Signature Restriction for Polymorphic Algebraic Effects

TARO SEKIYAMA, National Institute of Informatics & SOKENDAI, Japan
TAKESHI TSUKADA, The University of Tokyo, Japan
ATSUSHI IGARASHI, Kyoto University, Japan

The naive combination of polymorphic effects and polymorphic type assignment has been well known to break type safety. Existing approaches to this problem are classified into two groups: one for restricting how effects are triggered and the other for restricting how they are implemented. This work explores a new approach to ensuring the safety of polymorphic effects in polymorphic type assignment. A novelty of our work lies in finding a restriction on *effect interfaces*. To formalize our idea, we employ algebraic effects and handlers, where an effect interface is given by a set of operations coupled with type signatures. We propose *signature restriction*, a new notion to restrict the type signatures of operations, and show that signature restriction is sufficient to ensure type safety of an effectful language equipped with unrestricted polymorphic type assignment. We also develop a type-and-effect system to enable the use of both operations that satisfy and do not satisfy the signature restriction in a single program.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Polymorphism**; **Control structures**; **Formal language definitions**.

Additional Key Words and Phrases: polymorphic type assignment, polymorphic effects, algebraic effects and handlers

ACM Reference Format:
Taro Sekiyama, Takeshi Tsukada, and Atsushi Igarashi. 2020. Signature Restriction for Polymorphic Algebraic Effects. *Proc. ACM Program. Lang.* 4, ICFP, Article 117 (August 2020), 30 pages. https://doi.org/10.1145/3408999

1 INTRODUCTION

1.1 Background: Polymorphic Type Assignment with Computational Effects

Pervasive in programming are computational effects, such as mutable memory cells, backtracking, exception handling, concurrency/parallelism, and I/O processing for terminals, files, networks, etc. These effects have a variety of roles: I/O processing enables interaction with external environments; memory manipulation and concurrency/parallelism make software efficient; and backtracking and exception provide reusable, general operations that make it unnecessary to write boilerplate code. These effects have also been proven convenient in functional programming [Gordon et al. 1979; Peyton Jones and Wadler 1993; Wadler 1992].

In return for convenience, however, computational effects can introduce weird, counterintuitive behavior into programs and complicate program reasoning and verification. For example, incorporating effects into dependent type theory could easily lead to inconsistency [Pédrot and Tabareau 2020]. This fact encourages dependent type systems to separate term-level computation from types [Ahman 2017; Casinghino et al. 2014; Cong and Asai 2018; Sekiyama and Igarashi

Authors' addresses: Taro Sekiyama, National Institute of Informatics & SOKENDAI, Japan, tsekiyama@acm.org; Takeshi Tsukada, The University of Tokyo, Japan, tsukada@kb.is.s.u-tokyo.ac.jp; Atsushi Igarashi, Kyoto University, Japan, igarashi@kuis.kyoto-u.ac.jp.

2475-1421/2020/8-ART117
https://doi.org/10.1145/3408999

2017; Swamy et al. 2016; Xi 2007]. For program reasoning, the state transition caused by effectful computations has to be tracked [Ahmed et al. 2009; Dreyer et al. 2010; Pitts and Stark 1998].

These kinds of gaps between pure and effectful computations are also found in our target, i.e., polymorphic type assignment: although any pure expressions can safely be assigned polymorphic types [Leivant 1983], unrestricted polymorphic type assignment to effectful expressions may break type safety. This problem with polymorphic type assignment has been discovered in call-by-value languages with *polymorphic effects*, which are effects caused by polymorphic operations. For example, ML-style references are an instance of polymorphic effects because the operations for memory cell creation, assignment, and dereference are polymorphic [Leroy et al. 2020; Milner et al. 1990]. Gordon et al. [1979] showed that the ML-style references cannot cooperate safely with unrestricted polymorphic type assignment owing to the polymorphism of the operations. Another example is control effects, which are triggered by control operators such as call/cc [Clinger et al. 1985] and shift/reset [Danvy and Filinski 1990]. These operators can be assigned polymorphic types but the polymorphic control operators may cause unsafe behavior in unrestricted polymorphic type assignment [Harper and Lillibridge 1993b]. This fault even occurs in let-polymorphic type assignment [Milner 1978] where quantifiers only appear at the outermost positions.

Many approaches to the safe use of polymorphic effects in polymorphic type assignment have been proposed [Appel and MacQueen 1991; Asai and Kameyama 2007; Garrigue 2004; Hoang et al. 1993; Kammar and Pretnar 2017; Leroy and Weis 1991; Sekiyama and Igarashi 2019; Tofte 1990; Wright 1995]. These approaches are classified into two groups. The first group—to which most of the approaches belong—aims at restricting *how effects are triggered*. For example, the value restriction [Tofte 1990] restricts polymorphic expressions to be only values in order to prevent polymorphic expressions from triggering effects. The other group aims at restricting *how effects are implemented*. For example, Sekiyama and Igarashi [2019] proposed a type system that accepts only effects that are *safe*, i.e., that do not cause programs to get stuck no matter how they are used.

1.2 Our Work

This work explores a new approach to safe polymorphic type assignment for effectful call-by-value languages. A novelty of our approach lies in restriction on *effect interfaces*. In this work, the effect interfaces are represented by sets of *operations* coupled with *type signatures*. For example, an interface for exceptions consists of a single operation `raise` to raise an exception and its type signature $\forall \alpha. \mathsf{unit} \hookrightarrow \alpha$, which means that `raise` takes the unit value as an argument and returns a value of any type α if the control gets back to the caller at all. Quantification in the signature not only provides the clients of the operation with flexibility—they can instantiate α with any desired type and put a call of `raise` in any context—but also constrains its servers in that implementations of the operation have to abstract over types. In fact, the type signature of `raise` is sufficiently restrictive to guarantee that the exception effect is safe. Generalizing this idea, we provide a criterion to decide if an effect is safe. Our criterion is *simple* in that it only mentions the occurrences of bound type variables α in a type signature, *robust* in that it is independent of how effects are implemented, and *permissive* in that it is met by many safe effects—including exception, nondeterminism, and input streaming. We call the restriction on type signatures to meet the criterion *signature restriction*.

We formalize our idea with algebraic effects and handlers [Plotkin and Pretnar 2009, 2013], which are a programming mechanism to accommodate user-defined control effects in a modular way. Algebraic effects and handlers split an effect into an interface (i.e., a set of operations with type signatures) and an interpretation, so we can incorporate signature restriction into them naturally.

We provide two polymorphic type assignment systems for a λ-calculus equipped with algebraic effects and handlers. The first is a simple polymorphic type system based on Curry-style System F [Leivant 1983] (i.e., it supports implicit, full polymorphism). This type system allows arbitrary terms (rather than only values) that invoke effects to be given polymorphic types but is sound, thanks to signature restriction. The minimality of this simple type system reveals the essence of signature restriction. The second type assignment system is a polymorphic type-and-effect system. Using this system, we show that effect tracking is key to apply signature restriction for programs in which both safe and potentially unsafe polymorphic effects may happen.[1]

The contributions of our work are summarized as follows.

- We define a λ-calculus λ_{eff} with algebraic effects and handlers and provide a type system that supports implicit full polymorphism and allows any effectful expression to be polymorphic. We formalize signature restriction for λ_{eff} and prove soundness of the type system under the assumption that all operations satisfy signature restriction.
- As a technical development to justify signature restriction, we equip the type system with Mitchell's type containment [Mitchell 1988], which is an extension of type instantiation. In the literature [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007], the proof of type soundness of a calculus equipped with type containment rests on translation to another calculus, such as System F [Girard 1972; Reynolds 1974].[2] By contrast, we show soundness of our type system *directly*, i.e., without translation to any other calculus. As far as we know, this is the first work that achieves it.
- We extend λ_{eff} and its type system with standard programming features such as products, sums, and lists to demonstrate the generality and extensibility of signature restriction.
- We develop an effect system for λ_{eff}, which enables a single program to use both safe and potentially unsafe polymorphic effects. In this effect system, an expression can be polymorphic if all the effect operations performed by the expression satisfy signature restriction. It also indicates that signature restriction can cooperate with value restriction naturally.

We employ implicit full polymorphism and type containment to show type soundness, but either of them makes even type checking undecidable [Tiuryn and Urzyczyn 1996; Wells 1994]. It is thus desirable to identify a subset of our system where type checking—and type inference as well hopefully—is decidable. To prove the feasibility of this idea, we implement an interpreter for a subset of the extended λ_{eff} in which polymorphism is restricted to let-polymorphism [Damas and Milner 1982; Milner 1978] (the effect system is not supported either). This restriction on polymorphism ensures that both type checking and type inference are decidable but it is still expressive so that all of the motivating well-typed examples in this paper (except for those in Section 6, which rest on the effect system) are typechecked. The implementation is provided as the supplementary material; alternatively, it can also be found at: https://github.com/skymountain/MLSR .

Finally, we briefly relate our work with the *relaxed* value restriction [Garrigue 2004] here. It is similar to our signature restriction in that both utilize the occurrences of type variables to ensure soundness of polymorphic type assignment in the permissive use of polymorphic effects. Indeed, a *strong* version of signature restriction can be justified similarly to the relaxed value restriction. The strong signature restriction is, however, too restrictive and rejects many useful, safe effects. We generalize it to what we call signature restriction and prove its correctness with different techniques such as type containment. Readers are referred to Section 7.1 for further details.

Organization. The remainder of this paper is organized as follows. We start with an overview of this work (Section 2) and then define our base calculus λ_{eff} (Section 3). Section 4 introduces a

[1]As we will show in the paper, signature restriction is permissive and actually we find no useful effect that invalidates it. However, the universal enforcement of signature restriction *may* give rise to inconvenience in some cases, and we consider the capability of avoiding such (potential) inconvenience important in designing a general-purpose programming language.
[2]The translation inserts, as a replacement for type containment, functions that are computationally meaningless but work as type conversion statically.

polymorphic type system for λ_{eff}, formalizes signature restriction, and shows soundness of the polymorphic type system under the assumption that all operations satisfy signature restriction. Section 5 extends λ_{eff}, the polymorphic type system, and signature restriction with products, sums, and lists. Section 6 presents an effect system to allow programs to use both safe and unsafe effects. We finally discuss related work in Section 7 and conclude in Section 8.

In this paper, we may omit the formal definitions of some well-known notions and the statements and proofs of auxiliary lemmas for type soundness. The full definitions, the full statements, and the full proofs are provided in the supplementary material.

2 OVERVIEW

This section presents an overview of our work. After reviewing algebraic effects and handlers, their extension to polymorphic effects, and why a naive extension results in unsoundness, we describe our approach of signature restriction and informally discuss why it resolves the unsoundness problem. All program examples in this paper follow ML-like syntax.

2.1 Review: Algebraic Effects and Handlers

Algebraic effects and handlers [Plotkin and Pretnar 2009, 2013] are a mechanism that enables users to define their own effects. They are successfully able to separate the syntax and semantics of effects. The syntax of an effect is given by a set of *operations*, which are used to trigger the effect. For example, exception is triggered by the operation `raise` and store manipulation is triggered by `put` and `get`, which are used to write to and read from a store, respectively. The semantics is given by *handlers*, which decide how to interpret operations performed by effectful computation.

Our running example is nondeterministic computation which enumerates all of the possible outcomes [Plotkin and Pretnar 2009, 2013]. This computation utilizes two operations: `select`, which chooses an element from a given list, and `fail`, which signals that the current control flow is undesired and the computation should abort.[3]

```
effect select : int list ↪ int
effect fail   : unit ↪ unit

let filter (l : int list) (f : int → bool) =
  handle
    let x = #select(l) in
    let _ = if f x then () else #fail() in x
  with
    return z → [z]
    select l → concat (map l (λy. resume y))
    fail z   → []

filter [3; 5; 10] (λx. x mod 2 = 1)    (* will evaluate to [3; 5] *)
```

The first two lines declare the operations `select` and `fail`, which have the type signatures `int list ↪ int` and `unit ↪ unit`, respectively. A type signature $A \hookrightarrow B$ of an operation signifies that the operation is called with an argument of type A and, when the control gets back to the caller, it receives a value of B. We refer to A and B as the *domain type* and *codomain type*, respectively.[4]

[3] This describes only *intended* semantics; one can also give an *unintended* handler, e.g., one that always returns an integer 42 for a call of `select`. Certain unintended handlers can be excluded in a polymorphic setting, as is shown in Section 2.2.
[4] The domain and codomain types are also called the *parameter type* and the *arity*, respectively [Plotkin and Pretnar 2009].

The function `filter` in Lines 4–11 operates `select` and `fail` to filter out the elements of `l` that do not meet a given predicate `f`. Now, let's take a closer look at the body of the function, which consists of a single **`handle`–`with`** expression of the form **`handle`** M **`with`** H. This expression installs a *handler* H during the evaluation of M, which we refer to as the *handled expression*.

The handled expression (Lines 6–7) chooses an integer selected from `l` by calling `select`, tests whether the selected integer `x` satisfies `f`, and returns `x` if `f x` is true; otherwise, it aborts the computation by calling `fail`. We write #op(M) to call operation op with argument M. We now explain the handler in Lines 9–11, which collects all the values in `l` that satisfy `f` as a list, along with an intuitive meaning of **`handle`–`with`** expressions.

The handler H in **`handle`** M **`with`** H consists of a single *return clause* and zero or more *operation clauses*. The return clause takes the form **`return`** `x` $\rightarrow$ M and computes the entire result M of the **`handle`–`with`** expression using the value of the handled expression, which M refers to by `x`. For example, the return clause in this example is **`return`** `z` $\rightarrow$ `[z]`. Because `z` will be bound to the result of the handled expression `x`, the entire result is the singleton list consisting of `x`. An operation clause of the form `op x` $\rightarrow$ M for an operation `op` decides how to interpret the operation `op` called by the handled expression. Variable `x` will be bound to the argument of the call of `op` and M is the entire result of the **`handle`–`with`** expression. For example, the operation clause `fail z` $\rightarrow$ `[]` means that, if `fail` is called, the computation is aborted—similarly to exception handling—and the entire **`handle`–`with`** expression returns the empty list, meaning there is no result that satisfies f.

