FCore: From System F to Java Efficiently!

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) [Armstrong 2007].

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 [Odersky 2014] and Clojure [Hickey 2008], 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) [Steele 1977]. 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 [Odersky 2014; Hickey 2014]. 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 [Schinz and Odersky 2001] (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 [Girard 1972; Reynolds 1974] 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" [Steele 1978], 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 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:

```
Function id = new Function() {
  void apply() { res = arg; }
};
```

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:

```
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:

```
Function constant = new Function() {
  void apply() {
    res = new Function() {
      void apply() {res = constant.arg;}
    };
  }
};
```

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. We encode an application such as $constant\ 3\ 4$ as:

```
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.

An alternative way to encode *constant* is:

```
Function constant2 = new Function() {
  {
    res = new Function() {
      void apply() {res = constant2.arg;}
    };
  }
  void apply() {} // empty method
};
```

Differently from the previous definition, we set the first `res` field when initializing `constant2`, instead of only after we invoke the `apply` method. This approach has two benefits: 1) when we create `constant2`, the second lambda is also initialized; 2) because the `apply` method of the first lambda function becomes empty, it is redundant to call it. Therefore, as a consequence of 2), the invocation $constant\ 3\ 4$ becomes:

```
constant2.arg = 3;
// no call needed here
Function g = (Function) constant2.res;
g.arg = 4;
g.apply();
```

Instead of two `apply` method calls, we only need one call. This alternative encoding can improve memory and time performance of multi-argument functions. It is especially important in recursive multi-argument functions. With the first encoding, those functions create new `Function` objects on every recursive call and require multiple `apply` methods. This is exactly the same deficiency as of the FAO encoding. However, the alternative encoding avoids these problems.

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:

```
constant.arg = 3;
constant.apply();
```

If we use the alternative encoding it is even simpler, since we do not need to invoke the `apply` method:

```
constant2.arg = 3;
```

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

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

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 fields `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 [Steele 1978], 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`

```
//Naive even and odd
Function even = new Function() {
  void apply() {
    Integer n = (Integer) arg;
    if (n == 0)
      res = true;
    else {
      odd.arg = n-1;
      odd.apply();
      res = odd.res;
    }}
};
Function odd = new Function() {
  void apply() {
    Integer n = (Integer) arg;
    if (n == 0)
      res = false;
    else {
      even.arg = n-1;
      even.apply();
      res = even.res;
    }}
};
```

```
// tail call elimination
Function teven = new Function() {
  void apply() {
    Integer n = (Integer) arg;
    if (n == 0)
      res = true;
    else {
      todd.arg = n-1;
      // tail call
      Next.next = todd;
    }}
};
Function todd = new Function() {
  void apply() {
    Integer n = (Integer) arg;
    if (n == 0)
      res = false;
    else {
      teven.arg = n-1;
      // tail call
      Next.next = teven;
    }}
};
```

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

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 Java. **FCore** is an extension of System F (the polymorphic λ-calculus) [Girard 1972; Reynolds 1974] that can serve as a target for compiler writers. We translate **FCore** to Closure F, which is a variant of System F with multi-binders. Finally from Closure F, we generate Java code.

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:

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

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 $e\ \tau$.

Closure F This is the corresponding syntax of Closure F:

$$\begin{array}{ll} \textbf{Types} & T ::= \alpha \mid \forall \Delta . T \\ \textbf{Expressions} & E ::= x \mid \lambda \Delta . E \mid E\ E \mid E\ T \\ \textbf{Binders} & \Delta ::= \epsilon \mid \Delta(x : T) \mid \Delta\alpha \end{array}$$

A difference to System F is that it accumulates function types and type abstractions in the type multi-binder $\forall\Delta.T$. Correspondingly, expression abstractions over values and types are accumulated in the multi-binder $\lambda\Delta.E$.

The **FCore** compiler uses the multi-binders in Closure F to detect multi-arguments functions. Then, it can compile multi-argument functions using a variant of the alternative encoding for curried functions informally presented in Section 2.1. With this encoding, it is possible, among other uses, to adapt well-known techniques [Marlow and Jones 2006] to optimize multi-argument functions.

