Memory-efficient Tail Calls in the JVM with Imperative Functional Objects

Tomáš Tauber[1], Xuan Bi[1], Zhiyuan Shi[1], Weixin Zhang[1], Huang Li[2], Zhenrui Zhang[2], Bruno C. d. S. Oliveira[1]

[1] The University of Hong Kong, Pok Fu Lam Road, Hong Kong SAR
[2] Zhejiang University, 38 Zheda Road, Hangzhou, China

Abstract. This paper presents **FCore**: a JVM implementation of System F with support for full *tail-call elimination* (TCE). Our compilation technique for **FCore** is innovative in two respects: it uses a new representation for first-class functions called *imperative functional objects*; and it provides a way to do TCE on the JVM using constant space.
Unlike conventional TCE techniques on the JVM, allocated function objects are reused in chains of tail calls. Thus, programs written in **FCore** can use idiomatic functional programming styles, relying on TCE, and perform well without worrying about the JVM limitations. Our empirical results show that programs which use tail calls can run in constant space and with low execution time overhead when compiled with **FCore**.

1 Introduction

A runtime environment, such as the JVM, attracts both functional programming (FP) languages' compiler writers and users: it enables cross-platform development and comes with a large collection of libraries and tools. Moreover, FP languages give programmers on the JVM other benefits: simple, concise and elegant ways to write different algorithms; high code reuse via higher-order functions; and more opportunities for parallelism, by avoiding the overuse of side-effects (and shared mutable state) [2]. Unfortunately, compilers for functional languages are hard to implement efficiently in the JVM. FP promotes a programming style where *functions are first-class values* and *recursion* is used instead of mutable state and loops to define algorithms. The JVM is not designed to deal with such programs.

The difficulty in optimizing FP in the JVM means that: *while FP in the JVM is possible today, some compromises are still necessary for writing efficient programs.* Existing JVM functional languages, including Scala [16] and Clojure [10], usually work around the challenges imposed by the JVM. Those languages give programmers alternatives to a FP style. Therefore, performance-aware programmers avoid certain idiomatic FP styles, which may be costly in those languages, and use the available alternatives instead.

In particular, one infamous challenge when writing a compiler for a functional language targeting the JVM is: How to eliminate and/or optimize tail calls? Before tackling that, one needs to decide how to represent functions in the JVM. There are two standard options: *JVM methods* and *functions as objects* (FAOs). Encoding first-class functions using only JVM methods directly is limiting: JVM methods cannot encode currying and partial function application directly. To support these features, the majority of functional languages or extensions (including Scala, Clojure, and Java 8) adopt variants of the functions-as-objects approach:

```
interface FAO { Object apply(Object arg);}
```

With this representation, we can encode curried functions, partial application and pass functions as arguments. However, neither FAOs nor JVM methods offer a good solution to deal with *general tail-call elimination* (TCE) [22]. The JVM does not support proper tail calls. In particular scenarios, such as single, tail-recursive calls, we can easily achieve an optimal solution in the JVM. Both Scala and Clojure provide some support for tail-recursion [16,11]. However, for more general tail calls (such as mutually recursive functions or non-recursive tail calls), existing solutions can worsen the overall performance. For example, JVM-style trampolines [19] (which provide a general solution for tail calls) are significantly slower than normal calls and consume heap memory for every tail call.

Contributions. This paper presents a new JVM compilation technique for functional programs, and creates an implementation of System F [9,18] using the new technique. The compilation technique builds on a new representation of first-class functions in the JVM: *imperative functional objects* (IFOs). *With IFOs it is possible to use a single representation of functions in the JVM and still achieve memory-efficient TCE.* As a first-class function representation, IFOs also support currying and partial function applications.

We represent an IFO by the following abstract class:

```
abstract class Function {
  Object arg, res;
  abstract void apply();
}
```

With IFOs, we encode both the argument (`arg`) and the result of the functions (`res`) as mutable fields. We set the argument field before invoking the `apply()` method. At the end of the `apply()` method, we set the result field. An important difference between the IFOs and FAOs encoding of first-class functions is that, in IFOs, *function application is divided in two parts*: *setting the argument field*; and *invoking the apply method*. For example, if we have a function call `factorial 10`, the corresponding Java code using IFOs is:

```
factorial.arg = 10; // setting argument
factorial.apply(); // invoking function
```

The fact that we can split function application into two parts is key to enable new optimizations related to functions in the JVM. In particular, the TCE approach with IFOs does not require memory allocation for each tail call and has less execution time overhead than the JVM-style trampolines used in languages such as Clojure and Scala. Essentially, with IFOs, it is possible to provide a straightforward TCE implementation, resembling Steele's "UUO handler" [23], in the JVM.

Using IFOs and the TCE technique, we created **FCore**: a JVM implementation of an extension of *System F*. **FCore** aims to serve as an intermediate functional layer on top of the JVM, which ML-style languages can target. According to our experimental results, **FCore** programs perform competitively against programs using regular JVM methods, while still supporting TCE. Programs in **FCore** tend to have less execution time overhead and use less memory than programs using conventional JVM trampolines.