Unlike exception handling, which discards the continuation of where an exception is raised, however, handlers can *resume* computation from the point at which the operation was called. The ability to resume a computation suspended by the operation call provides algebraic effects and handlers with the expressive power to implement control effects [Bauer and Pretnar 2015; Forster et al. 2019; Leijen 2017]. In our language, we use the expression **`resume`** M to resume the computation of the handled expression with the value of M.

The operation clause for `select` enumerates all the possible outcomes by using **`resume`**. The clause first returns, for each integer `y` of a given list `l`, the integer `y` to the caller of `select` by resuming the computation from the point at which `select` was called. The handled expression in the example calls `select` only once, so each resumed computation (which is performed under the same handler) returns either a singleton list or the empty list (by calling `fail`). The next step after the completion of all the resumed computations is to concatenate their results. The two steps are expressed by `concat (map l (λy. resume y))`.

More formally, the suspended computation is expressed as a *delimited continuation* [Danvy and Filinski 1990; Felleisen 1988], and **`resume`** simply invokes it. For example, let us consider evaluating `filter [3; 5; 10] (λx. x mod 2 = 1)` in the last line. This reduces to the following expression:

```
handle
  let x = #select([3; 5; 10]) in
  let _ = if (λx. x mod 2 = 1) x then () else #fail() in x
with H
```

where H denotes the same handler as that in the example. At the call of `select`, the run-time system constructs the following delimited continuation c

$c \overset{\text{def}}{=}$
```
handle
  let x = [] in
  let _ = if (λx. x mod 2 = 1) x then () else #fail() in x
with H
```

(where [] is the hole to be filled with resumption arguments), and then evaluates the operation clause for `select`. The resumption expression in the operation clause invokes the delimited continuation c after filling the hole with an integer in list `[3; 5; 10]`. For the case of filling the hole with `3`, the remaining computation c[`3`] to resume is:

```
handle
  let x = 3 in
  let _ = if (λx. x mod 2 = 1) x then () else #fail() in x
with H .
```

Because `3` is an odd number, it satisfies the predicate `(λx. x mod 2 = 1)`, and therefore the final result of this computation is the singleton list `[3]`. The case of `5` behaves similarly and produces `[5]`. In the case of `10`, because the even number `10` does not meet the given predicate, the remaining computation c[`10`] would call `fail` and, from the operation clause for `fail`, the final result of c[`10`] would be the empty list. The operation clause for `select` concatenates all of these resulting lists of the resumptions and finally returns `[3; 5]`. This is the behavior that we expect of `filter` exactly.

The handler in the example works even when `select` is called more than once, e.g.:

```
handle
  let x = #select([2; 3]) in
  let y = #select([10; 20]) in
  let z = x * y in
  let _ = if z > 50 then #fail() else () in z
with H .
```

This program returns a list of the values of the handled expression that are computed with $(\mathtt{x}, \mathtt{y}) \in \{2, 3\} \times \{10, 20\}$ such that the multiplication `x * y` does not exceed `50`.

Typechecking. We also review the procedure to typecheck an operation clause `op x` $\rightarrow M$ for `op` of type signature $A \hookrightarrow B$. Since the operation `op` is called with an argument of A, the typechecking assigns argument variable `x` type A. As the value of M is the result of the entire **handle–with** expression, the typechecking checks M to have the same type as the other clauses including the return clause. The typechecking of resumption expressions **resume** M' is performed as follows. Since the value of M' will be used as a result of calling `op` in a handled expression, M' has to be of the type B, the codomain type of the type signature of `op`. On the other hand, since the resumption expression returns the evaluation result of the entire **handle–with** expression, the typechecking assumes it to have the same type as all of the clauses in the handler.

For example, let us consider the typechecking of the operation clause for `select` in the function `filter`. Since the type signature of `select` is `int list` $\hookrightarrow$ `int`, the variable `l` is assigned the type `int list`. Here, we suppose `map` and `concat` to have the following types:

```
   map    :  int list → (int → int list) → int list list
 concat   :  int list list → int list
```

(these types can be inferred automatically). The type of `map` requires that the arguments `l` and `λy.`**`resume`**` y` have the types `int list` and `int → int list`, respectively, and they *do* indeed. The requirement to `l` is met by the type assigned to `l`. We can derive that `λy.`**`resume`**` y` has type `int → int list` as follows: first, the typechecking assigns the bound variable `y` type `int` and checks **`resume`**` y` to have `int list`. An argument of a resumption expression has to be of the type `int`, which is the codomain type of the type signature, and `y` has that type indeed. Then, the typechecking assumes that **`resume`**` y` has the same type as the clauses of the handler, which is the type `int list`. Thus, `λy.`**`resume`**` y` has the desired type.

2.2 Polymorphic Effects

Polymorphic effects are a particular kind of effects that incorporate polymorphism,[5] providing a set of operations with *polymorphic* type signatures. We also call such operations *polymorphic*.

For example, we can assign select and fail polymorphic signatures and write the program as follows:

```
effect select : ∀α. α list ↪ α
effect fail   : ∀α. unit ↪ α

handle
  let b = #select([true; false])
  let x = if b then #select([2; 3]) else #select([20; 30]) in
  if x > 20 then #fail() else x
with
  return z → [z]
  select l → concat (map l (λy. resume y))
  fail z   → []
```

This program evaluates to the list [2; 3; 20] (30 is filtered out by the call of fail).

Polymorphic type signatures enable operation calls with arguments of different types. For example, #select([2; 3]) and #select([true; false]) are legal operation calls that instantiate the bound type variable α of the type signature with int and bool, respectively. The calls of the same operation are handled by the same operation clause, even if the calls involve different type instantiations. It is also interesting to see that the use of polymorphic type signatures makes programs more natural and succinct: Thanks to its polymorphic codomain type, a call to fail can be put anywhere, making it possible to put x in the else-branch, unlike the monomorphic case.

Another benefit of polymorphic type signatures is that they contribute to the exclusion of undesired operation implementations. For example, the polymorphic signature of fail ensures that, once we call fail, the control *never* gets back and that of select ensures that no other values than elements in an argument list are chosen. Parametricity [Reynolds 1983] enables formal reasoning for this; readers are referred to Biernacki et al. [2020] for parametricity with the support for polymorphic algebraic effects and handlers.

2.3 (Naive) Polymorphic Typechecking and Its Unsoundness

(Naive) typechecking of operation clauses for polymorphic operations is obtained by extending the monomorphic setting. The only difference is that the operation clauses have to abstract over types. Namely, an operation clause op x $\rightarrow$ M for op of polymorphic type signature $\forall\alpha. A \hookrightarrow B$ is typechecked as follows. The typechecking process allocates a fresh type variable α, which is bound in M, and assigns variable x type A (which may refer to the bound type variable α). Resumption expressions **resume** M' in M are typechecked as in the monomorphic setting; that is, the typechecking checks M' to be of B (which may refer to α) and assumes the resumption expressions to have the same type as the clauses in the handler. Finally, the typechecking checks whether M is of the same type as the other clauses in the handler. It is easy to see that the polymorphic version of the select and fail example typechecks.

[5]Another way to incorporate polymorphism is *parameterized* effects, where the declaration of an operation is parameterized over types [Wadler 1992].

However, this naive extension is unsound. In what follows, we revisit the counterexample given by Sekiyama and Igarashi [2019], which is an analogue to that found by Harper and Lillibridge [1991, 1993b] with call/cc [Clinger et al. 1985].

```
effect get_id : ∀α. unit ↪ (α → α)

handle
  let f = #get_id() in (* f : ∀α.α → α *)
  if (f true) then ((f 0) + 1) else 2
with
  return x → x
  get_id x → resume (λz1. let _ = resume (λz2. z1) in z1)
```

We first check that this program is well typed. The handled expression first binds the variable f to the result returned by get_id. In polymorphic type assignment, we can assign a polymorphic type $\forall \alpha. \alpha \rightarrow \alpha$ to f by allocating a fresh type variable α, instantiating the type signature of get_id with α, and generalizing α finally. The polymorphic type of f allows viewing f both as a function of the type bool → bool and of the type int → int. Thus, the handled expression is well typed. Turning to the operation clause, since the type signature of get_id is $\forall \alpha.$ unit $\hookrightarrow \alpha \rightarrow \alpha$, typechecking first allocates a fresh type variable α and assigns the argument variable x the type unit. The signature also requires the arguments of the resumption expressions to have the type $\alpha \rightarrow \alpha$, and both arguments λz1. ... z1 and λz2. z1 do indeed. The latter function is typed at $\alpha \rightarrow \alpha$ because the requirement to the former ensures that z1 has α. Thus, the entire program is well typed.

However, this program gets stuck. The evaluation starts with the call of get_id in the handled expression. It constructs the following delimited continuation:

```
         handle
           let f = [] in
c  ≝       if (f true) then ((f 0) + 1) else 2
         with
           return x → x
           get_id x → resume (λz1. let _ = resume (λz2. z1) in z1) .
```

The run-time system then replaces the resumption expressions with the invocation of the delimited continuation. Namely, the entire program evaluates to

$$M \stackrel{\text{def}}{=} c[\lambda\texttt{z1. let \_ = } c[\lambda\texttt{z2. z1}]\texttt{ in z1}]\ .$$

The evaluation of M proceeds as follows.

```
        handle
          let f = (λz1. let _ = c[λz2. z1] in z1) in
M  =      if (f true) then ((f 0) + 1) else 2
        with ...
   ⟶   handle if (λz1. let _ = c[λz2. z1] in z1) true then ... with ...
   ⟶   handle if (let _ = c[λz2. true] in true) then ... with ...
```

Subsequently, the term $c[\lambda$z2. **true**$]$ is to be evaluated. The delimited continuation c expects the hole to be filled with a polymorphic function of $\forall \alpha. \alpha \rightarrow \alpha$ but the function λz2. **true** is *not* polymorphic. As a result, the term gets stuck:

$$
\begin{array}{rcl}
c[\lambda \texttt{z2. }\textbf{true}] & = & \begin{array}{l}\textbf{handle}\\ \quad \textbf{let}\ \texttt{f = }\lambda\texttt{z2. }\textbf{true}\ \textbf{in}\\ \quad \textbf{if}\ \texttt{(f }\textbf{true}\texttt{)}\ \textbf{then}\ \texttt{((f 0) + 1)}\ \textbf{else}\ \texttt{2}\\ \textbf{with}\ \ldots\end{array}\\
& \longrightarrow^* & \textbf{handle}\ \texttt{((}\lambda\texttt{z2. }\textbf{true}\texttt{) 0) + 1}\ \textbf{with}\ \ldots\\
& \longrightarrow & \textbf{handle}\ \textbf{true}\ \texttt{+ 1}\ \textbf{with}\ \ldots
\end{array}
$$

A standard approach to this problem is to restrict operation calls in polymorphic expressions [Appel and MacQueen 1991; Asai and Kameyama 2007; Garrigue 2004; Hoang et al. 1993; Leroy and Weis 1991; Tofte 1990; Wright 1995]. While this kind of approach prevents `#get_id()` from having a polymorphic type, it disallows calls of any polymorphic operation inside polymorphic expressions even when the calls are safe; refer to Sekiyama and Igarashi [2019] for further discussion. Sekiyama and Igarashi [2019] have proposed a complementary approach to this problem, that is, restricting, by typing, the handler of a polymorphic operation, instead of restricting handled expressions.

2.4 Our Work: Signature Restriction

This work takes a new approach to ensuring the safety of any call of an operation. Instead of restricting how it is used or implemented, we restrict its type signature: An operation $\mathsf{op} : \forall\alpha. A \hookrightarrow B$ should not have a "bad" occurrence of α in A and B. We refer to this restriction as *signature restriction.*

To see how the signature restriction works, let us explain why type preservation is not easy to prove with the following example, where type abstraction $\Lambda\beta. M$ and type application $M\{A\}$ are explicit for the ease of readability:

$$\textbf{handle}\ \textbf{let}\ \texttt{f} = \Lambda\beta.\texttt{\#op}\{\beta\}(v)\ \textbf{in}\ M\ \textbf{with}\ H\ .$$

Here, we suppose the type signature of op to be $\forall\alpha. A \hookrightarrow B$. Notice that the type variable α in the signature $\forall\alpha. A \hookrightarrow B$ is instantiated to β, which is locally bound by $\Lambda\beta$. Handling of operation op constructs the following delimited continuation:

$$c \stackrel{\text{def}}{=} \textbf{handle}\ \textbf{let}\ \texttt{f} = \Lambda\beta.[]\ \textbf{in}\ M\ \textbf{with}\ H\ .$$

The problem is that an appropriate type cannot be assigned to it under the typing context of the handler H: the type of the hole should be $B[\beta/\alpha]$, but the type variable β is not in the scope of H. This is a kind of *scope extrusion.* We have focused on the scope extrusion via the continuation, but the operation argument v may cause a similar problem when its type $A[\beta/\alpha]$ contains β.

This analysis suggests that instantiating polymorphic operations with *closed types*, i.e., types without free type variables (especially β here), is safe because then the types of the hole and the operation argument should not contain β and, thus, the continuation and the argument could be typed under the typing context of H.[6] However, allowing only instantiation with closed types is too restrictive. For example, it even disallows a function wrapping `select`, $\lambda x.$ `#select(`x`)`, to have a polymorphic type $\forall\alpha. \alpha\ \texttt{list} \to \alpha$ because, for the function to have this type, the bound type variable of the type signature of `select` has to be instantiated with a non-closed type α.

