JFP **34**, e7, 44 pages, 2024.
doi:10.1017/S0956796824000054

Signature restriction for polymorphic algebraic effects

TARO SEKIYAMA
National Institute of Informatics & SOKENDAI, Tokyo, Japan
(*e-mail:* `tsekiyama@acm.org`)

TAKESHI TSUKADA
Chiba University, Chiba, Japan
(*e-mail:* `tsukada@math.s.chiba-u.ac.jp`)

ATSUSHI IGARASHI
Graduate School of Informatics, Kyoto University, Kyoto, Japan
(*e-mail:* `igarashi@kuis.kyoto-u.ac.jp`)

Abstract

The naive combination of polymorphic effects and polymorphic type assignment has been well known to break type safety. In the literature, there are two kinds of approaches to this problem: one is to restrict how effects are triggered and the other is to restrict how they are implemented. This work explores a new approach to ensuring the safety of the use of polymorphic effects in polymorphic type assignment. A novelty of our work is to restrict *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 ensures type safety of a language equipped with polymorphic effects and unrestricted polymorphic type assignment. We also develop a type-and-effect system to enable the use of both of the operations that satisfy and those that do not satisfy the signature restriction in a single program.

1 Introduction

1.1 Background: Polymorphic type assignment with computational effects

Computational effects are pervasive in programming, including 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 exceptions provide structured operations that make it unnecessary to write boilerplate code. These effects have also been proven convenient in functional programming (Gordon et al., 1979; Wadler, 1992; Peyton Jones and Wadler, 1993).

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 (Xi, 2007; Casinghino et al., 2014; Swamy et al., 2016; Sekiyama and Igarashi, 2017; Ahman, 2017; Cong and Asai, 2018). For program reasoning, the state transitions caused by effectful computations have to be tracked (Pitts and Stark, 1998; Ahmed et al., 2009; Dreyer et al., 2010).

These kinds of gaps between pure and effectful computations are also found in our target: polymorphic type assignment. Although 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 (Milner et al., 1990; Leroy et al., 2020). 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, 1993). 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 (Tofte, 1990; Leroy and Weis, 1991; Appel and MacQueen, 1991; Hoang et al., 1993; Wright, 1995; Garrigue, 2004; Asai and Kameyama, 2007; Kammar and Pretnar, 2017; Sekiyama and Igarashi, 2019). 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 does not restrict the use of effects and, instead, allows only the *safe* effects, i.e., the effects that do not cause programs to get stuck no matter how they are used. Their type system examines the implementations of effects to judge the safety of the effects.

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 implementations in that they have to abstract over types. We discover that the abstract nature of type variables in the type signatures can ensure the safety of the polymorphic effects. Based on this finding, we propose a criterion that decides the safety of an effect only by examining the type signatures of its operations. For example, we can find `raise` safe by this criterion. 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 based on this criterion *signature restriction*.

To formalize our idea, we choose *algebraic effects and handlers* (Plotkin and Pretnar, 2009, 2013) as a means to represent effects. Algebraic effects and handlers are a programming mechanism to accommodate user-defined control effects in a modular way, splitting an effect into an interface (i.e., a set of operations with type signatures) and an interpretation. Since our idea is to restrict effect interfaces, we can incorporate signature restriction into the framework of algebraic effects and handlers 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[1]). This type system allows arbitrary terms (rather than only values) that may invoke effects freely to be polymorphic. Despite the unrestricted use of effects, this type system is sound if signature restriction is enforced. 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 signature restriction can be applied to typecheck programs in which both safe and unsafe polymorphic effects may happen.[2]

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 (Peyton Jones et al., 2007; Dunfield and Krishnaswami, 2013), the proof of type soundness of a calculus equipped with type containment rests on translation to another calculus, such as System F (Reynolds, 1974; Girard,

[1] We mean by full polymorphism that the type constructor ∀ can appear at any position of types, as in System F (Reynolds, 1974; Girard, 1972). Polymorphism is implicit if no type annotations are required, unlike System F.

[2] As we will show in the article, signature restriction is permissive and actually we find no useful effect that invalidates it. However, the universal enforcement of signature restriction *might* 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.

1972).[3] Unlike the prior work, 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 proves soundness of a type system involving type containment without translation.

- 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 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 the value restriction naturally.

We employ implicit full polymorphism and type containment to show type soundness, but either of them makes even type checking undecidable (Wells, 1994; Tiuryn and Urzyczyn, 1996). 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 (Milner, 1978; Damas and Milner, 1982) (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 article (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 (which is introduced in Section 2.4) 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 8.1 for further details.

1.3 Relation to the prior publication

This article revises and extends the paper presented at ICFP'20 (Sekiyama et al., 2020). In summary, the following are added.

- The improvement of certain notations (e.g., for free type variables occurring negatively/positively/strictly positively).
- A discussion about relaxing the signature restriction (Section 4.6.2).
- A discussion about design decisions for adapting the signature restriction to effect systems (Section 6.3).

[3] The translation inserts, as a replacement for type containment, functions that are computationally meaningless but work as type conversion statically.

- A reduction of the restriction on effect implementations (Sekiyama and Igarashi, 2019) to the signature restriction (Section 7).

1.4 Organization

The remainder of this article 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 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 relate the restriction on effect implementations to the signature restriction in Section 7, discuss other related work in Section 8, and conclude in Section 9.

In this article, 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, statements, and proofs are provided in Appendix.

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 article 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, respectively, a store. 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.[4] For simplicity, we fix the list element type to be the type `int` of integers here; the polymorphic version shall be discussed in Section 2.2.

[4] This describes only *intended* semantics; one can also give an *unintended* handler, e.g., one that always returns an integer 42 for a call to `select`. Certain unintended handlers can be excluded in a polymorphic setting, as is shown in Section 2.2.

For example, consider an implementation of a filtering function using these operations:

```
1  effect select : int list ↪ int
2  effect fail   : unit ↪ unit
3
4  let filter (l : int list) (f : int → bool) =
5    handle
6      let x = #select(l) in
7      let _ = if f x then () else #fail() in
8      x
9    with
10     return z → [z]
11     select l → concat (map l (λy. resume y))
12     fail z   → []
13
14 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` $\hookrightarrow$ `int` and `unit` $\hookrightarrow$ `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 *parameter type* and *arity type*, respectively (Plotkin and Pretnar, 2008).

The function `filter` in Lines 4–12 operates `select` and `fail` to filter out the elements of a list `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 6–8) chooses an integer selected from the list `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 10–12, 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 to 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 that there is no result satisfying f.

Unlike exception handling, which discards the continuation of where an exception is raised, 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; Leijen, 2017; Forster et al., 2019). 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 resumes the computation from the point at which `select` was called, with each integer `y` of a given list `l` as a return value of the call to `select`. The handled expression in the example calls `select` only once, so each resumed computation (which is performed under the same handler[5]) returns either a singleton list (produced by the return clause) or the empty list (produced by the clause for `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))`, where the function `concat` concatenates a list of lists and `map` returns a new list obtained by applying a given function to each element of a given list.

More formally, the suspended computation is expressed as a *delimited continuation* (Felleisen, 1988; Danvy and Filinski, 1990), 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 to `select`, the run-time system constructs the following delimited continuation c

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

(where [] is the hole to be filled with a resumption argument), 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 the 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 .
```

[5] In this article, we suppose *deep* effect handlers (Kammar et al., 2013), which remain in captured continuations.

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 → M for op of type signature $A \hookrightarrow B$. Since the operation op is called with an argument of A, the typechecking assigns type A to argument variable x. 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 arity type of the operation 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 ↪ 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 for 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 arity type of select, 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,[6] providing a set of operations with *polymorphic* type signatures. We also call such operations *polymorphic*. We write a polymorphic type signature as $\forall \alpha_1 . \dots \forall \alpha_n . A \hookrightarrow B$, which quantifies the type variables $\alpha_1 \dots \alpha_n$ occurring in the parameter type A and those in the arity type B. To avoid ambiguity, we denote a type signature with a polymorphic parameter type $\forall \alpha . A$ by $(\forall \alpha . A) \hookrightarrow B$.

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 to `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 to 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 arity type, a call to `fail` can be put anywhere, making it possible to put `x` in the else-branch, unlike the monomorphic case in Section 2.1.

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 is unsound

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

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

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 type A (which may refer to the bound type variable α) to variable `x`. 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 versions of the `select` and `fail` example typecheck.

However, this naive extension is unsound under unrestricted polymorphic type assignment. In what follows, we revisit the counterexample given by Sekiyama and Igarashi (2019), which is an analogue to that found by Harper and Lillibridge (1993, 1991) for 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 $\forall\alpha.\alpha \to \alpha$ 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` $\to$ `bool` and of the type `int` $\to$ `int`. Thus, the handled expression is well typed. Turning to the operation clause, since the type signature of `get_id` is $\forall\alpha.$ `unit` $\hookrightarrow (\alpha \to \alpha)$, typechecking first allocates a fresh type variable α and assigns the type `unit` to the argument variable `x`. The signature also requires the arguments of the resumption expressions to have the type $\alpha \to \alpha$, and both arguments `λz1. ... z1` and `λz2. z1` do indeed. The latter function is typed at $\alpha \to \alpha$ because the requirement for 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 to `get_id` in the handled expression. It constructs the following delimited continuation:

