Logical Methods in Computer Science
Vol. 10(3:21)2014, pp. 1–43
www.lmcs-online.org

Submitted Dec. 5, 2013
Published Sep. 12, 2014

INFERRING ALGEBRAIC EFFECTS

MATIJA PRETNAR

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: matija@pretnar.info

Abstract. We present a complete polymorphic effect inference algorithm for an ML-style language with handlers of not only exceptions, but of any other algebraic effect such as input & output, mutable references and many others.

Our main aim is to offer the programmer a useful insight into the effectful behaviour of programs. Handlers help here by cutting down possible effects and the resulting lengthy output that often plagues precise effect systems. Additionally, we present a set of methods that further simplify the displayed types, some even by deliberately hiding inferred information from the programmer.

Though Haskell [10] fans may not think it is better to write impure programs in ML [18], they do agree it is easier. You can insert a harmless printout without rewriting the rest of the program, and you can combine multiple effects without a monad transformer. This flexibility comes at a cost, though — ML types offer no insight into what effects may happen. The suggested solution is to use an effect system [16, 29, 4, 31, 33, 3, 27], which enriches existing types with information about effects.

An effect system can play two roles: it can be *descriptive* and inform about potential effects, and it can be *prescriptive* and limit the allowed ones. In this paper, we focus on the former. It turns out that striking a balance between expressiveness and simplicity of a descriptive effect system is hard. One of the bigger problems is that effects tend to pile up, and if the effect system takes them all into account, we are often left with a lengthy output listing every single effect there is.

In this paper, we present a complete inference algorithm for an expressive and simple descriptive polymorphic effect system of *Eff* [2] (freely available at `http://eff-lang.org`), an ML-style language with handlers of not only exceptions, but of any other *algebraic effect* [22] such as input & output, non-determinism, mutable references and many others [23, 2]. Handlers prove to be extremely versatile and can express stream redirection, transactional memory, backtracking, cooperative multi-threading, delimited continuations, and, like monads, give programmers a way to define their own. And as handlers eliminate effects, they make the effect system *non-monotone*, which helps with the above issue of a snowballing output.

2012 ACM CCS: [**Theory of computation**]: Semantics and reasoning—Program reasoning—Program analysis.

Key words and phrases: algebraic effects, effect handlers, effect inference, effect system.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-10(3:21)2014

We start in Section 1 with an tour of *Eff* in which we informally explain handlers and show the main features of the proposed effect system. Afterwards, we switch to formal development and in Section 2, we recap the effect system for *core Eff* [1], a minimal formalization of *Eff*. Our contributions are:

- A set of syntax-directed rules for inferring types and constraints they must satisfy (Section 3). We show that these rules are sound and complete with regard to the effect system.
- A unification algorithm that decomposes a set of constraints to a more basic form and decides if it admits a solution (Section 4). Unification fails only in the case of type mismatch, which fits our goal of a descriptive effect system that just refines existing ML types with details about effects.
- A number of techniques that reduce the number of constraints without changing the set of solutions (Section 5). The heavy lifting is done by *garbage collection* of constraints [25, 28, 32], which we borrow and adapt slightly to our purpose. We also introduce a few more techniques particular to the algebraic setting.
- A further collection of tactics for simplifying the display of inferred types (Section 6). To fully achieve their purpose, some of the presented tactics deliberately hide information from the programmer, though entire information is always used in the background.

We conclude by showing a couple of full runs of the algorithm (Section 7) and by discussing related and future work. To preserve the flow of the paper, we gather the (mostly routine) proofs in Appendix A.

1. *Eff*

Effects aside, *Eff* should be familiar to anyone that has worked with OCaml [9]. For example, the `map` function, which applies a function `f` to each element of the list `xs` is defined as:

```
let rec map f xs =
  match xs with
  | [] -> []
  | x :: xs -> f x :: map f xs
```

The first important feature that distinguishes *Eff* from OCaml is its effect system. For example, the inferred type of `map` is

$$\texttt{map} : (\alpha \xrightarrow{\delta} \beta) \to (\alpha\ \texttt{list} \xrightarrow{\delta} \beta\ \texttt{list})$$

Here, the annotation on the arrow, called the *dirt*, describes any effects that the function may trigger. This additional information is very easy to understand: the function `map f` causes exactly the same effects δ as `f`. The lack of dirt on the second arrow signifies that the application of `map` to `f` is pure. Inferred types of some other typical higher-order functions

are:

$$\texttt{compose} : (\alpha \xrightarrow{\delta} \beta) \to (\beta \xrightarrow{\delta'} \gamma) \to (\alpha \xrightarrow{\delta \cup \delta'} \gamma)$$

$$\texttt{curry} : (\alpha \times \beta \xrightarrow{\delta} \gamma) \to (\alpha \to \beta \xrightarrow{\delta} \gamma)$$

$$\texttt{uncurry} : (\alpha \xrightarrow{\delta} \beta \xrightarrow{\delta'} \gamma) \to (\alpha \times \beta \xrightarrow{\delta \cup \delta'} \gamma)$$

$$\texttt{fold\_left} : (\alpha \xrightarrow{\delta} \beta \xrightarrow{\delta'} \alpha) \to \alpha \to \beta\ \texttt{list} \xrightarrow{\delta \cup \delta'} \alpha$$

$$\texttt{fold\_right} : (\alpha \xrightarrow{\delta} \beta \xrightarrow{\delta'} \beta) \to \alpha\ \texttt{list} \to \beta \xrightarrow{\delta \cup \delta'} \beta$$

$$\texttt{filter} : (\alpha \xrightarrow{\delta} \texttt{bool}) \to (\alpha\ \texttt{list} \xrightarrow{\delta} \alpha\ \texttt{list})$$

The dirt annotations are similar to ones usually given in existing polymorphic effect systems such as [14]. Also note that if we disable the display of annotations, *Eff* shows the programmer exactly the same types as OCaml.

The second distinguishing feature is that *Eff* is based on algebraic effects [21, 22]. This means that effects are accessed exclusively through a set of *operations*, which are all of the form `ins#op`, where an *operation symbol* `op` describes the action, and an *instance* `ins` describes where it should happen. This means that we print to the standard output channel using an operation `std#print` instead of a primitive function `print_string`, we access a reference `r` through operations `r#lookup` and `r#update` instead of through `!` and `:=`, and we raise an exception `exc` by calling `exc#raise` instead of using a special keyword `raise`. The latter constructs are, of course, all definable in terms of the former.

This uniform representation allows effects to be seamlessly combined [8], forms a natural basis for an effect system — possible effects of a given computation are captured accurately by the set of operations it calls — and paves the way for the third distinguishing feature of *Eff*: handlers of arbitrary algebraic effects [23].

Let us take a look at a few concrete examples. What follows is by no means an exhaustive list of what can be accomplished with handlers. For more examples, please see [23, 2, 11].

1.1. **Exceptions.**

We start with exceptions as they are the simplest algebraic effect and as exception handlers are already a well established concept.

1.1.1. *Effect types.*

Instances are first-class values in *Eff* and are given an *effect type*. For exception instances, called simply *exceptions*, the effect type is of the form α `exception`, where α is the type of any additional information that the exception may carry.

We declare each effect type together with an *effect signature* that lists available operation symbols together with the types of parameters they accept and of results they yield to the waiting continuation. The signature corresponding to α `exception` is

$$\{\texttt{raise} : \alpha \to \texttt{empty}\}$$

where the result type of `raise` is `empty` (a sum type with zero constructors) because raising an exception terminates the computation and yields nothing to the continuation.

Each term of an effect type is also annotated with a *region*, which describes (an overapproximation of) the set of instances it may occupy. For example, an exception `exc1` is given the type α `exception`$^{\{\texttt{exc1}\}}$, while the conditional `if cond then exc1 else exc2` is given the type α `exception`$^{\{\texttt{exc1},\texttt{exc2}\}}$ (any bigger sets would also be fine).

1.1.2. *Raising exceptions.* Since there is little we can do with a result of the empty type, we rarely raise exceptions directly with `exc#raise`. Instead, we define a convenience function, also named `raise`, which places the exception call inside the empty type eliminator:

```
let raise exc arg =
  match (exc#raise arg) with
```

with the inferred type

$$\texttt{raise} : \alpha\ \texttt{exception}^{\rho} \to \alpha \xrightarrow{\texttt{raise}:\rho} \beta$$

So, `raise` takes an exception from a region captured by a region parameter ρ and a matching argument of type α. The second application results in a computation that can raise any exception from ρ, while its return type β can be any arbitrary. This allows us to use `raise` at any point in the computation, for example in computing the tail of a list:

```
let tail xs =
  match xs with
  | [] -> raise emptyListTail
  | _ :: xs -> xs
```

where `emptyListTail` is the exception stating that the empty list has no tail. The inferred type of `tail` is

$$\texttt{tail} : \alpha\ \texttt{list} \xrightarrow{\texttt{raise}:\{\texttt{emptyListTail}\}} \alpha\ \texttt{list}$$

where the dirt shows that `emptyListTail` may be raised during execution.

In addition to inferring types of values, we can also infer *dirty types* $A\ !\ \Delta$ of computations. These consist of a value type A describing the result and of a dirt Δ describing the operations that may be called while computing it. For example, the inferred dirty type of the computation `tail [1; 2; 3]` is $\texttt{int list} ! \{\texttt{raise} : \{\texttt{emptyListTail}\}\}$. Note that the effect system is conservative and signals a possible exception even though it will not be raised during runtime. This also means that a computation is guaranteed to be pure whenever its inferred dirt is empty.

For a final example of the effect system before we turn to handlers, let us combine `map` and `tail` as

```
let map_tail f xs = map f (tail xs)
```

with the inferred type

$$\texttt{map\_tail} : (\alpha \xrightarrow{\texttt{raise}:\rho \mid \delta} \beta) \to (\alpha\ \texttt{list} \xrightarrow{\texttt{raise}:\{\texttt{emptyListTail}\} \cup \rho \mid \delta} \beta\ \texttt{list})$$

Unlike the argument to `map`, the argument of `map_tail` has its dirt split into two parts: the region parameter ρ captures the region of all the exceptions that `f` may raise, while the dirt parameter δ captures all other operations. The split allows us to express the fact that `map_tail f` may raise either `emptyListTail` or an exception from ρ, and that it may call any operation from δ that `f` can. This is similar to *row typing* [26], where we use a parameter to capture all the fields of a record that are not explicitly mentioned.

1.1.3. *Handling exceptions.* As exceptions are the simplest example of effects, exception handlers are the simplest example of handlers. A `try with` handling construct known from OCaml, like the one in

```
let print_ratio x y =
  try
    print_string ("Ratio is " ^ string_of_float (x /. y))
  with
  | Division_by_zero -> print_string "Ratio does not exist!"
```

would be used in *Eff* as

```
let print_ratio x y =
  handle
    std#print ("Ratio is " ^ string_of_float (x /. y))
  with
  | divisionByZero#raise _ _ -> std#print "Ratio does not exist!"
```

Each handler case takes two arguments which we ignore for now.

Like instances, handlers are first-class values in *Eff*, and the above is just special syntax for

```
let print_ratio x y =
  with h handle
    std#print ("Ratio is " ^ string_of_float (x /. y))
```

where `h` is given by

```
handler
| divisionByZero#raise _ _ -> std#print "Ratio does not exist!"
```

Handlers are given *handler types* $A \,!\, \Delta \Rightarrow B \,!\, \Delta'$, meaning that a handler takes a computation with *incoming* dirty type $A \,!\, \Delta$ and transforms it into a computation with *outgoing* dirty type $B \,!\, \Delta'$. The inferred type for `h` is

$$\texttt{unit} \,!\, \{\texttt{raise} : \rho_1, \texttt{print} : \rho_2 \mid \delta\} \;\Rightarrow\; \texttt{unit} \,!\, \{\texttt{raise} : \rho_1 - \texttt{divisionByZero}, \texttt{print} : \{\texttt{std}\} \cup \rho_2 \mid \delta\}$$

We see that `h` leaves the type of results to be `unit`. Its incoming dirt is split into three parts: exceptions ρ_1 that we may raise, channels ρ_2 that we may print to, and any other operations δ different from `raise` or `print` that we may call. The outgoing dirt is similarly split: the handled computation may still raise exceptions from ρ_1, except that now `divisionByZero` will be handled. Next, the handled computation may print to `std` in addition to any channel in ρ_2. Finally, any operation in δ will be neither caught nor called by `h`, so this part remains as it is.

Handler types usually (but not always) all have the same shape: they remove certain operations, possibly add some of their own, and pass through any unhandled dirt. This results in a repetitive type, which can be written in a more compact form that emphasises only the differences. For `h`, this is:

$$\texttt{unit} \xrightarrow{\texttt{raise}:-\texttt{divisionByZero},\ \texttt{print}:+\texttt{std}} \texttt{unit}$$

So, `h` removes `divisionByZero#raise`, adds `std#print`, and leaves the rest as it is.

In practice, exception handlers are rarely reused, because treatment of exceptions varies greatly depending on the context in which they are handled. An example of a more general exception handler is `optionalize`, which transforms a computation into one that yields an optional result, depending on if a given exception `exc` was raised or not. We define `optionalize` as:

```
let optionalize exc =
  handler
  | exc#raise _ _ -> None
  | val x -> Some x
```

The first thing to notice is the special `val` case, which determines what to do when the handled computation returns a value. In our case, we wrap it with the `Some` constructor of the `option` type. Then, if we define

```
let tail_opt xs =
  with (optionalize emptyListTail) handle (tail xs)
```

the call `tail_opt [1; 2; 3]` evaluates to `Some [2; 3]`, while `tail_opt []` evaluates to `None`. The inferred type of `tail_opt` is the pure α `list` $\to$ (α `list`) `option`, while the type of `optionalize` is

$$\texttt{optionalize} : \alpha\ \texttt{exception}^{\rho} \to (\beta \xRightarrow{\texttt{raise}:\div\rho} \beta\ \texttt{option})$$

So, we change the result type from β to β `option`, and remove any call of `exc#raise` for the exception `exc` determined by the region ρ.

1.1.4. *Handled regions.* You may have observed that we wrote $\div\rho$ instead of $-\rho$ in the type of `optionalize`. This is meant to point out that we may remove ρ from the dirt only if it denotes a singleton. The reason for this is as follows.

Take exceptions `exc1` and `exc2` and define a handler `h` as:

```
let exc = if cond then exc1 else exc2 in
let h =
  handler
  | exc#raise _ _ -> ...
```

So, does the `exc#raise` case of `h` handle `exc1#raise` or `exc2#raise` at runtime? Unfortunately, just by looking at the type $\texttt{exc} : \alpha\ \texttt{exception}^{\{\texttt{exc1},\texttt{exc2}\}}$, we cannot say anything, so we must assume that both are unhandled. The only way we can be sure during typechecking that a handling case removes a given operation is when the region of its instance is a singleton.

For this reason, we write $\div\rho$ whenever the region of the handled instance is still some parameter ρ. If this eventually turns out to denote some singleton $\{\texttt{ins}\}$, we may safely replace it by $-\texttt{ins}$. But if it turns to be a bigger set, we need to drop it from the handler type.

1.2. **Input & output.** Interactive input & output is also a very simple algebraic effect, yet its handlers expose almost all the important aspects of general ones. For input & output, the effect instances are called *channels* and have the effect type `channel` with the signature

$$\{\texttt{read} : \texttt{unit} \to \texttt{string}, \texttt{print} : \texttt{string} \to \texttt{unit}\}$$

A simple output handler is one that reverses the order of printouts:

```
handler
| std#print msg k -> k (); std#print msg
```

The `std#print` case takes two parameters: the string `msg` given to `std#print` and the continuation `k` waiting for its result. We first resume the continuation by passing it the unit value `()`, and only after that finishes, we print out `msg`. The handler recursively handles any other `std#print` that the continuation may call, so the order of printouts in the continuation is reversed as well. Note, however, that `std#print` on the right-hand side is outside the scope of the handler and remains unhandled (unless there are more handlers nested outside).

A more interesting example that can be useful in unit testing is a handler that collects all printouts to a list of strings and returns it together with the result:

```
handler
| val x -> (x, [])
| std#print msg k ->
    let (x, msgs) = k () in
    (x, msg :: msgs)
```

So, a computation that just returns the value `x` prints nothing, and we return an empty list `[]` together with `x`. If, however, the computation prints the string `msg`, we resume the continuation `k`. This is also handled with the same handler, so it returns some value `x` and a list of its messages `msgs`. Now, we only need to prepend `msg` to this list and return it together with `x`. The fact that the handler changes the type of the handled computation is reflected in its inferred type $\alpha \xRightarrow{\texttt{print:}-\texttt{std}} \alpha \times \texttt{string list}$.