As another approach to addressing the scope extrusion, we introduce *strong signature restriction*, which requires that, for each polymorphic operation $\mathsf{op} : \forall\alpha. A \hookrightarrow B$, the type variable α occur *only negatively* in A and *only positively* in B. This is a sufficient condition to prove type preservation. Consider, for example, the expression

$$M_1 \stackrel{\text{def}}{=} \textbf{handle}\ \textbf{let}\ \texttt{f} = \Lambda\beta_1\ldots\beta_n.\texttt{\#op}\{C\}(v)\ \textbf{in}\ M\ \textbf{with}\ H$$

[6]More precisely, the argument may contain free type variables even when its type does not. However, we could address this situation successfully by eliminating them with closing type substitution as in Sekiyama and Igarashi [2019].

where v is a value and C is a type with free type variables $\beta_1, \ldots, \beta_n$. The idea is to rewrite this expression, immediately before the call of op, to

$$M_1' \overset{\text{def}}{=} \textbf{handle let } \mathtt{f} = \Lambda\beta_1 \ldots \beta_n . \#\mathtt{op}\{\forall\beta_1 \ldots \beta_n . C\}(v) \textbf{ in } M \textbf{ with } H$$

(where the rewritten part is shaded). In M_1', because the type variable α in $\mathtt{op} : \forall\alpha. A \hookrightarrow B$ is instantiated with a closed type $\forall\beta_1 \ldots \beta_n . C$, this operation call should be safe provided that M_1' is well typed. This expression is indeed typable if the strong signature restriction is enforced, as seen below. To ensure that M_1' is typable, we need to have

$$\begin{aligned} v : A[\forall\beta_1 \ldots \beta_n . C/\alpha] &\quad \text{(for typing } \#\mathtt{op}\ \{\forall\beta_1 \ldots \beta_n . C\}(v)) \\ \#\mathtt{op}\ \{\forall\beta_1 \ldots \beta_n . C\}(v) : B[C/\alpha] &\quad \text{(for type preservation)}\ . \end{aligned}$$

To this end, we employ *type containment* [Mitchell 1988], which is also known as “subtyping for second-order types” [Tiuryn and Urzyczyn 1996]. Type containment $\sqsubseteq$ accepts the following key judgments:

$$\begin{aligned} A[C/\alpha] &\sqsubseteq A[\forall\beta_1 \ldots \beta_n . C/\alpha] \\ B[\forall\beta_1 \ldots \beta_n . C/\alpha] &\sqsubseteq B[C/\alpha]\ , \end{aligned}$$

which follow from the acceptance of type instantiation $(\forall\beta_1 \ldots \beta_n . C) \sqsubseteq C$ and the strong signature restriction which assumes that α occurs only negatively in A and only positively in B. Since M_1 is typable, we have $v : A[C/\alpha]$ and, by subsumption, $v : A[\forall\beta_1 \ldots \beta_n . C/\alpha]$. Therefore, the operation $\#\mathtt{op}\{\forall\beta_1 \ldots \beta_n . C\}$ is applicable to v and we have $\#\mathtt{op}\{\forall\beta_1 \ldots \beta_n . C\}(v) : B[\forall\beta_1 \ldots \beta_n . C/\alpha]$. Again, by subsumption, $\#\mathtt{op}\{\forall\beta_1 \ldots \beta_n . C\}(v) : B[C/\alpha]$ as desired. Therefore, M_1' is also typable. Note that the translation from M_1 to M_1' does not change the underlying untyped term, but only the types of (sub)expressions; hence, if M_1' does not get stuck, neither does M_1.

However, the strong signature restriction is still unsatisfactory in that the type signatures of many operations do not conform to it. For example, the signature of $\mathtt{select} : \forall\alpha. \alpha\ \mathtt{list} \hookrightarrow \alpha$ in Section 2.2 does not satisfy the requirements for the strong signature restriction; it disallows positive occurrences of a bound type variable in the left-hand side of $\hookrightarrow$.

Signature restriction is a relaxation of the strong signature restriction, allowing the type variable α in the signature $\forall\alpha. A \hookrightarrow B$ to occur at *strictly positive* positions in A in addition to negative positions. The proof of type preservation in this generalized case is essentially the same as above, but we need an additional type containment rule, known as the distributive law:

$$\forall\alpha. A \rightarrow B \sqsubseteq A \rightarrow \forall\alpha. B \quad \text{(if } \alpha \text{ does not occur free in } A)$$

to derive type containment judgments such as those derived above. The type signature of select conforms to this relaxed condition—α only occurs at a strictly positive position in the domain type $\alpha\ \mathtt{list}$; therefore, we can ensure the safety of the operation call of select in polymorphic expressions.

The signature restriction is a reasonable relaxation in that it rejects unsafe operations as expected. For example, get_id does not conform to the signature restriction because, in its type signature $\forall\alpha.\ \mathtt{unit} \hookrightarrow \alpha \rightarrow \alpha$, the bound type variable α occurs negatively in the codomain type $\alpha \rightarrow \alpha$.

3 A λ-CALCULUS WITH ALGEBRAIC EFFECTS AND HANDLERS

This section defines the syntax and semantics of our base language λ_{eff}, a λ-calculus extended with algebraic effects and handlers. They are based on those of the core calculus of the language Koka [Leijen 2017]. The only difference is that the Koka core calculus is equipped with let-expressions whereas λ_{eff} is not because we focus on implicit full polymorphism, rather than only on let-polymorphism. We will present a polymorphic type system for λ_{eff} that takes into account signature restriction in Section 4. We also extend λ_{eff} and the polymorphic type system with products, sums, and lists in

Variables x, y, z, f, k	**Effect operations** op		
Constants c		$::=$	true | false | 0 | + | ...
Terms M		$::=$	$x \mid c \mid \lambda x.M \mid M_1\, M_2 \mid \#\mathrm{op}(M) \mid \mathsf{handle}\, M\, \mathsf{with}\, H$
Handlers H		$::=$	$\mathsf{return}\, x \to M \mid H; \mathrm{op}(x, k) \to M$
Values v		$::=$	$c \mid \lambda x.M$
Evaluation contexts E		$::=$	$[] \mid E\, M_2 \mid v_1\, E \mid \#\mathrm{op}(E) \mid \mathsf{handle}\, E\, \mathsf{with}\, H$

Fig. 1. Syntax of λ_{eff}.

Section 5 and provide an effect system for the extended language in Section 6. Signature restriction differentiates these systems from the typing discipline of Koka. Besides, contrary to Koka's effect system, which is row-based, our effect system is not; refer to Section 6 for detail.

3.1 Syntax

Figure 1 presents the syntax of λ_{eff}. We use the metavariables x, y, z, f, k for variables and op for effect operations. Our language λ_{eff} is parameterized over constants, which are ranged over by c and may include basic values, such as Boolean and integer values, and basic operations for them, such as not, +, −, mod, etc.

Terms, ranged over by M, are from the λ-calculus, augmented with constructors for algebraic effects and handlers. They are composed of: variables; constants; lambda abstractions $\lambda x.M$, where variable x is bound in M; function applications $M_1\, M_2$; operation calls $\#\mathrm{op}(M)$ with arguments M; and handle–with expressions $\mathsf{handle}\, M\, \mathsf{with}\, H$, which install handler H to interpret effect operations performed by M. A resumption expression `resume` M that appears in Section 2 is the syntactic sugar of function application $k\, M$ where k is a variable that denotes delimited continuations and is introduced by an operation clause in a handler (we will see the definition of operation clauses shortly). The definition of evaluation contexts, ranged over by E, is standard; it indicates that the semantics of λ_{eff} is call-by-value and terms are evaluated from left to right.

Handlers, ranged over by H, consist of a single return clause and zero or more operation clauses. A return clause takes the form $\mathsf{return}\, x \to M$, where x is bound in M. The body M is evaluated once a handled expression produces a value, to which x is bound in M. An operation clause $\mathrm{op}(x, k) \to M$, where x and k are bound in M, is an implementation of the effect operation op. The body M is evaluated once a handled expression performs op, referring to the argument of op by x. Variable k denotes the delimited continuation from the point where op is called up to the handle–with expression that installs the operation clause. This ability to manipulate delimited continuations enables the implementation of various control effects. In this paper, we suppose that a handler may contain at most one operation clause for each operation.

Here, we introduce a few notions about syntax; they are standard, and therefore we omit their formal definitions. Term $M_1[M_2/x]$ is the one obtained by substituting M_2 for x in M_1 in a capture-avoiding manner. A term M is closed if it has no free variable. We also write $E[M]$ and $E[E']$ for the term and evaluation context obtained by filling the hole of E with M and E', respectively.

3.2 Semantics

This section defines the semantics of λ_{eff}. It consists of two binary relations over closed terms: the reduction relation $\rightsquigarrow$, which gives the notion of basic computation such as β-reduction, and the evaluation relation $\longrightarrow$, which defines how to evaluate programs. These relations are defined by the rules shown in Figure 2.

Reduction rules $\boxed{M_1 \rightsquigarrow M_2}$

$$\begin{array}{rcll}
c\ v & \rightsquigarrow & \zeta(c, v) & \text{R\_Const} \\
(\lambda x.M)\ v & \rightsquigarrow & M[v/x] & \text{R\_Beta} \\
\mathsf{handle}\ v\ \mathsf{with}\ H & \rightsquigarrow & M[v/x] \quad (\text{where } H^{\mathsf{return}} = \mathsf{return}\ x \rightarrow M) & \text{R\_Return} \\
\mathsf{handle}\ E[\#\mathsf{op}(v)]\ \mathsf{with}\ H & \rightsquigarrow & M[v/x][\lambda y.\mathsf{handle}\ E[y]\ \mathsf{with}\ H/k] & \text{R\_Handle} \\
 & & (\text{where } \mathsf{op} \notin E \text{ and } H(\mathsf{op}) = \mathsf{op}(x, k) \rightarrow M) &
\end{array}$$

Evaluation rules $\boxed{M_1 \longrightarrow M_2}$

$$\frac{M_1 \rightsquigarrow M_2}{E[M_1] \longrightarrow E[M_2]}\ \text{E\_Eval}$$

Fig. 2. Semantics of λ_{eff}.

The reduction relation is defined by four rules. The rule (R_Const) is for constant applications. The denotations of functional constants are given by ζ, which is a mapping from pairs of a constant c and a value v to the value that is the result of applying c to v. A function application $(\lambda x.M)\ v$ reduces to $M[v/x]$, as usual, by (R_Beta). The other two rules are for computation in terms of algebraic effects and handlers. The rule (R_Return) is for the case in which a handled expression evaluates to a value. In such a case, the return clause of the installed handler is evaluated with the value of the handled expression. We write H^{return} for the return clause of a handler H. The rule (R_Handle) is the core of effectful computation in algebraic effects and handlers, and looks for an operation clause to interpret an operation invoked by a handled expression. The redex is a handle–with expression that takes the form $\mathsf{handle}\ E[\#\mathsf{op}(v)]\ \mathsf{with}\ H$ where the handled expression $E[\#\mathsf{op}(v)]$ performs the operation op and E does not install handlers to interpret it. We call evaluation contexts that install no handler to interpret op op*-free*, which is formally defined as follows.

Definition 1 (op-free evaluation contexts). *Evaluation context E is* op-free*, written* $\mathsf{op} \notin E$*, if and only if, there exist no E_1, E_2, and H such that $E = E_1[\mathsf{handle}\ E_2\ \mathsf{with}\ H]$ and H has an operation clause for* op.

We also denote the operation clause for op in H by $H(\mathsf{op})$. Then, the conjunction of $\mathsf{op} \notin E$ and $H(\mathsf{op}) = \mathsf{op}(x, k) \rightarrow M$ in (R_Handle) means that the operation clause $\mathsf{op}(x, k) \rightarrow M$ installed by the handle–with expression is the innermost among the operation clauses for op from the point at which op is invoked. The handle–with expression with such an operation clause reduces to the body M of the operation clause after substituting the argument v of the operation call for x and the functional representation of the delimited continuation $\lambda y.\mathsf{handle}\ E[y]\ \mathsf{with}\ H$ for k.

The evaluation proceeds according to the evaluation rule (E_Eval) in Figure 2. A program is decomposed into the evaluation context E and the redex M_1 and evaluates to the term $E[M_2]$ obtained by filling the hole of E with the resulting term M_2 of the reduction of M_1.

4 A POLYMORPHIC TYPE SYSTEM FOR SIGNATURE RESTRICTION

This section defines a polymorphic type system for λ_{eff} that incorporates type containment as subsumption. We then formalize signature restriction and show that the type system is sound if all operations satisfy signature restriction. The type system in this section does not track effect information for simplicity, so a well-typed program may terminate at an unhandled operation call.

$$
\begin{array}{llll}
\textbf{Type variables} \quad \alpha, \beta, \gamma & & \textbf{Base types} \quad \iota ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots \\
\textbf{Types} \quad A, B, C, D & ::= & \alpha \mid \iota \mid A \rightarrow B \mid \forall\, \alpha.\, A \\
\textbf{Typing contexts} \quad \Gamma & ::= & \emptyset \mid \Gamma, x : A \mid \Gamma, \alpha
\end{array}
$$

Fig. 3. The type language.

4.1 Type Language

Figure 3 presents the type language of the polymorphic type system. It is standard and in fact the same as that of System F [Girard 1972; Reynolds 1974]. We use metavariables α, β, γ for type variables and ι for base types such as bool and int. Types, ranged over by A, B, C, D, consist of: type variables; base types; function types $A \rightarrow B$; and polymorphic types $\forall\, \alpha.\, A$, where type variable α is bound in A. Typing contexts, ranged over by Γ, are sequences of bindings of variables coupled with their types and type variables. We suppose that each constant c is assigned a first-order closed type $ty(c)$ of the form $\iota \rightarrow \ldots \rightarrow \iota_n \rightarrow \iota_{n+1}$ which is consistent with the denotation of c.

We use the following shorthand and notions. We write $\boldsymbol{\alpha}^I$ for $\boldsymbol{\alpha} = \alpha_1, \cdots, \alpha_n$ with $I = \{1, ..., n\}$. We apply this bold-font notation to other syntax categories as well; for example, $\boldsymbol{A}^I$ denotes a sequence of types. We often omit the index sets (I, $\mathcal{J}$, K) if they are clear from the context or irrelevant: for example, we may abbreviate $\boldsymbol{\alpha}^I$ to $\boldsymbol{\alpha}$. We also write $\forall\, \boldsymbol{\alpha}^I.\, A$ for $\forall \alpha_1. \ldots \forall \alpha_n.\, A$ with $I = \{1, ..., n\}$. We may omit the index sets and write $\forall\, \boldsymbol{\alpha}.\, A$ simply. We write $\forall\, \boldsymbol{\alpha}^I \boldsymbol{.}\, \boldsymbol{A}^{\mathcal{J}}$ for a sequence of types $\forall\, \boldsymbol{\alpha}^I.\, A_1, \ldots, \forall\, \boldsymbol{\alpha}^I.\, A_n$ with $\mathcal{J} = \{1, \ldots, n\}$. The notions of free type variables and capture-avoiding type substitution are defined as usual. We write $ftv(A)$ for the set of free type variables of A and $A[\boldsymbol{B}/\boldsymbol{\alpha}]$ for the type obtained by substituting each type of $\boldsymbol{B}$ for the corresponding type variable of $\boldsymbol{\alpha}$ simultaneously (here we suppose that $\boldsymbol{B}$ and $\boldsymbol{\alpha}$ share the same, omitted index set).

