recursion schemas
- case splitting
- learning conditional expressions
- cost-based search over possible synthesis steps

OOPSLA 2013
Synthesis and Constraint Solving

If we did not find an expression that solves it in all cases, we emit a runtime call to solver

Result: solver invoked only in some cases
  – for some components of result
  – for some conditions on inputs

after timeout, close the remaining branches by inserting a runtime solver call
Recent work: Automated Repair

```scala
def desugar(e : Trees.Expr) : SimpleE = { e match {
    case Trees.Plus (lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
    case Trees.Minus(lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
    case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
    case Trees.And (lhs, rhs) => Ite(desugar(lhs), desugar(rhs), Literal(1))
    case Trees.Or (lhs, rhs) => Ite(desugar(lhs), Literal(1), desugar(rhs))
    case Trees.Not(e) => Ite(desugar(e), Literal(0), Literal(1))
    case Trees.Eq(lhs, rhs) => Eq(desugar(lhs), desugar(rhs))
    case Trees.Ite(cond, thn, els) => Ite(desugar(cond), desugar(thn), desugar(els))
    case Trees.IntLiteral(v)  => Literal(v)
    case Trees.BoolLiteral(b) => Literal(b2i(b))
} } ensuring { res =>
    sem(res) == Semantics.semUntyped(e)
}
```

Equivalence-preserving transformation of a program into simpler form
Recent work: Automated Repair

```scala
def desugar(e : Trees.Expr) : SimpleE = { e match {
  case Trees.Plus (lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
  case Trees.Minus(lhs, rhs) => Plus(desugar(lhs), Neg(desugar(rhs)))
  case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
  case Trees.And (lhs, rhs) => Ite(desugar(lhs), desugar(rhs), Literal(1))
  case Trees.Or (lhs, rhs) => Ite(desugar(lhs), Literal(1), desugar(rhs))
  case Trees.Not(e) => Ite(desugar(e), Literal(0), Literal(1))
  case Trees.Eq(lhs, rhs) => Eq(desugar(lhs), desugar(rhs))
  case Trees.Ite(cond, thn, els) => Ite(desugar(cond), desugar(thn), desugar(els))
  case Trees.IntLiteral(v) => Literal(v)
  case Trees.BoolLiteral(b) => Literal(b2i(b))
} } ensuring { res =>
  sem(res) == Semantics.semUntyped(e)
}
```

Equivalence-preserving transformation of a program into simpler form
Recent work: Automated Repair

```scala
def desugar(e : Trees.Expr) : SimpleE = { e match {
  case Trees.Plus (lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
  case Trees.Minus(lhs, rhs) => Plus(desugar(lhs), Neg(desugar(rhs)))
  case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
  case Trees.And (lhs, rhs) => Ite(desugar(lhs), desugar(rhs), Literal(0))
  case Trees.Or (lhs, rhs) => Ite(desugar(lhs), Literal(1), desugar(rhs))
  case Trees.Not(e) => Ite(desugar(e), Literal(0), Literal(1))
  case Trees.Eq(lhs, rhs) => Eq(desugar(lhs), desugar(rhs))
  case Trees.Ite(cond, thn, els) => Ite(desugar(cond), desugar(thn), desugar(els))
  case Trees.IntLiteral(v)  => Literal(v)
  case Trees.BoolLiteral(b) => Literal(b2i(b))
} } ensuring { res =>
  sem(res) == Semantics.semUntyped(e)
}
```

After automated repair, it verifies with respect to a very strong spec

Equivalence-preserving transformation of a program into simpler form
Further Contributors to Leon

Regis Blanc
EPFL

Philippe Suter
IBM Research
(Leon founder)

Ali Sinan Koeksal
Berkeley

Ivan Kuraj
MIT

Eva Darulova

Ravichandran Madhavan

Nicolas Voirol
Synthesis of Invariants

Given templates, find inductive invariants in Leon

Combination of
  – Farkas lemma for encoding forall-exists into exists for non-linear problems
  – Encoding of algebraic data types
  – Incremental, on-demand handling of disjuncts and recursive functions
  – Specialized techniques for resource bounds (such as sequential and parallel execution time)

CAV’14
Synthesis of Real Computation in Real Software

def turbine(v: Real, w: Real, r: Real): Real = {
  3 + 2/(r*r) - 0.125*(3-2*v)*(w*w*r*r)/(1-v) - 4.5
}

def turbine(v: Double, w: Double, r: Double): Double = {
  3 + 2/(r*r) -
  0.125*(3-2*v)*(w*w*r*r)/(1-v) - 4.5
}

def turbine(v: Int, w: Int, r: Int): Int = {
  val tmp1 = (r * r) >> 15
  val tmp2 = (16384 << 11) / tmp1
  val tmp3 = ((24576 << 2) + tmp2) >> 2
  val tmp4 = (16384 * v) >> 14
  val tmp5 = (24576 - (tmp4 << 2)) >> 2
  val tmp6 = (16384 * tmp5) >> 14
  val tmp7 = (w * w) >> 15
  val tmp8 = (tmp7 * r) >> 15
  val tmp9 = (tmp8 * r) >> 15
  val tmp10 = (tmp6 * tmp9) >> 15
  val tmp11 = (16384 - (v << 2)) >> 2
  val tmp12 = (tmp10 << 14) / tmp11
  val tmp13 = (tmp3 - (tmp12 << 3)) >> 2
  val tmp14 = ((tmp13 << 1) - 18432) >> 2
  tmp14
}

• affine arithmetic
• non-linear solvers
• optimization
• symbolic algebra

POPL 2014

Eva Darulova
Your wish is my command

Agenda:

- requirement formalization
- specification relation (constraint): C
- implementation function (program): p
- conventional compilation

Command

11011001 01011101
11011001 01011101
11011001 01011101
11011001 01011101
Synthesizing Code from Free-Form Queries

public class Utils {
    public void backupFile(String fname) {
        String bname = fname + ".bak";
        copy file fname to bname
        FileUtils.copyFile(new File(fname), new File(bname))
        FileUtils.copyFile(new File(bname), new File(fname))
        FileUtils.copyFileToDirectory(new File(fname), new File(bname))
        FileUtils.copyFileToDirectory(new File(bname), new File(fname))
        FileUtils.copyFile(<arg>, new File(fname))
    }
}

Tihomir Gvero
wish

"Your wish my command" agenda

requirement formalization
( also exploring )

specification ( constraint ) : C

implementation ( program ) : p

conventional compilation

synthesis in Leon

Command
11011001 01011101
11011001 01011101
11011001 01011101
11011001 01011101