A matching handler that is also useful in unit testing is one that feeds a list of strings to `std#read`:

```
handler
| val x -> (fun strs -> x)
| std#read () k -> (fun strs ->
    match strs with
    | str :: strs' -> (k str) strs'
    | [] -> (k "") []
  )
```

We accomplish this by transforming a computation into a function that accepts a list of strings `strs`. If the computation returns some value `x`, the function ignores its argument and returns `x`. But if the computation calls `std#read`, we take a look at the list `strs`. If it is non-empty, we pass `k` its first element `str`. And since the continuation is further handled, it is also a function that accepts a list of strings, so we pass it the remainder `strs'`. If it is empty, we pass `k` the empty string and again the empty list.

The inferred type of the handler is

$$\alpha \,!\, \{\texttt{read} : \rho \mid \delta\} \;\Rightarrow\; (\texttt{string list} \xrightarrow{\Delta} \alpha) \,!\, \Delta$$

where $\Delta = \{\texttt{read} : \rho - \texttt{std} \mid \delta\}$. The reason Δ appears in two places is because operations other than `std#read` may occur before or after we handle the first `std#read` and so obtain a function. Since Δ appears twice, we unfortunately cannot use the compact form.

However, *Eff* extends handlers with an additional `finally` case, which first transforms the computation with the handler, and then applies the finally case computation to the resulting value. In particular, if `h` is some handler and `h_fin` is the same as `h` except that it also contains a case `finally x -> c_fin`, the computation `with h_fin handle c` behaves exactly as

```
let x = (with h handle c) in c_fin
```

Using this extension, we can define

```
let supply_input strs0 =
  handler
  | val x -> (* ...as before... *)
  | std#read () k -> (* ...as before... *)
  | finally f -> f strs0
```

So, once we get back a function `f` accepting a list of inputs, we apply it to the given list `strs0`. We use the handler as

```
with
  supply_input ["Alpha"; "Bravo"; "Charlie"]
handle
  ...
```

The inferred type then takes the simpler form

$$\texttt{supply\_input} : \texttt{string list} \to (\alpha \xrightarrow{\texttt{read}:-\texttt{std}} \alpha)$$

1.3. **References.** Similar to OCaml, mutable references in *Eff* are given the effect type α `ref` with the signature

$$\{\texttt{lookup} : \texttt{unit} \to \alpha,\ \texttt{update} : \alpha \to \texttt{unit}\}$$

We can define the OCaml accessors by:

```
let (!) r = r#lookup ()
let (:=) r v = r#update v
```

with the types

$$(\texttt{!}) : \alpha\ \texttt{ref}^{\rho} \xrightarrow{\texttt{lookup}:\rho} \alpha \qquad\qquad (\texttt{:=}) : \alpha\ \texttt{ref}^{\rho} \to \alpha \xrightarrow{\texttt{update}:\rho} \texttt{unit}$$

With reference handlers, we can temporarily alter the value stored in the reference, make it read-only, log all changes, and more. However, handlers are not meant only for overriding but also for defining effects. In particular, we can use handlers to implement references with a state monad:

```
let state r s0 =
  handler
  | val x -> (fun s -> x)
  | r#lookup () k -> (fun s -> k s s)
  | r#update s' k -> (fun s -> k () s')
  | finally f -> f s0
```

The function `state` gives a handler that handles a stateful computation using reference `r` into a pure function that accepts the current state `s` and passes it around. So, we handle `lookup` by a function that takes the state `s` and passes it to the continuation `k`, which expects the current state as the outcome of `lookup`. Since this continuation is further handled, it is again a function accepting current state. Looking up a reference does not change the state, so we pass `s` again, thus `k` is applied to `s` twice. In the case for `update`, the expected outcome of the call is the unit value, while the current state is overwritten by the parameter `s'`. And the `finally` case says that once we get back a function accepting current state, we apply it to a given initial state `s0`.

The inferred type of `state` is

$$\texttt{state} : \alpha\ \texttt{ref}^{\rho} \to \alpha \to (\beta \xRightarrow{\texttt{lookup}:\div\rho,\texttt{update}:\div\rho} \beta)$$

If we want to access the final state, we define `state'` to be exactly the same as `state`, except that its value case is `val x -> (fun s -> (x, s))`. In this case the inferred type is

$$\texttt{state'} : \alpha\ \texttt{ref}^{\rho} \to \alpha \to (\beta \xRightarrow{\texttt{lookup}:\div\rho,\texttt{update}:\div\rho} \beta \times \alpha)$$

We can use multiple references without a hitch. For example, given two references `r1` and `r2`, the computation

```
with (state r1 6) handle
  with (state r2 0) handle
    let x = !r1 in
    r2 := x + 1;
    !r2 * x
```

returns `42` and its inferred type is the pure `int` $!\ \emptyset$.

If we replace `state` by `state'`, the handled computation returns `(6, (7, 42))` and the inferred type is `int` $\times$ (`int` $\times$ `int`) $!\ \emptyset$. This shows how easy it is to change the effectful behaviour by just switching the handlers and keeping the imperative code as it is.

2. Core Eff

For formal development, we restrict our attention to *core* Eff, a minimal fragment of *Eff*. Core *Eff* differs from *Eff* in the following aspects:

- Core *Eff* is a fine-grain call-by-value calculus [15], which means that its terms are split into inert *expressions* and possibly effectful *computations*. The separation makes the formalization much simpler, but makes programming that much harder. For this reason, *Eff* allows the programmer to freely mix the terms, and performs the routine separation automatically. For example, a program such as

  ```
  let transform f m n = (f m 42, n)
  ```

  gets translated into

  ```
  let transform = fun f -> val (fun m -> val (fun n ->
    let tmp1 =
      let tmp2 = f m in
      tmp2 42
    in
    val (tmp1, n)
  ))
  ```

where `val` constructs a computation that immediately returns a value represented by a given expression.
- *Eff* allows programmers to define their own parametric inductive datatypes and effect types, while in core *Eff*, we fix the signature of effect types and drop inductive datatypes entirely.
- *Eff* provides a `new` construct that allows a programmer to create fresh instances at runtime [2], and can be used to model both exception declarations and reference allocations in ML. Since the formalization of this feature is still under investigation, we omit it from our development and only briefly discuss it in the Conclusion.
- Handlers in *Eff* allow the additional `finally` case in handlers, but as already discussed on page 8, this is merely a convenience that can be expressed with existing constructs.

2.1. **Terms.** We start with a given set of *effects* E, which are just labels such as `channel`, `exception` or `ref` for all possible effects we want to use in our programs. Next, for each effect E, we have a fixed set $\mathcal{O}_E$ of operation symbols `op` and a fixed set $\mathcal{I}_E$ of instances `ins`. We assume that in each operation `ins#op`, both `ins` and `op` belong to the same effect.

The *expressions* e and *computations* c of core *Eff* are given by:

$$\begin{aligned}
\text{expression } e &::= x \mid \texttt{true} \mid \texttt{false} \mid 0 \mid \texttt{succ}\, e \mid \texttt{()} \mid \texttt{fun}\, x \mapsto c \mid \texttt{ins} \mid h \\
\text{handler } h &::= (\texttt{handler val}\, x \mapsto c_v \mid (e_i \texttt{\#op}_i\, x\, k \mapsto c_i)_i) \\
\text{computation } c &::= \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \mid \texttt{iszero}\, e \mid \texttt{pred}\, e \mid \texttt{absurd}\, e \mid e_1\, e_2 \mid \\
&\quad \texttt{val}\, e \mid e_1 \texttt{\#op}\, e_2\, (y.\, c) \mid \texttt{let}\, x = c_1\, \texttt{in}\, c_2 \mid \texttt{let val}\, x = e\, \texttt{in}\, c \mid \\
&\quad \texttt{let rec}\, f\, x = c_1\, \texttt{in}\, c_2 \mid \texttt{with}\, e\, \texttt{handle}\, c
\end{aligned}$$

Here and everywhere, we write $(-)_{i \in I}$ or just $(-)_i$ to denote a finite repetition of $-$. The language constructs are standard except for: the empty type eliminator `absurd`, the computation `val` that immediately evaluates to a value, polymorphic let-binding `let val`, and the already discussed *instance constants*, *handlers*, *operation calls* and the *handling construct*. Note that both instance constants and handlers are first-class values in core *Eff*.

The operation calls in core *Eff* are slightly different from the ones in *Eff*. The call $\texttt{ins\#op}\, e\, (y.\, c)$ represents an application of an operation `ins#op` to a parameter e with a continuation $(y.\, c)$ waiting for the result of the call to be bound to y. Explicit continuations are convenient for operational semantics, but we do not expect the programmer to use them. Instead, *Eff* uses *generic effects* [22], defined as

$$e\texttt{\#op} \stackrel{\text{def}}{=} \texttt{fun}\, x \mapsto e\texttt{\#op}\, x\, (y.\, \texttt{val}\, y)$$

which take a parameter and perform the operation call with the trivial continuation. Then, the programmer can write the more familiar $\texttt{let}\, y = \texttt{ins\#op}\, e\, \texttt{in}\, c$ instead of $\texttt{ins\#op}\, e\, (y.\, c)$, and we shall see that the two exhibit equivalent behaviour.

The rough idea is that each non-divergent computation either evaluates to a value or calls an operation. We use a handler $h = \texttt{handler val}\, x \mapsto c_v \mid (\texttt{ins}_i \texttt{\#op}_i\, x\, k \mapsto c_i)_i$ on a computation c as follows:
- If c evaluates to a value $\texttt{val}\, e$, we use the value case and evaluate $c_v[e/x]$.
- If c performs an operation call $\texttt{ins}_j \texttt{\#op}_j\, e\, (y.\, c')$ and the handler contains a matching case $\texttt{ins}_j \texttt{\#op}_j\, x\, k \mapsto c_j$, we evaluate $c_j[e/x, (\texttt{fun}\ y \mapsto \texttt{with}\ h\ \texttt{handle}\ c')/k]$. We wrap

the handler h around the continuation so that it may continue handling future operation calls, though any operations called by c_j escape its scope.
- If the handler has no matching operation cases for the called operation, then just like in exception handlers, we propagate the call outwards for other handlers to catch, though we still wrap h around the continuation as it may handle some other operations.

Let-binding $\texttt{let}\ x = c_1\ \texttt{in}\ c_2$ works as follows: if c_1 evaluates to $\texttt{val}\ e$, we continue with $c_2[e/x]$, but if c_1 calls an operation, we propagate the call outwards just like when a handler has no matching cases. In fact, let-binding $\texttt{let}\ x = c_1\ \texttt{in}\ c_2$ works exactly as $\texttt{with}\ (\texttt{handler}\ \texttt{val}\ x \mapsto c_2)\ \texttt{handle}\ c_1$ Though this makes let binding redundant, we keep it in the language for convenient notation and to serve as a stepping stone to the less familiar handling construct.

To make the above intuition more precise and to motivate the effect system, we now give a small-step operational semantics, determined by a relation $c \rightsquigarrow c'$ defined in Figure 1, stating that a computation c takes a single step to c'. Note that the relation is given for computations only and that expressions are inert.

$$\frac{}{\texttt{if true then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow c_1} \qquad \frac{}{\texttt{if false then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow c_2} \qquad \frac{}{\texttt{iszero}\ 0 \rightsquigarrow \texttt{val true}}$$

$$\frac{}{\texttt{iszero}\ (\texttt{succ}\ e) \rightsquigarrow \texttt{val false}} \qquad \frac{}{\texttt{pred}\ 0 \rightsquigarrow \texttt{val}\ 0} \qquad \frac{}{\texttt{pred}\ (\texttt{succ}\ e) \rightsquigarrow \texttt{val}\ e}$$

$$\frac{}{(\texttt{fun}\ x \mapsto c)\ e \rightsquigarrow c[e/x]} \qquad \frac{c_1 \rightsquigarrow c_1'}{\texttt{let}\ x = c_1\ \texttt{in}\ c_2 \rightsquigarrow \texttt{let}\ x = c_1'\ \texttt{in}\ c_2} \qquad \frac{}{\texttt{let}\ x = \texttt{val}\ e\ \texttt{in}\ c \rightsquigarrow c[e/x]}$$

$$\frac{}{\texttt{let}\ x = \texttt{ins\#op}\ e\ (y.\, c_1)\ \texttt{in}\ c_2 \rightsquigarrow \texttt{ins\#op}\ e\ (y.\, \texttt{let}\ x = c_1\ \texttt{in}\ c_2)} \qquad \frac{}{\texttt{let val}\ x = e\ \texttt{in}\ c \rightsquigarrow c[e/x]}$$

$$\frac{}{\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2 \rightsquigarrow c_2[(\texttt{fun}\ x \mapsto \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_1)/f]}$$

$$\frac{c \rightsquigarrow c'}{\texttt{with}\ e\ \texttt{handle}\ c \rightsquigarrow \texttt{with}\ e\ \texttt{handle}\ c'} \qquad \frac{}{\texttt{with}\ h\ \texttt{handle}\ (\texttt{val}\ e) \rightsquigarrow c_v[e/x]}$$

$$\frac{}{\texttt{with}\ h\ \texttt{handle}\ (\texttt{ins}_j\texttt{\#op}_j\ e\ (y.\, c)) \rightsquigarrow c_j[e/x, (\texttt{fun}\ y \mapsto \texttt{with}\ h\ \texttt{handle}\ c)/k]}$$

$$\frac{\texttt{ins\#op} \notin \{\texttt{ins}_i\texttt{\#op}_i\}_i}{\texttt{with}\ h\ \texttt{handle}\ (\texttt{ins\#op}\ e\ (y.\, c)) \rightsquigarrow \texttt{ins\#op}\ e\ (y.\, \texttt{with}\ h\ \texttt{handle}\ c)}$$

Figure 1: The inductive definition of the relation $c \rightsquigarrow c'$. In the last three rules, we set $h = \texttt{handler}\ \texttt{val}\ x \mapsto c_v\ |\ (\texttt{ins}_i\texttt{\#op}_i\ x\ k \mapsto c_i)_i$.

Example 2.1. Take a reference handler

$$\begin{aligned} h \stackrel{\text{def}}{=}\ & \texttt{handler} \\ & \mid \texttt{val}\ x \mapsto r\texttt{\#update}\ x \\ & \mid r\texttt{\#lookup}\ x\ k \mapsto k\ (\texttt{succ}\ 0) \\ & \mid r\texttt{\#update}\ x\ k \mapsto k\ () \end{aligned}$$

which temporarily treats a reference $r \in \mathcal{I}_{\texttt{ref}}$ as if it always contains 1, and afterwards updates it with the final result of the handled computation. This update is not handled by h because it escapes its scope. If we apply h on the computation

$$\begin{aligned} c \stackrel{\text{def}}{=}\ & \texttt{let}\ x_1 = \\ & \quad \texttt{let}\ x_2 = r\texttt{\#lookup}\ ()\ \texttt{in} \\ & \quad r\texttt{\#update}\, x_2 \\ & \texttt{in}\ \texttt{val}\ 0 \end{aligned}$$

the outcome of the first lookup is 1, which is then bound to x_2, while the handler continues handling the continuation. Then, the update is ignored and finally, the handler applies the value case on 0 and terminates with a call of r`#update`. Note that *Eff* provides *resources* [2], which at this point trigger real-world effects and resume the continuation. The exact reduction sequence is given in Figure 2.

