λ-Based Object-Oriented Programming

Marco Servetto
ECS, Victoria University of Wellington, New Zealand

Elena Zucca
DIBRIS, University of Genova, Italy

Abstract

We show that a minimal subset of Java 8 excluding classes supports a simple and natural programming style, which we call *λ-based object-oriented programming*. That is, on one hand the programmer can use *tuples* in place of objects (class instances), and tuples can be desugared to lambdas following their classical encoding in the λ-calculus. On the other hand, lambdas can be equipped with additional behaviour, thanks to the fact that they may implement interfaces with default methods, hence inheritance and dynamic dispatch are still supported. We formally describe the encoding by a translation from FJ_λ, an FJ variant including lambdas and interfaces with default methods, to FJ^-_λ, a subset of FJ_λ with no classes (hence no constructors and fields). We provide several examples illustrating this novel programming style.

2012 ACM Subject Classification Software and its engineering → Object oriented languages; Software and its engineering → Functional languages

Keywords and phrases Programming paradigms, Java, lambda-calculus

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.21

Category Pearl

Acknowledgements We warmly thank the anonymous referees for suggestions which greatly improved the paper.

1 Introduction

Java 8 introduced lambdas and default interface methods. In Java 8, an interface with exactly one abstract method can be instantiated with the convenient lambda syntax. Lambdas are intended to represent first class functions; indeed, when such abstract method is invoked on a lambda, the body is executed as in standard application (β-rule). However, thanks to default methods, also interfaces with multiple methods can be instantiated with lambdas. In this way, lambdas also behave as regular objects, making it possibile to write object-oriented code without any need of classes and constructor invocations. Consider the following Java code example:

```
interface Person{
  String name();
  default String greet(){
    return "Hi, I'm "+this.name()+"; nice to meet you!";
  }
}
Person bob = ()->"bob";
bob.greet();

interface GamerPerson extends Person{
  default String greet(){
    return "Hi, I'm "+this.name()+"; and I love computer games!";
  }
}
Person p = (GamerPerson)()->"charles";
p.greet();// dynamic dispatch
```

In this example, the lambda, rather than be used as a first-class function, serves a role similar to an object with a `name` field. Correspondingly, the unique abstract method `String name();` behaves as a getter, even though a field `name` is never really declared.

At first, this programming style may look an odd curiosity or even a form of misguided code obfuscation. We started playing with these kinds of programming patterns as a funny exercise, to see how far we could push this; unexpectedly, in the end we realized that it is possible to program in a pretty natural way in such a paradigm, hence in a small Java 8 subset excluding two of the most iconic Java keywords: `class` and `new`. We call this programming style *λ-based object-oriented programming*: it can be seen as a novel way to conciliate OO and functional programming, that neither simply encodes one approach into the other, nor just merges constructs from both paradigms.

In other words, our aim is to explore a programming style which is basically *functional* (first-class values are only lambdas) but where functions may have, besides application, additional behaviour, thus supporting inheritance and dynamic binding, hence code reuse, as in object-oriented programming. Moreover, our aim is to present (and encode) this approach by a minimal and clean calculus. In other words, our focus here is not on increasing expressive power, but on simplicity: encoding the same features with fewer constructs.

To illustrate λ-based object-oriented programming, first of all in Section 2 we present a core language supporting this paradigm, by means of a simple calculus FJ_λ^-, in the style of Featherweight Java (FJ) [8]. An FJ_λ^- program is a table of interfaces, and expressions are only variables, method calls and lambdas. In a sense, this expression language is lambda-calculus enriched by the foundational feature of object-oriented programming, that is, dynamically dispatched method call. While some methods are implemented with lambdas, most code is provided inside (default) method bodies keeping the natural code organization typical of OO programming.

Then, we enrich FJ_λ^- by classes, fields and constructors, obtaining an extended calculus FJ_λ, and in Section 3 we show a simple translation from FJ_λ to FJ_λ^-, thus formally proving that such constructs are redundant language features. The translation is inspired by the classical encoding of tuples in the lambda-calculus; however, some additional work is needed to use the encoding in a language with only nominal types (interface names). We show that the translation preserves typing and semantics.

In Section 4, we provide several programming examples illustrating how to rely on this paradigm in a real language, notably a Java 8 subset excluding classes. Finally, in Section 5 we discuss possibile extensions and relation with other works, and in Section 6 we summarize the contribution of the paper and outline future directions.

2 The FJ_λ^- and FJ_λ calculi

Syntax, reduction rules, and typing rules of FJ_λ^- are given in Figure 1. We write Ds as metavariable for $D_1 \ldots D_n$, $n \geq 0$, and analogously for other sequences.

As anticipated, an FJ_λ^- program is a table of interfaces, and expressions are only variables, method calls and λ-expressions (lambdas). In Java, λ-abstractions can only be typed when occurring in a context requiring a given type (called the *target type*). Here, we directly assume lambdas to be annotated with their interface type, since the issue of deriving this annotation from the context is orthogonal to our topic, and has been faced in other works [2, 6]. Moreover, in the calculus we use the lambda-calculus syntax, both to help readability and to stress that it can be seen both as a subset of Java, and, conversely, as a λ-calculus equipped with an interface table.