3.2 From FCore to Closure F

We can translate System F to Closure F in a straightforward way. We consider two examples that illustrate how System F expressions get translated to Closure F. The first one is the System F *const* function defined as follows:

$const \equiv \Lambda A.(\lambda x : A).(\lambda y : A).x$

For a given type, this function takes two arguments and always returns the first one. In total, we have three single binders which get merged into a single one in Closure F:

$const \equiv \lambda A\ (x : A)\ (y : A).x$

In the second example, we apply the *const* function to a polymorphic single-argument function type, and partially apply it with the polymorphic identity function:

$const\ (\forall B.B \to B)\ (\Lambda B.(\lambda z : B).z)$

In this case, we have one type binder and a function type in the first argument ($\forall B.B \to B$), and two single-binders in the second argument ($\Lambda B.(\lambda z : B).z$). The corresponding Closure F expression is:

$const\ (\forall B\ (x : B).B)\ (\lambda B\ (z : B).z)$

In the first argument, the type abstraction and the function parameter are grouped into a single binder. In the second argument, a similar transformation happens at the value level.

Translation We define the translation of **FCore** to Closure F by two mutually recursive functions:

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![e_1\ e_2]\!] &= [\![e_1]\!]\ [\![e_2]\!] \\
[\![e\ \tau]\!] &= [\![e]\!]\ |\tau| \\
[\![e]\!] &= [\![e]\!](\epsilon) \\
\\
[\![\lambda x : \tau_1 . e]\!](\Delta) &= [\![e]\!](\Delta\ (x : |\tau_1|)) \\
[\![\Lambda \alpha . e]\!](\Delta) &= [\![e]\!](\Delta\ \alpha) \\
[\![e]\!](\Delta) &= \lambda \Delta . [\![e]\!]
\end{aligned}
$$

The first function recurses through the expressions that are identical in both languages. Its last case then calls the second function with an empty multi-binder. The second function captures all consecutive System F single-binders on the given sub-expression and inserts them into the Δ multi-binder. In the whole expression, we can have several "groups" of sub-expressions where each group has its own multi-binder. Type conversion works similarly. We have two mutually recursive functions, but we need to generate fresh variables when converting function types:

$$
\begin{aligned}
|\alpha| &= \alpha \\
|\tau| &= |\tau|(\epsilon) \\
\\
|\tau_1 \to \tau_2|(\Delta) &= |\tau_2|\ (\Delta\ (x : |\tau_1|))\ \textbf{where}\ x\ \textit{fresh} \\
|\forall \alpha . \tau|(\Delta) &= |\tau|\ (\Delta\ \alpha) \\
|\tau|(\epsilon) &= \tau \\
|\tau|(\Delta) &= \forall \Delta . \tau
\end{aligned}
$$

3.3 From Closure F to Java

Figure 2 shows the type-directed translation rules that generate Java code from given Closure F expressions. We exploit the fact that System F and Closure F have 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 three sets of rules in our translation. The first two are mutually recursive rules for translating expressions: one is matching each Closure F expression; the other is providing evidence for variables when translating value abstractions. The final set of rules resolves types. At the bottom of Figure 2, the function $\langle T \rangle$ describes how we translate Closure F types into Java types.

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

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

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 Closure F variables and Java variables in the generated code.

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

$\Gamma \vdash E : T \rightsquigarrow J\ \textbf{in}\ S$

It states that Closure F expression E with type T results in Java expression J created after executing a block of statements S with respect to translation environments Γ. `CJ-Var` checks whether a given value-type binding is present in an environment and generates a corresponding, previously initialized, Java variable. `CJ-Abs` uses the evidence derived from the second set of rules to translate value abstractions. `CJ-TApp` resolves the type of an abstraction and substitutes the applied type in it. `CJ-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: $S_1 \uplus S_2 \uplus S_3$.

The second set of rules processes the binders in Δ that we generate by the rule `CJ-Abs`. The set of rules has the form:

$\Gamma; \Delta \vdash E : T \rightsquigarrow J\ \textbf{in}\ S$

`CJD-Empty` handles the case when we encounter no binders in the Δ, whereas `CJD-Bind2` inspects type binders. Both

$$\boxed{\Gamma \vdash E : T \rightsquigarrow J \textbf{ in } S}$$

(CJ-Var)
$$\frac{(x_1 : T \mapsto x_2) \in \Delta}{\Gamma \vdash x_1 : T \rightsquigarrow x_2 \textbf{ in } \{\}}$$

(CJ-Abs)
$$\frac{\Gamma; \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S}{\Gamma \vdash \lambda\Delta.E : \forall\Delta.T \rightsquigarrow J \textbf{ in } S}$$