$c \stackrel{\text{def}}{=}$

```
handle
  let f = [] in
  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

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

The evaluation of M proceeds as follows.

$$\begin{array}{rcl} M & = & \begin{array}{l}\texttt{handle}\\ \quad\texttt{let f = (}\lambda\texttt{z1. let \_ = } c[\lambda\texttt{z2. z1] in z1) in}\\ \quad\texttt{if (f true) then ((f 0) + 1) else 2}\\ \texttt{with} \ldots\end{array} \\ & \longrightarrow & \texttt{handle if (}\lambda\texttt{z1. let \_ = } c[\lambda\texttt{z2. z1] in z1) true then} \ldots \texttt{ with} \ldots \\ & \longrightarrow & \texttt{handle if (let \_ = } c[\lambda\texttt{z2. true] in true) then} \ldots \texttt{ with} \ldots \end{array}$$

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

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

A standard approach to this problem is to restrict operation calls in polymorphic expressions (Tofte, 1990; Leroy and Weis, 1991; Appel and MacQueen, 1991; Hoang et al., 1993; Wright, 1995; Garrigue, 2004; Asai and Kameyama, 2007). While this kind of approach prevents `#get_id()` from having a polymorphic type, it disallows calls to any polymorphic operation inside polymorphic expressions even when the calls are safe; interested readers can be referred 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. While this approach allows polymorphic expressions to use any polymorphic effect, it requires all effect handlers to meet the proposed typing discipline. That is, effects implemented with effect handlers violating the discipline cannot be used anywhere—even in monomorphic contexts, although the use of polymorphic effects in such contexts should be safe (Tofte, 1990).

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 op : $\forall\alpha.A \hookrightarrow B$ should not have a "bad" occurrence of α in A or 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:

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

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:

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

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 the type variable β.

This analysis suggests that polymorphic operations instantiated with *closed types*, i.e., types without free type variables (especially β here), are safe to be performed 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.[7] However, allowing only instantiation with closed types is too restrictive. For example, it even disallows a function $\lambda x.\,\texttt{\#select}(x)$ wrapping `select` 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 solution to the scope extrusion, we introduce *strong signature restriction*, which requires that, for each polymorphic operation 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

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

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

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

(where the rewritten part is shaded). In M_1', because the type variable α in 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

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

[7] More precisely, (the typing derivation for) the argument may refer to free type variables even when the type of the argument does not. However, we could address this situation successfully by eliminating them with closing type substitution as in Sekiyama and Igarashi (2019).

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:

$$A[C/\alpha] \sqsubseteq A[\forall\beta_1 \ldots \beta_n.\, C/\alpha]$$
$$B[\forall\beta_1 \ldots \beta_n.\, C/\alpha] \sqsubseteq B[C/\alpha]\,,$$

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 $\texttt{\#op}\{\forall\beta_1 \ldots \beta_n.\, C\}$ is applicable to v and we have $\texttt{\#op}\{\forall\beta_1 \ldots \beta_n.\, C\}(v) : B[\forall\beta_1 \ldots \beta_n.\, C/\alpha]$. Again, by subsumption, $\texttt{\#op}\{\forall\beta_1 \ldots \beta_n.\, C\}(v) : B[C/\alpha]$ 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 `select` : $\forall\alpha.\, \alpha\ \texttt{list} \hookrightarrow \alpha$ in Section 2.2 does not satisfy the requirement of the strong signature restriction, which disallows the positive occurrences of bound type variables in parameter types (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 two changes. First, we close the value v by quantifying the type variables $\beta_1, \ldots, \beta_n$. Namely, we further rewrite M'_1 to

$$\texttt{handle let f = } \Lambda\beta_1 \ldots \beta_n.\, \texttt{\#op}\{\, \forall\beta_1 \ldots \beta_n.\, C\}(\Lambda\beta_1 \ldots \beta_n.\, v)\ \texttt{in}\ M\ \texttt{with}\ H\ .$$

Second, we use the following type containment for the parameter type A:

$$\forall\beta_1 \ldots \beta_n.\, A[C/\alpha] \sqsubseteq A[\forall\beta_1 \ldots \beta_n.\, C/\alpha]\,,$$

which is derivable using the polarity conditions of the signature restriction and an additional type containment rule, known as the distributive law:

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

The signature restriction is relaxed enough to accept many useful effects. For example, the type signature of `select` conforms to the signature restriction—α only occurs at a strictly positive position in the parameter type α `list`. As a result, we can ensure the safety of the calls to `select` in polymorphic expressions. Nevertheless, the signature restriction is still sound in that it rejects unsafe operations. For example, `get_id` does not conform to the signature restriction because, in its type signature $\forall\alpha.$ `unit` $\hookrightarrow (\alpha \to \alpha)$, the bound type variable α occurs negatively in the arity type $\alpha \to \alpha$.

The signature restriction has the following advantages over the previous approaches. By contrast with the approaches that restrict the use of effects, it allows polymorphic expressions to invoke any effect meeting the required restriction. By contrast with the approach that restricts effect handlers, the signature restriction is easily adaptable to allow the use of any polymorphic effect in monomorphic contexts, as shown in Section 6. Furthermore, the

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

Fig. 1. Syntax of λ_{eff}.

signature restriction is easy to understand once one admits the notion of polarities of type variables.

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.

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 constructs 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 $\#\mathsf{op}(M)$ with arguments M; and handle–with expressions $\mathsf{handle}\, M\, \mathsf{with}\, H$, which install a handler H to interpret effect operations performed during the evaluation of 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 $\mathsf{return}\, x \rightarrow M$, 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 $\mathsf{op}(x, k) \rightarrow M$, 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. We suppose that a handler may contain at most one operation clause for each operation.

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

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

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}.

Here, we introduce a few notions about syntax; they are standard, and therefore we omit their formal definitions. We write $M_1[M_2/x]$ for the term 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 (namely, closed terms). These relations are defined by the rules shown in Figure 2.

The reduction relation is defined by four rules. The rule (R_CONST) is for constant applications. The semantics of functional constants are given by ζ, which is a partial 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 that 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. It 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-*transparent*, which is formally defined as follows.

Definition 1 (op-transparent evaluation contexts)**.** *Evaluation context E is* op-*transparent, 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.

Type variables	α, β, γ		**Base types**	$\iota ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots$
Types	A, B, C, D	$::=$	$\alpha \mid \iota \mid A \to B \mid \forall\, \alpha. A$	
Typing contexts	Γ	$::=$	$\emptyset \mid \Gamma, x : A \mid \Gamma, \alpha$	

Fig. 3. The type language.

We also denote the operation clause for op in H by $H(\mathsf{op})$. Then, the conjunction of $\mathsf{op} \notin E$ and $H(\mathsf{op}) = \mathsf{op}(x, k) \to M$ in (R_Handle) means that the operation clause $\mathsf{op}(x, k) \to M$ 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 reduction result M_2 of the redex M_1.

4 A polymorphic type system for signature restriction

This section defines a polymorphic type system for λ_{eff} that incorporates subsumption by type containment. 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. Therefore, a well-typed program may terminate at an unhandled operation call; we will present an effect system that can reject programs causing unhandled operation calls in Section 6.

4.1 Type language

Figure 3 presents the type language of the polymorphic type system. It is the same as that of System F (Reynolds, 1974; Girard, 1972) with base types. 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 $A \to B$; and polymorphic types $\forall\, \alpha. A$, where type variable α is bound in type A. Typing contexts, ranged over by Γ, are sequences of bindings of variables coupled with their types and type variables. We suppose that the metafunction ty assigns to each constant c a first-order closed type $ty(c)$ of the form $\iota \to \cdots \to \iota_n$. We state the assumption on the consistency between the types and semantics of constants later (Assumption 1).

We use the following shorthand and notions. We write $\boldsymbol{\alpha}^I$ for $\boldsymbol{\alpha} = \alpha_1, \cdots, \alpha_n$ with the index set $I = \{1, ..., n\}$. 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, J, K) if they are clear from the context or irrelevant. For instance, we may abbreviate $\boldsymbol{\alpha}^I$ to $\boldsymbol{\alpha}$. We also write $\forall\, \boldsymbol{\alpha}^I. A$ for $\forall \alpha_1. ... \forall \alpha_n. A$ with $I = \{1, ..., n\}$. We may omit the index sets and write $\forall\, \boldsymbol{\alpha}. A$ simply. We write $\forall\, \boldsymbol{\alpha}^I \boldsymbol{.}\, \boldsymbol{A}^J$ for a sequence of types $\forall\, \boldsymbol{\alpha}^I. A_1, \ldots, \forall\, \boldsymbol{\alpha}^I. A_n$ with $J = \{1, \ldots, n\}$. The notions of free type variables and capture-avoiding type substitution are defined as usual. We write $\mathit{ftv}(A)$ for the set of free type variables of type A and $A[\boldsymbol{B}/\boldsymbol{\alpha}]$ for the type obtained by substituting each type of the sequence $\boldsymbol{B}$ for the corresponding type variable of the sequence $\boldsymbol{\alpha}$ 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 type A after applying an 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 type A and produces a value of type 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 $\mathit{dom}(\Gamma)$ 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 $\mathit{ftv}(A) \subseteq \mathit{dom}(\Gamma)$ (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 universal quantifier $\forall$ if the quantifier 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), 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 $\forall\alpha. A \to B$ to $A \to \forall\alpha. B$ 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 parameter 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)**.** *The metafunction ty assigns to each effect operation* op *a type signature* $\mathit{ty}(\mathsf{op})$ *of the form* $\forall\alpha_1. \dots \forall\alpha_n. A \hookrightarrow B$ *for some n, where* $\alpha_1, \dots, \alpha_n$ *are*

