Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html LastPage pages

Tail call elimination on the Java Virtual Machine

Michel Schinz [1] Martin Odersky [2]

École Polytechnique Fédérale de Lausanne

Abstract

A problem that often has to be solved by compilers for functional languages targeting the Java Virtual Machine is the elimination of tail calls. This paper explains how we solved it in our Funnel compiler and presents some experimental results about the impact our technique has on both performance and size of the compiled programs.

1 Introduction

A problem faced by authors of compilers for functional languages targeting the Java Virtual Machine (JVM) [7] is the elimination of tail calls. Functional programming languages typically do not have explicit looping constructs, and programmers therefore have to resort to recursion to perform loops. While this is no major problem for programmers, compiler writers have to make sure that their implementations perform *tail call elimination*, an optimisation which guarantees that loops implemented using recursion execute in constant space.

Implementing tail call elimination in its full generality typically requires some support from the target machine. Unfortunately, the JVM does not provide such support, and implementors either have to abandon general tail call elimination, or use special tricks.

We have implemented general tail call elimination for our Funnel compiler. This paper explains the technique we used, and shows its impact on code size and execution time for several benchmark programs. It is structured in the following way: Section 2 explains why tail call elimination is needed, especially in the current version of Funnel; Section 3 presents known techniques to eliminate tail calls, then our technique and some experimental results about it; Section 4 presents related work; finally, Section 5 concludes and talks about future work.

[1] Email: `Michel.Schinz@epfl.ch`
[2] Email: `Martin.Odersky@epfl.ch`

2 The Problem

Most functional programming languages, as well as some object-oriented languages like SmallTalk, do not provide loops as a built-in construct. Programmers in these languages use recursion to implement loops, and while the two constructs might appear equivalent to them, recursive calls, like any calls, consume stack space while loops do not.

An optimisation which is commonly done by these languages' implementations is tail call elimination, which replaces calls that appear in tail position with jumps. Intuitively, a call is said to be in tail position if it appears as the very last statement of a function.

A recursive tail call, that is one which calls the function to which it belongs, can usually be optimised without any kind of support from the target language, as most target languages permit jumps within function boundaries or loops. For most programming languages, the tail calls which are used to simulate (non-nested) loops are recursive. Therefore, optimising only recursive tail calls already covers a large number of cases.

The situation is a bit different in the currently implemented version of Funnel where visitors [5] are used to simulate algebraic data types [8]. This use of visitors unfortunately reduces the number of recursive tail calls and increases stack usage. For example, here is how the well-known `reverse_onto` function, which reverses its first list argument onto the second, can be written in Funnel:

```
def reverse_onto[a] (l1 : List[a], l2 : List[a]) : List[a] = l1.match {
  def Nil = l2
  def Cons (hl1, tl1) = reverse_onto (tl1, hl1::l2)
}
```

The body of this function is composed of a single call to the `match` function of the first list, `l1`. This `match` function expects a list visitor as argument. A list visitor is a record containing two functions: the first one, `Nil`, is called if the list is empty; the second one, `Cons`, is called if the list is not empty, with the list head and tail as arguments.

At first sight it might seem that the tail call to `reverse_onto` is recursive—and in a language with algebraic data types this would be the case—but it is not since it appears in the body of the visitor's `Cons` function. It cannot, therefore, be replaced by a simple `GOTO`. However, by expanding the `match` function, one could see that the recursive invocation of `reverse_onto` is in fact reached by a series of tail calls, two of which call dynamically bound methods. Hence, a general tail call elimination scheme that can also eliminate tail calls to statically unknown targets is required to make function `reverse_onto` operate in constant space.

Funnel is currently being redesigned to include a form of algebraic data types. In that version, the `reverse_onto` function could be formulated in a way that made the tail call directly recursive, and hence easy to eliminate statically. However, there are also many other examples that involve tail calls to unknown targets. For these, techniques like the ones developed here are still essential.

