Inferring Resource Bounds on Functional Scala Programs

Pratik Fegade
with
Viktor Kuncak and Ravi Madhavan

LARA, EPFL

July 10, 2015

Outline

Orb: An Introduction

An Overview

- A tool for proving specifications on resource bounds for Leon programs
- Resources can be specified as templates with numerical holes. For example,

$$\mathit{time} \;\leq ? * \mathit{size}(l) + ?$$

- Orb infers values for the numerical holes

Our Running Example

List distinct

```
def distinct(l: List): List = (
  l match {
    case Nil() => Nil()
    case Cons(x, xs) => {
      val newl = distinct(xs)
      if (contains(newl, x)) newl
      else Cons(x, newl)
  }
})
```

Writing Templates and Correctness Invariants in Orb

```
def distinct(l: List): List = {
  l match {
    case Nil() => Nil()
    case Cons(x, xs) => {
      val newl = distinct(xs)
      if (contains(newl, x)) newl
      else Cons(x, newl)
    }
  }
} ensuring(res => size(l) >= size(res) &&
    time <= ? * size(l) * size(l) + ?)
```

Inferring Holes

Instrumentation

```
def distinct(l: List): = {
  l match {
    case Nil() => (Nil(), 3)
    case Cons(head, tail) =>
      val e1 = distinct(tail)
      val e2 = contains(e1._1, head)
      val r1 =
        if (e2._1) (e1._1, 2 + e2._2)
        else (Cons(head, e54), 3 + e2._2)

      (r1._1, (7 + r1._2 + e1._2))
  }
} ensuring(res => size(l) >= size(res._1) &&
    res._2 <= ? * size(l) * size(l) + ?)
```

Inferring Holes

- The bounds are invariants on the instrumented program
- Orb constructs a $\exists\,\forall$ problem for inferring the numerical holes in the invariants.
- Solve them using a incremental counter-example driven algorithm

Inferring Stack Bounds

Instrumentation for Stack Bounds

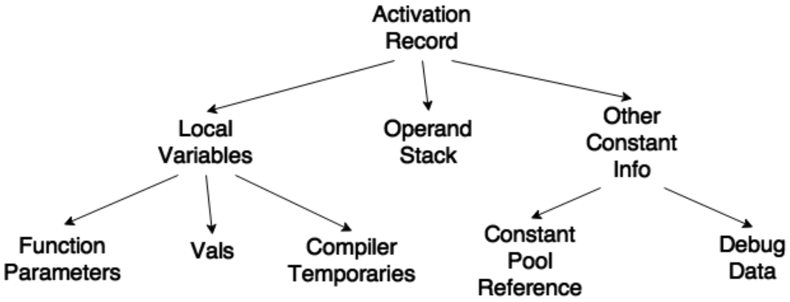
- We support inference of peak stack memory usage
- Somewhat different compared to time:
  - Memory is reusable unlike time. Thus,

$$stack(e1\ op\ e2) = a + max(stack(e1), stack(e2))$$

- Independent of the underlying JVM used as we work at the bytecode level
- Need to precise model the memory consumed by the activation record of a function

Estimation of Activation Record Size

The Structure of the Activation Frame (as per JVM specifications)



The Operand Stack

Consider the evaluation of the expression $g(f(x, y), z)$. The operand stack goes through the following stages

- $\$|x$
- $\$|x|y$
- $\$|x|y|f$
- $\$|r1$ $\qquad r1 = f(x, y)$
- $\$|r1|z$
- $\$|r1|z|g$
- $\$|r2$ $\qquad r2 = g(r1, z)$

Here the maximum stack size used is 3 units and the result finally appears on the stack

Compiler Temporaries

```
l match {
  case Nil() => ...
  case Cons(_, _) => ...
}
```

```
 0: aload_1
 1: astore_2                 // Temporary created
 2: aload_2
 3: instanceof      #16      // class Nil()
 6: ifeq            20
 ...
20: aload_2
21: instanceof      #27      // class Cons(_,_)
24: ifeq            59
```