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

$$\frac{}{\vdash \emptyset}\ \textsc{WF\_Empty} \qquad \frac{x \notin dom(\Gamma) \quad \Gamma \vdash A}{\vdash \Gamma, x : A}\ \textsc{WF\_ExtVar}$$

$$\frac{\alpha \notin dom(\Gamma) \quad \vdash \Gamma}{\vdash \Gamma, \alpha}\ \textsc{WF\_ExtTyVar}$$

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

$$\frac{\vdash \Gamma}{\Gamma \vdash A \sqsubseteq A}\ \textsc{C\_Refl} \qquad \frac{\Gamma \vdash A \sqsubseteq C \quad \Gamma \vdash C \sqsubseteq B}{\Gamma \vdash A \sqsubseteq B}\ \textsc{C\_Trans}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash \forall \alpha. A \sqsubseteq A[B/\alpha]}\ \textsc{C\_Inst} \quad \frac{\vdash \Gamma \quad \alpha \notin ftv(A)}{\Gamma \vdash A \sqsubseteq \forall \alpha. A}\ \textsc{C\_Gen} \quad \frac{\Gamma, \alpha \vdash A \sqsubseteq B}{\Gamma \vdash \forall \alpha. A \sqsubseteq \forall \alpha. B}\ \textsc{C\_Poly}$$

$$\frac{\Gamma \vdash B_1 \sqsubseteq A_1 \quad \Gamma \vdash A_2 \sqsubseteq B_2}{\Gamma \vdash A_1 \to A_2 \sqsubseteq B_1 \to B_2}\ \textsc{C\_Fun} \qquad \frac{\vdash \Gamma \quad \alpha \notin ftv(A)}{\Gamma \vdash \forall \alpha. A \to B \sqsubseteq A \to \forall \alpha. B}\ \textsc{C\_DFun}$$

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

$$\frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{T\_Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash c : ty(c)}\ \textsc{T\_Const} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \textsc{T\_Abs}$$

$$\frac{\Gamma \vdash M_1 : A \to B \quad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1\, M_2 : B}\ \textsc{T\_App} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash A \sqsubseteq B \quad \Gamma \vdash B}{\Gamma \vdash M : B}\ \textsc{T\_Inst}$$

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

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

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

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \mathsf{return}\, x \to M : A \Rightarrow B}\ \textsc{TH\_Return}$$

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

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

bound in the parameter type A and arity 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 $\forall \alpha_1. \ldots \forall \alpha_n. A \hookrightarrow B$ is closed, i.e., $ftv(A), ftv(B) \subseteq \{\alpha_1, \cdots, \alpha_n\}$.

We note that parameter and arity 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 parameter type of the instantiated type signature. We use notation $\Gamma \vdash \boldsymbol{C}$ for the conjunction of $\Gamma \vdash C_1, \cdots, \Gamma \vdash C_n$ when $\boldsymbol{C} = C_1, \cdots, C_n$.

The typing rules for handlers are also ordinary (Plotkin and Pretnar, 2013). A return clause $\mathsf{return}\, x \to M$ 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

$\mathsf{op}(x, k) \to M$ is typechecked by (TH_OP). Let the type signature of op be $\forall \boldsymbol{\alpha}. C \hookrightarrow D$. In typechecking M, variable x is assumed to have the parameter type C since variable x will be bound to the arguments to the operation op. Variable k is given type $D \to B$ where the type 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 arity 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 the type B.

4.3 Desired properties for type soundness

As mentioned in Section 2.3, the polymorphic type system is unsound—a well-typed program can get stuck—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:

$$\mathsf{handle}\, E[\#\mathsf{op}(v)]\, \mathsf{with}\, H \rightsquigarrow M[v/x][\lambda y.\mathsf{handle}\, E[y]\, \mathsf{with}\, H/k] \tag{4.1}$$

where $\mathsf{op} \notin E$ and $H(\mathsf{op}) = \mathsf{op}(x, k) \to M$. 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 two 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 of reduction (4.1). Assume 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

$$\Gamma, \boldsymbol{\alpha}^I, x : A, k : B \to D \vdash M : D \tag{4.2}$$

must have held by the use of the typing rule (TH_OP). Turning to the handled expression $E[\#\mathsf{op}(v)]$, we can find two facts about the typing judgment for the value v. The first fact originates from (T_OP): since the value v is an argument of operation op, it should be of type $A[\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$, which is a type obtained by substituting certain types $\boldsymbol{C}^I$ for type variables

$\boldsymbol{\alpha}^I$ in the parameter type A of the operation op. The second is from (T_GEN), which allows the generalization of types *anywhere*. Thus, the value v is well typed under a typing context $\Gamma, \boldsymbol{\beta}^J$, an extension of Γ with some type variables $\boldsymbol{\beta}^J$ (note that $I \neq J$ in general). In summary, the typing judgment for the value v takes the following form:

$$\Gamma, \boldsymbol{\beta}^J \vdash v : A[\boldsymbol{C}^I/\boldsymbol{\alpha}^I] \,. \tag{4.3}$$

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

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

by the typing derivation of judgment (4.3) and (T_GEN). Proposition 1 enables us to prove

$$\Gamma \vdash v : A[\forall \boldsymbol{\beta}^J . \boldsymbol{C}^I/\boldsymbol{\alpha}^I] \,. \tag{4.4}$$

We can also derive

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

by substituting $\forall \boldsymbol{\beta}^J . \boldsymbol{C}^I$ for $\boldsymbol{\alpha}^I$ in the typing judgment (4.2); 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

$$\Gamma, k : B[\forall \boldsymbol{\beta}^J . \boldsymbol{C}^I/\boldsymbol{\alpha}^I] \rightarrow D \vdash M[v/x] : D \tag{4.5}$$

using an ordinary substitution lemma with the derivation for judgment (4.4).

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

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

is derivable, jointly with the derivation of typing judgment (4.5). Namely, it suffices to derive

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

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

$$\Gamma, y : B\,[\forall \boldsymbol{\beta}^J . \boldsymbol{C}^I/\boldsymbol{\alpha}^I], \boldsymbol{\beta}^J \vdash y : B[\boldsymbol{C}^I/\boldsymbol{\alpha}^I] \,. \tag{4.6}$$

Because $\Gamma, y : B\,[\forall \boldsymbol{\beta}^J . \boldsymbol{C}^I/\boldsymbol{\alpha}^I], \boldsymbol{\beta}^J \vdash y : B[\forall \boldsymbol{\beta}^J . \boldsymbol{C}^I/\boldsymbol{\alpha}^I]$, we can derive

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

by Proposition 2. Furthermore, by instantiating $\forall \boldsymbol{\beta}^J . B[\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$ to $B[\boldsymbol{C}^I/\boldsymbol{\alpha}^I]$ with $\boldsymbol{\beta}^J$ in the typing context, we have succeeded in deriving the typing judgment (4.6).

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 support these properties, but imposing signature restriction produces a type system that does.

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 sets $\mathit{ftv}(A)^+$ and $\mathit{ftv}(A)^-$ of type variables that occur positively and negatively, respectively, in type A are defined by induction on A, as follows.*

$$\begin{array}{lcl} \mathit{ftv}(\alpha)^+ & \stackrel{\text{def}}{=} & \{\alpha\} \\ \mathit{ftv}(\alpha)^- & \stackrel{\text{def}}{=} & \emptyset \\ \mathit{ftv}(A \rightarrow B)^\pm & \stackrel{\text{def}}{=} & \mathit{ftv}(A)^\mp \cup \mathit{ftv}(B)^\pm \\ \mathit{ftv}(\forall\,\alpha.\,A)^\pm & \stackrel{\text{def}}{=} & \mathit{ftv}(A)^\pm \setminus \{\alpha\} \end{array}$$

The set $\mathit{ftv}(A)^+_{\mathbf{ns}}$ of type variables that occur non-strictly positively in type A is defined as follows.

$$\begin{array}{lcl} \mathit{ftv}(\alpha)^+_{\mathbf{ns}} & \stackrel{\text{def}}{=} & \emptyset \\ \mathit{ftv}(A \rightarrow B)^+_{\mathbf{ns}} & \stackrel{\text{def}}{=} & \mathit{ftv}(A)^- \cup \mathit{ftv}(B)^+_{\mathbf{ns}} \\ \mathit{ftv}(\forall\,\alpha.\,A)^+_{\mathbf{ns}} & \stackrel{\text{def}}{=} & \mathit{ftv}(A)^+_{\mathbf{ns}} \setminus \{\alpha\} \end{array}$$

Now, we define signature restriction. We write $\{\boldsymbol{\alpha}^I\}$ to view the sequence $\boldsymbol{\alpha}^I$ as a set by ignoring the order.

Definition 4 (Signature restriction)**.** *An operation* op *with type signature $\mathit{ty}(\mathsf{op}) = \forall\,\boldsymbol{\alpha}.\,A \hookrightarrow B$ satisfies the signature restriction if and only if: (1) the occurrences of each type variable of $\boldsymbol{\alpha}$ in the parameter type A are only negative or strictly positive (i.e., $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(A)^+_{\mathbf{ns}} = \emptyset$) and (2) the occurrences of each type variable of $\boldsymbol{\alpha}$ in the arity type B are only positive (i.e., $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(B)^- = \emptyset$).*

The signature restriction allows quantified type variables to occur at strictly positive positions of the parameter type of a type signature. This is crucial for some operations, such as `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 $\forall\,\alpha.\,\mathsf{unit} \hookrightarrow \alpha \rightarrow \alpha$, the quantified type variable α occurs not only at a positive position but also at a negative position in the arity type $\alpha \rightarrow \alpha$. By contrast, the operations `raise` and `fail` given in Section 2 satisfy the signature restriction because their type signature $\forall\,\alpha.\,\mathsf{unit} \hookrightarrow \alpha$ 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, but, briefly speaking, the type signature $\forall \alpha.\, \alpha\ \mathsf{list} \hookrightarrow \alpha$ of `select` satisfies the signature restriction because the type variable α occurs only strictly positively in the parameter type α list and only positively in the arity type α.

4.4.2 Proofs of the desired properties

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. *Assume that* $\vdash \Gamma$ *and* $\alpha \notin ftv(A)$.

1. *If* $\beta \notin ftv(A)^{+}_{\mathbf{ns}}$, *then* $\Gamma \vdash \forall \alpha. A[B/\beta] \sqsubseteq A[\forall \alpha. B/\beta]$.
2. *If* $\beta \notin ftv(A)^{-}$, *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: assume $ty(\mathsf{op}) = \forall \boldsymbol{\alpha}^{I}. A \hookrightarrow B$ and $\Gamma \vdash M : \forall \boldsymbol{\beta}^{J}. A[\boldsymbol{C}^{I}/\boldsymbol{\alpha}^{I}]$; 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).

We start by proving that any type A can be regarded as a type constructor contravariant and covariant with respect to a type variable α if α occurs only negatively and positively, respectively, in A.