Ideally, the JVMs themselves should eliminate tail calls. In [2], Benton, Kennedy and Russell explain why they do not, and claim to have good reasons to believe that JVMs performing tail call elimination would appear soon. Unfortunately, three years later, none of the JVMs we are aware of perform general tail call elimination, although some (Microsoft's and IBM's are examples) eliminate recursive tail calls. The reason usually given for the omission of tail-calls are possible conflicts with Java's stack-walking security mechanism.

3 Eliminating Tail Calls

Various techniques have been proposed over the years to eliminate general (and not only recursive) tail calls, mostly for compilers targeting C.

A first one is to put the whole program in a big function and to simulate function calls using direct jumps or switch statements inside this function. This is often inadequate for large programs, and certainly not realistic on the JVM which does not allow method bodies of more than 64 kilobytes. This technique can however be used to remove tail calls among a small set of mutually recursive functions, while other tail calls are handled differently.

A popular technique is to use a *trampoline* [9]. A trampoline is an outer function which repeatedly calls an inner function. Each time the inner function wishes to tail call another function, it does not call it directly but simply returns its identity (e.g. as a closure) to the trampoline, which then does the call itself. Unlimited stack growth is thus avoided, but some performance is inevitably lost, mostly because all the calls made by the trampoline are calls to statically unknown functions.

Another well-known technique is Henry Baker's "Cheney on the M.T.A." [1]. With his technique, the program first has to be converted to continuation-passing style (CPS), therefore turning all calls into tail calls, and can then be compiled as usual. At run-time, when the stack is about to overflow, the current continuation is built and returned (or `longjmp`ed) to the trampoline "waiting" at the bottom of the call stack. An interesting property of Baker's technique is that the stack can be used as an allocation region for the youngest generation of a generational garbage collector—provided, of course, that a minor garbage collection is triggered just before the stack is shrunk, in order to copy live young data to the next generation.

3.1 Baker's technique on the JVM

Although most of the techniques presented above were originally developed for C as target language, they could be adapted without too much trouble to the JVM. For example, although the JVM does not have direct equivalents to C's `setjmp`/`longjmp` functions, one could simply use a chain of `RETURN`s or exceptions to simulate them.

Using the stack as the young generation of a generational garbage collector, as with Baker's technique, is not feasible with the JVM, however, because it does not give enough control over the memory allocation process.

Baker's technique can nevertheless seem interesting, because it guarantees that the program will always execute in a bounded stack space. To get a very rough idea of how much slowdown CPS conversion would imply on the JVM, we wrote a Fibonacci function both in standard and CPS style, in Funnel, and measured the execution time of both versions. The CPS version was approximately 25 times slower than the standard version. [3]

Since the current Funnel compiler does not perform a lot of optimisations, we rewrote the benchmark directly in Java and hand-optimised it using all the techniques we hope to put in our compiler in the future. Unfortunately, the results were not much better: the CPS version was still 20 times slower than the standard version.

It should be noted that the Fibonacci function suffers a lot from CPS transformation, as it contains two non-tail calls which imply the creation of continuations, and little else. Nevertheless, these results did not encourage us to use Baker's technique in our Funnel compiler to remove tail calls. We therefore devised another technique, described in the next section.

We did not measure the cost of the trampoline technique on the JVM, but Baker reports a slowdown of 2–3 for trampoline calls in C. Considering that with our compiler, calls to unknown functions go through dispatch functions (as will be seen in Section 3.3), we can conjecture that for us the slowdown would be even worse.

3.2 Technique used by the Funnel compiler

Let's take a closer look at the stack space needed by the different tail call optimisations. As a baseline we take the stack space needed by a hypothetical implementation on a virtual machine with a tail call instruction. Both the trampoline technique and the technique of compiling many functions into one work with the baseline stack space plus maybe a constant amount of space. Baker's technique needs only a constant amount of stack space but requires corresponding space on the heap for storing continuations, so that the needed stack and heap space together again correspond to the baseline plus a constant. Performing no tail call optimisation at all, on the other hand, requires stack space which exceeds the baseline by a potentially unlimited factor.

