Named Arguments as Intersections, Optional Arguments as Unions

Yaozhu Sun and Bruno C. d. S. Oliveira

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

Abstract. Named and optional arguments are prevalent features in many mainstream programming languages, enhancing code readability and flexibility. Despite widespread use, their formalization has not been extensively studied. This paper bridges this gap by presenting a type-safe foundation for named and optional arguments using intersection and union types. We begin by identifying a critical type-safety issue in popular static type checkers for Python and Ruby, particularly in handling first-class named arguments in the presence of subtyping. Our solution involves rewriting call sites to ensure type safety, which we formalize through a translation into a core calculus called λ_{iu}. The type safety of the translation is proven using the Coq proof assistant. The practical implementation of our approach in the CP language validates our theoretical contributions. Furthermore, we informally discuss how our approach could be adapted to encode named and optional arguments in other existing languages.

1 Introduction

The λ-calculus, introduced by Alonzo Church [5], shows how to model computation solely with function abstraction and application. For example, natural numbers, boolean values, pairs, and lists, as well as various operations on them, can be represented by higher-order functions via Church encoding. 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 more than one argument is desired in those languages, we need to create a sequence of functions, each with a single argument, and perform an iteration of applications. This idea is called *currying*. Currying brings brevity to functional programming and naturally allows partial application, but it usually 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 in practical programming and can be met in a language that supports *named* and *optional* arguments. Named arguments also largely improve the readability of function calls. For example, it is unclear which is the source and which is the destination in `copy(x, y)`, while `copy(to: x, from: y)` is self-explanatory.

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

exp(10, 2)  #= exp(x=10, base=2) = 1024
exp(base=2, x=10)              #= 1024
exp(x=10)                      #= e^10

args = { "base": 2, "x": 10 }
exp(**args) #= exp(base=2, x=10) = 1024
```

(a) The Python way.

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

args = { base: 2, x: 10 }
exp(**args)         #= 1024
```

(b) The Ruby way.

Fig. 1: Named arguments in Python and Ruby.

Named arguments are widely supported in mainstream programming languages, such as Python, Ruby, OCaml, C#, and Scala, just to name a few. The earliest instance, to the best of our knowledge, is Smalltalk, where every method argument *must* be associated with a *keyword* (i.e. an external name). In other words, there are no positional arguments (i.e. arguments with no keywords) in Smalltalk. The syntax of modern languages is usually less rigid, so programmers can choose whether to attach keywords to arguments or not. There are two ways to reconcile positional and named arguments. One way, employed by Python and shown in Fig. 1a, is to make parameter names in a function definition as non-mandatory keywords. Thus, every argument can be passed with or without keywords by default. As shown in the Python code, `exp(10, 2)` is equivalent to `exp(x=10, base=2)`. To reconcile the two forms in the same call, a restriction is imposed that all named arguments must follow positional ones. The other way, shown in Fig. 1b and used in Ruby, is to strictly distinguish named arguments from positional ones. When defining a Ruby function, a keyword parameter should always end with a colon even if it does not have a default value. By this means, they are syntactically distinct from positional parameters, and their keywords cannot be omitted in a function call. There is also a restriction in Ruby that all named arguments must follow positional ones in both function definitions and call sites. The two kinds of arguments are usually used in different scenarios: positional arguments are used when the number of arguments is small and the order is clear, while named arguments are used in more complex cases especially when settings or configurations are involved.

More interestingly, named arguments are *first-class* values in Python and Ruby: they can be assigned to a variable. As shown at the bottom of Fig. 1, the variable `args` stores the two arguments named `base` and `x`, and we can later pass it to `exp` by unpacking it with `**` (sometimes called the splat operator). In fact, `args` is a dictionary in Python and similarly a hash in Ruby. Thus, first-class named arguments can be manipulated and passed around like standard data structures. This feature is widely used in Python and Ruby.

Table 1: Named arguments with different design choices in different languages.

	Smalltalk	Python	Ruby	Racket	OCaml	C#	Scala	Dart	Swift	This paper
Commutativity	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓
Optionality	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Currying	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Distinctness	*n.a.*†	✗	✓	✓	✓	✗	✗	✓	✓	✓
First-class value	✗	✓	✓	✓‡	✗	✗	✗	✗	✗	✓
Static typing	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Soundness proof	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓

† Smalltalk does not support positional arguments at all.
‡ Racket's support for first-class named arguments is limited and forbids commutativity.

Including the distinctness and first-class values illustrated above, we have identified five important design choices found in existing languages that support named arguments:

1. *Commutativity*: whether the order of (actual) arguments can be different from that of (formal) parameters originally declared.
2. *Optionality*: whether some arguments can be omitted in a function call if their default values are predefined.
3. *Currying*: whether a function that takes more than one argument is always converted into a chain of functions that each take a single argument.
4. *Distinctness*: whether named arguments are distinct from positional ones in how they are defined and passed.
5. *First-class value*: whether named arguments are first-class values.

As shown in Table 1, the first two properties hold for most mainstream programming languages, with Smalltalk and Swift being two exceptions. Commutativity and optionality are so useful that we believe they should not be compromised. Concerning the third point, OCaml is the only language that manages to reconcile currying with commutativity, though 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 here 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 or configurations. The fourth design, distinctness, is endorsed by Ruby, Racket, OCaml, Dart, and Swift. It improves the readability of call sites to enforce keywords whenever arguments are defined to be named. We advocate distinctness in this paper also because it simplifies the language design and allows us to focus on more important topics, especially type safety with first-class named arguments.

Although named arguments are ubiquitous, they have not attracted enough attention in the research of programming languages. Among the few related papers, the work by Garrigue et al. [1,9,12] formalizes a label-selective λ-calculus and eventually applies it to OCaml [11]. Another work by Rytz and Odersky [26]

discusses the design of named and optional arguments in Scala, but it mainly focuses on practical aspects. The core features of Scala are formalized in a family of DOT calculi, but named arguments are never included. The support for named arguments is implemented as macros in Racket [7]. So their extension is more like userland syntactic sugar and requires no changes to the core compiler. Haskell does not support named arguments natively, but the paradigm of *named arguments as records* is folklore. We will discuss OCaml, Scala, Racket, and Haskell in detail in Section 5. In short, named arguments are implemented in an ad-hoc manner and are not well founded from a type-theoretic perspective in most languages, especially object-oriented ones. Only OCaml and this paper provide *soundness proofs* for the feature of named arguments.

An important issue that has not been explored in the literature is the interaction between subtyping and first-class named arguments. A naive design can easily lead to a type-safety issue. We will show in Section 2.2 that the most widely used optional type checker for Python, mypy [13], fails to detect a type-unsafe use of first-class named arguments. The same issue also exists in Ruby with Steep [16] or Sorbet [28]. It arises from subtyping hiding some arguments from their static type and bypassing the type checking for optional arguments. As a result, an optional argument may have an unexpected type at run time, which leads to a runtime error.

In this paper, we present a type-safe foundation for named and optional arguments. At the heart of our approach is the translation into a core calculus called λ_{iu}, which features *intersection and union types* [2,6,8]. Our approach supports first-class named arguments like Python and Ruby, but the type-safety issue is addressed by us. The λ_{iu} calculus has been shown to be type-sound [23], and we show that our translation from our source language into λ_{iu} is type-safe. Thus, we establish the type safety of our approach. In addition, our design has recently been incorporated into the CP language [33].

In summary, the contributions of this paper are:

- We identify a type-safety issue with first-class named arguments in the presence of subtyping and propose a solution based on call site rewriting.
- We demonstrate how a minimal language with named and optional arguments can be translated to a core calculus with intersection and union types.
- We formalize the translation as an elaboration semantics and prove its type safety using the Coq proof assistant.
- We validate our theoretical contributions by implementing our approach in the CP language and showcasing a practical example.
- We conduct a survey of named arguments in existing programming languages and discuss the best practice for *named arguments as records* in Haskell.

2 Named and Optional Arguments: The Bad Parts

Since named and optional arguments are not well studied in most languages, the ad-hoc mechanisms employed in those languages may sometimes surprise programmers or even cause safety issues.

2.1 Gotcha! Mutable Default Arguments in Python

Let us consider a simple Python function that appends an element to a list. We provide a default value for the list, which is an empty list:

```
def append(x, xs=[]):
  xs.append(x)
  return xs
```

```
append(1) #= [1]
append(2) #= [1, 2]
```

After calling `append(1)`, we get the expected result `[1]`. However, continuing to call `append(2)` gives us `[1, 2]` instead of `[2]`. This is because Python only evaluates the default value once when the function is defined, so the same list initialized for `xs` is shared across different calls to `append`. When calling `append(2)`, the default value for `xs` is no longer an empty list but the list that has been modified by the previous call `append(1)`.

This issue, while seemingly minor, highlights the importance of understanding the semantics of default arguments. Our design strives to avoid such surprises, following the principle of least astonishment, yet this is not our main focus. We will discuss the more critical issue about type safety next.

