Formalizing an Object-Oriented Programming Language with Delimited Control

Akane Taniguchi
akane.taniguchi@prg.is.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Youyou Cong
cong@c.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Hidehiko Masuhara
masuhara@acm.org
Tokyo Institute of Technology
Tokyo, Japan

ABSTRACT

Delimited continuations are a tool for expressing complex control flow, and they are supported in languages of different paradigms. To ensure safety of languages with delimited continuations, it is important to develop a mathematical formalization of those languages. In this paper, we formalize an object-oriented language with delimited control operators `shift` and `reset`. Our approach is to represent continuations as λ-expressions. This allows us to treat continuations as first-class values, but it also increases the complexity of programs due to Java's requirement for type annotations.

KEYWORDS

Delimited continuations, object-oriented programming languages, Featherweight Java

ACM Reference Format:
Akane Taniguchi, Youyou Cong, and Hidehiko Masuhara. 2024. Formalizing an Object-Oriented Programming Language with Delimited Control. In *Proceedings of the 36th Symposium on Implementation and Applica- tion of Functional Languages (IFL '24).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Delimited continuations are a tool for expressing complex control flow, and they are supported in languages of various paradigms. To give a few instances of functional languages with continuations, OchaCaml [13] supports the delimited control operators `shift/reset`, and Eff [6], Koka [10], and Links [12] have algebraic effects and handlers. As an example of non-functional languages with continuations, JEff [9] is a Java-like object-oriented language that has effect handlers.

To ensure the safety of languages with delimited continuations, it is important to develop a mathematical formalization of those languages. For the above mentioned functional languages, there exist formalizations based on the lambda calculus [1, 2, 7, 11]. For JEff, there is a formalization called FJEff [9], but it is rather different from formalizations of functional languages, especially with regard to the treatment of continuations.

In this work, we explore formalizations of object-oriented languages with continuations that are close to existing formalizations of functional languages. To this end, we develop $\mathrm{FJ}_{\mathrm{s/r}}$, an extension of Featherweight Java (FJ) [8] with `shift/reset`. A key feature of $\mathrm{FJ}_{\mathrm{s/r}}$ is that it has lambda-expressions, borrowed from the work of Bettini et al. [3]. lambda-expressions allow us to treat continuations as first-class values, as in the aforementioned functional languages. However, they pose new challenges due to Java's requirement for type annotations of lambda expressions. This increases the complexity of programs as well as reduction rules and proofs.

In the rest of the paper, we present the specification of $\mathrm{FJ}_{\mathrm{s/r}}$ (Section 2), discuss related work (Section 3), and conclude the paper with future directions (Section 4).

2 FORMALIZATION OF $\mathrm{FJ}_{\mathrm{s/r}}$

In this section, we formalize $\mathrm{FJ}_{\mathrm{s/r}}$, an extension of Featherweight Java with generic types, λ-expressions, function interfaces, and the `shift/reset` operators.

We begin with an example showing what a program in $\mathrm{FJ}_{\mathrm{s/r}}$ looks like. To make the example easier to understand, we use booleans, numbers, addition, which we do not include in our formalization.

```
⟨ if Sk<Bool,Num>.
    k.apply(true) + k.apply(false)
                    then 1 else -1 ⟩
```

During evaluation of the above program, the `shift` operator `S` captures a continuation up to the `reset` operator ⟨⟩. It then evaluates the body with the captured continuation

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

(Types)	$\mathtt{S, T, U, V, W}$	::=	$\mathtt{X} \mid \mathtt{N}$
(Anotated Types)	$\mathtt{AT}$	::=	$\mathtt{T@T}$
(Non variable types)	$\mathtt{N, P, Q}$	::=	$\mathtt{C<\overline{T}>} \mid \mathtt{FT}$
(Functional types)	$\mathtt{FT}$	::=	$\mathtt{I<T,T,T>}$
(Class declarations)	$\mathtt{CD}$	::=	$\mathtt{class\ C<\overline{X} \lhd \overline{N}> \lhd C<\overline{T}>\{\overline{T}\ \overline{f};\ K\ \overline{M}\}}$
(Interface declarations)	$\mathtt{ID}$	::=	$\mathtt{interface\ I<X \lhd N, Y \lhd P, Z \lhd Q>\{Y@Z\ m(X\ x);\}}$
(Constructor declarations)	$\mathtt{K}$	::=	$\mathtt{C(\overline{T}\ \overline{f})\{super(\overline{f});\ this.\overline{f}=\overline{f};\}}$
(Header declarations)	$\mathtt{H}$	::=	$\mathtt{<\overline{X} \lhd \overline{N}>AT\ m(\overline{T}\ \overline{x})}$
(Method declarations)	$\mathtt{M}$	::=	$\mathtt{H\{return\ e;\}}$
(Terms)	$\mathtt{e, d}$	::=	$\mathtt{v} \mid \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m<\overline{T}>(\overline{e})} \mid \mathtt{new\ C<\overline{T}>(\overline{e})} \mid \mathtt{Sk<T,T>.e} \mid \langle \mathtt{e} \rangle$
(Values)	$\mathtt{u, v}$	::=	$\mathtt{w} \mid \mathtt{p} \rightarrow \mathtt{e}$
(Proper values)	$\mathtt{w}$	::=	$\mathtt{new\ C<\overline{T}>(\overline{v})} \mid (\mathtt{p} \rightarrow \mathtt{e})^{\mathtt{FT}}$
(Parameters)	$\mathtt{p}$	::=	$\mathtt{x} \mid \mathtt{T\ x}$

Figure 1: Syntax

`x` ⟶ ⟨ `if x then 1 else 0` ⟩ bound to the variable `k`. The continuation is called using the method `apply`. In this case, the two calls of the continuation evaluates to `1` and `-1`, respectively, hence the whole program evaluates to `0`.

