Named Arguments as Records
(TFP'22 draft)

Yaozhu Sun and Bruno C. d. S. Oliveira

The University of Hong Kong, China
{yzsun,bruno}@cs.hku.hk

Abstract. Named arguments are commonly supported in mainstream programming languages. However, there has been little work on formalizing the design of named arguments.
This paper shows a minimal calculus that encodes named arguments as a form of records. Our design is based on a variant of record types, which allows *optional fields* and introduces two alternative notions of *open-bindings* and *failable projections*. With this design, we are able to model an expressive form of named arguments, which supports optional arguments as well. In our design, named arguments are commutative, and they are distinct from positional arguments. We present an extension to $\lambda_{<:}$ and discuss its semantics. Our main goal is to obtain a calculus that is as simple as possible but still captures most of the desirable features for named arguments.

Keywords: Named arguments · Optional arguments · Record types

1 Introduction

The λ-calculus, introduced by Alonzo Church, shows us how to model computation solely with function abstraction and application. In the λ-calculus, a function only has one parameter and can only be applied to one argument. Many programming languages in the ML family inherit this feature. If we want a function with multiple arguments in those languages, we need to turn it into a sequence of functions, each with a single argument, which is called *currying*. Currying brings brevity to functional programming and naturally allows partial application, but it also limits the flexibility of function application. For example, we cannot pass arguments in a different order nor omit some of them by providing default values. Both demands are not rare at all in practical programming and can be met in a language that supports *named arguments*.

Named arguments are commonly supported in mainstream programming languages, such as C#, Python, Ruby, and Scala, just to name a few. The earliest instance, to the best of our knowledge, is Smalltalk, where every argument must be associated with a keyword. The syntax of modern languages is usually less rigid, so programmers can choose whether to attach keywords to arguments or not. More specifically, there are two ways to handle named arguments. The more

```python
def exp(x, base=math.e):
  return base ** x

exp(10) #= exp(x=10) = 22026
exp(base=2, x=10)   #=  1024
```

(a) The Python way

```ruby
def exp(x:, base: Math::E)
  base ** x
end
exp(10)    # ArgumentError!
exp(base: 2, x: 10) #= 1024
```

(b) The Ruby way

Fig. 1: Named arguments in two different ways

common way, shown in Fig. 1a, is to make variable names in the function definition as optional keywords. Thus every argument can be passed with or without keywords. As shown in the Python code, `exp(10)` is equivalent to `exp(x=10)`. To reconcile the two forms, a restriction is imposed that all positional arguments have to appear to the left of named ones. The other way, shown in Fig. 1b, is to strictly distinguish named arguments from positional ones. In Ruby, a named parameter should end with a colon even if it does not have a default value. Therefore, they are distinct from positional parameters, and their keywords cannot be omitted in a function call. In fact, these named arguments are desugared to a hash map.

Although named arguments are ubiquitous in practical programming, they do not attract enough attention in the research of programming languages. Among the few related papers, the work by Garrigue et al. [1,3,6] formalizes a label-selective λ-calculus and eventually applies it to OCaml [5]. We will discuss it in detail in Section 2.1. Another work by Rytz and Odersky [10] discusses the design of named and optional arguments in Scala but does not formalize it. In Haskell, the paradigm of *named arguments as records* is folklore, which will be elaborated in Section 2.2. From previous work in the literature, we identify four design choices related to named arguments:

1. *Commutativity*: whether the order of arguments can be different from that in the original definition.
2. *Optionality*: whether some arguments can be optional if their default values are given.
3. *Distinctness*: whether named arguments are distinct from positional arguments.
4. *Currying*: whether a function that takes more than one argument is always converted into a chain of functions that each take a single argument.

The first two properties hold for most mainstream programming languages that support named arguments. Commutativity and optionality are so useful that they should not be compromised. The third design is endorsed by Ruby and Racket, and we find it more intuitive than mixing two forms of arguments, so we advocate distinguishing them. In other words, we have to add argument keywords in the function application as long as they defined to be named. The last one, currying, is important for functional languages. OCaml manages to integrate

currying with commutativity, at the cost of introducing a very complicated core calculus. We agree that currying is very useful when we use normal positional arguments, but we argue that currying can be temporarily dropped when we use named arguments because the most common use case for named arguments is to represent a whole chunk of parameters like settings.