2.2 Caution! Type Safety with First-Class Named Arguments

As we have shown in Fig. 1, quite a few languages, especially dynamically typed ones like Python and Ruby, treat named arguments as first-class values. This feature is particularly helpful for passing settings because they are usually stored in a separate configuration file. We can read the settings from the file and pass them as named arguments using the `**` operator. For example, we can find such code in Python to run a web server:

```
class App: # from a web server library
  def run(self, host: str, port: int, debug: bool = False):
    assert isinstance(debug, bool) # actual code omitted...

args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args) #= app.run(host="0.0.0.0", port=80, debug=True)
```

Although Python is dynamically typed, there is continuous effort in the Python community to improve the detection of type errors earlier in the development process, primarily through static analysis. There is an optional static type checker for Python called mypy [13]. In the example above, we make use of *type hints*, introduced in Python 3.5, to specify the types of the parameters and the return value of the `run` method. The type hints have no effect at run time but can be used by external tools like mypy to statically check if the code is well-typed. Perhaps surprisingly, the code above cannot pass mypy's type checking, because the type inferred for `args` (i.e. `dict[str,object]`) is not precise enough. The type checker needs to know what keys `args` exactly has and what types the values associated with those keys have, in order to make sure that `**args` is compatible with the parameters of `app.run`.

Fortunately, `TypedDict` is added in Python 3.8 to represent a specific set of keys and their associated types. By default, every specified key is required, except when it is marked as `NotRequired`, which is a type qualifier added in Python 3.11. With `TypedDict` and `NotRequired`, we can now define a precise dictionary type for `args` that passes mypy's type checking:

```
class Args(TypedDict):
  host: str
  port: int
  debug: NotRequired[bool]

args0: Args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args0) # type-checks in mypy
args1: Args = { "host": "0.0.0.0", "port": 80 }
app.run(**args1) # type-checks in mypy, too
```

The mypy type checker will raise an error if we provide an argument with an incompatible type, such as a string for the `debug` key:

```
class In(TypedDict):
  host: str
  port: int
  debug: str

args2: In = { "host": "0.0.0.0", "port": 80, "debug": "Oops!" }
app.run(**args2) # TypeError: Argument "debug"
# has incompatible type "str"; expected "bool"  [arg-type]
```

However, mypy's type system is not completely type-safe. We can create a function `f` that takes a dictionary with three keys specified in type `In` and returns a dictionary with only two keys specified in type `Out`. The function type-checks in mypy because type `In` is compatible whenever type `Out` is expected. Roughly speaking, it means that `In` is a subtype of `Out`. Then we can use `f` to forget the `debug` key in the static type:

```
class Out(TypedDict):
  host: str
  port: int

def f(args: In) → Out: return args

args3 = f(args2) # still contains { "debug": "Oops!" }
app.run(**args3) # type-checks in mypy, but has a runtime error!
```

Here `args3` has type `Out` without the `debug` key specified. From a static viewpoint, `args3` only has two keys `host` and `port`, which are compatible with the parameters of `app.run` since `debug` is optional and has a default value. That is why `app.run(**args3)` type-checks in mypy. However, at run time, the `debug` key is still present in `args3`, so the string `"Oops!"` is passed as a named argument

to `app.run`, which originally expects a boolean value. This results in a runtime error since there is an assertion in `app.run` to ensure that `debug` is boolean.

This issue is not unique to Python and mypy. We have reproduced nearly the same issue in Ruby with two popular type checkers, namely Steep [16] and Sorbet [28], which is illustrated in Appendices A&B.

In conclusion, subtyping can lead to a fundamental type-safety issue when dealing with first-class named and optional arguments. In essence, the following subsumption chain is questionable:

```
   { host: str, port: int, debug: str }
<: { host: str, port: int }
<: { host: str, port: int, debug?: bool }
```

Following this chain bypasses mypy's type compatibility checking for the `debug` key. Next we will show how to break the chain and address the type-safety issue.

3 Our Type-Safe Approach

In this section, we informally present how we translate named and optional arguments into a core language with intersection and union types, while retaining type safety. We start by introducing the core language constructs that we need. Then we illustrate our translation scheme by example and demonstrate how it recovers type safety. After that, we showcase a practical example in the CP language, which has incorporated our approach to support named and optional arguments. Finally, we discuss how our translation scheme can be applied to other languages.

3.1 Core Language

The core language features intersection and union types, which establish an elegant duality in the type system. A value of the intersection type $A \wedge B$ can be assigned both A and B, whereas a value of the union type $A \vee B$ can be assigned either A or B. Intersection and union types correspond to the logical conjunction and disjunction respectively. Similar calculi are widely studied [2,6,8] and provide a well-understood foundation for named and optional arguments.

Named Arguments as Intersections. Named arguments are translated to multi-field records. However, the core language does not support multi-field records directly. There are only single-field records in the core language, and multiple fields are represented as intersections of single-field record types. For example, $\{x : \mathbb{Z}\} \wedge \{y : \mathbb{Z}\}$ represents a record type with two integer fields x and y. With intersection types, width subtyping for record types comes for free, and permutations of record fields are naturally allowed [24].

At the term level, a merge operator [6,23] (denoted by $,$ in this paper) is used to concatenate multiple single-field records to form multi-field records, reminiscent of Forsythe [24]. For example, $\{x = 1\} \,, \{y = 2\}$ forms a two-field record from two single-field records.

Optional Arguments as Unions. Optional arguments are translated to nullable types. A nullable type is not implicit in the core language but is represented as a union with the null type [18]. For example, an optional integer argument named z is translated to $\{z : \mathbb{Z} \vee \mathbf{Null}\}$.

At the term level, a type-based switch expression [8,23] is used to scrutinize a term of a union type, reminiscent of ALGOL 68 [32]. For example, $\mathbf{switch}\, z\, \mathbf{case}\, \mathbb{Z} \Rightarrow e_1\, \mathbf{case}\, \mathbf{Null} \Rightarrow e_2$ returns e_1 if z is an integer or e_2 if **null**.

3.2 Translation by Example

Let us review the previous Python function in Section 2.1 that appends an element to a list, which defaults to an empty list:

```
def append(x: int, xs: list[int] = []): ...
```

The function will be translated to a core function as follows:

$$
\begin{aligned}
\texttt{append} = \lambda &\mathit{args} : \{x : \mathbb{Z}\} \wedge \{xs : [\mathbb{Z}] \vee \mathbf{Null}\}. \\
&\mathbf{let}\ x = \mathit{args}.x\ \mathbf{in} \\
&\mathbf{let}\ xs = \mathbf{switch}\ \mathit{args}.xs\ \mathbf{as}\ xs\ \mathbf{case}\ \mathbb{Z} \Rightarrow xs\ \mathbf{case}\ \mathbf{Null} \Rightarrow [\,]\ \mathbf{in} \\
&\ldots
\end{aligned}
$$

Here we can see that the default value (i.e. the empty list) is not shared across different calls to `append` because the default value is evaluated within the function body. Therefore, calling `append(x=1)` will consistently return `[1]` instead of surprisingly modifying the default value. This design leads to less astonishment and more predictable behavior.

Since we translate named parameter types to record types, we correspondingly translate named arguments to records. For example, the function call `append(x=1, xs=[0])` will be translated to $\texttt{append}\,(\{x = 1\}\,, \{xs = [0]\})$.

Rewriting Call Sites. More importantly, we also rewrite call sites to add null values for absent optional arguments. For example, the function call `append(x=1)` will be rewritten and translated to $\texttt{append}\,(\{x = 1\}\,, \{y = \mathbf{null}\})$.

Dependent Default Values. Another advantage of our translation scheme is that it naturally allows default values to depend on earlier arguments. Python and Ruby do not support dependent default values, but this feature can be useful in some practical scenarios. For example, when setting up I/O, we may want to output error messages to the same stream as `out` by default:

```
def setIO(in_, out, err = out): ...
```

The variable `out` can be used in the default value of `err` because it has been brought into scope by the previous let-in binding:

$$
\begin{aligned}
&\mathbf{let}\ \mathit{out} = \mathit{args}.\mathit{out}\ \mathbf{in} \\
&\mathbf{let}\ \mathit{err} = \mathbf{switch}\ \mathit{args}.\mathit{err}\ \mathbf{as}\ \mathit{err}\ \mathbf{case}\ \mathbf{IO} \Rightarrow \mathit{err}\ \mathbf{case}\ \mathbf{Null} \Rightarrow \mathit{out}\ \mathbf{in}
\end{aligned}
$$

3.3 Recovering Type Safety

The type safety of our translation scheme is essentially guaranteed by call site rewriting. Besides adding null values for absent optional arguments, we also sanitize arguments to ensure that they are expected from the parameter list. Since named arguments are first-class and can be passed as a variable, we may not have literals like `append(x=1, xs=[0])` but splats like `append(**args)`. So the matching between (formal) parameters and (actual) arguments is performed based on their static types:

- If `args` has type $\{x : \mathbb{Z}\} \wedge \{xs : [\mathbb{Z}]\}$, the call site will be rewritten to something equivalent to `append(x=args.x, xs=args.xs)`.
- If `args` only has type $\{x : \mathbb{Z}\}$, the call site will be rewritten to something equivalent to `append(x=args.x, xs=null)`.

For the `append` function, no other cases can pass the sanitization process.

Let us review the previous type-unsafe Python example in Section 2.2:

```
def f(args: In) → Out: return args
args = f({ "host": "0.0.0.0", "port": 80, "debug": "Oops!" })
app.run(**args) #= app.run(host="0.0.0.0", port=80, debug="Oops!")
```

Recall that `args` has type `Out`, which is similar to `{ host: str, port: int }`. The `debug` key is forgotten in the static type but is still present at run time. It passes mypy's type checking but raises a runtime error. In our translation scheme, the call site will be rewritten to the following form based on the type of `args` (i.e. `Out`):

```
app.run(host=args.host, port=args.port, debug=null)
```

Therefore, type safety is recovered in our translation scheme.