The observation underlying our technique is that we can trade speed for space by accepting a worst-case stack space which corresponds to the base-

[3] Measurments were made using Sun's HotSpot Client JVM for Intel x86 architecture, version 1.3. Results are similar for HotSpot Server, however.

line multiplied by a predetermined factor. A factor of 1 corresponds to the trampoline technique. Higher factors give better performance but require correspondingly larger stack size. With a factor of infinity, one obtains the implementation without tail call optimisation.

Our technique adds a new integer argument to all functions. This parameter, called the *tail call counter* (TCC), keeps track of the number of successive tail calls currently on top of the call stack. That is, each time a tail call is made, the current TCC plus one is passed to the callee, whereas for non-tail calls, 0 is passed.

The TCC is used to decide when the stack should be shrunk: at the beginning of every function, code is inserted to check its value and, if it is above some predefined *tail call limit* (TCL), the current continuation is built and returned to a trampoline "waiting" at the bottom of the tail call chain.

For this to work, some trampoline code also has to be inserted at all tail call sites. This code checks whether the callee returned a continuation *and* the TCC is equal to zero—meaning that this tail call is the first in the chain—in which case the continuation is invoked.

Therefore, at any one point during execution, we might have up to TCL tail called procedure frames on the stack, followed by at least one non-tail called procedure frame. Assuming that all frames have the same size, we arrive at an increase of the worst case stack size by a factor of TCL with respect to an ideal implementation optimising every tail call.

A small optimisation can be applied to this general scheme. Functions which do not contain any tail call do not need to check the TCC on entry. In that case, the maximum number of successive tail calls will potentially be TCL + 1 and not just TCL as before, but this is not important.

This trick could even be carried further, for example to functions which only call functions which do not contain tail calls, and so on. What we are really interested in, after all, are potentially unbounded chains of tail calls.

3.3 Implementation on the JVM

The Funnel compiler [11] always replaces directly recursive tail calls by simple `GOTO` instructions. To handle the remaining tail calls, it uses the technique described in the previous section.

Implementing this technique on the JVM is relatively straightforward, but the question of how to shrink the stack remains: as explained in section 3.1, the stack can be shrunk either using a chain of `RETURN` instructions or using exceptions.

Shrinking the stack using a chain of `RETURN` instructions means that the topmost function in the call stack builds the current continuation and returns it to its caller. The caller will itself return it to its caller, and so on, until it reaches the trampoline waiting at the bottom of the chain. The trampoline then calls the continuation.

Shrinking the stack using exceptions means that the topmost function in the call stack builds the current continuation and throws it as an exception. This exception is immediately caught by the trampoline, provided that it installed an exception handler properly. The trampoline then calls the continuation.

We will now show how these two variants of our technique translate into real code by showing an example. The following Funnel function gets two functions and a value as arguments and applies the composition of the two functions to the value. It contains one non-tail call to `g` and one tail call to `f`.

```
def compose[a,b,c] (f : b→c, g : a→b, x : a) : c = f (g (x))
```

When the stack is shrunk using a chain of `RETURN` instructions, the code generated by our Funnel compiler for this function is equivalent to the Java code below—with some minor differences.

```
Object compose (int tcc, Object f, Object g, Object x) {
  if (tcc < TCL) {
    Object res = dispatch (tcc + 1, f, dispatch (0, g, x));
    if (tcc == 0 && res instanceof Continuation)
      return callContinuation (res);
    else
      return res;
  } else
    return new Continuation (COMPOSE_CLOSURE, f, g, x);
}
```

When the stack is shrunk using exceptions, the code looks like this:

```
Object compose (int tcc, Object f, Object g, Object x) {
  if (tcc < TCL) {
    Object res_g = dispatch (0, g, x);
    if (tcc == 0) {
      try { return dispatch (tcc + 1, f, res_g); }
      catch (Continuation c) { return callContinuation (c); }
    else
      return dispatch (tcc + 1, f, res_g);
  } else
    throw new Continuation (COMPOSE_CLOSURE, f, g, x);
}
```

This code makes use of the `Continuation` class, and calls the `dispatch` and `callContinuation` predefined functions. A short description of these parts of the Funnel run-time follows, but a more thorough description including code can be found in the appendix.

