Title of Your Project

Compiler Construction ’17 Final Report

First1 Last1 First2 Last2

KTH Royal Institute of Technology

{name}@kth.se

1. Introduction

We chose to the Tail-call optimization lab. However general Tail-call elemination would be a coxplex task In JVM so we inted to start out by only implementing the sub problem of Tail-recursion optimization.

This allows us to focus solv the problem without any changes to the existing compiler stages. We belive that all we need to do is to Introduce a new compiler stage for AST rewriting, and implement tail recursion elimination as an AST transformation from from a tail recursive method to a method with a whail loop.

This stage will have to be executed after the Type checking faze, as having the method symbols atached for on method call expressions would be helpfull.

2. Examples

Beniffits of tail recursion optimizations is the most obvious on method ecixutions that would otherwise run out of stack space. Considder for example the following program.

It recursevly incremetst i untill i overflows to 0 and returns the result.

```
class OverflowPlease {
  def makeZero( i:Int):Int = {
    if (i==0){
      i
    }else{
      this.makeZero(i+1)
    }
  }
}

object Main extends App {
  println(new OverflowPlease().makeZero(0))
}
```

each recursion creates a new stack fram, so a stack that fits over 4 million frames would be requiered to execute this program. But with tailrecursion optimization we expect this program to Sucessfully overflow i to 0 only using a couple of stack frams.

This should also make the programs that do not run out of stack space faster since poping and pushing stack frames is a relatively slow procidure.

here is a similar example from the lab descrption that recursively get the last element of a linked list

```
class Helper {
  def last(a: List): List = {
    if (a.hasNext()) {
      this.last(a.next())
    } else {
      a
    }
  }
}
```

2.1 Recursion on trees

Recursion on tree structures is interesting because in binary tries we need to recurse on the left and right subtrees, but only the one of the recursive calls may be in the tail position. Still removing one of the recursive calls is better then non, especially if the tree tends to be unbalanced.

this example has a function that tail recursively sums the contents of the tree.

and enother one that sets all the contents of the tree to 0.

```
class TreeHelper {
  def countNodes(a: Tree, acc: Int): List = {
    val r:Tree = NULL;
    val l:Tree = NULL;
    val sum:Int = acc;
    if(a==NULL){
        sum
    }else{
```

```
      r=a.getRight();
      l=a.getLeft();
      sum = this.countNodes(r, acc) + sum + 1;
      this.countNodes(l, sum)
    }
  }

  def cleanseTree(a: Tree): List = {
    val r:Tree = NULL;
    val l:Tree = NULL;
    if(!(a==NULL)){
      a.setValue(0);
      r=a.getRight();
      l=a.getLeft();
      this.cleanseTree(r);
      this.cleanseTree(l)
    }
  }
}
```

2.2 corecursive tail calls

Here is toy example of the more general problem that we do not inted to handle. Here two methods `isEven(Int)` and `isOdd(Int)` use tail calls to each other to determin if the input method is odd or even.

```
class numbers {
  def isEven(i:Int):Boolean = {
    if(i==0) True
    else isOdd(i−1)
  }

  def isOdd(i:Int):Boolean = {
    if(i==1) True
    else isEven(i−1)
  }

}
```

3. Implementation

This is a very important section, you explain to us how you made it work.

3.1 Theoretical Background

If you are using theoretical concepts, explain them first in this subsection. Even if they come from the course (e.g., lattices), try to explain the essential points *in your own words*. Cite any reference work you used like this [Appel 2002]. This should convince us that you know the theory behind what you coded.

3.2 Implementation Details

Describe all non-obvious tricks you used. Tell us what you thought was hard and why. If it took you time to figure out the solution to a problem, it probably means it wasn't easy and you should definitely describe the solution in detail here. If you used what you think is a cool algorithm for some problem, tell us. Do not however spend time describing trivial things (we know what a tree traversal is, for instance).

After reading this section, we should be convinced that you knew what you were doing when you wrote your extension, and that you put some extra consideration for the harder parts.

4. Possible Extensions

Handling corecusvine tail calls like in the even odd example in Section2.2 would be the obvious next step.

The benifit is that we would get improved performance and avoud some stack overflows in some co recursive programs.

But this extension is allot more chalaning. The typical aproach for call recursion elemination of simply jumping to the tail called method insted of poping and pushing stack frames would not be possible since every method in JVM has its own private address space.

One possible apreach to eliminate corecursive tail calls that would build well on work in this project would be to implement a AST transformation that rewrites corecursive methods into a single big method that can simulate all of the methods forming the corerusion. After that our tail recursion elimination AST transformation can be aplied to the AST generated by the method merger.

References

A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.