Takeaways. There are two important observations from our translation scheme:

1. `{ required: A, optional?: B }` is not equivalent to `{ required: A }` because the former contains more information that prevents `optional` from being associated with other types than `B`. In other words, the `optional` argument can be absent, but if it is present, it must have type `B`.
2. Corresponding to the above observation at the type level, we explicitly pass a null value as an optional argument if it is statically missing. The null value fills the position of a potentially forgotten argument that may have a wrong type. In other words, we implement the splat operator as per the static type of named arguments.

3.4 Implementation in the CP Language

Our approach to named and optional arguments has been implemented in the CP language, a statically typed language for *compositional programming* [33]. CP supports not only intersection and union types but also the merge operator

```
-- from a SVG library in CP
SVG: { width: Int; height: Int } → [Element] → Graphic;  -- <svg>
Rect: { x: Int; y: Int; width: Int; height: Int
      ; rx?: Int; ry?: Int; color?: String } → Element;   -- <rect>
......
-- client code
fractal { level = 4; x: Int; y: Int; width: Int; height: Int } =
  let args = { level = level-1; width = width/3; height = height/3 } in
  let center = Rect (args,{ x = x + width/3; y = y + height/3
                          ; color = "white" }) in
  if level == 0 then [center]
  else fractal (args,{ x = x;             y = y })
    ++ fractal (args,{ x = x + width/3;   y = y })
    ++ fractal (args,{ x = x + width*2/3; y = y })
    ++ fractal (args,{ x = x;             y = y + height/3 })
    ++ [center]
    ++ fractal (args,{ x = x + width*2/3; y = y + height/3 })
    ++ fractal (args,{ x = x;             y = y + height*2/3 })
    ++ fractal (args,{ x = x + width/3;   y = y + height*2/3 })
    ++ fractal (args,{ x = x + width*2/3; y = y + height*2/3 });
init = { x = 0; y = 0; width = 600; height = 600; color = "black" };
main = SVG init ([Rect init] ++ fractal init);
```

Fig. 2: Sierpiński carpets implemented in the CP language.

and type-based switch expression. The implementation of named and optional arguments in CP is a direct application of our translation scheme.

More interestingly, the sanitization process during call site rewriting comes for free because CP employs a *coercive* semantics for subtyping [14]. For example, a subtyping relation between `{ host: str, port: int, debug: str }` and `{ host: str, port: int }` implies a coercion function from subtype to supertype. In CP, such coercions are implicitly inserted to remove the forgotten fields (e.g. `debug` in this case). Therefore, the only remaining work is to add null values for absent optional arguments.

To demonstrate the use of named and optional arguments in CP, we show a fractal example in Fig. 2, which is adapted from previous work on CP's application to embedded DSLs [29]. The code makes use of named and optional arguments a lot, including both the `SVG`/`Rect` constructors from the library and the `fractal` function defined by the client. For example, `fractal` has five named arguments (`level`, `x`, `y`, `width`, and `height`), among which `level` is optional with a default value of `4`.

It is worth noting that named arguments are used as first-class values in the CP code. On the first line of the `fractal` body, we store three fields `level`, `width`, and `height` in a variable `args`, which are shared arguments for later calls. When constructing the center rectangle, we merge `args` with three more fields `x`, `y`, and `color` to form a full set of named arguments we need for the

`Rect` constructor. When recursively calling `fractal`, we pass `args` merged with different `x` and `y` values to draw the eight sub-copies. In the `main` function, we also use a variable `init` to avoid repeating the same set of arguments for `SVG`, `Rect`, and `fractal` calls. The `**` operator is not needed in CP when passing first-class named arguments. Note that the parameter lists of these three constructors/functions are not completely the same, but we can still use a larger set of named arguments to cover all the cases. This is possible because CP allows subtyping for named arguments while retaining type safety.

3.5 Applications to Other Languages

Although we base our translation scheme on a core language with intersection and union types for type-theoretic solidness and elegance, it can work for a wider range of languages. We discuss the alternatives to intersections and unions below.

Alternative to Intersections. Record types have existed long before intersection types were invented. In practice, multi-field records are rarely represented as intersections of single-field records. For example, *Software Foundations* [22] demonstrates how to directly model multi-field records and define depth, width, and permutation subtyping without intersections, though their formalization is more complex than ours.

There is a merge operator in our translation scheme, but we only use it to construct multi-field records statically. Although the merge operator can be powerful if we want to construct first-class named arguments at run time like in CP, its absence does not disable our translation scheme. In other words, we only assume a simplified version that does not merge terms dynamically.

Alternative to Unions. Nullable types are rarely represented as unions with the null type too. For example, C#, Kotlin, and Dart support nullable types as a primitive data structure. Putting a question mark behind any type makes it nullable in these languages (e.g. `int?`).

No matter how a nullable type is represented, there is usually some expression that can check whether a nullable value is null or not. For example, C# provides the `is` operator to examine the runtime type, which is generally known as type introspection and is similar to the type-based switch. C# also provides the null coalescing operator `??` and simplifies the common pattern **switch** e **as** x **case** $A \Rightarrow x$ **case Null** $\Rightarrow d$ as `e??d` for nullable values.

Dynamically Typed Languages. It may be surprising at first sight that dynamically typed languages can benefit from our work with static typing, but recall that the type-safety issue in Section 2.2 was found in Python. Nowadays, popular dynamically typed languages have been retrofitted with gradual typing. For example, Python has type hints and mypy [13], Ruby has RBS and Steep [16], JavaScript gets typed by TypeScript [17], and Lua gets typed by Luau [25]. All of

these typed versions support record-like and union types, and all except Python also support intersection types. Our translation scheme can almost directly apply to these languages. For a concrete example, we show how the aforementioned `exp` function can be encoded in TypeScript:

```
function exp(args: { x: number } & { base: number|null }) {
  let x = args.x;
  let base = (typeof args.base === "number") ? args.base : Math.E;
  return Math.pow(base, x);
}
exp({ base: 2, x: 10 })     //= 1024
exp({ x: 10, base: null })  //= e^10
```

The code is almost the same as in Section 3.2. Note that the **`typeof`** operator is the standard way to perform type introspection in TypeScript, and the type of `args.base` is refined from the **`number|null`** to **`number`** in the true-branch. We assume the call sites have been rewritten in the code above. In this manner, named and optional arguments can be added to TypeScript as syntactic sugar.

Although we have discussed several alternatives to intersection and union types, we believe that if a language is designed from scratch, our approach is a good choice. Intersection and union types not only subsume multi-field record and nullable types but also provide a solid and elegant foundation for other advanced features, such as function overloading and heterogeneous data structures. Castagna's essay [3] is an excellent further reading on the beauty of programming with intersection and union types.

4 Formalization

In this section, we formalize the translation of named and optional arguments as an elaboration semantics. The target of elaboration is called λ_{iu}, and the source is called Uaena. We prove that the source language with named and optional arguments is type-safe via (1) the type soundness of the target calculus and (2) the type soundness of elaboration. All the theorems are mechanically proven using the Coq proof assistant.

4.1 The Target Calculus: $\boldsymbol{\lambda_{\mathsf{iu}}}$

λ_{iu} is an extension to the calculus in Chapter 5 of Rehman's dissertation [23] with **null**, single-field records, and let-in bindings. The addition of let-in bindings is not essential because they can be desugared into lambda abstractions and applications:

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \quad \equiv \quad (\lambda x.\ e_2)\ e_1$$

However, we still have let-in bindings for the sake of readability, and this form of let-in bindings simplifies the rules of parameter elaboration (introduced later in Fig. 5). Another difference is that the original calculus uses the locally nameless representation [4] while ours directly uses names for bound variables.

Our changes to Rehman's calculus are relatively trivial, and we do not touch the rules for intersection and union types. We will not discuss his design choices in this paper, because our focus is on the type soundness with the addition of **Null** and record types. We have proven in Coq that these extensions preserve type soundness.

Syntax of λ_{iu}

$$\begin{array}{lll} \text{Types} & A, B ::= & \top \mid \bot \mid \mathbf{Null} \mid \mathbb{Z} \mid A \to B \mid \{\ell : A\} \mid A \wedge B \mid A \vee B \\ \text{Expressions} & e ::= & \{\} \mid \mathbf{null} \mid n \mid x \mid \lambda x\!:\!A.\ e\!:\!B \mid e_1\ e_2 \mid \{\ell : A = e\} \mid e.\ell \\ & & \mid e_1\ ,\ e_2 \mid \mathbf{switch}\ e_0\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 \mid \mathit{letin}\ e \end{array}$$

The types include the top type $\top$, the bottom type $\bot$, the null type **Null**, the integer type $\mathbb{Z}$, function types $A \to B$, record types $\{\ell : A\}$, intersection types $A \wedge B$, and union types $A \vee B$. **Null** is a unit type that has only one value **null**.

The expressions include the empty record $\{\}$, the null value **null**, integer literals n, variables x, lambda abstractions $\lambda x : A.\ e : B$, function applications $e_1\ e_2$, record literals $\{\ell : A = e\}$, record projections $e.\ell$, merges $e_1\,,e_2$, type-based switch expressions $\mathbf{switch}\ e_0\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2$, and let-in bindings $\mathit{letin}\ e$. The syntax of letin is as follows:

$$\mathit{letin} ::= \mathbf{let}\ x = e\ \mathbf{in} \mid \mathit{letin}_1 \circ \mathit{letin}_2 \mid \mathbf{id}$$

The composition of two let-in bindings is denoted by $\mathit{letin}_1 \circ \mathit{letin}_2$, and an empty binding is denoted by **id**.