4.2 Polymorphic Type System

We present a polymorphic type system for λ_{eff}, which consists of four judgments: well-formedness judgment $\vdash \Gamma$, which states that a typing context Γ is well formed; type containment judgment $\Gamma \vdash A \sqsubseteq B$, which states that, for the types A and B, which are assumed to be well formed under Γ, the inhabitants of A are contained in B; term typing judgment $\Gamma \vdash M : A$, which states that term M evaluates to a value of A after applying appropriate substitution for variables and type variables in Γ; and handler typing judgment $\Gamma \vdash H : A \Rightarrow B$, which states that handler H handles operations called by a handled term of A and produces a value of B after applying appropriate substitution according to Γ (we refer to A and B as the *input* and *output types* of the handler, respectively). These judgments are defined as the smallest relations that satisfy the rules in Figure 4.

The well-formedness rules are standard. A typing context is well formed if (1) variables and type variables bound by it are unique and (2) it assigns well-formed types to the variables. We write $dom(\Gamma)$ for the set of variables and type variables bound by Γ. A type A is well formed under typing context Γ, which is expressed by $\Gamma \vdash A$, if and only if $ftv(A) \subseteq dom(\Gamma)$ (i.e., Γ binds all of the free type variables in A).

The type containment rules originate from the work of Tiuryn and Urzyczyn [1996], which simplifies the rules of type containment of Mitchell [1988]. The rules (C_Refl) and (C_Trans) indicate that type containment is a preorder. The rule (C_Inst) instantiates polymorphic types with well-formed types. The rule (C_Gen) may add a quantifier $\forall$ if it does not bind free type variables. The rules (C_Poly) and (C_Fun) are for compatibility; note that type containment is a kind of subtyping and hence it is contravariant on the domain types of function types. The rule (C_DFun) is a simplified version [Tiuryn and Urzyczyn 1996] of the original "distributive" law [Mitchell 1988],

Well-formedness $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ \text{WF\_EMPTY} \qquad \frac{x \notin dom(\Gamma) \quad \Gamma \vdash A}{\vdash \Gamma, x{:}A}\ \text{WF\_EXTVAR} \qquad \frac{\alpha \notin dom(\Gamma) \quad \vdash \Gamma}{\vdash \Gamma, \alpha}\ \text{WF\_EXTTYVAR}$$

Type containment $\boxed{\Gamma \vdash A \sqsubseteq B}$

$$\frac{\vdash \Gamma}{\Gamma \vdash A \sqsubseteq A}\ \text{C\_REFL} \qquad \frac{\Gamma \vdash A \sqsubseteq C \quad \Gamma \vdash C \sqsubseteq B}{\Gamma \vdash A \sqsubseteq B}\ \text{C\_TRANS}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash \forall \alpha. A \sqsubseteq A[B/\alpha]}\ \text{C\_INST} \qquad \frac{\vdash \Gamma \quad \alpha \notin ftv(A)}{\Gamma \vdash A \sqsubseteq \forall \alpha. A}\ \text{C\_GEN} \qquad \frac{\Gamma, \alpha \vdash A \sqsubseteq B}{\Gamma \vdash \forall \alpha. A \sqsubseteq \forall \alpha. B}\ \text{C\_POLY}$$

$$\frac{\Gamma \vdash B_1 \sqsubseteq A_1 \quad \Gamma \vdash A_2 \sqsubseteq B_2}{\Gamma \vdash A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2}\ \text{C\_FUN} \qquad \frac{\vdash \Gamma \quad \alpha \notin ftv(A)}{\Gamma \vdash \forall \alpha. A \rightarrow B \sqsubseteq A \rightarrow \forall \alpha. B}\ \text{C\_DFUN}$$

Term typing $\boxed{\Gamma \vdash M : A}$

$$\frac{\vdash \Gamma \quad x{:}A \in \Gamma}{\Gamma \vdash x : A}\ \text{T\_VAR} \qquad \frac{\vdash \Gamma}{\Gamma \vdash c : ty(c)}\ \text{T\_CONST} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \rightarrow B}\ \text{T\_ABS}$$

$$\frac{\Gamma \vdash M_1 : A \rightarrow B \quad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1\, M_2 : B}\ \text{T\_APP} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash A \sqsubseteq B \quad \Gamma \vdash B}{\Gamma \vdash M : B}\ \text{T\_INST}$$

$$\frac{\Gamma, \alpha \vdash M : A}{\Gamma \vdash M : \forall \alpha. A}\ \text{T\_GEN} \qquad \frac{ty(\mathsf{op}) = \forall \boldsymbol{\alpha}. A \hookrightarrow B \quad \Gamma \vdash M : A[\boldsymbol{C}/\boldsymbol{\alpha}] \quad \Gamma \vdash \boldsymbol{C}}{\Gamma \vdash \#\mathsf{op}(M) : B[\boldsymbol{C}/\boldsymbol{\alpha}]}\ \text{T\_OP}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash H : A \Rightarrow B}{\Gamma \vdash \mathsf{handle}\, M\, \mathsf{with}\, H : B}\ \text{T\_HANDLE}$$

Handler typing $\boxed{\Gamma \vdash H : A \Rightarrow B}$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \mathsf{return}\, x \rightarrow M : A \Rightarrow B}\ \text{TH\_RETURN}$$

$$\frac{\Gamma \vdash H : A \Rightarrow B \quad ty(\mathsf{op}) = \forall \boldsymbol{\alpha}. C \hookrightarrow D \quad \Gamma, \boldsymbol{\alpha}, x{:}C, k{:}D \rightarrow B \vdash M : B}{\Gamma \vdash H; \mathsf{op}(x, k) \rightarrow M : A \Rightarrow B}\ \text{TH\_OP}$$

Fig. 4. Polymorphic type system for λ_{eff}.

which is the core of type containment. This rule allows $\forall$ that quantifies a function type to move to its codomain type if the quantified type variable does not occur free in the domain type. This rule is justified by the fact that we can supply a function from $\forall \alpha. A \rightarrow B$ to $A \rightarrow \forall \alpha. B$ in System F and the result of applying type erasure to it is equivalent to the identity function [Mitchell 1988]. This rule is crucial for allowing the domain type of a type signature to refer to quantified type variables in strictly positive positions, which makes signature restriction permissive.

The typing rules for terms are almost standard, coming from Mitchell [1988] for polymorphism and Plotkin and Pretnar [2013] for effects. The rule (T_INST) converts types by type containment. The rule (T_OP) is for operation calls. We formalize a type signature of an operation as follows.

DEFINITION 2 (TYPE SIGNATURE). *Each effect operation* op *is assigned a type signature* $ty(\mathsf{op})$ *of the form* $\forall \alpha_1. \ldots \forall \alpha_n. A \hookrightarrow B$ *for some* n*, where* $\alpha_1, \ldots, \alpha_n$ *are bound in the domain type* A *and* *codomain type B. It may be abbreviated to $\forall\, \boldsymbol{\alpha}^I.\, A \hookrightarrow B$ or, more simply, to $\forall\, \boldsymbol{\alpha}.\, A \hookrightarrow B$. We suppose that $\forall \alpha_1. \dots \forall \alpha_n.\, A \hookrightarrow B$ is closed, i.e., $ftv(A), ftv(B) \subseteq \{\alpha_1, \cdots, \alpha_n\}$.*

We note that domain and codomain types may involve polymorphic types.

The rule (T_Op) instantiates the type signature of the operation with well-formed types and checks that an argument is typed at the domain type of the instantiated signature. We use notation $\Gamma \vdash \boldsymbol{C}$ for $\Gamma \vdash C_1, \cdots, \Gamma \vdash C_n$ when $\boldsymbol{C} = C_1, \cdots, C_n$.

The typing rules for handlers are also ordinary [Plotkin and Pretnar 2013]. A return clause $\mathsf{return}\, x \longrightarrow M$ is typechecked by (TH_Return), which allows the body M to refer to the values of the handled expression via bound variable x. An operation clause $\mathsf{op}(x, k) \longrightarrow M$ is typechecked by (TH_Op). Let the type signature of op be $\forall\, \boldsymbol{\alpha}.\, C \hookrightarrow D$. In typechecking M, variable x is assigned the codomain type C since variable x will be bound to the arguments to the operation op. Variable k is assigned to type $D \longrightarrow B$ where B is the output type of the handler. This is because k will be bound to the functional representations of delimited continuations such that: the delimited continuations suppose that their holes are filled with values of the codomain type D of the type signature; and they are wrapped by the handle–with expression installing the handler and therefore they would produce values of B.

4.3 Desired Propositions for Type Soundness

As mentioned in Section 2.3, the polymorphic type system is unsound if we impose no further restriction on it. This section details the proof sketch of type preservation provided in Section 2.4 and formulates two propositions such that they do not hold in the polymorphic type system but, *if they did*, the type system would be sound. In Section 4.4.2, we show that the propositions hold if all operations satisfy signature restriction.

We start by considering an issue that arises when proving soundness of the polymorphic type system. This issue relates to the handling of an operation call by (R_Handle), which enables the following reduction:

$$\mathsf{handle}\, E[\#\mathsf{op}(v)]\, \mathsf{with}\, H \rightsquigarrow M[v/x][\lambda y.\mathsf{handle}\, E[y]\, \mathsf{with}\, H/k]$$

where $\mathsf{op} \notin E$ and $H(\mathsf{op}) = \mathsf{op}(x, k) \longrightarrow M$. The problem is that the RHS term does not preserve the type of the LHS term. If this type preservation were successful, we would be able to prove soundness of the polymorphic type system, but it is contradictory to the existence of the counterexample presented in Section 2.3.

A detailed investigation of this problem reveals two propositions that are lacking but sufficient to make the polymorphic type system sound.

Proposition 1. *If $ty(\mathsf{op}) = \forall\, \boldsymbol{\alpha}^I.\, A \hookrightarrow B$ and $\Gamma \vdash M : \forall\, \boldsymbol{\beta}^J.\, A[\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$, then $\Gamma \vdash M : A[\forall\, \boldsymbol{\beta}^J.\, \boldsymbol{C}^I/\boldsymbol{\alpha}^I]$.*

Proposition 2. *If $ty(\mathsf{op}) = \forall\, \boldsymbol{\alpha}^I.\, A \hookrightarrow B$ and $\Gamma \vdash M : B[\forall\, \boldsymbol{\beta}^J.\, \boldsymbol{C}^I/\boldsymbol{\alpha}^I]$, then $\Gamma \vdash M : \forall\, \boldsymbol{\beta}^J.\, B[\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$.*

In what follows, we show how these propositions allow us to prove type soundness. Before that, we first fix and examine the type information of the terms appearing in the LHS term. Let us suppose that $ty(\mathsf{op}) = \forall\, \boldsymbol{\alpha}^I.\, A \hookrightarrow B$ and that the LHS term has a type D under a typing context Γ. We can then find that

$$\Gamma, \boldsymbol{\alpha}^I, x : A, k : B \longrightarrow D \vdash M : D \tag{1}$$

is derived. Turning to the handled expression $E[\#\mathsf{op}(v)]$, we can find two facts about the typing judgment for v. The first fact originates from (T_Op): since v is an argument of operation op, it should be of $A[\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$, which is a type obtained by substituting certain types $\boldsymbol{C}^I$ for type variables $\boldsymbol{\alpha}^I$ in the domain type A of the type signature of op. The second is from (T_Gen), which allows the generalization of types *anywhere*. Thus, v is well typed under a typing context $\Gamma, \boldsymbol{\beta}^J$, an extension of Γ with some type variables $\boldsymbol{\beta}^{\mathcal{J}}$ (note that $I \neq \mathcal{J}$ in general). In summary, the typing judgment for v takes the following form:

$$\Gamma, \boldsymbol{\beta}^{\mathcal{J}} \vdash v : A[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}] \,. \tag{2}$$

Now, we show that Proposition 1 makes $M[v/x]$ typed at D. First, we can derive

$$\Gamma \vdash v : \forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, A[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}]$$

by the typing derivation of judgment (2) and (T_Gen). Proposition 1 enables us to prove

$$\Gamma \vdash v : A[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}] \,. \tag{3}$$

We can also derive

$$\Gamma, x : A\,[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}], k : B[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}] \to D \vdash M : D$$

by substituting $\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}$ for $\boldsymbol{\alpha}^{I}$ in the typing judgment (1); note that the type variables in $\boldsymbol{\alpha}^{I}$ do not occur free in D because they are bound by the type signature. Thus, we can derive

$$\Gamma, k : B[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}] \to D \vdash M[v/x] : D \tag{4}$$

using an ordinary substitution lemma with the derivation for judgment (3).

Next, we show that Proposition 2 makes $M[v/x][\lambda y.\mathsf{handle}\, E[y]\, \mathsf{with}\, H/k]$ typed at D. This is possible if

$$\Gamma \vdash \lambda y.\mathsf{handle}\, E[y]\, \mathsf{with}\, H : B[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}] \to D$$

is derivable, jointly with the derivation of typing judgment (4). Namely, it suffices to derive

$$\Gamma, y : B\,[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}] \vdash \mathsf{handle}\, E[y]\, \mathsf{with}\, H : D \,.$$

By an observation similar to v, we find that $\#\mathrm{op}(v)$ is typed at $B[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}]$ under $\Gamma, \boldsymbol{\beta}^{\mathcal{J}}$ (note that B is the codomain type of the type signature of op). Thus, for the above typing judgment to hold, it suffices for y to have the same type as $\#\mathrm{op}(v)$. Hence, we will derive

$$\Gamma, y : B\,[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}], \boldsymbol{\beta}^{\mathcal{J}} \vdash y : B[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}] \,. \tag{5}$$

Because $\Gamma, y : B\,[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}], \boldsymbol{\beta}^{\mathcal{J}} \vdash y : B[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}]$, we can derive $\Gamma, y : B\,[\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, \boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}], \boldsymbol{\beta}^{\mathcal{J}} \vdash y : \forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, B[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}]$ by Proposition 2; and, by instantiating $\forall\, \boldsymbol{\beta}^{\mathcal{J}}.\, B[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}]$ to $B[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}]$ with $\boldsymbol{\beta}^{\mathcal{J}}$ in the typing context, we have succeeded in deriving the typing judgment (5).