Continuations are represented as instances of the predefined `Continuation` class which simply holds a closure and the arguments it has to receive. The predefined `dispatch` function applies a closure to some arguments. The predefined `callContinuation` function calls the continuation it receives, and if this continuation returns (or throws) another continuation, it is also invoked, and so on until something else than a continuation is returned. This last returned value is the result of `callContinuation`. Finally, `COMPOSE_CLOSURE` is the constant closure for the `compose` function.

3.4 Impact on performances and code size

The technique presented above increases both execution time and code size of compiled programs. In order to get an idea of the cost of our tail call elimination technique, we measured the increase of both code size and execution time it implied on five benchmarks.

The programs used as benchmarks are described in table 1. The line count given is the number of non-blank lines as seen by the compiler. This number is furthermore split in two: the number of lines of the program itself, and the number of lines imported from the Funnel "standard library".

For each benchmark, we measured how many stack shrinks happened during their execution, using a tail call limit TCL of 40. For the first three benchmarks, no shrinks ever happen, because they do not contain enough successive tail calls; therefore, they give a good idea of the overhead imposed by our tail call elimination technique when it is not actually used.

The last two benchmarks, on the other hand, make heavy use of tail calls and thus benefit from their elimination. Notice that, strictly speaking, tail call elimination could be turned on only for these two programs. In practice, it can however be hard for a programmer to guess whether tail call elimination should be turned on or off for a given program.

Name	Description	Line count (approx.)			Stack
		own	library	total	shrinks
Othello	Play a full Othello game.	400	500	900	0
Expr	Build a 100 000 elements algebraic expression, derive it, evaluate it.	90	350	440	0
Fibo	Compute fibonacci(37) using naive algorithm.	3	0	3	0
Queens	Solve the 8 queens problem 20 times.	40	350	390	10 020
Sort	Build 500 lists of 250 random elements, merge-sort them.	100	0	100	149 500

Table 1
Benchmarks description

We measured code size increase by looking at two things: the size in bytes of the generated `.class` file, and the size of the code part of these class files. As can be seen in table 2, in all cases but one, the size of the `.class` file increases by a little less than 30% when tail call elimination (TCE) is turned on, independently of the technique used. The higher increase—in percent—for the `Fibo` benchmark is due to its very small size: when TCE is turned on, some utility functions have to be added to the executable, and in this case their size is important compared to the overall program size.

In the following tables and graphs, the abbreviation RET is used for the technique which uses a chain of `RETURN` instructions to shrink the stack, while the abbreviation EXN is used for the technique which uses exceptions.

	`.class` size		Code size	
Name	TCE RET	TCE EXN	TCE RET	TCE EXN
Othello	25%	27%	57%	31%
Expr	25%	27%	65%	35%
Fibo	70%	70%	65%	48%
Queens	28%	29%	73%	38%
Sort	23%	24%	40%	21%

Table 2
Increase in code size

We measured the cost of our technique in terms of speed on three different virtual machines: Sun's HotSpot Client VM v1.3 (build 1.3.0), IBM's VM v1.3 (build cn130-20010609) and Microsoft's SDK 4.0 VM (build 3802). All measurements were made on a 600 Mhz Pentium running Microsoft Windows NT 4.0, except for one case (`Queens`) which was tested on the same machine running Linux, because of stack overflow problems on NT. For each benchmark, we measured elapsed time (i.e. wall-clock time) three times in a row, and kept only the best time of the three.

Figure 1 shows graphically the impact of tail call elimination on execution time, using the two different techniques to shrink the stack. As was said above, the `Queens` benchmark causes a (legitimate) stack overflow when compiled without tail call elimination on all three VMs on NT. For this reason measures were made on Linux, and thus no data exist for this benchmark on the Microsoft VM.

