FCore: From System F to Java Efficiently!

Tomáš Tauber
The University of Hong Kong
ttauber@cs.hku.hk

Xuan Bi
The University of Hong Kong
xbi@cs.hku.hk

Zhiyuan Shi
The University of Hong Kong
zyshi@cs.hku.hk

Weixin Zhang
The University of Hong Kong
wxzhang2@cs.hku.hk

Huang Li
Zhejiang University
lihuanglx@gmail.com

Zhenrui Zhang
Zhejiang University
jerryzh168@gmail.com

Bruno C. d. S. Oliveira
The University of Hong Kong
bruno@cs.hku.hk

Abstract

This paper presents **FCore**: an efficient JVM implementation of System F with support for full *tail-call elimination* (TCE). Developing compilers for functional languages targeting the JVM is well-known to be difficult: compiler writers usually have to implement difficult optimizations; or find compromising solutions for their languages to work around the limitations of the JVM. **FCore** aims at removing the high burden faced by functional compiler writers in the JVM. Variants of System F are often used by compilers for languages such as ML or Haskell as a target for the source language. Therefore, instead of developing a compiler from scratch, compiler writers can simply use **FCore** as a target and enjoy of multiple optimizations for free.

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 new way to do TCE on the JVM using constant space. Programs written in **FCore** can use idiomatic functional programming styles (such as relying on TCE or using partial-application) without worrying about the limitations of the JVM. Our empirical results show that programs compiled with **FCore** achieve a better performance than with existing TCE approaches in the JVM.

1. Introduction

Efficient compilers for functional languages in platforms such as the JVM are attractive to both compiler writers and programmers: it enables cross-platform development and comes with a large collection of libraries and tools that took many man-years to develop. Moreover, programmers can benefit from functional programming (FP) languages on the JVM. Functional languages provide simple, concise and elegant ways to write different algorithms and achieve code reuse via higher-order functions. They also offer 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. Functional Programming promotes a programming style where *functions are first-class values* and are often *partially applied* and *curried*. FP also promotes the use of *recursion* instead of mutable state and loops to define algorithms. The JVM is not designed to deal with programs that make intensive use of a functional programming style.

The difficulty in optimizing FP in the JVM means that: *while functional programming in the JVM is possible today, some compromises are still necessary for writing efficient programs.* Existing JVM functional languages, including Scala [19] and Clojure [12], 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 functional styles, which may be costly in those languages, and use the available alternatives instead.

In particular, two important challenges when writing a compiler for a functional language targeting the JVM are:

1. How to efficiently represent first-class functions, currying, and partial function application.
2. How to eliminate and/or optimize tail calls.

For the first problem, there are two standard options to represent functions in the JVM: *JVM methods* and *functions as objects* (FAOs). Encoding first-class functions using only JVM methods directly is limiting: JVM methods do not support currying and partial function application directly. To support these features, the majority of functional languages or extensions (including Scala, Clojure, and Java 8) also 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.

Nevertheless, encoding functions as objects can have substantial time and memory overheads, especially when defining multi-argument functions. Every time a function is needed, we need to allocate a new object. For recursive functions, this can be particularly costly, since we may need a number of object allocations (for functions) proportional to the number of recursive calls. Therefore, programmers (and compiler writers) that care about performance try to avoid using this representation when possible. In other words, we face a dilemma: FAOs have severe performance penalties, but they seem unavoidable to support currying and partial function applications. As a result, languages that care about performance have to use both representations: JVM methods when possible; and FAOs when necessary.

For the second problem, neither FAOs nor JVM methods offer a good solution to deal with *general tail-call elimination* (TCE) [25]. 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 [13, 19]. 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 [22] (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 efficient implementation of System F [11, 21] using the new technique. The compilation technique builds on a new representation of first-class functions in the JVM: *imperative functional objects* (IFOs). IFOs are representations of functions that support currying, partial function applications, and TCE. *With IFOs it is possible to use a single representation of functions in the JVM and still achieve reasonable efficiency for FP*. Thus, IFOs provide a good solution for the two problems mentioned above.

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, this paper focuses on showing how this allows a new and more efficient way to do TCE. The new TCE approach does not require memory allocation for each tail call and is faster 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" [26], in the JVM.

Using IFOs and the new TCE technique, we created **FCore**: an efficient 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. Compiler writers can simply translate from their source functional language into **FCore** and get a number of FP optimizations for free! This liberates compiler writers from implementing a number of tedious and difficult compiler optimizations by themselves, allowing them to focus on the development of their source languages.

Our experimental results show that **FCore** programs perform competitively against programs using regular JVM methods, while still supporting TCE. Programs in **FCore** are also faster and use less memory than programs using conventional JVM trampolines. Therefore, **FCore** has the performance needed to serve as a target for a source language.

In summary, the contributions of this paper are:

- **FCore**: An efficient implementation of a System F-based intermediate language that can be used to target the JVM by FP compilers.
- **Imperative Functional Objects:** A new representation of first-class functions in the JVM, offering new ways to optimize functional programs.
- **A new approach to tail-call elimination:** A new way to implement TCE in the JVM using IFOs.
- **Formalization and empirical results:** Our compilation method from a subset of **FCore** into Java is formalized. Our empirical results indicate that the performance of **FCore** is competitive with regular JVM methods and significantly better than existing TCE approaches.