Lemma 2. *Assume that* $\Gamma \vdash B \sqsubseteq C$.

1. *If* $\alpha \notin ftv(A)^{+}$, *then* $\Gamma \vdash A[C/\alpha] \sqsubseteq A[B/\alpha]$.
2. *If* $\alpha \notin ftv(A)^{-}$, *then* $\Gamma \vdash A[B/\alpha] \sqsubseteq A[C/\alpha]$.

Proof Straightforward by induction on A. ■

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

Proof of Lemma 1 (2). By (C_TRANS), it suffices to show that $\Gamma \vdash A[\forall \alpha. B/\beta] \sqsubseteq \forall \alpha. A[\forall \alpha. B/\beta]$ and $\Gamma \vdash \forall \alpha. A[\forall \alpha. B/\beta] \sqsubseteq \forall \alpha. A[B/\beta]$. The former is derived by (C_GEN). The latter is derived as follows, where we have $\vdash \Gamma, \alpha$ because we can assume that $\alpha \notin dom(\Gamma)$ without loss of generality:

$$\dfrac{\dfrac{\dfrac{\vdash \Gamma, \alpha}{\Gamma, \alpha \vdash \forall \alpha. B \sqsubseteq B}\ (\text{C\_INST})}{\Gamma, \alpha \vdash A[\forall \alpha. B/\beta] \sqsubseteq A[B/\beta]}\ \text{by Lemma 2}}{\Gamma \vdash \forall \alpha. A[\forall \alpha. B/\beta] \sqsubseteq \forall \alpha. A[B/\beta]}\ (\text{C\_POLY})$$

■

Proof of Lemma 1 (1). By induction on type A. Here, we consider only the interesting cases: A is a function type or a polymorphic type; see the supplementary material for the other cases.

Case $A = C \rightarrow D$ for some C and D: By the assumption $\beta \notin \mathit{ftv}(A)^{+}_{\mathbf{ns}} = \mathit{ftv}(C \rightarrow D)^{+}_{\mathbf{ns}}$, we have $\beta \notin \mathit{ftv}(C)^{-}$ and $\beta \notin \mathit{ftv}(D)^{+}_{\mathbf{ns}}$. By (C_Trans), it suffices to show the following two type containment judgments:

$$\Gamma \vdash \forall \alpha.\, C[B/\beta] \rightarrow D[B/\beta] \sqsubseteq (\forall \alpha.\, C[B/\beta]) \rightarrow \forall \alpha.\, D[B/\beta]$$
$$\Gamma \vdash (\forall \alpha.\, C[B/\beta]) \rightarrow \forall \alpha.\, D[B/\beta] \sqsubseteq C[\forall \alpha.\, B/\beta] \rightarrow D[\forall \alpha.\, B/\beta]\,.$$

The former is derived using (C_Trans) with the following two derivations:

$$\dfrac{\dfrac{\dfrac{\vdash \Gamma, \alpha}{\Gamma, \alpha \vdash \forall \alpha.\, C[B/\beta] \sqsubseteq C[B/\beta]}\,(\textsc{C\_Inst}) \qquad \dfrac{\vdash \Gamma, \alpha}{\Gamma, \alpha \vdash D[B/\beta] \sqsubseteq D[B/\beta]}\,(\textsc{C\_Refl})}{\Gamma, \alpha \vdash C[B/\beta] \rightarrow D[B/\beta] \sqsubseteq (\forall \alpha.\, C[B/\beta]) \rightarrow D[B/\beta]}\,(\textsc{C\_Fun})}{\Gamma \vdash \forall \alpha.\, C[B/\beta] \rightarrow D[B/\beta] \sqsubseteq \forall \alpha.\, (\forall \alpha.\, C[B/\beta]) \rightarrow D[B/\beta]}\,(\textsc{C\_Poly})$$

and

$$\dfrac{\vdash \Gamma \qquad \alpha \notin \mathit{ftv}(\forall \alpha.\, C[B/\beta])}{\Gamma \vdash \forall \alpha.\, (\forall \alpha.\, C[B/\beta]) \rightarrow D[B/\beta] \sqsubseteq (\forall \alpha.\, C[B/\beta]) \rightarrow \forall \alpha.\, D[B/\beta]}\,(\textsc{C\_DFun})\quad .$$

The latter is derived as follows:

$$\dfrac{\dfrac{\beta \notin \mathit{ftv}(C)^{-}}{\Gamma \vdash C[\forall \alpha.\, B/\beta] \sqsubseteq \forall \alpha.\, C[B/\beta]}\,\text{by Lemma 1 (2)} \quad \dfrac{\beta \notin \mathit{ftv}(D)^{+}_{\mathbf{ns}}}{\Gamma \vdash \forall \alpha.\, D[B/\beta] \sqsubseteq D[\forall \alpha.\, B/\beta]}\,\text{by the IH}}{\Gamma \vdash (\forall \alpha.\, C[B/\beta]) \rightarrow \forall \alpha.\, D[B/\beta] \sqsubseteq C[\forall \alpha.\, B/\beta] \rightarrow D[\forall \alpha.\, B/\beta]}\,(\textsc{C\_Fun})$$

Note that $\alpha \notin \mathit{ftv}(C) \cup \mathit{ftv}(D) = \mathit{ftv}(A)$.

Case $A = \forall \gamma.\, C$ for some γ and C: Without loss of generality, we can assume that $\gamma \notin \{\alpha, \beta\} \cup \mathit{dom}(\Gamma) \cup \mathit{ftv}(B)$. Because $\beta \notin \mathit{ftv}(A)^{+}_{\mathbf{ns}}$, we have $\beta \notin \mathit{ftv}(C)^{+}_{\mathbf{ns}}$. Furthermore, $\vdash \Gamma, \gamma$ as $\vdash \Gamma$, and $\alpha \notin \mathit{ftv}(C)$ as $\alpha \notin \mathit{ftv}(A)$. Thus, by the IH, $\Gamma, \gamma \vdash \forall \alpha.\, C[B/\beta] \sqsubseteq C[\forall \alpha.\, B/\beta]$. By (C_Poly), $\Gamma \vdash \forall \gamma. \forall \alpha.\, C[B/\beta] \sqsubseteq \forall \gamma.\, C[\forall \alpha.\, B/\beta]$. Because the type containment allows the commutation of universal quantifiers (see Lemma 9 in the supplementary material for detail), we have $\Gamma \vdash \forall \alpha. \forall \gamma.\, C[B/\beta] \sqsubseteq \forall \gamma. \forall \alpha.\, C[B/\beta]$. Thus, by (C_Trans), $\Gamma \vdash \forall \alpha. \forall \gamma.\, C[B/\beta] \sqsubseteq \forall \gamma.\, C[\forall \alpha.\, B/\beta]$, that is, $\Gamma \vdash \forall \alpha.\, A[B/\beta] \sqsubseteq A[\forall \alpha.\, B/\beta]$.

■

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.

Type containment is thus a key notion to prove type soundness, but proving its inversion properties is complicated. In the literature (Peyton Jones et al., 2007; Dunfield and

Krishnaswami, 2013), type soundness of a language with subtyping based on type containment 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 most difficult property to prove is the inversion of type containment judgments relating function types.

Lemma 3 (Type containment inversion: monomorphic function types). *If* $\Gamma \vdash A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2$*, 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 $\Gamma \vdash A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2$ because a premise in the derivation may relate the (monomorphic) 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: $\Gamma \vdash \forall \boldsymbol{\alpha}^I. A_1 \rightarrow A_2 \sqsubseteq \forall \boldsymbol{\beta}^J. B_1 \rightarrow B_2$. 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 $\boldsymbol{C}_{\mathbf{0}}^{I_{03}}$. 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 $\{\boldsymbol{\alpha}^I\} \uplus \{\boldsymbol{\beta}^J\}$ for the union of disjoint sets $\{\boldsymbol{\alpha}^I\}$ and $\{\boldsymbol{\beta}^J\}$.

Lemma 4 (Type containment inversion: polymorphic 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* $\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}$, $\boldsymbol{\alpha}_{\mathbf{2}}^{I_2}$, $\boldsymbol{\gamma}^K$*, and* $\boldsymbol{C}^{I_1}$ *such that*

- $\{\boldsymbol{\alpha}^I\} = \{\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}\} \uplus \{\boldsymbol{\alpha}_{\mathbf{2}}^{I_2}\}$,
- $\Gamma, \boldsymbol{\beta}^J, \boldsymbol{\gamma}^K \vdash \boldsymbol{C}^{I_1}$,
- $\Gamma, \boldsymbol{\beta}^J \vdash B_1 \sqsubseteq \forall \boldsymbol{\gamma}^K. A_1[\boldsymbol{C}^{I_1}/\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}]$,
- $\Gamma, \boldsymbol{\beta}^J \vdash \forall \boldsymbol{\alpha}_{\mathbf{2}}^{I_2}. \forall \boldsymbol{\gamma}^K. A_2[\boldsymbol{C}^{I_1}/\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}] \sqsubseteq B_2$*, and*
- $\{\boldsymbol{\gamma}^K\} \cap (\mathit{ftv}(A_1) \cup \mathit{ftv}(A_2)) = \emptyset$.

In the statement, the sequence $\boldsymbol{\alpha}_{\mathbf{2}}^{I_2}$ corresponds to $\boldsymbol{\alpha}_{\mathbf{02}}^{I_{02}}$ in the above informal description and $\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}$ includes the type variables that remain in $\boldsymbol{\beta}^J$ (i.e., $\boldsymbol{\alpha}_{\mathbf{01}}^{I_{01}}$) and those instantiated with some types in $\boldsymbol{C}^{I_1}$ (i.e., $\boldsymbol{\alpha}_{\mathbf{03}}^{I_{03}}$). Type substitution $[\boldsymbol{C}^{I_1}/\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}]$ replaces a type variable in $\boldsymbol{\alpha}_{\mathbf{01}}^{I_{01}}$ with itself. We omit the detailed proof of Lemma 4 from the article because of its complexity; interested readers are referred to the supplementary material.

Now, we show progress, subject reduction, and type soundness. In what follows, the metavariable Δ ranges over typing contexts that consist only of 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. For proving progress and subject reduction, we assume the consistency between the type and semantics of every functional constant as follows.

Assumption 1. *We assume the following for any constants c_1 and c_2: (1) $\zeta(c_1, c_2)$ is defined if and only if $ty(c_1) = \iota \rightarrow A$ and $ty(c_2) = \iota$ for some ι and A; and (2) if $\zeta(c_1, c_2)$ is defined, $\zeta(c_1, c_2)$ is a constant and $ty(\zeta(c_1, c_2)) = A$ where $ty(c_1) = \iota \rightarrow A$ for some ι and A.*

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

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

Lemma 6 (Subject reduction)**.** *Assume 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 $\Delta \vdash M_2 : A$.*

Proof By induction on the typing derivation for M_1. The only interesting cases are for function applications and handle–with expressions. The case for function applications uses Lemma 4 to relate the type information of function bodies to that of function applications. The case for handle–with expressions is proven as described in Section 4.3 with Lemma 1. We refer to the supplementary material for the details of the proof. ■

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)**.** *Assume 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*
- *$M' = E[\#\mathsf{op}(v)]$ for some E, op, and v such that $\mathsf{op} \notin E$.*

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

4.6 Is it possible to relax the signature restriction further?

It is natural to ask whether the signature restriction can be further relaxed. This section examines the condition of strictly positive occurrences of quantified type variables on parameter types (Section 4.6.1) and other criteria for the cases that, given a type signature, its bound type variables occur only in the parameter type (Section 4.6.2) or in the arity type (Section 4.6.3).

4.6.1 The condition of strictly positive occurrences is necessary

Consider a type signature $\forall\, \alpha. A \hookrightarrow B$. We show that a nonstrictly positive occurrence of α in the parameter type A is problematic. 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

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

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.

4.6.2 Further relaxation for closed arity types is possible

Consider an operation op : $\forall\alpha. A \hookrightarrow B$ where the type variable α does not appear in the arity type B. This operation can be regarded as being safe even if α occurs nonstrictly positively in the parameter type A. This is justified by a simple program transformation, regarding the polymorphic operation op : $\forall\alpha. A \hookrightarrow B$ as a monomorphic operation op' : $(\exists\alpha.A) \hookrightarrow B$.[8] To justify this translation, a key observation is that, if α has no occurrence in a type D, the polymorphic function type $\forall\alpha.\, C \to D$ is isomorphic to the type $(\exists\alpha.\, C) \to D$ in the sense that a term of the former type also has the latter type.

- For a caller of the operation, the operation op is seen as a term of type $\forall\alpha. A \to B$; the translation replaces a term op of a type $\forall\alpha. A \to B$ by another term op' of an isomorphic type $(\exists\alpha.A) \to B$ and hence preserves the typability of the caller.