In summary, the contributions of this paper are:

- **Imperative Functional Objects:** A new representation of first-class functions in the JVM, offering new ways to optimize functional programs.
- **A memory-efficient approach to tail-call elimination:** A way to implement TCE in the JVM using IFOs without allocating memory per each tail call.
- **FCore**: An implementation of a System F-based intermediate language that can be used to target the JVM by FP compilers.
- **Formalization and empirical results:** Our basic compilation method from a subset of **FCore** into Java is formalized. Our empirical results indicate that **FCore** allows general TCE in constant memory space and with execution time comparable to regular JVM methods.

2 FCore and IFOs, Informally

This section informally presents **FCore** programs and their IFO-based encoding and how to deal with tail-call elimination. Sections 3 and 4 present a formalized compilation method for a subset of **FCore** (System F) into Java, based on the ideas from this section. Note that, for purposes of presentation, we show slightly simplified encodings in this section compared to the formal compilation method.

2.1 Encoding Functions with IFOs

In **FCore**, we compile all functions to classes extending the `Function` class presented in Section 1. For example, consider a simple identity function on integers. In **FCore** or System F (extended with integers), we represent it as follows:

$id \equiv \lambda(x : Int).\ x$

We can manually encode this definition with an IFO in Java as follows:

```
class Id extends Function {
   public void apply () {
      final Integer x = (Integer) this.arg;
      res = x;
   }
}
```

The `arg` field encodes the argument of the function, whereas the `res` field encodes the result. Thus, to create the identity function, all we need to do is to copy the argument to the result. A function invocation such as *id* 3 is encoded as follows:

```
Function id = new Id();
id.arg = 3; // setting argument
id.apply(); // invoking apply()
```

The function application goes in two steps: it first sets the `arg` field to `3` and then invokes the `apply()` method.

Curried Functions. IFOs can naturally define curried functions, such as:

$$constant \equiv \lambda(x : Int).\ \lambda(y : Int).\ x$$

Given two integer arguments, this function will always return the first one. Using IFOs, we can encode *constant* in Java as follows:

```
class Constant extends Function {
   public void apply () {
      final Integer x = (Integer) this.arg;
      class IConstant extends Function {
         public void apply() {
            final Integer y = (Integer) this.arg;
            res = x;
         }
      }
      res = new IConstant();
   }
}
```

Here, the first lambda function sets the second one as its result. The definition of the second `apply` method sets the result of the function to the argument of the first lambda function. The use of inner classes enforces the lexical scoping of functions. We encode an application such as *constant* 3 4 as:

```
Function constant = new Constant();
constant.arg = 3;
constant.apply();
Function f = (Function) constant.res;
f.arg = 4;
f.apply();
```

We first set the argument of the `constant` function to 3. Then, we invoke the `apply` method and store the resulting function to a variable `f`. Finally, we set the argument of `f` to 4 and invoke `f`'s `apply` method. Note that the alias `x` for **`this`**`.arg` is needed to prevent accidental overwriting of arguments in partial applications. For example in *constant* 3 (*constant* 4 5), the inner application *constant* 4 5 would overwrite 3 to 4 and the outer one would incorrectly return 4.

Partial Function Application. With curried functions, we can encode partial application easily. For example, consider the following expression: *three* $\equiv$

```
// tail-call elimination
class Mutual {
 Function teven;
 Function todd;
 class TEven extends Function {
  public void apply () {
   final Integer n =
    (Integer) this.arg;
   if (n == 0) {
    res = true;
   }
   else {
    todd.arg = n - 1;
    // tail call
    Next.next = todd;
   }
  }
 }
 class TOdd extends Function {
  public void apply () {
   final Integer n =
    (Integer) this.arg;
   if (n == 0) {
    res = false;
   }
   else {
    teven.arg = n - 1;
    // tail call
    Next.next = teven;
   }
  }
 }
 { // initialization block
  todd = new TOdd();
  teven = new TEven();
 }
}
```

Fig. 1. Functions `even` and `odd` using IFOs with tail-call elimination

constant 3. The code for this partial application is simply:

```
Function constant = new Constant();
constant.arg = 3;
constant.apply();
```

Recursion. **FCore** supports simple recursion, as well as mutual recursion. For example, consider the functions *even* and *odd* defined to be mutually recursive:

$even \equiv \lambda(n : Int).\ \textbf{if}\ (n = 0)\ \textbf{then}\ true\ \textbf{else}\ odd(n - 1)$
$odd \equiv \lambda(n : Int).\ \textbf{if}\ (n = 0)\ \textbf{then}\ false\ \textbf{else}\ even(n - 1)$

These two functions define a naive algorithm for detecting whether a number is even or odd. We can encode recursion using Java's own recursion: the Java references `even` and `odd` are themselves mutually recursive (Figure 1).

2.2 Tail-call Elimination

The recursive calls in *even* and *odd* are tail calls. IFOs present new ways for doing tail-call elimination in the JVM. The key idea, inspired by Steele's work on encoding tail-call elimination [23], is to use a simple auxiliary structure

```
class Next {static Function next = null;}
```

that keeps track of the next call to be executed. Figure 1 illustrates the use of the `Next` structure. This is where we make a fundamental use of the fact that function application is divided into two parts with IFOs. In tail calls, we set the arguments of the function, but we delay the `apply` method calls. Instead, the

```
// TCE with JVM-style trampolines
interface Thunk {
 Object apply();
}
...
 static Object teven(final int n) {
   if(n == 0) return true;
   else return new Thunk() {
     public Object apply() {
       return todd(n-1);
     }
   };
 }
 static Object todd(final int n) {
   if(n == 0) return false;
   else return new Thunk() {
     public Object apply() {
       return teven(n-1);
     }
   };
...
Object trampoline = even(10);
while(trampoline instanceof Thunk)
     trampoline =
     ((Thunk) trampoline).apply();
return (Boolean) trampoline;
```

```
// TCE with IFOs + Next
Mutual m = new Mutual();
Function teven = m.teven;
...
teven.arg = 10;
Next.next = teven;
Function c;
Boolean res;
do {
  c = Next.next;
  Next.next = null;
  c.apply();
} while (Next.next != null);
res = (Boolean) c.res;
```

Fig. 2. This figure contrasts the TCE approach with JVM-style trampolines (left, custom implementation) and with IFOs and the Next handler (right, see Figure 1 for implementation).