2. FCore and IFOs, Informally

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

2.1 Encoding Functions with IFOs

In **FCore**, we compile all functions to objects that are instances of 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 such function 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
{
   Function x = this;
   public void apply ()
   {
      final Integer y = (Integer) x.arg;
      res = y;
   }
}
```

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. Of course, a fundamental feature in functional programming is currying. Therefore in **FCore**, it is also possible to 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
{
   Function x = this;
   public void apply ()
   {
      final Integer y = (Integer) x.arg;
      class IConstant extends Function
      {
         Function x1 = this;
         public void apply()
         {
           final Integer y1 = (Integer) x1.arg;
           res = y; // overwrite if: res = x.arg;
         }
      }
      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 `y` for `x.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 if `x.arg` was used instead of `y`, and the outer one would incorrectly return 4 instead of 3.

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

$$three \equiv constant\ 3$$

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. The left-side of Figure 1 shows a naive encoding of these two functions using IFOs in Java. Recursion is encoded using Java's own recursion: the Java references `even` and `odd` are themselves mutually recursive.

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 [26], is to use a simple auxiliary structure

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

that keeps track of the next call to be executed. The right-side of Figure 1 illustrates the use of the `Next` structure. The code differs from the code on the left-side on the recursive (tail) calls of `even` and `odd`. 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 `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. For example, the following code illustrates the call `even 10`:

```
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;
```

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 Middleweight Java (MJ) [5], a formalized subset of Java. **FCore** is an extension of System F (the polymorphic λ-calculus) [11, 21] that can serve as a target for compiler writers. MJ is a minimal imperative core calculus for Java that can facilitate formal reasoning.

3.1 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, 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:

Types	$\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau$
Expressions	$e ::= x \mid \lambda(x : \tau).e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau$

Types τ consist of type variables α, function types $\tau_1 \to \tau_2$, 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 $e\ \tau$.

Middleweight Java. MJ is a valid subset of Java, i.e. every MJ program is an executable Java program. We do not show its full