(CJ-App)
$$\frac{\begin{array}{c}\Gamma \vdash E_1 : \forall(x : T_2)\Delta.T_1 \rightsquigarrow J_1 \textbf{ in } S_1 \\ \Gamma \vdash E_2 : T_2 \rightsquigarrow J_2 \textbf{ in } S_2 \qquad \Delta; T_1 \Downarrow T_3 \\ f,\ x_f\ \mathit{fresh}\end{array}}{\Gamma \vdash E_1\, E_2 : T_3 \rightsquigarrow x_f \textbf{ in } S_1 \uplus S_2 \uplus S_3}$$

```
$S_3$ := {
  Function f = $J_1$;
  f.arg = $J_2$;
  f.apply();
  $\langle T_3 \rangle$ $x_f$ = ($\langle T_3 \rangle$) f.res;}
```

(CJ-TApp)
$$\frac{\begin{array}{c}\Gamma \vdash E : \forall\alpha\Delta.T_2 \rightsquigarrow J \textbf{ in } S \\ \Delta; T_2 \Downarrow T_3\end{array}}{\Gamma \vdash E\, T_1 : T_3[T_1/\alpha] \rightsquigarrow J \textbf{ in } S}$$

$$\boxed{\Gamma; \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S}$$

(CJD-Empty)
$$\frac{\Gamma \vdash E : T \rightsquigarrow J \textbf{ in } S}{\Gamma; \epsilon \vdash E : T \rightsquigarrow J \textbf{ in } S}$$

(CJD-Bind1)
$$\frac{\begin{array}{c}\Gamma\ (y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S \\ \mathit{FC},\ x_1,\ x_2,\ f\ \mathit{fresh}\end{array}}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f \textbf{ in } S'}$$

```
$S'$ := {
  class FC extends Function {
    Function $x_1$ = this;
    void apply() {
      $\langle T_1 \rangle$ $x_2$ = ($\langle T_1 \rangle$) x1.arg;
      $S$;
      res = J;
    }
  };
  Function f = new FC();}
```

(CJD-Bind2)
$$\frac{\Gamma\ \alpha; \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S}{\Gamma; \alpha\ \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S}$$

$$\boxed{\Delta; T_1 \Downarrow T_2}$$

(D-Empty)
$$\frac{}{\epsilon; T \Downarrow T}$$

(D-NonEmpty)
$$\frac{}{\Delta; T \Downarrow \forall\Delta.T}$$

Translation of Closure F types to Java types:

$\langle \alpha \rangle$ = Object
$\langle \forall\Delta.T \rangle$ = Function

Figure 2. Type-Directed Translation from Closure F to Java

rules are straightforward. CJD-Bind1 shows the more involved case with value binders. Given evidence for resolving E and given bound variable y of type T_1, we wrap the generated Java expression J and its deriving statements S as follows. We create an inner class with a fresh name *FC*, extending the abstract *Function* class. This class contains a self-reference field with a fresh name x_1. This is for the case when functions nested in this function's body need to refer to it. In the function's body, we first create an alias for its argument with a fresh name x_2, then execute all statements S deriving its resulting Java expression J that we assign as the output of this function. Finally, we create a fresh alias f for the instance of the mentioned function, representing the inner class *FC*. Note that we abuse the notation slightly by assuming that we can inspect the top of Δ in CJD-Bind1 and CJD-Bind2.

In the third set of rules for resolving types, we have the typing judgment for resulting types of value and type applications (CJ-App and CJ-TApp):
$\Delta; T_1 \Downarrow T_2$
We have two options for correct transformations in value or type applications (D-Empty and D-NonEmpty). If we have an empty environment, the type is resolved to itself. Otherwise, we abstract the type over an environment.

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 (a subset of Java in our case). Unfortunately, as far as we know, the subset of Java that we use has not been completely formalized yet. There are two key Java features that are problematic: 1) *an imperative subset of Java*; 2) *inner classes*. Featherweight Java [Igarashi et al. 2001] is clearly not enough for our purposes as it lacks both features. We know of existing work [Igarashi and Pierce 2002; Bierman et al. 2003] that formalizes either 1) or 2), but not both. Therefore, before embarking on the formalization of the two proofs, we must first formalize the semantics of a subset of Java with both features. Since this is a major endeavor, which is not in-scope for this paper, we leave this and proving the properties of the translation 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 [Abelson et al. 1998]. 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:

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : \forall(x : T_2)\Delta.T_1 \rightsquigarrow J_1 \textbf{ in } S_1 \\ \Gamma \vdash E_2 : T_2 \rightsquigarrow J_2 \textbf{ in } S_2 \qquad \Delta; T_1 \Downarrow T_3 \\ x_f,\ f,\ c\ \mathit{fresh}\end{array}}{\Gamma \vdash E_1\, E_2 : T_3 \rightsquigarrow x_f \textbf{ in } S_1 \uplus S_2 \uplus S_3}$$

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 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.

