Tail Call Elimination for Amy

Computer Language Design '18 Final Report

Jaron Maene
EPFL
jaron.maene@epfl.ch

1. Introduction

In this project, I implemented a simple compiler for the Amy language. Amy is a simple toy language based on Scala, it is statically typed and purely functional (besides the Std library and error). The compiler was implemented step-by-step in a pipelined fashion:

1. **Lexer**. The lexer converts the input text files into a stream of tokens. Comments and whitespace are disregarded.
2. **Parser**. Using an LL1 grammar, the tokenstream gets parsed into an AST. The main complexity in this phase are the operator precendence rules and choosing the appropriate associativity.
3. **Name-analyzer**. The analyzer has three main purposes. Firstly it makes sure that the naming rules are being adhered to correctly, asserting that e.g. there are no undefined variables are used. Secondly it assigns unique names to all variables and functions, so that there can be no confusion when resolving a variable in later phases. Lastly it populates the symbol table.
4. **Type-checker**. A simplified Hindley-Milner type-inference algorithm checks is used to check that all typing rules are observed.
5. **Code-generation**. The backend of the compiler converts the AST into wasm bytecode, which can be executed with nodejs.

To keep this project manageble Amy has some obvious lacking features that usually are included in functional languages. Some that come to mind are the lack of any memory management, or a kind of higher-order functions. My specific project focusses on a third one: tail call elimination (TCE).

Like any functional language, Amy only provides support for iterations through recursion. However this warrants some kind of TCE, indeed the lack of this whould prevent including infinite loops in your Amy programs.

My extention to the amy compiler not only optimizes tail recursion, but also modifies the code-generation to include trampolines, allowing general TCE.

2. Examples

As mentioned before there are two kinds of TCE being performed. Trampolines, which are more generally useable, and Tail recursion which is more performant, but only useable for recusive tail calls.

For example: the following function whould be optimized with trampolines:

```
object MutualRecursion {
  def odd(n: Int): Boolean = {
    if (n==0) {
      false
    } else {
      even(n−1)
    }
  }

  def even(n: Int): Boolean = {
    if (n==0) {
      true
    } else {
      odd(n−1)
    }
  }

  Std.printBoolean(odd(1000001))
}
```

While this program could use tail recursion:[1]

```
object InfiniteLoop {
  def loop():Unit = {
    loop()
  }

  loop()
}
```

This leaves the edge case of a program with both kinds of recursion. Here I chose to use trampolines, in order to avoid the complexity of a hybrid approach.

3. Implementation

3.1 Theoretical Background

Tail call recursion addresses the issue of stack usage, mainly in functional languages. Indeed, when using recursion as your primary iteration mechanism, stack space can become excessive. Even simple algorithms like factorial will have a linear space complexity.

TCE abuses the fact that these recursive calls often occur at the end of the function (tail calls). This allows us to release the current stack frame before recursing. This is the essence of tail call elimination: keep reusing the current stack frame when performing tail calls, and as such regain constant memory usage when iterating.

The application of TCE is complicated however, by the choice of our backend. Web Assembly is a kind of stack machine, only allowing for a somewhat higher level interface with the hardware. Because of this it is not possible to just overwrite the stack frame. So altough TCE is on it's way to being included in wasm, it is currently not possible to perform TCE in the naive way.

There many possible solutions to this (cf. [Schinz 2018]). Notably languages like Scala and Clojure have the same problem. They are both functional languages with the JVM as backend. The JVM, much like in wasm, is a type of stack machine that doesn't provide primitives for TCE.[2]

Maybe the simplest approach to TCE on stack based machines are trampolines. Using trampolines, a function does not perform a tailcall itself. Instead, it returns, and the caller performs the call (= the trampoline). This may sound familiar if you're accustomed to continuation-passing style.

Of course this has the downside of increasing some overhead on every tail call. There are no techniques to decrease the amount of overhead spent on trampolines, but none can completely remove it. For this reason, Scala and Clojure still have no general support for TCE. Scala does however optimize recursive tail calls and has a tailcalls library providing some kind of trampolines. (See [Bjarnarson 2012]) Clojure chose to promote the use of imperative iteration to avoid the problem.

Only optimizing recursive tail-calls, like Scala does, is far easier than the general case. The tail-call can be replaced by a jump to the start of the function, after making sure that the parameters are overwritten. (So effectivly a tail-recursive function is compiled as a non-recursive function with a while loop.)