Thus, if Propositions 1 and 2 held, we could derive

$$\Gamma \vdash M[v/x][\lambda y.\mathsf{handle}\, E[y]\, \mathsf{with}\, H/k] : D \,.$$

The polymorphic type system in Section 4.2 does not actually have these propositions, but imposing signature restriction produces a type system that does have them.

4.4 Signature Restriction

This section formalizes signature restriction for λ_{eff} and shows that it implies Propositions 1 and 2.

4.4.1 Definition. As described in Section 2.4, signature restriction rests on the polarity of the occurrences of quantified type variables in a type signature. The polarity is defined in a standard manner, as follows.

Definition 3 (Polarity of type variable occurrence). *The positive and negative occurrences of a type variable in a type A are defined by induction on A, as follows.*

- *The occurrence of α in type α is positive.*
- *The positive (resp. negative) occurrences of α in $A \to B$ are the negative (resp. positive) occurrences of α in A and the positive (resp. negative) occurrences of α in B.*

- *The positive (resp. negative) occurrences of α in $\forall\,\beta.\,A$, where β is supposed to be distinct from α, are the positive (resp. negative) occurrences of α in A.*

The strictly positive occurrences of a type variable in a type are defined as follows.

- *The occurrence of α in type α is strictly positive.*
- *The strictly positive occurrences of α in $A \rightarrow B$ are the strictly positive occurrences of α in B.*
- *The strictly positive occurrences of α in $\forall\,\beta.\,A$, where β is supposed to be distinct from α, are the strictly positive occurrences of α in A.*

Definition 4 (Operations satisfying signature restriction). *An operation* op *having type signature $ty(\mathrm{op}) = \forall\,\boldsymbol{\alpha}.\,A \hookrightarrow B$ satisfies the signature restriction if and only if: (1) the occurrences of each type variable of $\boldsymbol{\alpha}$ in A are only negative or strictly positive; and (2) the occurrences of each type variable of $\boldsymbol{\alpha}$ in B are only positive.*

The signature restriction allows quantified type variables to occur at strictly positive positions of the domain type of a type signature. This is crucial for many operations, such as `raise`, `fail`, and `select`, to conform to signature restriction. The rule (C_DFun) plays an important role to permit this capability, as seen in the next section.

We can easily confirm whether an operation satisfies the signature restriction. For example, it is easy to determine that `get_id` does not satisfy the signature restriction: since its type signature is $\forall\,\alpha.\,\mathrm{unit} \hookrightarrow \alpha \rightarrow \alpha$, the quantified type variable α occurs not only at a positive position but also at a negative position in the codomain type $\alpha \rightarrow \alpha$. By contrast, the operations `raise` and `fail` given in Section 2 satisfy the signature restriction because their type signature $\forall\,\alpha.\,\mathrm{unit} \hookrightarrow \alpha$ meets the conditions in Definition 4. To determine whether `select` satisfies the signature restriction, we need to extend λ_{eff} and the polymorphic type system by introducing other programming constructs such as lists. Particulars of this extension are presented in Section 5.

4.4.2 *Proofs of the Desired Propositions.* The signature restriction enables us to prove Propositions 1 and 2, which are crucial to show that reduction preserves typing. Below is the key lemma for that.

Lemma 1. *Suppose that α does not appear free in A.*

(1) If the occurrences of β in A are only negative or strictly positive, then $\Gamma \vdash \forall\,\alpha.\,A[B/\beta] \sqsubseteq A[\forall\,\alpha.\,B/\beta]$.
(2) If the occurrences of β in A are only positive, then $\Gamma \vdash A[\forall\,\alpha.\,B/\beta] \sqsubseteq \forall\,\alpha.\,A[B/\beta]$.

This lemma means that an operation op conforming to the signature restriction satisfies Propositions 1 and 2. For Proposition 1: suppose $ty(\mathrm{op}) = \forall\,\boldsymbol{\alpha}^I.\,A \hookrightarrow B$ and $\Gamma \vdash M : \forall\,\boldsymbol{\beta}^J.\,A[\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$; since op satisfies the signature restriction, we can apply case (1) of Lemma 1, which implies $\Gamma \vdash \forall\,\boldsymbol{\beta}^J.\,A[\boldsymbol{C}^I/\boldsymbol{\alpha}^I] \sqsubseteq A[\forall\,\boldsymbol{\beta}^J.\,\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$; thus, we can derive $\Gamma \vdash M : A[\forall\,\boldsymbol{\beta}^J.\,\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$ by (T_Inst). Proposition 2 is proven similarly by using case (2) of Lemma 1 instead of case (1).

It is easy to prove Lemma 1 if the occurrences of β in A are only negative in case (1). In fact, the following lemma handles such a case (the statement is generalized slightly).

Lemma 2. *Suppose that α does not appear free in A.*

(1) If the occurrences of β in A are only negative, then $\Gamma_1, \alpha, \Gamma_2 \vdash A[B/\beta] \sqsubseteq A[\forall\,\alpha.\,B/\beta]$.
(2) If the occurrences of β in A are only positive, then $\Gamma_1, \alpha, \Gamma_2 \vdash A[\forall\,\alpha.\,B/\beta] \sqsubseteq A[B/\beta]$.

Proof. We prove both cases simultaneously by structural induction on A. The polarity condition on the occurrences of β ensures that, if $A = \beta$, it suffices to show $\Gamma_1, \alpha, \Gamma_2 \vdash \forall\,\alpha.\,B \sqsubseteq B$, which is derived by (C_Inst). □

Now, we prove Lemma 1 with Lemma 2 and (C_DFun), which is the key rule for the signature restriction to allow strictly positive occurrences of quantified type variables in the domain type of a type signature.

Proof of Lemma 1. We prove both cases simultaneously by structural induction on A. The case (2) can be proven by Lemma 2: it enables us to show $\Gamma, \alpha \vdash A[\forall \alpha. B/\beta] \sqsubseteq A[B/\beta]$; then, by (C_Poly), (C_Gen), and (C_Trans), we can derive $\Gamma \vdash A[\forall \alpha. B/\beta] \sqsubseteq \forall \alpha. A[B/\beta]$.

Let us consider case (1) where A is a function type $C \to D$; the other cases are easy to show. Suppose that the occurrences of β in $C \to D$ are only negative or strictly positive. By definition, the occurrences of β in C are only positive. Thus, by the IH, $\Gamma \vdash C[\forall \alpha. B/\beta] \sqsubseteq \forall \alpha. C[B/\beta]$. Furthermore, by definition, the occurrences of β in D are only negative or strictly positive. Thus, by the IH, $\Gamma \vdash \forall \alpha. D[B/\beta] \sqsubseteq D[\forall \alpha. B/\beta]$. By (C_Fun),

$$\Gamma \vdash (\forall \alpha. C[B/\beta]) \to \forall \alpha. D[B/\beta] \sqsubseteq C[\forall \alpha. B/\beta] \to D[\forall \alpha. B/\beta]\ . \tag{6}$$

Thus:

$$\begin{array}{ll}
\Gamma \vdash \forall \alpha. C[B/\beta] \to D[B/\beta] & \\
\quad \sqsubseteq \forall \alpha. (\forall \alpha. C[B/\beta]) \to D[B/\beta] & \text{(by (C\_Poly), (C\_Fun), and } \Gamma, \alpha \vdash \forall \alpha. C[B/\beta] \sqsubseteq C[B/\beta]) \\
\quad \sqsubseteq (\forall \alpha. C[B/\beta]) \to \forall \alpha. D[B/\beta] & \text{(by (C\_DFun))} \\
\quad \sqsubseteq C[\forall \alpha. B/\beta] \to D[\forall \alpha. B/\beta] & \text{(by (6))}\ .
\end{array}$$

□

4.5 Type Soundness

This section shows soundness of the polymorphic type system under the assumption that all operations satisfy the signature restriction. As usual, our proof rests on two properties: progress and subject reduction [Wright and Felleisen 1994]. As discussed in Sections 4.3 and 4.4, the signature restriction, together with type containment, enables us to prove subject reduction.

In this work, type containment is thus a key notion to prove type soundness, but it complicates certain inversion properties. In the literature [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007], type soundness of a language with subtyping such as $\sqsubseteq$ has been shown by translation to another language—typically, System F—where the use of subtyping is replaced by "coercions" (i.e., certain term representations for type conversion by subtyping). This approach is acceptable in the prior work because the semantics of the source language is determined by the target language. However, this approach is *not* acceptable in our setting because the terms checked by our type system should be interpreted by the semantics of λ_{eff} as they are. We thus show soundness of the polymorphic type system directly, without translation to other languages.

The property that is the most difficult to prove in the direct approach is the inversion of type containment judgments for function types.

Lemma 3. *If* $\Gamma \vdash A_1 \to A_2 \sqsubseteq B_1 \to B_2$, *then* $\Gamma \vdash B_1 \sqsubseteq A_1$ *and* $\Gamma \vdash A_2 \sqsubseteq B_2$.

We cannot prove this lemma as it is by induction on the derivation of $\Gamma \vdash A_1 \to A_2 \sqsubseteq B_1 \to B_2$ because a premise in the derivation may relate the (nonpolymorphic) function type on one side to a polymorphic function type on the other side. Thus, we need to generalize the assumption to a type containment judgment that may relate polymorphic function types: $\Gamma \vdash \forall \boldsymbol{\alpha}^{I}. A_1 \to A_2 \sqsubseteq \forall \boldsymbol{\beta}^{J}. B_1 \to B_2$. By investigating the type containment rules, we find that $\boldsymbol{\alpha}^{I}$ is split into three sequences $\boldsymbol{\alpha}_{\mathbf{01}}^{I_{01}}$, $\boldsymbol{\alpha}_{\mathbf{02}}^{I_{02}}$, and $\boldsymbol{\alpha}_{\mathbf{03}}^{I_{03}}$ depending on how the rules handle the type variables in $\boldsymbol{\alpha}^{I}$: those of $\boldsymbol{\alpha}_{\mathbf{01}}^{I_{01}}$ stay in $\boldsymbol{\beta}^{J}$; those of $\boldsymbol{\alpha}_{\mathbf{02}}^{I_{02}}$ are quantified in the return type B_2; and those of $\boldsymbol{\alpha}_{\mathbf{03}}^{I_{03}}$ are instantiated with some types $\boldsymbol{C_0}^{I_{03}}$. Furthermore, we have to take into account certain, unrevealed type variables

$\boldsymbol{\gamma}^K$ that initially emerge at the outermost position by (T_Gen) and are subsequently distributed into subcomponent types. For example:

$$A_1 \rightarrow A_2 \sqsubseteq \forall \gamma . A_1 \rightarrow A_2 \sqsubseteq (\forall \gamma . A_1) \rightarrow (\forall \gamma . A_2)$$

where $\gamma \notin \mathit{ftv}(A_1) \cup \mathit{ftv}(A_2)$. These observations are formulated in the following inversion lemma for type containment, which implies Lemma 3. We write $\{\boldsymbol{\alpha}^I\}$ to view the sequence $\boldsymbol{\alpha}^I$ as a set by ignoring the order and $\{\boldsymbol{\alpha}^I\} \uplus \{\boldsymbol{\beta}^J\}$ for the union of disjoint sets $\{\boldsymbol{\alpha}^I\}$ and $\{\boldsymbol{\beta}^J\}$.

Lemma 4 (Type containment inversion: function types). *If* $\Gamma \vdash \forall \boldsymbol{\alpha}^I . A_1 \rightarrow A_2 \sqsubseteq \forall \boldsymbol{\beta}^J . B_1 \rightarrow B_2$*, then there exist* $\boldsymbol{\alpha}_1^{I_1}$, $\boldsymbol{\alpha}_2^{I_2}$, $\boldsymbol{\gamma}^K$*, and* $\boldsymbol{C}^{I_1}$ *such that*

- $\{\boldsymbol{\alpha}^I\} = \{\boldsymbol{\alpha}_1^{I_1}\} \uplus \{\boldsymbol{\alpha}_2^{I_2}\}$,
- $\Gamma, \boldsymbol{\beta}^J, \boldsymbol{\gamma}^K \vdash \boldsymbol{C}^{I_1}$,
- $\Gamma, \boldsymbol{\beta}^J \vdash B_1 \sqsubseteq \forall \boldsymbol{\gamma}^K . A_1[\boldsymbol{C}^{I_1}/\boldsymbol{\alpha}_1^{I_1}]$, *and*
- $\Gamma, \boldsymbol{\beta}^J \vdash \forall \boldsymbol{\alpha}_2^{I_2} . \forall \boldsymbol{\gamma}^K . A_2[\boldsymbol{C}^{I_1}/\boldsymbol{\alpha}_1^{I_1}] \sqsubseteq B_2$.

In this statement, the sequence $\boldsymbol{\alpha}_2^{I_2}$ corresponds to $\boldsymbol{\alpha}_{02}^{I_{02}}$ in the above informal description and $\boldsymbol{\alpha}_1^{I_1}$ includes the type variables that remain in $\boldsymbol{\beta}^J$ (i.e., $\boldsymbol{\alpha}_{01}^{I_{01}}$) and those instantiated with some types in $\boldsymbol{C}^{I_1}$ (i.e., $\boldsymbol{\alpha}_{03}^{I_{03}}$). Type substitution $[\boldsymbol{C}^{I_1}/\boldsymbol{\alpha}_1^{I_1}]$ replaces a type variable in $\boldsymbol{\alpha}_{01}^{I_{01}}$ with itself.