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.

5.1 Syntax Extensions and Java Interoperability

Our actual implementation of **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 type-check 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 Additional Optimizations

Our compiler also performs standard optimizations that are not accounted by the basic translations in Sections 3 and 4. 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 [Marlow and Jones 2006] 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 func-

[1] **Note to reviewers:** We submit a file bundle with the current **FCore** implementation together with the paper submission. But due to the anonymous process of PLDI, we currently do not include an URL to our implementation in the paper. The URL will be provided if the submission is accepted.

tions: 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. The idea employed in our implementation is similar to the alternative encoding of curried functions briefly presented in Section 2.1.

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 [Odersky 2014]. It directly compiles down to the Java bytecode and applies various optimizations. In order to encode mutually recursive tail calls, we used 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 [Hickey 2008]. 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 [Friberg et al. 2014]. 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 4 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.

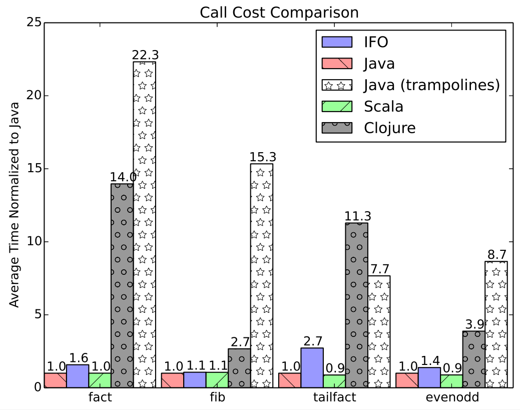
For the high input values, the method-based implementations threw a `StackOverflow` exception in default JVM settings, unlike IFOs and trampoline implementations which can continue executing with this input. IFOs ran 3.9 to 12.2-times faster (excluding Clojure) than trampoline implementations. Again, 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

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



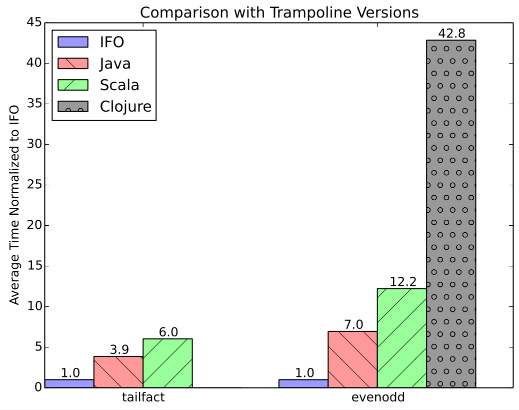
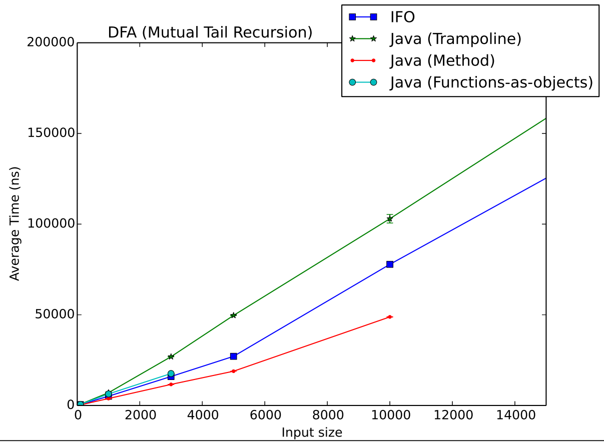


Figure 3. The isolated call behavior experiments: the reported times are averages of 10 measured runs and corresponding standard deviations. The plots are normalized to Java's (left table and plot) and IFO's (right table and plot) results – the lower, the faster.