In this paper, we propose a new approach by desugaring named arguments into records in a minimal core calculus. The approach is simpler than the OCaml way and avoids some drawbacks of the design pattern used in Haskell. To some degree, we absorb the design pattern into the language support. Our approach supports commutativity and optionality but does not support currying when using named arguments. This is a trade-off between simplicity and expressiveness. Nevertheless, users have the right to choose either to use named arguments without currying or to use positional arguments with currying.

To make it clear, our goal is not to find a solution that is as powerful as OCaml has, but to propose a core calculus that is *as simple as possible* but still supports named and optional arguments via desugaring. We believe such a simple calculus leads to less feature interaction when integrated with other languages.

Our source language benefits from named arguments in two aspects. On the one hand, argument keywords serve as extra documentation at the language level. We do not need to open the code in an IDE and look up the definition of a function to figure out for what its arguments are used. This design is more friendly to those users who read code literally. Moreover, keywords are part of a function type, so we can know more from the type information without referring to ad-hoc documentation. On the other hand, named arguments lay the foundation for supporting commutativity and optionality. Without argument keywords, it is unclear how to naturally support these useful features.

In summary, the contributions of this paper are:

- We document the folklore paradigm of *named arguments as records* in Haskell.
- We demonstrate how a functional language with named and optional arguments can be desugared to a minimal core calculus.
- We propose two core calculi supporting optional fields in record types, which extend $\lambda_{<:}$ with open-bindings and failable projections respectively.
- In order to give a correct small-step operational semantics to open-bindings, we explore environment-based evaluation with closures instead of using textual substitution.

2 Named Arguments in Existing Functional Languages

In this section, we review the techniques used for named arguments in two existing functional languages, namely OCaml and Haskell. We will explain why their approaches still have some drawbacks.

2.1 OCaml

Originally, just like other languages in the ML family, OCaml did not support named arguments. Later, Garrigue et al. [1,3,6] conducted research on the label-selective λ-calculus and implemented it in OLabl [4]. OLabl extended OCaml with labeled and optional arguments, among others. All features of OLabl were merged into OCaml 3, though with subtle differences [5].

Here is an example of the exponential function defined in a labeled style:

```
let exp ?(base = 2.71828) x = base ** x
(* val exp : ?base:float → float → float = <fun> *)
exp 10.0              (*= 22026. *)
exp 10.0 ~base:2.0    (*=  1024. *)
(exp 10.0) ~base:2.0  (* TypeError! *)
```

In the definition of `exp`, `base` is an optional labeled parameter while `x` is a normal positional parameter. We cannot change `x` into a second labeled parameter here because OCaml imposes a restriction that there must be a positional parameter after all optional parameters. This restriction is at the heart of how OCaml resolves the ambiguity introduced by currying. For example, consider the function application `exp 10.0`. Is it a partially applied function or a fully applied one using the default value of `base`? The presence of the positional argument (`x` in this example) is used to decide whether such a function has been fully applied. So `exp 10.0` is considered to be a full application because `x` is already given. However, this feature may confuse users since `(exp 10.0) ~base:2.0` will raise a type error but `exp 10.0 ~base:2.0` will not. Currying does not seem to hold in such a situation.

In OCaml, optional arguments are internally implemented as `option` types. Here is an equivalent definition for `exp`, together with two examples of the transformation of optional arguments:

```
let exp ?base x =
  let base = match base with None → 2.71828 | Some b → b in
  base ** x
(* val exp : ?base:float → float → float = <fun> *)
exp 10.0            (*> exp 10.0 ~base:None *)
exp 10.0 ~base:2.0  (*> exp 10.0 ~base:(Some 2.0) *)
```

This encoding is quite natural, but it assumes `option` types to be built in. Unfortunately, there are plenty of languages that do not regard `option` as a built-in type, especially in those languages that do not support algebraic data types.