Subtyping. Fig. 3 shows the subtyping rules of λ_{iu}. The rules are standard for a type system with intersection and union types. Rule Sub-Top shows that the

$\boxed{A <: B}$ *(Subtyping)*

$$\textsc{Sub-Null}\ \frac{}{\mathbf{Null} <: \mathbf{Null}} \qquad \textsc{Sub-Int}\ \frac{}{\mathbb{Z} <: \mathbb{Z}} \qquad \textsc{Sub-Arrow}\ \frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2} \qquad \textsc{Sub-Rcd}\ \frac{A <: B}{\{\ell : A\} <: \{\ell : B\}}$$

$$\textsc{Sub-And}\ \frac{A <: B \qquad A <: C}{A <: B \wedge C} \qquad \textsc{Sub-AndL}\ \frac{A <: C}{A \wedge B <: C} \qquad \textsc{Sub-AndR}\ \frac{B <: C}{A \wedge B <: C} \qquad \textsc{Sub-Or}\ \frac{A <: C \qquad B <: C}{A \vee B <: C}$$

$$\textsc{Sub-OrL}\ \frac{A <: B}{A <: B \vee C} \qquad \textsc{Sub-OrR}\ \frac{A <: C}{A <: B \vee C} \qquad \textsc{Sub-Top}\ \frac{}{A <: \top} \qquad \textsc{Sub-Bot}\ \frac{}{\bot <: A}$$

Fig. 3: Subtyping of λ_{iu}.

top type $\top$ is a supertype of any type, and rule Sub-Bot shows that the bottom type $\bot$ is a subtype of any type. Rules Sub-And, Sub-AndL, and Sub-AndR handle the subtyping for intersection types, while rules Sub-Or, Sub-OrL, and Sub-OrR are for union types. Rules Sub-Null and Sub-Rcd added by us are straightforward. We prove that the subtyping relation is reflexive and transitive.

Theorem 1 (Subtyping Reflexivity). $\forall A, A <: A$.

Theorem 2 (Subtyping Transitivity). *If* $A <: B$ *and* $B <: C$*, then* $A <: C$.

Typing. Fig. 4 shows the typing rules of λ_{iu}. The empty record $\{\}$ has the top type $\top$, as shown in rule Typ-Top. Rule Typ-Merge is the introduction rule for intersection types. Merging two functions is used for function overloading, and merging two records is used for record concatenation. Rule Typ-Switch is the elimination rule for union types. The type-based switch expression scrutinizes an expression having a union of the two scrutinizing types (i.e. $e_0 : A \vee B$). This premise ensures the exhaustiveness of the cases in the switch. The **as**-variable x is refined to type A in e_1 and to type B in e_2. Rules Typ-Null,

Typing contexts $\quad \Gamma ::= \cdot \mid \Gamma, x : A$

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\textsc{Typ-Top}\ \frac{}{\Gamma \vdash \{\} : \top} \qquad \textsc{Typ-Null}\ \frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Null}} \qquad \textsc{Typ-Int}\ \frac{}{\Gamma \vdash n : \mathbb{Z}} \qquad \textsc{Typ-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Typ-Abs}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x\!:\!A.\ e\!:\!B) : A \to B} \qquad \textsc{Typ-App}\ \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B} \qquad \textsc{Typ-Rcd}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \{\ell : A = e\} : \{\ell : A\}}$$

$$\textsc{Typ-Prj}\ \frac{\Gamma \vdash e : \{\ell : A\}}{\Gamma \vdash e.\ell : A} \qquad \textsc{Typ-Let}\ \frac{\Gamma \vdash \mathit{letin} \dashv \Gamma' \quad \Gamma' \vdash e : A}{\Gamma \vdash \mathit{letin}\ e : A} \qquad \textsc{Typ-Merge}\ \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1\ ,\ e_2 : A \wedge B}$$

$$\textsc{Typ-Switch}\ \frac{\Gamma \vdash e_0 : A \vee B \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, x : B \vdash e_2 : C}{\Gamma \vdash \mathbf{switch}\ e_0\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 : C} \qquad \textsc{Typ-Sub}\ \frac{\Gamma \vdash e : A \quad A <: B}{\Gamma \vdash e : B}$$

Fig. 4: Typing of λ_{iu}.

Typ-Rcd, and Typ-Prj added by us are straightforward. Rule Typ-Let uses an auxiliary judgment $\Gamma \vdash \mathit{letin} \dashv \Gamma'$ (defined in Appendix C) to obtain the typing context for the body of the let-in binding. For example, if e_1 has type A, then $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$ adds $x : A$ to the typing context before type-checking e_2.

Dynamic Semantics. We have a small-step operational semantics for λ_{iu}. The judgment $e \longrightarrow e'$ means that e reduces to e' in one step, and $e \longrightarrow^* e'$ is for multi-step reduction. We extend the original dynamic semantics by adding rules for records and projections. Similarly to the applicative dispatch for function applications in the original calculus, we add a relation called projective dispatch for record projections. For example, $(\{x = 1\} \,,\, \{y = 2\}).x$ reduces to $\{x = 1\}.x$ via projective dispatch to select the needed field.

Since the dynamic semantics of λ_{iu} is independent of the elaboration from Uaena to λ_{iu}, we omit the rules here but leave them in Appendix D. Note that the operational semantics is not commonplace in that it is type-directed and non-deterministic. Please refer to Rehman's dissertation [23] for detailed explanations.

Theorem 3 (Progress). *If* $\cdot \vdash e : A$*, then either* e *is a value or* $\exists e', e \longrightarrow e'$*.*

Theorem 4 (Preservation). *If* $\cdot \vdash e : A$ *and* $e \longrightarrow e'$*, then* $\cdot \vdash e' : A$*.*

Putting progress and preservation together, we conclude that λ_{iu} is type-sound: a well-typed term can never reach a stuck state.

Corollary 1 (Type Soundness). *If* $\cdot \vdash e : A$ *and* $e \longrightarrow^* e'$*, then either* e' *is a value or* $\exists e'', e' \longrightarrow e''$*.*

4.2 The Source Calculus: Uaena

Uaena (*Unnamed Arguments Extended with Named Arguments*) is a minimal calculus with named and optional arguments. Although the calculus is small, named arguments are supported as first-class values and can be passed to or returned by a function. Besides functions with named arguments, Uaena also supports normal functions with positional arguments. The two kinds of functions are distinguished in the syntax, as seen in Ruby, Racket, OCaml, etc.

Syntax of Uaena

Types	$\mathcal{A}, \mathcal{B} ::= \mathbb{Z} \mid (\mathcal{A}) \to \mathcal{B} \mid \{\mathcal{P}\} \to \mathcal{B} \mid \{\mathcal{K}\}$
Named parameter types	$\mathcal{P} ::= \cdot \mid \mathcal{P}; \ell : \mathcal{A} \mid \mathcal{P}; \ell? : \mathcal{A}$
Named argument types	$\mathcal{K} ::= \cdot \mid \mathcal{K}; \ell : \mathcal{A}$
Expressions	$\epsilon ::= n \mid x \mid \lambda(x{:}\mathcal{A}).\, \epsilon \mid \lambda\{\rho\}.\, \epsilon \mid \epsilon_1\, \epsilon_2 \mid \{\kappa\}$
Named parameters	$\rho ::= \cdot \mid \rho; \ell : \mathcal{A} \mid \rho; \ell = \epsilon$
Named arguments	$\kappa ::= \cdot \mid \kappa; \ell = \epsilon$

The types include the integer type $\mathbb{Z}$, normal function types $(\mathcal{A}) \to \mathcal{B}$, function types with named parameters $\{\mathcal{P}\} \to \mathcal{B}$, and (first-class) named argument types $\{\mathcal{K}\}$. The expressions include integer literals n, variables x, normal lambda abstractions $\lambda(x : \mathcal{A}).\ \epsilon$, lambda abstractions with named parameters $\lambda\{\rho\}.\ \epsilon$, function applications $\epsilon_1\ \epsilon_2$, and (first-class) named arguments $\{\kappa\}$.

A named parameter type $\mathcal{P}$ can be required $(\ell : \mathcal{A})$ or optional $(\ell? : \mathcal{A})$. If a named parameter is optional, its default value must be provided in the function definition. For example, $\lambda\{x : \mathbb{Z};\ y = 0\}.\ x + y$ has type $\{x : \mathbb{Z};\ y? : \mathbb{Z}\} \to \mathbb{Z}$. A function with named parameters can only be applied to named arguments, which are basically a list of key-value pairs. For example, the previous function can be applied to $\{x = 1;\ y = 2\}$ or $\{x = 1\}$ or a variable having a compatible type. The variable case demonstrates the first-class nature of named arguments in Uaena.

Careful readers may notice that a named argument type can also serve as the parameter of a normal function. This also demonstrates the first-class nature of named arguments. But note that a normal function that takes named arguments is different from a function with named parameters. Consider the following two functions, the former of which is a function with named parameters and the latter is a normal function:

$$\begin{array}{rcl} (\lambda\{x : \mathbb{Z};\ y = 0\}.\ x + y) & : & \{x : \mathbb{Z};\ y? : \mathbb{Z}\} \to \mathbb{Z} \\ (\lambda(\mathit{args} : \{x : \mathbb{Z};\ y : \mathbb{Z}\}).\ \mathit{args}) & : & (\{x : \mathbb{Z};\ y : \mathbb{Z}\}) \to \{x : \mathbb{Z};\ y : \mathbb{Z}\} \end{array}$$

Although both functions can be applied to $\{x = 1;\ y = 2\}$, there are two main differences between them. First, optional parameters cannot be defined in a normal function. So we cannot provide $y = 0$ as a default value in the second function. Second, x and y are not brought into the scope of the function body in a normal function. So the only accessible variable is *args* in the second function.