$\texttt{with}\ h\ \texttt{handle}$
$\quad \texttt{let}\ x_1 = (\texttt{let}\ x_2 = \underline{r\texttt{\#lookup}\ ()}\ \texttt{in}\ r\texttt{\#update}\, x_2)\ \texttt{in}\ \texttt{val}\ 0 \rightsquigarrow$
$\texttt{with}\ h\ \texttt{handle}$
$\quad \texttt{let}\ x_1 = (\texttt{let}\ \underline{x_2} = \underline{r\texttt{\#lookup}\ ()\ (y_1.\, \texttt{val}\ y_1)}\ \texttt{in}\ r\texttt{\#update}\, x_2)\ \texttt{in}\ \texttt{val}\ 0 \rightsquigarrow$
$\texttt{with}\ h\ \texttt{handle}$
$\quad \texttt{let}\ \underline{x_1} = \underline{r\texttt{\#lookup}\ ()\ (y_1.\, \texttt{let}\ x_2 = \texttt{val}\ y_1\ \texttt{in}\ r\texttt{\#update}\, x_2)}\ \texttt{in}\ \texttt{val}\ 0 \rightsquigarrow$
$\texttt{with}\ \underline{h}\ \texttt{handle}$
$\quad \underline{r\texttt{\#lookup}\ ()\ (y_1.\, \texttt{let}\ x_1 = (\texttt{let}\ x_2 = \texttt{val}\ y_1\ \texttt{in}\ r\texttt{\#update}\, x_2)\ \texttt{in}\ \texttt{val}\ 0)} \rightsquigarrow$
$\underline{(\texttt{fun}\ y_1 \mapsto \texttt{with}\ h\ \texttt{handle}\ (\texttt{let}\ x_1 = (\texttt{let}\ x_2 = \texttt{val}\ y_1\ \texttt{in}\ r\texttt{\#update}\, x_2)\ \texttt{in}\ \texttt{val}\ 0))}\ \underline{1} \rightsquigarrow$
$\texttt{with}\ h\ \texttt{handle}\ (\texttt{let}\ x_1 = (\texttt{let}\ \underline{x_2} = \underline{\texttt{val}\ 1}\ \texttt{in}\ r\texttt{\#update}\, x_2)\ \texttt{in}\ \texttt{val}\ 0) \rightsquigarrow$
$\texttt{with}\ h\ \texttt{handle}\ (\texttt{let}\ x_1 = \underline{r\texttt{\#update}\, 1}\ \texttt{in}\ \texttt{val}\ 0) \rightsquigarrow$
$\texttt{with}\ h\ \texttt{handle}\ (\texttt{let}\ \underline{x_1} = \underline{r\texttt{\#update}\, 1\ (y_2.\, \texttt{val}\ y_2)}\ \texttt{in}\ \texttt{val}\ 0) \rightsquigarrow$
$\texttt{with}\ \underline{h}\ \texttt{handle}\ (\underline{r\texttt{\#update}\, 1\ (y_2.\, \texttt{let}\ x_1 = \texttt{val}\ y_2\ \texttt{in}\ \texttt{val}\ 0)}) \rightsquigarrow$
$\underline{(\texttt{fun}\ y_2 \mapsto \texttt{with}\ h\ \texttt{handle}\ (\texttt{let}\ x_1 = \texttt{val}\ y_2\ \texttt{in}\ \texttt{val}\ 0))}\ \underline{()} \rightsquigarrow$
$\texttt{with}\ h\ \texttt{handle}\ (\texttt{let}\ \underline{x_1} = \underline{\texttt{val}\ ()}\ \texttt{in}\ \texttt{val}\ 0) \rightsquigarrow$
$\texttt{with}\ \underline{h}\ \texttt{handle}\ \underline{(\texttt{val}\ 0)} \rightsquigarrow \underline{r\texttt{\#update}\, 0} \rightsquigarrow r\texttt{\#update}\, 0\, (y_3.\, \texttt{val}\ y_3)$

Figure 2: The evaluation of `with` h `handle` c, as described in Example 2.1. We underline the active parts of each step and shorten `succ` 0 to 1. We can see how the operation call to r`#lookup` in the first line propagates outwards to the matching handler while its continuation builds up. Once the call reaches a handler, it is replaced with the handling term in which k is replaced by the further handled continuation.

2.2. **Types.** The types, which are also split into pure and potentially effectful (here called *dirty*) ones, are given by

$$\begin{aligned} \text{type } A, B &::= \texttt{bool} \mid \texttt{nat} \mid \texttt{unit} \mid \texttt{empty} \mid A \to \underline{C} \mid E^R \mid \underline{C} \Rightarrow \underline{D} \\ \text{dirty type } \underline{C}, \underline{D} &::= A \,!\, \Delta \end{aligned}$$

We have the usual ground types and the function type $A \to \underline{C}$ of functions that take expressions of type A and perform computations of type $\underline{C}$. Next, we have the *effect type* E^R of instances of effect E from a *region* R, which is just a *non-empty* finite set of instances $\{\texttt{ins}_1, \ldots, \texttt{ins}_n\} \subseteq \mathcal{I}_E$. Finally, we have the *handler type* $\underline{C} \Rightarrow \underline{D}$ of handlers that take a computation of type $\underline{C}$ and transform it into a computation of type $\underline{D}$. We call $\underline{C}$ the *incoming* and $\underline{D}$ the *outgoing* type. Finally, dirty type $A \,!\, \Delta$ contains computations that return values of type A and may cause effects described by a *dirt* Δ, which is a set of operations

$$\{\texttt{ins}_1\texttt{\#op}_1, \ldots, \texttt{ins}_n\texttt{\#op}_n\}$$

To lighten the syntax, we write $A_1 \to A_2 \,!\, \Delta$ as $A_1 \xrightarrow{\Delta} A_2$ where we also omit the outer braces around Δ.

Note that for simplicity, the types of core *Eff* are monomorphic. However, we are going to shift to polymorphic types with type, region and dirt parameters when we start with type inference in Section 3.

2.3. **Subtyping.** As in most effect systems, we need to take care of the *poisoning problem* [34]. For example, what type should we give to *ignore* in

$$\begin{aligned} &\texttt{let } \mathit{ignore} = \texttt{val } (\texttt{fun } \mathit{msg} \mapsto \texttt{val ()}) \texttt{ in} \\ &\texttt{let } f = \texttt{if } b \texttt{ then } (\texttt{val } \mathit{ignore}) \texttt{ else } (\texttt{val std\#write}) \texttt{ in} \\ &\texttt{val } \mathit{ignore} \end{aligned}$$

for some suitable boolean b? If we give it the type $\texttt{string} \to \texttt{unit}$ (for this example, we allow ourselves the type `string` of strings), we cannot type the conditional statement as the two branches cannot have the same type, but if we give it the type $\texttt{string} \xrightarrow{\texttt{std\#write}} \texttt{unit}$, we lose information that the final result is a pure function.

The simplest antidote for the poisoning problem is to allow subtyping, so that we may give `ignore` the type with an empty dirt, and use subsumption to suitably enlarge this dirt in the conditional statement.

Subtyping also solves a similar problem with regions of handled instances. Consider the computation

$$\begin{aligned} &\texttt{let } u = \texttt{val ins in} \\ &\texttt{let } v = \texttt{if } b \texttt{ then val } u \texttt{ else val ins}' \texttt{ in} \\ &\texttt{val } (\texttt{handler val } x \mapsto \cdots \mid u\texttt{\#op } x\, k \mapsto \cdots) \end{aligned}$$

Without subtyping we are forced to give both u and v the type $E^{\{\texttt{ins},\texttt{ins}'\}}$. Therefore, as discussed in Section 1.1.4, the type of u does not tell us whether h handles `ins#op` or `ins'#op`, and so we must assume that both may remain unhandled. With subtyping we may give u the type $E^{\{\texttt{ins}\}}$, which makes it clear that h handles `ins#op`.

For our purposes, it is enough to use *structural* subtyping [7], where we relate only types of the same shape. The subtyping relations between types and between dirty types

$$\textsc{Sub-bool}\ \frac{}{\texttt{bool} \leqslant \texttt{bool}} \qquad \textsc{Sub-nat}\ \frac{}{\texttt{nat} \leqslant \texttt{nat}} \qquad \textsc{Sub-unit}\ \frac{}{\texttt{unit} \leqslant \texttt{unit}} \qquad \textsc{Sub-empty}\ \frac{}{\texttt{empty} \leqslant \texttt{empty}}$$

$$\textsc{Sub-}{\to}\ \frac{A' \leqslant A \qquad \underline{C} \leqslant \underline{C'}}{A \to \underline{C} \leqslant A' \to \underline{C'}}$$

$$\textsc{Sub-}E\ \frac{R \subseteq R'}{E^R \leqslant E^{R'}} \qquad \textsc{Sub-}{\Rightarrow}\ \frac{\underline{C'} \leqslant \underline{C} \qquad \underline{D} \leqslant \underline{D'}}{\underline{C} \Rightarrow \underline{D} \leqslant \underline{C'} \Rightarrow \underline{D'}} \qquad \textsc{Sub-!}\ \frac{A \leqslant A' \qquad \Delta \subseteq \Delta'}{A \,!\, \Delta \leqslant A' \,!\, \Delta'}$$

Figure 3: The inductive definition of the subtyping relations $A \leqslant A'$ and $\underline{C} \leqslant \underline{C'}$.

are defined in Figure 3. Sometimes, we shall be interested in types that have the same shape. So, we define $\approx$ as the equivalence relation on types, generated by $\leqslant$. Equivalence classes of $\approx$ are called *skeletons* [28].

2.4. **Effect system.** Our effect system is built on two typing judgements, defined in Figure 4. The judgement $\Gamma \vdash_\Sigma e : A$ states that in context Γ and *signature* Σ, an expression e has a type A. The judgement $\Gamma \vdash_\Sigma c : \underline{C}$ states a similar thing for a computation c and a dirty type $\underline{C}$. In both cases, the context Γ is a unique assignment of (pure) types to variables, while the signature Σ consists of *effect signatures* $\Sigma(E)$ for each effect E. These are of the form

$$\{\mathtt{op}_1 : A^{\mathtt{op}_1} \to B^{\mathtt{op}_1}, \ldots, \mathtt{op}_n : A^{\mathtt{op}_n} \to B^{\mathtt{op}_n}\}$$

and assign a *parameter type* $A^{\mathtt{op}}$ and a *result type* $B^{\mathtt{op}}$ to each listed operation $\mathtt{op}$. For example, the effect signatures for references is:

$$\Sigma(\mathtt{ref}) = \{\mathtt{lookup} : \mathtt{unit} \to \mathtt{nat}, \mathtt{update} : \mathtt{nat} \to \mathtt{unit}\}$$

For technical reasons, we assume that both the parameter and the result type for each operation do not contain any regions or dirt, which limits them to the basic ground types such as `nat` or `unit`. We further discuss this restriction in Remark 3.4.

The purpose of the presented effect system is to offer guarantees on the behaviour of programs, not (yet) to lead to an efficient inference algorithm. One sign of that is rules like Val, Inst or Pred, where we assign types that are safe, but much coarser than needed. A more obvious sign is the rule LetVal, where we employ a very naive form of let-polymorphism that performs an explicit substitution. This is, of course, extremely inefficient, but lets us postpone the use of parameters to the inference rules, which use the more efficient variant with universally quantified types.

All the typing rules are standard except for:

Inst: in which we check that `ins` is contained in the region R that belongs to an effect E.

Op: in which we first check that e_1 and `op` belong to the same effect. Then, we need to check that the dirt Δ covers not just all possible operations that the operation call may cause (recall that R may contain more than one instance), but also any operations in the continuation c. We may assume that c has the same dirt, as we can use SubComp otherwise. We use the same reasoning in rules IfThenElse and Let.

With: where the handling construct is typed like an application, except that it is applied to a computation rather than an expression.

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}} \qquad \textsc{Zero}\ \frac{}{\Gamma \vdash 0 : \mathtt{nat}} \qquad \textsc{Succ}\ \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}\, e : \mathtt{nat}}$$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \textsc{Fun}\ \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathtt{fun}\, x \mapsto c : A \to \underline{C}} \qquad \textsc{Inst}\ \frac{\mathtt{ins} \in R \subseteq \mathcal{I}_E}{\Gamma \vdash \mathtt{ins} : E^R} \qquad \textsc{Hand} \text{ — in the text}$$

$$\textsc{SubExpr}\ \frac{\Gamma \vdash e : A \qquad A \leqslant A'}{\Gamma \vdash e : A'} \qquad \textsc{IfThenElse}\ \frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash c_1 : \underline{C} \qquad \Gamma \vdash c_2 : \underline{C}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \underline{C}} \qquad \textsc{IsZero}\ \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{iszero}\, e : \mathtt{bool} \,!\, \Delta}$$

$$\textsc{Pred}\ \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{pred}\, e : \mathtt{nat} \,!\, \Delta} \qquad \textsc{Absurd}\ \frac{\Gamma \vdash e : \mathtt{empty}}{\Gamma \vdash \mathtt{absurd}\, e : \underline{C}} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : A \to \underline{C} \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : \underline{C}}$$

$$\textsc{Val}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathtt{val}\, e : A \,!\, \Delta} \qquad \textsc{Op}\ \frac{\Gamma \vdash e_1 : E^R \qquad \mathtt{op} : A^{\mathtt{op}} \to B^{\mathtt{op}} \in \Sigma(E) \qquad \Gamma \vdash e_2 : A^{\mathtt{op}} \qquad \Gamma, y : B^{\mathtt{op}} \vdash c : A \,!\, \Delta \qquad \forall \mathtt{ins} \in R.\ \mathtt{ins\#op} \in \Delta}{\Gamma \vdash e_1 \mathtt{\#op}\, e_2\, (y.\, c) : A \,!\, \Delta}$$

$$\textsc{Let}\ \frac{\Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, x : A \vdash c_2 : B \,!\, \Delta}{\Gamma \vdash \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : B \,!\, \Delta} \qquad \textsc{LetVal}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash c[e/x] : \underline{C}}{\Gamma \vdash \mathtt{let\ val}\ x = e\ \mathtt{in}\ c : \underline{C}}$$

$$\textsc{LetRec}\ \frac{\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C} \qquad \Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}}{\Gamma \vdash \mathtt{let\ rec}\ f\, x = c_1\ \mathtt{in}\ c_2 : \underline{D}} \qquad \textsc{With}\ \frac{\Gamma \vdash e : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathtt{with}\ e\ \mathtt{handle}\ c : \underline{D}}$$

$$\textsc{SubComp}\ \frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \leqslant \underline{C}'}{\Gamma \vdash c : \underline{C}'}$$

Figure 4: The inductive definition of the typing judgements $\Gamma \vdash_\Sigma e : A$ and $\Gamma \vdash_\Sigma c : \underline{C}$. As the signature Σ does not change, we omit its display from all the judgements. The rule for handlers is given in the main text.

Hand**:** which is a bit more daunting, so we write it out separately:

$$\textsc{Hand}\ \frac{\Gamma, x : A \vdash c_v : B \,!\, \Delta' \qquad (\Psi_i)_i \qquad \Psi_\Delta}{\Gamma \vdash (\mathtt{handler\ val}\ x \mapsto c_v \mid (e_i \mathtt{\#op}_i\, x\, k \mapsto c_i)_i) : A \,!\, \Delta \Rightarrow B \,!\, \Delta'}$$

For a handler to be of type $A \,!\, \Delta \Rightarrow B \,!\, \Delta'$, we first check that it takes values of type A to computations of type $B \,!\, \Delta'$. Then, for each operation case $e_i \mathtt{\#op}_i\, x\, k \mapsto c_i$, we check the premises Ψ_i, comprising:

$$\Gamma \vdash e_i : E_i^{R_i} \qquad \mathtt{op}_i : A^{\mathtt{op}_i} \to B^{\mathtt{op}_i} \in \Sigma(E_i) \qquad \Gamma, x : A^{\mathtt{op}_i}, k : B^{\mathtt{op}_i} \to B \,!\, \Delta' \vdash c_i : B \,!\, \Delta'$$

Like in Op, we check that e_i and $\mathtt{op}_i$ belong to the same effect E_i. Then, the handling computation c_i needs to have the same type $B \,!\, \Delta'$, assuming that parameter x is of type $A^{\mathtt{op}_i}$, and the continuation k of type $B^{\mathtt{op}_i} \to B \,!\, \Delta'$. Observe that since the continuation is further handled, it already has the outgoing type.