P	$::=$	Ds	program
D	$::=$	$\texttt{interface}\ I\ \texttt{extends}\ \mathit{Is}\ \{\ \mathit{IMs}\ \}$	declaration
IM	$::=$	$\mathit{MH};\mid \texttt{default}\ M$	interface method
MH	$::=$	$T\ m\texttt{(}T_1\ x_1,\ldots,T_n\ x_n\texttt{)}$	method header
M	$::=$	$\mathit{MH}\ \{\texttt{return}\ e;\}$	method
R,S,T	$::=$	I	type
e	$::=$	$x \mid e\texttt{.}m\texttt{(}\mathit{es}\texttt{)} \mid (\lambda \mathit{xs}.e)^I$	expression
v	$::=$	$(\lambda \mathit{xs}.e)^I$	value
$\mathcal{E}$	$::=$	$[\,]\texttt{.}m\texttt{(}\mathit{es}\texttt{)} \mid v\texttt{.}m\texttt{(}\mathit{vs}\ [\,]\ \mathit{es}\texttt{)}$	evaluation context
Γ	$::=$	$x_1{:}T_1\ldots x_n{:}T_n$	typing context

$$\text{(CTX)}\ \frac{e \rightarrow_P e'}{\mathcal{E}[e] \rightarrow_P \mathcal{E}[e']} \qquad \text{(INVK)}\ \frac{}{v\texttt{.}m\texttt{(}\mathit{vs}\texttt{)} \rightarrow_P e[\mathit{xs}{\leftarrow}\mathit{vs}][\texttt{this}{\leftarrow}v]}\ \begin{array}{l}\mathsf{typeof}(v) = T\\ \mathsf{mbody}_P(T,m) = \langle \mathit{xs},\ e\rangle\end{array}$$

$$(\beta)\ \frac{}{v\texttt{.}m\texttt{(}\mathit{vs}\texttt{)} \rightarrow_P e[\mathit{xs}{\leftarrow}\mathit{vs}][\texttt{this}{\leftarrow}v]}\ \begin{array}{l}v = (\lambda \mathit{xs}.e)^I\\ !\mathsf{absmeth}_P(I) = m\end{array}$$

$$\text{(T-PROG)}\ \frac{\vdash_P D_1 \ldots \vdash_P D_n}{\vdash P}\ P = D_1 \ldots D_n$$

$$\text{(T-INTERFACE)}\ \frac{I \vdash_P \mathit{IM}_1 \ldots I \vdash_P \mathit{IM}_n}{\vdash_P \texttt{interface}\ I\ \texttt{extends}\ \mathit{Is}\ \{\ \mathit{IM}_1 \ldots \mathit{IM}_n\ \}}\ \mathit{Is} \subseteq \mathsf{inames}(P)$$

$$\text{(T-MH)}\ \frac{}{T \vdash_P R\ m\texttt{(}T_1\ x_1,\ldots,T_n\ x_n\texttt{)}}\ \begin{array}{l}\{R, T_1,\ldots,T_n\} \subseteq \mathsf{tnames}(P)\\ (T \leq_P S \text{ and } \mathsf{mtype}_P(S,m) = \mathit{Ts} \rightarrow R') \text{ imply}\\ \quad\quad \mathit{Ts} = T_1 \ldots T_n\\ \quad\quad R' = R\end{array}$$

$$\text{(T-ABS)}\ \frac{T \vdash_P \mathit{MH}}{T \vdash_P \mathit{MH};} \qquad \text{(T-DEFAULT)}\ \frac{T \vdash_P M}{T \vdash_P \texttt{default}\ M}$$

$$\text{(T-METH)}\ \frac{\vdash_P \mathit{MH}\quad \Gamma \vdash_P e : R'}{T \vdash_P \mathit{MH}\ \{\texttt{return}\ e;\}}\ \begin{array}{l}\mathit{MH} = R\ m\texttt{(}T_1\ x_1,\ldots,T_n\ x_n\texttt{)}\\ \Gamma = x_1{:}T_1\ldots x_n{:}T_n\ \texttt{this}{:}T\\ R' \leq_P R\end{array}$$

$$\text{(T-VAR)}\ \frac{}{\Gamma \vdash_P x : T}\ \Gamma(x) = T$$

$$\text{(T-INVK)}\ \frac{\Gamma \vdash_P e_i : T_i\quad \forall i \in 0..n}{\Gamma \vdash_P e_0\texttt{.}m\texttt{(}e_1,\ldots,e_n\texttt{)} : R}\ \begin{array}{l}\mathsf{mtype}_P(T_0,m) = S_1 \ldots S_n \rightarrow R\\ T_i \leq_P S_i\quad \forall i \in 1..n\end{array}$$

$$\text{(T-LAM)}\ \frac{\Gamma[x_1{:}T_1\ldots x_n{:}T_n] \vdash_P e : S}{\Gamma \vdash_P (\lambda x_1 \ldots x_n.e)^I : I}\ \begin{array}{l}!\mathsf{absmtype}_P(I) = T_1 \ldots T_n \rightarrow R\\ S \leq_P R\end{array}$$

Figure 1 Formal definition of FJ^{-}_{λ}.

The formal definition is straightforward. Given a program P, $\mathsf{inames}(P)$ and $\mathsf{tnames}(P)$ are the declared interface names and type names; $\mathsf{typeof}(v)$ the (dynamic) type of value v, which for a lambda is its annotation; $\mathsf{mtype}_P(T, m)$ and $\mathsf{mbody}_P(T, m)$ the type and body of method m in T, if any; $!\mathsf{absmeth}_P(I)$ and $!\mathsf{absmtype}_P(I)$, only defined if I has exactly one abstract method[1], the name and type of such method; finally, $\leq_P$ the reflexive and transitive closure of the `extends` relation. For brevity, we omit the straightforward formal definitions. Only note that we require mtype and mbody, defined as in FJ, to be actually functions; this constraint implies that an interface I cannot inherit the same method m with different signatures, and cannot inherit more than one default method m, unless m is declared by I as well.