In short, the OCaml way cannot scale smoothly. OCaml has a very powerful label-selective core calculus that reconciles commutativity and currying, but it is quite complicated, hindering its integration with other languages. The assumption of `option` types makes the situation even worse. In contrast to OCaml, we want a minimal core calculus that supports named and optional arguments via desugaring.

```
data Settings = Settings
  { settingsPort :: Port
  , settingsHost :: HostPreference
  , settingsTimeOut :: Int
  , ...
  }
```

(a) The record type containing settings

```
defaultSettings = Settings
  { settingsPort = 3000
  , settingsHost = "*4"
  , settingsTimeout = 30
  , ...
  }
```

(b) Default values

```
runSettings :: Settings → Application → IO ()
runSettings = ...

main :: IO ()
main = runSettings settings app
  where settings = defaultSettings { settingsPort = 4000
                                   , settingsHost = "*6" }
```

(c) Update some settings before running a server application

Fig. 2: Named arguments as records in Haskell

2.2 Haskell

Unlike OCaml, Haskell does not support named arguments natively. However, the paradigm of *named arguments as records* has long existed in the Haskell community. Although we have to uncurry a function to have all parameters labeled in a record, it is clearer and more human-readable, especially when different parameters have the same type. For example, in the web server library *warp* [11], various server settings are bundled in the data type `Settings`, as shown in Fig. 2. It is obvious how named arguments correspond to record fields, but it needs some thought on how to encode default values for optional arguments. The simplest approach, also used by *warp*, is to define a record `defaultSettings`. Users can update whatever fields they want to change while keeping others.

Such an approach works fine here but still has two drawbacks. The first issue is the dependency on `defaultSettings`. It is awkward for users to look for a record containing particular default values, especially when there are quite a few similar records in a library. A better solution is to change the parameter from concrete `Settings` to a function that updates `Settings`:

```
runSettings' :: (Settings → Settings) → Application → IO ()
runSettings' update = runSettings (update defaultSettings)

main :: IO ()
main = runSettings' update app
  where update settings = settings { settingsPort = 4000
                                   , settingsHost = "*6" }
```

With the new interface, users do not need to look for default values anymore, and the use of `runSettings'` is fully decoupled from `defaultSettings`. However, this design still has the second drawback: all arguments are optional. Sometimes we do not want to provide any default value for some argument, `settingsPort` for example, and users are required to fill it in. A workaround employed by `SqlBackend` in the library *persistent* [7] is to have another function that takes required arguments and supplements default values for optional arguments:

```
{-# language DuplicateRecordFields, RecordWildCards #-}

data ReqSettings = ReqSettings { settingsPort :: Port }

mkSettings :: ReqSettings → Settings
mkSettings ReqSettings {..} =
  Settings { settingsHost = "*4", settingsTimeout = 30, .. }
```

Although the new `mkSettings` function solves the second issue, there is a regression concerning the first issue: users have to look for `mk*` functions now. Fortunately, we can harmonize the essence of both design patterns to develop a third approach:

```
{-# language DuplicateRecordFields, RecordWildCards #-}

data OptSettings = OptSettings { settingsHost :: HostPreference
                               , settingsTimeOut :: Int }

runSettings'' :: (OptSettings → Settings) → Application → IO ()
runSettings'' update = runSettings (update defaultSettings)
  where defaultSettings = OptSettings { settingsHost = "*4"
                                      , settingsTimeout = 30 }

main :: IO ()
main = runSettings'' update app
  where update OptSettings {..} =
          Settings { settingsPort = 4000, .. }
                   { settingsHost = "*6" }
```

This last approach is probably the best practice at the moment in Haskell, though it is already complicated for novices and requires two GHC language extensions. Of course, there could be other approaches we did not mention to encoding named and optional arguments in Haskell. Users may get users confused about the various available design patterns. This is partly due to the lack of the language support for named arguments. We believe it is better for a functional language to provide some standard syntax instead.

3 Encoding Named Arguments as Records

In this section, we demonstrate our approach to encoding named and optional arguments. Two possible ways of desugaring are presented, both of which are based on a minimal extension of $\lambda_{<:}$. The first one adds *open-bindings* while the second one adds *failable projections*. A new kind of record type with *optional fields* is also needed for type safety.

3.1 Desugaring with Open-Bindings

Let us revisit the example of the exponential function. This time, `exp` is defined and applied using our source language in ML-like syntax:

```
exp { x: Double; base: Double = 2.71828 } = base ** x

exp { x = 10.0 }              --> 22026.
exp { x = 10.0; base = 2.0 }  -->  1024.
exp { base = 2.0; x = 10.0 }  -->  1024.
```

In the definition of `exp`, we provide the default value for `base`. When applying `exp`, we can choose whether to pass `base` or not as long as the required argument `x` is present. We can freely swap the order of `x` and `base` while keeping the meaning of each argument clear.