```
//Naive even and odd
class Mutual {
  Function even;
  Function odd;
  class Even extends Function {
    Function x = this;
    public void apply () {
      final Integer n = (Integer) x.arg;
      if (n == 0) {
        res = true;
      }
      else {
        odd.arg = n - 1;
        odd.apply();
        res = odd.res;
      }
    }
  }
  class Odd extends Function {
    Function x = this;
    public void apply () {
      final Integer n = (Integer) x.arg;
      if (n == 0) {
        res = false;
      }
      else {
        even.arg = n - 1;
        even.apply();
        res = even.res;
      }
    }
  }
  { // initialization block
    odd = new Odd();
    even = new Even();
  }
}
```

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

Figure 1. Functions `even` and `odd` using IFOs naively (left) and with tail-call elimination (right).

syntax and rules here due to their relative verbosity. They can be found in MJ's technical report [5].

3.2 From System F to Middleweight Java (MJ)

Figure 2 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** $\Gamma_J; C; S$

It states that System F expression e with type τ (with respect to translation environments Γ) results in MJ expression J created after executing an *MJ program* in an MJ typing environment Γ_J. In the shown rules, MJ expressions J are always Java variables (to prevent overwriting, as discussed in Section 2). As defined in the technical report [5], an *MJ program* consists of a collection of class definitions (here denoted by C) and a sequence of statements (here denoted by S). We write $C_1\ C_2$ to denote a collection that is composed of class definitions from C_1 and class definitions from C_2. Similarly, a sequence $\{S_1; S_2\}$ executes first all statements in S_1 and then statements in S_2. As the translation nests and decomposes these MJ programs, the sequence of statements of the outermost one corresponds to the body of the `main` method in a Java program. An MJ typing environment, Γ_J, is a map from program variables to expression types [5] (denoted as $x : cd$). We denote joined MJ typing environments by $\uplus$, and an extended typing environment by a comma ($\Gamma_J, x : cd$). Typing environments keep track of the generated temporary Java variables.

As shown in Figure 2, `FJ-Var` checks whether a given value-type binding is present in an environment and generates a corresponding, previously initialized, Java variable. `FJ-TApp` and `FJ-TAbs` do not generate any extra code. `FJ-App` translates value applications. 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 join the typing environments Γ_{J1} and Γ_{J2} and the class collections C_1 and C_2, and execute all dependent statement sequences in order: $S_1; S_2; S_3$. `FJ-Abs` translates term abstractions. Note that we abuse overline notation here. In the typing judgment, $\Gamma \vdash \bar{x} : \bar{\tau} \rightsquigarrow \bar{y}$ **in** $\{\bar{y} : \langle\bar{\tau}\rangle\}; \{\}; \{\}$ denotes zero of more typing judgements ($\Gamma \vdash x_1 : \tau_1 \rightsquigarrow y_1 \ldots \Gamma \vdash x_n : \tau_n \rightsquigarrow y_n$ **in** $\{y_n : \langle\tau_n\rangle\}; \{\}; \{\}$). In the generated code, the meaning of $\bar{y}$ and $\langle\bar{\tau}\rangle$ depends on their context: zero or more *fields* named $y_1 \ldots y_n$ and declared with the corresponding translated types $\langle\tau_1\rangle \ldots \langle\tau_n\rangle$, or *class constructor*

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

(FJ-Var)
$$\frac{(x_1 : \tau \mapsto x_2) \in \Gamma}{\Gamma \vdash x_1 : \tau \rightsquigarrow x_2 \textbf{ in } \{x_2 : \langle\tau\rangle\}; \{\}; \{\}}$$

(FJ-TApp)
$$\frac{\Gamma \vdash e : \forall\alpha.\tau_2 \rightsquigarrow J \textbf{ in } \Gamma_J; C; S}{\Gamma \vdash e\,\tau_1 : \tau_2[\tau_1/\alpha] \rightsquigarrow J \textbf{ in } \Gamma_J; C; S}$$

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

(FJ-App)
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow J_1 \textbf{ in } \Gamma_{J1}; C_1; S_1 \\ \Gamma \vdash e_2 : \tau_2 \rightsquigarrow J_2 \textbf{ in } \Gamma_{J2}; C_2; S_2 \qquad f,\ x_f\ fresh\end{array}}{\Gamma \vdash e_1\,e_2 : \tau_1 \rightsquigarrow x_f \textbf{ in } \Gamma_{J1} \uplus \Gamma_{J2} \uplus \{f : Function, x_f : \langle\tau_1\rangle\}; C_1\ C_2; \{S_1; S_2; S_3\}}$$

S_3 := {
  Function f;
  f = J_1;
  f.arg = J_2;
  f.apply();
  $\langle\tau_1\rangle$ x_f;
  x_f = ($\langle\tau_1\rangle$) f.res;
}

(FJ-Abs)
$$\frac{\begin{array}{c}\Gamma, x : \tau_1 \mapsto x_2 \vdash e : \tau_2 \rightsquigarrow J \textbf{ in } \Gamma_J, x_2 : \langle\tau_1\rangle; C_1; S_1 \\ f,\ x_1\ ,\ x_2\ ,\ FC\ fresh \qquad \Gamma \vdash \bar{x} : \bar{\tau} \rightsquigarrow \bar{y} \textbf{ in } \{\bar{y} : \langle\bar{\tau}\rangle\}; \{\}; \{\}\end{array}}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2 \rightsquigarrow f \textbf{ in } \Gamma_J \uplus \{x_2 : \langle\tau_1\rangle, f : Function, x_1 : Function\}; C_1\ C_2; S_2}$$

C_2 := {
  class FC **extends** Function {
    Function x_1;
    $\langle\bar{\tau}\rangle$ $\bar{y}$;

    public FC($\langle\bar{\tau}\rangle$ $\bar{y}$) {
      super();
      this.$\bar{y}$ = $\bar{y}$;
      this.x_1 = **this**;
    }

    void apply() {
      $\langle\tau_1\rangle$ x_2;
      x_2 = ($\langle\tau_1\rangle$) x_1.arg;
      S_1;
      x_1.res = J;
    }
  }
}
S_2 := {
  Function f;
  f = **new** FC($\bar{y}$);
}

Translation of System F types to Java types:

$\langle\alpha\rangle$ = Object
$\langle\forall\alpha.\tau\rangle$ = $\langle\tau\rangle$
$\langle\tau_2 \to \tau_1\rangle$ = Function

Figure 2. Type-Directed Translation from System F to Middleweight Java

parameters, or *field assignment statements*, or *class instantiation arguments*. For translating term abstractions, we need evidence for resolving *e*, scoped variables $\bar{x}$ of types $\bar{\tau}$, and a bound variable *x* of type τ_1. We then wrap the generated MJ expression *J* and its deriving statements *S* as follows. We create a class with a fresh name *FC*, extending the *Function* class. This class contains fields referring to all translated variables that are in scope ($\bar{y}$) as well as a self-reference field with a fresh name x_1. These fields are initialized in the constructor. In the body of `apply`, we first create an alias for the function argument with a fresh name x_2, then execute all statements S_1 deriving its resulting Java expression *J* that we assign as the output of this function. In the sequence of statements, we create a fresh alias *f* for the instance of the mentioned function, representing the class *FC*, initialized with the translated variables in scope $\bar{y}$. Passing all scoped variables is sub-optimal and the implementation avoids this by using inner classes. For the sake of simplicity, we chose to use this sub-optimal version in the formal presentation.