Instrumentation for Stack

```
def distinct(l : List) = {
  val bd =
    l match {
      case Nil() => (Nil(), 1)
      case Cons(head, tail) =>
        val e1 = distinct(tail)
        val e2 = contains(e1._1, head)
        val r1 =
          if (e2._1) (e1._1, e2._2)
          else (Cons(head, e1._1), e2._2)

        (r1._1, max(r1._2, e1._2))
    }
  (bd._1, bd._2 + 17)
} ensuring (res => size(l) >= size(res._1) &&
    res._2 <= ?*size(l) + ?)
```

Inferring Many Bounds at Once

Combining Instrumentations

- Functions can be instrumented for multiple resources
- $e \xrightarrow{time\&stack} (e, time(e), stack(e))$
- Resources can be interdependent. For example, ‘tpr’ - time-per-recursive-step
- We have a flexible framework for instrumenting programs in Leon
  - Performs many simplifications to keep the instrumented programs readable and simple

Combined Instrumentation

Stack and Time

```
def distinct(l : List) = {
  val bd = l match {
    case Nil() => (Nil(), 1, 3)
    case Cons(head, tail) =>
      val e1 = distinct(tail)
      val e2 = contains(e1._1, head)
      val r1 =
        if (e2._1) (e1._1, e2._2, 2 + e2._3)
        else (Cons(head, e1._1), e2._2, 3 + e2._3)

      (r1._1,
        max(e1._2, r1._2),
        7 + r1._3 + e1._3)
  }
  (bd._1, bd._2 + 17, bd._3)
}
```

Instrumentation	Instrumentation Variable
Execution Time	time
Depth (a measure of the inherent parallelism in the implementation)	depth
Number of recursive calls made (including mutually recursive calls)	rec
Stack size	stack
Time per one (mutually) recursive call	tpr

Table: Resources/Program Characteristics Supported

Compositional Reasoning for Time bounds

Existing Approach Towards Non-linearity

- Solving time bounds like $time <= ? * size(l) * size(l) + ?$
- To get around this, we may treat multiplication as an UF and instantiate axioms for it
- But, we can decompose reasoning about nonlinear time bounds in some cases

Intuition

```
for (var i = 0; i < n; i++)
  for (var j = 0; j < m; j++)
    {...}
```

- The total execution time can be given by the product of the number of recursions of the outer loop and an upper bound on the execution time of its body
- This (loosely) translates to recursive functions in the functional world

From Loops to Recursion

Consider the following code snippet

```
def foo(i: Int, n: Int, m: Int)  = {
   if(i < n) {
      bar(0, m)
      foo(i + 1, n, m)
   } else 0
} ensuring(res => time <= ? * (n - i) * m + ? * m + ?)

def bar(j: Int, m: Int) = {
   if(j < m) {
      bar(j + 1, m)
   } else 0
} ensuring(res => time <= ? * (m - j) + ?)
```

From Loops to Recursion

bar(j, m) : Count up from j to m
foo(i, n, m) : Count up from i to n with *bar*$(0, m)$ in each iteration

- Time of *bar* is linear in $m - j$ (can be easily established by Orb)
- Number of recursions of *foo* is bounded by $n - i$
- Time of *foo* trivially follows from the above facts

From Loops to Recursion

$bar(j, m)$: Count up from j to m
$foo(i, n, m)$: Count up from i to n with $bar(0, m)$ in each iteration

- Time of *bar* is linear in $m - j$ (can be easily established by Orb)
- Number of recursions of *foo* is bounded by $n - i$
- Time of *foo* trivially follows from the above facts
- What if the recursive call in the function *foo* is $foo(i + 1, n, m + 1)$?