Notice that the `shift` expression has two type annotations `Bool` and `Num`. These are the input and output types of the captured continuation, respectively.

2.1 Syntax

We define the syntax of FJ$_{s/r}$ in Figure 1. Most constructs are adopted from FJ and its extensions. Specifically, generic types are borrowed from FGJ [8], whereas λ-expressions and interfaces are adopted from FJ&λ [3]. We highlight our additions in gray.

There is one syntactic category that is unique to our language, namely annotated types `AT`. An annotated type is literally a type annotated with a type, where the second type is called an *answer type*. An answer type represents the return type of a delimited context, and it is needed for safe execution of `shift` expressions.

We use several abbreviations borrowed from FJ and its extensions for conciseness. The notation on the right-hand side of colon is abbreviated to the one on the left-hand side.

- $\lhd$: `extends`
- $\overline{\mathtt{f}}$: $\mathtt{f}_1, \ldots, \mathtt{f}_n$ $(\forall n \in \mathbb{N} \cup \{0\})$
- $\bullet$: empty sequence
- $\#(\overline{\mathtt{x}})$: the length of a sequence $\overline{\mathtt{x}}$
- `C` : `C<>`
- $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$: $\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n$
- $\overline{\mathtt{T}}\ \overline{\mathtt{f}};$: $\mathtt{T}_1\ \mathtt{f}_1; \ldots \mathtt{T}_n\ \mathtt{f}_n;$
- $\mathtt{this.\overline{f}=\overline{f};}$: $\mathtt{this.f_1=f_1;} \ldots \mathtt{this.f_n=f_n;}$
- $\mathtt{<\overline{X} \lhd \overline{N}>}$: $\mathtt{<X_1 \lhd N_1, \ldots, X_n \lhd N_n>}$

We also assume that any sequences of field declarations, method declarations, and parameter names does not contain duplicate names.

A class declaration $\mathtt{class\ C<\overline{X} \lhd \overline{N}> \lhd D<\overline{V}>\{\overline{T}\ \overline{f};\ K\ \overline{M}\}}$ defines a class named `C`, which has type variables $\overline{\mathtt{X}}$ bounded by $\overline{\mathtt{N}}$ (i.e. `X` is restricted to the subtype of `N`), a superclass $\mathtt{D<\overline{V}>}$, fields `f` with types `T`, a constructor `K`, and methods $\overline{\mathtt{M}}$.

An interface declaration $\mathtt{interface\ I<X \lhd N, Y \lhd P, Z \lhd Q>\{Y@Z\ m(X\ x);\}}$ defines an interface named `I`, which has type variables `X`, `Y` and `Z` bounded by `N`, `P` and `Q` respectively, and a method header whose argument type is `X`, result type is `Y` and answer type of the body of the method is `Z`. A function does not perform effects as it is a value, but its body may perform effects after application. Thus, it requires an answer type annotation. Although we could use two type variables, one for the input and one for the annotated type (including both the output type and the answer type), we opt for three type variables to ensure safe instantiation. We use interfaces only for λ-expressions; therefore, in FJ$_{s/r}$, we restrict interfaces to functional interfaces with a single method that takes one argument. Consequently, we do not have interface implementations for classes or interfaces.

We assume a pre-defined interface `Cont` for continuations. This interface has a method called `apply`, whose argument type, return type, and answer type are polymorphic.

```
interface Cont<S extends Object,
               T extends Object,
               U extends Object>{
    T@U apply(S x);
}
```

The reason we have a separate interface for continuations is that it enables finer-grained reasoning when the language is extended with purity. Since the body of a continuation

$$\mathit{fields}(\mathtt{Object}) = \bullet \text{ [F-Object]}$$

$$\frac{\mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft N\{\overline{S}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathit{fields}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C<\overline{T}>}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}} \text{ [F-Class]}$$

$$\frac{\mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft N\{\overline{S}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{<\overline{Y} \triangleleft \overline{P}>U@W\ m(\overline{U}\ \overline{x})\{return\ e;\}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<\overline{T}>}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{<\overline{Y} \triangleleft \overline{P}>\overline{U}} \longrightarrow \mathtt{U@W}} \text{ [MT-Class]}$$

$$\frac{\mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft N\{\overline{S}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<\overline{T}>}) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})} \text{ [MT-Super]}$$