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



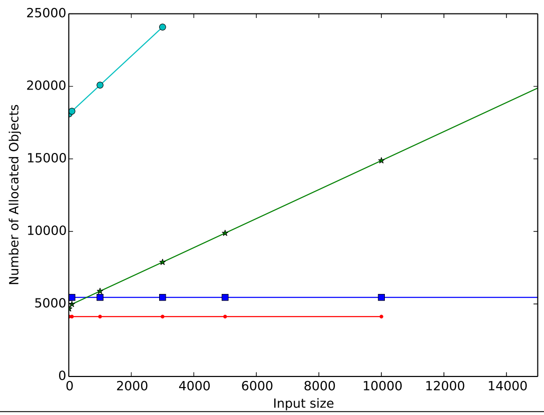


Figure 4. The DFA encoding: the reported times are averages of 10 measured runs and corresponding standard deviations; the last two columns show the minimum and maximum numbers of total allocated objects on heap from isolated profiled runs with all input lengths. Due to space limitations, the x-axes 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 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 [O'Hair 2004], 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 [Krishnamurthi 2006]. 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 [Steele and Sussman 1975] 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 [Kennedy and Syme 2004]. 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 like 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 [Benton et al. 1998] 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 [Wakeling 1999; Tullsen 1996; Choi et al. 2001] used different variations of the *graph reduction machine* [Wadsworth 1971] for their code generation, whereas we translate from System F. Nu [Dyer and Rajan 2008] 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 [Shao and Appel 1994] 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 [Steele 1977], implementors of FP languages often recognize TCE as a necessary feature. Steele's Rabbit Scheme compiler [Steele 1978] 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." [Baker 1995] 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 [Odersky 2014] or Clojure [Hickey 2008], 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 [Benton et al. 1998] or Frege [Wechsung 2014]. Later work [Minamide 2003] 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) [Choi et al. 2001]. With such explicit stack for TCE, the approach from Steele's pioneering work [Steele 1978] 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 [Schinz and Odersky 2001] 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 [League et al. 2001], 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 [Clements and Felleisen 2004] refuted this claim. There exists one modified Java HotSpot^{TM}VM [Schwaighofer 2009] with TCE support. The research Maxine VM with its new self-optimizing runtime system [Würthinger et al. 2013] 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 [Tismer 2000]. 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 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. 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 [Dragos 2010]. 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.

References

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), 1998. doi: 10.1023/A:1010051815785.

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

H. G. Baker. CONS should not CONS its arguments, part II. *ACM SIGPLAN Notices*, 30(9), 1995. doi: 10.1145/214448.214454.

N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java Bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 1998. doi: 10.1145/289423.289435.

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.

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*, 2001.

J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6), 2004. doi: 10.1145/1034774.1034778.

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

R. Dyer and H. Rajan. Nu: A Dynamic Aspect-oriented Intermediate Language Model and Virtual Machine for Flexible Runtime Adaptation. In *Proceedings of the 7th International Conference on Aspect-oriented Software Development*, 2008. doi: 10.1145/1353482.1353505.

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

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

R. Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, 2008. doi: 10.1145/1408681.1408682.

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

A. Igarashi and B. C. Pierce. On inner classes. *Information and Computation*, 177(1), 2002. doi: 10.1006/inco.2002.3092.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001. doi: 10.1145/503502.503505.

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), 2004. doi: 10.1002/cpe.801.

S. Krishnamurthi. Educational Pearl: Automata via Macros. *Journal of Functional Programming*, 16(3), 2006. doi: 10.1017/S0956796805005733.

C. League, V. Trifonov, and Z. Shao. Functional Java Bytecode. *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, 2001.

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), 2006.

Y. Minamide. Selective Tail Call Elimination. In *Proceedings of the 10th International Conference on Static Analysis*, 2003.

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

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

J. C. Reynolds. Towards a Theory of Type Structure. In *Symposium on Programming*, 1974.

M. Schinz and M. Odersky. Tail call elimination on the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*, 59 (1), 2001. doi: 10.1016/S1571-0661(05)80459-1.

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

Z. Shao and A. W. Appel. Space-efficient closure representations. *ACM SIGPLAN Lisp Pointers*, VII(3), 1994. doi: 10.1145/182590.156783.

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*, New York, New York, USA, 1977. doi: 10.1145/800179.810196.

G. L. Steele. Rabbit: A compiler for scheme. Technical report, Massachusetts Institute of Technology, 1978.

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

C. Tismer. Continuations and stackless Python. In *Proceedings of the 8th International Python Conference*, volume 1, 2000.

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

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

D. Wakeling. Compiling lazy functional programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6), 1999. doi: 10.1017/S0956796899003603.

I. Wechsung. Frege, 2014. URL github.com/Frege/frege.

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*, 2013. doi: 10.1145/2509578.2509581.