Unlike trampolines, this approach has no real disadvantages in Amy. So when possible it will be applied. In other cases, trampolines will be deployed.

3.2 Implementation Details

The implementation has two facets. First of the tail calls need be properly recognized as such, secondly the code-generation needs to support tailcalls.

3.2.1 Tail call detection

The first part is implemented in a separate compiler phase, Optimization, wedged in between the type-checking and code-generation. Extra case classes in the TreeModule where added to express the different forms of tailcalls: `IndirectCall`, `Trampoline` and `TailCall`. The `TailCall` is used to indicate that a call is a recursive tailcall. The `IndirectCall` is used when a function wants to call a function by returning to a trampoline. Finally the `Trampoline` indicates the placement of a trampoline.

The detection and rewriting of the AST happens by recursivly traversing the AST, in the usual way. Maybe the only interesting case in this phase, is when a function is both tail recursive and has a non-recursive tail call. In this case I opted to use trampolines. If whould however be possible to merge the approaches, but I opted to avoid the complexity of this fairly rare edge case.

[1] Note that this example has the most notable diverging behaviour. Without TCE it whould cause a stack overflow almost instantaneously.

[2] Contrary to wasm however, JVM doesn't have tail call primitives because of technical reasons. For more information see [Clements and Felleisen 2004].

3.2.2 Tail call code generation

The next step was modifying the code generation. For tail-recursion this fairly limited. A loop block encloses all tail-recursive functions, and a `TailCall` puts arguments into the appropriate locals after which it jumps.

The most difficult part of the project was the code generation for trampolines. A Trampoline does not know which function to call at compile time (as it is returned by the `IndirectCall`). So there was a need to implement a kind of dynamic dispatch. In wasm this can be handled by using the `call_indirect` instruction. This pops a value of the stack and does a lookup in a table that maps ints to a function. Next, the found function is called in the usual way.

To generate the function table, functions where assigned an index in the name-analyzer phase (much like for the constructor signatures). Using these, the ModulePrinter makes a table that maps the function, and the code-generator can find the appropriate values in the symboltable as needed.

Before we continue at how the `Trampoline` is handled it might be worth looking at `IndirectCall`. This needs to store all the arguments and which function to call in memory. Notice that we can reuse the tools we developed to store adt's. They both store a variable amount of arguments with some index.

The trampoline can now perform the inverse operation, fetching the arguments from memory and put them on the stack, so the `call_indirect` has access to them. Notice however that this is somewhat more complicated as the called function might not want to call anything (handled by returning a -1 instead of the memory location), furthermore the amount of arguments for the trampoline need not be fixed, so some kind of while loop is needed, deriving the amount of argument to be loaded from the distance of the returned memory location and the global memory pointer.

This leads us to the final issue: the static type checking of wasm. The problem is that `call_indirect` can dynamically choose a function, but not its type. This needs to be known at compile time, as it is an explicit parameter. This is problematic as the trampoline might need to call functions with a variable amount of parameters. I ended up not solving this problem, and only allowing trampolining with functions of a single parameter.

The only feasible possible solution I could envision to this whould have involved a lot of complexity. Indeed, one could make all trampolinable function void, and let them load their arguments from memory. As such we effectivly implement a custom call stack.

Alternativly, this could be solved by adding higher-order functions and making sure all trampolined functions are curried. I did not find this to be feasible either.

4. Possible Extensions

First of all the trampolining support should be finished, by allowing multiple argument trampolines. This was described in the previous section, so I will not come back to how to do this.

An alternative extension, whould be by not trampolining in the compiler itself as was done here, but as a library like in Scala. This whould push more responsabilty to the Amy programmer, but does allow to choose whether or not to use trampolining. Furthermore, it whould avoid the whole mess of having to implement a custom call stack.

The implementation could be handled like for the Std library, with some stuff handled in amy code, and some things modified behind the scene by the compiler. There could be Trampoline ADT, that has a string parameter (containing the function name) and a arguments list. Trampoline calls could be made using a custom hardcoded function.

References

R. Bjarnarson. Stackless scala with free monads. *Scala Days*, 2012.

J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):1029–1052, 2004.

M. Schinz. Tail calls. `https://cs420.epfl.ch/archive/18/s/acc18_10_tail-calls.pdf`, 2018. Accessed: 2018-12-27.