- A handler of op : $\forall\alpha. A \to B$ is essentially a term of type $\forall\alpha. A \to ((B \to C) \to C)$ where C is the output type of the handler, which has no occurrence of α, and $B \to C$ is the type of continuations. Similarly, a handler of op' is a term of type $(\exists\alpha.A) \to ((B \to C) \to C)$. Since the two types for handlers are isomorphic, the translation preserves the typability of handlers as well.

In general, given an operation op : $\forall\beta_1 \ldots \beta_m.\forall\alpha_1 \ldots \alpha_n. A \hookrightarrow B$ where the type variables $\alpha_1, \ldots, \alpha_n$ do not appear in the arity type B, it suffices for the safety of the operation to check whether $\forall\beta_1 \ldots \beta_m.(\exists\alpha_1 \ldots \alpha_n. A) \hookrightarrow B$ satisfies the signature restriction.

4.6.3 Further relaxation for closed parameter types is harder

One may expect that the safety of an operation op : $\forall\alpha. A \hookrightarrow B$ where the type variable α does not appear in the parameter type A is reducible to the safety of op' : $A \hookrightarrow (\forall\alpha.B)$.

[8] We assume a calculus extended with existential types here and in Section 7.

Unfortunately, this is not true. A counterexample is $\texttt{get\_id} : \forall\alpha.\,\texttt{unit} \hookrightarrow (\alpha \to \alpha)$ and $\texttt{get\_id'} : \texttt{unit} \hookrightarrow (\forall\alpha.\alpha \to \alpha)$, of which the former is unsafe as we have seen but the latter is safe by the signature restriction. Actually, the translation replacing op with op' does not preserve the typability of the handler. A handler of op can be seen as a term of type

$$\forall\alpha.\,A \to ((B \to C) \to C),$$

where the type C is the output type of the handler, the type $B \to C$ is of continuations, and the type variable α never occurs in the output type C. This type is isomorphic to

$$A \to \forall\alpha.((B \to C) \to C)$$

and

$$A \to (\exists\alpha.(B \to C)) \to C$$

but differs from the type for a handler for op'

$$A \to ((\forall\alpha.B) \to C) \to C;$$

the latter is stronger than the former, i.e., a term of the former type does not necessarily have the latter type, although the converse implication holds.

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 Figures 5 and 6, 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; 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 augmented. 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 6 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.

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

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

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

Fig. 5. The extended part of the syntax and semantics.

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

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

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

$$
\frac{\Gamma \vdash A \sqsubseteq B}{\Gamma \vdash A\ \mathsf{list} \sqsubseteq B\ \mathsf{list}}\ \textsc{C\_List}
\qquad
\frac{\vdash \Gamma}{\Gamma \vdash \forall\,\alpha.\,A\ \mathsf{list} \sqsubseteq (\forall\,\alpha.\,A)\ \mathsf{list}}\ \textsc{C\_DList}
$$

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

$$
\frac{\Gamma \vdash M_1 : A \quad \Gamma \vdash M_2 : B}{\Gamma \vdash (M_1, M_2) : A \times B}\ \textsc{T\_Pair}
\quad
\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1 M : A}\ \textsc{T\_Proj1}
\quad
\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2 M : B}\ \textsc{T\_Proj2}
$$

$$
\frac{\Gamma \vdash M : A \quad \Gamma \vdash B}{\Gamma \vdash \mathsf{inl}\ M : A + B}\ \textsc{T\_InL}
\qquad
\frac{\Gamma \vdash M : B \quad \Gamma \vdash A}{\Gamma \vdash \mathsf{inr}\ M : A + B}\ \textsc{T\_InR}
$$

$$
\frac{\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash M_1 : C \quad \Gamma, y : B \vdash M_2 : C}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \rightarrow M_1;\ \mathsf{inr}\ y \rightarrow M_2 : C}\ \textsc{T\_Case}
$$

$$
\frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{nil} : A\ \mathsf{list}}\ \textsc{T\_Nil}
\quad
\frac{\Gamma \vdash M : A \times A\ \mathsf{list}}{\Gamma \vdash \mathsf{cons}\ M : A\ \mathsf{list}}\ \textsc{T\_Cons}
\quad
\frac{\Gamma, f : A \rightarrow B, x : A \vdash M : B}{\Gamma \vdash \mathsf{fix}\ f.\lambda x.M : A \rightarrow B}\ \textsc{T\_Fix}
$$

$$
\frac{\Gamma \vdash M : A\ \mathsf{list} \quad \Gamma \vdash M_1 : B \quad \Gamma, x : A \times A\ \mathsf{list} \vdash M_2 : B}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \mathsf{nil}\ \rightarrow M_1;\ \mathsf{cons}\ x \rightarrow M_2 : B}\ \textsc{T\_CaseList}
$$

Fig. 6. The extended part of the type system.

Remark 1. *The type containment rule* (C_DSUM) *in Figure 6 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 defined as in Definition 3 and that in product, sum, and list types is defined as follows.

Definition 5 (Polarity of type variable occurrence in product, sum, and list types)**.** *The sets $ftv(A)^+$ and $ftv(A)^-$ of type variables that occur positively and negatively, respectively, in type A are extended to product types, sum types, and list types, as follows.*

$$\begin{aligned} ftv(A \times B)^{\pm} &\stackrel{\text{def}}{=} ftv(A)^{\pm} \cup ftv(B)^{\pm} \\ ftv(A + B)^{\pm} &\stackrel{\text{def}}{=} ftv(A)^{\pm} \cup ftv(B)^{\pm} \\ ftv(A\ \mathsf{list})^{\pm} &\stackrel{\text{def}}{=} ftv(A)^{\pm} \end{aligned}$$

The set $ftv(A)^+_{\mathbf{ns}}$ of type variables that occur nonstrictly positively in type A is extended as follows.

$$\begin{aligned} ftv(A \times B)^{+}_{\mathbf{ns}} &\stackrel{\text{def}}{=} ftv(A)^{+}_{\mathbf{ns}} \cup ftv(B)^{+}_{\mathbf{ns}} \\ ftv(A + B)^{+}_{\mathbf{ns}} &\stackrel{\text{def}}{=} ftv(A)^{+}_{\mathbf{ns}} \cup ftv(B)^{+}_{\mathbf{ns}} \\ ftv(A\ \mathsf{list})^{+}_{\mathbf{ns}} &\stackrel{\text{def}}{=} ftv(A)^{+}_{\mathbf{ns}} \end{aligned}$$

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. The type soundness of the extended language can be proven, as in Section 4.5, under the assumption that all operations conform to the signature restriction extended to product, sum, and list types. See the supplementary material for detail.

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 parameter type $\alpha\ \mathsf{list}$ and only at a positive position in the arity 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 parameter 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 arity 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—because values perform no operation. The definition of signature restriction changes to take into account effect information on types. Soundness of the effect system ensures 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 invoked 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 reason about 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 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 nonterminating (Kammar and Pretnar, 2017).

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

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

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

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

$$\frac{\Gamma \vdash M_1 : A \to^{\varepsilon'} B \mid \varepsilon \quad \Gamma \vdash M_2 : A \mid \varepsilon \quad \varepsilon' \subseteq \varepsilon}{\Gamma \vdash M_1\, M_2 : B \mid \varepsilon}\ \textsc{Te\_App}$$

$$\frac{\Gamma, \alpha \vdash M : A \mid \varepsilon \quad \mathit{SR}(\varepsilon)}{\Gamma \vdash M : \forall \alpha. A \mid \varepsilon}\ \textsc{Te\_Gen}$$

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

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

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

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

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

Fig. 7. The effect system (excerpt).

6.1 Effect system

Figure 7 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 $\Gamma \vdash M : A \mid \varepsilon$ asserts that M is a computation that produces a value of A possibly with effect ε. A handler typing judgment $\Gamma \vdash H : A \mid \varepsilon \Rightarrow B \mid \varepsilon'$ 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 $SR\,(\varepsilon)$, which asserts that any operation in ε satisfies the signature restriction for the type language in Figure 7; we will formalize $SR\,(\varepsilon)$ after explaining the type containment rules. A by-product 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 $\Gamma \vdash H : A \mid \varepsilon \Rightarrow B \mid \varepsilon'$. 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 $\varepsilon \uplus \{\mathsf{op}\}$ is the same as $\varepsilon \cup \{\mathsf{op}\}$ except that it requires $\mathsf{op} \notin \varepsilon$.

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 $\Gamma \vdash \forall\,\alpha.\,A \rightarrow^{\varepsilon} B \sqsubseteq A \rightarrow^{\varepsilon} \forall\,\alpha.\,B$ has an additional condition that $SR\,(\varepsilon)$ 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 $\forall\,\alpha.\,A \rightarrow^{\varepsilon} B$, the body M may also have another polymorphic type $\forall\,\alpha.\,B$. In general, M may be a nonvalue 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 $SR\,(\varepsilon)$. This is the reason why (C_DFUNEFF) requires that $SR\,(\varepsilon)$ hold.

Now, we formalize the predicate $SR\,(\varepsilon)$, which states that any operation in ε satisfies signature restriction extended by effect information. In what follows, we suppose the notions of positive/negative/nonstrictly positive occurrences of a type variable for the type language in Figure 7; 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 the polarity of the occurrences of type variables to those of types.

Definition 6 (Signature restriction on effects). *The predicate $SR\,(\varepsilon)$ holds if and only if, for any $\mathsf{op} \in \varepsilon$ such that $ty\,(\mathsf{op}) = \forall\,\boldsymbol{\alpha}.\,A \hookrightarrow B$:*

- $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(A)^{+}_{\mathbf{ns}} = \emptyset$;
- $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(B)^{-} = \emptyset$; *and*
- *for any function type $C \rightarrow^{\varepsilon'} D$ occurring at a strictly positive position in the type A, if $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(D) \neq \emptyset$, then $SR\,(\varepsilon')$.*

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 the type 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 $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(D) = \emptyset$ (as well as $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(C) = \emptyset$), then we can derive $\Gamma \vdash \forall \boldsymbol{\alpha}. C \rightarrow^{\varepsilon'} D \sqsubseteq C \rightarrow^{\varepsilon'} \forall \boldsymbol{\alpha}. D$ without (C_DFunEff). Thus, the third condition does not require $\mathit{SR}(\varepsilon')$ if $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}(D) = \emptyset$.

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 $\lambda_{\text{eff}}^{\text{ext}}$).