Elaboration. The type-directed elaboration from Uaena to λ_{iu} is defined at the top of Fig. 5. $\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e$ means that the source expression ϵ has type $\mathcal{A}$ and elaborates to the target expression e under the typing context Δ. Rules Ela-Abs and Ela-App for normal functions are straightforward. In rule Ela-NAbs for functions with named parameters, besides inferring the type of the function body ϵ and elaborating it to e, we generate let-bindings for the named parameters, which is delegated to the auxiliary judgment $\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'$. In rule Ela-NApp, there is also an auxiliary judgment $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ that rewrites call sites according to the parameter and argument types. Rules Ela-NEmpty and Ela-NField are used to elaborate named arguments.

Named Parameter Elaboration. As shown at the bottom of Fig. 5, $\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'$ means that the named parameter ρ is inferred to have type $\mathcal{P}$ and elaborates to a series of let-in bindings *letin*, given that the named parameters correspond to the target bound variable x. In the meanwhile, the typing context Δ is extended with the types of the named parameters to form Δ'. Δ' is

Typing contexts $\quad \Delta ::= \cdot \mid \Delta,\, x : \mathcal{A}$

$\boxed{\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e}$ *(Elaboration)*

$$\textsc{Ela-Int}\ \frac{}{\Delta \vdash n : \mathbb{Z} \rightsquigarrow n} \qquad \textsc{Ela-Var}\ \frac{x : \mathcal{A} \in \Delta}{\Delta \vdash x : \mathcal{A} \rightsquigarrow x}$$

$$\textsc{Ela-Abs}\ \frac{\Delta,\, x : \mathcal{A} \vdash \epsilon : \mathcal{B} \rightsquigarrow e}{\Delta \vdash \lambda(x{:}\mathcal{A}).\, \epsilon : (\mathcal{A}) \to \mathcal{B} \rightsquigarrow \lambda x{:}|\mathcal{A}|.\, e{:}|\mathcal{B}|} \qquad \textsc{Ela-App}\ \frac{\Delta \vdash \epsilon_1 : (\mathcal{A}) \to \mathcal{B} \rightsquigarrow e_1 \quad \Delta \vdash \epsilon_2 : \mathcal{A} \rightsquigarrow e_2}{\Delta \vdash \epsilon_1\, \epsilon_2 : \mathcal{B} \rightsquigarrow e_1\, e_2}$$

$$\textsc{Ela-NAbs}\ \frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta' \quad \Delta' \vdash \epsilon : \mathcal{B} \rightsquigarrow e}{\Delta \vdash \lambda\{\rho\}.\, \epsilon : \{\mathcal{P}\} \to \mathcal{B} \rightsquigarrow \lambda x{:}|\mathcal{P}|.\, \mathit{letin}\, e{:}|\mathcal{B}|} \qquad \textsc{Ela-NApp}\ \frac{\Delta \vdash \epsilon_1 : \{\mathcal{P}\} \to \mathcal{B} \rightsquigarrow e_1 \quad \Delta \vdash \epsilon_2 : \{\mathcal{K}\} \rightsquigarrow e_2 \quad \Delta \vdash_{e_2} \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e_2'}{\Delta \vdash \epsilon_1\, \epsilon_2 : \mathcal{B} \rightsquigarrow e_1\, e_2'}$$

$$\textsc{Ela-NEmpty}\ \frac{}{\Delta \vdash \{\cdot\} : \{\cdot\} \rightsquigarrow \{\}} \qquad \textsc{Ela-NField}\ \frac{\Delta \vdash \{\kappa\} : \{\mathcal{K}\} \rightsquigarrow e' \quad \Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e}{\Delta \vdash \{\kappa;\, \ell = \epsilon\} : \{\mathcal{K};\, \ell : \mathcal{A}\} \rightsquigarrow e'\,,\{\ell : |\mathcal{A}| = e\}}$$

$\boxed{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'}$ *(Named parameter elaboration)*

$$\textsc{PEla-Empty}\ \frac{}{\Delta \vdash_x \cdot : \cdot \rightsquigarrow \mathbf{id} \dashv \Delta}$$

$$\textsc{PEla-Required}\ \frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'}{\Delta \vdash_x (\rho;\, \ell : \mathcal{A}) : (\mathcal{P};\, \ell : \mathcal{A}) \rightsquigarrow \mathit{letin} \circ \mathbf{let}\, \ell = x.\ell\, \mathbf{in} \dashv \Delta',\, \ell : \mathcal{A}}$$

$$\textsc{PEla-Optional}\ \frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta' \quad \Delta' \vdash \epsilon : \mathcal{A} \rightsquigarrow e}{\begin{array}{l}\Delta \vdash_x (\rho;\, \ell = \epsilon) : (\mathcal{P};\, \ell? : \mathcal{A}) \\ \rightsquigarrow \mathit{letin} \circ \mathbf{let}\, \ell = \mathbf{switch}\, x.\ell\, \mathbf{as}\, y\, \mathbf{case}\, |\mathcal{A}| \Rightarrow y\, \mathbf{case}\, \mathbf{Null} \Rightarrow e\, \mathbf{in} \dashv \Delta',\, \ell : \mathcal{A}\end{array}}$$

Fig. 5: Type-directed elaboration from Uaena to λ_{iu}.

$\boxed{\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}$ *(Call site rewriting)*

$$\textsc{PMat-Empty}\ \frac{}{\Delta \vdash_e \cdot \diamond \mathcal{K} \rightsquigarrow \{\}} \qquad \textsc{PMat-Required}\ \frac{\mathcal{K} :: \ell \Rightarrow \mathcal{A} \quad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e', \{\ell : |\mathcal{A}| = e.\ell\}}$$

$$\textsc{PMat-Present}\ \frac{\mathcal{K} :: \ell \Rightarrow \mathcal{A} \quad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell? : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e', \{\ell : |\mathcal{A}| \vee \mathbf{Null} = e.\ell\}}$$

$$\textsc{PMat-Absent}\ \frac{\mathcal{K} :: \ell \not\Rightarrow \quad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell? : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e', \{\ell : |\mathcal{A}| \vee \mathbf{Null} = \mathbf{null}\}}$$

Fig. 6: Type-directed call site rewriting in Uaena.

used for typing the body of the function with named parameters. Rule PEla-Required simply generates $\mathbf{let}\, \ell = x.\ell\, \mathbf{in}$, while rule PEla-Optional generates $\mathbf{let}\, \ell = \mathbf{switch}\, x.\ell\, \mathbf{as}\, y\, \mathbf{case}\, |\mathcal{A}| \Rightarrow y\, \mathbf{case}\, \mathbf{Null} \Rightarrow e\, \mathbf{in}$ to provide a default value e for the **Null** case.

Call Site Rewriting. As shown in Fig. 6, $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ means that if the parameter type $\mathcal{P}$ is compatible with the argument type $\mathcal{K}$, the target expression e, which corresponds to the named arguments, will be rewritten to e'. The compatibility check is based on the parameter type $\mathcal{P}$. Rule PMat-Required handles the case where the argument is required, while rules PMat-Present and PMat-Absent handle the cases where the optional argument with a specific type is present and where the optional argument is absent, respectively. The remaining case, where the optional argument is present but associated with a wrong type, is prohibited and cannot elaborate to any term. We have two more auxiliary judgments $\mathcal{K} :: \ell \Rightarrow \mathcal{A}$ and $\mathcal{K} :: \ell \not\Rightarrow$ to indicate that the argument type $\mathcal{K}$ contains a field ℓ with type $\mathcal{A}$ or $\mathcal{K}$ does not contain ℓ, whose definitions can be found in Appendix E.

Type Translation. As we have informally mentioned in Section 3.1, we translate named parameters to intersection types and optional parameters to union types. The rules for $|\cdot|$ can be found in Appendix F. Having defined the translation, we can prove the soundness of call site rewriting and elaboration.

Theorem 5 (Soundness of Call Site Rewriting). *If* $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ *and* $|\Delta| \vdash e : |\mathcal{K}|$*, then* $|\Delta| \vdash e' : |\mathcal{P}|$.

Theorem 6 (Soundness of Elaboration). *If* $\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e$*, then* $|\Delta| \vdash e : |\mathcal{A}|$.

With the two theorems above and the type soundness of λ_{iu}, we can conclude that Uaena is type-safe.

5 Discussion and Related Work

In this section, we first discuss OCaml, the only language we know of that has well-studied support for named and optional arguments, though its mechanism does not go well with higher-order functions. Then we briefly show how named arguments are handled very differently in Scala and Racket. After that, we illustrate how named arguments can be encoded as records in Haskell while not natively supported. Finally, we discuss two more approaches we find in record calculi [19,20]. We will also explain why all these approaches have drawbacks.

5.1 OCaml

OCaml did not support named arguments originally. Nevertheless, Garrigue et al. [1,9,12] conducted research on the label-selective λ-calculus and implemented it in OLabl [10], which extends OCaml with labeled and optional arguments, among others. All features of OLabl were merged into OCaml 3, despite subtle differences [11].