We denote by $\mathcal{E}[e]$ the expression obtained by filling the hole of the context $\mathcal{E}$ with e, and by $e[xs{\leftarrow}vs]$ the expression obtained from e by replacing variables xs with values vs. Typing contexts are assumed to represent finite maps from variables into types, hence the notation $\Gamma(x)$ is well-defined; we denote by $\Gamma[\Gamma']$ the type context which coincides with Γ' when the latter is defined, with Γ otherwise.

In Figure 2, we extend $\mathrm{FJ}^{-}_{\lambda}$ with classes, fields and constructor invocations. New productions are emphasized, and we only write the new reduction and typing rules. The extended calculus is similar to other calculi extending FJ with interfaces and lambdas [2, 6]. Here, we do not include subclassing, since our focus is to show that the role of classes as object's generators can be replaced by instantiating functional interfaces with lambdas; instead, we expect the role of subclassing for code reuse to be achieved by interface inheritance and default methods.

The reduction relation $\rightarrow_P$ is defined on closed expressions. The calculus enjoys standard properties; notably, reduction is deterministic, and the type system is sound, that is, reduction of (closed) well-typed expressions with respect to well-typed programs does not go stuck, as formally stated below. We write $\rightarrow^{\star}_P$ for the transitive and reflexive closure of $\rightarrow_P$.

▶ **Theorem 1** (Determinism). *If $e \rightarrow_P e_1$ and $e \rightarrow_P e_2$, then $e_1 = e_2$.*

Proof. By structural induction on e, observing that at most one (instantation of meta-)rule is applicable. ◀

▶ **Theorem 2** (Soundness). *If $\vdash P$, and $\emptyset \vdash_P e : T$, and $e \rightarrow^{\star}_P e'$, then either e' is a value or $e' \rightarrow_P e''$ for some e''.*

Proof. Straightforward adaptation of the proof provided for a richer language in [2]. ◀

Set $v^{\infty} ::= v \mid \infty$. The relation $e \Rightarrow_P v^{\infty}$, associating to an expression e its *semantics* in P, is defined as follows:

- $e \Rightarrow_P v$ if $e \rightarrow^{\star}_P v$
- $e \Rightarrow_P \infty$ if e has an infinite reduction sequence in P.

If $\vdash P$, and $\emptyset \vdash_P e : T$, then Theorem 1 and Theorem 2 above ensure that the semantics of e in P is well-defined, that is, $e \Rightarrow_P v^{\infty}$ for a unique v^{∞}.

[1] That is, is a *functional* interface.

P	$::=$	Ds	program
D	$::=$	$\texttt{interface}\ I\ \texttt{extends}\ Is\ \{\ IMs\ \}$	declaration
		$\mid\ \texttt{class}\ C\ \texttt{implements}\ Is\ \{Fs\ Ms\}$	
IM	$::=$	$MH \mid \texttt{default}\ M$	interface method
MH	$::=$	$T\ m\ \texttt{(}T_1\, x_1, \ldots, T_n\, x_n\texttt{)}$	method header
M	$::=$	$MH\, \{\texttt{return}\ e\texttt{;}\}$	method
F	$::=$	$T\, f\, \texttt{;}$	field
T	$::=$	$I \mid C$	type
e	$::=$	$x \mid e\texttt{.}m\texttt{(}es\texttt{)} \mid (\lambda xs.e)^I \mid \texttt{new}\ C\texttt{(}es\texttt{)} \mid e.f$	expression
v	$::=$	$(\lambda xs.e)^I \mid \texttt{new}\ C\texttt{(}vs\texttt{)}$	value
$\mathcal{E}$	$::=$	$[\,]\texttt{.}m\texttt{(}es\texttt{)} \mid v\texttt{.}m\texttt{(}vs\,[\,]\,es\texttt{)} \mid \texttt{new}\ C\texttt{(}vs\,[\,]\,es\texttt{)} \mid [\,].f$	evaluation context
Γ	$::=$	$x_1{:}T_1 \ldots x_n{:}T_n$	typing context

$$\text{(FIELD)}\ \frac{}{\texttt{new}\ C(v_1 \ldots v_n).f_i \rightarrow_P v_i} \quad \begin{array}{l} \mathsf{fields}_P(C) = T_1\ f_1\,\texttt{;} \ldots T_n\ f_n\,\texttt{;} \\ i \in 1..n \end{array}$$

$$\text{(T-CLASS)}\ \frac{I \vdash_P M_1 \ldots I \vdash_P M_n}{\vdash_P \texttt{class}\ C\ \texttt{implements}\ Is\ \{T_1\ f_1\,\texttt{;} \ldots T_k\ f_k\,\texttt{;}\quad M_1 \ldots M_n\}} \quad \begin{array}{l} T_1 \ldots T_k \subseteq \mathsf{tnames}(P) \\ I_s \subseteq \mathsf{inames}(P) \end{array}$$

$$\text{(T-FIELD)}\ \frac{\Gamma \vdash_P e : C}{\Gamma \vdash_P e.f_i : T_i} \quad \begin{array}{l} \mathsf{fields}_P(C) = T_1\ f_1\,\texttt{;} \ldots T_n\ f_n\,\texttt{;} \\ i \in 1..n \end{array}$$

$$\text{(T-NEW)}\ \frac{\Gamma \vdash_P e_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash_P \texttt{new}\ C(e_1, \ldots, e_n) : C} \quad \begin{array}{l} \mathsf{fields}_P(C) = S_1\ f_1\,\texttt{;} \ldots S_n\ f_n\,\texttt{;} \\ T_i \leq_P S_i \quad \forall i \in 1..n \end{array}$$

Figure 2 Formal definition of FJ$_\lambda$.