```
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 (because it is a value) 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`.

6.3 Alternative designs

When defining the effect system presented thus far, we kept simplicity in mind, which affected the design of the effect system. Specifically, we decided that 1) the type signatures of operations are assumed to be assigned globally and 2) type containment does not involve subeffecting (it is used only for type instantiation). This section informally discusses the alternatives of these decisions: *local assignment of type signatures* and *type containment with subeffecting*.

6.3.1 Local assignment of type signatures

In the formalized system, effects are sets only of operations, and their type signatures are uniquely determined by metafunction ty. We call this assignment of type signatures *global*.

The global type signature assignment models the treatment of effects in programming languages, such as Koka (Leijen, 2023) and Frank (Lindley et al., 2022), where every effect operation has to be declared with a type signature before used.

An alternative is to represent effects as sets of *pairs of an operation and its type signature* (Kammar et al., 2013), as

$$\varepsilon_{\mathrm{local}} ::= \{\mathsf{op}_1 : \forall \boldsymbol{\alpha}_{\mathbf{1}}^{I_1}. A_1 \hookrightarrow B_1, \;\ldots,\; \mathsf{op}_n : \forall \boldsymbol{\alpha}_{\boldsymbol{n}}^{I_n}. A_n \hookrightarrow B_n\}\,.$$

In this representation, we can assign different type signatures to different occurrences of the same operation. Hence, we call this type assignment *local*. For example, consider the following function application:

$$(\mathsf{handle}\ \#\mathsf{op}(1)\ \mathsf{with}\ H_1)\ (\mathsf{handle}\ \#\mathsf{op}(\mathsf{true})\ \mathsf{with}\ H_2)$$

where the function part supposes operation op's type signature to be $\mathsf{int} \hookrightarrow A$ for some type A, while the argument part supposes it to be $\mathsf{bool} \hookrightarrow B$ for some type B. The handlers H_1 and H_2 should involve operation clauses conforming to these type signatures, respectively. More interestingly, the same operation can be given different type signatures per occurrence even in the same effect. For example, consider the following effect:

$$\{\mathsf{op} : A_1 \hookrightarrow (B_1 \rightarrow^{\{\mathsf{op}:C_1 \hookrightarrow C_2\}} B_2)\}\,.$$

It states that operation op returns a function of the type $B_1 \rightarrow^{\{\mathsf{op}:C_1 \hookrightarrow C_2\}} B_2$, that is, when the returned function is applied and then invokes operation op with an argument of type C_1, the caller expects the operation to return a value of type C_2. One critical difference between the local and global assignment of type signatures is in program termination: one can write divergent programs under the global assignment, whereas the local assignment can guarantee well-typed programs terminating (Kammar and Pretnar, 2017). The local assignment is employed by, e.g., the links programming language (The Links team, 2022).

To adapt the signature restriction to the local type signature assignment, we need to make changes to the definitions of the polarities of type variables, type containment's distributive law, and type generalization, in addition to a minor change that the type signature of an operation is found in an effect, not assigned by the metafunction *ty*. We describe the details of the three major changes below, but it is left open whether only these changes (including the minor one) can ensure soundness of an effect system that assigns type signatures locally.

Change 1: Polarities of type variables. Effects in the local assignment may contain free type variables. Therefore, we need to extend the polarities of type variables to take effects into account, as follows:

$$\begin{array}{lcl}
ftv(A \rightarrow^{\varepsilon_{\mathrm{local}}} B)^{\pm} & \stackrel{\mathrm{def}}{=} & ftv(A)^{\mp} \cup ftv(B)^{\pm} \cup ftv(\varepsilon_{\mathrm{local}})^{\pm} \\
ftv(\varepsilon_{\mathrm{local}})^{\pm} & \stackrel{\mathrm{def}}{=} & ((ftv(A_1)^{\mp} \cup ftv(B_1)^{\pm}) \setminus \{\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}\}) \cup \ldots \cup \\
 & & ((ftv(A_n)^{\mp} \cup ftv(B_n)^{\pm}) \setminus \{\boldsymbol{\alpha}_{\boldsymbol{n}}^{I_n}\}) \\
 & & \\
ftv(A \rightarrow^{\varepsilon_{\mathrm{local}}} B)^{+}_{\mathbf{ns}} & \stackrel{\mathrm{def}}{=} & ftv(A)^{-} \cup ftv(B)^{+}_{\mathbf{ns}} \cup ftv(\varepsilon_{\mathrm{local}})^{+}_{\mathbf{ns}} \\
ftv(\varepsilon_{\mathrm{local}})^{+}_{\mathbf{ns}} & \stackrel{\mathrm{def}}{=} & ((ftv(A_1)^{-} \cup ftv(B_1)^{+}_{\mathbf{ns}}) \setminus \{\boldsymbol{\alpha}_{\mathbf{1}}^{I_1}\}) \cup \ldots \cup \\
 & & ((ftv(A_n)^{-} \cup ftv(B_n)^{+}_{\mathbf{ns}}) \setminus \{\boldsymbol{\alpha}_{\boldsymbol{n}}^{I_n}\})
\end{array}$$

where $\varepsilon_{\mathrm{local}}$ is supposed to be $\{\mathsf{op}_1 : \forall \boldsymbol{\alpha}_{\mathbf{1}}^{I_1}. A_1 \hookrightarrow B_1, \;\ldots,\; \mathsf{op}_n : \forall \boldsymbol{\alpha}_{\boldsymbol{n}}^{I_n}. A_n \hookrightarrow B_n\}$.

Change 2: Type containment's distributive law. The distributive law can be modified so that $\forall$ quantifying a function type moves not only to its codomain type but also to the arity types of the type signatures in its latent effect. Namely, the adapted distributive law can be given as

$$\frac{\begin{array}{c}\varepsilon_{\mathrm{local}} = \{\mathsf{op}_1 : \forall \boldsymbol{\beta}_{\boldsymbol{1}}^{I_1}.\, C_1 \hookrightarrow D_1,\ \ \ldots,\ \mathsf{op}_n : \forall \boldsymbol{\beta}_{\boldsymbol{n}}^{I_n}.\, C_n \hookrightarrow D_n\} \\ \varepsilon'_{\mathrm{local}} = \{\mathsf{op}_1 : \forall \boldsymbol{\beta}_{\boldsymbol{1}}^{I_1}.\, C_1 \hookrightarrow \forall \alpha.\, D_1,\ \ \ldots,\ \mathsf{op}_n : \forall \boldsymbol{\beta}_{\boldsymbol{n}}^{I_n}.\, C_n \hookrightarrow \forall \alpha.\, D_n\} \\ \vdash \Gamma \quad \alpha \notin \mathit{ftv}(A) \cup \mathit{ftv}(C_1) \cup \cdots \cup \mathit{ftv}(C_n) \quad \mathit{SR}\,(\varepsilon_{\mathrm{local}})\end{array}}{\Gamma \vdash \forall \alpha.\, A \rightarrow^{\varepsilon_{\mathrm{local}}} B \sqsubseteq A \rightarrow^{\varepsilon'_{\mathrm{local}}} \forall \alpha.\, B}$$

First, this rule means that, as (C_DFunEff), given a function of a type $\forall \alpha.\, A \rightarrow^{\varepsilon_{\mathrm{local}}} B$, its body can be regarded as a polymorphic computation of the type $\forall \alpha.\, B$. Suppose that the body performs operation, say, op_i. The effect $\varepsilon_{\mathrm{local}}$ in the LHS type requires op_i to return a value of type D_i where type variable α has been replaced by, say, a type A', while the effect $\varepsilon'_{\mathrm{local}}$ in the RHS type means that an implementation of the operation has to return a polymorphic value of the type $\forall \alpha.\, D_i$. This gap is bridged as follows: given a value of type $\forall \alpha.\, D_i$, the body can instantiate it with the type A' used to instantiate the body. Because the instantiated value should be of type $D[A'/\alpha]$, it meets the requirement of the LHS type.

The above rule can be generalized slightly using existential types: when existential types of the form $\exists \alpha.\, A$ are available, we can allow α to occur in the parameter types $C_1, \ldots, C_n$ and instead set the type signature of each op_i in $\varepsilon'_{\mathrm{local}}$ to $\forall \boldsymbol{\beta}_{\boldsymbol{i}}^{I_i}.\, (\exists \alpha.\, C_i) \hookrightarrow \forall \alpha.\, D_i$. Because an operation clause conforming to this type signature must be independent of the concrete type for α in C_i and return a value polymorphic over α in D_i, it can interpret any call to op_i in the function body no matter what type A' is used to instantiate the body.