Finally, in Ψ_Δ we check that any operation in the incoming dirt Δ that is not guaranteed to be caught by the handler must appear in the outgoing dirt Δ' as well. As discussed in Section 1.1.4, an operation `ins#op` will be (if not sooner) surely caught by the case for $\mathtt{ins}_i\#\mathtt{op}_i$ whenever $\mathtt{op} = \mathtt{op}_i$ and the region R_i is the singleton $\{\mathtt{ins}_i\}$. Thus, we define Ψ_Δ to be

$$\forall \mathtt{ins\#op} \in \Delta.\ (\mathtt{ins} \in \bigcup\!\!\!\!\bullet_{\mathtt{op}=\mathtt{op}_i} R_i) \vee (\mathtt{ins\#op} \in \Delta')$$

where the *singleton union* $\cup\!\!\!\!\bullet$ behaves like a union, except that it considers only singletons. For example, $\{\mathtt{ins}_1, \mathtt{ins}_2\} \cup\!\!\!\!\bullet \{\mathtt{ins}_2\} \cup\!\!\!\!\bullet \{\mathtt{ins}_3, \mathtt{ins}_4\} \cup\!\!\!\!\bullet \{\mathtt{ins}_5\} = \{\mathtt{ins}_2, \mathtt{ins}_5\}$. More precisely, we define:

$$\bigcup\!\!\!\!\bullet_i R_i = \{\mathtt{ins} \mid \{\mathtt{ins}\} \in \{R_i\}_i\}$$

The given effect system is then safe with respect to the operational semantics: a computation $\vdash c : A \,!\, \Delta$ can only call operations from Δ. In particular, if Δ is empty, then c is guaranteed to be pure, though it may diverge.

Theorem 2.2 (Safety)**.** *If for a computation c, the typing judgement $\vdash c : A \,!\, \Delta$ holds, then either:*

- *c is of the form $\mathtt{val}\ e$ for some expression $\vdash e : A$, or*
- *c is of the form $\mathtt{ins\#op}\, e\, (y.\, c')$ for some $\mathtt{ins\#op} \in \Delta$, or*
- *there exists a computation c' such that $c \rightsquigarrow c'$ and $\vdash c' : A \,!\, \Delta$.*

We do not give a proof of Theorem 2.2 in this paper. Instead, a full formalization of core *Eff* in Twelf is available at `https://github.com/matijapretnar/twelf-eff/`

Example 2.3. To illustrate the type system, let us revisit Example 2.1. First, assume that the reference $r \in \mathcal{I}_{\mathtt{ref}}$ is given the type $\mathtt{ref}^{\{r\}}$. Then, the stateful computation c has the dirty type $\mathtt{nat} \,!\, \{r\#\mathtt{lookup}, r\#\mathtt{update}\}$, while the handler h has the type

$$(\mathtt{nat} \,!\, \{r\#\mathtt{lookup}, r\#\mathtt{update}\}) \Rightarrow (\mathtt{unit} \,!\, \{r\#\mathtt{update}\})$$

as it handles both `lookup` and `update`, but then triggers `update` in the value case. This `update` also changes the type of computation from `nat` to `unit`.

If the reference r has a less precise type $\mathtt{ref}^{\{r,r'\}}$, the dirt of c is

$$\Delta \stackrel{\text{def}}{=} \{r\#\mathtt{lookup}, r'\#\mathtt{lookup}, r\#\mathtt{update}, r'\#\mathtt{update}\}$$

while the best type we can give to h is $(\mathtt{nat} \,!\, \Delta) \Rightarrow (\mathtt{unit} \,!\, \Delta)$. Since the region of r is not a singleton, we unfortunately cannot give any guarantees on the handled operations.

3. Inferring constraints

Turning to our inference algorithm, we first describe a collection of syntax-directed inference rules that are readily transcribed into a recursive function that infers a type and a set of constraints from a given term.

3.1. **Parametric types.** As indicated in Section 2.2, we switch to a language that is more suited for inference:

$$\begin{aligned} \text{type } A, B &::= \alpha \mid \texttt{bool} \mid \texttt{nat} \mid \texttt{unit} \mid \texttt{empty} \mid A \to \underline{C} \mid E^{\dot\rho} \mid \underline{C} \Rightarrow \underline{D} \\ \text{dirty type } \underline{C}, \underline{D} &::= A \,!\, \Delta \\ \text{dirt } \Delta &::= \{\texttt{op}_1 : \rho_1, \ldots, \texttt{op}_n : \rho_n \mid \delta^{\mathcal{O}}\} \end{aligned}$$

From now on, we refer to types and dirt from Section 2.2, which contain no parameters, as *closed* ones.

To enable polymorphism, types are extended with the *type parameters* α. Then, regions are not just extended, but completely replaced with *region parameters* ρ, as this greatly simplifies the inference. We are going to capture the information about instances using constraints instead. Recall that regions R describing the possible instances in an effect type E^R are always inhabited. This information will prove useful in Section 5 as it allows further simplification. For this reason, we designate a special subset of region parameters, called *inhabited* and marked by $\dot\rho$.

Finally, we adopt a row-like [26] representation of dirt as described in Section 1.1.2. The first part is a set of operation symbols together with a region parameter that captures the (possibly empty) region of all the instances on which these symbols are used. This is similar to before, except that operations are grouped by their operation symbols. The reason for this grouping is that we are always able to precisely determine the operation symbols, but not the instances of called or handled operations.

The second part consists of a single *dirt parameter* δ, intended to capture the rest of operations. If the first part is empty, we write the dirt simply as δ. To keep track of the operation symbols captured by δ and ensure that it does not capture any symbols from the first part, we sometimes write the parameter as $\delta^{\mathcal{O}}$ to emphasise the set $\mathcal{O}$ of symbols not captured by δ (though $\mathcal{O}$ can always be reconstructed by looking at the first part of any dirt in which δ appears because our algorithm ensures that all such dirts consistently list the same operation symbols).

Any additional information is captured with *constraints*. For example, a conditional can call any operation that one of its branches does. If these two branches cause dirt captured by δ_1 and δ_2, we can represent the dirt of the whole conditional with a fresh parameter δ together with constraints $\delta_1 \leqslant \delta$ and $\delta_2 \leqslant \delta$. Constraints can be of one of the following five kinds:

- $A \leqslant A'$ states that the type A needs to be smaller than A',
- $\underline{C} \leqslant \underline{C'}$ states the same for dirty types,
- $\rho \leqslant \rho' \cup \bigcup\!\!\!\bullet_i \dot\rho_i$ is a generalisation of the inequality $\rho \leqslant \rho'$ due to handlers. It states that all instances from ρ are either in ρ' or in some $\dot\rho_i$ that is a singleton,
- $\texttt{ins} \in \rho \cup \bigcup\!\!\!\bullet_i \dot\rho_i$ similarly states that $\texttt{ins}$ is either in ρ or in some $\dot\rho_i$ that is a singleton,
- $\Delta \leqslant \Delta'$ states that the dirt Δ is smaller than Δ'.

In the right-hand side $\rho \cup \bigcup\!\!\!\bullet_i \dot\rho_i$ of two region constraints, we refer to ρ as the *covering*, and to $\dot\rho_i$ as the *handled* region parameters.

Unlike the subtyping relation, constraints do not have any inherent reasoning principles, but are just a way of writing down the relationship between parameters. Instead, we give constraints a meaning by specifying their solutions.

Definition 3.1. A *closed substitution* σ is a partial mapping that maps: each type parameter α to a closed type $\sigma(\alpha)$, each region parameter ρ to a closed region $\sigma(\rho)$, each inhabited

region parameter $\dot{\rho}$ to a non-empty closed region $\sigma(\dot{\rho})$, and each dirt parameter $\delta^{\mathcal{O}}$ to a closed dirt $\sigma(\delta^{\mathcal{O}})$ that contains no operations with operation symbols in $\mathcal{O}$.

We write substitutions by specifying a set of mappings of parameters, for example

$$\{\alpha_1 \mapsto \mathtt{nat}, \alpha_2 \mapsto \mathtt{unit}, \rho \mapsto \emptyset, \delta \mapsto \{\mathtt{ins\#op}\}\}$$

We can extend a closed substitution to other constructs by:

$$\begin{aligned} \sigma(\mathtt{bool}) &= \mathtt{bool} & \sigma(A \to \underline{C}) &= \sigma(A) \to \sigma(\underline{C}) \\ \sigma(\mathtt{nat}) &= \mathtt{nat} & \sigma(E^{\dot{\rho}}) &= E^{\sigma(\dot{\rho})} \\ \sigma(\mathtt{unit}) &= \mathtt{unit} & \sigma(\underline{C} \Rightarrow \underline{D}) &= \sigma(\underline{C}) \Rightarrow \sigma(\underline{D}) \\ \sigma(\mathtt{empty}) &= \mathtt{empty} & \sigma(A \,!\, \Delta) &= \sigma(A) \,!\, \sigma(\Delta) \end{aligned}$$

$$\sigma(\{\mathtt{op}_1 : \rho_1, \ldots, \mathtt{op}_n : \rho_n \mid \delta\}) = \Big(\bigcup_{i=1}^{n} \{\mathtt{ins\#op}_i \mid \mathtt{ins} \in \sigma(\rho_i)\} \Big) \cup \sigma(\delta)$$

Definition 3.2. A closed substitution σ is a *solution* of a set of constraints $\mathcal{C}$, which we write as $\sigma \models \mathcal{C}$, if it satisfies all the constraints in $\mathcal{C}$. This is defined by:

$$\begin{aligned} \sigma \models A \leqslant A' &\iff \sigma(A) \leqslant \sigma(A') \\ \sigma \models \underline{C} \leqslant \underline{C}' &\iff \sigma(\underline{C}) \leqslant \sigma(\underline{C}') \\ \sigma \models \rho \leqslant \rho' \cup \bigcup\!\!\!\bullet_i \dot{\rho}_i &\iff \sigma(\rho) \subseteq \sigma(\rho') \cup \bigcup\!\!\!\bullet_i \sigma(\dot{\rho}_i) \\ \sigma \models \mathtt{ins} \in \rho \cup \bigcup\!\!\!\bullet_i \dot{\rho}_i &\iff \mathtt{ins} \in \sigma(\rho) \cup \bigcup\!\!\!\bullet_i \sigma(\dot{\rho}_i) \\ \sigma \models \Delta \leqslant \Delta' &\iff \sigma(\Delta) \subseteq \sigma(\Delta') \end{aligned}$$

A parametric type A together with a set of constraints $\mathcal{C}$ between its parameters then describes a family of closed types, obtained by taking all instances $\sigma(A)$ of A for all solutions $\sigma \models \mathcal{C}$, and all their supertypes. More precisely, we define

$$[\![A \mid \mathcal{C}]\!] \stackrel{\text{def}}{=} \{A' \mid \sigma(A) \leqslant A', \sigma \models \mathcal{C}\} \qquad [\![\underline{C} \mid \mathcal{C}]\!] \stackrel{\text{def}}{=} \{\underline{C}' \mid \sigma(\underline{C}) \leqslant \underline{C}', \sigma \models \mathcal{C}\}$$

Our aim now is to take an expression e and compute a type A and a set of constraints $\mathcal{C}$ such that the set of all possible types A' we can assign to e is captured exactly by $[\![A \mid \mathcal{C}]\!]$.

3.2. **Inference rules.** We infer types and constraints using syntax-directed inference rules of the form $\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}$ for expressions and $\Gamma; \Xi \vdash_F c : \underline{C} \mid \mathcal{C}$ for computations, defined in Figure 5. Here, $\mathcal{C}$ is a set of constraints, F is a set of all fresh parameters introduced in the derivation, and Ξ is the *polymorphic context*, which is collection of unique assignments $x_i : \forall F_i.\ A_i \mid \mathcal{C}_i$ of *type schemes* to variables (assumed to be different from the ones in Γ). As in typing judgements, we assume (though never write) a fixed signature Σ.

The type schemes are similar to polymorphic types of ML, which are types, universally quantified over a given set of type parameters. In our case, we may also quantify over region and dirt parameters, but we need to keep information about the constraints these parameters need to satisfy. Even though the ordinary context Γ can be seen as a particular instance of Ξ, we keep the two separate in order to relate the inference judgements to typing judgements, as the latter employ only Γ.

Though $\mathcal{C}$ and F are sets, we sometimes write them as sequences to save space. For example, we write F_1, F_2, α, δ instead of $F_1 \cup F_2 \cup \{\alpha, \delta\}$. We also assume that all parameters

listed in F are distinct and this implies the usual freshness conditions [20, p. 321]. For example, the above sequence implies that sets F_1 and F_2 are disjoint and do not contain α or δ. In particular, in the rule Cstr-PolyVar, we implicitly rename any bound parameters F so that a fresh copy is obtained at each use.

As announced at the beginning of Section 3.1, regions and dirts have a fixed representation with parameters. Thus in Cstr-Inst, we assign each instance a fresh region parameter and add a suitable constraint. Similarly, we cannot simply state that the dirt of `val` is empty. Instead, in Cstr-Val, we assign it a fresh dirt parameter δ that needs to satisfy no constraints. That means that we may replace δ by anything, including the empty set.

Though we get an equivalent set of constraints in the rule Cstr-Op if we use a single region parameter, we introduce two for technical reasons, discussed in Section 5.2. The rule Cstr-With is analogous to Cstr-App.

Otherwise, the rules for the standard constructs are similar to ones in the Hindley-Milner algorithm [20, p. 322], except that we need to use (correctly oriented) inequalities instead of equalities in the constraints. In Cst-LetVal we can safely generalize over all the fresh parameters F_1 generated while inferring the type of e because they are guaranteed to be distinct from any parameters appearing in Γ.

This leaves us with

$$\textsc{Cstr-Hand}\quad \frac{\Gamma, x : \alpha_{\text{in}}; \Xi \vdash_{F_v} c_v : \underline{D}_v \mid \mathcal{C}_v \qquad (\Psi_i)_i}{\Gamma; \Xi \vdash_F (\texttt{handler val}\ x \mapsto c_v \mid (e_i \texttt{\#op}_i\, x\, k \mapsto c_i)_i) : \underline{C} \Rightarrow \underline{D} \mid \mathcal{C}}$$

To start, we take type parameters α_{in} and α_{out} to represent the incoming and outgoing type of the handler. Next, we take $\mathcal{O}$ to be the set of all distinct operation symbols listed in operation cases. For each $\texttt{op} \in \mathcal{O}$, we take fresh parameters $\rho_{\text{in}}^{\texttt{op}}$ and $\rho_{\text{out}}^{\texttt{op}}$ that represent the region assigned to `op` in the incoming and outgoing dirt of the handler. Finally, we take fresh parameters $\delta_{\text{in}}^{\mathcal{O}}$ and $\delta_{\text{out}}^{\mathcal{O}}$ to represent the rest of incoming and outgoing dirt. The incoming and outgoing types are then

$$\underline{C} \stackrel{\text{def}}{=} \alpha_{\text{in}} \,!\, \{(\texttt{op} : \rho_{\text{in}}^{\texttt{op}})_{\texttt{op} \in \mathcal{O}} \mid \delta_{\text{in}}\} \qquad \text{and} \qquad \underline{D} \stackrel{\text{def}}{=} \alpha_{\text{out}} \,!\, \{(\texttt{op} : \rho_{\text{out}}^{\texttt{op}})_{\texttt{op} \in \mathcal{O}} \mid \delta_{\text{out}}\}$$

After introducing the necessary parameters, we infer the type and constraints of the value case. Next, for each operation case, in the premises Ψ_i, consisting of:

$$\Gamma \vdash_{F_i} e_i : A_i \mid \mathcal{C}_i \qquad \texttt{op}_i : A^{\texttt{op}_i} \to B^{\texttt{op}_i} \in \Sigma(E_i) \qquad \Gamma, x : A^{\texttt{op}_i}, k : B^{\texttt{op}_i} \to \underline{D} \vdash_{F_i'} c_i : \underline{D}_i \mid \mathcal{C}_i'$$

we check the suitability of operation, and infer the types and constraints of the handled instance e_i and of the operation case c_i.