$$\frac{\mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft N\{\overline{S}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{<\overline{Y} \triangleleft \overline{P}>U\ m(\overline{U}\ \overline{x})\{return\ e;\}} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m<\overline{V}>}, \mathtt{C<\overline{T}>}) = \overline{\mathtt{x}}.[\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}, \overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\mathtt{e}} \text{ [MB-Class]}$$

$$\frac{\mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft N\{\overline{S}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m<\overline{V}>}, \mathtt{C<\overline{T}>}) = \mathit{mbody}(\mathtt{m<\overline{V}>}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N})} \text{ [MB-Super]}$$

$$\frac{\mathtt{interface\ I<X \triangleleft N, Y \triangleleft P, Z \triangleleft Q>\{Y@Z\ m(X\ x);\}}}{\mathit{mtype}(\mathtt{m}, \mathtt{I<S,T,U>}) = \mathtt{S} \longrightarrow \mathtt{T@U}} \text{ [MT-Interface]}$$

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathtt{<\overline{Z} \triangleleft \overline{Q}>\overline{U}} \longrightarrow \mathtt{U_0} \text{ implies } \overline{\mathtt{P}}, \overline{\mathtt{T}} = [\overline{\mathtt{Z}} \mapsto \overline{\mathtt{Y}}](\overline{\mathtt{Q}}, \overline{\mathtt{U}}) \text{ and } \overline{\mathtt{Y}} \mathtt{<:} \overline{\mathtt{P}} \vdash \mathtt{T_0} \mathtt{<:} [\overline{\mathtt{Z}} \mapsto \overline{\mathtt{Y}}]\mathtt{U_0}}{\mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<\overline{Y} \triangleleft \overline{P}>\overline{T}} \longrightarrow \mathtt{T_0})} \text{ [Override]}$$

Figure 2: Auxiliary Functions

captured by `shift` is surrounded by `reset`, we know that it must be pure. If we only had general function interfaces, we would need to treat every continuation as effectful.

A constructor declaration $\mathtt{C(\overline{S}\ \overline{g}, \overline{T}\ \overline{f})\{super(\overline{G});\ this.\overline{f}=\overline{f};\}}$ defines how the fields of an instance of `C` are initialized. The superclass fields are initialized by the call to `super` and the fields of `C` are initialized by assignment.

A method declaration `H{return e;}` consists of a method header `H` and a method body that returns term `e`. A header declaration $\mathtt{<\overline{X} \triangleleft \overline{N}>AT\ m(\overline{T}\ \overline{x})}$ has type variables $\overline{\mathtt{X}}$ bounded by $\overline{\mathtt{N}}$, a method name `m`, arguments $\overline{\mathtt{x}}$ with its types $\overline{\mathtt{T}}$, and return type annotated with answer type `AT`. The variables $\overline{\mathtt{x}}$ and special variable `this` are bound in the `e`.

A term `e` is either a value `v`, a variable `x`, a field access `e.f`, a method invocation $\mathtt{e.m<\overline{T}>(\overline{e})}$, an object creation $\mathtt{new\ N(\overline{e})}$, a `shift` expression `Sk<S,T>.e`, or a `reset` expression $\langle \mathtt{e} \rangle$. A `shift` expression carries an annotation `<S,T,U>`, where `S`, `T` and `U` are the input, output and answer types of the captured continuation. Note that we do not include casting in $\mathrm{FJ}_{\mathrm{s/r}}$ for simplicity.

A value `v` is either a proper value `w` or a plain[1] λ-expression $\mathtt{p} \longrightarrow \mathtt{e}$. A proper value `w` is either an object creation whose arguments are all values $\mathtt{new\ N(\overline{v})}$ or a decorated λ-expression $(\mathtt{p} \longrightarrow \mathtt{e})^{\mathtt{FT}}$ that has an annotation `FT` representing a functional type. A functional type is the type of a λ-expression, often referred to as "target type" in the Java community. It is an interface with one method, and has the information of the argument, return, and answer types. The decoration is used to track the type of λ-expressions at runtime, and does not appear in a user program. We write $\mathtt{e}_\lambda$ to mean plain λ-expressions. The parameter `p` of a λ-expression may or may not carry its type.

A class table *CT* is a mapping from non-variable types to their declarations. A program in $\mathrm{FJ}_{\mathrm{s/r}}$ is a pair of a class table and a term. We will hereafter assume the existence of a fixed class table.

2.2 Auxiliary Functions

We next define several auxiliary functions (Figure 2), which we use in the reduction and typing rules. Most of them are also adopted from FJ and its extensions. Lookup functions give fields of a class, the type and body of a method in a

[1]What we call a plain lambda expression is called a pure lambda expression in FJ&λ. We use the word "plain" to avoid confusion with the concept of purity (absence of effects).