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

```
let exp ?(base = Float.exp 1.0) x = base ** x
(* val exp : ?base:float → float → float *)
exp 10.0              (*= e^10. *)
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 positional parameter. Changing `x` into a second labeled parameter will trigger an `unerasable-optional-argument` warning because OCaml expects that there should be a positional parameter after all optional parameters. This expectation 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`? Both interpretations are possible, but OCaml considers it to be a full application because the trailing positional argument `x` is given. The presence of the positional argument is used to indicate that the optional arguments before it can be replaced by their default values. However, this design may confuse users since `(exp 10.0) ~base:2.0` will raise a type error but `exp 10.0 ~base:2.0` will not. Partial application does not lead to an equivalent program in such situations.

Option Types. In OCaml, an optional argument is internally represented as an `option` type, which comprises two constructors: `None` and `Some`. Here is an equivalent definition for `exp`:

```
let exp ?(base : float option) x =
  let base = match base with
             | None → Float.exp 1.0
             | Some b → b in
  base ** x
(* val exp : ?base:float → float → float *)
exp 10.0            (*> exp 10.0 ~base:None *)
exp 10.0 ~base:2.0  (*> exp 10.0 ~base:(Some 2.0) *)
```

This encoding is similar to union types, but it depends on the `option` type in the standard library. Unfortunately, this specific kind of `option` is not a built-in type in many mainstream languages, especially in those languages that do not support algebraic data types.

Higher-Order Functions. A surprising gotcha in OCaml is that the commutativity breaks down when we pass a function with labeled arguments to another function. *Real World OCaml* [15] gives the following example:

```
let apply1 f (fst,snd) = f ~fst ~snd
(* val apply1 : (fst:'a → snd:'b → 'c) → 'a * 'b → 'c *)
let apply2 f (fst,snd) = f ~snd ~fst
(* val apply2 : (snd:'a → fst:'b → 'c) → 'b * 'a → 'c *)
let divide ~fst ~snd = fst / snd
(* val divide : fst:int → snd:int → int *)
apply1 divide (48,3)  (*= 16 *)
apply2 divide (48,3)
(* TypeError: "divide" has type fst:int → snd:int → int
       but was expected of type snd:'a → fst:'b → 'c *)
```

Normally, the order of named arguments does not matter in OCaml, so it type-checks whether we call `divide ~fst ~snd` or `divide ~snd ~fst`. However, order matters when we pass `divide` to a higher-order function. That is why `apply1 divide` type-checks while `apply2 divide` does not. It turns out that the OCaml way of handling labeled arguments does not go well with other features like higher-order functions. Our approach scales better in this regard and the commutativity still holds in higher-order contexts via intersection subtyping.

In short, OCaml has a very powerful label-selective core calculus that reconciles commutativity and currying, but it is quite complicated and may hinder its integration with other language features. Another thing worth mentioning is that labeled arguments in OCaml are not first-class values, so they cannot be assigned to a variable or passed around by functions. In contrast to OCaml, our approach supports first-class named arguments and targets a minimal core calculus with intersection and union types, which is compatible with many popular languages like Python, Ruby, JavaScript, etc.

5.2 Scala

Rytz and Odersky [26] described the design of named and default arguments in Scala. Like in Python, parameter names in a method definition are non-mandatory keywords in Scala, and thus every argument can be passed with or without keywords. Furthermore, the parameter names are not part of the public interface of a method. This design is partly due to the backward compatibility with earlier versions of Scala, so the addition of named arguments will not break any existing code. As a result of the conservative treatment, named arguments are not first-class values in Scala and cannot be defined in an anonymous function. In short, named arguments are more like syntactic sugar in Scala and do not interact with the type system.

Below we show an example in Scala. In order to let the default value of `c` depend on `a` and `b`, we make the function partly curried:

```
def f(a: Int, b: Int)(c: Int = a+b) = c
f(b = 1+1, a = 1)()
```

The code will be translated to equivalent code without keywords or defaults:

```
def f(a: Int, b: Int)(c: Int) = c
def f$default$3(a: Int, b: Int): Int = a+b
{
  val x$1 = 1+1
  val x$2 = 1
  val x$3 = f$default$3(x$2, x$1)
  f(x$2, x$1)(x$3)
}
```

There are two things to note here. First, a new function `f$default$3` is generated for the default value of `c`, taking two parameters `a` and `b`. Second, the call site is translated to a series of variable assignments for each argument and a keyword-free call to `f` with arguments reordered. The whole call site is wrapped in a block to avoid polluting the namespace.

In conclusion, named arguments in Scala are handled in a very different way from OCaml and our approach. The Scala way is more syntactic than type-theoretic, so it is hard to do an apples-to-apples comparison with our approach.

5.3 Racket

Flatt and Barzilay [7] introduced keyword and optional arguments into Racket, which was known as PLT Scheme at that time. A keyword is prefixed with `#:` in syntax and is implemented as a new built-in type in Racket. Keyword arguments are supported by replacing **`define`**, **`lambda`**, and the core application form with newly defined macros that recognize keyword-argument forms. Here is an example of a function `f` with three keyword arguments `a`, `b`, and `c`, among which `c` is optional and defaults to `a+b`:

```
(define (f #:a a #:b b #:c [c (+ a b)]) c)
(f #:b (+ 1 1) #:a 1)
```

The function call with keywords seems hard to implement because it just lists the function and arguments in juxtaposition. In fact, an application form in Racket implicitly calls `#%app` in its lexical scope, so the support for keyword arguments is done by supplying an `#%app` macro. A new `keyword-apply` function is also defined to accept keyword arguments as first-class values. For example, we can rewrite the function call above as follows[1]:

```
(keyword-apply f '(#:a #:b) `(1 ,(+ 1 1)) '());OK!
(keyword-apply f '(#:b #:a) `(,(+ 1 1) 1) '());Contract violation!
```

Note that we need to separate keywords and corresponding arguments into two lists. The third list is for positional arguments, so it is empty in this case. We cannot list keywords in arbitrary order: a contract violation will be signaled unless the keywords are sorted in alphabetical order. In other words, commutativity is lost for first-class keyword arguments in Racket.

In Typed Racket [31], Racket's gradually typed sister language, `f` can be typed as `(→* (#:a Number #:b Number) (#:c Number) Number)`. The first list contains the required arguments (`#:a` and `#:b`), and the second list contains the optional ones (`#:c`). However, Typed Racket does not provide a typed version of `keyword-apply`, and it is unclear how to properly type it.

In conclusion, Racket supports keyword and optional arguments in a unique way via its powerful macro system. However, the support for first-class keyword arguments is very limited and cannot easily transfer to a type-safe setting.

5.4 Haskell

Unlike the aforementioned languages, 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* [27], various server settings are bundled in the data type `Settings`, as shown in Fig. 7a. 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`, as shown in Fig. 7b. Users can update whatever fields they want to change while keeping others. For example, we update `settingsPort` and `settingsHost` while keeping the rest unchanged in Fig. 7c. Finally, we call the library function `runSettings` with the updated `settings` to run a server.

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 a

[1] ' is `quote`, ` is `quasiquote`, and , is `unquote` in Racket.

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

(a) Record type.

```
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) Updating some settings before running a server application.

Fig. 7: Named arguments as records in Haskell.

few similar records in a library. A better solution is to change the parameter of `runSettings` from a complete `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. However, this design still has a second drawback: all arguments must have default values. Usually, we do not consider every argument to be optional. For example, we may want to require users to fill in `settingsPort`. A workaround employed by `SqlBackend` in the library *persistent* [21] is to have another function that asks for 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, .. }
```

Best Practice. Although `mkSettings` resolves the second issue, there is a regression concerning the first issue: users have to look for `mk*` functions now. Fortunately, we can harmonize 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 in Haskell, though it is already quite complicated and requires two GHC language extensions. Of course, there could be other approaches to encoding named and optional arguments in Haskell. Users could get confused about the various available design patterns. This is largely due to lack of language-level support. We believe it is better for a language to natively support named and optional arguments.

Sidenote. The design pattern of *named arguments as records* can also be found in other functional languages like Standard ML, Elm, and PureScript, just to name a few. It is worth mentioning that these languages have first-class support for record types, so no separate type declarations are needed like in Fig. 7a. However, they still suffer from lack of native support for optional arguments.

5.5 Record Calculi

Ohori discussed how to model optional arguments in the future work of his seminal paper on compiling a polymorphic record calculus [19]. He proposed to extend a record calculus with optional-field selection ($e.\ell$? d) which behaves like $e.\ell$ if ℓ is present in the record e or evaluates to d otherwise. However, his proposal is subject to a similar type-safety issue as mypy. The static type of e can easily lose track of the optional field ℓ and fail to ensure that $e.\ell$ has the same type as d at run time. Since Ohori did not explicitly mention how to type-check optional-field selection, we cannot make any firm conclusion about the type safety of his proposal.

Osinski also discussed the support for optional arguments in Section 3.5 of his dissertation on compiling record concatenation [20]. His approach is based on row polymorphism and makes use of a sort of predicate on rows: $row_1 \blacktriangleright row_2$, which means that row_1 consists of all the fields in row_2. With this predicate, a function

has type $\forall\rho.\ row_o \blacktriangleright \rho \Rightarrow \{row_r, \rho\} \to \tau$ if the required and optional arguments are denoted by row_r and row_o, respectively. Roughly speaking, it means the parameter has a type between $\{row_r\}$ and $\{row_r, row_o\}$. At the term level, he introduced a compatible concatenation operator `|&|`, which allows overlapping fields with the same types and prefers the fields on the right-hand side when overlapping occurs. An example of their translation is as follows:

```
fun add { x, y = 0 } = ...
(* is translated to *)
fun add r = let r' = { y = 0 } |&| r in
            let x = r'.x in let y = r'.y in ...
```

His approach is free from the type-safety issue, though based on a more sophisticated row-polymorphic system. There are two sorts of predicates and three variants of record concatenation operators in his calculus, for example, demonstrating some sophistication of his calculus.

It is worth noting that neither Ohori's nor Osinski's calculus supports subtyping. This is a significant limitation since subtyping is a common feature in many popular languages, especially object-oriented languages.

6 Conclusion

The benefits from named arguments are twofold. On the one hand, argument keywords serve as extra documentation at the language level. On the other hand, they lay the foundation for supporting commutativity and optionality.

Named and optional arguments are widely supported in mainstream programming languages but are hardly formalized. Our approach is inspired by existing mechanisms in OCaml and Haskell but further considers the interaction between first-class named arguments and subtyping. Static type checkers for Python and Ruby both suffer from a type-safety issue in this regard. We show that λ_{iu} can serve as a type-safe core calculus with compact support for intersection and union types.

We hope that this paper will call more attention to the foundation for named and optional arguments and inspire future language developers to consider potential type-safety issues more carefully in their designs.

Future Work. A natural extension of our work is to support omitting keywords if the order of arguments is not changed from how they are defined. In other words, this is a version without the property of distinctness as classified in Table 1. The syntactic approach employed by Scala can be a good reference. Another interesting direction is to evaluate our approach in a more practical setting, for example, by typing popular libraries like TensorFlow, where named and optional arguments are ubiquitous, and dependent default values can help simplify the functions and avoid some ill-defined parameter combinations.

Data Availability. The Coq formalization for this paper is available [30].

References

1. Aït-Kaci, H., Garrigue, J.: Label-selective λ-calculus: syntax and confluence. Theor. Comput. Sci. **151**(2) (1995). https://doi.org/10.1016/0304-3975(95)00072-5
2. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Intersection and union types: syntax and semantics. Inf. Comput. **119**(2) (1995). https://doi.org/10.1006/inco.1995.1086
3. Castagna, G.: Programming with union, intersection, and negation types. In: The French School of Programming, chap. 12 (2023). https://doi.org/10.1007/978-3-031-34518-0_12
4. Charguéraud, A.: The locally nameless representation. J. Autom. Reason. **49**(3) (2012). https://doi.org/10.1007/s10817-011-9225-2
5. Church, A.: The Calculi of Lambda-Conversion. No. 6 in Annals of Mathematics Studies, Princeton University Press (1941), https://press.princeton.edu/books/paperback/9780691083940/the-calculi-of-lambda-conversion
6. Dunfield, J.: Elaborating intersection and union types. J. Funct. Program. **24**(2–3) (2014). https://doi.org/10.1017/S0956796813000270
7. Flatt, M., Barzilay, E.: Keyword and optional arguments in PLT Scheme. In: Scheme (2009), https://users.cs.utah.edu/plt/publications/scheme09-fb.pdf
8. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**(4) (2008). https://doi.org/10.1145/1391289.1391293
9. Furuse, J.P., Garrigue, J.: A label-selective lambda-calculus with optional arguments and its compilation method. Tech. rep., Kyoto University (1995), https://www.math.nagoya-u.ac.jp/~garrigue/papers/rims-1041.pdf
10. Garrigue, J.: Objective Label trilogy, http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/
11. Garrigue, J.: Labeled and optional arguments for Objective Caml. In: JSSST SIGPPL (2001), https://www.math.nagoya-u.ac.jp/~garrigue/papers/ppl2001.pdf
12. Garrigue, J., Aït-Kaci, H.: The typed polymorphic label-selective lambda-calculus. In: POPL (1994). https://doi.org/10.1145/174675.174434
13. Lehtosalo, J., et al.: Mypy: Optional static typing for Python, https://www.mypy-lang.org
14. Luo, Z., Soloviev, S., Xue, T.: Coercive subtyping: Theory and implementation. Inf. Comput. **223** (2013). https://doi.org/10.1016/j.ic.2012.10.020
15. Madhavapeddy, A., Minsky, Y.: Real World OCaml: Functional Programming for the Masses. Cambridge University Press (2022). https://doi.org/10.1017/9781009129220
16. Matsumoto, S., et al.: Steep: Gradual typing for Ruby, https://github.com/soutaro/steep
17. Microsoft: TypeScript: JavaScript with syntax for types, https://www.typescriptlang.org
18. Nieto, A., Zhao, Y., Lhoták, O., Chang, A., Pu, J.: Scala with explicit nulls. In: ECOOP (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.25
19. Ohori, A.: A polymorphic record calculus and its compilation. ACM Trans. Program. Lang. Syst. **17**(6) (1995). https://doi.org/10.1145/218570.218572

20. Osinski, E.: A polymorphic type system and compilation scheme for record concatenation. Ph.D. thesis, New York University (2006), https://cs.nyu.edu/media/publications/osinski_edward.pdf
21. Parsons, M., et al.: Persistent: Type-safe, multi-backend data serialization, https://hackage.haskell.org/package/persistent
22. Pierce, B.C., et al.: Programming language foundations. In: Software Foundations. https://softwarefoundations.cis.upenn.edu/plf-current/
23. Rehman, B.: A blend of intersection types and union types. Ph.D. thesis, The University of Hong Kong (2023), https://hub.hku.hk/handle/10722/335090
24. Reynolds, J.C.: Design of the programming language Forsythe. In: Algol-like Languages, vol. 1, chap. 8 (1997). https://doi.org/10.1007/978-1-4612-4118-8_9
25. Roblox: Luau: A fast, small, safe, gradually typed embeddable scripting language derived from Lua, https://luau.org
26. Rytz, L., Odersky, M.: Named and default arguments for polymorphic object-oriented languages: A discussion on the design implemented in the Scala language. In: SAC (2010). https://doi.org/10.1145/1774088.1774529
27. Snoyman, M., et al.: Warp: A fast, light-weight web server for WAI applications, https://hackage.haskell.org/package/warp
28. Stripe: Sorbet: A static type checker for Ruby, https://sorbet.org
29. Sun, Y., Dhandhania, U., Oliveira, B.C.d.S.: Compositional embeddings of domain-specific languages. In: OOPSLA (2022). https://doi.org/10.1145/3563294
30. Sun, Y., Oliveira, B.C.d.S.: Named arguments as intersections, optional arguments as unions (artifact). Zenodo (2025). https://doi.org/10.5281/zenodo.14697184
31. Tobin-Hochstadt, S., St-Amour, V., Dobson, E., Takikawa, A.: The Typed Racket Guide, https://docs.racket-lang.org/ts-guide/
32. van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., Fisker, R.G.: Revised report on the algorithmic language ALGOL 68. Acta Informatica **5**(1–3) (1975). https://doi.org/10.1007/BF00265077
33. Zhang, W., Sun, Y., Oliveira, B.C.d.S.: Compositional programming. ACM Trans. Program. Lang. Syst. **43**(3) (2021). https://doi.org/10.1145/3460228

A Type-Safety Issue in Ruby with Steep

```ruby
# rbs_inline: enabled

class App
  # @rbs host: String
  # @rbs port: Integer
  # @rbs debug: bool
  def run(host:, port:, debug: false)
    if debug != true && debug != false
      raise "Argument debug is not Boolean!"
    end
  end
end
app = App.new

# @type var args0: { host: String, port: Integer, debug: bool }
args0 = { host: "0.0.0.0", port: 80, debug: true }
app.run(**args0) # OK!

# @type var args1: { host: String, port: Integer }
args1 = { host: "0.0.0.0", port: 80 }
app.run(**args1) # OK!

# @type var args2: { host: String, port: Integer, debug: String }
args2 = { host: "0.0.0.0", port: 80, debug: "Oops!" }
# app.run(**args2) # TypeError: ArgumentTypeMismatch!

class App
  # @rbs args: { host: String, port: Integer, debug: String }
  # @rbs return: { host: String, port: Integer }
  def f(args) = args
end

# @type var args3: { host: String, port: Integer }
args3 = app.f(args2)
app.run(**args3) # Type-checks in Steep, but has a runtime error:
                 # Argument debug is not Boolean!
```

B Type-Safety Issue in Ruby with Sorbet

```ruby
# typed: true
require "sorbet-runtime"

class App
  extend T::Sig

  sig {params(host: String, port: Integer, debug: T::Boolean).void}
  def run(host:, port:, debug: false)
    if debug != true && debug != false
      raise "Argument debug is not Boolean!"
    end
  end
end
app = App.new

args0 = { host: "0.0.0.0", port: 80, debug: true }
app.run(**args0) # OK!

args1 = { host: "0.0.0.0", port: 80 }
app.run(**args1) # OK!

args2 = { host: "0.0.0.0", port: 80, debug: "Oops!" }
# app.run(**args2) # TypeError: Expected T::Boolean
                   # but found String("Oops!") for argument debug!