We end up with the a set of constraints $\mathcal{C}$ consisting of the following five parts:

- constraints $\mathcal{C}_v$ inherited from the value case and constraints $\mathcal{C}_i$ and $\mathcal{C}_i'$ inherited from each operation case;
- constraint $\underline{D}_v \leqslant \underline{D}$ stating that the outgoing type $\underline{D}$ subsumes the type of the value case and constraints $\underline{D}_i \leqslant \underline{D}$ stating the same for all the operation cases;
- constraints $A_i \leqslant E^{\dot{\rho}_i}$ stating that the type A_i of the instance expression e_i is subsumed by the effect type $E^{\dot{\rho}_i}$ for some fresh $\dot{\rho}_i$;
- constraints $\rho_{\text{in}}^{\texttt{op}} \leqslant \rho_{\text{out}}^{\texttt{op}} \cup \bigcup_{\texttt{op} = \texttt{op}_i} \dot{\rho}_i$ for each $\texttt{op} \in \mathcal{O}$ — the outgoing dirt must be big enough to cover all operations in the incoming dirt that are not surely handled by one of the operation cases; and
- a constraint $\delta_{\text{in}} \leqslant \delta_{\text{out}}$ — any operation that is not listed in a handler cannot be handled, so must appear in the outgoing dirt as well.

$$\textsc{Cstr-Var}\ \frac{(x : A) \in \Gamma}{\Gamma; \Xi \vdash_\emptyset x : A \mid \emptyset} \qquad \textsc{Cstr-PolyVar}\ \frac{(x : \forall F.\ A \mid \mathcal{C}) \in \Xi}{\Gamma; \Xi \vdash_F x : A \mid \mathcal{C}} \qquad \textsc{Cstr-True}\ \frac{}{\Gamma; \Xi \vdash_\emptyset \mathtt{true} : \mathtt{bool} \mid \emptyset} \qquad \textsc{Cstr-False}\ \frac{}{\Gamma; \Xi \vdash_\emptyset \mathtt{false} : \mathtt{bool} \mid \emptyset}$$

$$\textsc{Cstr-Zero}\ \frac{}{\Gamma; \Xi \vdash_\emptyset 0 : \mathtt{nat} \mid \emptyset} \qquad \textsc{Cstr-Succ}\ \frac{\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}}{\Gamma; \Xi \vdash_F \mathtt{succ}\ e : \mathtt{nat} \mid \mathcal{C}, A \leqslant \mathtt{nat}} \qquad \textsc{Cstr-Unit}\ \frac{}{\Gamma; \Xi \vdash_\emptyset () : \mathtt{unit} \mid \emptyset}$$

$$\textsc{Cstr-Fun}\ \frac{\Gamma, x : \alpha; \Xi \vdash_F c : \underline{C} \mid \mathcal{C}}{\Gamma; \Xi \vdash_{F,\alpha} \mathtt{fun}\ x \mapsto c : \alpha \to \underline{C} \mid \mathcal{C}} \qquad \textsc{Cstr-Inst}\ \frac{}{\Gamma; \Xi \vdash_{\dot\rho} \mathtt{ins} : E^{\dot\rho} \mid \mathtt{ins} \in \dot\rho} \qquad \textsc{Cstr-Hand} \text{ — in the text}$$

$$\textsc{Cstr-IfThenElse}\ \frac{\Gamma; \Xi \vdash_F e : A \mid \mathcal{C} \qquad \Gamma; \Xi \vdash_{F_1} c_1 : \underline{C}_1 \mid \mathcal{C}_1 \qquad \Gamma; \Xi \vdash_{F_2} c_2 : \underline{C}_2 \mid \mathcal{C}_2}{\Gamma; \Xi \vdash_{F,F_1,F_2,\alpha,\delta} \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \alpha\ !\ \delta \mid \mathcal{C}, \mathcal{C}_1, \mathcal{C}_2, A \leqslant \mathtt{bool}, \underline{C}_1 \leqslant (\alpha\ !\ \delta), \underline{C}_2 \leqslant (\alpha\ !\ \delta)}$$

$$\textsc{Cstr-IsZero}\ \frac{\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}}{\Gamma; \Xi \vdash_{F,\delta} \mathtt{iszero}\ e : \mathtt{bool}\ !\ \delta \mid \mathcal{C}, A \leqslant \mathtt{nat}} \qquad \textsc{Cstr-Pred}\ \frac{\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}}{\Gamma; \Xi \vdash_{F,\delta} \mathtt{pred}\ e : \mathtt{nat}\ !\ \delta \mid \mathcal{C}, A \leqslant \mathtt{nat}}$$

$$\textsc{Cstr-Absurd}\ \frac{\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}}{\Gamma; \Xi \vdash_{F,\alpha,\delta} \mathtt{absurd}\ e : \alpha\ !\ \delta \mid \mathcal{C}, A \leqslant \mathtt{empty}} \qquad \textsc{Cstr-App}\ \frac{\Gamma; \Xi \vdash_{F_1} e_1 : A_1 \mid \mathcal{C}_1 \qquad \Gamma; \Xi \vdash_{F_2} e_2 : A_2 \mid \mathcal{C}_2}{\Gamma; \Xi \vdash_{F_1,F_2,\alpha,\delta} e_1\ e_2 : \alpha\ !\ \delta \mid \mathcal{C}_1, \mathcal{C}_2, A_1 \leqslant (A_2 \xrightarrow{\delta} \alpha)}$$

$$\textsc{Cstr-Val}\ \frac{\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}}{\Gamma; \Xi \vdash_{F,\delta} \mathtt{val}\ e : A\ !\ \delta \mid \mathcal{C}}$$

$$\textsc{Cstr-Op}\ \frac{\mathtt{op} : A^{\mathtt{op}} \to B^{\mathtt{op}} \in \Sigma(E) \qquad \Gamma; \Xi \vdash_{F_1} e_1 : A_1 \mid \mathcal{C}_1 \qquad \Gamma; \Xi \vdash_{F_2} e_2 : A_2 \mid \mathcal{C}_2 \qquad \Gamma, y : B^{\mathtt{op}}; \Xi \vdash_F c : A\ !\ \Delta \mid \mathcal{C}}{\Gamma; \Xi \vdash_{F_1,F_2,F,\rho,\dot\rho,\delta} e_1 \mathtt{\#op}\, e_2\, (y.\, c) : A\ !\ \{\mathtt{op} : \rho \mid \delta\} \mid \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}, A_1 \leqslant E^{\dot\rho}, A_2 \leqslant A^{\mathtt{op}}, \dot\rho \leqslant \rho, \Delta \leqslant \{\mathtt{op} : \rho \mid \delta\}}$$

$$\textsc{Cstr-Let}\ \frac{\Gamma; \Xi \vdash_{F_1} c_1 : A\ !\ \Delta_1 \mid \mathcal{C}_1 \qquad \Gamma, x : \alpha; \Xi \vdash_{F_2} c_2 : B\ !\ \Delta_2 \mid \mathcal{C}_2}{\Gamma; \Xi \vdash_{F_1,F_2,\alpha,\delta} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : B\ !\ \delta \mid \mathcal{C}_1, \mathcal{C}_2, A \leqslant \alpha, \Delta_1 \leqslant \delta, \Delta_2 \leqslant \delta}$$

$$\textsc{Cstr-LetVal}\ \frac{\Gamma; \Xi \vdash_{F_1} e : A \mid \mathcal{C}_1 \qquad \Gamma; \Xi, (x : \forall F_1.\ A \mid \mathcal{C}_1) \vdash_{F_2} c : \underline{C} \mid \mathcal{C}_2}{\Gamma; \Xi \vdash_{F_2} \mathtt{let\ val}\ x = e\ \mathtt{in}\ c : \underline{C} \mid \mathcal{C}_2}$$

$$\textsc{Cstr-LetRec}\ \frac{\Gamma, f : \alpha_1 \xrightarrow{\delta} \alpha_2, x : \alpha_1; \Xi \vdash_{F_1} c_1 : \underline{C} \mid \mathcal{C}_1 \qquad \Gamma, f : \alpha_1 \xrightarrow{\delta} \alpha_2; \Xi \vdash_{F_2} c_2 : \underline{D} \mid \mathcal{C}_2}{\Gamma; \Xi \vdash_{F_1,F_2,\alpha_1,\alpha_2,\delta} \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 : \underline{D} \mid \mathcal{C}_1, \mathcal{C}_2, \underline{C} \leqslant \alpha_2\ !\ \delta}$$

$$\textsc{Cstr-With}\ \frac{\Gamma; \Xi \vdash_{F_1} e : A \mid \mathcal{C}_1 \qquad \Gamma; \Xi \vdash_{F_2} c : \underline{C} \mid \mathcal{C}_2}{\Gamma; \Xi \vdash_{F_1,F_2,\alpha,\delta} \mathtt{with}\ e\ \mathtt{handle}\ c : \alpha\ !\ \delta \mid \mathcal{C}_1, \mathcal{C}_2, A \leqslant (\underline{C} \Rightarrow \alpha\ !\ \delta)}$$

Figure 5: The inductive definition of the inference judgements $\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}$ and $\Gamma; \Xi \vdash_F c : \underline{C} \mid \mathcal{C}$. The rule for handlers is given in the main text.

The set F of all fresh parameters gathers all fresh parameters mentioned above and equals

$$F \stackrel{\text{def}}{=} F_v, (F_i)_i, (F'_i)_i, \alpha_{\text{in}}, \alpha_{\text{out}}, \delta_{\text{in}}, \delta_{\text{out}}, (\dot{\rho}_i)_i, (\rho_{\text{in}}^{\mathtt{op}})_{\mathtt{op}\in\mathcal{O}}, (\rho_{\text{out}}^{\mathtt{op}})_{\mathtt{op}\in\mathcal{O}}$$

Looking at the presented inference rules, we see that there is exactly one rule that applies to each language construct, so we can assign a unique type and set of constraints to each term (up to a renaming of fresh parameters). This allows us to turn the rules into a recursive function, which computes exactly the information about all the types we can assign to a given term:

Theorem 3.3 (Soundness & completeness)**.** *Let Γ be a closed context.*

- *If we have $\Gamma \vdash e : A'$ for some A', then $\Gamma \vdash_F e : A \mid \mathcal{C}$ holds and*
$$[\![A \mid \mathcal{C}]\!] = \{A'' \mid (\Gamma \vdash e : A'')\}$$
- *We have $\Gamma \vdash_F c : \underline{C} \mid \mathcal{C}$ if and only if $\Gamma \vdash c : \underline{C}'$ holds for some $\underline{C}'$. In this case*
$$[\![\underline{C} \mid \mathcal{C}]\!] = \{\underline{C}'' \mid (\Gamma \vdash c : \underline{C}'')\}$$

Remark 3.4. We limit the parameter and result types in the signature Σ to basic types because Σ is shared between typing judgements, which feature concrete regions and dirt, and inference judgements, which represent regions and dirt exclusively with parameters. There are three ways of reconciling this conflict:

- Extend the language of constraints with concrete upper bounds of the form $\rho \leqslant R$ and $\delta \leqslant \emptyset$. Then we may, say, replace any occurrence of $A^{\mathtt{op}} = E^{\{\mathtt{ins}_1, \mathtt{ins}_2\}}$ in inference judgements with $E^{\dot{\rho}}$ for some fresh $\dot{\rho}$, and add constraints $\mathtt{ins}_1 \in \dot{\rho}, \mathtt{ins}_2 \in \dot{\rho}, \dot{\rho} \leqslant \{\mathtt{ins}_1, \mathtt{ins}_2\}$, or replace $B^{\mathtt{op}} = \mathtt{unit} \xrightarrow{\mathtt{ins\#op}} \mathtt{unit}$ with a suitably fresh $\mathtt{unit} \xrightarrow{\mathtt{op}:\rho|\delta} \mathtt{unit}$ and constraints $\mathtt{ins} \in \rho, \rho \leqslant \{\mathtt{ins}\}, \delta \leqslant \emptyset$.
- Extend monomorphic types with *wild card* regions and dirt. For example, the type $\mathtt{exception}^{\top}$ would capture any exception, no matter which concrete region it comes from. Similarly, the dirt $\top\mathtt{\#raise}$ would mean that a computation raises some exception, though we do not know which one, while the dirt $\top$ would mean that any operation may get called.

  This solution agrees with practice [2], where most of types that appear in the signature are already basic, and the only two deviations so far are cooperative multithreading and delimited continuations, which both take functions as parameters. However, in both cases, we are not interested in imposing any limits on this dirt, so a wild card dirt would fit our goal.

  To add wild card regions and dirt to core *Eff*, we need to: (1) add subtyping rules such as $R \subseteq \top$, (2) extend Op with a condition that if $e_1 : E^{\top}$ then $\top\mathtt{\#op} \in \Delta$, and (3) adapt the rule Hand to ensure that any wild card dirt cannot be handled.
- Each of the above approaches has its advantages, and in addition, the two are compatible, so one may consider both in a practical implementation. However, the first approach leads a more powerful *prescriptive* effect system which is well beyond the scope of this paper (discussed more in the Conclusion), while the second one is routine but messy. Thus, we opt for the simplest option: prohibit any types that include regions and dirt from appearing in Σ.

4. Unifying constraints

Unfortunately, unlike in ML, subtyping prevents us from computing a principal type from a given set of constraints [24]. For example (ignoring dirt for a moment), if we just drop the constraint in the type

$$(\alpha \to \beta) \to (\alpha \to \beta) \mid \beta \leqslant \alpha$$

we get a type that captures too many closed types. On the other hand, the more restricted parametric type $(\gamma \to \gamma) \to (\gamma \to \gamma)$ is too strict because it fails to capture the type

$$(E^{\{\mathtt{ins},\mathtt{ins}'\}} \to E^{\{\mathtt{ins}\}}) \to (E^{\{\mathtt{ins},\mathtt{ins}'\}} \to E^{\{\mathtt{ins}\}})$$

or any of its subtypes (otherwise, the subtyping rules would imply $E^{\{\mathtt{ins},\mathtt{ins}'\}} \leqslant \gamma \leqslant E^{\{\mathtt{ins}\}}$).

Instead, the best we can do is to simplify the constraints as much as possible. First, we are going to reduce the constraints down to a more convenient and basic form. Constraints in this form always admit a solution, so the reduction also detects any unsolvable constraints.

Definition 4.1. A set of constraints $\mathcal{C}$ is *unified*, if all constraints are *decomposed* down to ones between parameters and the set is *closed* under logical implication. In particular, $\mathcal{C}$ may contain only constraints of the form

$$\alpha \leqslant \alpha', \qquad \rho \leqslant \rho' \cup \bigcup\!\!\bullet_i \dot{\rho}_i, \qquad \mathtt{ins} \in \rho \cup \bigcup\!\!\bullet_i \dot{\rho}_i, \qquad \delta \leqslant \delta',$$

and the following closure properties must hold:

- if $(\alpha_1 \leqslant \alpha_2) \in \mathcal{C}$ and $(\alpha_2 \leqslant \alpha_3) \in \mathcal{C}$, then $(\alpha_1 \leqslant \alpha_3) \in \mathcal{C}$;
- if $(\rho_1 \leqslant \rho_2 \cup \bigcup\!\!\bullet_{i\in I} \dot{\rho}_i) \in \mathcal{C}$ and $(\rho_2 \leqslant \rho_3 \cup \bigcup\!\!\bullet_{i\in J} \dot{\rho}_i) \in \mathcal{C}$, then $(\rho_1 \leqslant \rho_3 \cup \bigcup\!\!\bullet_{i\in I\cup J} \dot{\rho}_i) \in \mathcal{C}$;
- if $(\mathtt{ins} \in \rho_1 \cup \bigcup\!\!\bullet_{i\in I} \dot{\rho}_i) \in \mathcal{C}$ and $(\rho_1 \leqslant \rho_2 \cup \bigcup\!\!\bullet_{i\in J} \dot{\rho}_i) \in \mathcal{C}$, then $(\mathtt{ins} \in \rho_2 \cup \bigcup\!\!\bullet_{i\in I\cup J} \dot{\rho}_i) \in \mathcal{C}$;
- if $(\delta_1 \leqslant \delta_2) \in \mathcal{C}$ and $(\delta_2 \leqslant \delta_3) \in \mathcal{C}$, then $(\delta_1 \leqslant \delta_3) \in \mathcal{C}$.

To avoid circular types, we need to track all type parameters of the same shape. So, we assume that $\mathcal{C}$ is equipped with an equivalence relation $\approx_{\mathcal{C}}$ on type parameters, such that $(\alpha \leqslant \alpha') \in \mathcal{C}$ implies $\alpha \approx_{\mathcal{C}} \alpha'$. For solutions of a unified set of constraints $\mathcal{C}$, we consider only such $\sigma \models \mathcal{C}$, for which we also have $\sigma(\alpha) \approx \sigma(\alpha')$ for any $\alpha \approx_{\mathcal{C}} \alpha'$.