Desugared code. How does it work? Actually, we desugar the previous definition of `exp` to a core expression of this form:

```
exp = λargs: { x: Double }. let base = 2.71828 in
  open args in  -- let x = 10.0 in let base = 2.0 in
  base ** x                      ----------------- shadowing!
```

The first thing to note is that the type of `args` does not contain any optional argument. Here, we leverage *width subtyping* between record types to accept additional arguments. That is why the record type only contains required arguments like `x`.

In the desugared definition, the most interesting part is the *open-binding* on the second line. It will dynamically convert each field within the record into a corresponding let-binding. While optional arguments are already bound to their default values, the new values in the argument record, if provided by users, will *shadow* the previous let-bindings. Note that we cannot statically convert open-bindings to let-bindings because we cannot know what optional fields are available in the record `args` until run time. In other words, `open` is a *dynamic* operation that inspects the evaluated value of the argument record.

A stricter open. Although our desugaring works fine so far, one may have a concern about accidental shadowing caused by open-bindings. For example, if a user applies `exp` to `{ x = 10.0; foo = "bar" }`, the field `foo` may cause accidental shadowing if this variable has already been defined. To avoid such an

embarrassing situation, we propose a stricter version of `open` with a permitted label set. The given labels limit the range of fields that can be opened. With this version of open, the desugared code can be rewritten like this:

```
exp = λargs: { x: Double }. let base = 2.71828 in
  open base, x of args in
  base ** x
```

However, this stricter version is still unsatisfying in terms of type safety. We will revisit open-bindings in Section 3.3.

3.2 Desugaring with Failable Projections

In the first way of desugaring, we said that open-bindings cannot be statically converted to let-bindings because the presence of optional fields is unknown until run time. Thus we cannot guarantee that record projections are always safe. But what if we allow failable projections? In OCaml, `option` types are used to represent such a failable result. As we have argued, we want to keep the core calculus as simple as possible, so we choose to provide default values for failable projections instead. It can be regarded as eliminating `option` values with the `Option.value` function immediately. (`Option.value` is equivalent to `fromMaybe` in Haskell.)

Desugared code. With failable projections, the previous definition of `exp` is desugared into a core expression of this form:

```
exp = λargs: { x: Double }.
  let base = args.base ? 2.71828 in  -- failable projection
  let x = args.x in           -- definitely safe projection
  base ** x
```

The syntax of failable projection is $e_1.\ell$? e_2. If a field of the form $\{\ell = v\}$ is present in e_1 then v is returned, otherise use e_2 as a default. Note that the second projection is definitely safe because the type of `args` ensures that `x` is present.

Lazy evaluation. In most situations, failable projections and the stricter version of open-bindings are equally valid. But one thing to note is that failable projections have better compatibility with lazy evaluation. This issue is about the strict evaluation of e_1 in the expression `open` e_1 `in` e_2. For example, consider the following code:

```
const = λargs: Top. let foo = 0 in open args in 48
const { foo = undefined }
```

Here, we assume `undefined` to be a stuck term. The application of `const` is certainly stuck since the argument is strictly evaluated. This is not a bug but a feature in the *call-by-value* λ-calculus. To avoid this, we can employ *call-by-name* evaluation, but our open-bindings do not have a lazy version. The evaluation will still be stuck when evaluating `args` in the expression `open args in ...`.

Therefore, we have to choose an alternative way using failable projections and lazy let-bindings:

```
const = λargs: Top. let foo = args.foo ? 0 in 48
const { foo = undefined }
```

Since `foo` is unused, the expression `args.foo ? 0` is never evaluated. Thus the code terminates in a call-by-name semantics.

3.3 Toward Type Safety

There is still a serious problem in our approach: *neither* open-bindings *nor* failable projections are type-safe! To address this problem, we need to provide the type system with more information about optional fields. The way of desugaring should be changed a bit:

```
exp : { x: Double; base: Double = 2.71828 } = ...
-- will be desugared to:
exp = λargs: { x: Double | base?: Double }. ...
```

In the type of `args` in the desugared code, the optional argument `base` is appended after a vertical bar. The question mark after the label is used to visually distinguish optional fields from required ones. With the extra type information, we can do more checks to ensure type safety.

