Tail Call Elimination and Data Representation for Functional Languages on the Java Virtual Machine

Magnus Madsen
Aalborg University, Denmark
magnus@cs.aau.dk

Ramin Zarifi
University of Waterloo, Canada
rzarifi@uwaterloo.ca

Ondřej Lhoták
University of Waterloo, Canada
olhotak@uwaterloo.ca

Abstract

The Java Virtual Machine (JVM) offers an attractive runtime environment for programming language implementors. The JVM has a simple bytecode format, excellent performance, multiple state-of-the art garbage collectors, robust backwards compatibility, and it runs on almost all platforms. Further, the Java ecosystem grants access to a plethora of libraries and tooling, including debuggers and profilers.

Yet, the JVM was originally designed for Java, and its representation of code and data may cause difficulties for other languages. In this paper, we discuss how to efficiently implement functional languages on the JVM. We focus on two overall challenges: (a) how to efficiently represent algebraic data types in the presence of tags and tuples, option types, newtypes, and parametric polymorphism, and (b) how to support full tail call elimination on the JVM.

We present two technical contributions: A fused representation of tags and tuples and a full tail call elimination strategy that is thread-safe. We implement these techniques in the Flix language and evaluate their performance.

CCS Concepts • **Software and its engineering → Functional languages**;

Keywords tail call elimination, tag tuple fusion, jvm, java

ACM Reference Format:
Magnus Madsen, Ramin Zarifi, and Ondřej Lhoták. 2018. Tail Call Elimination and Data Representation for Functional Languages on the Java Virtual Machine. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3178372.3179499

1 Introduction

The Java Virtual Machine (JVM) has proven to be a popular runtime environment for a multitude of languages invented after Java, e.g. Ceylon, Clojure, Kotlin and Scala.

The popularity of the JVM is due to several factors: (i) the Java platform comes with an extensive standard library and the Java ecosystem offers a very large collection of libraries and frameworks, (ii) the JVM is available for most platforms and has excellent backwards compatibility, (iii) the JVM bytecode format, the low-level instructions for the JVM, is easy to understand and to generate, (iv) the JIT compiler in the JVM emits fast machine code, (v) the JVM comes with state-of-the-art garbage collectors, a vital feature for object-oriented and functional languages, and finally (vi) the JVM offers excellent tooling for debugging and profiling.

Despite these numerous advantages, the JVM also has several limitations: Originally designed for Java, the JVM lacks several features that are important for languages that do not follow the object-oriented paradigm, although recently, some progress has been made to accommodate other languages [Wimmer and Würthinger 2012].

In this paper, we study the compilation of statically typed functional languages to JVM bytecode. Functional programming promotes a programming style that emphasizes the use of algebraic data types, closures, higher-order functions, pattern matching, and recursion, as opposed to mutable data structures and control-flow based on loops.

We identify two overall challenges for functional languages that wish to target the JVM: (**A**) How to efficiently represent algebraic data types? With the specific sub-challenges: (A1) how to represent tagged tuples? (A2) how to represent optional values? (A3) how to represent new type values?, (A4) how to represent polymorphic data types and functions?, and (A5) how to compile pattern matches? and (**B**) How to support full tail call elimination? In typical functional programs, iteration is expressed through the use of recursion and without full tail call elimination such programs may exhaust the available stack and crash.

In this work, we implement and evaluate compilation strategies for the above challenges in the compiler for the Flix programming language [Madsen et al. 2016a,b].

In summary, the contributions of this paper are:

- We present two technical contributions: A technique to fuse the representation of tags and tuples at runtime (challenge **A1**) and a technique for thread-safe full tail call elimination on the JVM (challenge **B**).
- We implement these two techniques, along with known techniques for the challenges **A2**–**A5**, in the Flix language and evaluate their performance impact.

2 Motivation

We begin with a discussion of the challenges we face in supporting common functional programming idioms and features on the JVM. We give examples in Flix code, but the challenges are equally applicable to any statically-typed functional programming language.

Algebraic Data Types. Most functional languages offer at least three categories of data types: primitive types (e.g. integers), tagged unions (i.e. sum types), and tuples. From these types, more interesting types can be constructed. In Flix, we can use a pair of integers, a 2-tuple, to represent a point:

```
def point(x: Int, y: Int): (Int, Int) = (x, y)
```

The empty tuple is called `Unit` and is denoted by `()`. In Flix, a tagged union is defined with the `enum` keyword:

```
enum Color {
    case Red,
    case Blue,
    case Green
}
```

The three cases `Red`, `Blue`, and `Green` are the tags of the enum `Color`. Each tag has an inner type. If the inner type is omitted, it is implicitly assumed to be `Unit`. Tags and tuples are orthogonal features, but we can combine them to express richer data types such as a list data type:

```
enum List[a] {
    case Nil,
    case Cons(a, List[a])
}
```

The `List[a]` type is polymorphic in the type variable `a`. As usual, a value of type `List[a]` is either the empty list denoted by the tag `Nil` or a cons cell denoted by `Cons` which is a pair, i.e. a 2-tuple, of an element of type `a` and the tail of the list of type `List[a]`. It is instructive to compare this definition to the Scala equivalent:

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](x: A, xs: List[A]) extends List[A]
```

Here, the `trait` keyword defines a common super-type for the `Nil` object and the `Cons` case class. In Scala, a case class combines several concepts into one: A case class is both its own distinct type, a record type with named fields, and a product type. However, a case class is *not* a tuple. The consequence is that the value of the case class cannot be treated separately. For example, the following code is legal in Flix, but cannot be expressed in Scala:

```
enum Shape {
    case Point(Int, Int),
    case Polygon(List[(Int, Int)])
}

def rotatePoint(p: (Int, Int)): (Int, Int) =
    (-snd(p), fst(p))

def rotateShape(s: Shape): Shape = match p with {
    case Point(p)    => Point(rotatePoint(p))
    case Polygon(ps) => Polygon(List.map(rotatePoint, ps))
}
```

Here the `rotatePoint` function takes a point and rotates it 90 degrees. The `rotateShape` takes a shape and rotates the points associated with the shape using `rotatePoint`.

As the example illustrates, the conceptual separation of tags and tuples is useful and less restrictive than the case classes in Scala. It is straightforward to compile tags and tuples to JVM bytecode. However, if we compile tags to one set of classes and tuples to another set of classes, then the runtime representation of a cons cell will require two objects: one for the tag and one for the pair! This causes additional memory allocation and garbage collection churn compared to the case classes in Scala. This leads us to the first challenge:

Challenge A1: How can we efficiently represent tagged tuples while keeping tags and tuples distinct at the language level?

Optional Values. Many languages support `null` values. In Java, `null` is a subtype of any type τ and can be used whenever a value of type τ is expected. Programmers use null values to represent uninitialized values, absent values, or illegal values. At first sight, null values may seem like a convenient feature, but they come at a high cost: Accessing a field or invoking a method on a null value throws the dreaded `NullPointerException`. Such crashes are often difficult to debug since it is hard to determine from where the null value originated. Tony Hoare famously described null values as his "billion dollar mistake". Functional languages tend to eschew null values and instead rely on the `Option` type. In Flix, this type is defined as the algebraic data type:

```
enum Option[a] {
    case None,
    case Some(a)
}
```

Here the `None` tag represents an absent value, whereas the `Some(a)` tag represents the presence of a value of type `a`. `Option` types are good for safety, but come with a performance cost: Every use of `Some(x)` requires allocation of an object and later its garbage collection, thus slowing down execution. This leads us to the second challenge:

Challenge A2: How can we efficiently represent optional values?

New Types. In typed functional programming, it is considered good style to encode as much information as possible in the type system. For example, if we wanted to represent temperatures in both Celsius and Fahrenheit we should introduce a data type for each unit of measure:

```
type Celsius = Celsius(Int)
type Fahrenheit = Fahrenheit(Int)
```

In Flix, such a type declaration is syntatic sugar for:

```
enum Celsius {
    case Celsius(Int)
}
```

The use of such "new types" ensures that we can never confuse a temperature in Celsius with one in Fahrenheit.

However, as in the previous challenges, this design requires additional memory allocation and garbage collection, thus slowing down execution. This leads us to the third challenge:

Challenge A3: How can we efficiently represent new type values?

Parametric Polymorphism. Generic programming, in the form of parametric polymorphism, is common in typed functional languages. The `List[a]` type is an example of a polymorphic data type and its operations are polymorphic functions. The JVM only supports primitive types (e.g. integers) and reference types (i.e. objects). This means we cannot just emit a single JVM method that works for both primitive types and reference types. This leads us to the fourth challenge:

Challenge A4: How can we efficiently represent polymorphic data types and functions to avoid boxing of primitive values?

Pattern Matching. Algebraic data types are useful, but their real power comes when combined with pattern matching. For example, if we wanted to determine whether a list ends with the integers 1, 2, and 3, we could write:

```
def endsWith123(l: List[Int]): Bool = match l with {
    case Nil                            => false
    case Cons(_, Nil)                   => false
    case Cons(_, Cons(_, Nil))          => false
    case Cons(1, Cons(2, Cons(3, Nil))) => true
    case Cons(_, xs)                    => endsWith123(xs)
}
```

As pattern matching is not expressible in JVM bytecode, a functional language compiler must translate pattern matches to a sequence of lower-level instructions. This leads us to the fifth and final challenge related to algebraic data types:

Challenge A5: How should pattern matches be compiled to JVM bytecode?

Tail Call Elimination. Functional languages tend *not* to rely on imperative control-flow structures such as while- and for-loops, but instead use recursion. For example, consider the two functions:

```
def odd(n: Int): Bool = match n with {
    case 0 => false
    case 1 => true
    case n => even(n - 1)
}
def even(n: Int): Bool = match n with {
    case 0 => true
    case 1 => false
    case n => odd(n - 1)
}
```

The two functions compute whether an integer is odd or even. The functions are mutually recursive and each performs a *tail call* to the other. A function call is a tail call if the calling function immediately returns after the call.

We can compile functions, like odd and even, to regular JVM methods, and compile function calls to regular JVM method calls. However, each JVM method call requires a stack frame, and in a program like the above, all stack space would be quickly exhausted leading to a crash.

The purpose of *tail call elimination* is to avoid the creation of new stack frames whenever a tail call occurs. Runtime environments typically offer instructions to perform tail calls, but unfortunately the JVM does not.

The lack of tail calls on the JVM is not just a theoretical problem, but a real-world issue. At best, functional languages without tail call elimination force programmers to manually rewrite their code using imperative loops, trampolines, or worklists. At worst, such problems become ticking time bombs waiting for an input that will exhausts the available stack space and crash the program.

For Scala, a quick look through mailing lists, forum posts, and issue trackers shows that the lack of tail call elimination is a real pain point. The books *Functional Programming in Scala* by Chiusano and Bjarnason and *Programming Scala* by Wampler and Payne both acknowledge this issue, but rather meekly suggest the use of trampolines or a tail recursive monad! The documentation for Cats, a popular Scala library for functional programming, states:

In addition to requiring `flatMap` and `pure`, Cats has chosen to require `tailRecM` which encodes stack safe monadic recursion, [...]. Because monadic recursion is so common in functional programming but is not stack safe on the JVM, Cats has chosen to require this method of all monad implementations as opposed to just a subset. [...]

This shows that the lack of tail call elimination imposes a real burden on functional programmers who use the JVM. This leads us to the last major challenge:

Challenge B: How can we support full tail call elimination on the JVM?

In 1998, Benton et al. [1998] optimistically suggested that:

"We have reason to believe that JVMs which do tail call elimination may appear soon, ..."

Almost twenty years later, the JVM instruction set still lacks support for tail calls.

3 Implementation

In the previous section, we described several features that are essential to functional programming and the challenges they present. In this section, we describe how we address these challenges in the compiler for Flix.

We will not follow the same structure as in the previous section, but instead present things in an order that makes sense from an implementation point of view. We will devote the majority of the space to the tag and tuple fusion and tail call elimination strategies, as these are the two major technical contributions of the paper. The remainder of the challenges, which have known solutions, we touch on briefly.

We begin with a discussion of tail call elimination, as this turns out to be quite involved.

3.1 Tail Call Elimination (Challenge B)

Over the years, several strategies for tail call elimination on the JVM have been proposed, including trampolines [Bjarnason 2012; Stadler et al. 2009], stack shrinking [Schinz and Odersky 2001], and impure functional objects [Tauber et al. 2015]. We briefly discuss some of these, before we describe our technique which is a thread-safe variant of impure functional objects. We will proceed by example. Assume we want to implement the factorial function shown below:

```
public int factorial(int num, int acc) {
    if (num == 0) {
        return acc;
    }
    return factorial(num - 1, num * acc);
}
```

We will focus on the general case of tail call elimination, ignoring for the moment that this function is tail recursive. We now discuss how to express this function using trampolines and impure functional objects. Other techniques are discussed in Section 5.

Trampolines. A trampoline is a classic technique to implement tail call elimination. The idea is that each function is modified to return a *continuation*, i.e. an object that represents either the next tail call or holds the result of the entire computation. The trampoline is itself a loop that evaluates the continuation until the result is available. We can implement the factorial function using a trampoline:

First, we define an interface to capture the continuation:

```
interface Continuation {
    int getResult();
    Continuation apply();
}
```

We invoke the continuation by calling the `apply` method. This call either returns a new continuation or `null`, in which case the result is available by calling the `getResult` method.

Second, we define a class for the body of the factorial function which implements the `Continuation` interface:

```
class Factorial implements Continuation {
    private int arg0, arg1, res;
    public Factorial(int num, int acc) { ... }
    public int getResult() { ... }
    public Continuation apply() {
        int num = this.arg0;
        int acc = this.arg1;
        if (num == 0) {
            this.res = acc;
            return null;
        }
        return new Factorial(num - 1, num * acc)
    }
}
```

The arguments to the factorial function are passed as parameters to the constructor which stores them into two fields: `arg0` and `arg1`. The `apply` method performs the computation and either stores the result in the `res` field or returns a new continuation object.

Third, and finally, to call the factorial function we create an instance of the factorial class with the appropriate arguments and repeatedly invoke the `apply` method:

```
Continuation next = new Factorial(inputNumber);
Continuation prev = null;
do {
    prev = next;
    next = next.apply();
} while (next != null);
int result = prev.getResult();
```

The trampoline repeatedly invokes the current continuation until the result is available. The disadvantage of trampolines is that each tail call allocates a new object (which is later garbage collected) and on the JVM a single object allocation is more expensive than a single method call.

A key insight is that we do not actually need to *allocate* an object for the continuation, but instead we can *reuse* the same continuation objects over and over. This is the idea behind *impure functional objects* of Tauber et al. [2015].

Impure Functional Objects. The main idea behind impure functional objects (IFOs) is to create only a single mutable continuation object for each function in the program, and reuse that same object for every tail call to that function.

Here is how to express the factorial function using IFOs:

```
class Factorial implements Continuation {
    private static Factorial INSTANCE = new Factorial();
    private int num, acc, res;

    void setArg0(int arg0) { ... }
    void setArg1(int arg1) { ... }
    int getResult() { ... }

    public void apply() {
        if (this.num == 0) {
            this.res = this.acc;
            return;
        }
        Factorial c = Factorial.INSTANCE;
        int arg0 = this.num - 1;
        int arg1 = this.num * this.acc;
        c.setArg0(arg0);
        c.setArg1(arg1)
        Global.continuation = c
    }
}
```

The Factorial class has only one instance, stored in the static field `INSTANCE`. The body of the `apply` method performs the same computation as before, but instead of returning a new continuation object, it sets the arguments of the singleton factorial instance, and it sets the static `continuation` field to point to the factorial instance.

The `Global` class is simply defined as:

```
class Global {
    public static Continuation continuation = null;
}
```

It holds the next continuation to evaluate, or `null` if all tail calls have been evaluated. To call the factorial function, and to ensure that all tail calls are evaluated, we use the loop:

```
Continuation prev = null;
do {
    prev = Global.continuation;
    Global.continuation = null;
    prev.apply();
} while (Global.continuation != null);
int result = prev.getResult();
```

The loop repeatedly evaluates the static `continuation` field of the `Global` class until it becomes `null`. At this point, all tail calls have finished and the result is `prev.getResult()`.

As shown by Tauber et al. [2015], IFOs are significantly faster than trampolines, since they are allocation free. However, an important challenge remains: Unlike trampolines, IFOs are not thread-safe. Each function is represented by one object and multiple threads may call the same function at the same time. If they do, then there is a data race on the arguments and result values of that function. Even worse, there is a data race on the globally shared static field that holds the next continuation. In the original paper, the authors allude to these issues with the comment:

> [...], we will adopt the thread-safe version of our translation – one main difference is that IFOs should be allocated at their call sites rather than at their definition sites.

It is not clear (to us) what is meant by "allocated at their call sites", since one of the main strengths of the IFO technique is that it is *allocation free*. In fact, the whole idea of IFOs was to have a single mutable object for each function.

We propose two thread-safe and allocation free variants of IFOs. The idea in both is that each thread should have its own continuation pointer and set of IFOs. Before we present the two proposals, it is worth stating explicitly that simply sprinkling enough `synchronized` keywords or adding manual locks to the IFOs would not achieve the desired effect: Any locking scheme would be totally disastrous for multi-threaded performance and would not even be safe, unless a single global lock was used.

ThreadLocal IFOs. A straightforward way to ensure that each thread has its own continuation pointer and set of IFOs is to use `ThreadLocal`, a Java class that allows each thread to maintain its own version of a (static) field. We can wrap every static field in a `ThreadLocal` object and use its getters and setters to access the underlying value. Using `ThreadLocal` is simple to understand and implement. Unfortunately, as we shall see in Section 4, it is not very performant. It turns out that internally `ThreadLocal` relies on a hash map and consequently every access operation requires some indirection through this map.

Contextual IFOs. We propose a solution where every call is explicitly passed a *context object* with the IFOs. Each thread has its own context object and hence there are no data races nor any need for synchronization.

We introduce a `Context` class which has a field for the continuation and a field for every IFO instance:

```
class Context {
    Continuation continuation = null;
    Factorial factorial = new Factorial();
    // ...
}
```

We pass an instance of the `Context` class to every IFO which provides access to other IFOs and the ability to set the continuation. In reality, our implementation is significantly more complex since: (i) we split the `Context` class into multiple namespaces since a single class can hold at most 65,536 fields, (ii) we specialize every type to avoid boxing, and (iii) we have to represent closures.

We now describe this implementation in greater detail.

3.1.1 Continuation Interfaces

We emit a *continuation interface* for every primitive Java type and one for reference types. For example:

```
interface Cont$Bool {
    boolean getResult();
    void apply(Context c);
}
```

As shown, we explictly pass an instance of the `Context` class to the `apply` method.

3.1.2 Function Interfaces

We emit a *function interface* for each function type in the program. For example, for the type `Int` → `Bool`, we emit:

```
interface Fn1$Int$Bool extends Cont$Bool {
    void setArg0(int v);
}
```

For the type (`Option[Int], Int`) → `Int`, we emit:

```
interface Fn2$Obj$Int$Int extends Cont$Int {
    void setArg0(Object v);
    void setArg1(int v);
}
```

The function interfaces extend the continuation interfaces and add methods to set the arguments of the IFO. We need the function interfaces to support closures where the exact IFO being invoked is not statically known.

We must keep the continuation and function interfaces separate since they are used at different places in the code. The function interface is used to pass the arguments of a call at a tail call site, whereas the continuation interface is used for evaluation of tail calls at a non-tail call site.

3.1.3 Function Classes

We emit a *function class* for each function in the program. The instances of these classes are the IFOs of the program.

For example, for the function `List.length` specialized to the type `List[Int]` → `Int`, we emit the class:

```
package List;
class Def$length extends Fn1$Obj$Int {
    private Object arg0;
    private int res;
    void setArg0(Object v) { this.arg0 = v; }
    int getResult() { return this.res; }
    void apply(Context c) { ... }
}
```

At runtime, we will arrange for each instance of the `Context` class to have one instance of each function class. Thus the memory overhead is constant: We have one function object per context object of which we have one per thread.

3.1.4 Namespace Classes

We could put all instances of the function classes, i.e. the IFOs, into the `Context` object, but a class is limited to `65,536` fields and methods which would limit a Flix program to the same number of functions. Instead, we organize the function definitions into a set of *namespaces classes*. This limits us to `65,536` functions per namespace. Each namespace class holds references to all IFOs that appear in it. For example, if we have a program with the two namespaces:

```
namespace Option {
    def isEmpty[a](o: Option[a]): Bool = ...
}
namespace List {
    def length[a](l: List[a]): Int = ...
}
```

we emit the two namespace classes:

```
package Option;
class Ns {
    final Fn1$Obj$Bool isEmpty = new Def$isEmpty()
}
package List;
class Ns {
    final Fn1$Obj$Int length = new Def$length()
}
```

3.1.5 The Context Class

All namespace classes are instantiated by the `Context` class which also holds the current continuation:

```
class Context {
    final Option.Ns Option = new Option.Ns();
    final List.Ns List = new List.Ns();
    Object continuation = null;
}
```

The namespaces are flat. For example, the namespace `A.B.C` becomes a field named `A$B$C`. This ensures that to lookup an IFO requires at most two field dereferences: First, lookup the namespace object on the `Context` object. Second, lookup the IFO on the namespace object. The namespace fields of the `Context` class and the IFO fields of the namespace classes are all final. Hence, the JVM has the opportunity to inline these lookups and avoid the overhead of the indirections.

3.1.6 Function Calls

We emit code for five different types of function calls:

- ApplyClo(*clo*, *args*): An indirect call to the closure expression *clo* with the arguments *args*.
- ApplyDef(*sym*, *args*): A direct call to the function with symbol *sym* with the arguments *args*.
- ApplyCloTail(*clo*, *args*): An indirect *tail* call to the closure expression *clo* with the arguments *args*.
- ApplyDefTail(*sym*, *args*): A direct *tail* call to the function with symbol *sym* with the arguments *args*.
- ApplySelfTail(*sym*, *args*): A tail recursive call to the function with symbol *sym* with the arguments *args*.

We now discuss how to emit code for each node. In our presentation, we will treat `ApplyClo` and `ApplyDef` together, and similarly for `ApplyCloTail` and `ApplyDefTail`.

ApplyClo and ApplyDef. Assume we want to call a function in non-tail position of type `Int` → `Bool`. Assume further that the local variables `t`, `a`, and `c` hold the closure object, the argument, and the `Context` object. We then emit the code:

```
c.continuation = t;
t.setArg0(a);

Cont$Bool f = null;
do {
  f = (Cont$Bool) c.continuation;
  c.continuation = null;
  f.apply(c);
} while (c.continuation != null)
```

We first set the `continuation` field of the `Context` object to the IFO stored in the local variable `t`. Next, we set the argument of `t` to the value of `a`. Note that the type of `t` is `Fn1$Int$Bool` and hence we are allowed to call `setArg0` with an integer argument. Then, since this is a *non-tail call*, we repeatedly call the IFO stored in `c.continuation` until all tail calls inside `t` have been evaluated. Finally, the result is available in `f.getResult()`. If we were calling a function instead of a closure, i.e. in the `ApplyDef` case, we would simply retrieve the IFO from the context object. For example, if we wanted to call `List.length` we would emit:

```
Fn1$Int$Bool t = c.List.length;
```

immediately before the code shown above.

ApplyCloTail and ApplyDefTail. Assume we want to call a closure in tail position of type `Int` → `Bool`. In this case, we emit the code:

```
c.continuation = t
t.setArg0(a)
```

where, as before, the local variables `t`, `a`, and `c` are assumed to hold the IFO, the integer argument, and the `Context` object.

We do not need to perform any other actions, as to perform a tail call, we simply need to configure the continuation and its arguments, and then the loop further down the call stack will take care of calling the IFO.

ApplySelfTail. Assume we want a function to call *itself* tail recursively and that it has type `Int` → `Bool`. In this case, we can rewrite the entire call into a loop. We emit the code:

```
this.arg0 = v
goto entry
```

where the local variable `v` is assumed to hold the argument to the recursive call. The code assigns `v` to the argument field of the current IFO, i.e. `this`, and jumps to a label placed at the entry of the `apply` method.

3.1.7 Representation of Closures

Closures are straightforward to support in the current design. We introduce a *closure class* for each closure type. A closure class captures its free variables as fields and implements the appropriate function interface which allows it to operate as any other IFO. For example, here is the representation we would emit for a closure of type `Int` → `Bool` that has captured a variable of type `Char`:

```
class Clo$f implements Fn1$Int$Bool {
    // fields for captured variables ...
    private char cv0;
    // fields for formal arguments and return value ...
    // methods defined by Cont$Bool and Fn1$Int$Bool ...
}
```

A potential issue is that a closure can be passed from one thread to another. If that happens, a closure could inadvertently use the wrong `Context` object. We can avoid this situation by having each closure capture its `Context` object. When a closure is called, it compares its own `Context` object to the one passed as an argument. If they are the same, nothing is to be done, because the closure is on the same thread. Otherwise, the closure is copied with the appropriate `Context` object and execution continues as before. An alternative solution is to copy each closure when it is passed from one thread to another.

3.2 Tag and Tuple Fusion (Challenge A1)

It is simple to represent tags and tuples as classes. Each tag becomes a class and each arity of a tuple becomes a class. Figure 1 shows such a representation for the `List` algebraic data type, instantiated for integers. The two tags `Nil` and `Cons` are represented as the two classes `Nil` and `Cons$Obj`. The `Nil` class has a `value` field that points to the `Unit` value, and the `Cons$Obj` class has a `value` field that points to a pair. The pair, i.e. the 2-tuple, is represented by the class `Tuple2$Int$Obj` which holds the integer value directly and a reference to the rest of the list.

The problem with this design is that to represent a cons cell requires both a tag object (an instance of the `Cons$Obj` class) and a tuple object (an instance of the `Tuple2$Int$Obj` class.) This wastes not only memory, but also puts extra pressure on the garbage collector. Ideally each cons cell should be represented by a *single* object. We can achieve this with what we call *tag and tuple fusion*.

Figure 2 shows such a representation. The hierarchy is conceptually similar to before, except we introduce a new interface for each tuple type. Here we have the interface `ITuple2$Int$Obj` which is implemented by the class `Tuple2-$Int$Obj`. Instead of the class `Cons$Obj`, we introduce the fused class `Cons$T2$Int$Obj` which represents both the `Cons` tag and a pair of an integer and an object. Hence this class implements *both* the `IList$Int` and `ITuple2$Int$Obj` interfaces. We still need the class `Tuple2$Int$Obj` to represent tuples that are not tagged.

At compile time, when the compiler emits code for tagged expressions, it distinguishes two cases: If a tag has an inner tuple and the elements of the inner tuple are statically known, then the compiler can directly use the fused class representation. However, if the elements of the tuple are unknown, e.g. if the tuple is passed as a parameter, the compiler

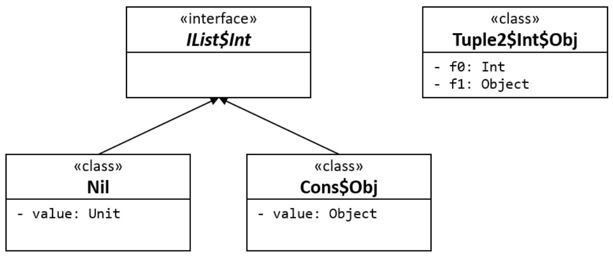

Figure 1. Standard Representation of Tags and Tuples.

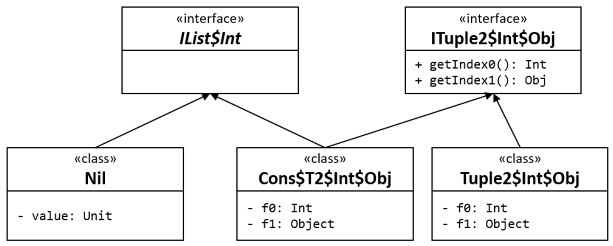

Figure 2. Fused Representation of Tags and Tuples.

must emit code to construct the fused class and copy over the fields of the tuple.

For example, consider the two functions:

```
def f(): List[Int] = Cons(42, Nil)
def g(p: (Int, List[Int])): List[Int] = Cons(p)
```

For the function `f`, the compiler emits bytecode to immediately construct an instance of the fused class, since both the tag and tuple elements are statically known. For the function `g`, on the other hand, the compiler has to emit code to construct the fused class by copying the fields of the tuple object passed into `g`.

3.3 Optional Values (Challenge A2)

In the previous section, we discussed how the Option type can be used to safely represent the absence of a value. In some cases, the compiler can represent optional values compactly by using `null` values internally. Thus we get the best of both worlds. For example, if we have a type `Option[List[Int]]` we can represent the `None` tag of the Option type as `null` and the `Some` tag as simply the list `List[Int]` itself.

Unfortunately, we cannot always use this representation. For example, if the type is `Option[Int]` we cannot represent the tag `None` as `null` since `null` is not a legal primitive int.

3.4 New Types (Challenge A3)

We can easily avoid the runtime overhead associated with type safe wrappers such as Celcius and Fahrenheit:

```
type Celsius = Celsius(Int)
type Fahrenheit = Fahrenheit(Int)
```

Every time we encounter an expression that constructs or destructs an algebraic data type which has a single tag, we simply use the underlying type. Hence the type only exists at compile time. A little care has to be taken to ensure that equality and `toString` work correctly on such types.

3.5 Polymorphic Code and Data (Challenge A4)

As we have seen, Flix supports generic programming with parametric polymorphism. This leads to the question of how to represent polymorphic code and data at runtime. The JVM supports primitive types (e.g. booleans, floats, and integers) and reference types (e.g. objects). Java and Scala typically compile polymorphic functions to bytecode that works on references. That way, for example, a list can be used to hold both objects and boxed primitive values. The advantage is that each polymorphic data type and function is only represented once. The disadvantage is that primitive values must be boxed, which imposes additional runtime overhead.

Flix follows in the footsteps of MLton and performs whole-program monomorphization. That is, each polymorphic function is fully specialized to its instantiated types. In other words, `List[Int]` and `List[String]` have their own representation and each function is specialized to operate on that representation. Internally the three types `List[Int]`, `List[Option[Int]]`, and `List[List[Int]]` are all distinct. However, the JVM backend knows that the representation of `List[Option[Int]]` and `List[List[Int]]` can be shared, since both lists hold an object. Hence these two types are *erased* to `List[Object]`. The result is the best of both worlds: primitive values are never boxed, but reference types share the same representation.

3.6 Pattern Matching (Challenge A5)

Flix implements pattern matching expressions by compilation to a sequence of lower-level instructions. For example, consider the match expression:

```
match l with {
    case Nil        => exp1
    case Cons(x, r) => exp2
}
```

Such an expression can be compiled into a sequence of nested lambda expressions [Jones and Lester Jones and Lester]:

```
let default = xs -> throw MatchError;
let lambda2 = xs ->
    if (xs Is Cons) {
        let tmp = Untag(xs);
        let x = Index(tmp, 0);
        let r = Index(tmp, 1);
            exp2
    } else default()
let lambda1 = xs ->
    if (xs Is Nil) exp1 else lambda2(xs);
lambda1(l)
```

Each case is compiled to a lambda expression that checks whether the pattern matches, and if not, calls the lambda for the next case. The lower-level operations `Is`, `Tag`, and `Untag` are used to check the tag of a value, create a tagged value, and to retrieve the inner value of a tagged value, respectively. The `Index` operation retrieves a value from a tuple offset.

An alternative representation is to compile pattern matches to a sequence of labels and jumps, as shown below:

```
label1:
    if (xs Is Nil) exp1 else goto label2
label2:
    if (xs Is Cons) {
        let tmp = Untag(xs);
        let x = Index(tmp, 0);
        let r = Index(tmp, 1);
            exp2
    } else goto default
default:
    throw MatchError
```

Here each case gives rise to a labelled piece of code. When a value fails to match a case, the control jumps to the label for the next case or eventually falls through to the default.

We now briefly discuss the advantages and disadvantages of each approach. Compilation to lambda expressions and function calls is easy since it maps pattern matches to constructs that already exist in the abstract syntax tree. The disadvantage is that the additional lambdas and calls lead to code with many small closures. On one hand, small methods are good for the JVM. On the other hand, many of these closures are needlessly passed around and may obscure the control flow for the JVM. When using labels and jumps, everything is kept in the same method, which may grow large and inhibit some JVM optimizations, but it avoids redundant closure allocation.

4 Evaluation

We now evaluate the performance impact of the proposed data representations and tail call elimination strategies.

4.1 Research Questions

We consider three research questions to guide our evaluation:

Q1: What is the performance cost of the various proposals for full tail call elimination on the JVM?

Q2: What is the performance impact of each proposed data representation strategy?

Q3: How does the performance of Flix programs compare to that of equivalent programs written in Scala?

4.2 Benchmarks

We perform four separate experiments to answer Q1–Q3. First, we use a micro benchmark to evaluate the likely performance cost of each proposed tail call elimination strategy, including our own proposal. Second, we compare the old Flix backend, which does not support tail call elimination, to the new backend which supports contextual IFOs. Third, we implement 20 small benchmarks in Flix and Scala to measure the performance impact of each proposed optimization and to compare the performance of the two languages. Fourth, we compare the performance of Flix and Scala on 4 programs ported from the Computer Language Benchmarks Game.[1]

We wrote the 20 small benchmarks to be representative of typical computations in functional languages. We used three criteria when crafting these programs, specifically we wanted: (i) programs that use features such as algebraic data

[1] at http://benchmarksgame.alioth.debian.org/

types, pattern matching, and tail calls, (ii) programs that perform many function calls to make the impact of the tail call elimination strategy clear, and (iii) programs that were short and would be easy to write in Flix and Scala. Thus, we ended up with simple benchmarks such as: (i) filtering a list of integers, (ii) folding a function over a list of integers, (iii) zipping two lists of integers, and so on.

All benchmarks are available in the GitHub repository:

https://github.com/flix/benchmarks/

4.3 Experimental Setup

We implemented the techniques in the Flix compiler which is open source and freely available online[2]. The Flix compiler is currently 45,000 lines of Scala code and the Flix test suite is 35,000 lines of Flix code. The new JVM backend required approximately 6,000 lines of code.

All experiments were performed on an Intel Core i5-4300U CPU @ 2.49GHz with 12GB of main memory on Windows 10. We used Java (JRE) version 1.8 (8u151) and Scala version 2.12.3. We warmed up the JVM and then ran each benchmark 1,000 times to measure the *median* execution time.

4.4 Results and Analysis

We now address each research question.

4.4.1 Q1: Tail Call Elimination (Challenge B)

In this paper, we described several tail call elimination strategies. We briefly recall each and how they work:

- **Trampolines.** Every function returns a continuation object or `null` if the computation is complete.
- **Impure Functional Objects (IFO).** Every function is represented as a mutable object where the arguments and result values are accessed via getters and setters.
- **IFOs + ThreadLocal.** A thread-safe variant of IFOs where each IFO is stored in a static `ThreadLocal` field.
- **IFOs + Context.** A thread-safe variant of IFOs where each IFO is stored in a `Context` object.

To evaluate the relative performance of these strategies, we implemented variants of the `odd` and `even` program using each strategy. We ran the programs on inputs of increasing sizes and compared the performance to a baseline implementation based on regular method calls. Figure 3 shows the result of this experiment. The x-axis is the input number. The y-axis is the execution time normalized to the program with regular method calls.

The trampoline approach has a slow-down of between 3.1x and 7.2x, which decreases as the call depth increases. The original IFOs fares significantly better with a slow-down of between 1.4x and 3.3x. This confirms that IFOs are faster than trampolines, as reported by Tauber et al. [2015]. The performance of `ThreadLocal` IFOs is much worse. The graph is clipped at 8.0x, and the slow-down of `ThreadLocal` IFOs

[2]at http://flix.github.io and http://github.com/flix

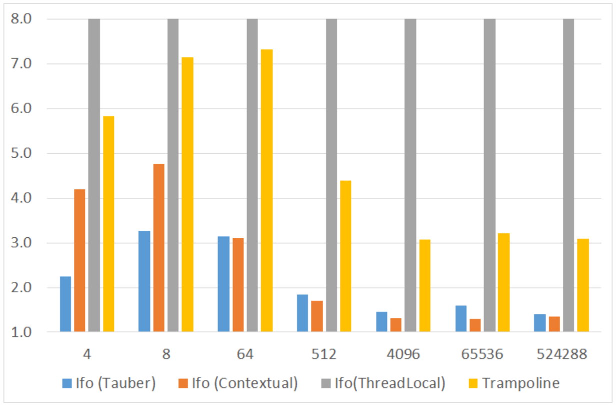

Figure 3. Performance of Tail Call Elimination Strategies.

is actually between 10.0x and 22.0x! Contextual IFOs have a slow-down of between 1.3x and 4.8x. For shallow call stacks, contextual IFOs are a bit slower than regular IFOs, but as the call depth grows, contextual IFOs become competitive.

Based on these results, we decided to implement a new backend for Flix based on contextual IFOs so that we could measure their performance on a wider range of programs. The old backend was based on regular method calls. The left part of Table 1 compares the performance of the old and new backends. The results show that the switch to contextual IFOs typically impose a slowdown of between 1.2x and 2.0x. This is towards the better end of the range suggested by the micro benchmark, which was between 1.3x and 4.8x.

The comparison between the old and new backend was performed with pattern matches compiled to labels and jumps for both backends. This was important since the new backend, with contextual IFOs, benefitted significantly more from this optimization than the old backend.

4.4.2 Q2: Data Representation (Challenges A1–A5)

We evaluate the performance impact of each data representation optimization. The results are shown in the right part of Table 1. The `Baseline` column shows the execution time of each benchmark with the new backend with every optimization *disabled*. The `PM` column shows the speed up with pattern matches compiled to labels and gotos. The next three columns, marked with ⋆, show the speed-up relative to `PM`.

Pattern Match Compilation (A5). The `PM` column shows the speed-up when pattern matches are compiled to labels and jumps. We observe a typical speed-up of 1.6x to 2.4x. It turns out that this speed-up is most significant with contextual IFOs compared to the old backend with regular method calls. This makes intuitive sense, since calls are slightly more expensive with IFOs, and hence any optimization that reduces the number of calls is most beneficial to IFOs.

Tag and Tuple Fusion (A1). The `TTF` column shows the speed-up when tag and tuple fusion is enabled. We see a significant performance increase with speed-ups typically

Table 1. Impact of Performance Optimizations.

Name	Old	New	Baseline	PM	TTF*	OV*	NT*	All
Bench01	3.6ms	1.4x	7.4ms	1.8x	1.2x	1.0x	1.0x	2.3x
Bench02	7.5ms	1.4x	9.5ms	1.9x	1.3x	1.0x	1.0x	2.5x
Bench03	3.4ms	2.0x	7.4ms	1.9x	1.5x	1.1x	1.0x	2.7x
Bench04	4.7ms	1.4x	12.0ms	2.4x	1.3x	1.0x	1.0x	3.0x
Bench05	8.0ms	0.9x	10.8ms	1.8x	1.2x	1.0x	1.0x	2.3x
Bench06	10.1ms	1.4x	23.2ms	2.0x	1.3x	1.0x	1.1x	2.7x
Bench07	6.3ms	1.4x	14.7ms	2.0x	1.4x	1.0x	1.0x	2.8x
Bench08	3.7ms	1.4x	10.9ms	2.4x	1.2x	1.0x	1.0x	3.0x
Bench09	4.5ms	1.4x	6.3ms	1.9x	1.4x	1.0x	1.0x	2.5x
Bench10	3.7ms	2.3x	10.1ms	1.9x	0.5x	1.0x	1.0x	0.9x
Bench11	10.9ms	1.2x	19.2ms	1.7x	1.3x	1.0x	1.0x	2.3x