Lemma 4.2. *If a set of constraints $\mathcal{C}$ is unified, there exists a solution $\sigma \models \mathcal{C}$.*

In Figure 6, we define an algorithm `unify`, which is similar to Robinson's unification algorithm [20, p. 327], except that it returns a set of unified constraints in addition to the unifying substitution. The algorithm is defined recursively, passing around a triple $(\sigma; \mathcal{C}; \mathcal{Q})$, where σ is a unifying substitution of replaced parameters (initially taken to be the identity), $\mathcal{C}$ is a set of already unified constraints (initially $\mathcal{C}$ is empty while $\approx_{\mathcal{C}}$ is the identity relation on all type parameters in $\mathcal{Q}$), and the queue $\mathcal{Q}$ is a set of constraints yet to be processed.

The unifying substitution is not a closed substitution (Definition 3.1), which maps type parameters to closed types, etc., but one that maps them to parametric ones.

Definition 4.3. A *substitution* σ is a mapping that maps: each type parameter α to a type $\sigma(\alpha)$, each region parameter ρ to a region parameter $\sigma(\rho)$, each inhabited region parameter $\dot{\rho}$ to an inhabited region parameter $\sigma(\dot{\rho})$, and each dirt parameter $\delta^{\mathcal{O}}$ to a dirt $\sigma(\delta^{\mathcal{O}})$ of the form $\{(\mathtt{op} : \rho_{\mathtt{op}})_{\mathtt{op}\in\mathcal{O}'} \mid \delta'^{\mathcal{O}\cup\mathcal{O}'}\}$, where $\mathcal{O}'$ is disjoint from $\mathcal{O}$. This ensures that the dirt $\sigma(\delta^{\mathcal{O}})$ does not capture any operations from $\mathcal{O}$.

We write substitutions by listing a set of all the non-idempotent rules. We can extend a substitution from parameters to other constructs just like we extended closed substitutions.

$$
\begin{array}{l}
\texttt{unify}(\sigma;\ \mathcal{C};\ \cdot) = (\sigma,\ \mathcal{C}) \\
\texttt{unify}(\sigma;\ \mathcal{C};\ A \leqslant A', \mathcal{Q}) = \\
\quad \texttt{match}\ A \leqslant A'\ \texttt{with} \\
\quad\ \mid A \leqslant A \mapsto \texttt{unify}(\sigma;\ \mathcal{C};\ \mathcal{Q}) \\
\quad\ \mid \alpha \leqslant \alpha' \mapsto \texttt{unify}(\sigma;\ \mathcal{C} \uplus (\alpha \leqslant \alpha');\ \mathcal{Q}) \\
\quad\ \mid \alpha \leqslant A \mapsto \texttt{if}\ \exists \alpha' \approx_{\mathcal{C}} \alpha.\ \alpha' \in \texttt{free}(A)\ \texttt{then failure else} \\
\qquad \texttt{let}\ \sigma' = \{\alpha' \mapsto \texttt{refresh}(A) \mid \alpha' \approx_{\mathcal{C}} \alpha\}\ \texttt{in} \\
\qquad \texttt{let}\ \mathcal{C}' = \{(\alpha' \leqslant \alpha'') \in \mathcal{C} \mid \alpha \approx_{\mathcal{C}} \alpha' \approx_{\mathcal{C}} \alpha''\}\ \texttt{in} \\
\qquad \texttt{unify}(\sigma' \circ \sigma;\ \mathcal{C} - \mathcal{C}';\ \sigma'(\mathcal{Q}), \sigma'(\alpha) \leqslant A, \sigma'(\mathcal{C}')) \\
\quad\ \mid A \leqslant \alpha \mapsto \texttt{if}\ \exists \alpha' \approx_{\mathcal{C}} \alpha.\ \alpha' \in \texttt{free}(A)\ \texttt{then failure else} \\
\qquad \texttt{let}\ \sigma' = \{\alpha' \mapsto \texttt{refresh}(A) \mid \alpha' \approx_{\mathcal{C}} \alpha\}\ \texttt{in} \\
\qquad \texttt{let}\ \mathcal{C}' = \{(\alpha' \leqslant \alpha'') \in \mathcal{C} \mid \alpha \approx_{\mathcal{C}} \alpha' \approx_{\mathcal{C}} \alpha''\}\ \texttt{in} \\
\qquad \texttt{unify}(\sigma' \circ \sigma;\ \mathcal{C} - \mathcal{C}';\ \sigma'(\mathcal{Q}), A \leqslant \sigma'(\alpha), \sigma'(\mathcal{C}')) \\
\quad\ \mid E^{\dot\rho} \leqslant E^{\dot\rho'} \mapsto \texttt{unify}(\sigma;\ \mathcal{C} \uplus \dot\rho \leqslant \dot\rho';\ \mathcal{Q}) \\
\quad\ \mid (A \to \underline{C}) \leqslant (A' \to \underline{C}') \mapsto \texttt{unify}(\sigma;\ \mathcal{C};\ \mathcal{Q}, A' \leqslant A, \underline{C} \leqslant \underline{C}') \\
\quad\ \mid (\underline{C} \Rightarrow \underline{D}) \leqslant (\underline{C}' \Rightarrow \underline{D}') \mapsto \texttt{unify}(\sigma;\ \mathcal{C};\ \mathcal{Q}, \underline{C}' \leqslant \underline{C}, \underline{D} \leqslant \underline{D}') \\
\quad\ \mid \texttt{otherwise} \mapsto \texttt{failure} \\
\texttt{unify}(\sigma;\ \mathcal{C};\ A\,!\,\Delta \leqslant A'\,!\,\Delta', \mathcal{Q}) = \texttt{unify}(\sigma;\ \mathcal{C};\ \mathcal{Q}, A \leqslant A', \Delta \leqslant \Delta') \\
\texttt{unify}(\sigma;\ \mathcal{C};\ \rho \leqslant \rho' \cup \bigdot\bigcup_i \dot\rho_i, \mathcal{Q}) = \texttt{unify}(\sigma;\ \mathcal{C} \uplus (\rho \leqslant \rho' \cup \bigdot\bigcup_i \dot\rho_i);\ \mathcal{Q}) \\
\texttt{unify}(\sigma;\ \mathcal{C};\ \texttt{ins} \in \rho \cup \bigdot\bigcup_i \dot\rho_i, \mathcal{Q}) = \texttt{unify}(\sigma;\ \mathcal{C} \uplus (\texttt{ins} \in \rho \cup \bigdot\bigcup_i \dot\rho_i);\ \mathcal{Q}) \\
\texttt{unify}(\sigma;\ \mathcal{C};\ \{(\texttt{op} : \rho_{\texttt{op}})_{\texttt{op} \in \mathcal{O}} \mid \delta_1^{\mathcal{O}}\} \leqslant \{(\texttt{op} : \rho'_{\texttt{op}})_{\texttt{op} \in \mathcal{O}'} \mid \delta_2^{\mathcal{O}'}\}, \mathcal{Q}) = \\
\quad \texttt{if}\ \mathcal{O} = \mathcal{O}'\ \texttt{then} \\
\qquad \texttt{unify}(\sigma;\ \mathcal{C} \uplus \{\rho_{\texttt{op}} \leqslant \rho'_{\texttt{op}} \mid \texttt{op} \in \mathcal{O}\} \uplus (\delta_1 \leqslant \delta_2),\ \mathcal{Q}) \\
\quad \texttt{else} \\
\qquad \texttt{let}\ \sigma_1 = \texttt{if}\ \mathcal{O}' \subset \mathcal{O}\ \texttt{then}\ \{\}\ \texttt{else}\ \{\delta_1 \mapsto \{(\texttt{op} : \rho_{\texttt{op}})_{\texttt{op} \in \mathcal{O}' - \mathcal{O}} \mid \delta_1'^{\mathcal{O} \cup \mathcal{O}'}\}\}\ \texttt{in} \\
\qquad \texttt{let}\ \sigma_2 = \texttt{if}\ \mathcal{O} \subset \mathcal{O}'\ \texttt{then}\ \{\}\ \texttt{else}\ \{\delta_2 \mapsto \{(\texttt{op} : \rho'_{\texttt{op}})_{\texttt{op} \in \mathcal{O} - \mathcal{O}'} \mid \delta_2'^{\mathcal{O} \cup \mathcal{O}'}\}\}\ \texttt{in} \\
\qquad \texttt{let}\ \mathcal{C}' = \{(\delta \leqslant \delta') \in \mathcal{C} \mid \delta \sim_{\mathcal{C}} \delta' \sim_{\mathcal{C}} \delta_1\} \cup \{(\delta \leqslant \delta') \in \mathcal{C} \mid \delta \sim_{\mathcal{C}} \delta' \sim_{\mathcal{C}} \delta_2\}\ \texttt{in} \\
\qquad \texttt{unify}(\sigma_1 \circ \sigma_2 \circ \sigma;\ (\mathcal{C} - \mathcal{C}'),\ \sigma'(\mathcal{Q}) \cup \sigma'(\mathcal{C}'))
\end{array}
$$

Figure 6: Definition of the constraint unification algorithm `unify`.

This allows us to compose substitutions and additionally, to compose a closed substitution σ with a substitution σ' and obtain a closed substitution $\sigma \circ \sigma'$.

The unification works as follows. Take, say, a constraint $\alpha_1 \leqslant (\alpha_2 \xrightarrow{\delta_1} \texttt{bool})$. Since the rules of structural subtyping admit only comparison between types of the same shape, the only way to satisfy this constraint is to set α_1 to be some function type into `bool`. So, we decompose the constraint by replacing α_1 with some fresh $\alpha_3 \xrightarrow{\delta_2} \texttt{bool}$ and adding constraints $\alpha_2 \leqslant \alpha_3$ and $\delta_2 \leqslant \delta_1$

We add these constraints using the closure operator $\uplus$, defined in Figure 7, which extends $\mathcal{C}$ with a given constraint and all constraints it implies. This is done with a simplified version of an algorithm for computing the transitive closure of a graph [24], except that for region parameters, we also have to take instances and handled regions into account.

$$\begin{aligned}
\mathcal{C} \uplus (\alpha_1 \leqslant \alpha_2) &= \mathcal{C} \cup \{\alpha_1' \leqslant \alpha_2' \mid (\alpha_1' \leqslant \alpha_1) \in \mathcal{C}, (\alpha_2 \leqslant \alpha_2') \in \mathcal{C}\} \\
\mathcal{C} \uplus (\rho_1 \leqslant \rho_2 \cup \bigcup_{i \in I} \dot{\rho}_i) &= \mathcal{C} \cup \{\rho_1' \leqslant \rho_2' \cup \bigcup_{i \in I \cup J \cup K} \dot{\rho}_i \mid (\rho_1' \leqslant \rho_1 \cup \bigcup_{i \in J} \dot{\rho}_i) \in \mathcal{C}, (\rho_2 \leqslant \rho_2' \cup \bigcup_{i \in K} \dot{\rho}_i) \in \mathcal{C}\} \\
&\quad \cup \{\mathtt{ins} \in \rho_2' \cup \bigcup_{i \in I \cup J \cup K} \dot{\rho}_i \mid (\mathtt{ins} \leqslant \rho_1 \cup \bigcup_{i \in J} \dot{\rho}_i) \in \mathcal{C}, (\rho_2 \leqslant \rho_2' \cup \bigcup_{i \in K} \dot{\rho}_i) \in \mathcal{C}\} \\
\mathcal{C} \uplus (\mathtt{ins} \in \rho \cup \bigcup_{i \in I} \dot{\rho}_i) &= \mathcal{C} \cup \{\mathtt{ins} \in \rho' \cup \bigcup_{i \in I \cup J} \dot{\rho}_i \mid (\rho \leqslant \rho' \cup \bigcup_{i \in J} \dot{\rho}_i) \in \mathcal{C}\} \\
\mathcal{C} \uplus (\delta_1 \leqslant \delta_2) &= \mathcal{C} \cup \{\delta_1' \leqslant \delta_2' \mid (\delta_1' \leqslant \delta_1) \in \mathcal{C}, (\delta_2 \leqslant \delta_2') \in \mathcal{C}\}
\end{aligned}$$

Figure 7: Definition of the closure operator $\uplus$. In all cases, we assume that $\mathcal{C}$ implicitly contains all reflexive constraints, so that, for example in the first case, the added set also includes $\alpha_1 \leqslant \alpha_2$ and all constraints of the form $\alpha_1 \leqslant \alpha_2'$ and $\alpha_1' \leqslant \alpha_2$.

Before decomposition, we need to perform an *occur check* in order to prevent ill-formed types and ensure termination. This check is slightly more involved than usually [20, p. 327]. Say that we want to unify the set of constraints (let us again ignore dirt):

$$\{(\alpha_1 \to \alpha_2) \leqslant \alpha_3, \alpha_4 \leqslant \alpha_1, \alpha_4 \leqslant \alpha_3\}$$

We decompose α_3 as some $\alpha_5 \to \alpha_6$ and end up with constraints

$$\{\alpha_5 \leqslant \alpha_1, \alpha_2 \leqslant \alpha_6, \alpha_4 \leqslant \alpha_1, \alpha_4 \leqslant (\alpha_5 \to \alpha_6)\}$$

Now, we need to decompose α_4, then α_1, then α_5 and the whole thing repeats. So we need to check not only that α is not in $\mathtt{free}(A)$, the set of all parameters that occur in A, but also that no other parameters in its skeleton are.

When decomposing $\alpha \leqslant A$, we also replace each $\alpha' \approx_{\mathcal{C}} \alpha$ with a (distinct) fresh copy of A. We do this by repeatedly calling a function $\mathtt{refresh}(A)$ that on each call returns a type of the same form as A, except with all its type, region, and dirt parameters replaced with fresh ones. Because of this expansion, any constraints $\mathcal{C}' \subseteq \mathcal{C}$ that mention parameters from the skeleton of α are no longer decomposed. So we need to take them out of the otherwise unified $\mathcal{C}$ and put them back into the queue $\mathcal{Q}$.

On the remaining unified set of constraints $\mathcal{C} - \mathcal{C}'$, we define $\approx_{\mathcal{C}-\mathcal{C}'}$ to be as before, except that we remove the whole skeleton of α, and add the freshly generated parameters into the skeletons of matching parameters in A. For example, if the skeletons of $\approx_{\mathcal{C}}$ were

$$\{\alpha_1, \alpha_2\}, \{\alpha_3, \alpha_4, \alpha_5\}, \{\alpha_6\}$$

and we decompose the constraint $\alpha_1 \leqslant \alpha_3 \xrightarrow{\delta_1} \alpha_6$, we replace α_1 by $\alpha_7 \xrightarrow{\delta_2} \alpha_8$ and α_2 by $\alpha_9 \xrightarrow{\delta_3} \alpha_{10}$, and the skeletons of $\approx_{\mathcal{C}-\mathcal{C}'}$ are

$$\{\alpha_3, \alpha_4, \alpha_5, \alpha_7, \alpha_9\}, \{\alpha_6, \alpha_8, \alpha_{10}\}$$

When we unify a constraint $\alpha \leqslant \alpha'$, we merge the skeletons of α and α'.

For dirt constraints, we similarly expand both sides so to list the same operations. For example, if we have a constraint $\{\mathtt{op} : \rho_1 \mid \delta_1^{\mathtt{op}}\} \leqslant \{\mathtt{op}' : \rho_2 \mid \delta_2^{\mathtt{op}'}\}$, we replace δ_1 with some fresh $\{\mathtt{op}' : \rho_3 \mid \delta_3^{\mathtt{op},\mathtt{op}'}\}$ and δ_2 with $\{\mathtt{op} : \rho_4 \mid \delta_4^{\mathtt{op},\mathtt{op}'}\}$ and add constraints $\rho_1 \leqslant \rho_4$, $\rho_3 \leqslant \rho_2$, and $\delta_3 \leqslant \delta_4$. We define $\sim_{\mathcal{C}}$ to be the equivalence relation on dirt parameters generated by $(\delta \leqslant \delta') \in \mathcal{C}$, so that we can capture all related dirt parameters and expand them at the same time.