3 Translation

First we explain the translation on a simple example: a class `Pair` with two fields, where for simplicity types `A` and `B` can be thought to be primitive types, e.g., `int` and `boolean`, respectively.

```
class Pair {
  A fst;
  B snd;
}
```

An instance of a class with n fields is essentially a tuple with n components, so our translation is based on the classical encoding of tuples in λ-calculus: a tuple is a function which, taken a *selector*, returns the corresponding component. In the example, a pair, e.g., `mypair` $= \langle 1, \texttt{true}\rangle$, is encoded by the function $\lambda\texttt{s}.\texttt{s}\,1\,\texttt{true}$, and there are only two expected selectors: `fst` $= \lambda x.\lambda y.x$ and `snd` $= \lambda x.\lambda y.y$. For instance, `mypair fst` reduces to 1.

We consider now the problem of assigning types to the functions encoding tuples and selectors. Of course this is easy if we have polymorphic types. However, in each concrete case, there are only n different types of selectors which can be given as argument to the tuple, and for each of them a different result type. In other words, union types are enough. In a language with algebraic types, such as, e.g., Haskell, the same effect can be achieved by constructors which act as embeddings, as shown below.

```
type A = Int
type B = Bool
data AorB = FromA A | FromB B

type Fst = A -> B -> A
type Snd = A -> B -> B
data Sel = FromFst Fst | FromSnd Snd

type Pair = Sel -> AorB
--mypair = <1,True>
mypair :: Pair
mypair (FromFst s) = FromA (s 1 True)
mypair (FromSnd s) = FromB (s 1 True)

getFst :: Pair -> A
getFst p =
  let FromA a = p (FromFst (\a b -> a))
  in a

getSnd :: Pair -> B
getSnd p =
  let FromB b = p (FromSnd (\a b -> b))
  in b
```

Note that the getter methods could in principle raise a pattern matching error, but this will never happen at runtime.

The encoding in Java is based on the same idea. However, in this case the union of the result types is encoded by an interface, and the embedding of an element of type `A` into `AorB` is the constant function `(FromA)()->a`. On the other hand, it is enough to have the interface `Sel` corresponding to the various selectors, since in this case the embedding is silently obtained by subtyping.

```
interface AorB{
  default A toA(){/*error*/}
  default B toB(){/*error*/}
}

interface FromA extends AorB{ A toA();}
interface FromB extends AorB{ B toB(); }

interface Sel { AorB apply(A a, B b);}}

interface Pair {
  default A getFst(){
    return this.apply((Sel)(a,b)->(FromA)()->a).toA();}
  default B getSnd(){
    return this.apply((Sel)(a,b)->(FromB)()->b).toB();}
  AorB apply (Sel sel);
  }
```

For instance, the object `new Pair(myA,myB)` is encoded by `(Pair)s->s.apply(myA,myB)`. Also in this case the getter methods could in principle raise an error, but this will never happen at runtime. Hence, the body of the default methods `toA` and `toB` could be arbitrary well-typed expressions, which will be never executed, as indicated by the `/*error*/` comment. In the minimal syntax of FJ_λ^-, such arbitrary expressions could be the recursive calls `this.toA()` and `this.toB()`. In full Java, an exception could be thrown.

This pattern is applied for all classes with $n \geq 2$ fields, as formally defined below; for each such class $n+3$ interfaces are generated. Classes with zero or one field have specific (simpler) encodings, explained below.

As shown in the `Person` example before, a class with a single field can be encoded by an interface which defines a single abstract no-args getter method for such field, and an instance of the class can be encoded by a constant function which returns the field value.

A class with no fields, in a functional setting, has only one instance, and offers a set of methods to be invoked on such unique instance, which hence can be seen as class methods.The corresponding interface offers such methods, but also needs an abstract method, since the unique instance of the class should be encoded by a lambda. Note that an arbitrary abstract method, and an arbitrary lambda providing the implementation could be used, since the method will never be called. We conventionally use a method `dummy` with argument and return type `Void`, where `Void` is an empty interface, and as dummy lambda the identity function. In the following examples in full Java, we use a `void dummy()` method and the empty block as body of the dummy lambda.

We provide now the formal translation, denoted $[\![\_]\!]$. To the aim of this translation, we assume that, in the FJ_λ program to be translated, arguments of constructor invocations are only variable or values, since otherwise (possibly non terminating or stuck) reduction of arguments would be not simulated in the translation. This is not a restriction, since general constructor invocations can be encoded by auxiliary methods in FJ_λ, and could be translated adding local variable declarations in FJ_λ^-. Moreover, as mentioned above, we assume a declaration `interface Void {}`. Finally, we assume that all the interface and method names introduced by the translation are fresh, that is, they do not clash with existing names. We first provide the translation of class declarations. The C at the beginning of interface names is necessary to get unique names. Moreover, the `to` methods are decorated by indexes, rather than field types as in the introductory example, since two fields could have the same type.

```
⟦interface I extends Is {IM_1 ... IM_n}⟧ = interface I extends Is {⟦IM_1⟧ ... ⟦IM_n⟧}

⟦class C implements Is {MH_1 {return e_1;} ... MH_n {return e_n;}}⟧ = (zero fields)
    interface C extends Is {
        default MH_1 {return ⟦e_1⟧;}
        ...
        default MH_n {return ⟦e_n⟧;}
        Void dummy(Void x);
    }

⟦class C implements Is {T f; MH_1 {return e_1;} ... MH_n {return e_n;}}⟧ = (one field)
    interface C extends Is {
        default MH_1 {return ⟦e_1⟧;}
        ...
        default MH_n {return ⟦e_n⟧;}
        T getf();
    }

⟦class C implements Is
            {T_1 f_1; ... T_k f_k; MH_1 {return e_1;} ... MH_n {return e_n;}}⟧ = (≥ 2 fields)
    interface CUnion {
        default T_1 to1(){return /*error*/;}
        ...
        default T_k tok(){return /*error*/;}
    interface CFrom1 extends CUnion{to1();}
    ...
    interface CFromk extends CUnion{tok();}
    interface CSel{CUnion apply(T_1 x_1, ..., T_k x_k);}
    interface C extends Is {
        default MH_1 {return ⟦e_1⟧;}
        ...
        default MH_n {return ⟦e_n⟧;}
        default T_1 getf_1(){return this.apply((λx_1 ... x_k.(λ.x_1)^{CFromk})^{CSel}).to1();}
        ...
        default T_k getf_k(){return this.apply((λx_1 ... x_k.(λ.x_k)^{CFromk})^{CSel}).tok();}
        CUnion apply(CSel s);
    }
```