Bench12	14.4ms	1.5x	31.1ms	2.0x	1.4x	1.0x	1.0x	2.8x
Bench13	5.5ms	0.5x	16.0ms	6.7x	1.5x	1.0x	1.0x	10.0x
Bench14	5.0ms	2.4x	35.5ms	3.2x	1.0x	1.0x	1.0x	3.6x
Bench15	11.9ms	1.2x	22.1ms	1.8x	1.4x	1.0x	1.0x	2.5x
Bench16	17.1ms	1.3x	34.1ms	1.8x	1.4x	1.1x	1.0x	2.7x
Bench17	3.1ms	1.2x	4.6ms	1.4x	1.4x	1.1x	1.0x	2.0x
Bench18	5.2ms	2.7x	12.9ms	1.0x	0.7x	1.0x	1.0x	0.7x
Bench19	10.8ms	1.2x	19.6ms	1.6x	1.2x	1.0x	1.2x	2.2x
Bench20	7.2ms	1.9x	21.9ms	1.7x	1.2x	1.0x	1.2x	2.4x

between 1.2x and 1.4x. Interestingly, in two cases, the tag and tuple fusion optimization causes a slow down of between 0.5x and 0.7x. We plan to investigate this further in the future.

Optional Values (A2). The `OV` column shows the speed-up when the optional values representation is enabled. As it turns out, the optimization is not applicable to most of the benchmarks. For the few benchmarks where it makes a difference, the performance increase is very modest, around a 1.1x speed-up.

New Types (A3). The `NT` column shows the speed-up when the new type optimization is enabled. As with the optional values representation, this optimization is rarely applicable. For the few benchmarks where it makes a difference, the performance increase is modest, around a 1.2x speed-up.

Polymorphic Code and Data (A4). We did not directly evaluate the impact of monomorphization as we do not have a version of the Flix compiler that supports erasure.

All Optimizations. The `All` column shows the speed-up, over the baseline, with all optimizations enabled. The typical speed-up is between 2.3x to 2.8x. In two cases, the optimizations cause a slow down of 0.7x to 0.9x. We attribute this to the slow-down caused by tag and tuple fusion.

4.4.3 Q3: Performance Comparison

We wanted to compare the performance of Flix, with its contextual IFOs and optimizations, to Scala, an established programming language widely used in industry.

Let us briefly describe how Scala addresses the challenges (A1–A5) and (B). For (A1), Scala does not have algebraic data types *per se* but rather *case classes* which fuse the representation of tags and tuples, but are less flexible than our design, as described earlier. For (A2) and (A3), optional and new types values are always boxed. For (A4), Scala generally performs type erasure for polymorphic code and data. For (A5), both Flix and Scala compile pattern matches to labels and jumps. And finally, for (B) Scala does not support full tail call elimination.

Table 2. Performance Comparison for Flix and Scala.

	Scala		Flix	
Name	**Imperative**	**Functional**	**Unoptimized**	**Optimized**
Bench01	2.1ms	1.3x	3.6x	1.5x
Bench02	2.4ms	1.4x	4.0x	1.6x
Bench03	2.1ms	1.2x	3.6x	1.3x
Bench04	2.7ms	1.2x	4.4x	1.5x
Bench05	2.8ms	1.4x	3.9x	1.8x
Bench06	6.1ms	1.2x	3.9x	1.5x
Bench07	3.3ms	1.6x	4.5x	1.7x
Bench08	3.1ms	0.7x	3.5x	1.2x
Bench09	2.7ms	1.1x	2.3x	0.9x
Bench10	3.1ms	1.2x	3.3x	3.5x
Bench11	5.3ms	1.5x	3.7x	1.6x
Bench12	6.7ms	1.5x	4.7x	1.7x
Bench13	1.3ms	1.5x	12.2x	1.2x
Bench14	2.9ms	0.9x	12.2x	3.7x
Bench15	-	-	-	-
Bench16	7.6ms	1.5x	4.5x	1.7x
Bench17	1.9ms	1.2x	2.4x	1.3x
Bench18	-	-	-	-
Bench19	4.0ms	1.4x	5.0x	2.2x
Bench20	4.0ms	1.4x	5.5x	2.2x

As described earlier, we implemented 20 small benchmarks in Flix and Scala. We implemented two versions of the Scala benchmarks, an apples-to-apples version and an apples-to-oranges version. The apples-to-apples version is written in idiomatic functional style whereas the apples-to-oranges version uses the Scala standard library which internally uses impure features, such as mutation and loops.

The results of this experiment are shown in Table 2. The table is divided into Scala and Flix. For Scala, the two columns `Imperative` and `Functional` correspond to the apples-to-oranges and apples-to-apples implementations. For Flix, the two columns are without and with all optimizations. We use the `Imperative` column as the baseline, since it is the fastest. Two benchmarks had to be omitted due to missing functionality in the Scala standard library.

For example, for benchmark 1, the impure Scala program took 2.1ms, the functional Scala program took 1.3x times longer, the unoptimized Flix program took 3.6x times longer, and the optimized Flix program took 1.5x times longer.

The data shows two interesting things: imperative Scala programs are faster than their functional counterparts and the overhead of optimized Flix programs is comparable to functional Scala programs. Moreover, the overhead of Flix programs to impure Scala programs is typically between 1.3x and 2.2x. We believe many programmers will accept this overhead as an acceptable cost for full tail call elimination.

Table 3. The Computer Language Benchmarks Game.

Name	Lines of Code	Scala	Flix	Slowdown
Binary Trees	45	86ms	217ms	2.5x
Fibonacci	5	59ms	103ms	1.7x
Pi Digits	150	129ms	246ms	1.9x
NBody	35	48ms	48ms	1.0x

To further investigate the performance of Flix and Scala, we ported 4 programs from the Computer Language Benchmarks Game. We selected these benchmarks as they were the easiest to implement in both languages. The programs were ported over in a functional style, to enable an apples-to-apples comparison.

Figure 3 shows the results of this experiment. We observe slow-downs of between 1.0x to 2.5x. We make a few observations about these programs: The computation they performed is mostly based on lots of function calls coupled with various types of arithmetic, e.g. floating-point, integer, or big-integer operations. For example, the Fibonacci and Binary Trees programs perform lots of function calls and hence experience the overhead of IFOs. The other benchmarks experience less of a slow-down as their execution time is not only dominated by function calls, but also by the arithmetic. Thus for the NBody experiment the cost of function calls becomes so insignificant that Flix and Scala are on par.

5 Related Work

We discuss two types of related work: (a) compilation of functional programming languages to the JVM, and (b) tail call elimination strategies for the JVM.

5.1 Funtional Languages on the JVM

Benton et al. [1998] present MLJ, a compiler from Standard ML to JVM bytecode. The compiler is a whole-program optimizing compiler that performs monomorphization, similar to Flix. In their compiler, a major design goal and significant challenge is to ensure that the generated code is small such that it can be quickly transmitted over the internet. The paper lists three reasons why monomorphization does not blow up the code size: (i) the specialization is performed with respect to JVM types and not ML types, (ii) the alternative, to use boxing and unboxing requires additional code itself, and (iii) polymorphic functions tend to be small and can often be inlined and then removed. The MLJ compiler performs several optimizations to improve performance, including uncurrying, expansion of functions that take or return tuples, and elimination of the unit value. The compiler also uses more efficient representations of algebraic data types. For example, optional values are unboxed and pure enumerations are compiled to a single integer. The paper does not evaluate the performance impact of these choices.

Bothner [1998] presents Kawa, a framework for implementation of dynamic programming languages on the JVM. Kawa has been used to implement ECMAScript and Scheme. The work addresses two pain points for functional languages: the representation of closures and continuations. Kawa represents closures with a single abstract class that defines several apply methods `apply1`, `apply2`, and so on. Each closure and function is implemented as a subclass that overrides the appropriate apply method. The formal arguments of each apply method are of type `Object` and hence primitives must be boxed. Closures capture a reference to the current environment, which is a function object, and unlike most other approaches, these environments are linked. Hence, to retrieve a closure-captured variable may require traversal of several function objects. Kawa supports a limited form of continuations which are implemented using exceptions.