An example of a full run of the unification algorithm is given in Figure 8.

$$
\begin{aligned}
&\mathtt{unify}\Big(\Big\{\Big\};\quad \Big\{\Big\};\quad \Big\{(\alpha_1 \xrightarrow{\delta_1} \mathtt{nat}) \leqslant \alpha_2,\ \{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \delta_1)\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\delta_3} \mathtt{nat})\Big\};\quad \Big\{\Big\};\quad \Big\{(\alpha_1 \xrightarrow{\delta_1} \mathtt{nat}) \leqslant (\alpha_3 \xrightarrow{\delta_3} \mathtt{nat}),\ \{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \delta_1\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\delta_3} \mathtt{nat})\Big\};\quad \Big\{\Big\};\quad \Big\{\alpha_3 \leqslant \alpha_1,\ (\mathtt{nat}\ !\ \delta_1) \leqslant (\mathtt{nat}\ !\ \delta_3),\ \{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \delta_1\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\delta_3} \mathtt{nat})\Big\};\quad \Big\{\alpha_3 \leqslant \alpha_1\Big\};\quad \Big\{(\mathtt{nat}\ !\ \delta_1) \leqslant (\mathtt{nat}\ !\ \delta_3),\ \{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \delta_1\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\delta_3} \mathtt{nat})\Big\};\quad \Big\{\alpha_3 \leqslant \alpha_1\Big\};\quad \Big\{\mathtt{nat} \leqslant \mathtt{nat},\ \delta_1 \leqslant \delta_3,\ \{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \delta_1\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\delta_3} \mathtt{nat})\Big\};\quad \Big\{\alpha_3 \leqslant \alpha_1\Big\};\quad \Big\{\delta_1 \leqslant \delta_3,\ \{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \delta_1\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\delta_3} \mathtt{nat})\Big\};\quad \Big\{\alpha_3 \leqslant \alpha_1,\ \delta_1 \leqslant \delta_3\Big\};\quad \Big\{\{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \delta_1\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\mathtt{op}:\dot\rho_3\mid\delta_5} \mathtt{nat}),\ \delta_1 \mapsto \{\mathtt{op} : \dot\rho_2 \mid \delta_4\},\ \delta_3 \mapsto \{\mathtt{op} : \dot\rho_3 \mid \delta_5\}\Big\}; \\
&\qquad \Big\{\alpha_3 \leqslant \alpha_1\Big\};\quad \Big\{\{\mathtt{op} : \dot\rho_1 \mid \delta_2\} \leqslant \{\mathtt{op} : \dot\rho_2 \mid \delta_4\},\ \{\mathtt{op} : \dot\rho_2 \mid \delta_4\} \leqslant \{\mathtt{op} : \dot\rho_3 \mid \delta_5\}\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\mathtt{op}:\dot\rho_3\mid\delta_5} \mathtt{nat}),\ \delta_1 \mapsto \{\mathtt{op} : \dot\rho_2 \mid \delta_4\},\ \delta_3 \mapsto \{\mathtt{op} : \dot\rho_3 \mid \delta_5\}\Big\}; \\
&\qquad \Big\{\alpha_3 \leqslant \alpha_1,\ \dot\rho_1 \leqslant \dot\rho_2,\ \delta_2 \leqslant \delta_4\Big\};\quad \Big\{\{\mathtt{op} : \dot\rho_2 \mid \delta_4\} \leqslant \{\mathtt{op} : \dot\rho_3 \mid \delta_5\}\Big\}\Big) = \\
&\mathtt{unify}\Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\mathtt{op}:\dot\rho_3\mid\delta_5} \mathtt{nat}),\ \delta_1 \mapsto \{\mathtt{op} : \dot\rho_2 \mid \delta_4\},\ \delta_3 \mapsto \{\mathtt{op} : \dot\rho_3 \mid \delta_5\}\Big\}; \\
&\qquad \Big\{\alpha_3 \leqslant \alpha_1,\ \dot\rho_1 \leqslant \dot\rho_2 \leqslant \dot\rho_3,\ \delta_2 \leqslant \delta_4 \leqslant \delta_5\Big\};\quad \Big\{\Big\}\Big) = \\
&\qquad \Big(\Big\{\alpha_2 \mapsto (\alpha_3 \xrightarrow{\mathtt{op}:\dot\rho_3\mid\delta_5} \mathtt{nat}),\ \delta_1 \mapsto \{\mathtt{op} : \dot\rho_2 \mid \delta_4\},\ \delta_3 \mapsto \{\mathtt{op} : \dot\rho_3 \mid \delta_5\}\Big\}, \\
&\qquad \Big\{\alpha_3 \leqslant \alpha_1,\ \dot\rho_1 \leqslant \dot\rho_2 \leqslant \dot\rho_3,\ \delta_2 \leqslant \delta_4 \leqslant \delta_5\Big\}\Big)
\end{aligned}
$$

Figure 8: The unification of the set of constraints $\{(\alpha_1 \xrightarrow{\delta_1} \mathtt{nat}) \leqslant \alpha_2, \{\mathtt{op} : \dot\rho_2 \mid \delta_2\} \leqslant \delta_1\}$. We display constraints in a coalesced form. For example, we write $\dot\rho_1 \leqslant \dot\rho_2 \leqslant \dot\rho_3$ instead of $\dot\rho_1 \leqslant \dot\rho_2$, $\dot\rho_2 \leqslant \dot\rho_3$ and $\dot\rho_1 \leqslant \dot\rho_3$.

Proposition 4.4. *For any set of constraints $\mathcal{C}$, the algorithm* $\mathtt{unify}(\emptyset;\emptyset;\mathcal{C})$ *always halts.*

- *If it halts with a failure, then $\mathcal{C}$ has no solutions.*
- *If it halts returning $(\sigma', \mathcal{C}')$, then $\mathcal{C}$ has a solution. Furthermore, solutions $\sigma \models \mathcal{C}$ are exactly all the ones of the form $\sigma = \sigma'' \circ \sigma'$, where $\sigma'' \models \mathcal{C}'$.*

Note that `unify` can fail for two reasons: either we detect a cyclic constraint during occur check, or we try to unify two incompatible types — no failure can happen due to unification

of dirt and region constraints. These are exactly the cases in which the Hindley-Milner algorithm fails [20, p. 327], so our algorithm indeed infers the usual ML types, except annotated with information about effects.

Corollary 4.5. *The two-way inference rules*

$$\textsc{Unify-Expr}\ \frac{\Gamma \vdash_F e : A \mid \mathcal{C}}{\sigma(\Gamma) \vdash_F e : \sigma(A) \mid \mathcal{C}'} \qquad \textsc{Unify-Comp}\ \frac{\Gamma \vdash_F c : \underline{C} \mid \mathcal{C}}{\sigma(\Gamma) \vdash_F c : \sigma(\underline{C}) \mid \mathcal{C}'}$$

where $\texttt{unify}(\mathcal{C}) = (\sigma, \mathcal{C}')$*, are sound.*

Since Unify-Expr and Unify-Comp are two-way rules, we can be sure that no information is lost, so we can perform unification phase not only after, but also while gathering the constraints. In fact, in *Eff*, constraints are always kept in unified form, so unification is performed each time we add a new constraint. Though this strategy seems expensive, it offers many advantages:

(1) Any ill-typed terms are caught as soon as possible. This does not influence the efficiency too much, but gives much more informative error messages.
(2) In each rule, the inferred constraints consist mostly of constraints inherited from subterms. In *Eff*, the inference algorithm uses a technique called *λ-lifting* [24] to ensure that subderivations share no common parameters. Then, if constraints $\mathcal{C}_1$ and $\mathcal{C}_2$, say, are unified, so is their union $\mathcal{C}_1 \cup \mathcal{C}_2$, and we need to perform closure only for the small number of additional constraints, specific to the current rule.
(3) As we shall soon see, unified constraints may be garbage collected, drastically reducing their size and speeding up the algorithm.

5. Simplifying constraints

5.1. **Garbage collection.** The main simplification technique we employ is *garbage collection* [25, 28, 32]. We first recap the existing idea for type parameters, and then extend it to dirt and region parameters so to fit into our setting.

Recall that a parametric type A together with a unified set of constraints $\mathcal{C}$ captures exactly the closed types in the set $[\![A \mid \mathcal{C}]\!]$. Can we obtain a smaller set of constraints $\mathcal{C}'$ but keep $[\![A \mid \mathcal{C}']\!] = [\![A \mid \mathcal{C}]\!]$?

Mark type parameters in A as *positive* or *negative*, if they appear in a covariant or contravariant position, respectively. For example, in $(\alpha_1 \to \alpha_2) \to \alpha_3$, the parameters α_1 and α_3 are positive while α_2 is negative. It turns out that in $\mathcal{C}'$, we only need to keep the constraints of the form $\alpha^- \leqslant \alpha^+$, where α^- is a negative and α^+ is a positive.

Since $\mathcal{C}'$ contains less constraints than $\mathcal{C}$, any solution of $\mathcal{C}'$ is a solution of $\mathcal{C}$ as well, so we get $[\![A \mid \mathcal{C}]\!] \subseteq [\![A \mid \mathcal{C}']\!]$. For the other direction, we need to show that for any solution $\sigma' \models \mathcal{C}'$, there exists some $\sigma \models \mathcal{C}$ such that $\sigma(\alpha^+) \leqslant \sigma'(\alpha^+)$ holds for all positive α^+, and $\sigma'(\alpha^-) \leqslant \sigma(\alpha^-)$ holds for all negative α^-. It is then easy to see that $\sigma(A) \leqslant \sigma'(A)$ holds as well, and so any type A' such that $\sigma'(A) \leqslant A'$ already appears in $[\![A \mid \mathcal{C}]\!]$. For exact details, see the proof of Proposition 5.3.

Definition 5.1. The sets $\mathtt{pos}(A)$ of positive and $\mathtt{neg}(A)$ negative parameters in a given type A are defined as:

$$\begin{aligned} \mathtt{pos}(\alpha) &= \{\alpha\} & \mathtt{neg}(\alpha) &= \emptyset \\ \mathtt{pos}(A \to \underline{C}) &= \mathtt{neg}(A) \cup \mathtt{pos}(\underline{C}) & \mathtt{neg}(A \to \underline{C}) &= \mathtt{pos}(A) \cup \mathtt{neg}(\underline{C}) \\ \mathtt{pos}(E^{\rho}) &= \{\rho\} & \mathtt{neg}(E^{\rho}) &= \emptyset \\ \mathtt{pos}(\underline{C} \Rightarrow \underline{D}) &= \mathtt{neg}(\underline{C}) \cup \mathtt{pos}(\underline{D}) & \mathtt{neg}(\underline{C} \Rightarrow \underline{D}) &= \mathtt{pos}(\underline{C}) \cup \mathtt{neg}(\underline{D}) \end{aligned}$$

The sets of both positive and negative parameters in ground types ($\mathtt{bool}$, $\mathtt{nat}$, ...) are empty. For dirty types, the sets of positive and negative parameters are defined as:

$$\begin{aligned} \mathtt{pos}(A \,!\, \{\mathtt{op}_1 : \rho_1, \ldots, \mathtt{op}_n : \rho_n \mid \delta\}) &= \mathtt{pos}(A) \cup \{\rho_1, \ldots, \rho_n, \delta\} \\ \mathtt{neg}(A \,!\, \Delta) &= \mathtt{neg}(A) \end{aligned}$$

Definition 5.2. For a unified set of constraints $\mathcal{C}$ and sets of parameters P and N, we define the *garbage collected* constraints $\mathtt{gc}_{P,N}(\mathcal{C})$ as:

$$\begin{aligned} \mathtt{gc}_{P,N}(\mathcal{C}) = \{&(\alpha^- \leqslant \alpha^+) \in \mathcal{C} \mid \alpha^- \in N, \alpha^+ \in P\} \cup \\ &\{(\rho^- \leqslant \rho^+ \cup \textstyle\bigcup\!\!\!\bullet_i \, \dot\rho_i) \in \mathcal{C} \mid \rho^- \in N, \rho^+ \in P\} \cup \\ &\{(\mathtt{ins} \in \rho^+ \cup \textstyle\bigcup\!\!\!\bullet_i \, \dot\rho_i) \in \mathcal{C} \mid \rho^+ \in P\} \cup \\ &\{(\delta^- \leqslant \delta^+) \in \mathcal{C} \mid \delta^- \in N, \delta^+ \in P\} \end{aligned}$$

On $\mathtt{gc}_{P,N}(\mathcal{C})$, we define $\approx_{\mathtt{gc}_{P,N}(\mathcal{C})}$ to be the restriction of $\approx_{\mathcal{C}}$ to the set $P \cup N$.

To perform garbage collection on constraints in a typing judgement $\Gamma \vdash e : A \mid \mathcal{C}$, we define P to contain not just $\mathtt{pos}(A)$, but also $\mathtt{neg}(A_i)$ for all $(x_i : A_i) \in \Gamma$. Conversely, N must contain $\mathtt{neg}(A)$ and all $\mathtt{pos}(A_i)$. We similarly extend P with all region parameters $\dot\rho_i$ in constraints of the form $\rho \leqslant \rho' \cup \bigcup\!\!\!\bullet_i \, \dot\rho_i$ and $\mathtt{ins} \in \rho \cup \bigcup\!\!\!\bullet_i \, \dot\rho_i$. Otherwise, garbage collection may drop some of their lower bounds and thus relax the constraints. Recall that regions $\sigma(\dot\rho_i)$ are always non-empty, so decreasing them can only increase $\bigcup\!\!\!\bullet_i \, \dot\rho_i$.

Proposition 5.3. *If the set of constraints $\mathcal{C}$ is unified, the two-way inference rules*

GC-Expr
$$\frac{\Gamma \vdash_F e : A \mid \mathcal{C}}{\Gamma \vdash_{F \cap (P \cup N)} e : A \mid \mathtt{gc}_{P,N}(\mathcal{C})}$$

GC-Comp
$$\frac{\Gamma \vdash_F c : \underline{C} \mid \mathcal{C}}{\Gamma \vdash_{F \cap (P \cup N)} c : \underline{C} \mid \mathtt{gc}_{P,N}(\mathcal{C})}$$

where

$$\begin{aligned} P_0 &= \mathtt{pos}(A) \cup \mathtt{neg}(\Gamma) \\ P &= P_0 \cup \{\dot\rho_i \mid (\rho^- \leqslant \rho^+ \cup \textstyle\bigcup\!\!\!\bullet_i \, \dot\rho_i) \in \mathcal{C}, \rho^- \in N, \rho^+ \in P_0\} \cup \\ &\qquad \{\dot\rho_i \mid (\mathtt{ins} \in \rho^+ \cup \textstyle\bigcup\!\!\!\bullet_i \, \dot\rho_i) \in \mathcal{C}, \rho^+ \in P_0\} \\ N &= \mathtt{neg}(A) \cup \mathtt{pos}(\Gamma) \end{aligned}$$

are sound.

Furthermore, the set $\mathtt{gc}_{P,N}(\mathcal{C})$ is unified.

We present garbage collection in the form of two-way rules so that, like unification, we can perform it while still gathering constraints. This allows us to dispose any intermediate constraints as soon as possible, making the whole algorithm very efficient.

Ignoring handled region parameters in region constraints, the number of parameters is linear in the size of type, and each constraint relates two parameters, so the number of constraints is roughly quadratic in the size of the inferred type after garbage collection. In practice, though, the number of constraints is often much smaller and for typical functional programs, the current implementation of the inference algorithm in *Eff* is on a par with the one in OCaml (more details can be found in Table 1). Without garbage collection, the inference algorithm experiences exponential blow-up and is unusable.

filename	*Eff*	OCaml
`garsia_wachs.eff/.ml`	16 ms	16 ms
`list.eff/.ml`	46 ms	42 ms
`map.eff/.ml`	93 ms	60 ms
`set.eff/.ml`	53 ms	33 ms

Table 1: The table contains average ($N = 50, \sigma < 15\%$) file loading times for *Eff* 3.1 and OCaml 4.01.0 on Mac OS X 10.9.3 running on a 1.7 GHz Intel Core i5 with 4 GB of RAM. Except for a few initial definitions needed to bridge the differences, the *Eff* and OCaml files are identical (they can be found in the `examples/benchmarks/` folder of the *Eff* distribution). The test files contain only definitions and no executable code, so most of the loading time is spent on type-checking. To compensate for the time needed to start the interpreter and load the standard library, we subtracted the average time spent to load a blank file from all the entries (23 ms for *Eff* and 12 ms for OCaml).

5.2. **Simplifying region constraints.** Garbage collection is extremely efficient, but there is a small number of additional tactics we can offer that simplify region constraints. These are not meant to make the algorithm fast, but to make its output simpler. For as we shall see in Section 6.3, region constraints tend to be the most difficult to present succinctly, so it is crucial that they are as simple as possible before we display them. The main idea behind all optimizations is that in region constraints, handled region parameters $\dot\rho_i$ contribute to the right-hand side $\rho \cup \bigcup_i \dot\rho_i$ only when they denote a singleton.