The translation of expressions is given below.

Set $xv ::= x \mid v$.
$[\![x]\!] = x$
$[\![e.\,m(e_1, \ldots, e_n)]\!] = [\![e]\!].\,m([\![e_1]\!], \ldots, [\![e_n]\!])$
$[\![(\lambda xs.e)^I]\!] = (\lambda xs.[\![e]\!])^I$
$[\![\texttt{new } C()]\!] = (\lambda x.x)^C$
$[\![\texttt{new } C(xv)]\!] = (\lambda.[\![xv]\!])^C$
$[\![\texttt{new } C(xv_1, \ldots xv_n)]\!] = (\lambda\texttt{s.s.apply}([\![xv_1]\!], \ldots, [\![xv_n]\!]))^C$
$[\![e.f]\!] = [\![e]\!].\texttt{get}f\texttt{()}$

The translation preserves typing and semantics, as formally stated below.

▶ **Theorem 3.** *If $\vdash P$ and $\emptyset \vdash_P e : T$, then the following hold:*
1. $\vdash [\![P]\!]$.
2. $\emptyset \vdash_{[\![P]\!]} [\![e]\!] : T$.
3. $e \Rightarrow_P v^\infty$ *iff* $[\![e]\!] \Rightarrow_{[\![P]\!]} [\![v^\infty]\!]$, *where* $[\![\infty]\!] = \infty$.

Proof. The first two points can be proved by straightforward induction on the typing rules. The third point is a consequence of the following properties:

- e is a value iff $[\![e]\!]$ is a value
- if $e \rightarrow_P e'$, then $[\![e]\!] \rightarrow^\star_{[\![P]\!]} [\![e']\!]$

The first property trivially holds, since e is closed, whereas the second one can be proved by structural induction on e. We show the most interesting case, which is $e.f$. There are the following subcases:

- Rule (CTX) is applicable, hence we have $e \rightarrow_P e'$ and $e.f \rightarrow_P e'.f$. By inductive hypothesis, $[\![e]\!] \rightarrow^\star_{[\![P]\!]} [\![e']\!]$ in n steps. Moreover, $[\![e.f]\!] = [\![e]\!]\texttt{.get}f\texttt{()}$, which is of shape $\mathcal{E}[[\![e]\!]]$, then the thesis follows by applying n times rule (CTX).
- Rule (FIELD) is applicable. We distinguish the following subcases:
  - $n = 1$: we have $\texttt{new}\ C(v).f \rightarrow_P v$ and $\mathsf{fields}_P(C) = T\,f\texttt{;}$; moreover, $[\![\texttt{new}\ C(v).f]\!] = (\lambda.)^C\texttt{.get}f\texttt{()}$. By the second clause in the translation of classes, class C exists in $[\![P]\!]$ and has a unique abstract method $\texttt{get}f$, hence $(\lambda.)^C\texttt{.get}f\texttt{()} \rightarrow_{[\![P]\!]} [\![v]\!]$ by rule (β).
  - $n > 1$: we have $\texttt{new}\ C(v_1 \ldots v_n).f \rightarrow_P v_i$ and $\mathsf{fields}_P(C) = T_1\ f_1\texttt{;} \ldots T_n\ f_n\texttt{;}$. Moreover, $[\![\texttt{new}\ C(v_1 \ldots v_n).f]\!] = (\lambda\texttt{s.s.apply(}[\![v_1]\!], \ldots, [\![v_n]\!]\texttt{)})^C\texttt{.get}f_i\texttt{()}$. By the third clause in the translation of classes, classes C, $C\texttt{From}T_i$, and $C\texttt{Sel}$ exist in $[\![P]\!]$ with the specified methods.
    Hence, $(\lambda\texttt{s.s.apply(}[\![v_1]\!], \ldots, [\![v_n]\!]\texttt{)})^C\texttt{.get}f_i\texttt{()} \rightarrow^\star_{[\![P]\!]} [\![v_i]\!]$ by the following reduction sequence in $[\![P]\!]$, where we write the computational rule applied at each step:

$$(\lambda\texttt{s.s.apply(}[\![v_1]\!], \ldots, [\![v_n]\!]\texttt{)})^C\texttt{.get}f_i\texttt{()} \rightarrow \quad (\text{INVK})$$

$$(\lambda\texttt{s.s.apply(}[\![v_1]\!], \ldots, [\![v_n]\!]\texttt{)})^C\texttt{.apply((}\lambda x_1 \ldots x_n.(\lambda.x_i)^{C\texttt{From}T_i}\texttt{)}^{C\texttt{Sel}}\texttt{).to}i\texttt{()} \rightarrow \quad (\beta)$$