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ N}(\overline{\mathtt{v}})\mathtt{.f}_i \longrightarrow (\mathtt{v}_i)^{?\mathtt{T}_i}}\ \text{[R-ProjNew]}$$

$$\frac{\mathit{mbody}(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}, \mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \to \mathtt{U@W}}{\mathtt{new\ N}(\overline{\mathtt{v}})\mathtt{.m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{u}}) \longrightarrow [\overline{\mathtt{x}} \mapsto (\overline{\mathtt{u}})^{?[\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\overline{\mathtt{U}}}, \mathtt{this} \mapsto \mathtt{new\ N}(\overline{\mathtt{v}})](\mathtt{e}_0)^{?[\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\mathtt{U}}}\ \text{[R-InvkNew]}$$

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{FT}) = \mathtt{S} \to \mathtt{T@U}}{(\mathtt{y} \to \mathtt{e}_0)^{\mathtt{FT}}\mathtt{.m}(\mathtt{v}) \longrightarrow [\mathtt{y} \mapsto (\mathtt{v})^{?\mathtt{S}}]\mathtt{e}_0{}^{?\mathtt{T}}}\ \text{[R-Invk}\lambda\text{U]}$$

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{FT}) = \mathtt{S} \to \mathtt{T@U}}{(\mathtt{S\ y} \to \mathtt{e}_0)^{\mathtt{FT}}\mathtt{.m}(\mathtt{v}) \longrightarrow [\mathtt{y} \mapsto (\mathtt{v})^{?\mathtt{S}}]\mathtt{e}_0{}^{?\mathtt{T}}}\ \text{[R-Invk}\lambda\text{T]}$$

$$\langle F[\mathtt{Sk}\texttt{<S,T>}\mathtt{.e}]\rangle \longrightarrow \langle[\mathtt{k} \mapsto (\mathtt{x} \to \langle F[\mathtt{x}]\rangle)^{\mathtt{Cont}\texttt{<S,T,U>}}]\mathtt{e}^{?\mathtt{T}}\rangle\ [\text{ R-Shift }]$$

$$\langle \mathtt{w} \rangle \longrightarrow \mathtt{w}\ [\text{ R-Reset }]$$

$$\frac{\mathtt{R} \longrightarrow \mathtt{e}}{E[\mathtt{R}] \longrightarrow E[\mathtt{e}]}\ \text{[E-Step]}$$

$$\begin{array}{rll}
\text{(Pure Eval Ctx)}\quad F[\,] & ::= & [\,] \mid F[[\,]\mathtt{.f}] \mid F[[\,]\mathtt{.m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})] \mid F[\mathtt{w.m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{v}},[\,],\overline{\mathtt{e}})] \mid F[\mathtt{new\ N}(\overline{\mathtt{v}},[\,],\overline{\mathtt{e}})] \\
\text{(Eval Ctx)}\quad E[\,] & ::= & [\,] \mid E[[\,]\mathtt{.f}] \mid E[[\,]\mathtt{.m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})] \mid E[\mathtt{w.m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{v}},[\,],\overline{\mathtt{e}})] \mid E[\mathtt{new\ N}(\overline{\mathtt{v}},[\,],\overline{\mathtt{e}})] \mid E[\langle[\,]\rangle] \\
\text{(Redex)}\quad R & ::= & \mathtt{new\ N}(\overline{\mathtt{v}})\mathtt{.f} \mid \mathtt{w.m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{v}}) \mid \langle F[\mathtt{Sk}\texttt{<S,T>}\mathtt{.U}e]\rangle \mid \langle \mathtt{w} \rangle
\end{array}$$

Figure 3: Reduction Rules and Evaluation context

class, and the type of method in an interface. $\mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$ gives fields of a non-variable type class $\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ and its superclass N. Type variables are instantiated accordingly with the notation $[\mathtt{X} \mapsto \mathtt{T}]$. $\mathit{mtype}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$ gives argument types and a return type of a method m in class $\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ or its superclass N. $\mathit{mbody}(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$ gives the arguments and return term of a method m in class $\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ or its superclass N. $\mathit{mtype}(\mathtt{m}, \mathtt{I}\texttt{<S,T,U>})$ gives an argument type and a return type annotated with an answer type of a method m in interface `I<S,T,U>`. The override validation function ensures that a method in the subclass overrides one in the superclass properly.

2.3 Evaluation

2.3.1 Reduction Rules. In Figure 3, we define reduction rules, which give a call-by-value semantics to $\mathrm{FJ}_{\mathrm{s/r}}$. The rule [R-ProjNew] reduces field access and the rule [R-InvkNew] reduces method invocation. We will explain the notation $(\mathtt{t})^{?\mathtt{T}}$ in Section 2.3.2. The rule [R-InvkλU] and [R-InvkλT] reduce an application of a plain and a decorated λ-expression. Note that the λ-expression must have a target type annotation to determine the type of λ-expression at the end of evaluation. The details of target type annotation are described in Section 2.3.2.

