Tail Recursion Elimination

Compiler Construction '17 Final Report

Daniil Pintjuk   Kim Björk
KTH Royal Institute of Technology
{pintjuk, kimbjork}@kth.se

1. Introduction

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

This allows us to focus on solving 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 a tail recursive method to a method with a wile loop.

This stage will have to be executed after the type checking phase, seing as having the method symbols attached to method call expressions would be helpfull.

2. Examples

The benefits of tail recursion optimizations could be viewed clearly on method exicutions that would otherwise run out of stack space. Consider for example the following program.

It recursevly increments i until 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 frame, 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 frames.

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 procedure.

Here is a similar example from the lab description 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
    }
  }
}
```

Recursion on tree structures is interesting because in binary trees 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 none, especially if the tree tends to be unbalanced.

This example has a function that tail recursively sums the contents of the tree and another 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)
    }
  }
}
```

\subsection co−recursive fuctions

Here is toy example of a tail calls that we **do** not intend to optimize

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

If you did not finish what you had planned, explain here what's missing.

In any case, describe how you could further extend your compiler in the direction you chose. This section should convince us that you understand the challenges of writing a good compiler for high-level programming languages.

References

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