The differences in speed between the normal version and the versions with tail call elimination are summed up in table 3. As one can see, the cost of tail call elimination is moderate, since most benchmarks are not slowed down by more than 15%. Comparing the two techniques is instructive: on benchmarks which do not shrink the stack, EXN seems preferable; but on benchmarks

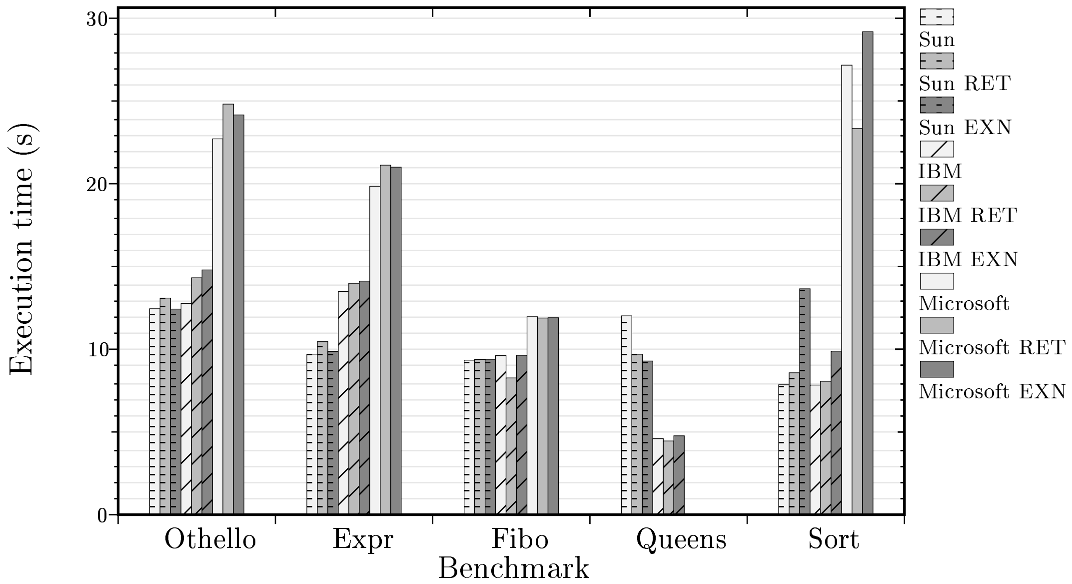

Fig. 1. Running time with and without tail call elimination

which do shrink the stack, RET seems to perform better. We can conjecture that this is partly due to the implementation of exceptions on JVMs: since in Java exceptions are used only to signal real errors, and not as a control-flow mechanism, installing exception handlers should be cheap whereas throwing exceptions can be more expensive.

	Sun		IBM		Microsoft	
Name	RET	EXN	RET	EXN	RET	EXN
Othello	5%	0%	12%	16%	9%	6%
Expr	8%	2%	4%	5%	6%	6%
Fibo	0%	1%	−14%	0%	−1%	−1%
Queens	−19%	−23%	−3%	4%	n/a	n/a
Sort	9%	74%	3%	26%	−14%	7%

Table 3
Slowdown implied by tail call elimination

Some benchmarks—e.g. `Queens`—even run faster when tail call elimination is turned on. What happens in these cases is that tail call elimination, by periodically shrinking the stack, destroys pointers to dead objects which are still on the stack, thereby making them reclaimable by the garbage collector. This reduces the time spent in the garbage collector, and thus also the overall running time.

There are two ways to circumvent this problem. The first one is to start the JVM with a bigger heap size. For example, if the `Queens` benchmark is started with an initial heap size of 50 Mb, then the version without tail call elimination runs slightly faster than the two other versions. The second way to circumvent this problem is to clear all pointers before doing a tail call. We experimented with this technique in our compiler, but the results were disappointing: even though the version which clears the pointers usually runs faster than the normal version, it is still slower than the version performing tail call elimination.

As a final note, it should be said that the current version of the Funnel compiler does not generate highly optimised code. As things improve, the time spent in tail call elimination code will inevitably represent a higher percentage of the total execution time. The above numbers should therefore be taken more as lower bounds than as definitive values. That said, some optimisations we plan to introduce—most notably inlining—will reduce the number of calls, thus also reducing the cost of tail call elimination.