The two presented variants of the distributive law are designed so that the latent effect $\varepsilon'_{\mathrm{local}}$ satisfies the signature restriction. However, it is open whether such a property is necessary to prove type soundness. If it turns out to be unnecessary, the typing rule could be further generalized by setting the type signature of op_i in $\varepsilon'_{\mathrm{local}}$ to $\forall \boldsymbol{\beta}_{\boldsymbol{i}}^{I_i}.\forall \alpha.\, (C_i \hookrightarrow D_i)$. Note that $\forall \boldsymbol{\beta}_{\boldsymbol{i}}^{I_i}.\forall \alpha.\, (C_i \hookrightarrow D_i)$ invalidates the signature restriction if α occurs nonstrictly positively in C_i or negatively in D_i. Because $\forall \boldsymbol{\beta}_{\boldsymbol{i}}^{I_i}.\, (\exists \alpha.\, C_i) \hookrightarrow \forall \alpha.\, D_i$ is a "subtype" of $\forall \boldsymbol{\beta}_{\boldsymbol{i}}^{I_i}.\forall \alpha.\, (C_i \hookrightarrow D_i)$, the former imposes more restrictions on operation clauses for op_i than the latter.

Change 3: Type generalization. A simple way to adapt type generalization to the local assignment is to allow type generalization only when effects do not refer to type variables being quantified:

$$\frac{\Gamma, \alpha \vdash M : A \mid \varepsilon_{\mathrm{local}} \quad \mathit{SR}\,(\varepsilon_{\mathrm{local}}) \quad \alpha \notin \mathit{ftv}(\varepsilon_{\mathrm{local}})}{\Gamma \vdash M : \forall \alpha.\, A \mid \varepsilon_{\mathrm{local}}}$$

where $\mathit{ftv}(\varepsilon_{\mathrm{local}})$ is the set of type variables that occur free in $\varepsilon_{\mathrm{local}}$. This rule can also be generalized to allow α to occur in $\varepsilon_{\mathrm{local}}$, as discussed in the previous paragraph, by assigning M an effect $\varepsilon'_{\mathrm{local}}$ where: the parameter and arity types of the type signatures quantify α existentially and universally, respectively; or the type signatures universally quantify α (if $\varepsilon'_{\mathrm{local}}$ does not have to satisfy the signature restriction to show type soundness).

Table 1. Features of type systems for polymorphic algebraic effects

Approach	Signature	Handler (with result type C)	Safety
Naive	All signatures of the form $\forall \boldsymbol{\alpha}. A \hookrightarrow B$	All terms of type $\forall \boldsymbol{\alpha}. A \to (B \to C) \to C$	✗
Ours	Restricted	All	✓
Sekiyama and Igarashi (2019)	All	Restricted	✓

6.3.2 *Type containment with subeffecting*

The type containment rule (C_FUNEFF) for function types requires the latent effects of the RHS and LHS types to be the same. However, we can relax this requirement by involving subeffecting in type containment, which allows the latent effect of the LHS type to be a subset of that of the RHS type, as in the previous work (Bauer and Pretnar, 2014). Although our type containment does not adopt subeffecting for simplicity, we can simulate it by eta-expansion. For example, given a function v of a type $A \to^{\varepsilon} B$ and effect $\varepsilon' \supseteq \varepsilon$, the eta-expansion $\lambda x.v\,x$ of v can be of type $A \to^{\varepsilon'} B$ because the body $v\,x$ can be typed with ε' by (TE_WEAK). Therefore, we suspect that there is no challenge to support subeffecting in type containment.

7 Restriction on signatures versus restriction on handlers

Sekiyama and Igarashi (2019) proposed another approach to safe polymorphic algebraic effects, which ensures the safety by posing an additional constraint on handler implementations. Table 1 summarizes the differences between their approach and ours (as well as the naive unsound approach). Perhaps surprisingly, despite these differences, the two approaches to sound polymorphic algebraic effects are closely related. This section aims to give a brief discussion on this connection, namely a translation from the Sekiyama-Igarashi (SI) type system to our type system.

Let us first explain the basic idea of the SI type system. The exposition here is informal and incomplete; see the paper (Sekiyama and Igarashi, 2019) for the formal and complete description of the type system.

Recall the following counterexample to the safety of the naive polymorphic algebraic effect system.

```
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)
```

Since this is not type safe, a sound type system must reject this example. Sekiyama and Igarashi (2019) focused on the handler implementation:

```
get_id x → resume (λz1. let _ = resume (λz2. z1) in z1) .
```

This handler has the two occurrences of `resume`, which takes arguments of the same type $\alpha \to \alpha$, and Sekiyama and Igarashi (2019) revealed that the source of the problem is the types for the arguments of the two occurrences of `resume` being the same. Actually, in the problematic run of the above program illustrated in Section 2.3, the type of the argument of the outer `resume` is instantiated to $\mathsf{bool} \to \mathsf{bool}$ but that of the inner `resume` is $\mathsf{int} \to \mathsf{int}$. Therefore, their proposal is to regard the two occurrences of $\alpha \to \alpha$ as different types, say $\alpha_1 \to \alpha_1$ for the argument of the outer `resume` and $\alpha_2 \to \alpha_2$ for the argument of the inner `resume`. Then the above handler is not typable, since the argument of the inner `resume` has type $\alpha_2 \to \alpha_1$ while it is expected to have $\alpha_2 \to \alpha_2$. We call a type system extended only with this idea (i.e, assigns fresh type variables to each occurrence of `resume`) the SI$_0$ system; the SI system is more powerful as seen shortly.

The idea of the SI$_0$ system can be explained in terms of the type signature of the effect `get_id`. Instead of the above given signature $\forall\,\alpha.\,\mathsf{unit} \hookrightarrow (\alpha \to \alpha)$, we should regard the type of `get_id` as $\mathsf{unit} \hookrightarrow (\forall\,\alpha.\,\alpha \to \alpha)$. Then the types of the two occurrences of the arguments of `resume` are $\forall\,\alpha.\,\alpha \to \alpha$. Since the type variable α in the type is bound, α for the outer `resume` actually differs from α for the inner `resume`. The resulting type signature satisfies the signature restriction (since both the argument and return types have no free type variable), and hence the type safety of the program is ensured by the argument of this article.

In general, the following two conditions are equivalent for every handler:

- it is typable in the SI$_0$ system as a handler of $\mathsf{op} : \forall\,\alpha.\,A \hookrightarrow B$; and
- it is typable in our system as a handler of $\mathsf{op} : (\exists\,\alpha.\,A) \hookrightarrow (\forall\,\alpha.\,B)$.

The safety of the latter follows from the signature restriction. Hence, our type system explains the SI system's mechanism of fresh type variable assignment for `resume`.

However, the SI system is more powerful than the SI$_0$ system. To explain the difference, consider the following example, which is a minor modification of the previous example:

```
effect get_id' : ∀α. α ↪ (α → α)

handle
  ...
with
  ...
  get_id' x → resume (λz1. let _ = resume (λz2. x) in x) .
```

Note that the type of the argument of the operation `get_id'` has the type variable α. This program is safe independent of an expression between `handle` and `with` (except for the case that the expression itself is unsafe). Unfortunately, this program cannot be typed in the SI_0 system, which only assigns fresh type variables to each occurrence of `resume`. Let $\alpha_1 \to \alpha_1$ and $\alpha_2 \to \alpha_2$ be the types of the arguments of the outer and inner `resume`, respectively. Then one occurrence of `x` requires that $\mathtt{x} : \alpha_1$ whereas the other implies $\mathtt{x} : \alpha_2$, a contradiction.

The SI system has another mechanism that makes the above safe program well-typed. As we have seen, if the type of `resume` has a type variable α, each occurrence of `resume` in a handler introduces its fresh copy, say α_i. Their type system puts a special status to the arguments of the operator (`x` in the above example): a type variable α for the arguments can be identified with any copy α_i. This identification does not need to be consistent, i.e., $\mathtt{x} : \alpha$ can be seen as $\mathtt{x} : \alpha_1$ in an occurrence of `resume` and $\mathtt{x} : \alpha_2$ in another occurrence.

One can mimic this mechanism in our type system by program transformation. The second mechanism allows an argument `x` of the operation may have many different types $\alpha, \alpha_1, \alpha_2, \ldots$ depending on the occurrence of `x`; the number of the types for `x` is 1 + (the number of the occurrences of `resume`). We regard that `x` of different types are actually different variables (or different copies of `x`); each occurrence of `resume` takes a new copy of `x`. The translation $\longmapsto$ is given by

$$\begin{array}{lcl}
\mathtt{op} : \forall \alpha. A \hookrightarrow B & \longmapsto & \mathtt{op}' : (\exists \alpha. A) \hookrightarrow (\forall \alpha. A \to B) \\
\mathtt{op}(M) & \longmapsto & \mathtt{let}\, x = M \,\mathtt{in}\, \mathtt{op}'(x)\, x \\
\mathtt{op}(x) \to \ldots (\mathtt{resume}\, M) \ldots & \longmapsto & \mathtt{op}'(x) \to \ldots (\mathtt{resume}\, (\lambda x. M)) \ldots .
\end{array}$$

After the translation, the caller passes the same argument twice and each `resume` in the handler takes a copy of x. This translation maps a well-typed program in the SI system to a well-typed program in our system.[9]

8 Related work

8.1 *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, 1993). Researchers have developed many approaches to reconcile these conflicting features thus far (Tofte, 1990; Leroy and Weis, 1991; Appel and MacQueen, 1991; Hoang et al., 1993; Wright, 1995; Garrigue, 2004; Asai and Kameyama, 2007; Sekiyama and Igarashi, 2019).

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

[9] What is left is the preservation of the operational semantics by the translation $\longmapsto$. We have not given any formal proof of the semantics preservation, which is left for future development. This property should be obvious for some special cases, e.g., the case that arguments of `resume` are values that are hereditary effect-free.

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 to 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), but they can be applied to other effects as well.