`next` field of `Next` is set to the function with the `apply` method. The `apply` method is then invoked at the call-site of the functions. The code in Figure 2 illustrates the call `even 10`. In JVM-style trampolines, each (method) call creates a Thunk. IFOs, however, are reused throughout the execution. The idea is that a function call (which is not a tail-call) has a loop that jumps back-and-forth into functions. The technique is similar to some trampoline approaches in C-like languages. However, an important difference to JVM-style trampolines is that utilization of heap space is not growing. In other words, tail-calls do not create new objects for their execution, which improves memory and time performance. Note that this method is *general*: it works for *simple recursive tail calls*, *mutually recursive tail calls*, and *non-recursive tail calls*.

3 Compiling FCore

This section formally presents **FCore** and its compilation to Java. **FCore** is an extension of System F (the polymorphic λ-calculus) [9,18] that can serve as a target for compiler writers.

Syntax. In this section, for space reasons, we cover only the **FCore** constructs that correspond exactly to System F. Nevertheless, the constructs in System F represent the most relevant parts of the compilation process. As discussed in Section 5.1, our implementation of **FCore** includes other constructs that are needed to create a practical programming language.

System F. The basic syntax of System F is:

$$\begin{array}{ll} \textbf{Types} & \tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha . \tau \\ \textbf{Expressions} & e ::= x \mid \lambda(x : \tau).e \mid e_1\ e_2 \mid \Lambda \alpha . e \mid e\ \tau \end{array}$$

Types τ consist of type variables α, function types $\tau_1 \rightarrow \tau_2$, and type abstraction $\forall \alpha . \tau$. A lambda binder $\lambda(x : \tau).e$ abstracts expressions e over values (bound by a variable x of type τ) and is eliminated by function application $e_1\ e_2$. An expression $\Lambda \alpha . e$ abstracts an expression e over some type variable α and is eliminated by a type application $e\ \tau$.

From System F to Java. Figure 3 shows the type-directed translation rules that generate Java code from given System F expressions. We exploit the fact that System F has an erasure semantics in the translation. This means that type abstractions and type applications do not generate any code or have any overhead at run-time.

We use two sets of rules in our translation. The first one is translating System F expressions. The second set of rules, the function $\langle \tau \rangle$, describes how we translate System F types into Java types.

In order to do the translation, we need *translation environments*:

$$\Gamma ::= \epsilon \mid \Gamma\ (x_1 : \tau \mapsto x_2) \mid \Gamma \alpha$$

Translation environments have two purposes: 1) to keep track of the type and value bindings for type-checking purposes; 2) to establish the mapping between System F variables and Java variables in the generated code.

The translation judgment in the first set of rules adapts the typing judgment of System F:

$\Gamma \vdash e : \tau \rightsquigarrow J$ **in** S

It states that System F expression e with type τ results in Java expression J created after executing a block of statements S with respect to translation environments Γ. `FJ-Var` checks whether a given value-type binding is present in an environment and generates a corresponding, previously initialized, Java variable. `FJ-TApp` resolves the type of an abstraction and substitutes the applied type in it. `FJ-TAbs` translates the body of type abstractions – note that, in the extended language, type abstractions would need to generate suspensions. `FJ-Abs` translates term abstractions. For translating term abstractions, we need evidence for resolving the body e and a bound variable x of type τ_1. We then wrap the generated expression J and its deriving statements S as follows. We create a class with a fresh name *FC*, extending the *Function* class. In the body of `apply`, we first create an alias for the function argument with a fresh name

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow J \textbf{ in } S}$

$$(\text{FJ-Var})\ \frac{(x_1 : \tau \mapsto x_2) \in \Gamma}{\Gamma \vdash x_1 : \tau \rightsquigarrow x_2 \textbf{ in } \{\}} \qquad (\text{FJ-TApp})\ \frac{\Gamma \vdash e : \forall \alpha.\tau_2 \rightsquigarrow J \textbf{ in } S}{\Gamma \vdash e\,\tau_1 : \tau_2[\tau_1/\alpha] \rightsquigarrow J \textbf{ in } S}$$

$$(\text{FJ-TAbs})\ \frac{\Gamma, \alpha \vdash e : \tau \rightsquigarrow J \textbf{ in } S}{\Gamma \vdash \Lambda\alpha.e : \forall \alpha.\tau \rightsquigarrow J \textbf{ in } S}$$