We also prove other lemmas such as weakening, substitution, canonical forms, and value inversion. We omit their formal statements and proofs in this paper; the details can be found in the supplementary material.

Now, we show progress and subject reduction. In what follows, the metavariable Δ ranges over typing contexts that consist of only type variable bindings. Note that the polymorphic type system is not equipped with a mechanism to track effects, so the operations that are carried out may not be handled.

Lemma 5 (Progress). *If* $\Delta \vdash M : A$*, then:*

- $M \longrightarrow M'$ *for some* M'*;*
- M *is a value; or*
- $M = E[\#\mathrm{op}(v)]$ *for some* E, op, *and* v *such that* op $\notin E$.

Lemma 6 (Subject reduction). *Suppose that all operations satisfy the signature restriction.*

(1) *If* $\Delta \vdash M_1 : A$ *and* $M_1 \rightsquigarrow M_2$*, then* $\Delta \vdash M_2 : A$.
(2) *If* $\Delta \vdash M_1 : A$ *and* $M_1 \longrightarrow M_2$*, then* $\Delta \vdash M_2 : A$.

We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$ and $M \not\longrightarrow$ to mean that there exists no term M' such that $M \longrightarrow M'$.

Theorem 1 (Type soundness). *Suppose that all operations satisfy the signature restriction. If* $\Delta \vdash M : A$ *and* $M \longrightarrow^* M'$ *and* $M' \not\longrightarrow$*, then:*

- M' *is a value; or*
- $M' = E[\#\mathrm{op}(v)]$ *for some* E, op, *and* v *such that* op $\notin E$.

Proof. By progress (Lemma 5) and subject reduction (Lemma 6). □

Remark 1. It is natural to ask whether the signature restriction can be further relaxed. Consider a type signature $\forall \alpha . A \hookrightarrow B$. A negative occurrence of α in B is problematic as `get_id`, which has type signature $\forall \alpha . \mathsf{unit} \hookrightarrow \alpha \rightarrow \alpha$, is unsafe (see Section 2.3). A non-strictly positive occurrence of α in A is also problematic, as the following example shows. Let us consider a calculus with int, bool, and sum types $D_1 + D_2$ for simplicity (we write $\mathsf{inl}\, M$ and $\mathsf{inr}\, M$ for injection into sum types). Consider an operation op of the signature $\forall\, \alpha.\, ((\alpha \to \mathsf{int}) \to \alpha) \hookrightarrow \alpha$ and let

$$\begin{array}{lcl} v & \overset{\text{def}}{=} & \lambda f.\lambda x.\mathsf{inr}\,(f\,(\lambda y.\mathsf{inl}\, x)) \;:\; ((\beta \to (\beta + \mathsf{int})) \to \mathsf{int}) \to (\beta \to (\beta + \mathsf{int})) \\ M & \overset{\text{def}}{=} & \mathsf{let}\, g = \#\mathsf{op}(v)\,\mathsf{in}\,\mathsf{case}\, g\, 0\,\mathsf{of}\,\mathsf{inl}\, z \to z;\ \mathsf{inr}\, z \to E[g\,\mathsf{true}] \;:\; \mathsf{int}, \end{array}$$

where E is an evaluation context such that $x : \mathsf{bool} + \mathsf{int} \vdash E[x] : \mathsf{int}$ and $E[\mathsf{inr}\,\mathsf{true}]$ causes a run-time error (it is easy to construct such E). It is not difficult to check that M has type int. In $\#\mathsf{op}(v)$, the type variable α bound by the type signature is instantiated with $\beta \to (\beta + \mathsf{int})$, and thus g has type $\forall\, \beta.\, \beta \to (\beta + \mathsf{int})$. The type variable β is instantiated with int in $g\, 0$ and with bool in $g\,\mathsf{true}$. Then the counterexample is given by

$$\mathsf{handle}\, M\,\mathsf{with}\,\mathsf{return}\, x \to x; \mathsf{op}(x, k) \to k\,(x\, k) \;:\; \mathsf{int},$$

which is reduced to $\mathsf{handle}\, E[\mathsf{inr}\,\mathsf{true}]\,\mathsf{with}\,\mathsf{return}\, x \to x; \mathsf{op}(x, k) \to k\,(x\, k)$ and causes an error.

5 AN EXTENSION OF λ_{EFF}

This section demonstrates the extensibility of the signature restriction. To this end, we extend λ_{eff}, the polymorphic type system, and the signature restriction with products, sums, and lists and show soundness of the extended polymorphic type system under the extended signature restriction. We also provide a few examples of operations that satisfy the extended signature restriction.

5.1 Extended Language

The extension of λ_{eff} and the polymorphic type system is shown in Figure 5, in which the extended part of the syntax is highlighted. Terms support: pairs; projections; injections; case expressions for sums; the nil constant; cons expressions; case expressions for lists; and the fixed-point operator. A case expression matching injections $\mathsf{case}\, M\,\mathsf{of}\,\mathsf{inl}\, x \to M_1;\ \mathsf{inr}\, y \to M_2$ binds x in M_1 and y in M_2, respectively; a case expression matching lists $\mathsf{case}\, M\,\mathsf{of}\,\mathsf{nil} \to M_1;\ \mathsf{cons}\, x \to M_2$ binds x in M_2; the fixed-point operator $\mathsf{fix}\, f.\lambda x.M$ binds f and x in M. Pairs, injections, and cons expressions are values if their immediate subterms are also values. Types are extended with product types, sum types, and list types. The extension of evaluation contexts follows that of terms. For the semantics, the reduction rules for projections, case expressions, and the fixed-point operator are added. The extension of the polymorphic type system is also straightforward. Type containment is extended by adding six rules: the three rules on the left in Figure 5 are for compatibility and the three rules on the right are for distributing $\forall$ over immediate subcomponent types. All of the additional typing rules are standard and are thus omitted.

Remark 2. The rule (C_DSum) in Figure 5 may look peculiar or questionable. Actually, there exists no term M in (implicitly typed) System F such that $x : \forall\, \alpha.\, (A + B) \vdash M : (\forall\, \alpha.\, A) + (\forall\, \alpha.\, B)$, and thus the expected coercion function of $(\forall\, \alpha.\, (A + B)) \to (\forall\, \alpha.\, A) + (\forall\, \alpha.\, B)$ is not definable in System F. A justification can be given by the following fact: for every *closed value* $\vdash v : \forall\, \alpha.\, (A + B)$, one has $\vdash v : (\forall\, \alpha.\, A) + (\forall\, \alpha.\, B)$. In fact $\vdash v : \forall\, \alpha.\, (A + B)$ implies $v = \mathsf{inl}\, v'$ or $\mathsf{inr}\, v''$. Assuming the former for definiteness, $\alpha \vdash v' : A$ and thus $\vdash v' : \forall\, \alpha.\, A$.

We also extend the polarity of the occurrences of a type variable. The polarity of the occurrences in type variables, function types, and polymorphic types is given in Definition 3. We also define the polarity in product, sum, and list types as follows.

Definition 5 (Polarity of type variable occurrence in product, sum, and list types). *The positive and negative occurrences of a type variable in a product, sum, and list type are defined as follows.*

$$
\begin{array}{llll}
\textbf{Terms} & M & ::= & x \mid c \mid \lambda x.M \mid M_1\, M_2 \mid \#\mathsf{op}(M) \mid \mathsf{handle}\, M\, \mathsf{with}\, H \mid (M_1, M_2) \mid \\
& & & \pi_1 M \mid \pi_2 M \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M \mid \mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, x \rightarrow M_1;\ \mathsf{inr}\, y \rightarrow M_2 \mid \\
& & & \mathsf{nil} \mid \mathsf{cons}\, M \mid \mathsf{case}\, M\, \mathsf{of}\, \mathsf{nil}\ \rightarrow M_1;\ \mathsf{cons}\, x \rightarrow M_2 \mid \mathsf{fix}\, f.\lambda x.M \\
\textbf{Values} & v & ::= & c \mid \lambda x.M \mid (v_1, v_2) \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \mathsf{nil} \mid \mathsf{cons}\, v \\
\textbf{Types} & A, B, C, D & ::= & \alpha \mid \iota \mid A \rightarrow B \mid \forall\, \alpha.\, A \mid A \times B \mid A + B \mid A\, \mathsf{list} \\
\textbf{Evaluation contexts} & E & ::= & [] \mid E\, M_2 \mid v_1\, E \mid \#\mathsf{op}(E) \mid \mathsf{handle}\, E\, \mathsf{with}\, H \mid (E, M_2) \mid (v_1, E) \mid \\
& & & \pi_1 E \mid \pi_2 E \mid \mathsf{inl}\, E \mid \mathsf{inr}\, E \mid \mathsf{case}\, E\, \mathsf{of}\, \mathsf{inl}\, x \rightarrow M_1;\ \mathsf{inr}\, y \rightarrow M_2 \mid \\
& & & \mathsf{cons}\, E \mid \mathsf{case}\, E\, \mathsf{of}\, \mathsf{nil}\ \rightarrow M_1;\ \mathsf{cons}\, x \rightarrow M_2
\end{array}
$$

Reduction rules $\boxed{M_1 \rightsquigarrow M_2}$

$$\pi_1(v_1, v_2) \rightsquigarrow v_1 \qquad \pi_2(v_1, v_2) \rightsquigarrow v_2 \qquad \mathsf{fix}\, f.\lambda x.M \rightsquigarrow (\lambda x.M)[\mathsf{fix}\, f.\lambda x.M/f]$$
$$\mathsf{case}\ \mathsf{inl}\, v\, \mathsf{of}\, \mathsf{inl}\, x \rightarrow M_1;\ \mathsf{inr}\, y \rightarrow M_2 \rightsquigarrow M_1[v/x]$$
$$\mathsf{case}\ \mathsf{inr}\, v\, \mathsf{of}\, \mathsf{inl}\, x \rightarrow M_1;\ \mathsf{inr}\, y \rightarrow M_2 \rightsquigarrow M_2[v/y]$$
$$\mathsf{case}\ \mathsf{nil}\ \mathsf{of}\ \mathsf{nil}\ \rightarrow M_1;\ \mathsf{cons}\, x \rightarrow M_2 \rightsquigarrow M_1 \qquad \mathsf{case}\ \mathsf{cons}\, v\, \mathsf{of}\ \mathsf{nil}\ \rightarrow M_1;\ \mathsf{cons}\, x \rightarrow M_2 \rightsquigarrow M_2[v/x]$$

Type containment $\boxed{\Gamma \vdash A \sqsubseteq B}$

$$\frac{\Gamma \vdash A_1 \sqsubseteq B_1 \quad \Gamma \vdash A_2 \sqsubseteq B_2}{\Gamma \vdash A_1 \times A_2 \sqsubseteq B_1 \times B_2}\ \text{C\_PROD} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \forall\, \alpha.\, A \times B \sqsubseteq (\forall\, \alpha.\, A) \times (\forall\, \alpha.\, B)}\ \text{C\_DPROD}$$

$$\frac{\Gamma \vdash A_1 \sqsubseteq B_1 \quad \Gamma \vdash A_2 \sqsubseteq B_2}{\Gamma \vdash A_1 + A_2 \sqsubseteq B_1 + B_2}\ \text{C\_SUM} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \forall\, \alpha.\, A + B \sqsubseteq (\forall\, \alpha.\, A) + (\forall\, \alpha.\, B)}\ \text{C\_DSUM}$$

$$\frac{\Gamma \vdash A \sqsubseteq B}{\Gamma \vdash A\, \mathsf{list} \sqsubseteq B\, \mathsf{list}}\ \text{C\_LIST} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \forall\, \alpha.\, A\, \mathsf{list} \sqsubseteq (\forall\, \alpha.\, A)\, \mathsf{list}}\ \text{C\_DLIST}$$

Fig. 5. The extended part.

- *The positive (resp. negative) occurrences of α in $A \times B$ are the positive (resp. negative) occurrences of α in A and those in B.*
- *The positive (resp. negative) occurrences of α in $A + B$ are the positive (resp. negative) occurrences of α in A and those in B.*
- *The positive (resp. negative) occurrences of α in A list are the positive (resp. negative) occurrences of α in A.*

The strictly positive occurrences of a type variable in a product, sum, and list type are defined as follows.

- *The strictly positive occurrences of α in $A \times B$ are the strictly positive occurrences of α in A and those in B.*
- *The strictly positive occurrences of α in $A + B$ are the strictly positive occurrences of α in A and those in B.*
- *The strictly positive occurrences of α in A list are the strictly positive occurrences of α in A.*

The signature restriction for the extended language is defined as in Definition 4 except that the polarity of occurrences of type variables is defined by both of Definitions 3 and 5.

We can prove that the extended polymorphic type system satisfies type soundness under the assumption that all operations conform to the signature restriction for the extended language in a similar way as in Section 4.5; refer to the supplementary material for the proof.

5.2 Examples

This section presents two operations that satisfy the signature restriction in the extended language.

The first example is `select`, which is an operation given in Section 2.1 for nondeterministic computation. The operation has the type signature $\forall \alpha. \alpha\ \mathsf{list} \hookrightarrow \alpha$, where the quantified type variable α occurs only at a strictly positive position in the domain type $\alpha\ \mathsf{list}$ and only at a positive position in the codomain type α. Thus, `select` satisfies the signature restriction and, therefore, it can be safely called by any polymorphic expression.