The rule [R-Reset] removes the `reset` expression surrounding a proper value. The rule [R-Shift] reduces a `shift` expression surrounded by a pure evaluation context F and a `reset` expression. The λ-expression $(\mathtt{x} \to \langle F[\mathtt{x}]\rangle)^{\mathtt{Cont}\texttt{<S,T,U>}}$ here represents the captured continuation. The context F is guaranteed to have no reset surrounding a hole, which means the reset on the left-hand side of [R-Shift] is the innermost one. That is, $\langle F[\mathtt{Sk}\texttt{<S,T>}\mathtt{.e}]\rangle$ means $F[\,]$ is the continuation delimited by the closest `reset` corresponding to the `shift` `Sk<S,T>.e`. The captured continuation is expressed in λ-expression and substitutes continuation variable k in `shift` body $\mathtt{e}_0$. The target type of the continuation is easily obtained from annotation of `shift`.

[R-Step] defines one-step evaluation as reduction inside an evaluation context. Here, evaluation contexts are designed in a way that enforces call-by-value, left-to-right evaluation.

2.3.2 Target Type Decorations. As we briefly mentioned earlier, the target type of λ-expressions is represented as a functional type. It is determined by the context surrounding a λ-expression. Therefore, a λ-expression may appear only at places where the compiler can infer its types from its location. In $\mathrm{FJ}_{\mathrm{s/r}}$, a λ-expressions is allowed in

- arguments of object creations
- arguments of method invocations
- a body of λ-expressions

$$\Delta \vdash \mathtt{T} <: \mathtt{T}\ \text{[S-Refl]} \qquad \frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \quad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}}\ \text{[S-Trans]} \qquad \Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})\ \text{[S-Var]}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{N}}\ \text{[S-Class]} \qquad \frac{\Delta \vdash \mathtt{U} <: \mathtt{S} \quad \Delta \vdash \mathtt{T} <: \mathtt{V}}{\Delta \vdash \mathtt{I<S,T,W>} <: \mathtt{I<U,V,W>}}\ \text{[S-Func]}$$

Figure 4: Subtyping

$$\Delta \vdash \mathtt{Object}\ \text{ok [WF-Object]} \qquad \frac{\mathtt{X} \in dom(\Delta)}{\Delta \vdash \mathtt{X}\ \text{ok}}\ \text{[WF-Var]}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\} \quad \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{N}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok}}\ \text{[WF-CVar]}$$

$$\frac{\begin{array}{c}\mathtt{interface\ I<X \triangleleft N, Y \triangleleft P, Z \triangleleft Q>\{\ldots\}} \quad \Delta \vdash \mathtt{S,T,U}\ \text{ok} \\ \Delta \vdash \mathtt{S} <: [\mathtt{X} \mapsto \mathtt{S}, \mathtt{Y} \mapsto \mathtt{T}, \mathtt{Z} \mapsto \mathtt{U}]\mathtt{N}, \mathtt{T} <: [\mathtt{X} \mapsto \mathtt{S}, \mathtt{Y} \mapsto \mathtt{T}, \mathtt{Z} \mapsto \mathtt{U}]\mathtt{P}, \mathtt{U} <: [\mathtt{X} \mapsto \mathtt{S}, \mathtt{Y} \mapsto \mathtt{T}, \mathtt{Z} \mapsto \mathtt{U}]\mathtt{Q}\end{array}}{\Delta \vdash \mathtt{I<S,T,U>}\ \text{ok}}\ \text{[WF-IVar]}$$

Figure 5: Well-formed Types

- a body of`shift` expressions
- method declarations

A decorated λ-expression can appear as a method receiver for function application, but this happens only at runtime (in the rule [R-InvkλU] and [R-InvkλT]). Any other appearance of λ-expressions causes a compile error.

During reduction, we need to decorate λ-expressions with their target and keep track of it. This is important because the reduction process loses information about the structure surrounding the λ-expression, making it impossible to determine its type after the evaluation. To decorate λ-expressions, we use a mapping $(\mathtt{e})^{?\mathtt{FT}}$ in the following manner:

$$(\mathtt{e})^{?\mathtt{FT}} = \begin{cases} (\mathtt{e})^{\mathtt{FT}} & \text{(if } \mathtt{t} \text{ is a plain } \lambda\text{-expressions)} \\ \mathtt{e} & \text{(otherwise)} \end{cases}$$

The subterm on the left-hand side of the reduction which can be a λ-expression, gets annotated with $(\mathtt{e})^{?\mathtt{FT}}$ on the right-hand side.

2.4 Typing

In this section, we define typing of $\mathrm{FJ}_{s/r}$.

There are two kinds of environments in $\mathrm{FJ}_{s/r}$. An environment Γ maps variables to types, and a type environment Δ maps type variables to non-variable types within its bound.

$$\begin{array}{rcl} \text{(Environment)} \quad \Gamma & ::= & \emptyset \mid \Gamma, \mathtt{x} : \mathtt{T} \\ \text{(Type environment)} \quad \Delta & ::= & \emptyset \mid \Delta, \mathtt{X} <: \mathtt{N} \end{array}$$

There are also three judgments in $\mathrm{FJ}_{s/r}$: $\Delta \vdash \mathtt{S} <: \mathtt{T}$ for subtyping: $\Delta \vdash \mathtt{S}$ ok for well-formedness of types, and $\Gamma; \Delta \vdash \mathtt{x} : \mathtt{T}$ for typing terms. We use the following abbreviations for conciseness: $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$ as a shorthand for $\Delta \vdash \mathtt{S}_1 <: \mathtt{T}_1, \ldots, \mathtt{S}_n <: \mathtt{T}_n$ and $\Delta \vdash \overline{\mathtt{S}}$ ok as a shorthand for $\Delta \vdash \mathtt{S}_1$ ok, $\ldots, \Delta \vdash \mathtt{S}_n$ ok and $\Gamma; \Delta \vdash \overline{\mathtt{x}} : \overline{\mathtt{T}}$ as a shorthand for $\Gamma; \Delta \vdash \mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n$.