MacKenzie and Wolverson [2003] present Camelot, a functional language in the ML-family with explicit heap management, and Grail, an intermediate representation for functional programs with resource constraints. Camelot supports *diamonds* which are annotations that inform the compiler when a resource, typically a memory cell, can safely be reused. For example, this allows reuse of cons cells when mapping over a list, if the compiler knows that the original input list will no longer be used. Camelot compiles to JVM bytecode via Grail. An interesting design choice is that tagged unions (i.e. sum types) are represented as a *single* class to maximize the potential for memory reuse. This makes sense, since if most data share the same representation, then the chance that some representation can be reused is much higher. The authors briefly mention tail call elimination, writing: *"... Unfortunately, all of these strategies* [trampolines, etc.] *tend to require extra heap usage and thus compromise the transparency of the compilation process..."* which is a problem we can now overcome with IFOs or contextual IFOs, which are allocation free.

Clerc [2012] presents `ocamlwrap`, a compiler from OCaml source code to JVM bytecode. The compilation strategy is straightforward: Each OCaml value is represented as an object of some class. Primitive values are boxed. Each tuple is represented as an object of a class specialized to the arity of the tuple, similarly for functions and closures. Tags are represented using a *single* class with an integer field to distinguish the specific tag.

Wimmer and Würthinger [2012] present Truffle and Graal, a programming language implementation framework for the JVM. Using Truffle, the language implementor writes a simple AST-based interpreter. Truffle runs the interpreter and collects profiling and type information about the program. This information is then used to perform AST-rewritings to optimize the execution. Once the AST, or a sub-AST, reaches a stable state, Graal compiles the AST directly to JVM bytecode. For future work, we think it would be interesting to try to extend Truffle and Graal with tail call elimination strategies, e.g. trampolines and contextual IFOs.

5.2 Tail Call Elimination

Tauber et al. [2015] present a technique for tail call elimination based on impure functional objects (IFOs). The idea is that each function is represented as an object that has fields for each of its arguments and a field for its return value. Each of these objects, the IFOs, are allocated when the program begins execution. In addition to IFOs, the program has a single static field that holds a reference to the current continuation, i.e. the next IFO to evaluate. At a tail call, the program writes the arguments to the fields of the IFO and writes the IFO to the continuation field. At a non tail call, the program works like a trampoline and repeatedly evaluates the IFO stored in the continuation field. The key benefit of the technique is that tail calls do not require object allocation and hence the memory usage is constant. A challenge with IFOs, not adequately adressed in the original paper, is how to ensure thread-safety. In this paper, we have proposed two thread-safe variants of IFOs and evaluated their performance.

Schinz and Odersky [2001] present a technique for tail call elimination on the JVM called *stack shrinking*. The idea is that each method receives an extra argument, the *tail call counter* (TCC), an integer which tracks the current number of tail calls on the stack. Intuitively, every tail call increments the current counter whereas every non-tail call passes zero. Upon entering a method, the counter is checked against a predefined limit, the *tail call limit* (TCL), if the counter exceeds the limit the stack is shrunk. The paper presents two ways to shrink the stack, either of which must capture the current continuation: (i) returning a special continuation object down all calls on the stack, or (ii) throwing a special exception object with the continuation. In either case, the continuation is "caught" by a trampoline that sits at the bottom of the current tail calls which can resume execution. We can understand stack shrinking as a technique that is in the design space between regular method calls and trampolines. Conceptually, when the tail call limit (TCL) is infinite the execution corresponds to regular method calls (which may require unbounded stack size) whereas when the TCL is zero the execution corresponds to trampolining. Stack shrinking is a hybrid in between these two extremes. As one author eloquently put it: "You can avoid making many small jumps by occassionally jumping of a cliff."

Techniques for tail call elimination have also been studied for logic languages [Codognet et al. 1995; Morales et al. 2004, 2012; Ross et al. 1999]. Ross et al. present techniques for tail call elimination for the Mercury language and Codognet et al. discuss compilation of Prolog to C. A scheme similar to IFOs, but realized for C, has been used in the Ciao Prolog engine [Morales et al. 2004].

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

6 Conclusion

In this paper, we have worked on two overall challenges for implementing functional programming languages on the JVM: (A) how to represent algebraic data types, and (B) how to support full tail call elimination on the JVM. For (A), we identified five sub-challenges related to efficient representation and compilation of tagged tuples (A1), optional values (A2), new type values (A3), polymorphic data types and functions (A4), and pattern matches (A5).

We have proposed two novel techniques: *tag tuple fusion*, which allows separation of tags and tuples at the language level while compiling to a fused representation (challenge A1) and *contextual impure functional objects*, a thread-safe variant of [Tauber et al. 2015]'s IFO's for full tail call elimination on the JVM (challenge B).

We have implemented these techniques along with known techniques for the challenges (A2–A5) in the Flix compiler and evaluated their performance. The results show that significant speed-ups are attainable, the cost of full tail call elimination is reasonable, and that for functional programs the performance of Flix is not too far from Scala.

References

Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *ICFP*.

Runar Oli Bjarnason. 2012. Stackless Scala With Free Monads.

Per Bothner. 1998. Kawa: Compiling Scheme to Java.

Paul Chiusano and Rúnar Bjarnason. *Functional Programming in Scala.*

Xavier Clerc. 2012. OCaml-Java: OCaml on the JVM. In *TFP*.

Philippe Codognet, Daniel Diaz, et al. 1995. WAMCC: Compiling Prolog to C. In *ICLP*.

Simon Peyton Jones and David Lester. *Implementing Functional Languages.*

Kenneth MacKenzie and Nicholas Wolverson. 2003. Camelot and Grail: Resource-aware Functional Programming for the JVM. In *TFP*.

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016a. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *PLDI*.

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016b. Programming a Dataflow Analysis in Flix. In *TAPAS*.

J Morales, Manuel Carro, and Manuel Hermenegildo. 2004. Improved Compilation of Prolog to C using Moded Types and Determinism Information. *PADL* (2004).

Jose F Morales, Rémy Haemmerlé, Manuel Carro, and Manuel V Hermenegildo. 2012. Lightweight compilation of (C)LP to JavaScript. *ICLP* (2012).

Peter Ross, David Overton, and Zoltan Somogyi. 1999. Making Mercury Programs Tail Recursive. In *LOPSTR*.

Michel Schinz and Martin Odersky. 2001. Tail Call Elimination on the Java Virtual Machine. In *BABEL*.

Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. 2009. Lazy Continuations for Java Virtual Machines. In *PPPJ*.

Tomáš Tauber, Xuan Bi, Zhiyuan Shi, Weixin Zhang, Huang Li, Zhenrui Zhang, and Bruno Oliveira. 2015. Memory-Efficient Tail Calls in the JVM with Imperative Functional Objects. In *APLAS*.

Dean Wampler and Alex Payne. *Programming Scala: Scalability = Functional Programming + Objects.*

Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *SPLASH*.