The second example is from Leijen [2017], who implemented parser combinators using algebraic effects and handlers. The effect for parsing provides a basic operation `satisfy` which has the type signature

$$\forall \alpha. (\mathsf{str} \rightarrow (\alpha \times \mathsf{str}) + \mathsf{unit}) \hookrightarrow \alpha$$

where str is the type of strings. This operation takes a parsing function of $\mathsf{str} \rightarrow (\alpha \times \mathsf{str}) + \mathsf{unit}$ such that: the parsing function returns the unit value if an input string does not conform to the grammar; otherwise, it returns the parsing result of α and the unparsed, remaining string. The operation `satisfy` would return the result of parsing if it succeeds. For example, we can give `satisfy` a parsing function that returns the first character of a given input—and returns the unit value if the input is the empty string—as follows:

$$\#\mathsf{satisfy}(\lambda x. \mathsf{if}\ (\mathsf{length}\ x) \ > \ 0\ \mathsf{then}\ \mathsf{inl}\ (\mathsf{first}\ x, \mathsf{last}\ x)\ \mathsf{else}\ \mathsf{inr}\ ()).$$

Here: length is a function of $\mathsf{str} \rightarrow \mathsf{int}$ that returns the length of a given string; first is of $\mathsf{str} \rightarrow \mathsf{char}$ (char is the type of characters) that returns the first character of a given string; and last is of $\mathsf{str} \rightarrow \mathsf{str}$ that returns the same string as an input except that it does not contain the first character of the input. In this example, the call of `satisfy` is of the type char because the argument function is of the type $\mathsf{str} \rightarrow (\mathsf{char} \times \mathsf{str}) + \mathsf{unit}$, which requires the quantified type variable α to be instantiated to char. The operation `satisfy` satisfies the signature restriction clearly. The quantified type variable α occurs only at a strictly positive position in the domain type $\mathsf{str} \rightarrow (\alpha \times \mathsf{str}) + \mathsf{unit}$ of the type signature and it also occurs only at a positive position in the codomain type α.

6 COOPERATION OF SAFE AND UNSAFE EFFECTS

This section describes an effect system for $\lambda^{\mathrm{ext}}_{\mathrm{eff}}$, which enables the type-safe cooperation of safe and unsafe effects in a single program. Our effect system allows expressions to be polymorphic if their evaluation performs only operations that satisfy the signature restriction. This capability makes it possible for the effect system to incorporate value restriction—i.e., any value can be polymorphic. The definition of signature restriction changes to take into account effect information on types. Soundness of the effect system enables us to ensure that programs handle all the operations performed at run time.

Our effect system is inspired by Kammar et al. [2013], where the effect system tracks involved effect operations by their names together with their type signatures. There are, however, two differences between Kammar et al.'s and our effect systems. The first difference comes from that of the evaluation strategies the calculi adopt: the calculus of Kammar et al. is based on call-by-push-value (CBPV) [Levy 2001] and we adopt call-by-value (CBV). This difference influences the design of effect systems because the two strategies have different notions for the value representations of suspended computations and effect systems have to manage the effects caused by their run. CBPV views functions as (not suspended) computations, and thus Kammar et al. did not equip function types with effect information; instead, they augmented the types of thunks (which are value representations of suspended computations in CBPV) with it. By contrast, because CBV views functions as values that represent suspended computations, our effect system equips function types

$$\begin{array}{llll} \textbf{Effects} & \epsilon & ::= & \{\mathsf{op}_1, \cdots, \mathsf{op}_n\} \\ \textbf{Types} & A, B, C, D & ::= & \alpha \mid \iota \mid A \rightarrow^{\epsilon} B \mid \forall \alpha. A \mid A \times B \mid A + B \mid A\,\mathsf{list} \end{array}$$

Type containment $\boxed{\Gamma \vdash A \sqsubseteq B}$

$$\frac{\Gamma \vdash B_1 \sqsubseteq A_1 \quad \Gamma \vdash A_2 \sqsubseteq B_2}{\Gamma \vdash A_1 \rightarrow^{\epsilon} A_2 \sqsubseteq B_1 \rightarrow^{\epsilon} B_2}\;\textsc{C\_FunEff} \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathit{ftv}(A) \quad \mathit{SR}(\epsilon)}{\Gamma \vdash \forall \alpha. A \rightarrow^{\epsilon} B \sqsubseteq A \rightarrow^{\epsilon} \forall \alpha. B}\;\textsc{C\_DFunEff} \quad \cdots$$

Term typing $\boxed{\Gamma \vdash M : A \mid \epsilon}$

$$\frac{\Gamma \vdash M_1 : A \rightarrow^{\epsilon'} B \mid \epsilon \quad \Gamma \vdash M_2 : A \mid \epsilon \quad \epsilon' \subseteq \epsilon}{\Gamma \vdash M_1\, M_2 : B \mid \epsilon}\;\textsc{Te\_App} \qquad \frac{\Gamma, \alpha \vdash M : A \mid \epsilon \quad \mathit{SR}(\epsilon)}{\Gamma \vdash M : \forall \alpha. A \mid \epsilon}\;\textsc{Te\_Gen}$$

$$\frac{\Gamma \vdash M : A \mid \epsilon \quad \Gamma \vdash H : A \mid \epsilon \Rightarrow B \mid \epsilon'}{\Gamma \vdash \mathsf{handle}\, M \,\mathsf{with}\, H : B \mid \epsilon'}\;\textsc{Te\_Handle}$$

$$\frac{\Gamma, f : A \rightarrow^{\epsilon} B, x : A \vdash M : B \mid \epsilon}{\Gamma \vdash \mathsf{fix}\, f.\lambda x.M : A \rightarrow^{\epsilon} B \mid \epsilon'}\;\textsc{Te\_Fix} \qquad \frac{\Gamma \vdash M : A \mid \epsilon' \quad \epsilon' \subseteq \epsilon}{\Gamma \vdash M : A \mid \epsilon}\;\textsc{Te\_Weak} \quad \cdots$$

Handler typing $\boxed{\Gamma \vdash H : A \mid \epsilon \Rightarrow B \mid \epsilon'}$

$$\frac{\Gamma, x : A \vdash M : B \mid \epsilon' \quad \epsilon \subseteq \epsilon'}{\Gamma \vdash \mathsf{return}\, x \rightarrow M : A \mid \epsilon \Rightarrow B \mid \epsilon'}\;\textsc{THe\_Return}$$

$$\frac{\Gamma \vdash H : A \mid \epsilon \Rightarrow B \mid \epsilon' \quad \mathit{ty}(\mathsf{op}) = \forall \boldsymbol{\alpha}. C \hookrightarrow D \quad \Gamma, \boldsymbol{\alpha}, x : C, k : D \rightarrow^{\epsilon'} B \vdash M : B \mid \epsilon'}{\Gamma \vdash H; \mathsf{op}(x, k) \rightarrow M : A \mid \epsilon \uplus \{\mathsf{op}\} \Rightarrow B \mid \epsilon'}\;\textsc{THe\_Op}$$

Fig. 6. The effect system (excerpt).

with effect information. The second difference is that we include only operation names and not their type signatures in the effect information. This is merely for simplifying the presentation but it makes the calculus non-terminating [Kammar and Pretnar 2017].

6.1 Effect System

Figure 6 shows only the key part of the effect system; the full definition is found in the supplementary material.

The type language includes effect information. Effects, ranged over by ϵ, are finite sets of operations. Function types are augmented with effects that may be triggered in applying the functions of those types.

Typing judgments also incorporate effects. A term typing judgment $\Gamma \vdash M : A \mid \epsilon$ asserts that M is a computation that produces a value of A possibly with effect ϵ. A handler typing judgment $\Gamma \vdash H : A \mid \epsilon \Rightarrow B \mid \epsilon'$ asserts that H handles a computation that produces values of A possibly with effect ϵ and the handling produces values of B possibly with effect ϵ'. Type containment judgments $\Gamma \vdash A \sqsubseteq B$ and well-formedness judgments $\vdash \Gamma$ take the same forms as those of the polymorphic type system in Section 4.

Most of the typing rules for terms are almost the same as those of the polymorphic type system except that they take into account effect information. The rule (Te_App) shows how effects are incorporated into the typing rules: the effect triggered by a term is determined by its subterms. Besides, (Te_App) requires effect ϵ' triggered by a function to be a subset of the effect ϵ of the subterms. The rule (Te_Gen) is the key of the effect system, allowing a term to have a polymorphic type if it triggers only safe effects. The safety of an effect ϵ is checked by the predicate $SR(\epsilon)$, which asserts that any operation in ϵ satisfies the signature restriction for the type language in Figure 6; we will formalize $SR(\epsilon)$ after explaining the type containment rules. A byproduct of adopting (Te_Gen) is that the effect system incorporates the value restriction [Tofte 1990] successfully: it allows values to have polymorphic types because the values perform no operation (thus, their effects can be the empty set $\emptyset$) and $SR(\emptyset)$ obviously holds. The rule (Te_Fix) gives any effect ϵ' to the fixed-point operator. This means that the fixed-point operator can be viewed as a pure computation because it only produces a lambda abstraction without triggering effects. The rule (Te_Weak) weakens the effect information of a term.

There are two rules for deriving a handler typing judgment $\Gamma \vdash H : A \,|\, \epsilon \Rightarrow B \,|\, \epsilon'$. They state that the effect of a handle–with expression installing H consists of the operations that the handled expression may call but H does not handle and those that the return clause or some operation clause of H may call. The effect $\epsilon \uplus \{\mathrm{op}\}$ is the same as $\epsilon \cup \{\mathrm{op}\}$ except that it requires $\mathrm{op} \notin \epsilon$.

Most of the type containment rules of the effect system are the same as those of the polymorphic type system. The exception is the rules for function types (C_Fun) and (C_DFun), which are replaced by (C_FunEff) and (C_DFunEff) to take into account effect information. The rule (C_DFunEff) for deriving $\Gamma \vdash \forall\, \alpha.\, A \rightarrow^{\epsilon} B \sqsubseteq A \rightarrow^{\epsilon} \forall\, \alpha.\, B$ has an addition conditional that $SR(\epsilon)$ must hold. This condition originates from (Te_Gen). The rule (C_DFunEff) allows that, if a lambda abstraction $\lambda x.M$ has a polymorphic type $\forall\, \alpha.\, A \rightarrow^{\epsilon} B$, the body M may also have another polymorphic type $\forall\, \alpha.\, B$. In general, M may be a non-value term. In such a case, only (Te_Gen) justifies that M has a polymorphic type; however, to apply (Te_Gen) the effect ϵ triggered by M has to meet $SR(\epsilon)$. This is the reason why (C_DFunEff) requires that $SR(\epsilon)$ hold.

Now, we formalize the predicate $SR(\epsilon)$, which states that any operation in ϵ satisfies signature restriction extended by effect information. In what follows, we suppose the notions of positive/negative/strictly positive occurrences of a type variable for the type language in Figure 6; they are defined naturally as in Definitions 3 and 5. In addition, we can decide whether a type occurs at a strictly positive position in a type by generalizing Definitions 3 and 5 from the occurrences of type variables to those of types.

Definition 6 (Effects satisfying signature restriction). *The predicate $SR(\epsilon)$ holds if and only if, for any* $\mathrm{op} \in \epsilon$ *such that* $ty(\mathrm{op}) = \forall\, \boldsymbol{\alpha}.\, A \hookrightarrow B$*:*

- *the occurrences of each type variable of $\boldsymbol{\alpha}$ in A are only negative or strictly positive;*
- *the occurrences of each type variable of $\boldsymbol{\alpha}$ in B are only positive; and*
- *for any function type $C \rightarrow^{\epsilon'} D$ occurring at a strictly positive position in A, if $\{\boldsymbol{\alpha}\} \cap ftv(D) \neq \emptyset$, then $SR(\epsilon')$.*

The first and second conditions of Definition 6 are the same as those of Definition 4, signature restriction without effect information. The third condition is necessary to apply (C_DFunEff). The signature restriction in the polymorphic type system allows type variables $\boldsymbol{\alpha}$ in type signature $\forall\, \boldsymbol{\alpha}.\, A \hookrightarrow B$ to occur at a strictly positive position in A (see Definition 4). As discussed in Section 4.4.2, this capability originates from (C_DFun). In the effect system, the counterpart (C_DFunEff) is applied to retain this capability, but (C_DFunEff) requires the effect of a function type to satisfy SR. This is the reason why the signature restriction for the effect system imposes the third condition. Note that, if type variables in $\boldsymbol{\alpha}$ do not occur free in D (and they do not in C either), then we can derive $\Gamma \vdash \forall\, \boldsymbol{\alpha}.\, C \rightarrow^{\epsilon'} D \sqsubseteq C \rightarrow^{\epsilon'} \forall\, \boldsymbol{\alpha}.\, D$ without (C_DFunEff). Thus, the third condition does not require $SR(\epsilon')$ if $\{\boldsymbol{\alpha}\} \cap ftv(D) = \emptyset$.

We finally state soundness of the effect system, which ensures that a well-typed program handles all the operations performed at run time. We prove it by progress and subject reduction; their formal statements and proofs are found in the supplementary material.

Theorem 2 (Type soundness). *If $\Delta \vdash M : A \mid \emptyset$ and $M \longrightarrow^{*} M'$ and $M' \not\longrightarrow$, then M' is a value.*

6.2 Example

The effect system allows us to use both safe and unsafe effects in a single program. For example, let us consider the following program (which can be represented in $\lambda_{\mathrm{eff}}^{\mathrm{ext}}$).

```
let f : ∀α.α →{get_id} α = λx. #get_id() x in
let g : ∀α.α →{get_id} α = #select([λx. x; f]) in
if g true then (g 2) + 1 else 0
```

This example would be rejected if we were to enforce all operations to follow the signature restriction as in Section 4 because it uses the unsafe operation `get_id`. By contrast, the effect system accepts it because: the polymorphic expression `λx. #get_id() x` calls no operation and `#select([λx. x; f])` calls only `select`, which satisfies the signature restriction, during the evaluation; therefore, they can have the polymorphic type $\forall \alpha. \alpha \rightarrow^{\{\texttt{get\_id}\}} \alpha$ by (Te_Gen). Note that the effect system still rejects the counterexample given in Section 2.3 because it disallows polymorphic expressions to call operations that do not satisfy the signature restriction, such as `get_id`.

7 RELATED WORK

7.1 Restriction for the Use of Effects in Polymorphic Type Assignment

The problem that type safety is broken in naively combining polymorphic effects and polymorphic type assignment was initially discovered in a language with polymorphic references [Gordon et al. 1979] and later in one with polymorphic control operators [Harper and Lillibridge 1991, 1993b]. Researchers have developed many approaches to reconcile these conflicting features thus far [Appel and MacQueen 1991; Asai and Kameyama 2007; Garrigue 2004; Hoang et al. 1993; Leroy and Weis 1991; Sekiyama and Igarashi 2019; Tofte 1990; Wright 1995].