$$(\text{FJ-Abs})\ \frac{\begin{array}{c}\Gamma, x : \tau_1 \mapsto y \vdash e : \tau_2 \rightsquigarrow J \textbf{ in } S_1 \\ f,\ y\ ,\ FC\ fresh\end{array}}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2 \rightsquigarrow f \textbf{ in } S_2}$$

```
S2 := {
  class FC extends Function {
     void apply() {
       ⟨T1⟩ y = (⟨T1⟩) this.arg;
       S1;
       res = J;
     }
  };
  Function f = new FC();}
```

$$(\text{FJ-App})\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow J_1 \textbf{ in } S_1 \\ \Gamma \vdash e_2 : \tau_2 \rightsquigarrow J_2 \textbf{ in } S_2 \qquad f,\ x_f\ fresh\end{array}}{\Gamma \vdash e_1\,e_2 : \tau_1 \rightsquigarrow x_f \textbf{ in } S_1 \uplus S_2 \uplus S_3}$$

```
S3 := {
   Function f = J1;
   f.arg = J2;
   f.apply();
   ⟨T1⟩ xf = (⟨T1⟩) f.res;}
```

Translation of System F types to Java types:

$$\begin{aligned}\langle\alpha\rangle &= \texttt{Object} \\ \langle\forall\alpha.\tau\rangle &= \langle\tau\rangle \\ \langle\tau_2 \to \tau_1\rangle &= \texttt{Function}\end{aligned}$$

Fig. 3. Type-Directed Translation from System F to Java

y, then execute all statements S_1 deriving its resulting Java expression J that we assign as the output of this function. Following that, we create a fresh alias f for the instance of the mentioned function, representing the class FC. `FJ-App` is the most vital rule. Given the evidence that e_1 is a function type, we generate a fresh alias f for its corresponding Java expression J_1. The S_3 block contains statements to derive the result of the application. As described in Section 2, we split applications into two parts in IFOs. We first set the argument of f to the Java expression J_2, given the evidence resulting from e_2. Then, we call f's `apply` method and store the output in a fresh variable x_f. Before executing statements in S_3, we need to execute statements S_1 and S_2 deriving J_1 and J_2 respectively. To derive x_f, we need to execute all dependent statements: $S_1 \uplus S_2 \uplus S_3$.

Properties of the Translation. Two fundamental properties are worthwhile proving for this translation: *translation generates well-typed (cast-safe) Java programs*; and *semantic preservation*. Proving these two properties requires the static and dynamic semantics (as well as the soundness proof) of the target language (an imperative subset of Java with inner classes in our case). Unfortunately, as far as we know, the exact subset of Java that we use has not been completely formalized yet. Three possibilities exist: 1) choosing an existing Java subset formalization and emulating its missing features in the translation, 2) developing our own formalized Java subset, 3) relating the translation to complete Java semantics within matching logic [5]. Each option would require complex changes beyond this paper's scope, hence it is a part of future work.

4 Tail-call Elimination

In this section, we show how we can augment the basic translation in Section 3 to support tail-call elimination.

As shown in Figure 1, we can do TCE with IFOs. To capture this formally, we augment the `apply` method call generation, in rule `FJ-App`, with two possibilities:

1. The `apply` method is in a tail position. This means we can immediately return by setting the `next` field of the controlling auxiliary `Next` class to the current `Function` object, without calling the apply method.
2. The `apply` method is not in a tail position. This means we need to evaluate the corresponding chain of calls, starting with the current call, followed by any apply calls within it.

We need to make two changes to achieve this goal: 1) add a tail call detection mechanism; and 2) use a different way of compiling function applications.

Detecting Tail Calls. We base the detection mechanism on the tail call context from the Revised Report on Scheme [1]. When we translate a value application $e_1\ e_2$, we know that e_2 is not in a tail position, whereas e_1 may be if the current context is a tail context. In type applications and abstractions, we know they only affect types: they do not affect the tail call context. Thus, they preserve the state we entered with for translating the apply calls. In λ abstractions, we enter a new tail call context. This detection mechanism is integrated in our translation and used when compiling function applications.

Compiling Function Applications. We augment the `apply` method call generation as follows. We extend the premise of `FJ-App` to include one extra freshly generated variable c:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow J_1 \ \textbf{in}\ S_1 \\ \Gamma \vdash e_2 : \tau_2 \rightsquigarrow J_2 \ \textbf{in}\ S_2 \qquad f,\ x_f,\ c\ \mathit{fresh}\end{array}}{\Gamma \vdash e_1\, e_2 : \tau_1 \rightsquigarrow x_f \ \textbf{in}\ S_1 \uplus S_2 \uplus S_3}$$

In the conclusion, we change S_3. For tail calls, we define it as follows:

```
S₃ := {
    Function f = J₁;
    f.arg = J₂;
    Next.next = f;
}
```

Note that x_f is not bound in S_3 here. Because the result of a tail call is delayed, the result of the tail call is still not available at this point. However, this does not matter: since we are on a tail call, the variable would be immediately out of its scope anyway and cannot be used.

For non-tail calls, we initialize x_f in S_3 as the final result:

```
S₃ := {
    Function f = J₁;
    f.arg = J₂;
    Next.next = f;
    Function c;
    Object x_f;
    do {
      c = Next.next;
      Next.next = null;
      c.apply();
    } while (Next.next != null);
    x_f = c.res;
}
```

This generated code resembles the example in Section 2, except for the general `Object` x_f being in place of the specialized `Boolean res`. The idea of looping through a chain of function calls remains the same.