$$(\lambda x_1 \ldots x_k.(\lambda.x_i)^{C\texttt{From}T_i}\texttt{)}^{C\texttt{Sel}}\texttt{.apply(}[\![v_1]\!], \ldots, [\![v_n]\!]\texttt{).to}i\texttt{()} \rightarrow \quad (\beta)$$

$$(\lambda.[\![v_i]\!])^{C\texttt{From}T_i}\texttt{.to}i\texttt{()} \rightarrow \quad (\beta)$$

$$[\![v_i]\!]$$ ◀

4 Programming with only lambdas

We describe how this programming paradigm can be effectively used. In these examples, we assume a language extended with numbers and string literals, local variable declarations, type inference for lambdas and generics as in Java. First of all, to avoid to have to manually encode tuples by lambdas, we expect this encoding to be provided by libraries. That is, we expect to have library types `Tuple2<T1,T2>`, `Tuple3<T1,T2,T3>`, and so on. In this way, the programmer can just extend the opportune `Tuple` interface to encode a class with many fields.

```
//Library code:
//case for 2 fields
interface Union2<T1,T2>{
  default T1 to1(){ /*error*/}
  default T2 to2(){ /*error*/}
  }
interface From2_1<T1,T2> extends Union2<T1,T2>{ T1 to1(); }
interface From2_2<T1,T2> extends Union2<T1,T2>{ T2 to2(); }
interface Tuple2<T1,T2>{
  Union2<T1,T2> apply(Sel2<T1,T2> sel);
  default T1 get1(){
    return this.apply(
      (x1,x2)->(From2_1<T1,T2>)()->x1
      ).to1();
  }
  default T2 get2(){/*as above, using From2_2/to2()*/}
  }
interface Sel2<T1,T2>{ Union2<T1,T2> apply(T1 f1, T2 f2); }

...//case for 3 or more fields

interface Tuple0 { void dummy(); }

interface Function<T,R>{ R apply(T t);}//as in Java 8
interface Bifunction<T1,T2,R>{ R apply(T1 t1, T2 t2);}
//... other functional interfaces
```

With such simple standard library, we can write an extended version of the example in the introduction as follows:

```
interface Person extends Tuple2<String,Integer>{
  default String name(){return this.get1();}
  default Integer age(){return this.get2();}
  default String greet(){return "Hi, I'm "+this.name();}
  ...
  }
Bifunction<String,Integer,Person> makePerson
  = (name,age)->s->s.apply(name,age);
Person bob = makePerson.apply("bob",23);
bob.greet();
```

The code above shows that it is easy to encode classes with fields and instantiate them. It is also easy to abstract over object creation and provide factories, as the `Bifunction` above.

In the second example, we show how we can encode private methods; a library may want to encapsulate the richer `ConcretePoint` implementation and only expose a minimal `Point` interface.

```
//Point library code:
/**Point docs*/
interface Point{Integer x(); Integer y(); Integer distance(Point p);}
/**ConcretePoint is only for internal library usage
and may change in a future release*/
interface ConcretePoint extends Point, Tuple2<Integer,Integer>{
  //point implemented with a Tuple
  default Integer x(){return this.get1();}
  default Integer y(){return this.get2();}
  default Integer distance(Point p){/*uses aux*/}
  default Integer aux(Point p){..} //"private" method
  }
/**NewPoint docs do not need to mention ConcretePoint*/
interface NewPoint extends Tuple0, Bifunction<Integer,Integer,Point>{
  default Point apply(Integer x,Integer y){
    return (ConcretePoint) s->s.apply(x,y);
    }
  }
//User code
Point myPoint = ((NewPoint)()->{}).apply(3,5);
myPoint.distance(myPoint);//ok
//myPoint.aux(myPoint);//no
```

The interface `NewPoint` extends `Tuple0`, leaving the `dummy` method abstract, hence, as explained before, is expected to be instantiated by the dummy lambda. Moreover, it extends `BiFunction`, providing an implementation for the abstract method `apply`.

Note how implementation hiding is encoded by using subtyping and factories. This approach is a natural extension of what is discussed in TraitRecordJ [3], where methods of classes are implicitly private, and the only way to publicly expose behaviour is to implement an interface method.

Likely, the interface `ConcretePoint` should not be public. However, even with `ConcretePoint` exposed to the user, the implementation of `Point` is still hidden, and the user in no way can be aware that a `Point` is indeed a `ConcretePoint`, or call the `ConcretePoint.aux` method on an object provided by `NewPoint`.

Finally, we show that in this programming style recursive types are as easy as in plain Java. This is important to notice, since in other encodings of objects mutually recursive types are not trivial to handle. Consider the standard implementation of lists as pairs consisting of head and tail.

```
interface List<T>{
  T head();
  List<T> tail();
  boolean isEmpty();
  default  List<T> add(T elem){
    return (Cons<T>)s->s.apply(elem,this);
  }
}
interface Empty<T> extends List<T>,Tuple0 {
  default T head(){/*error*/}
  default List<T> tail(){/*error*/}
  default boolean isEmpty(){return true;}
}
interface Cons<T> extends Tuple2<T,List<T>>, List<T> {
  default T head(){return this.get1();}
  default List<T> tail(){return this.get2();}
  default boolean isEmpty(){return false;}
}
...//usage example
List<Integer> list = ((Empty<Integer>)()->{}).add(1).add(2).add(3);
```

Interestingly, we can also easily encode the iconic functional match:

```
interface List<T>{
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons);
  default  List<T> add(T elem){
    return (Cons<T>)s->s.apply(elem,this);
  }
}
interface Empty<T> extends List<T>,Tuple0 {
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons){
    return onEmpty.get();
  }
}
interface Cons<T> extends Tuple2<T,List<T>>, List<T> {
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons){
    return onCons.apply(this.get1(),this.get2());
  }
}
```

```
...//example methods using match
default Integer sumAll(List<Integer> l){
  return l.match(
    ()->0,
    (head,tail)->head+sumAll(tail)
  );
}
default Integer getHead(List<Integer> l, Integer orElse){
  return l.match(
    ()->orElse,
    (head,tail)->head
  );
}
```

The above example employs a variant of the visitor pattern [5] getting popular in Java 8, which provides functions as arguments to the visitor. The point to be noted here is that the λ-based paradigm makes this approach very natural. Note that `List` as defined above is open: any user can define new kinds of `List` by providing an implementation for the `match` method.