3.5 Choosing the right limit

As explained in section 3.2, our technique of tail call elimination counts the number of successive tail calls and shrinks the stack when more than a predefined number of them have been made. Choosing intuitively a good value for this tail call limit is not easy, since a compromise has to be made: on the one hand, a low value seems preferable to minimise stack usage; on the other hand, a high value seems preferable for performance, since some time is lost each time the stack is shrunk.

To know how the choice of this limit affects performance, we compiled the `Sort` benchmark with different limits and measured its execution time. The results are presented graphically in figure 2. As could be expected, low values (10 and below) have a very bad impact on performance, but then the curve tends to stabilise. A limit of 40 looks like a good compromise between low stack usage and decent performances.

4 Related Work

We looked at only a subset of the huge amount of compilers which produce code for the JVM in order to know how they dealt with the problem of tail call elimination.

MLj [2] optimises recursive tail calls, but does not do general tail call elimination.

Kawa [3], like our Funnel compiler, has an option to turn on general tail call elimination. Although Kawa's author originally planned to eliminate tail calls by compiling every compilation unit to one big function [4], the current version of Kawa uses the standard trampoline technique.

Wakeling's Haskell compiler for the JVM [10] apparently uses the standard trampoline technique.

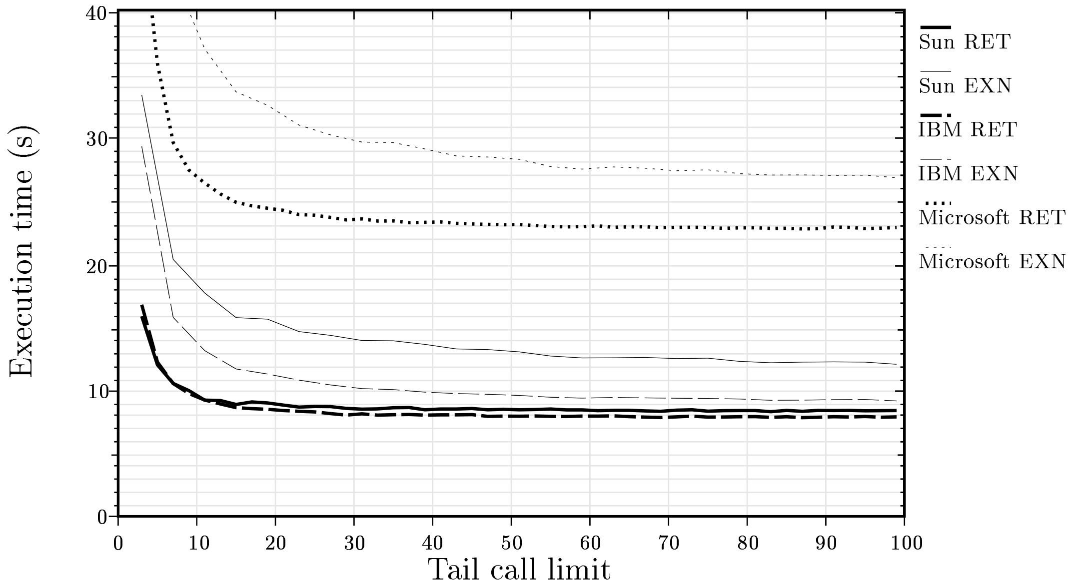

Fig. 2. Influence of tail call limit on performance

5 Conclusion and Future Work

Although we still do not have experience with large Funnel programs, the tail call elimination technique presented above has proved useful in avoiding stack overflows in many practical cases.

Unfortunately, as explained in section 2, because of their heavy use of visitors, typical Funnel programs put a lot of pressure on the JVM stack. Since our technique only optimises tail calls, it cannot do anything to reduce stack usage of normal calls.