class App
  sig do
    params(args: { host: String, port: Integer, debug: String })
    .returns({ host: String, port: Integer })
  end
  def f(args) = args
end

args3 = app.f(args2)
app.run(**args3)
# This call passes Sorbet's static type checking,
# but the Sorbet runtime raises a dynamic type error in App#f:
# Return value expected type {host: String, port: Integer},
# but got type {host: String, port: Integer, debug: String}!
```

C Typing of let-in bindings in λ_{iu}

$\boxed{\Gamma \vdash \mathit{letin} \dashv \Gamma'}$ *(Let-in binding)*

$$\textsc{LB-Let}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathbf{let}\, x = e\, \mathbf{in} \dashv \Gamma, x : A} \qquad \textsc{LB-Comp}\ \frac{\Gamma \vdash \mathit{letin}_1 \dashv \Gamma' \quad \Gamma' \vdash \mathit{letin}_2 \dashv \Gamma''}{\Gamma \vdash \mathit{letin}_1 \circ \mathit{letin}_2 \dashv \Gamma''} \qquad \textsc{LB-Id}\ \frac{}{\Gamma \vdash \mathbf{id} \dashv \Gamma}$$

D Dynamic Semantics of λ_{iu}

Values	$v ::= \{\} \mid \mathbf{null} \mid n \mid \lambda x\!:\!A.\ e\!:\!B \mid \{\ell : A = v\} \mid v_1 \,,\, v_2$
Evaluation contexts	$E ::= [\cdot] \mid E\ e \mid v\ E \mid \{\ell : A = E\} \mid E.\ell \mid E \,,\, e \mid v \,,\, E$
	$\mid \mathbf{switch}\ E\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 \mid E : A$

$\boxed{e \longrightarrow e'}$ *(Small-step operational semantics)*

$$\textsc{Step-App}\ \frac{v \longrightarrow_A v'}{(\lambda x\!:\!A.\ e\!:\!B)\ v \longrightarrow ([v'/x]\ e) : B} \qquad \textsc{Step-Prj}\ \frac{v \longrightarrow_A v'}{\{\ell : A = v\}.\ell \longrightarrow v'}$$

$$\textsc{Step-AppDispatch}\ \frac{v_1 \,,\, v_2 \bullet v \longrightarrow e'}{(v_1 \,,\, v_2)\ v \longrightarrow e'} \qquad \textsc{Step-PrjDispatch}\ \frac{v_1 \,,\, v_2 \bullet \ell \longrightarrow e'}{(v_1 \,,\, v_2).\ell \longrightarrow e'}$$

$$\textsc{Step-SwitchL}\ \frac{v \longrightarrow_A v'}{\mathbf{switch}\ v\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 \longrightarrow [v'/x]\ e_1} \qquad \textsc{Step-Anno}\ \frac{v \longrightarrow_A v'}{v : A \longrightarrow v'}$$

$$\textsc{Step-SwitchR}\ \frac{v \longrightarrow_B v'}{\mathbf{switch}\ v\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 \longrightarrow [v'/x]\ e_2} \qquad \textsc{Step-Ctx}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

Ordinary types $A^\circ, B^\circ ::= \mathbf{Null} \mid \mathbb{Z} \mid A \to B \mid \{\ell : A\}$

$\boxed{v \longrightarrow_A v'}$ *(Type casting)*

CAST-TOP
$$\frac{}{v \longrightarrow_\top \{\}}$$

CAST-NULL
$$\frac{}{\mathbf{null} \longrightarrow_{\mathbf{Null}} \mathbf{null}}$$

CAST-INT
$$\frac{}{n \longrightarrow_\mathbb{Z} n}$$

CAST-ARROW
$$\frac{A_1 \to B_1 <: A_2 \to B_2}{\lambda x{:}A_1.\ e{:}B_1 \longrightarrow_{A_2 \to B_2} \lambda x{:}A_1.\ e{:}B_2}$$

CAST-RCD
$$\frac{\{\ell : A\} <: \{\ell : B\}}{\{\ell : A = v\} \longrightarrow_{\{\ell:B\}} \{\ell : B = v\}}$$

CAST-MERGE
$$\frac{v \longrightarrow_A v_1 \quad v \longrightarrow_B v_2}{v \longrightarrow_{A \wedge B} v_1 \,,\, v_2}$$

CAST-MERGEL
$$\frac{v_1 \longrightarrow_{A^\circ} v_1'}{v_1 \,,\, v_2 \longrightarrow_{A^\circ} v_1'}$$

CAST-MERGER
$$\frac{v_2 \longrightarrow_{B^\circ} v_2'}{v_1 \,,\, v_2 \longrightarrow_{B^\circ} v_2'}$$

CAST-ORL
$$\frac{v \longrightarrow_A v'}{v \longrightarrow_{A \vee B} v'}$$

CAST-ORR
$$\frac{v \longrightarrow_B v'}{v \longrightarrow_{A \vee B} v'}$$

$\boxed{v_1 \,,\, v_2 \bullet v \longrightarrow e}$ *(Applicative dispatch)*

AD-LEFT
$$\frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^\lambda \quad \neg(\lfloor v \rfloor <: \lfloor v_2 \rfloor^\lambda)}{v_1 \,,\, v_2 \bullet v \longrightarrow v_1\ v}$$

AD-RIGHT
$$\frac{\lfloor v \rfloor <: \lfloor v_2 \rfloor^\lambda \quad \neg(\lfloor v \rfloor <: \lfloor v_1 \rfloor^\lambda)}{v_1 \,,\, v_2 \bullet v \longrightarrow v_2\ v}$$

AD-BOTH
$$\frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^\lambda \quad \lfloor v \rfloor <: \lfloor v_2 \rfloor^\lambda}{v_1 \,,\, v_2 \bullet v \longrightarrow v_1\ v \,,\, v_2\ v}$$

$\boxed{v_1 \,,\, v_2 \bullet \ell \longrightarrow e}$ *(Projective dispatch)*

PD-LEFT
$$\frac{\lfloor v_1 \rfloor <: \{\ell : \top\} \quad \neg(\lfloor v_2 \rfloor <: \{\ell : \top\})}{v_1 \,,\, v_2 \bullet \ell \longrightarrow v_1.\ell}$$

PD-RIGHT
$$\frac{\lfloor v_2 \rfloor <: \{\ell : \top\} \quad \neg(\lfloor v_1 \rfloor <: \{\ell : \top\})}{v_1 \,,\, v_2 \bullet \ell \longrightarrow v_2.\ell}$$

PD-BOTH
$$\frac{\lfloor v_1 \rfloor <: \{\ell : \top\} \quad \lfloor v_2 \rfloor <: \{\ell : \top\}}{v_1 \,,\, v_2 \bullet \ell \longrightarrow v_1.\ell \,,\, v_2.\ell}$$

$\boxed{\lfloor v \rfloor}$ Dynamic type

$\lfloor \{\} \rfloor \equiv \top \qquad \lfloor \mathbf{null} \rfloor \equiv \mathbf{Null} \qquad \lfloor n \rfloor \equiv \mathbb{Z} \qquad \lfloor \lambda x{:}A.\ e{:}B \rfloor \equiv A \to B$

$\lfloor \{\ell : A = v\} \rfloor \equiv \{\ell : A\} \qquad \lfloor v_1 \,,\, v_2 \rfloor \equiv \lfloor v_1 \rfloor \wedge \lfloor v_2 \rfloor$

$\boxed{\lfloor v \rfloor^\lambda}$ Input type

$\lfloor \lambda x{:}A.\ e{:}B \rfloor^\lambda \equiv A \qquad \lfloor v_1 \,,\, v_2 \rfloor^\lambda \equiv \lfloor v_1 \rfloor^\lambda \vee \lfloor v_2 \rfloor^\lambda \qquad \lfloor \ldots \rfloor^\lambda \equiv \bot$

E Label Lookup in Uaena Argument Types

$\boxed{\mathcal{K} :: \ell \Rightarrow \mathcal{A}}$ *(Successful lookup)*

$$\frac{\mathcal{K} :: \ell \nRightarrow}{(\mathcal{K};\, \ell : \mathcal{A}) :: \ell \Rightarrow \mathcal{A}}\ \text{LU-Present} \qquad \frac{\ell' \neq \ell \qquad \mathcal{K} :: \ell \Rightarrow \mathcal{A}}{(\mathcal{K};\, \ell' : \mathcal{B}) :: \ell \Rightarrow \mathcal{A}}\ \text{LU-Absent}$$

$\boxed{\mathcal{K} :: \ell \nRightarrow}$ *(Failed lookup)*

$$\frac{}{\cdot :: \ell \nRightarrow}\ \text{LD-Empty} \qquad \frac{\ell' \neq \ell \qquad \mathcal{K} :: \ell \nRightarrow}{(\mathcal{K};\, \ell' : \mathcal{A}) :: \ell \nRightarrow}\ \text{LD-Absent}$$

F Type Translation from Uaena to λ_{iu}

$\boxed{|\mathcal{A}|}$ Type translation

$$|\mathbb{Z}| \equiv \mathbb{Z} \qquad |(\mathcal{A}) \to \mathcal{B}| \equiv |\mathcal{A}| \to |\mathcal{B}| \qquad |\{\mathcal{P}\} \to \mathcal{B}| \equiv |\mathcal{P}| \to |\mathcal{B}| \qquad |\{\mathcal{K}\}| \equiv |\mathcal{K}|$$

$\boxed{|\mathcal{P}|}$ Parameter type translation

$$|\cdot| \equiv \top \qquad |\mathcal{P};\, \ell : \mathcal{A}| \equiv |\mathcal{P}| \wedge \{\ell : |\mathcal{A}|\} \qquad |\mathcal{P};\, \ell? : \mathcal{A}| \equiv |\mathcal{P}| \wedge \{\ell : |\mathcal{A}| \vee \mathbf{Null}\}$$

$\boxed{|\mathcal{K}|}$ Argument type translation

$$|\cdot| \equiv \top \qquad |\mathcal{K};\, \ell : \mathcal{A}| \equiv |\mathcal{K}| \wedge \{\ell : |\mathcal{A}|\}$$

$\boxed{|\Delta|}$ Typing context translation

$$|\cdot| \equiv \cdot \qquad |\Delta,\, x : \mathcal{A}| \equiv |\Delta|,\, x : |\mathcal{A}|$$