Composition of resource bounds

The Setting

- Let $f(\bar{x})$ be a self recursive function with $\bar{x}$ as formal parameters. It has k recursive calls with parameters $p_0(\bar{x})$, $p_1(\bar{x})$, ..., $p_{k-1}(\bar{x})$. Let *res* denote the result of f
- The 'tpr' instrumentation for f can be thought of as a function $g(\bar{x})$
- The user provides us with a parametric upper bound $ug(\bar{x}, res)$ to $g(\bar{x})$
- Suppose a call $f(\bar{y_0})$ leads to n recursive calls with arguments being $\bar{y}_0$, $\bar{y}_1$, ..., $\bar{y}_{n-1}$ and 'tpr's tpr_0, tpr_1, ..., tpr_{n-1}

Composition of resource bounds

VC Generation

$$
\begin{array}{ccccc}
tpr_0 & \leq & g(y_0) & \leq & ug(y_0, res_0) \\
 & & & & \vee\!| \\
\cdot & & \cdot & & \cdot \\
\cdot & & \cdot & & \cdot \\
\cdot & & \cdot & & \cdot \\
 & & & & \vee\!| \\
tpr_{n-2} & \leq & g(y_{n-2}) & \leq & ug(y_{n-2}, res_{n-2}) \\
 & & & & \vee\!| \\
tpr_{n-1} & \leq & g(y_{n-1}) & \leq & ug(y_{n-1}, res_{n-1})
\end{array}
$$

Composition of resource bounds

- By the virtue of instrumentation, $\forall i \; tpr_i \leq g(\bar{y}_i)$
- We can infer the coefficients in *ug* such that $\forall \bar{y} \; g(\bar{y}) \leq ug(\bar{y}, f(\bar{y}))$
- Now, if are able to prove that *ug* monotonically across recursive, calls, i.e. $\forall j \; ug(p_j(\bar{x}), f(p_j(\bar{x}))) \leq ug(\bar{x}, f(\bar{x}))$, then we can bound the total time taken as shown on the next slide

Composition of resource bounds

If $t_{f(\bar{y}_0)}$ is the total time we need to bound, then we have,

$$
\begin{aligned}
t_{f(\bar{y}_0)} &= \sum_{i=0}^{n-1} tpr_i \\
&\leq \sum_{i=0}^{n-1} g(\bar{y}_i) \\
&\leq \sum_{i=0}^{n-1} ug(\bar{y}_i, res_i) \\
&\leq \sum_{i=0}^{n-1} ug(\bar{y}_0, res_0) \\
&= n * ug(\bar{y}_0, res_0)
\end{aligned}
$$

Composition of resource bounds

An Example

```
@compose
def distinct(l: List): List = (
  l match {
    case Nil() => Nil()
    case Cons(x, xs) => {
      val newl = distinct(xs)
      if (contains(newl, x)) newl
      else Cons(x, newl)
  }
}) ensuring (res =>
     time <= ? * size(l) * size(l) + ? &&
     rec  <= size(l) + ? &&
     tpr  <= ? * size(l) + ?)
```

Composition of resource bounds

An Example

We need to as stated before do the following

- Infer the constants in the template for ‘rec’
- Infer the constants in the template for ‘tpr’ in accordance to the following condition $d * size(xs) + e <= d * size(l) + e$
- Infer the constants in the template for time in with the help of the above inferred invariants and the additional axiom $time \leq (rec + 1) * (? * size(l) + ?)$ (We need to add the 1 to rec because by the nature of our instrumentation, $rec = n - 1$)

Results

Results

Composition for Time Bounds

Function	Inferred Inv.	Time
quickSort	$rec \leq 2size(l) \wedge$ $tpr \leq 30size(l) + 5 \wedge$ $time \leq 60size(l)^2 + 40size(l) + 5$	27
insertionSort	$rec \leq size(l) \wedge$ $tpr \leq 8size(l) + 5 \wedge$ $4time \leq 59size(l)^2 + 46$	4
listDistinct	$rec \leq size(l) \wedge$ $tpr \leq 8size(l) + 5 \wedge$ $2time \leq 29size(l)^2 + 24$	10