Open-bindings. Even with the stricter version of `open`, we could not guarantee that every opened argument has the same type as its default value. After we include optional fields in the parameter type, the desugared code is now:

```
exp = λargs: { x: Double | base?: Double }.
  let base = 2.71828 in open args in base ** x
```

Since we statically know the names and types of the optional arguments from the type of `args`, we can check if they are bound with default values of appropriate types before `args` are opened. At call sites, optional arguments are also checked against the parameter type to ensure that they have the correct types. Furthermore, passing undeclared arguments is forbidden to avoid accidental shadowing.

Failable projections. A similar issue can be found in the approach with failable projections: we cannot guarantee that the value we obtain from a projection has the same type as the default value. But with the new way, the desugared code is now:

```
exp = λargs: { x: Double | base?: Double }.
  let base = args.base ? 2.71828 in let x = args.x in base ** x
```

When type checking `args.base ? 2.71828`, we can statically know the potential type of the `base` field. It is easy to make sure the failable projection is type-safe by comparing that type and the type of the default value.

4 Formalization of Core Calculi

After the informal introduction of language constructs in the core calculi, we go deep into the formalization in this section. We formalize failable projections and open-bindings in λ_{proj} and λ_{open}, respectively. As demonstrated before, *either* of the two calculi is enough to encode named arguments. Furthermore, a special record type with extra information about optional fields is added to keep both calculi type-safe.

4.1 Syntax and Semantics of λ_{proj}

The syntax of λ_{proj} is presented in Fig. 3. The $\lambda_{<:}$ components are standard as we follow the formalization in *Software Foundations* [8], except that we use bidirectional typing [2] to make typing rules clear and evaluation contexts [9] to simplify evaluation rules. Therefore, we focus on the novel parts about failable projections and record types with optional fields. It is worth noting that all of our new rules are *modularly* added, which means no existing rules need modification.

Subtyping. The subtyping rules are inherited from $\lambda_{<:}$ intact, including ordinary record subtyping (width, depth, and permutation subtyping). Note that there is no subtyping relation with respect to the record types with optional fields. Consequently, these types will never go through the rule of subsumption.

Typing. Following the convention of bidirectional typing, we use $\Gamma \vdash e \Rightarrow A$ to denote type inference and $\Gamma \vdash e \Leftarrow A$ to denote type checking. There is no rule that infers an expression to be a record type with optional fields, so such a type can only occur in parameters that are annotated by users. When a function is applied to optional arguments, the argument is checked against the parameter type, that is, a record type with optional fields. Such a check is handled by T-OptRcd in Fig. 4. In essence, optional fields add a *lower bound* to a record

Types	$A, B ::= \top \mid A \to B \mid \{\overline{\ell : A}\} \mid \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}\}$
Expressions	$e ::= x \mid \lambda x : A.\ e \mid e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid$
	$\{\overline{\ell = e}\} \mid e.\ell \mid e_1.\ell\ ?\ e_2$
Values	$v ::= \lambda x : A.\ e \mid \{\overline{\ell = v}\}$
Evaluation Contexts	$E ::= \square \mid E\ e \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid$
	$\{\overline{\ell_i = v_i};\ \ell = E;\ \overline{\ell_j = e_j}\} \mid E.\ell \mid E.\ell\ ?\ e$
Typing Environments	$\Gamma ::= \cdot \mid \Gamma, e : A$

Fig. 3: Syntax of λ_{proj}

$$\textsc{T-OptRcd}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad \{\overline{\ell_i : A_i}; \overline{\ell_j : A_j}\} <: A <: \{\overline{\ell_i : A_i}\}}{\Gamma \vdash e \Leftarrow \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}\}}$$

$$\textsc{T-OptProj}\quad \frac{\Gamma \vdash e_1 \Rightarrow \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}; \ell ? : A; \overline{\ell_{j'} ? : A_{j'}}\} \qquad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1.\ell \ ?\ e_2 \Rightarrow A}$$

Fig. 4: Typing rules of optional fields and failable projections

$$\textsc{E-ProjSome}\quad \{\overline{\ell_i = v_i}; \ell = v; \overline{\ell_j = v_j}\}.\ell \ ?\ e_2 \longrightarrow v$$

$$\textsc{E-ProjNone}\quad \frac{\ell \notin \{\overline{\ell_i}\}}{\{\overline{\ell_i = v_i}\}.\ell \ ?\ e_2 \longrightarrow e_2}$$

Fig. 5: Evaluation rules of failable projections

type. The record type $\{\overline{\ell_i : A_i}\}$ without optional fields still acts as the upper bound; meanwhile, we construct another ordinary record type $\{\overline{\ell_i : A_i}; \overline{\ell_j : A_j}\}$ consisting of both optional and required fields as the lower bound. The inferred type of a record should lie between the two bounds. The typing rule of failable projections is also shown in Fig. 4. T-OptProj checks if the optional field with label ℓ has the same type as e_2. If not, type checking fails.