We end this section with a remark. A subtle detail in Java 8 is that lambdas cannot be used to implement generic methods. This means that the following more natural tuple encoding would only work by "desugaring" the lambda syntax.

```
public interface Tuple2<A,B> {
  <T> T apply(BiFunction<A, B, T> f);
  default A a(){return apply((a,b)->a);}
  default B b(){return apply((a,b)->b);}
}
...//code trying to instantiate Tuple2
default <A, B> Tuple2<A,B> of(A a, B b){
    //return f -> f.apply(a, b);//does not compile :-(
    return new Tuple2<A,B>(){//works without lambda syntax
      public <T> T apply(BiFunction<A,B,T> f){
        return f.apply(a,b);}};}
```

Allowing lambdas to implement generic methods was planned in a pre-release of Java 8, but this feature was removed before release. Note how the generic method `match` above is the only abstract method of `List`, thus if generic methods were instantiable with lambdas, we could omit the `Cons` code and simply write

```
interface List<T>{
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons);
  default List<T> add(T elem){
    return (onEmpty,onCons)->onCons.apply(elem,this);
  }
}
```

5 Discussion and related work

The encoding presented in Section 3 strikes a balance between being not too cumbersome and useful. A nice feature is that it does not change the way that the programmers write their program. That is, it is not a transformation that turns the program inside-out and obscures the original intent: the original classes are still there (as interfaces) and their construction happens at the same sites (with lambda syntax). On the other hand, the representation of object's state as a tuple is mainly intended to show the completeness of the approach, and could be replaced by an efficient private implementation as, e.g., values of primitive types are seen as objects in Smalltalk. The translation is shown over a simplification of the FJ calculus, which makes it composable with other work on Java semantics. Keeping the original program structure (classes/methods) means that the original program is still extensible in the same way. Also, Java reflection would still work with the output of this encoding in a natural way. So, this encoding looks quite powerful to support more features, e.g., simulate field shadowing or more complex inheritance patterns.

Since our motivation was to explore a programming style which is basically *functional*, we did not consider imperative features, as it is in FJ. They can be added, as usually in functional languages, introducing reference types and constructs for referencing, assigning and dereferencing. The classical encoding of tuples by functions immediately extends, as shown by the code below, which is the previous Haskell example rewritten in OCaml with references:

```
type a = int
type b = bool
type aOrBref = FromA of a ref | FromB of b ref
type fst = a ref -> b ref -> a ref
type snd = a ref -> b ref -> b ref
type sel = FromFst of fst | FromSnd of snd
type pair = sel -> aOrBref

let mypair : pair =
  let first = ref 1 in
  let second = ref true in
    function
      FromFst s -> FromA (s first second)
      | FromSnd s -> FromB (s first second)

let getFst : pair -> a =
  function p ->
    let FromA a = p (FromFst (function a -> function b -> a))
    in !a
let setFst : pair -> a -> unit =
  function p -> function x ->
    let FromA a = p (FromFst (function a -> function b -> a))
    in a; a := x;
```

In a Java-like language, the same can be achieved having reference types (as in C++), or, otherwise, by using local variables. Unfortunately, in Java local variables used in a lambda expression must be final or effectively final. Without this restriction, a class Person with an updatable field `name` could be encoded as follows:

```
interface Person extends Tuple2<Supplier<String>, Consumer<String>>{
  default String getName(){return this.get1().apply();}
  default void setName(String name){return this.get2().apply(name);}
}
interface PersonFactory{
  void dummy();
  default Person makePerson(String name){
    return  s -> s.apply(()->name, newName->name=newName);
  }
}
```

One commonly used way to circumvent this Java limitation is to use an array of size 1, as shown below:

```
default Person makePerson(String name){
  String[] n={name};
  return  s -> s.apply(  ()->n[0],    newName->n[0]=newName  );
}
```

In general, adding any kind of array or collections would make our encoding simpler, but, as said above, here our goal is to achieve minimality.

Though, as already said, addressing limitations of existing programming techniques was not our aim, we can mention some side benefits of the approach. Replacing "real" fields by getters/setters avoids the limitation that the type of fields must be invariant; thus more code reuse patterns become available, see the extended discussion in [12]. Moreover, Java8 interfaces support reusing code from multiple sources, as for multiple inheritance. Avoiding classes means that all the code is usable for multiple inheritance.

We already mentioned that inheritance and dynamic dispatch are supported through default methods in interfaces. In addition, we could easily extend our core to support `InterfaceName.super.methName(...)` and this would transparently allow super method calls as in Java. Moreover, the above syntax could be syntactic sugar; if any interface declared methods with a standard long name `returnType InterfaceName$methName(...){...}` with a delegator method `returnType methName(...){ return this.InterfaceName$methName(...);}` then `InterfaceName.super` could be emulated simply using the longer name for the method.

An OO style without class declarations, called interface-based object-oriented programming, has been proposed in [12], and exploited in Java by defining *Classless Java*. However, differently from our proposal, Classless Java has objects, obtained as instances of anonymous inner classes, hence fields as well, and `static` methods.