One view for relating our work to these prior works is that, while the prior works suppose that no dangerous type variables can be generalized safely, our work discovers that they can be if the polymorphic effect of interest meets the signature restriction. Note that the assumption of the prior works is natural because they focus on the reference effect, which does not comply with the signature restriction. To see it, assume that we are given a type constructor $A\ \mathsf{ref}$ for references pointing to values of type A. We also suppose that the reference effect supports an operation to create a new reference cell, and it is equipped with type signature $\forall\alpha.\alpha \hookrightarrow \alpha\ \mathsf{ref}$. Because the polarities of the type variables appearing in a reference type $A\ \mathsf{ref}$ are invariant (i.e., positive and negative), the type signature does not meet 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., the type variable α occurs only negatively in the type A and only positively in the type 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 parameter 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 and value argument v can be closed by *quantifying* them and that $\forall\beta_1 \ldots \beta_n.A[C/\alpha] \sqsubseteq A[\forall\beta_1 \ldots \beta_n.\ C/\alpha]$ is provable by type containment, where the distributive law plays a key role. This change 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. As discussed in Section 7, their type system can be explained in our type system without effects. Furthermore, we provide an effect system that allows the use of both of the operations that satisfy and those that do not satisfy the restriction criterion—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, respectively, to ensure that polymorphic expressions are observationally pure. 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.

8.2 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 (2013, 2009) 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), and 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, 1993). They do not provide a means to assign fine-grained type interfaces to individual effects. 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 (Wadler, 1992; Peyton Jones and Wadler, 1993). 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.

9 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 formalized 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 proved that a polymorphic type system is sound if all operations satisfy the signature restriction. We also gave 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 the operations that satisfy and those that 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. Recently, Sekiyama and Tsukada (2021) developed a type-preserving CPS transformation for Curry-style System F with the call-by-value semantics. The crux of their CPS transformation is that System F uses continuations only linearly. Type-preserving CPS transformation for the signature restriction would need a finer-grained constraint on continuations because the presence of algebraic effects and handlers allows the multiple uses of continuations. Another direction is to apply the signature restriction to evaluation strategies other than call-by-value. Harper and Lillibridge (1993) showed that polymorphic type assignment and the polymorphic version of the control operator $\mathsf{call/cc}$ can be reconciled safely in call-by-name at the small cost of the expressive power 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/AEC and JFP 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).

Conflicts of interest

None.

Supplementary material

For supplementary material for this article, please visit https://doi.org/10.1017/S0956796824000054.

References

Ahman, D. (2017) *Fibred Computational Effects*. Ph.D. thesis. University of Edinburgh.

Ahmed, A., Dreyer, D. & Rossberg, A. (2009) State-dependent representation independence. In Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 340–353.

Appel, A. W. & MacQueen, D. B. (1991) Standard ML of New Jersey. In 3rd International Symposium Programming Language Implementation and Logic Programming, PLILP 1991, Proceedings, pp. 1–13.

Asai, K. & Kameyama, Y. (2007) Polymorphic delimited continuations. In 5th Asian Symposium Programming Languages and Systems, APLAS 2007, Proceedings, pp. 239–254.

Bauer, A. & Pretnar, M. (2014) An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci*. **10**(4).

Bauer, A. & Pretnar, M. (2015) Programming with algebraic effects and handlers. *J. Log. Algebr. Methods Program*. **84**(1), 108–123.

Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2020) Binders by day, labels by night: effect instances via lexically scoped handlers. *PACMPL*. **4**(POPL), 48:1–48:29.

Casinghino, C., Sjöberg, V. & Weirich, S. (2014) Combining proofs and programs in a dependently typed language. The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 33–46.

Clinger, W. D., Friedman, D. P. & Wand, M. (1985) A scheme for a higher-level semantic algebra. In *Algebraic Methods in Semantics*, Nivat, M. & Reynolds, J. C. (eds), chapter 6, Cambridge University Press, pp. 237–250.

Cong, Y. & Asai, K. (2018) Handling delimited continuations with dependent types. *PACMPL*. **2**(ICFP), 69:1–69:31.

Damas, L. & Milner, R. (1982) Principal type-schemes for functional programs. In Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, pp. 207–212.

Danvy, O. & Filinski, A. (1990) Abstracting control. In LISP and Functional Programming, pp. 151–160.

Dolan, S., Eliopoulos, S., Hillerström, D., Madhavapeddy, A., Sivaramakrishnan, K. C. & White, L. (2017) Concurrent system programming with effect handlers. In Trends in Functional Programming - 18th International Symposium, TFP 2017, Revised Selected Papers, pp. 98–117.

Dreyer, D., Neis, G. & Birkedal, L. (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, pp. 143–156.

Dunfield, J. & Krishnaswami, N. R. (2013) Complete and easy bidirectional typechecking for higher-rank polymorphism. In ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 429–442.

Felleisen, M. (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, pp. 180–190.

Filinski, A. (2010) Monads in action. In Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 483–494.

Forster, Y., Kammar, O., Lindley, S. & Pretnar, M. (2019) On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* **29**, e15.

Garrigue, J. (2004) Relaxing the value restriction. In 7th International Symposium Functional and Logic Programming, FLOPS 2004, Proceedings, pp. 196–213.

Girard, J. Y. (1972) *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat d'état. Université Paris 7.

Gordon, M. J. C., Milner, R. & Wadsworth, C. P. (1979) *Edinburgh LCF*. vol. 78. Lecture Notes in Computer Science. Springer.

Gunter, C. A., Rémy, D. & Riecke, J. G. (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, pp. 12–23.

Harper, R. & Lillibridge, M. (1991) ML with callcc is unsound. Announcement on the `Types` Electronic Forum.

Harper, R. & Lillibridge, M. (1993) Explicit polymorphism and CPS conversion. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 206–219.

Harper, R. & Lillibridge, M. (1993) Polymorphic type assignment and CPS conversion. *Lisp Symb. Comput*. **6**(3-4), 361–380.

Hoang, M., Mitchell, J. C. & Viswanathan, R. (1993) Standard ML-NJ weak polymorphism and imperative constructs. In Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), pp. 15–25.

Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. In ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 145–158.

Kammar, O. & Pretnar, M. (2017) No value restriction is needed for algebraic effects and handlers. *J. Funct. Program*. **27**, e7.

Leijen, D. (2017) Type directed compilation of row-typed algebraic effects. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 486–499.

Leijen, D. (2023) The Koka programming langauge. `https://koka-lang.github.io/koka/doc/index.html`.

Leivant, D. (1983) Polymorphic type inference. In Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, POPL 1983, pp. 88–98.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. & Vouillon, J. (2020) The OCaml system release 4.10: Documentation and user's manua.

Leroy, X. & Weis, P. (1991) Polymorphic type inference and assignment. In Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages, pp. 291–302.

Levy, P. B. (2001) *Call-by-push-value*. Ph.D. thesis. Queen Mary University of London, UK.

Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 500–514.

Lindley, S., McBride, C., McLaughlin, C. & Convent, L. (2022) The Frank programming langauge `https://github.com/frank-lang/frank`.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17**(3), 348–375.

Milner, R., Tofte, M. & Harper, R. (1990) *The Definition of Standard ML*. MIT Press.

Mitchell, J. C. (1988) Polymorphic type inference and containment. *Inf. Comput.* **76**(2/3), 211–249.

Moggi, E. (1991) Notions of computation and monads. *Inf. Comput.* **93**(1), 55–92.

Pédrot, P. & Tabareau, N. (2020) The fire triangle: how to mix substitution, dependent elimination, and effects. *PACMPL*. **4**(POPL), 58:1–58:28.

Peyton Jones, S. L., Vytiniotis, D., Weirich, S. & Shields, M. (2007) Practical type inference for arbitrary-rank types. *J. Funct. Program.* **17**(1), 1–82.

Peyton Jones, S. L. & Wadler, P. (1993) Imperative functional programming. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 71–84.

Pitts, A. & Stark, I. (1998) Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, Gordon, A. & Pitts, A. (eds). Publications of the Newton Institute, Cambridge University Press, pp. 227–273. Available at: http://www.inf.ed.ac.uk/~stark/operfl.html.

Plotkin, G. D. & Power, J. (2003) Algebraic operations and generic effects. *Appl. Categor. Struct.* **11**(1), 69–94.

Plotkin, G. D. & Pretnar, M. (2008) A logic for algebraic effects. In Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS. IEEE Computer Society, pp. 118–129.

Plotkin, G. D. & Pretnar, M. (2009) Handlers of algebraic effects. In 18th European Symposium on Programming Programming Languages and Systems, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, Proceedings, pp. 80–94.

Plotkin, G. D. & Pretnar, M. (2013) Handling algebraic effects. *Logical Methods in Computer Science*. **9**(4).

Reynolds, J. C. (1974) Towards a theory of type structure. In Programming Symposium, Proceedings Colloque sur la Programmation, pp. 408–423.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. In IFIP Congress, pp. 513–523.

Sekiyama, T. & Igarashi, A. (2017) Stateful manifest contracts. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 530–544.

Sekiyama, T. & Igarashi, A. (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, pp. 353–380.

Sekiyama, T. & Tsukada, T. (2021) CPS transformation with affine types for call-by-value implicit polymorphism. *Proc. ACM Program. Lang.* **5**(ICFP), 1–30.

Sekiyama, T., Tsukada, T. & Igarashi, A. (2020) Signature restriction for polymorphic algebraic effects. *Proc. ACM Program. Lang.* **4**(ICFP), 117:1–117:30.

Sitaram, D. (1993) Handling control. In Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), pp. 147–155.

Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J. K. & Béguelin, S. Z. (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, pp. 256–270.

The Links team. (2022) The Links programming langauge https://links-lang.org/.

Tiuryn, J. & Urzyczyn, P. (1996) The subtyping problem for second-order types is undecidable. Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96), pp. 74–85.

Tofte, M. (1990) Type inference for polymorphic references. *Inf. Comput.* **89**(1), 1–34.

Wadler, P. (1992) The essence of functional programming. In Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 1–14.

Wells, J. B. (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), pp. 176–185.

Wright, A. K. (1995) Simple imperative polymorphism. *Lisp Symb. Comput.* **8**(4), 343–355.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94.

Xi, H. (2007) Dependent ML an approach to practical programming with dependent types. *J. Funct. Program.* **17**(2), 215–286.