5 Implementation & Evaluation

5.1 Implementation

We implemented[3] a compiler for **FCore** based on the representation and type-directed translation we described in Sections 3 and 4. Our actual implementation has extra constructs, such as primitive operations, types and literals, let bindings, conditional expressions, tuples, and fixpoints. It also contains constructs for a basic Java interoperability. The compiler performs other common forms of

[3] **FCore** code repository: https://github.com/hkuplg/fcore

optimizations, such as optimizing multi-argument function applications, partial evaluation, inlining, and unboxing. We wrote the compiler in Haskell and the code repository contains several example programs as well as a large test suite.

5.2 Evaluation

We evaluate two questions with the respect to IFOs:

1. Do IFOs support general TCE in constant memory space?
2. What is the execution time overhead of IFOs?

The first question is assessed through measuring total allocated objects on heap in an implementation of DFA. The second question is evaluated in two parts. Firstly, we use microbenchmarks to isolate different simple call behaviors. Secondly, we come back to the DFA implementation's time performance.

# ***of Objects***	Min	Max
IFO	5451	5451
Java (T)	4665	104879
Java (M)	4128	4128
Java (FAO)	18102	24082

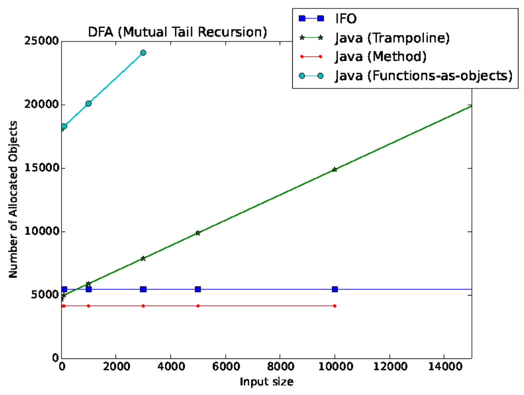

Fig. 4. The DFA encoding: the two columns show the minimum and maximum numbers of total allocated objects on heap from isolated profiled runs with all input lengths. Due to space limitations, the x-axes of plots are cropped at 15000 for clarity.

General TCE in Constant Memory. One common idiom in functional programming is encoding finite states as tail recursive functions and state transitions as mutual calls among these functions. One trivial example of this is the naive even-odd program which switches between two states. A more useful application is in the implementation of finite state automata [13]. Normally, functional language programmers seek this idiom for its conciseness. However in JVM-hosted functional languages, programmers tend to avoid this idiom, because they either lose correctness (`StackOverflow` exceptions in a method-based representation) or performance (in a trampoline-based one). In this experiment, we implemented a DFA recognizing a regular expression $(AAB^*|A^*B)^+$ and measured the performance on randomly generated Strings with different lengths.

We implemented it in **FCore** to assess IFOs (with all the optimizations mentioned in Sections 4 and 5.1) and in Java (1.8.0_25) to assess different closure representations: method calls, Java 8's lambdas (functions-as-objects), and custom trampolines. We chose plain Java implementation, because we can examine the runtime behavior of different representations without potential compiler overheads. All implementations used primitive **`char`** variables and did not allocated any new objects on heap when reading from the input Strings. We report the total number of allocated objects on heap in the isolated application runs, as measured by HPROF [17], the JDK's profiling tool.
We executed all benchmarks on the following platform with the HotSpot™VM (1.8.0_25): Intel®Core™i5 3570 CPU, 1600MHz DDR3 4GB RAM, Ubuntu 14.04.1.

We show the result of this experiment in Figure 4. The IFO- and trampoline-based implementations continued executing after method-based and FAO-based ones threw a `StackOverflow` exception. IFOs, similarly to the method-based implementation, allocated a constant number of objects on heap. The trampoline one, however, increased its object allocation with the input, because it needed to create an object for each tail call.

Time Overhead: Isolated Call Behavior. For measuring time overhead, we show two experiments: isolated simple call behavior in different microbenchmarks and the time performance of the DFA implementation. We wrote the benchmark programs in the extended System F for our compilation process and in the following JVM-hosted languages in their stable versions: *Scala* (2.11.2)[16], *Clojure* (1.6.0)[10], and Java (1.8.0_25, as before). For encoding mutually recursive tail calls, we used the provided trampoline facilities in Scala (`scala.util.control.TailCalls`) and Clojure (`tramp` from `clojure.core.logic`).
The programs were executed on the same platform as the memory experiment. For the automation of performance measurement, we used the Java Microbenchmark Harness (JMH) tool which is a part of OpenJDK [8]. Based on the provided annotations, JMH measures execution of given programs. In addition to that, it takes necessary steps to gain stable results. They include non-measured warm-up iterations for JITC, forcing garbage collection before each benchmark, and running benchmarks in isolated VM instances. We configured JMH for 10 warm-up runs and 10 measured runs from which we compute averages.

We chose four programs to represent the following behaviors:

– *Non-tail recursive calls*: Computing the factorial and Fibonacci numbers using naive algorithms.
– *Single method tail recursive calls*: Computing factorial using a tail recursive implementation.
– *Mutually recursive tail calls*: Testing evenness and oddness using two mutually recursive functions.