We define a mapping $bound_\Delta(\mathtt{T})$ that returns the upper bound of a given type $\mathtt{T}$ in Δ. $bound_\Delta(\mathtt{T})$ always returns a non-variable type because the syntax of declaration ensure that the bound of a type variable must be a non-variable type.

$$bound_\Delta(\mathtt{T}) = \begin{cases} \Delta(\mathtt{X}) & (\mathtt{T} = \mathtt{X}) \\ \mathtt{N} & (\mathtt{T} = \mathtt{N}) \end{cases}$$

Subtyping is defined in Figure 4. $\Delta \vdash \mathtt{S} <: \mathtt{T}$ means $\mathtt{S}$ is a subtype of $\mathtt{T}$ in Δ. Subtyping is the reflexive and transitive closure of subtype relation yielded by `extends` in CT. Subtyping for functional types exhibits contravariance in the argument position, similar to the arrow type subtype relation. Note that Subtyping for non-variable class types does not have this property, i.e. $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$ does not imply $\Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>} <: \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$.

We need to validate proper substitution for type variables in typing. For example, if class $\mathtt{C}$ is declared as $\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\}$, then type $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ should satisfy bound of its type variables $\Delta \vdash \mathtt{T} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{N}}$. Types satisfying this property, called well-formed types, are defined in Figure 5.

The typing rule for terms and classes, interfaces, and method declarations are given in Figure 6.

[T-λU] and [T-λT] type λ-expressions with target type decoration. As mentioned in Section 2.3.2, λ-expression only appears in positions where the compiler can infer its type by the surrounding structure. This is why there exist typing rules for decorated λ-expressions only.

$$\Gamma;\Delta \vdash \mathtt{x} : \Gamma(\mathtt{x})@\mathtt{T}\ [\text{T-Val}] \qquad \frac{\Gamma;\Delta \vdash \mathtt{e}_0 : \mathtt{T@U} \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T})) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}}{\Gamma;\Delta \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}_i@\mathtt{U}}\ [\text{T-Field}]$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \mathtt{e}_0 : \mathtt{T@W} \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T})) = \mathtt{<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>}\bar{\mathtt{U}} \longrightarrow \mathtt{U@W} \\ \Delta \vdash \bar{\mathtt{V}}\ \text{ok} \qquad \Delta \vdash \bar{\mathtt{V}} \mathrel{<:} [\bar{\mathtt{Y}} \mapsto \bar{\mathtt{V}}]\bar{\mathtt{P}} \qquad \Gamma;\Delta \vdash^* \bar{\mathtt{e}} : [\bar{\mathtt{Y}} \mapsto \bar{\mathtt{V}}]\bar{\mathtt{U}}@\mathtt{W}\end{array}}{\Gamma;\Delta \vdash \mathtt{e}_0.\mathtt{m<}\bar{\mathtt{V}}\mathtt{>(}\bar{\mathtt{e}}\mathtt{)} : [\bar{\mathtt{Y}} \mapsto \bar{\mathtt{V}}]\mathtt{U@W}}\ [\text{T-Invk}]$$

$$\frac{\Delta \vdash \mathtt{N}\ \text{ok} \qquad \mathit{fields}(\mathtt{N}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}} \qquad \Gamma;\Delta \vdash^* \bar{\mathtt{e}} : \bar{\mathtt{T}}@\mathtt{U}}{\Gamma;\Delta \vdash \mathtt{new\ N(}\bar{\mathtt{e}}\mathtt{)} : \mathtt{N@U}}\ [\text{T-New}]$$

$$\frac{\Delta \vdash \mathtt{S,T,U,W}\ \text{ok} \qquad \Gamma, \mathtt{y}:\mathtt{S};\Delta \vdash^* \mathtt{e}_0 : \mathtt{T@U}}{\Gamma;\Delta \vdash (\mathtt{y} \longrightarrow \mathtt{e}_0)^{\mathtt{I<S,T,U>}} : \mathtt{I<S,T,U>@W}}\ [\text{U-}\lambda\text{U}] \qquad \frac{\Delta \vdash \mathtt{S,T,U,W}\ \text{ok} \qquad \Gamma, \mathtt{y}:\mathtt{S};\Delta \vdash^* \mathtt{e}_0 : \mathtt{T@U}}{\Gamma;\Delta \vdash (\mathtt{S\ y} \longrightarrow \mathtt{e}_0)^{\mathtt{I<S,T,U>}} : \mathtt{I<S,T,U>@W}}\ [\text{T-}\lambda\text{T}]$$