Operational semantics. As shown in Fig. 5, there are two evaluation rules about failable projections: E-ProjSome succeeds in finding the field $\{\ell = v\}$ and steps to v, while E-ProjNone steps to e_2 since ℓ is absent in the record.

4.2 Syntax and Semantics of λ_{open}

The syntax of λ_{open} is presented in Fig. 6. The components about optional fields are omitted since they are the same as those introduced in λ_{proj}. The extension of open-bindings is more difficult than failable projections because we have to employ a different operational semantics. The root cause is that open-bindings are incompatible with *textual substitution*. Since a substitution eagerly replaces all occurrences of a variable by traversing the syntax tree, open-bindings cannot foresee whether a substitution is shadowed by the labels to be opened before the record is evaluated. For example, consider such an expression:

```
let x = 1 in let args = { x = 2 } in open args in x
```

It evaluates to 1 if we evaluate let-bindings with substitution. The substitution of `let x = 1` is not shadowed by `open args` because the record `args` has not been evaluated and the labels it contains are unknown at this moment. Therefore, we abandon substitution and propose a environment-based operational semantics with closures.

$$
\begin{array}{lll}
\text{Expressions} & e ::= & x \mid \lambda x : A.\ e \mid e_1\ e_2 \mid \{\overline{\ell = e}\} \mid e.\ell \mid \\
& & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{open}\ e_1\ \texttt{in}\ e_2 \mid \langle \Delta \mid e \rangle \\
\text{Values} & v ::= & \langle \Delta \mid \lambda x : A.\ e \rangle \mid \{\overline{\ell = v}\} \\
\text{Evaluation Contexts} & E ::= & \square \mid E\ e \mid v\ E \mid \{\overline{\ell_i = v_i};\ \ell = E;\ \overline{\ell_j = e_j}\} \mid E.\ell \mid \\
& & \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{open}\ e_1\ \texttt{in}\ e_2 \\
\text{Valuation Environments} & \Delta ::= & \cdot \mid \Delta, x \mapsto v
\end{array}
$$

Fig. 6: Syntax of λ_{open}

$$\textsc{E-Var}\ \frac{x \mapsto v \in \Delta}{\Delta \vdash x \longrightarrow v} \qquad \textsc{E-Abs}\ \Delta \vdash \lambda x : A.\ e \longrightarrow \langle \Delta \mid \lambda x : A.\ e \rangle$$

$$\textsc{E-App}\ \Delta \vdash \langle \Delta' \mid \lambda x : A.\ e \rangle\ v \longrightarrow \langle \Delta', x \mapsto v \mid e \rangle \qquad \textsc{E-Proj}\ \Delta \vdash \{\overline{\ell_i = v_i};\ \ell = v;\ \overline{\ell_j = v_j}\}.\ell \longrightarrow v$$

$$\textsc{E-Let}\ \Delta \vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow \langle \Delta, x \mapsto v_1 \mid e_2 \rangle$$

$$\textsc{E-Open}\ \Delta \vdash \texttt{open}\ \{\overline{\ell = v}\}\ \texttt{in}\ e \longrightarrow \texttt{let}\ \overline{\ell = v}\ \texttt{in}\ e \qquad \textsc{E-Closure}\ \frac{\Delta' \vdash e \longrightarrow e'}{\Delta \vdash \langle \Delta' \mid e \rangle \longrightarrow \langle \Delta' \mid e' \rangle}$$

$$\textsc{E-ClosureV}\ \Delta \vdash \langle \Delta' \mid v \rangle \longrightarrow v \qquad \textsc{E-Context}\ \frac{\Delta \vdash e \longrightarrow e'}{\Delta \vdash E[e] \longrightarrow E[e']}$$

Fig. 7: Environment-based evaluation rules of λ_{open}

$$\textsc{T-Open}\ \frac{\Gamma, \overline{\ell_j : A_j} \vdash e_1 \Rightarrow \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}\} \qquad \Gamma, \overline{\ell_j : A_j}, \overline{\ell_i : A_i} \vdash e_2 \Rightarrow B}{\Gamma, \overline{\ell_j : A_j} \vdash \texttt{open}\ e_1\ \texttt{in}\ e_2 \Rightarrow B}$$