A major direction shared among them is to prevent the generalization of type variables assigned to an expression if the type variables are used to instantiate polymorphic effects triggered by the expression. Leroy and Weis [1991] called such type variables *dangerous*. The value restriction [Tofte 1990; Wright 1995], which allows only syntactic values to be polymorphic, is justified by this idea because these values trigger no effect and therefore no dangerous type variable exists. Similarly, Asai and Kameyama [2007] and Leijen [2017] allowed only observationally pure expressions to be polymorphic. Tofte [1990] proposed another approach that classifies type variables into applicative ones, which cannot be used to instantiate effects, and imperative ones, which may be used, and allows the generalization of only applicative type variables. Weak polymorphism [Appel and MacQueen 1991; Hoang et al. 1993] extends this idea by assigning a type variable the number of function applications necessary to trigger effects instantiated with the type variable. If the numbers assigned to type variables are positive, effects instantiated with the type variables are not triggered; therefore, they are not dangerous and can be generalized safely. Leroy and Weis [1991] prevented the generalization of dangerous type variables by making the type information of free variables in closures accessible. These approaches focused on a specific effect (especially, the ML-style reference effect) basically, but they can be applied to other effects as well. We prevent the generalization of dangerous type variables by *closing* the type arguments of an operation call at run time, as discussed in Section 2.4. This type transformation is not always acceptable, but we find that it is if the operation satisfies the signature restriction.

Garrigue [2004] proposed the relaxed value restriction, which allows the generalization of type variables assigned to an expression if the type variables occur only at positive positions in the type of the expression. The polarity condition on generalized type variables makes it possible to use the empty type as a surrogate of the type variables and, as a result, prevents instantiating effects with the type variables. The relaxed value restriction is similar to signature restriction in that both utilize the polarity of type variables. In fact, the *strong* signature restriction, introduced in Section 2.4, is explainable by using the empty type zero and subtyping <: for it (i.e., deeming zero a subtype of any type) as in the relaxed value restriction. First, let us recall the key idea of the strong signature restriction: it is to rewrite an operation call $\Lambda\beta_1 \ldots \beta_n . \#\mathsf{op}\{C\}(v)$ for $\mathsf{op} : \forall\alpha. A \hookrightarrow B$ to $\Lambda\beta_1 \ldots \beta_n . \#\mathsf{op}\{\forall\beta_1 \ldots \beta_n . C\}(v)$ to close the type argument C and to use provable type containment judgments $A[C/\alpha] \sqsubseteq A[\forall\beta_1 \ldots \beta_n . C/\alpha]$ and $B[\forall\beta_1 \ldots \beta_n . C/\alpha] \sqsubseteq B[C/\alpha]$ for typing the latter term. We can rephrase this idea with zero, instead of $\forall\beta_1 \ldots \beta_n . C$, as follows: the operation call is rewritten to $\Lambda\beta_1 \ldots \beta_n . \#\mathsf{op}\{\mathsf{zero}\}(v)$ and this term can be typed by using the subtyping judgments $A[C/\alpha] <: A[\mathsf{zero}/\alpha]$ and $B[\mathsf{zero}/\alpha] <: B[C/\alpha]$, which are provable owing to the polarity condition of the strong signature restriction (i.e., α occurs only negatively in A and only positively in B). However, this argument does *not* extend to the (non-strong) signature restriction because it allows the bound type variable α to occur at strictly positive positions in the domain type A and then $A[C/\alpha] <: A[\mathsf{zero}/\alpha]$ no longer holds. Thus, our technical contributions include the findings that the type argument C can be closed by *quantifying* it and that $A[C/\alpha] \sqsubseteq A[\forall\beta_1 \ldots \beta_n . C/\alpha]$ is provable by type containment, where the distributive law plays a key role. This change—which may seem minor perhaps—renders the signature restriction quite permissive.

Sekiyama and Igarashi [2019] followed another line of research: they restricted the definitions of effects instead of their usage. They also employed algebraic effects and handlers to accommodate effect definitions in a programmable way and provided a type system that accepts only effects such that programs do not get stuck even if they are instantiated with dangerous type variables. However, a problem with their work is that all effects have to be safe for any usage and a program cannot use both safe and unsafe effects. Our work, by contrast, provides an effect system that allows the use of both operations that satisfy and do not satisfy the signature restriction—inasmuch as they are performed appropriately. The effect system utilizes the benefit of the signature restriction that it only depends on the type interfaces of effects.

Effect systems have been used to safely introduce effects in polymorphic type assignment thus far. Asai and Kameyama [2007] and Leijen [2017] utilized effect systems for the control operators shift/reset [Danvy and Filinski 1990] and algebraic effects and handlers to ensure that polymorphic expressions are observationally pure, respectively. Kammar and Pretnar [2017] proposed an effect system for *parameterized* algebraic effects, which are declared with type parameters and invoked with type arguments. Unlike polymorphic effects, parameterized effects invoked with different type arguments are deemed different. Kammar and Pretnar utilized the effect system to prevent the generalization of the type variables involved by type arguments of parameterized effects.

7.2 User-Defined Effects

Our work employs algebraic effects and handlers as a technical development to describe a variety of effects. Algebraic effects were originally proposed as a denotational framework to describe the meaning of an effect by separating the interface of an effect, which is given by a set of operations, and its interpretation, which is given by the equational theory over the operations [Plotkin and Power 2003]. Plotkin and Pretnar [2009, 2013] introduced effect handlers in order to represent the semantics of exception handling in an equational theory. The idea of separating an effect interface and its interpretation makes it possible to handle user-defined effects in a modular way and encourages the emergence of languages equipped with algebraic effect handlers, such as Eff [Bauer and Pretnar 2015], Koka [Leijen 2017], Frank [Lindley et al. 2017], Multicore OCaml [Dolan et al. 2017]. We also utilize the separation and restrict only effect interfaces in order to achieve type safety in polymorphic type assignment.

Another approach to user-defined effects is to use control operators, which enable programmers to make access to continuations. Many control operators have been developed thus far—e.g., call/cc [Clinger et al. 1985], control/prompt [Felleisen 1988], shift/reset [Danvy and Filinski 1990], fcontrol/run [Sitaram 1993], and cupto/prompt [Gunter et al. 1995]. These operators are powerful and generic, but, in return for that, it is unsafe to naively combine them with polymorphic type assignment [Harper and Lillibridge 1993b]. They do not provide a means to assign individual effects fine-grained type interfaces. Thus, it is not clear how to apply the idea of signature restriction for the effects implemented by control operators.

Monads can also express the interpretation of an effect in a denotational manner [Moggi 1991] and have been used as a long-established, programmable means for user-defined effects [Peyton Jones and Wadler 1993; Wadler 1992]. Filinski [2010] extracted the essence of monadic effects and proposed a language equipped with a type system and an operation semantics for them. We expect our idea of restriction on effect interfaces to be applicable to monadic effects as well, but for that we would first need to consider how to introduce polymorphic effects into a monadic language because Filinski's language supports parametric effects but not polymorphic effects.

8 CONCLUSION

This work addresses a classic problem with polymorphic effects in polymorphic type assignment. Our key idea is to restrict the type interfaces of effects. We formalize our idea with polymorphic algebraic effects and handlers, propose the signature restriction, which restricts the type signatures of operations by the polarity of occurrences of quantified type variables, and prove that a polymorphic type system is sound if all operations satisfy the signature restriction. We also give an effect system in which operations performed by polymorphic expressions have to satisfy the signature restriction but those performed by monomorphic expressions do not have. This effect system enables us to use both operations that satisfy and do not satisfy the signature restriction in a single program safely.

There are several directions for future work. First, we are interested in analyzing the signature restriction from a more semantic perspective. For example, the semantics of a language with control effects is often given by transformation to continuation-passing style (CPS). It would be interesting to study CPS transformation for implicit polymorphism by taking the signature restriction into account. Another direction would be to apply the signature restriction to evaluation strategies other than call-by-value. Harper and Lillibridge [1993a] showed that polymorphic type assignment and the polymorphic version of the control operator call/cc can be reconciled safely in call-by-name at the cost of expressivity and by changing the timing of type instantiation slightly. However, it is unclear—and we would imagine impossible—whether similar reconcilement is achievable in other strategies such as call-by-need and call-by-push-value. Exporting the idea of signature restriction to other evaluation strategies would be beneficial also for testing the robustness and developing a more in-depth understanding of signature restriction.

ACKNOWLEDGMENTS

We would like to thank Yusuke Matsushita for a fruitful discussion at an early stage of the research and the anonymous reviewers of ICFP 2020 PC and AEC for their close reading and valuable comments. This work was supported in part by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST and JSPS KAKENHI Grant Numbers JP19K20247 (Sekiyama), JP19K20211 (Tsukada), and JP15H05706 (Igarashi).

REFERENCES

Danel Ahman. 2017. *Fibred Computational Effects.* Ph.D. Dissertation. University of Edinburgh. https://danel.ahman.ee/papers/phd-thesis.pdf

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009.* 340–353. https://doi.org/10.1145/1480881.1480925

Andrew W. Appel and David B. MacQueen. 1991. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP 1991, Proceedings.* 1–13. https://doi.org/10.1007/3-540-54444-5_83

Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic Delimited Continuations. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Proceedings.* 239–254. https://doi.org/10.1007/978-3-540-76637-7_16

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *PACMPL* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining proofs and programs in a dependently typed language. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014.* 33–46. https://doi.org/10.1145/2535838.2535883

William D. Clinger, Daniel P. Friedman, and Mitchell Wand. 1985. *A Scheme for a Higher-Level Semantic Algebra.* Cambridge University Press, 237–250.

Youyou Cong and Kenichi Asai. 2018. Handling delimited continuations with dependent types. *PACMPL* 2, ICFP (2018), 69:1–69:31. https://doi.org/10.1145/3236764

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages.* 207–212. https://doi.org/10.1145/582153.582176

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming.* 151–160. https://doi.org/10.1145/91556.91622

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Revised Selected Papers.* 98–117. https://doi.org/10.1007/978-3-319-89719-6_6

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010.* 143–156. https://doi.org/10.1145/1863543.1863566

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013.* 429–442. https://doi.org/10.1145/2500365.2500582

Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1988.* 180–190. https://doi.org/10.1145/73560.73576

Andrzej Filinski. 2010. Monads in action. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010.* 483–494. https://doi.org/10.1145/1706299.1706354

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* 29 (2019), e15. https://doi.org/10.1017/S0956796819000121

Jacques Garrigue. 2004. Relaxing the Value Restriction. In *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Proceedings.* 196–213. https://doi.org/10.1007/978-3-540-24754-8_15

J. Y. Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de Doctorat d'État. Université Paris 7.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF.* Lecture Notes in Computer Science, Vol. 78. Springer. https://doi.org/10.1007/3-540-09724-4

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995.* 12–23. https://doi.org/10.1145/224164.224173

Robert Harper and Mark Lillibridge. 1991. ML with callcc is unsound. Announcement on the types electronic forum. https://www.cis.upenn.edu/~bcpierce/types/archives/1991/msg00034.html

Robert Harper and Mark Lillibridge. 1993a. Explicit Polymorphism and CPS Conversion. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 206–219. https://doi.org/10.1145/158511.158630

Robert Harper and Mark Lillibridge. 1993b. Polymorphic Type Assignment and CPS Conversion. *Lisp and Symbolic Computation* 6, 3-4 (1993), 361–380.

My Hoang, John C. Mitchell, and Ramesh Viswanathan. 1993. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93)*. 15–25. https://doi.org/10.1109/LICS.1993.287604

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013*. 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming* 27 (2017), e7. https://doi.org/10.1017/S0956796816000320

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. 486–499. http://dl.acm.org/citation.cfm?id=3009872

Daniel Leivant. 1983. Polymorphic Type Inference. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, POPL 1983*. 88–98. https://doi.org/10.1145/567067.567077

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. The OCaml system release 4.10: Documentation and userâĂŹs manua. https://caml.inria.fr/pub/docs/manual-ocaml/

Xavier Leroy and Pierre Weis. 1991. Polymorphic Type Inference and Assignment. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*. 291–302. https://doi.org/10.1145/99583.99622

Paul Blain Levy. 2001. *Call-by-push-value.* Ph.D. Dissertation. Queen Mary University of London, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. 500–514. http://dl.acm.org/citation.cfm?id=3009897

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML.* MIT Press.

John C. Mitchell. 1988. Polymorphic Type Inference and Containment. *Inf. Comput.* 76, 2/3 (1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The fire triangle: how to mix substitution, dependent elimination, and effects. *PACMPL* 4, POPL (2020), 58:1–58:28. https://doi.org/10.1145/3371126

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *J. Funct. Program.* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 71–84. https://doi.org/10.1145/158511.158524

Andrew Pitts and Ian Stark. 1998. Operational Reasoning for Functions with Local State. In *Higher Order Operational Techniques in Semantics*, Andrew Gordon and Andrew Pitts (Eds.). Publications of the Newton Institute, Cambridge University Press, 227–273. http://www.inf.ed.ac.uk/~stark/operfl.html

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, Proceedings*. 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*. 408–423. https://doi.org/10.1007/3-540-06859-7_148

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Taro Sekiyama and Atsushi Igarashi. 2017. Stateful manifest contracts. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. 530–544. http://dl.acm.org/citation.cfm?id=3009875

Taro Sekiyama and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings*. 353–380. https://doi.org/10.1007/978-3-030-17184-1_13

Dorai Sitaram. 1993. Handling Control. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*. 147–155. https://doi.org/10.1145/155090.155104

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*. 256–270. https://doi.org/10.1145/2837614.2837655

Jerzy Tiuryn and Pawel Urzyczyn. 1996. The Subtyping Problem for Second-Order Types is Undecidable. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*. 74–85. https://doi.org/10.1109/LICS.1996.561306

Mads Tofte. 1990. Type Inference for Polymorphic References. *Inf. Comput.* 89, 1 (1990), 1–34. https://doi.org/10.1016/0890-5401(90)90018-D

Philip Wadler. 1992. The Essence of Functional Programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1–14. https://doi.org/10.1145/143165.143169

J. B. Wells. 1994. Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94)*. 176–185. https://doi.org/10.1109/LICS.1994.316068

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.* 17, 2 (2007), 215–286. https://doi.org/10.1017/S0956796806006216