Low Input Values	fact(20) in ns	fib(20) in ns	tailfact(20) in ns	evenodd(256) in μs
IFO	204.84 ± 2.35	35.50 ± 0.47	49.52 ± 0.72	32.95 ± 0.09
Java	147.95 ± 0.65	22.50 ± 0.06	18.18 ± 0.19	30.93 ± 0.12
Java (T)	1280.23 ± 20.99	502.35 ± 9.42	139.39 ± 1.79	474.41 ± 6.29
Scala	130.46 ± 0.40	22.55 ± 0.14	15.94 ± 0.05	32.79 ± 0.09
Clojure	573.95 ± 3.41	314.24 ± 2.25	205.21 ± 0.35	82.61 ± 0.95

High Input Values	evenodd(214748) in μs	tailfact(10000) in μs
IFO	152.47 ± 0.43	166.64 ± 0.51
Java	1060.35 ± 14.52	644.10 ± 3.89
Scala	1864.34 ± 31.24	1004.13 ± 13.49
Clojure	6533.14 ± 92.65	N/A

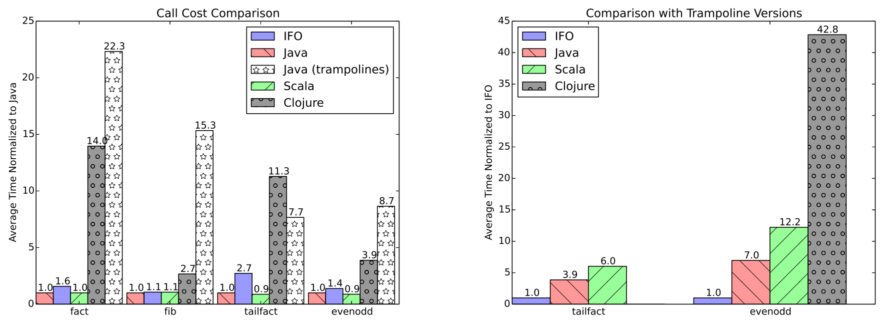

Fig. 5. The isolated call behavior experiments: the reported times are averages of 10 measured runs and corresponding standard deviations. The plots are normalized to Java's (left table and plot) and IFO's (right table and plot) results – the lower, the faster.

Non-tail recursive programs present two examples of general recursive calls and we executed them, altogether with the tail recursive programs, on low input values (not causing `StackOverflow` exceptions in default JVM settings). In addition to that, we executed the tail recursive programs on high input values in which method-based implementations threw `StackOverflow` exceptions in default JVM settings. We show the results in Figure 5. Its left part shows the result for low input values in IFOs, method implementations in all the other languages and the fastest trampoline implementation (Java); the plot is normalized to the Java method-based implementation's results. The right part shows the result for high input values in IFO- and trampoline-based implementations; the plot is normalized to results of IFO-based implementations. For low input values, we can see that IFO-based implementations run slightly slower than method-based ones. However, their overhead is small compared with the fastest trampoline implementations in our evaluation. IFOs ran 0.1 to 1.7-times slower than method-based representations, whereas the fastest trampolines ran 7.7 to 22.3-times slower. In the tail recursive programs, Scala ran slightly faster than standard Java methods due to its compiler optimizations. Clojure has an additional overhead, because its compiler enforces integer overflow checking. For the high input values, the method-based implementations threw a `StackOverflow` exception in default JVM settings, unlike IFOs and trampoline implementations which can continue executing with this input. IFOs ran 3.9 to 12.2-times faster (excluding Clojure) than trampoline implementations. Again, Clojure suffered from its additional overhead and threw an integer overflow exception in the tail recursive factorial. Using BigIntegers would prevent this, but we wanted to isolate the call behavior in this experiment, i.e. avoid any extra overhead from other object allocations.

Input length (time unit)	1000 (μs)	3000 (μs)	10000 (μs)	100000 (μs)
IFO	5.10 ± 0.10	15.98 ± 0.07	77.81 ± 0.83	933.58 ± 13.40
Java (Trampoline-based)	7.03 ± 0.130	26.89 ± 0.10	102.98 ± 2.36	1099.80 ± 15.46
Java (Method-based)	3.80 ± 0.07	11.61 ± 0.10	48.83 ± 0.13	N/A
Java (FAO-based)	6.37 ± 0.01	17.62 ± 0.05	N/A	N/A

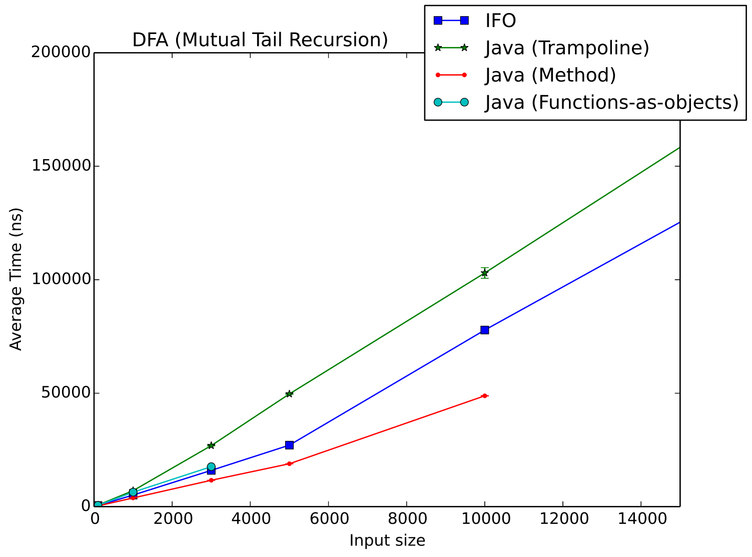

Fig. 6. The DFA encoding: the reported times are averages of 10 measured runs and corresponding standard deviations. Due to space limitations, the x-axes of plots are cropped at 15000 for clarity.