Results

Stack Bounds

Function	Stack Bound	Time
RedBlackTrees		58
add	$stack \leq 48 blackHeight(t) + 105$	
ins	$stack \leq 48 blackHeight(t) + 93$	
MergeSort		8
length	$stack \leq 14 size(l) + 15$	
merge	$stack \leq 22 size(aList) + 22 size(bList) + 23$	
mergeSort	$stack \leq 28 size(list) + 39$	
split	$stack \leq 23 size(l) + 23$	

Experiments on Stack Usages

- JVM provides an option to set the stack size provided it is above a threshold
- Fill the stack with a filler function with a known activation frame size
- Call function under consideration on the top of this filler
- Use the stack overflow error to estimate frame size

Results

Stack Bounds

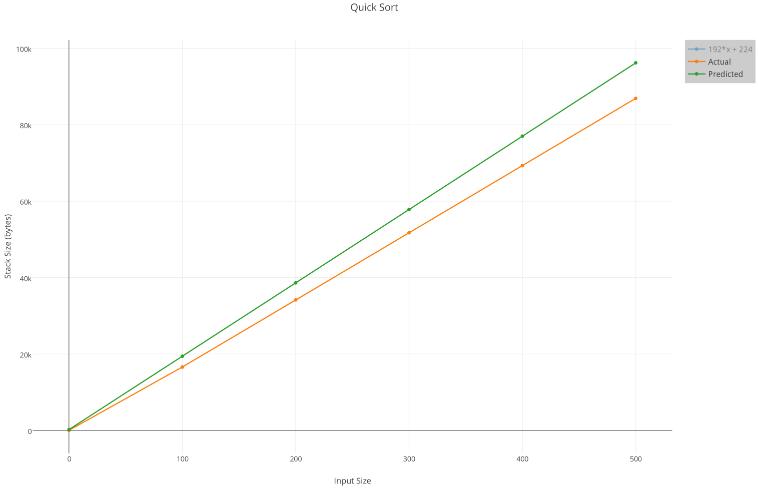


Figure: Quick Sort

Results

Stack Bounds

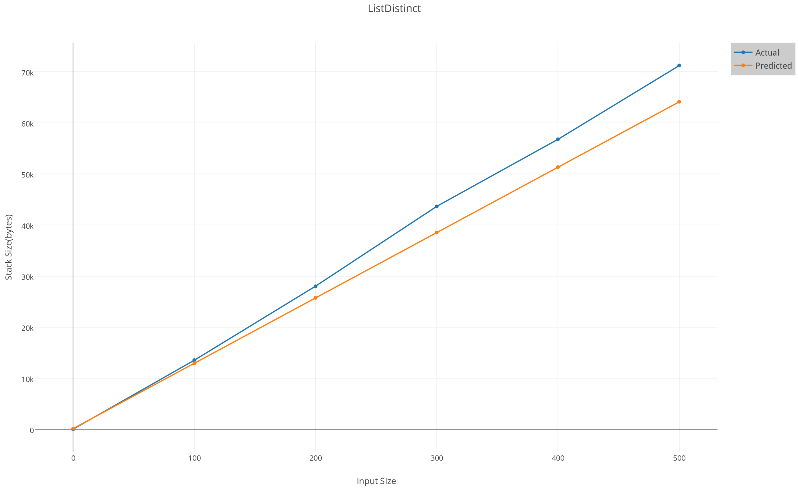


Figure: List Distinct

Conclusions

Possible Future Work

- Extending composition of ‘rec’ and ‘tpr’ to mutually recursive functions
- Extensions to infer bounds on the (peak) heap usage of a program
- Comparing and contrasting the obtained bounds with the actual resources used up by the program and refining the current models

Thank You!