More generally, many languages support objects without classes, that is, follow the so-called "object-based" paradigm. That is, objects are not created as class instances, but, e.g., by directly writing "object literals". Our proposal goes a bit further, since in our calculus there are no objects at all; the *only values* are functions (lambdas).

Differently from the Java approach, in [11] a minimal core Java is extended to λ-expressions by adding function types, following the style of functional languages. Complexity of type inference increases in a substantial way, with respect to Java's one. Moreover, adding real function types entails that a method must have a different signature according to whether it can accept an object or a function. This contrasts with Java philosophy to fuse language innovations into the old layer.

Empirical methodologies are used in [10] to illustrate when, how and why imperative programmers adopt λ-expressions.

Classical encodings of objects [4] are as records of mutually recursive functions, where all such functions are closures over the record. In this kind of encoding, objects contain all their behaviour, and a program is fully expressed by an expression. Typing in this setting is non trivial, especially recursive types, which need to be handled by some variation of fixpoints. Our proposed language, instead, embraces the idea of an externally defined table of types (in our case interface names) also including most of the behaviour, in the form of default implementations. In this way, the key technical characteristic of OOP following, e.g., [1], that is, dynamic dispatch, is provided anyway, and such language design is more friendly toward module systems and module composition languages.

Grace [9, 7] offers an interesting middle ground: it is structurally typed, but objects can have (generic) type-alias declarations as members, and these type names can be mutually recursive. In most Grace programs, a top level object (called a *module*) plays the role of the table of types, and reduction actually takes place inside such a module object. Still, the Grace approach is more flexible than having a fixed top-level type table, since multiple objects can be nested into each other, and lexical scope and nesting allow for interesting forms of code composition. However, to make static reasoning feasible, only type members of *known objects* (objects created in a controlled way) can be used as type annotations. Moreover, due to generic type aliases, subtyping is undecidable.

Formalization of lambdas as in Java 8 have been provided in [2, 6], the former covering intersection types and default methods as well. These works focus on typing issues, notably on the fact that lambdas can only be typed when occurring in a context requiring a given type (called the *target type*). In a small-step semantics, this poses a problem: reduction can move λ-abstractions into arbitrary contexts, leading to intermediate terms which would be ill-typed. To maintain subject reduction, in [2] λ-abstractions are decorated with their initial target type. In a big-step semantics, as in [6], there is no need of such intermediate terms and annotations.

6 Conclusion

We have described a novel way to conciliate OO and functional programming, where objects (instances of classes) are replaced by tuples (encoded by lambdas). Lambdas can be equipped with an additional behaviour, thanks to the fact that they may implement interfaces with default methods, hence inheritance and dynamic binding are still supported. The encoding has been formally defined by a translation from a calculus including classes to one with only lambdas and interfaces, shown to preserve typing and semantics. This novel programming style has been illustrated by several examples.

Concerning further work, a first step is about the use of the λ-based paradigm in Java 8, illustrated in Section 4. We assumed the encoding of tuples by lambdas to be provided once and for all by libraries. However, in the examples in Section 4, the programmer still has to manually write some tedious and rather cryptic code, notably lambdas such as `(name,age)->s->s.apply(name,age)`, or dummy lambdas. To make the paradigm more user-friendly, suitable syntactic sugar should be provided for these constructions, likely in form of macros.

More in general, we could leave the Java world and investigate the design of a language especially suited to express this paradigm, and its integration with other typical language features. For instance, which would be the best, simplest way to encode mutation in FJ_λ^- where fields are implicit? An inspiration could come from Smalltalk, which allows to update local variables, even when they are captured by a closure.

In this design investigation, in particular an interesting direction is to take an approach which is complementary to that of this paper, that is, to start from a functional kernel and to enrich it by a table of types.

References

1 Jonathan Aldrich. The power of interoperability: why objects are inevitable. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH'13*, pages 101–116. ACM Press, 2013. `doi:10.1145/2509578.2514738`.

2 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight story. *Logical Methods in Computer Science*, 14(3), 2018. `doi:10.23638/LMCS-14(3:17)2018`.

3 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. TraitRecordJ: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013. doi:10.1016/j.scico.2011.06.007.

4 Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999. doi:10.1006/inco.1999.2829.

5 Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. *Electronic Notes in Theoretical Computer Science*, 155:309–329, 2006. Mathematical Foundations of Programming Semantics - MFPS 2005. doi:10.1016/j.entcs.2005.11.061.

6 Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming - ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 169–196. Springer, 2020. doi:10.1007/978-3-030-44914-8_7.

7 Michael Homer, Timothy Jones, and James Noble. First-class dynamic types. In *Dynamic Languages Symposium 2019*, pages 1–19. ACM Press, 2019. doi:10.1145/3359619.3359740.

8 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

9 Timothy Jones, Michael Homer, James Noble, and Kim Bruce. Object inheritance without classes. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *ECOOP'16 - Object-Oriented Programming*, volume 56, pages 13:1–13:26. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.13.

10 Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in Java. *Proceedings of ACM on Programming Languages*, 1(OOPSLA 2017):85:1–85:31, 2017. doi:10.1145/3133909.

11 Martin Plümicke. Well-typings for Java$_\lambda$. In Christian W. Probst and Christian Wimmer, editors, *Principles and Practice of Programming in Java - PPPJ 2011*, pages 91–100. ACM Press, 2011. doi:10.1145/2093157.2093171.

12 Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. Classless Java. In Bernd Fischer and Ina Schaefer, editors, *Generative Programming: Concepts and Experiences - GPCE 2016*, pages 14–24. ACM Press, 2016. doi:10.1145/2993236.2993238.