3.3 Properties of the Translation

In this section, we describe two important properties of our translation: *type preservation*; and *semantic preservation*. For space reasons, we only show a proof sketch of the first property.

Type Preservation. This property states that *translation generates well-typed MJ programs*. Note that all casts introduced by the translation succeed and that the type preservation is partial, because of the property (trivial from translation): $if\ \tau_1 = \tau_2,\ then\ \langle\tau_1\rangle = \langle\tau_2\rangle$ which is not true in the opposite direction.

We state the partial type preservation theorem for open terms below. Every well-formed MJ program *p* induces a class table Δ_p [5] which provides typing information about the methods, constructors, and fields in the given program. As we nest and decompose MJ programs in the translation, we need to express it in the corresponding class tables. We write $\Delta_1 \uplus \Delta_2$ (assuming Δ_1 and Δ_2 are disjoint) to denote a class table that provides typing information of both Δ_1 and Δ_2. $\Delta \supseteq \Delta_1 \uplus \Delta_2$ denotes that the class table Δ subsumes Δ_1 and Δ_2, i.e. it provides typing information of at least what Δ_1 and Δ_2 provide. We express a similar notion in typing environments, i.e. $\Gamma_J \supseteq \Gamma_{J1} \uplus \Gamma_{J2}$. We adapt the typing judgment for MJ [5]:

$\Delta; \Gamma_J \vdash J : T$

Where Δ is a class table, Γ_J is a typing environment, J is an MJ expression, and T is an MJ type. In the theorem below, we assume Δ_0 to be a class table induced by the MJ program $C_0; S_0$, and Δ_G to be a valid global class table (which can be checked by MJ rules) that contains the `Function` class. Note we additionally abuse overline notation in $\Delta; \Gamma_J \vdash \bar{J} : \langle\bar{\tau}\rangle$ to denote zero or more MJ typing judgments ($\Delta; \Gamma_J \vdash J_1 : \langle\tau_1\rangle \ldots \Delta; \Gamma_J \vdash J_n : \langle\tau_n\rangle$) and in $\Gamma = \Gamma_0 \uplus (\bar{x} : \bar{\tau} \mapsto \bar{J})$ to denote splitting of translation environments into Γ_0 (nothing or type variables) and environment mapping of all System F term variables to Java variables.

Theorem 3.1. *Suppose* $\Gamma \vdash e : \tau_0 \rightsquigarrow J_0$ ***in*** $\Gamma_{J0}; C_0; S_0$ *with* $\Gamma = \Gamma_0 \uplus (\bar{x} : \bar{\tau} \mapsto \bar{J})$ *and* $\Gamma \vdash \bar{x} : \bar{\tau} \rightsquigarrow \bar{J}$ ***in*** $\{\bar{J} : \langle\bar{\tau}\rangle\}; \{\}; \{\}$. *For any valid class table* $\Delta \supseteq \Delta_0 \uplus \Delta_G$ *and typing environment* $\Gamma_J \supseteq \Gamma_{J0}$ *such that* $\Delta; \Gamma_J \vdash \bar{J} : \langle\bar{\tau}\rangle$, *it is the case that* $\Delta; \Gamma_J \vdash J_0 : \langle\tau_0\rangle$.

Proof. The proof proceeds by the induction on the translation rules:

—**case** `FJ-Var` follows immediately from the assumptions about Δ and Γ_J.

—**cases** `FJ-TAbs` and `FJ-TApp` do not generate any code. The results follow from applying the induction hypothesis and the translation of types with type erasure.

—**case** `FJ-App`: Let Δ_1 and Δ_2 be class tables induced by the MJ programs $C_1; S_1$ and $C_2; S_2$ respectively. We assume $\Delta \supseteq \Delta_1 \uplus \Delta_2 \uplus \Delta_G$ and $\Gamma_J \supseteq \Gamma_{J1} \uplus \Gamma_{J2}$. We apply the induction hypothesis to $\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow J_1$ **in** $\Gamma_{J1}; C_1; S_1$, so we know $\Delta_1 \uplus \Delta_G$ is a valid class table and $\Delta; \Gamma_J \vdash J_1 : \langle\tau_2 \to \tau_1\rangle$.

We apply the induction hypothesis to $\Gamma \vdash e_2 : \tau_2 \rightsquigarrow J_2$ **in** $\Gamma_{J2}; C_2; S_2$, so we know $\Delta_2 \uplus \Delta_G$ is a valid class table and $\Delta; \Gamma_J \vdash J_2 : \langle\tau_2\rangle$.

Assuming Δ_1, Δ_2, and Δ_G are distinct, $\Delta_1 \uplus \Delta_2 \uplus \Delta_G$ is also a valid class table.

For $\Gamma \vdash e_1\, e_2 : \tau_1 \rightsquigarrow x_f$ **in** $\Gamma_{J1} \uplus \Gamma_{J2}; C_1\ C_2; \{S_1; S_2; S_3\}$, we show that $C_1\ C_2; \{S_1; S_2; S_3\}$ is a valid MJ program by `T-ProgDef`. The class collections C_1 C_2 contain valid class definitions by `T-CDefn`. We can check the sequences of statements $S_1; S_2; S_3$ and S_3 by `TS-Seq`. Then:

$\Delta; \Gamma_J \vdash Function\ f; : Function$ by `TS-Intro`
$\Delta; \Gamma_J \vdash f = J_1; : void$ by `TS-Write`
$\Delta; \Gamma_J \vdash f.arg = J_2; : void$ by `TS-FieldWrite`
$\Delta; \Gamma_J \vdash f.apply(); : void$ by `TE-Method`
$\Delta; \Gamma_J \vdash \langle\tau_1\rangle\ x_f; : \langle\tau_1\rangle$ by `TS-Intro`
$\Delta; \Gamma_J \vdash (\langle\tau_1\rangle)\ f.res; : \langle\tau_1\rangle$ by `TE-DownCast`
$\Delta; \Gamma_J \vdash x_f = (\langle\tau_1\rangle)\ f.res; : void$ by `TS-Write`

Hence, we have a valid MJ program, which implies $\Delta; \Gamma_J \vdash x_f : \langle\tau_1\rangle$ by `TE-Var`.

—**case** `FJ-Abs` Let Δ_1 be a class table induced by the MJ program $C_1; S_1$. We assume $\Delta_3 \supseteq \Delta_1 \uplus \Delta_G$ and $\Gamma_{J3} \supseteq \Gamma_J, x_2 : \langle\tau_1\rangle$.

We apply the induction hypothesis to $\Gamma, x : \tau_1 \mapsto x_2 \vdash e : \tau_2 \rightsquigarrow J$ **in** $\Gamma_J; C; S$

and we have $\Delta_3; \Gamma_{J3} \vdash J : \langle\tau_2\rangle$.

We apply the induction hypothesis to $\Gamma \vdash \bar{x} : \bar{\tau} \rightsquigarrow \bar{y}$ **in** $\Gamma_J; \{\}; \{\}$

and we have $\Delta_3; \Gamma_{J3} \vdash \bar{y} : \langle\bar{\tau}\rangle$.

Let Δ_2 be a class table induced by the MJ program $C_1\ C_2; S_2$. We assume $\Delta_4 \supseteq \Delta_2 \uplus \Delta_G$ and $\Gamma_{J4} \supseteq \Gamma_J \uplus \{x_2 : \langle\tau_1\rangle, f : Function, x_1 : Function\}$.

For $\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2 \rightsquigarrow f$ **in** $\Gamma_J, x_2 : \langle\tau_1\rangle, f : Function, x_1 : Function; C_1\ C_2; S_2$, we check that $C_1\ C_2; S_2$ is a valid MJ program by `T-ProgDef`. The class collection C_1 contains valid class definitions by `T-CDefn`. The class collection C_2 has a valid class definition by `T-CDefn`, `T-FieldsOk`, `T-ConsOK`, `T-Mbodys`, `T-MethOk1`, and we check its constructor statements by `TS-Seq`:

$\Delta_4; \Gamma_{J4} \vdash super(); : void$ by `T-CSuper`.
$\Delta_4; \Gamma_{J4} \vdash this.\bar{y} = \bar{y}; : void$ and $\Delta_4; \Gamma_{J4} \vdash this.x_1 = this; : void$ by `TS-FieldWrite`