$$\frac{\Delta \vdash \mathtt{S,T,U}\ \text{ok} \qquad \Gamma, \mathtt{k} : \mathtt{Cont<S,T,U>};\Delta \vdash^* \mathtt{e}_0 : \mathtt{T@T}}{\Gamma;\Delta \vdash \mathtt{Sk<S,T>.e}_0 : \mathtt{S@T}}\ [\text{T-Shift}] \qquad \frac{\Gamma;\Delta \vdash \mathtt{e}_0 : \mathtt{T@T}}{\Gamma;\Delta \vdash \langle \mathtt{e}_0 \rangle : \mathtt{T@U}}\ [\text{T-Reset}]$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{...\}} \qquad \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>}\bar{\mathtt{T}} \longrightarrow \mathtt{T}_0) \\ \Delta = \bar{\mathtt{X}} \mathrel{<:} \bar{\mathtt{N}}, \bar{\mathtt{Y}} \mathrel{<:} \bar{\mathtt{P}} \qquad \Delta \vdash \bar{\mathtt{T}}, \mathtt{T}, \bar{\mathtt{P}} \qquad \bar{\mathtt{x}} : \bar{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\bar{\mathtt{X}}\mathtt{>};\Delta \vdash^* \mathtt{e}_0 : \mathtt{S}@\alpha \qquad \Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}\end{array}}{\mathtt{<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>T}@\alpha\ \mathtt{m(}\bar{\mathtt{T}}\ \bar{\mathtt{x}}\mathtt{)\{return\ e;\}\ OK\ IN\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}}\ [\text{T-Method}]$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C(}\bar{\mathtt{S}}\ \bar{\mathtt{g}}\mathtt{,}\bar{\mathtt{T}}\ \bar{\mathtt{f}}\mathtt{)\{super(}\bar{\mathtt{g}}\mathtt{);\ this.}\bar{\mathtt{f}}\mathtt{=}\bar{\mathtt{f}}\mathtt{;\}} \\ \bar{\mathtt{X}} \mathrel{<:} \bar{\mathtt{N}} \vdash \bar{\mathtt{N}}, \mathtt{N}, \bar{\mathtt{T}}\ \text{ok} \qquad \mathit{fields}(\mathtt{N}) = \bar{\mathtt{S}}\ \bar{\mathtt{g}} \qquad \bar{\mathtt{M}}\ \mathtt{OK\ IN\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\end{array}}{\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\{}\bar{\mathtt{T}}\ \bar{\mathtt{f}}\mathtt{;\ K\ }\bar{\mathtt{M}}\mathtt{\}\ OK}}\ [\text{T-Class}]$$

$$\frac{\mathtt{X} \mathrel{<:} \mathtt{N}, \mathtt{Y} \mathrel{<:} \mathtt{P} \vdash \mathtt{N}, \mathtt{P}\ \text{ok}}{\mathtt{interface\ I<X} \lhd \mathtt{N,Y} \lhd \mathtt{P,Y} \lhd \mathtt{m>\{Y@Y\ X(X\ x);\}\ OK}}\ [\text{T-Interface}]$$

Figure 6: Typing Rules

In [T-Shift], continuation variable k is typed in functional type because continuations are function. The body of the shift has the return type of continuation and the shift itself has the argument type of continuation. It is because where shift is placed is where the argument variable of the continuation is placed. [T-Reset] shows the removal of reset bracket does nothing to its typing.

Judgment $\vdash^*$ plays a different role depending on whether t is a plain λ-expression, similar to the notation $(\mathtt{e})^{?\mathtt{T}}$ in reduction rules. If t is a plain λ-expression, $\vdash^*$ annotate t with a type. Otherwise, $\vdash^*$ checks if the term has a subtype of an expected type.

$$\frac{\Gamma;\Delta \vdash (\mathtt{e}_\lambda)^{\mathtt{FT}} : \mathtt{FT@U}}{\Gamma;\Delta \vdash^* \mathtt{e}_\lambda : \mathtt{FT@U}} \qquad \frac{\Gamma;\Delta \vdash \mathtt{e} : \mathtt{S@U} \qquad \Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma;\Delta \vdash^* \mathtt{e} : \mathtt{T@U}}$$

Combining the two yields the following judgment using $(\mathtt{e})^{?\mathtt{T}}$:

$$\frac{\Gamma;\Delta \vdash (\mathtt{e})^{?\mathtt{T}} : \mathtt{S@U} \qquad \Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma;\Delta \vdash^* \mathtt{e} : \mathtt{T@U}}\ [\vdash\vdash^*]$$

Without this, we must consider both cases for every single term that could be a plain λ-expression. Note that $\vdash^*$ only achieves a simpler notation of the typing rule and is not another typing system other than $\vdash$.

[T-Method], [T-Class], and [T-Interface] check the well-formedness of method, class, and interface declarations. In [T-Method], typing of method body $\mathtt{e}_0$ uses judgment $\vdash^*$ because λ-expression can appear in the method body. Function *override* guarantees proper method overriding on superclass methods. [T-Class] checks valid super call to superclass fields.