Fig. 8: The typing rule of open-bindings

Operational semantics. As shown in Fig. 7, a valuation environment Δ, which binds variable names to their corresponding values, is added to each evaluation rule. The expression $\langle \Delta \mid e \rangle$ saves an environment inside so that evaluation can later resume with a saved environment, among which $\langle \Delta | \lambda x : A.\ e \rangle$ is well knwon as a *function closure*. Briefly speaking, closures are used to ensure lexical scoping. The extension of open-bindings is rather simple: E-Open converts open-bindings to let-bindings depending on the evaluated result of the record.

Typing. The typing rule of open-bindings in Fig. 8 may need some explaining. T-Open first figures out the optional fields from the type of e_1 and checks if these names are already in the typing environment. This is because we assume that all optional arguments have their default values defined in advance. If the requirement is met, we go on to calculate the type of e_2 with the type information of all fields appended to the environment. By the way, the check of an open-binding degenerate into something like a check of multiple let-bindings if there is no optional fields.

Remarks. Overall, we prefer λ_{proj} to λ_{open} because we want to keep the extension to $\lambda_{<:}$ as simple as possible. Although the operational semantics of λ_{open} is unusual, the implementation of open-bindings should not be harder than failable projections since we seldom use textual substitution owing to inefficiency. Instead, a practical implementation is probably more close to our closure-based operational semantics. Moreover, an open-binding itself is a useful language construct, similar to the `open` directive in the ML module system or record wildcards in Haskell. It is also interesting to us that a seemingly concise design can finally lead to a relatively sophisticated formalization.

5 Conclusion

Named and optional arguments are widely supported in object-oriented programming languages but are hardly formalized. Garrigue et al. formalized a label-selective λ-calculus for OCaml that combines commutativity and currying, but it is non-trivial to be integrated with other sophisticated λ-calculi. OCaml goes to the extreme of pursuing fancy features, while Haskell goes to the other extreme of lacking native support for named arguments. It is well known in Haskell that named arguments can be encoded as records, but it requires a lot of boilerplate code to support both required and optional arguments. In this paper, we presented a minimal extension to $\lambda_{<:}$ that serves as a type-safe core calculus. Based on two alternative ways of desugaring, named and optional arguments can be encoded as records.

Although we keep the calculus as simple and modular as possible, it is impossible to avoid *every* potential conflict caused by feature interaction. If a language is simply incompatible with subtyping, for an extreme example, our approach does not work. Nevertheless, we believe that our approach works in most situations. We hope that functional language designers who are concerned about named arguments can benefit from this paper.

Future work. We plan to prove the type soundness of λ_{proj} and λ_{open} using Coq in the near future. Moreover, it is worth investigating how to adapt our approach for a record calculus that uses row polymorphism rather than subtyping.

Acknowledgments. This work has been sponsored by the Hong Kong Research Grant Council project numbers 17209519, 17209520, and 17209821.

References

1. Aït-Kaci, H., Garrigue, J.: Label-selective lambda-calculus: syntax and confluence. Theor. Comput. Sci. **151**(2) (1995)
2. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5) (2021)
3. Furuse, J.P., Garrigue, J.: A label-selective lambda-calculus with optional arguments and its compilation method. Tech. rep., Kyoto University (1995)
4. Garrigue, J.: Objective Label trilogy, `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/`
5. Garrigue, J.: Labeled and optional arguments for Objective Caml. In: JSSST SIGPPL (2001)
6. Garrigue, J., Aït-Kaci, H.: The typed polymorphic label-selective lambda-calculus. In: POPL (1994)
7. Parsons, M.: Persistent: type-safe, multi-backend data serialization, `https://hackage.haskell.org/package/persistent`
8. Pierce, B.C., et al.: Programming Language Foundations, Software Foundations, vol. 2. `https://softwarefoundations.cis.upenn.edu/plf-current/`
9. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2) (1975)
10. Rytz, L., Odersky, M.: Named and default arguments for polymorphic object-oriented languages. In: SAC (2010)
11. Snoyman, M.: Warp: a fast, light-weight web server for WAI applications, `https://hackage.haskell.org/package/warp`