We check its apply method body by `T-MDefn` and `TS-Seq`:

$\Delta_4; \Gamma_{J4} \vdash \langle\tau_1\rangle\ x_2; : \langle\tau_1\rangle$ by `TS-Intro`
$\Delta_4; \Gamma_{J4} \vdash x_1 : Function$ by `TE-FieldAccess`
$\Delta_4; \Gamma_{J4} \vdash (\langle\tau_1\rangle)\ x_1.arg; : \langle\tau_1\rangle$ by `TE-DownCast`
$\Delta_4; \Gamma_{J4} \vdash x_2 = (\langle\tau_1\rangle)\ x_1.arg; : void$ by `TS-Write`
$\Delta_4; \Gamma_{J4} \vdash S_1; : void$ by `TS-Seq`
$\Delta_4; \Gamma_{J4} \vdash x_1.res = J; : void$ by `TS-FieldWrite`

Then, we check the sequence of statements S_2 by `TS-Seq`:

$\Delta_4; \Gamma_{J4} \vdash Function\ f; : Function$ by `TS-Intro`
$\Delta_4; \Gamma_{J4} \vdash new\ FC(\bar{y}) : Function$ by `TE-New`
$\Delta_4; \Gamma_{J4} \vdash f = new\ FC(\bar{y}); : void$ by `TS-Write`

Thus, $C_1\ C_2; S_2$ is a valid MJ program, which implies $\Delta_4; \Gamma_{J4} \vdash f : Function$ by `TE-Var`.

□

The partial type preservation also holds for closed terms:

Corollary 3.2. *Suppose* $\vdash e : \tau \rightsquigarrow J$ ***in*** $\Gamma_J; C; S$. *For a valid class table* $\Delta \uplus \Delta_G$ *and typing environment* Γ_J, *it is the case that* $\Delta \uplus \Delta_G; \Gamma_J \vdash J : \langle\tau\rangle$.

Note that even though we have empty translation environments here, we may not have empty MJ typing environments Γ_J due to the generation of temporary variables.

Semantic Preservation. This property states that for any valid System F expression, the result of evaluating that expression under its standard call-by-value semantics is the same as the result of executing the corresponding compiled code. The proof of this property is more involved due to the lack of commutativity between the translation and System F evaluation. It is left for 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 `CJ-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 `CJ-App` to include one extra freshly generated variable c:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow J_1 \textbf{ in } \Gamma_{J1}; C_1; S_1 \\ \Gamma \vdash e_2 : \tau_2 \rightsquigarrow J_2 \textbf{ in } \Gamma_{J2}; C_2; S_2 \qquad f,\ x_f\ ,\ c\ fresh\end{array}}{\begin{array}{c}\Gamma \vdash e_1\, e_2 : \tau_1 \rightsquigarrow x_f \textbf{ in } \Gamma_{J1} \uplus \Gamma_{J2} \\ \uplus\{f : Function, c : Function, x_f : \langle\tau_1\rangle\}; C_1, C_2; \{S_1; S_2; S_3\}\end{array}}$$

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 xf;
      do {
        c = Next.next;
        Next.next = null;
        c.apply();
      } while (Next.next != null);
      xf = 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. Note that the rules here do not produce valid MJ programs, because MJ does not support **`static`** fields and looping constructs. For restating the translation properties, we would need to use a different formalization of Java that supports both of these features. The *type preservation* proof would be augmented in two ways: the global class table Δ_G would also contain information about the `Next` class and the value application case would have two sub-cases for tail and non-tail calls. The *semantic preservation* proof would require formalization of the tail call detection mechanism and be even more involved due to the interaction between tail and non-tail calls. Since this is a major endeavor, which is not in-scope for this paper, we leave formalizing an extended subset of Java and proving the properties of the TCE-augmented translation for future work.

5. Implementation

We implemented[1] a compiler for **FCore** based on the representation and type-directed translation we described in Sections 3 and 4. We wrote the compiler in Haskell. This section overviews additional information about the implementation. Our actual implementation first translates System F to an equivalent intermediate form with multi-binders (Closure F), from which it generates Java code.

5.1 Syntax Extensions and Java Interoperability

The implemented **FCore** and Closure F extend the basic System F with other constructs. These include primitive operations, types and literals, let bindings, conditional expressions, tuples, and fixpoints. We also have different frontends, which extend **FCore** with convenient language constructs that helped us in development. We wrote our benchmark programs in one of these frontends.

We also added constructs for a basic interoperability mechanism with Java and the JVM: creating instances of classes, calling methods, accessing fields, and executing blocks of statements. We typecheck these constructs using an "oracle". In the implementation, the type-checker sends queries about Java constructs to a server with a set classpath and the server answers these queries using the Reflection API. One advantage of this design is that it allows us to target different Java-based platforms. Apart from supporting the JVM, we can support the Android platform. The Android platform uses a different set of libraries from the JVM SDK.

Besides benchmark programs, we have a number of other programs written in **FCore**. These programs include functional programs for drawing fractals (such as the Mandelbrot set, the Burning Ship, or the Sierpinski Triangle) which helped us to test the Java interoperability. We also have a test suite with unit tests of the **FCore** compiler.

5.2 Thread Safety and Delayed Memory Allocation

System F is not a concurrent calculus and the presented technique focuses on sequential functional-style code compilation. Neverthe-

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

less, JVM is inherently a concurrent platform and the use of mutable fields may seem to prevent thread safety. This would be a major drawback of IFOs, because, for instance, the generated code could not be safely used from multi-threading Java applications.

Before addressing the problem, we describe what multi-threading scenarios we are concerned about. We consider multi-threading Java programs, in which a programmer imports some code generated by the compiler. According to the formalization of **FCore**, the programmer gets a `Function` instance. This instance may be shared by multiple threads in the application program, thus results of function invocations may be incorrect due to the interleaving of these threads.

This, however, can be easily fixed: we enforce a separate instance for each thread by allocating objects at their call sites rather than at their definition sites. This is reflected in our actual implementation: each *FC* class definition is followed by a wrapping `Thunk` instance. Here, we briefly illustrate the idea.

Suppose we compiled the following *increment* function:

$$inc \equiv (\lambda x : Int).\ x + 1$$

into Java code:

```
class Inc extends Function
{
 Function x2 = this;
 public void apply ()
 {
  final Integer x3 = (Integer) x2.arg;
  final Integer x4 = x3 + 1;
  res = x4;
 }
}
Thunk inc = () -> new Inc();
```

Note that on the last line, we create a `Thunk` instance instead of `Function inc = new Inc()`. A `Thunk` is simply a Java functional interface with a single abstract method:

```
public interface Thunk {
 public Function compute();
}
```

Now a programmer can make a thread-safe call to, for instance, `inc(inc(0))` as follows:

```
Function inc1 = inc.compute();
inc1.arg = 0;
inc1.apply();
Function inc2 = inc.compute();
inc2.arg = inc1.res;
inc2.apply();
```

because `inc1` and `inc2` are two different instances of the same `Inc` class. With the `Thunk` interface, we can wrap the code of object allocation inside the `compute` method in order to delay memory allocation. Therefore, in a multi-threading scenario, even if a `Thunk` instance is shared by multiple threads, each invocation of `compute` will result in a separate object instance. This makes different function invocations reside in different memory spaces without interfering with each other.

The `Thunk` interface also works with higher-order functions. Consider the following example:

$$twice \equiv (\lambda f : Int \to Int).\ (\lambda x : Int).\ f\ (f\ x)$$

Here *twice* takes another function as an input. In the exported Java code, a programmer is provided with a Java interface:

```
interface Twice {
 default int twice (Thunk f, int x) {
  Function fun1 = f.compute();
  fun1.arg = (Integer) x;
  fun1.apply();
  Integer res1 = (Integer) fun1.res;
  Function fun2 = f.compute();
  fun2.arg = (Integer) res1;
  fun2.apply();
  Integer res2 = (Integer) fun2.res;
  return res2;
 }
}
```

If the programmer wants to call, for instance, `twice inc`, he/she is forced to pass a `Thunk` instance of `inc` to the `twice` method as an argument. This ensures that each call with the function argument sees a new instance of `Function` class.

The `Thunk` interface works well with System F binders and *let* expressions. At the moment, we are implementing a modified version to work with TCE. Because in the TCE-augmented translation, there would be an additional race condition due to the static `Next` field. Again, we only need a small modification: the `Next` field should not be static and `Next` must have a local instance in each thread. Function applications would be done via this local instance of `Next` rather than the class. We leave a full implementation for future work.

5.3 Additional Optimizations

Our compiler also performs standard optimizations that are not accounted by the basic translations in Sections 3 and 4. It uses Java's inner classes to enforce lexical scoping of generated variables. This nesting optimization avoids the initialization overhead of passing the scoped variables through constructors of top level classes. For unboxing, different `Function` classes exist with specialized primitive fields. The translation chooses among them and generates variables with primitive types when possible. The compiler inlines expressions of let bindings, value abstractions, and fixpoints; and partially evaluates value applications, conditionals, and primitive type operations.

An important optimization that our compiler also does is multi-argument function optimization. This kind of optimization is standard in FP [17] and provides a major boost in time and memory performance. The straightforward translation described in the previous section has one drawback when translating multi-argument functions: for each single value application, we generate a function object that requires an apply call to execute the body. Using the additional information that Closure F has about multi-argument binders, our compiler creates a single multi-argument function, instead of multiple single argument functions.

6. Evaluation

FCore performs well in common FP scenarios. Our evaluation of this claim consists of two parts. We first compare time performance results of benchmark programs that isolate different call behaviors. Then, we demonstrate two application use cases and examine their scalability.

6.1 Isolated Call Behavior

In this experiment, we want to evaluate runtime performance of programs representing different common call behaviors in FP. We compare the measured average times of these programs written in **FCore** (to assess IFOs) and of corresponding programs written in Scala, Clojure, and Java using standard functions (as methods) and trampolines.

Benchmark Design. We wrote the benchmark programs in the extended System F for our compilation process and in the following JVM-hosted languages in their latest stable versions:

- *Scala* (2.11.2): Scala is one of the most popular strongly-typed multi-paradigm languages on the JVM [19]. It directly compiles down to the Java bytecode and applies various optimizations. In order to encode mutually recursive tail calls, we used

Low Input	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	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

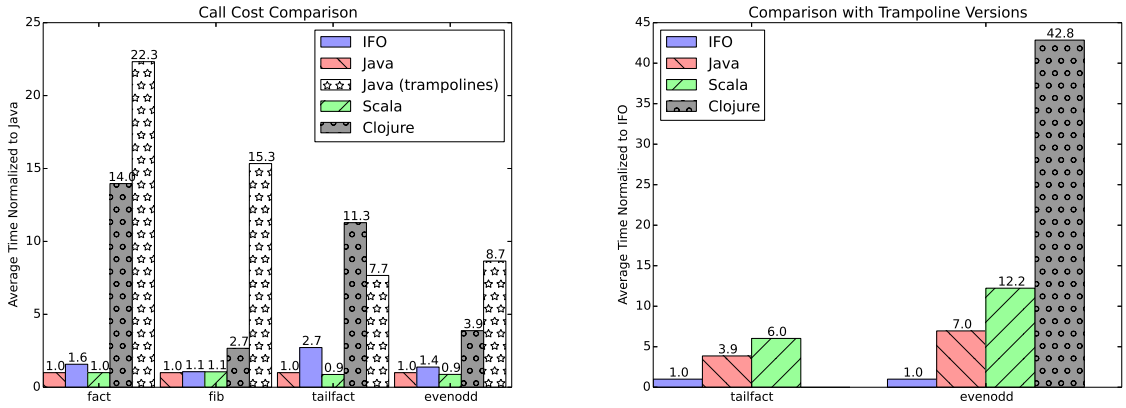

Figure 3. The isolated call behavior experiments: the reported times are averages of 10 measured runs and corresponding standard deviations. The plots are normalized to Java's (left table and plot) and IFO's (right table and plot) results – the lower, the faster.

Input length (time unit)	1000 (μs)	3000 (μs)	10000 (μs)	100000 (μs)	Objects (Min)	Objects (Max)
IFO	5.10 ± 0.10	15.98 ± 0.07	77.81 ± 0.83	933.58 ± 13.40	5451	5451
Java (Trampoline-based)	7.03 ± 0.130	26.89 ± 0.10	102.98 ± 2.36	1099.80 ± 15.46	4665	104879
Java (Method-based)	3.80 ± 0.07	11.61 ± 0.10	48.83 ± 0.13	N/A	4128	4128
Java (FAO-based)	6.37 ± 0.01	17.62 ± 0.05	N/A	N/A	18102	24082

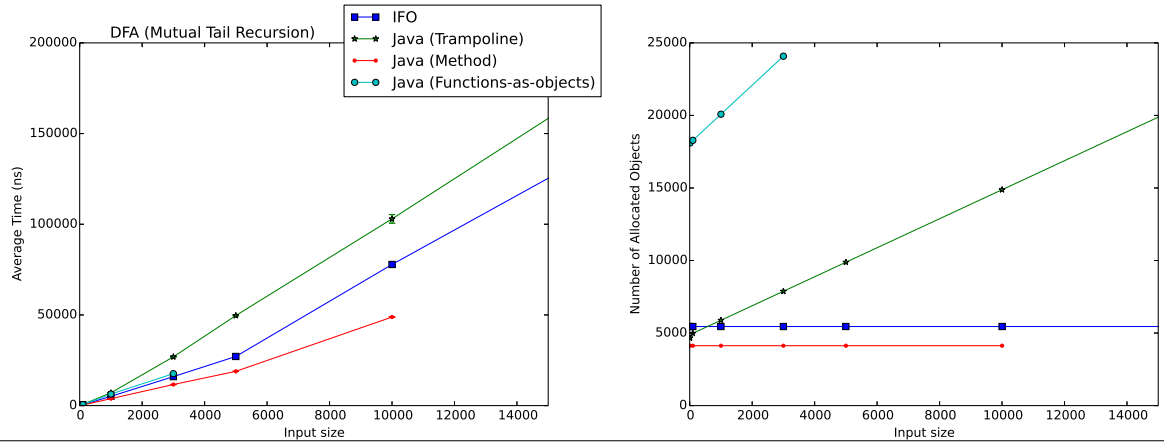

Figure 4. The DFA encoding: the reported times are averages of 10 measured runs and corresponding standard deviations; the last two columns show the minimum and maximum numbers of total allocated objects on heap from isolated profiled runs with all input lengths. Due to space limitations, the x-axes of plots are cropped at 15000 for clarity.

Input length (time units)	10 (μs)	20 (μs)	40 (ms)	50 (ms)	75 (ms)
Java (Trampoline-based)	17.10 ± 0.26	320.42 ± 5.88	14.84 ± 0.27	62.35 ± 1.06	1044.61 ± 12.99
IFO	11.06 ± 0.55	289.19 ± 6.52	12.70 ± 0.26	48.47 ± 1.08	805.06 ± 7.06
Relative speedup	54.6%	10.8%	16.9%	28.6%	29.8%

Figure 5. The 0-1 Knapsack Problem encoded using CPS for different input sizes (the length of weight and value lists) for a fixed total weight of 10: the reported times are averages of 10 measured runs and corresponding standard deviations.

the provided trampoline facility (`scala.util.control.TailCalls`).

- *Clojure* (1.6.0): Clojure is a dynamically-typed, functional language which compiles to the Java bytecode [12]. For mutually recursive tail calls, we used `tramp` from `clojure.core.logic`.
- *Java* (1.8.0_25): We implemented two versions, one version using method calls in Java 8 and one using custom (hand-written and optimized) trampolines.

To evaluate IFOs, we used the **FCore** compiler with all the optimizations mentioned in Sections 4 and 5.
We executed all benchmarks on the following platform with the latest Java HotSpot™VM (1.8.0_25): Intel®Core™i5 3570 CPU, 1600MHz DDR3 4GB RAM, Ubuntu 14.04.1.
For the automation of performance measurement, we used the Java Microbenchmark Harness (JMH) tool which is a part of OpenJDK [10]. 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.

Programs. 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.

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.

Results. We show the results in Figure 3. 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.

6.2 Applications

In this experiment, we examine the time and memory scalability of IFOs and alternative closure representations (methods, functions-as-objects, and trampolines) in two applications which make use of tail calls. Unlike Section 6.1, where the programs isolated costs of plain recursive calls, the applications here represent a more realistic behavior with other costs, such as non-recursive method calls, calls to other API methods or partial applications.

We implemented the applications in **FCore** to assess IFOs and in Java to assess different closure representations: method calls, Java 8's lambdas (functions-as-objects), and custom trampolines. We chose Java, because our custom implementation of trampolines performed best in the isolated call behavior experiments. Using plain Java implementations, we can examine the runtime behavior of different representations without potential compiler overheads. We performed the time measurement in the same setting as in the previous experiment. For the memory, we report the total number of allocated objects on heap in the isolated application runs, as measured by HPROF [20], the JDK's profiling tool.

DFA Encoding. 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 [15]. 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 show the result of this experiment in Figure 4. 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.

What is more important here is the memory consumption. 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.

CPS Encoding of Knapsack. Tail calls find their application in continuation-passing style [27] where each call is a tail call. In this application, we encoded a naive implementation of the 0-1 Knapsack Problem with multiple recursive calls and also non-recursive tail calls. In the experiment, we fixed the total weight to 10 and generated input values and weights lists of different sizes: we generated weights as consecutive sequences 1 to 5 and values as $i \times weight[i]$.