2.5 Soundness

We prove the soundness of $\mathrm{FJ}_{\mathrm{s/r}}$ by showing the progress and subject reduction properties. So far, we have proven these properties for a variation of the formalization that does not have answer types. Here, we outline what needs to be proven for our current formalization.

The progress property is stated as: if a term e is closed and well-typed, it is a value, or it takes a step, or it is a stuck term composed of a shift expression and a pure evaluation context. Note that the stuck case is often present in the progress property for effectful languages.

Theorem 2.1 (Progress).
If $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T@U}$, then e *is a proper value, or there is* e' *with*

$\texttt{e} \longrightarrow \texttt{e'}$, *or* $\texttt{e} = F[\texttt{Sk<S,V>.e}_0]$ *for some pure evaluation context* F.

The subject reduction property is stated as: if a term `e` is well-typed and takes a step, then the term `e'` after one-step evaluation is also well-typed.

Theorem 2.2 (Subject Reduction).
If $\Gamma; \Delta \vdash \texttt{e} : \texttt{T@U}$ *and* $\texttt{e} \rightarrow \texttt{e'}$, *then* $\Gamma; \Delta \vdash \texttt{e'} : \texttt{S@U}$ *for some* $\texttt{S@U} <: \texttt{T@U}$.

Both properties are proven by induction on the typing derivation of `e`.

3 RELATED WORK

There are several attempts on introducing computational effects into object-oriented languages. JavaEffekt [4] is a Java library for effect handlers. In JavaEffekt, effect signatures are defined as interfaces and handlers as classes that implement those interfaces. Continuations are represented using λ-expressions as in $\text{FJ}_{\text{s/r}}$.

JEff [9] is a Java-like language with native support for effect handlers. In JEff, effect handlers are expressed in a similar way to JavaEffekt, but continuations are represented as a resumption object containing the definition of the `resume` method. The language has a formalization, called FJEff, which extends FJ with interfaces and generics but not λ-expressions.

In the functional programming community, different control operators have been formalized with different typing features. Danvy and Filinski [5] present a simple type system for `shift/reset`. The type system is derived from the CPS semantics of `shift/reset`, and allows modification of answer types. Asai and Kameyama [1] formalize a polymorphic type system for `shift/reset`. Similar to Danvy and Filinski, they allow answer type modification, and in addition to that, they distinguish between pure and impure terms by introducing the notion of answer type polymorphism. Materzok and Biernacki [14] propose a type system for `shift0/reset0`, a minor variation of `shift/reset`. In their type system, an effect is a list of answer types representing the return types of nesting delimiters, and they can be adjusted via a subtyping relation defined in terms of the length of effects.

4 CONCLUSION AND FUTURE WORK

In this paper, we formalized $\text{FJ}_{\text{s/r}}$, an extension of Featherweight Java with generic types, λ-expressions, function interfaces, and the `shift/reset` operators. We also outlined our ongoing proof of the soundness theorem.

In future work, we intend to introduce the concept of purity to $\text{FJ}_{\text{s/r}}$. Purity expresses the absence of effects. The introduction of purity would change how answer types appear in types of terms. Currently, every term has an answer type annotation, even if it does not perform any effects and hence there is no need to consider the type of the context. After introduction of purity, terms will be divided into two categories: pure terms, which do not have an answer type annotation, and impure terms, which have an annotation.

The next thing we would like to do is to add casting to $\text{FJ}_{\text{s/r}}$. Casting is used to convert types based on subtyping relations. Upcast converts a type to its superclass, which is always safe. In contrast, downcast converts a type to its subclass, which can be unsafe. The ability of casting types is crucial in a language with purity, because we often wish to treat pure terms as impure ones. We plan to support upcast of answer types so that we can type more programs while enjoying soundness.

REFERENCES

[1] Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic delimited continuations. In *Programming Languages and Systems: 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007. Proceedings 5*. Springer, 239–254.

[2] Andrej Bauer and Matija Pretnar. 2014. An effect system for algebraic effects and handlers. *Logical methods in computer science* 10 (2014).

[3] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. 2018. Java & lambda: a featherweight story. *Logical Methods in Computer Science* 14 (2018).

[4] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect handlers for the masses. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[5] Olivier Danvy and Andrzej Filinski. 1989. *A functional abstraction of typed contexts*. Univ.

[6] Eff. [n. d.]. Eff Programming Language. https://www.eff-lang.org/.

[7] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*. 15–27.

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

[9] Pablo Inostroza and Tijs van der Storm. 2018. JEff: Objects for effect. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 111–124.

[10] Koka. [n. d.]. The Koka Programming Language. https://koka-lang.github.io/koka/doc/index.html.

[11] Daan Leijen. 2014. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061* (2014).

[12] Links. [n. d.]. The Links Programming Language. https://links-lang.org/.

[13] Moe Masuko and Kenichi Asai. 2009. Direct implementation of shift and reset in the MinCaml compiler. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*. 49–60.

[14] Marek Materzok and Dariusz Biernacki. 2011. Subtyping delimited continuations. *ACM SIGPLAN Notices* (2011).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009