Time Overhead: DFA Performance. Unlike the first experiment, where the programs isolated costs of plain recursive calls, DFA encoding represents a more realistic behavior with other costs, such as non-recursive method calls and calls to other API methods (e.g. reading input). The setting was the same as in the constant memory experiment and we performed time measurement in the same way as in the isolated call behavior experiment. We show the result of this experiment in Figure 6. The FAO-based implementation ran slowest out of all implementations and threw `StackOverflow` exception with a smaller input than the method-based implementation. That is because it creates extra objects and performs extra calls due to its representation. As in the isolated calls experiment, the IFO-based implementation ran about 0.5-times slower than method-based implementation. Trampolines, however, ran about 2-times slower. The IFO- and trampoline-based implementations continued executing after method-based one threw a `StackOverflow` exception. The IFO-based implementation was about 0.2-times faster than the trampoline one for larger inputs.

6 Related Work

This section discusses related work: intermediate functional languages on top of the JVM, TCE and function representations, TCE on the JVM, and the JVM modifications.

Intermediate Functional Languages on top of the JVM. A primary objective of our work is to create an efficient intermediate language that targets the JVM. With such intermediate language, compiler writers can easily develop FP compilers in the JVM. System F is an obvious candidate for an intermediate language as it serves as a foundation for ML-style or Haskell-style FP languages. However, there is no efficient implementation of System F in the JVM. The only implementation of System F that we know of (for a JVM-like platform) was done by Kennedy and Syme [12]. They showed that System F can be encoded, in a type-preserving way, into .NET's C#. That encoding could easily be employed in Java or the JVM as well. However, their focus was different from ours. They were not aiming at having an efficient implementation of System F. Instead, their goal was to show that the type system of languages such as C# or Java is expressive enough to faithfully encode System F terms. They used a FAO-based approach and have not exploited the erasure semantics of System F. As a result, the encoding suffers from various performance drawbacks and cannot be realistically used as an intermediate language. MLj [4] compiled a subset of SML '97 (interoperable with Java libraries) to the Monadic Intermediate Language, from which it generated Java bytecode. Various Haskell-to-JVM compiler backends [27,25,6] used different variations of the *graph reduction machine* [26] for their code generation, whereas we translate from System F.

Tail-Call Elimination and Function Representations. A choice of a function representation plays a great role [21] in time and space efficiency as well as in how difficult it is to correctly implement tail calls. Since Steele's pioneering work on tail calls [22], implementors of FP languages often recognize TCE as a necessary feature. Steele's Rabbit Scheme compiler [23] introduced the "UUO handler" that inspired our TCE technique using IFOs. Early on, some Scheme compilers targeted C as an intermediate language and overcame the absence of TCE in the backend compiler by using trampolines. Trampolines incur on performance penalties and different techniques, with "Cheney on the M.T.A." [3] being the most known one, improved upon them. The limitations of the JVM architecture, such as the lack of control over the memory allocation process, prevent a full implementation of Baker's technique.

Tail-Call Elimination on the JVM. Apart from the recent languages, such as Scala [16] or Clojure [10], functional languages have targeted the JVM since its early versions. Several other JVM functional languages support (self) tail recursion optimization, but not full TCE. Examples include MLj [4] or Frege [28]. Later work [15] extended MLj with Selective TCE. This work used an effect system to estimate the number of successive tail calls and introduced trampolines only when necessary. Another approach to TCE in the JVM is to use an explicit stack on the heap (an `Object[]` array) [6]. With such explicit stack for TCE, the approach from Steele's pioneering work [23] can also be encoded in the JVM. Our work avoids the need for an explicit stack by using IFOs, thus allowing for a more direct implementation of this technique. The Funnel compiler for the JVM [19] used standard method calls and shrank the stack only after the execution reached a predefined "tail call limit". This dynamic optimization needs careful tuning of the parameters, but can be possibly used to further improve performance of our approach.

JVM Modifications. Proposals to modify the JVM [14], which would arguably be a better solution for improving support for FP, appeared early on. One reason why the JVM does not support tail calls was due to a claimed incompatibility of a security mechanism based on stack inspection with a global TCE policy. The abstract continuation-marks machine [7] refuted this claim. There exists one modified Java HotSpot$^{\text{TM}}$VM [20] with TCE support. The research Maxine VM with its new self-optimizing runtime system [29] allows a more efficient execution of JVM-hosted languages. Despite these and other proposals and JVM implementations, such as IBM J9, we are not aware of any concrete plans for adding TCE support to the next official JVM release. Some other virtual machines designed for imperative languages do not support TCE either. For example, the standard Python interpreter lacks it, even though some enhanced variants can overcome this issue [24]. Hence, ideas from our work can be applied outside of the JVM ecosystem.

7 Conclusion & Future Work

Functional Programming in the JVM is already possible today. However, when efficiency is a concern, programmers and compiler writers still need to be aware of the limitations of the JVM. Some of the problems are the need for two function representations; and the lack of a good solution for TCE. This paper shows that IFOs allow for a uniform representation of functions, while being competitive in terms of time performance and supporting TCE in constant space.