We show the results in Figure 5. This implementation contains partial applications and a pure method-based implementation is impossible in this style. The method- and FAO-based implementations could not run beyond the input of length 10 due

to a `StackOverflow` exception; their execution times at 10 were $9.68 \pm 0.20\ \mu s$ and $13.79 \pm 0.22\ \mu s$ respectively. IFOs ran 10.8% to 54.6% faster than the trampoline implementation in different input lengths.

6.3 Discussion

The programs benchmarked in Section 6.1 stressed the time spent on recursive calls. These programs are quite simple, but representative of common patterns of functional programming. IFOs gave a 3.9 to 12.2-times speed-up over the trampoline implementations. This huge gain is also due to other optimizations in our compiler. In particular, since those programs have small and simple definitions, inlining is beneficial and gives a big performance gain on top of the gains for TCE. In the more complex scenarios of Section 6.2, the tail calls are not the only runtime cost, so the overhead of trampolines is much smaller. Still, IFO programs gained a speed-up of 0.1 to 0.5 over our hand-written and optimized trampolines. The other benefit for programs with tail recursion is the constant memory overhead of calls in IFOs. This also appears in method-based implementations, but they throw a `StackOverflow` exception in larger inputs and IFOs do not.

7. 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 [14]. 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 [6, 29, 31] used different variations of the *graph reduction machine* [30] for their code generation, whereas we translate from System F. Nu [9] provided a similar intermediate layer targeting the JVM for developing Aspect Oriented languages.

Tail-Call Elimination and Function Representations. A choice of a function representation plays a great role [24] 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 [25], implementors of FP languages often recognize TCE as a necessary feature. Steele's Rabbit Scheme compiler [26] 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 [19] or Clojure [12], 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 [32]. Later work [18] 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 [26] 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 [22] 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 [16], 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™ VM [23] with TCE support. The research Maxine VM with its new self-optimizing runtime system [33] 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 [28]. Hence, ideas from our work can be applied outside of the JVM ecosystem.

8. Conclusion

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. We believe that IFOs bring transparency to FP in the JVM: programmers will be able to be oblivious of the limitations of the JVM. Furthermore, using **FCore** will bring transparency to compiler writers: compiler writers will be able to easily target the JVM, without having to spend lots of effort working around the limitations of the JVM.

There is much to be done for future work. We would like to prove semantic preservation of our translation from System F to Middleweight Java. We have also barely begun exploring what optimizations can be done with IFOs. We would like to formalize and refine a number of optimizations that we have been experimenting with in **FCore**. We are particularly interested in addressing the pressing problem of boxing and unboxing of primitive types in the JVM. Currently, our compiler supports a mechanism based on specialization, much like the one employed in Scala [8]. However, we believe IFOs offer a new alternative to unboxing that avoids the code bloat problems of specialization, without giving up the performance benefits. Finally, we want to build frontends for realistic

functional languages on top of **FCore** and write large functional programs, such as a full bootstrapping compiler of **FCore**, in those frontends.

Acknowledgments

We would like to thank all members of the newly formed Programming Languages Research Group at the University of Hong Kong and all contributors to **FCore** code repository. We especially thank Haoyuan Zhang, Boya Peng, and Ningning Xie for their contributions.

References

[1] H. Abelson, R. Dybvig, C. Haynes, G. Rozas, I. Adams, N.I., D. Friedman, E. Kohlbecker, J. Steele, G.L., D. Bartley, R. Halstead, D. Oxley, G. Sussman, G. Brooks, C. Hanson, K. Pitman, and M. Wand. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] H. G. Baker. CONS should not CONS its arguments, part II. *ACM SIGPLAN Notices*, 30(9):17–20, 1995.

[4] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java Bytecodes. In *Proceedings of the 3^{rd} ACM SIGPLAN International Conference on Functional Programming*, pages 129–140, 1998.

[5] G. Bierman, M. Parkinson, and A. Pitts. An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.

[6] K. Choi, H.-i. Lim, and T. Han. Compiling lazy functional programs based on the spineless tagless G-machine for the Java virtual machine. *Functional and Logic Programming*, pages 92–107, 2001.

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

[8] I. Dragos. *Compiling Scala for Performance*. PhD thesis, PhD thesis, École Polytechnique Fédérale de Lausanne, 2010.

[9] R. Dyer and H. Rajan. Nu: A Dynamic Aspect-oriented Intermediate Language Model and Virtual Machine for Flexible Runtime Adaptation. In *Proceedings of the 7^{th} International Conference on Aspect-oriented Software Development*, pages 191–202, 2008.

[10] S. Friberg, A. Shipilev, A. Astrand, S. Kuksenko, and H. Loef. OpenJDK: jmh, 2014. URL `openjdk.java.net/projects/code-tools/jmh/`.

[11] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[12] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, page 1, 2008.

[13] R. Hickey. Recur construct, Clojure documentation, 2014. URL `clojuredocs.org/clojure.core/recur`.

[14] A. Kennedy and D. Syme. Transposing F to C#: expressivity of parametric polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16(7):707–733, 2004.

[15] S. Krishnamurthi. Educational Pearl: Automata via Macros. *Journal of Functional Programming*, 16(3):253–267, 2006.

[16] C. League, V. Trifonov, and Z. Shao. Functional Java Bytecode. *Proceedings 5^{th} World Conference on Systemics, Cybernetics, and Informatics*, 2001.

[17] S. Marlow and S. Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16 (4-5):415–449, 2006.

[18] Y. Minamide. Selective Tail Call Elimination. In *Proceedings of the 10^{th} International Conference on Static Analysis*, pages 53–170, 2003.

[19] M. Odersky. *The Scala Language Specification, Version 2.9*. École Polytechnique Fédérale de Lausanne, 2014.

[20] K. O'Hair. HPROF: a Heap/CPU profiling tool in J2SE 5.0. *Sun Developer Network, Developer Technical Articles & Tips*, 2004.

[21] J. C. Reynolds. Towards a Theory of Type Structure. In *Symposium on Programming*, pages 408–425, 1974.

[22] M. Schinz and M. Odersky. Tail call elimination on the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*, 59(1): 158–171, 2001.

[23] A. Schwaighofer. Tail Call Optimization in the Java HotSpot^{TM}VM, 2009. Master Thesis, Johannes Kepler Universität Linz.

[24] Z. Shao and A. W. Appel. Space-efficient closure representations. *ACM SIGPLAN Lisp Pointers*, VII(3):150–161, 1994.

[25] G. L. Steele. Debunking the "Expensive Procedure Call"" Myth or, Procedure Call Implementations Considered Harmful or, LAMDBA: The Ultimate GOTO. In *Proceedings of the 1977 Annual Conference*, pages 153–162, 1977.

[26] G. L. Steele. Rabbit: A Compiler for Scheme. Technical report, Massachusetts Institute of Technology, 1978.

[27] G. L. Steele and G. J. Sussman. Scheme: An interpreter for the extended lambda calculus. *Artificial Intelligence Lab Memo*, 349, 1975.

[28] C. Tismer. Continuations and stackless Python. In *Proceedings of the 8^{th} International Python Conference*, volume 1, 2000.

[29] M. Tullsen. Compiling Haskell to Java. Technical Report YALEU/DCS/RR-1204, Yale University, 1996.

[30] C. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, University of Oxford, 1971.

[31] D. Wakeling. Compiling lazy functional programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):579–603, 1999.

[32] I. Wechsung. Frege, 2014. URL `github.com/Frege/frege`.

[33] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 187–204, 2013.