One way of reducing stack usage of Funnel programs would be inlining. Using techniques similar to Hölzle's polymorphic inline caching [6], visitors could even be removed completely. We plan to implement such optimisations in future versions of our compiler.

Acknowledgement

Philippe Altherr, Christoph Zenger, Matthias Zenger and the anonymous referees gave very useful feedback about the paper.

References

[1] Henry G. Baker, Jr. CONS should not CONS its arguments, part II: Cheney on the M. T. A. Draft Version, January 1994.

[2] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference*

on Functional Programming (ICFP '98), volume 34(1) of *ACM SIGPLAN Notices*, pages 129–140. ACM, June 1999.

[3] Per Bothner. The Kawa Scheme system. `http://www.gnu.org/software/kawa/`.

[4] Per Bothner. Kawa internals: Compiling Scheme to Java. `http://www.gnu.org/software/kawa/internals.html`, November 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] Urs Hölzle. *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, 1994.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, April 1999.

[8] Martin Odersky. An overview of Functional Nets. Lecture Notes, APPSEM Summer School, September 2000.

[9] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[10] David Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):579–603, November 1999.

[11] Matthias Zenger and Michel Schinz. Funnel compiler release 8. `http://lampwww.epfl.ch/funnel/`.

Appendix: run-time classes and functions

The `Continuation` class

The `Continuation` class holds a closure and the arguments it has to receive. It can hold up to 4 separate arguments, and when more arguments are required, they are simply wrapped in a tuple, which is passed as a single argument. This wrapping is *not* done in the `Continuation` class, but in the code which uses it, and therefore does not appear below.

The `Continuation` class extends the standard Java class `Throwable` so that continuations can be thrown when the stack is shrunk using exceptions. Following [2], we redefine the `fillInStackTrace` method to avoid losing time creating a stack trace that will never be used.

```
final public class Continuation extends Throwable {
    final public Closure closure;
    final public int args;
    public Object arg1, arg2, arg3, arg4;

    public Continuation (Closure closure) {
        this.closure = closure;
        this.args = 0;
    }

    public Continuation (Closure closure, Object arg1) {
        this.closure = closure;
        this.arg1 = arg1;
        this.args = 1;
    }

    // ... constructors for 2, 3 and 4 arguments are similar.

    public Throwable fillInStackTrace () {
        return this;
    }
}
```

Dispatch functions

The `dispatch` functions are used to apply closures and to call functions of records. For simplicity, we only consider here that they are used to apply closures. Notice that if they were really used only to apply closures, other, more efficient techniques could be used, like the one of MLj [2].

There is a `dispatch` function for functions taking 0 arguments, one for functions taking 1 argument, and so on to cover all the functions in the program. In the code below, only `dispatch` functions with 1 and 3 arguments are shown.

Closures are composed of an integer index, which plays the role of the code pointer, and an environment.

```
Object dispatch (int tcc, Closure c, Object arg1) {
    switch (c.index) {
    case 2:
        return f1 (tcc + 1, c.env, arg1);
    case 11:
        return f2 (tcc + 1, c.env, arg1);
    // ... and so on ...
    }
}

Object dispatch (int tcc, Closure c, Object arg1, Object arg2, Object arg3) {
    switch (c.index) {
    case 15:
        return compose (tcc + 1, c.env, arg1, arg2, arg3);
    // ... and so on ...
    }
}
```

5.1 The `callContinuation` function

The `callContinuation` function gets a continuation as argument and applies it repeatedly until a "real" result (i.e. something else than a continuation) is returned, and finally return that result.

There are in fact two different versions of `callContinuation`, one which is used when continuations are returned, and one which is used when continuations are thrown as exceptions. Only the first version is presented here.

```
Object callContinuation (Continuation c) {
    Object res;
    while (true) {
        switch (c.args) {
        case 0:
            res = dispatch (1, c.closure); break;
        case 1:
            res = dispatch (1, c.closure, c.arg1); break;
            // ... cases for 2, 3 and 4 arguments are similar
        }
        if (! (res instanceof Continuation))
            break;
        c = (Continuation) res;
    }
    return res;
}
```