There is much to be done for future work. We would like to prove correctness results for our translation from System F to Java. To achieve this, we will first need a suitable formalization of Java that includes inner classes and imperative features. Furthermore, 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. One other aspect is with currying and partial applications, where the uniform function representation is important. FAOs here can have substantial time and memory overheads, especially when defining multi-argument recursive functions, so current languages tend to avoid them and use two representations: JVM methods when possible; and FAOs when necessary. With additional optimizations in **FCore**, such as multi-argument closure optimization and unboxing, IFOs serve as one uniform efficient function representation. We would like to formalize and refine a number of optimizations that we have been experimenting with in **FCore**; and explore what other optimizations are possible with IFOs. Finally, we want to build frontends for realistic functional languages on top of **FCore** and write large functional programs, including a full bootstrapping compiler of **FCore**, in those frontends.

Acknowledgments. We would like to thank T. H. Tse, C.-L. Wang, Vlad Ureche, Rúnar Bjarnason, and **FCore** contributors for help and valuable feedback on previous drafts of this work. We also thank the anonymous reviewers for their helpful suggestions. The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. HKU 27200514).

References

1. Abelson, H., Dybvig, R., Haynes, C., Rozas, G., Adams, N.I., I., Friedman, D., Kohlbecker, E., Steele, G.L., J., Bartley, D., Halstead, R., Oxley, D., Sussman, G., Brooks, G., Hanson, C., Pitman, K., Wand, M.: Revised^5 Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation 11(1) (1998)
2. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
3. Baker, H.G.: CONS should not CONS its arguments, part II. ACM SIGPLAN Notices 30(9), 17–20 (Sep 1995)
4. Benton, N., Kennedy, A., Russell, G.: Compiling Standard ML to Java Bytecodes. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (1998)
5. Bogdanas, D., Roşu, G.: K-Java: A complete semantics of Java. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 445–456. POPL '15, ACM, New York, NY, USA (2015)
6. Choi, K., Lim, H.i., Han, T.: Compiling lazy functional programs based on the spineless tagless G-machine for the Java virtual machine. Functional and Logic Programming (2001)
7. Clements, J., Felleisen, M.: A tail-recursive machine with stack inspection. ACM Transactions on Programming Languages and Systems 26(6), 1029–1052 (Nov 2004)
8. Friberg, S., Shipilev, A., Astrand, A., Kuksenko, S., Loef, H.: OpenJDK: jmh (2014), `openjdk.java.net/projects/code-tools/jmh/`

9. Girard, J.Y.: Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Université Paris VII (1972)
10. Hickey, R.: The Clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages (2008)
11. Hickey, R.: Recur construct, Clojure documentation (2014), `clojuredocs.org/clojure.core/recur`
12. Kennedy, A., Syme, D.: Transposing F to C#: expressivity of parametric polymorphism in an object-oriented language. Concurrency and Computation: Practice and Experience 16(7), 707–733 (2004)
13. Krishnamurthi, S.: Educational Pearl: Automata via Macros. Journal of Functional Programming 16(3) (2006)
14. League, C., Trifonov, V., Shao, Z.: Functional Java Bytecode. Proceedings 5^{th} World Conference on Systemics, Cybernetics, and Informatics (2001)
15. Minamide, Y.: Selective Tail Call Elimination. In: Proceedings of the 10^{th} International Conference on Static Analysis (2003)
16. Odersky, M.: The Scala Language Specification, Version 2.9. École Polytechnique Fédérale de Lausanne (2014)
17. O'Hair, K.: HPROF: a Heap/CPU profiling tool in J2SE 5.0. Sun Developer Network, Developer Technical Articles & Tips (2004)
18. Reynolds, J.C.: Towards a Theory of Type Structure. In: Symposium on Programming (1974)
19. Schinz, M., Odersky, M.: Tail call elimination on the Java Virtual Machine. Electronic Notes in Theoretical Computer Science 59(1), 158–171 (Nov 2001)
20. Schwaighofer, A.: Tail Call Optimization in the Java HotSpot^{TM}VM (2009), master Thesis, Johannes Kepler Universität Linz
21. Shao, Z., Appel, A.W.: Space-efficient closure representations. ACM SIGPLAN Lisp Pointers VII(3), 150–161 (Jul 1994)
22. Steele, G.L.: Debunking the "Expensive Procedure Call"" Myth or, Procedure Call Implementations Considered Harmful or, LAMDBA: The Ultimate GOTO. In: Proceedings of the 1977 Annual Conference (1977)
23. Steele, G.L.: Rabbit: A Compiler for Scheme. Tech. rep., Massachusetts Institute of Technology (1978)
24. Tismer, C.: Continuations and stackless Python. In: Proceedings of the 8^{th} International Python Conference. vol. 1 (2000)
25. Tullsen, M.: Compiling Haskell to Java. Tech. Rep. YALEU/DCS/RR-1204, Yale University (1996)
26. Wadsworth, C.: Semantics and Pragmatics of the Lambda-Calculus. Ph.D. thesis, University of Oxford (1971)
27. Wakeling, D.: Compiling lazy functional programs for the Java Virtual Machine. Journal of Functional Programming 9(6) (1999)
28. Wechsung, I.: Frege (2014), `github.com/Frege/frege`
29. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to Rule Them All. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (2013)