When determining if an inhabited region parameter $\dot\rho$ denotes a singleton, it is helpful to see that $\dot\rho$ may appear as a covering region only in constraints of the form $\rho \leqslant \dot\rho$ and $\mathtt{ins} \in \dot\rho$, so ones with no handled regions $\bigcup_i \dot\rho_i$. To see this, one needs to laboriously check that no other constraints are added during inference, unification, or garbage collection. Adding an extra region parameter to Cstr-Op was a part of that effort.

Unlike garbage collection, each tactic is very basic, so we only sketch how they work.

- Any parameter $\dot\rho$ for which we have both $\mathtt{ins} \in \dot\rho$ and $\mathtt{ins}' \in \dot\rho$ for some $\mathtt{ins} \neq \mathtt{ins}'$ cannot denote a singleton. Thus, it may be removed from all singleton unions $\bigcup_i \dot\rho_i$ in which it appears. Similarly, if we have $\mathtt{ins} \in \dot\rho$, we may remove any occurrence of $\dot\rho$ as a handled region in constraints of the form $\mathtt{ins}' \in \rho \cup \bigcup_i \dot\rho_i$ as it cannot contain $\mathtt{ins}'$ and be a singleton at the same time.
- Next, take inhabited region parameters $\dot\rho_1 \leqslant \dot\rho_2$ (recall that all inequalities between inhabited region parameters are of that form) that both appear in some $\bigcup_i \dot\rho_i$. If $\dot\rho_1$ does

not denote a singleton, neither does $\dot{\rho}_2$, and if $\dot{\rho}_1$ does denote a singleton, $\dot{\rho}_2$ can make no further contribution to $\bigcup\!\!\!\bullet_i \dot{\rho}_i$. So, we may safely remove $\dot{\rho}_2$ in both cases.
- We may reduce the number of constraints by observing that if $I \subseteq J$, the constraint $\rho \leqslant \rho' \cup \bigcup\!\!\!\bullet_{i \in I} \dot{\rho}_i$ implies $\rho \leqslant \rho' \cup \bigcup\!\!\!\bullet_{i \in J} \dot{\rho}_i$ and we may safely throw the latter one away.
- After all the simplifications, we may further reduce the number of constraints by another round of garbage collection, in which we remove all lower bounds on region parameters that no longer occur in singleton unions.

5.3. **Further simplifications.** Our algorithm also applies to the much simpler case when one triggers effects with only operation symbols and no instances [11]. As this amounts to having a single instance $\star$ of each effect, we can drop all instance constraints $\texttt{ins} \in \rho \cup \bigcup\!\!\!\bullet_i \dot{\rho}_i$ with at least one handled region parameter $\dot{\rho}_i$. This is because `ins` must be $\star$ and all inhabited region parameters $\dot{\rho}$ must denote the singleton region $\{\star\}$, so any such constraint is always satisfied.

Another tempting way to drop constraints is to drop all constraints $\texttt{ins} \in \rho \cup \bigcup\!\!\!\bullet_i \dot{\rho}_i$ for which $\texttt{ins} \in \dot{\rho}_i$ is the only lower bound for some $\dot{\rho}_i$. But, unlike other simplification techniques, this may not be done until after we have gathered all the constraints, because $\dot{\rho}_i$ may still receive further lower bounds.

Similarly, we may want to improve the second tactic from Section 5.2, and remove any handled parameters $\dot{\rho}_2$ not only if we have $\dot{\rho}_1 \leqslant \dot{\rho}_2$, but also in case all lower bounds of $\dot{\rho}_1$ are also lower bounds of $\dot{\rho}_2$. If $\dot{\rho}_1 \leqslant \dot{\rho}_2$, this is implied by closure properties. For example, if we have lower bounds $\dot{\rho}_3 \leqslant \dot{\rho}_1$, $\dot{\rho}_3 \leqslant \dot{\rho}_2$ and $\dot{\rho}_4 \leqslant \dot{\rho}_2$, we can always assign a smaller region to $\dot{\rho}_1$ than to $\dot{\rho}_2$. Thus, if $\dot{\rho}_1$ and $\dot{\rho}_2$ appear in the same singleton union, we can safely drop $\dot{\rho}_2$, for if it denotes a singleton, so does $\dot{\rho}_1$. This condition is also stable under garbage collection — if $\dot{\rho}_1$ is not negative, the constraint $\dot{\rho}_1 \leqslant \dot{\rho}_2$ will be dropped, but since $\dot{\rho}_1$ and $\dot{\rho}_2$ are both positive, all their lower bounds will be kept. However, $\dot{\rho}_1$ may similarly receive further lower bounds, so we can perform the simplification only at the very end.

6. Displaying inferred types

However, once the algorithm finishes, we may perform further simplifications that lose information but make the output much easier to understand. These simplifications can be regarded as configurable, so one may, if desired, omit some of them to reveal more details. We emphasize that the presented techniques incur information loss and should be used only for displaying output to a programmer. For example, in *Eff*'s interactive loop, when we define a value, *Eff* shows the simplified type, but stores the full type in its typing environment.

6.1. **Displaying dirt parameters.** First, we can get rid of *all* inequalities between dirt parameters. Recall that increasing positive parameters always yields a valid typing, so we are interested only in their smallest possible value. Since this is exactly the union of all lower bounds, we display a positive dirt parameter δ^+ as the union $\bigcup_{\delta^- \leqslant \delta^+} \delta^-$ of all smaller negative parameters. So, given the constraints

$$\delta_1^- \leqslant \delta_2^+, \delta_1^- \leqslant \delta_3^+, \delta_4^- \leqslant \delta_2^+$$

we can replace δ_2^+ with its lower bound $\delta_1^- \cup \delta_4^-$ and δ_3^+ with its lower bound δ_1^-. In particular, if a positive dirt parameter δ has no lower bounds, we write $A \,!\, \delta$ as $A \,!\, \emptyset$, and $A \xrightarrow{\delta} B$ as $A \to B$. We keep the negative parameters as they are, because each usually captures a dirt of some input argument.

Since we can recover the constraints back from the displayed unions, this technique loses no information, though due to complex dirts, the displayed types are no longer in a form we can use for inference.

6.2. **Displaying type parameters.** Next, we could use the same approach for representing type parameters. However, though inequalities between type parameters are crucial for inference, they do not offer much additional information to the programmer besides telling what type parameters are of the same shape. So, we can get rid of *all* inequalities between type parameters, and label all type parameters in the same skeleton with the same symbol. So, instead of

$$(\alpha_1 \xrightarrow{\delta} \alpha_2) \to \alpha_3 \xrightarrow{\delta} \alpha_4 \mid \alpha_3 \leqslant \alpha_1, \alpha_2 \leqslant \alpha_4$$

which is the inferred type of application $\texttt{fun}\ f \mapsto \texttt{val}\ (\texttt{fun}\ x \mapsto f\ x)$, we can write

$$(\alpha \xrightarrow{\delta} \beta) \to \alpha \xrightarrow{\delta} \beta$$

This simplification does incur some information loss — see example at the beginning of Section 4.

6.3. **Displaying region parameters.** Finally, we can get rid of region inequalities in the same way as dirt inequalities, if we replace positive region parameters with their lower bounds. The problem is that these lower bounds are more complex due to handled regions. For inhabited parameters $\dot\rho$, which have simpler constraints, we can use the same technique, except that we also need to collect instance lower bounds. For example, we may write a parameter $\dot\rho$ with lower bounds

$$\texttt{ins}_1 \in \dot\rho, \texttt{ins}_2 \in \dot\rho, \rho_1 \leqslant \dot\rho, \rho_2 \leqslant \dot\rho$$

as $\{\texttt{ins}_1, \texttt{ins}_2\} \cup \rho_1 \cup \rho_2$.

For other region parameters, we first employ the techniques suggested in Section 5.3. Then, we display each constraint $\rho^- \leqslant \rho^+ \cup (\dot\rho_1 \cup \cdots \cup \dot\rho_n)$ by adding $\rho^- \dot- \dot\rho_1 \dot- \cdots \dot- \dot\rho_n$ to the lower bounds of ρ^+. We similarly add lower bounds of the form $\{\texttt{ins}\} - \cdots$ for constraints involving instances. We write $\dot-$ to emphasise the fact that we remove a region only if it is a singleton. If it turns out that any $\dot\rho$ has a single lower bound $\texttt{ins} \in \dot\rho$, we write $-\texttt{ins}$ instead of $\dot-\dot\rho$.

If $\rho \dot- \cdots$ appears in multiple lower bounds but with different handled regions, we display only the regions that appear in all constraints. Thus, instead of

$$(\rho \dot- \dot\rho_1 \dot- \dot\rho_2 \dot- \dot\rho_3) \cup (\rho \dot- \dot\rho_2 \dot- \dot\rho_3 \dot- \dot\rho_4) \cup (\rho \dot- \dot\rho_2 \dot- \dot\rho_3 \dot- \dot\rho_5)$$

we write just $\rho \dot- \dot\rho_2 \dot- \dot\rho_3$.

If there are multiple constraints with the same handled regions, which happens when we use the same handler more than once, we may merge them. For example, we may write $\{\texttt{ins}\} \cup (\{\texttt{ins}'\} \cup \rho) \dot- \dot\rho_1 \dot- \dot\rho_2$. instead of $\{\texttt{ins}\} \cup (\{\texttt{ins}'\} \dot- \dot\rho_1 \dot- \dot\rho_2) \cup (\rho \dot- \dot\rho_1 \dot- \dot\rho_2)$

Still, this may be too much information in some cases, so we can also display ρ' as $\{\texttt{ins}\} \cup (\{\texttt{ins}'\} \cup \rho)$? where the question mark lets the programmer know that some of

instances in $\{\mathtt{ins}'\} \cup \rho$ may be handled and that this lower bound may be decreased if further information is available. Finally, we can safely omit the question mark and over-approximate the lower bound with $\{\mathtt{ins}, \mathtt{ins}'\} \cup \rho$.

Determining the exact level of detail to show is subjective, and further techniques may arise when the effect system is used in practice.

6.4. **Displaying handler types.** As mentioned in Section 1.1.3, handler types often have a repetitive form that we can write in a more compact way. For example, a handler type

$$\alpha \,!\, \{\mathtt{lookup} : \rho_1, \mathtt{raise} : \rho_2 \mid \delta\} \Rightarrow \beta \,!\, \{\mathtt{lookup} : \rho_1 \div \rho, \mathtt{raise} : (\rho_2 \div \rho') \cup \rho'' \mid \delta\}$$

which describes a handler that handles memory lookups on a region ρ, handles exceptions from ρ' and raises exceptions from ρ'', can be written simply as

$$\alpha \xrightarrow{\mathtt{lookup}:\div\rho,\ \mathtt{raise}:\div\rho',+\rho''} \beta$$

7. Examples

7.1. **Function composition.** Before we turn to handlers, we display a typical run of the algorithm on an example with no special algebraic features: function composition, defined as

$$\mathtt{compose} \stackrel{\text{def}}{=} \mathtt{fun}\ f \mapsto \mathtt{val}\ (\mathtt{fun}\ g \mapsto \mathtt{val}\ (\mathtt{fun}\ x \mapsto \mathtt{let}\ y = f\ x\ \mathtt{in}\ g\ y))$$

The constraints are computed by the following derivation

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \alpha_f \mid \emptyset \qquad \Gamma \vdash x : \alpha_x \mid \emptyset}{\Gamma \vdash f\ x : \alpha_1 \,!\, \delta_1 \mid \alpha_f \leqslant (\alpha_x \xrightarrow{\delta_1} \alpha_1)} \qquad \dfrac{\Gamma, y{:}\alpha_y \vdash g : \alpha_g \qquad \Gamma, y{:}\alpha_y \vdash y : \alpha_y}{\Gamma, y{:}\alpha_y \vdash g\ y : \alpha_2 \,!\, \delta_2 \mid \alpha_g \leqslant (\alpha_y \xrightarrow{\delta_2} \alpha_2)}}{\Gamma \vdash \mathtt{let}\ y = f\ x\ \mathtt{in}\ g\ y : \alpha_2 \,!\, \delta_3 \mid \mathcal{C}}}{\vdots}}{\emptyset \vdash \mathtt{compose} : \alpha_f \xrightarrow{\delta_5} \alpha_g \xrightarrow{\delta_4} \alpha_x \xrightarrow{\delta_3} \alpha_2 \mid \mathcal{C}}$$

where $\Gamma = f : \alpha_f, g : \alpha_g, x : \alpha_x$ and

$$\mathcal{C} = \{\alpha_f \leqslant (\alpha_x \xrightarrow{\delta_1} \alpha_1), \alpha_g \leqslant (\alpha_y \xrightarrow{\delta_2} \alpha_2), \alpha_1 \leqslant \alpha_y, \delta_1 \leqslant \delta_3, \delta_2 \leqslant \delta_3\}$$

Unifying the constraints, we see that α_f and α_g need to be replaced with fresh function types, and the result of $\mathtt{unify}(\mathcal{C})$ is the substitution

$$\sigma = \{\alpha_f \mapsto (\alpha_3 \xrightarrow{\delta_6} \alpha_4), \alpha_g \mapsto (\alpha_5 \xrightarrow{\delta_7} \alpha_6)\}$$

and the unified constraints, which are:

$$\mathcal{C}' = \{\alpha_x \leqslant \alpha_3, \alpha_4 \leqslant \alpha_1 \leqslant \alpha_y \leqslant \alpha_5, \alpha_6 \leqslant \alpha_2, \delta_6 \leqslant \delta_1 \leqslant \delta_3, \delta_7 \leqslant \delta_2 \leqslant \delta_3\}$$

Under σ, the inferred type is

$$\mathtt{compose} : (\alpha_3 \xrightarrow{\delta_6} \alpha_4) \xrightarrow{\delta_5} (\alpha_5 \xrightarrow{\delta_7} \alpha_6) \xrightarrow{\delta_4} (\alpha_x \xrightarrow{\delta_3} \alpha_2)$$

so the sets of positive and negative parameters are

$$N = \{\alpha_x, \alpha_4, \alpha_6, \delta_6, \delta_7\} \qquad \text{and} \qquad P = \{\alpha_2, \alpha_3, \alpha_5, \delta_3, \delta_4, \delta_5\}$$

After the garbage collection, the only constraints that remain are

$$\mathtt{gc}_{P,N}(\mathcal{C}') = \{\alpha_x \leqslant \alpha_3, \alpha_4 \leqslant \alpha_5, \alpha_6 \leqslant \alpha_2, \delta_6 \leqslant \delta_3, \delta_7 \leqslant \delta_3\}$$

Finally, we replace all positive dirt parameters with the union of their lower bounds, merge all type parameters in the same skeleton, introduce fresh and readable parameters, and obtain the final type

$$\mathtt{compose} : (\alpha \xrightarrow{\delta} \beta) \to (\beta \xrightarrow{\delta'} \gamma) \to (\alpha \xrightarrow{\delta \cup \delta'} \gamma)$$

7.2. **Counting printouts.** Next, let us define a handler that computes the number of calls to `print` on a given channel c:

$$\begin{aligned}\mathtt{count\_print} \stackrel{\text{def}}{=}\ & \mathtt{fun}\ c \mapsto \mathtt{val}\ (\mathtt{handler} \\ & \quad \mid \mathtt{val}\ x \mapsto \mathtt{val}\ 0 \\ & \quad \mid c\#\mathtt{print}\ y\ k \mapsto \mathtt{let}\ n = k\ ()\ \mathtt{in}\ \mathtt{val}\ (\mathtt{succ}\ n) \\ &)\end{aligned}$$

The computed type of `count_print` is $\alpha_c \xrightarrow{\delta_0} (\underline{C} \Rightarrow \underline{D})$, where the form of the dirty types

$$\underline{C} \stackrel{\text{def}}{=} \alpha_{\text{in}}\ !\ \{\mathtt{print} : \rho_{\text{in}} \mid \delta_{\text{in}}\} \qquad \text{and} \qquad \underline{D} \stackrel{\text{def}}{=} \alpha_{\text{out}}\ !\ \{\mathtt{print} : \rho_{\text{out}} \mid \delta_{\text{out}}\}$$

reflects that `print` is the only operation symbol, appearing in the handler, while the computed constraints (in the order described on page 19) are

$$\begin{aligned}\big\{&(\mathtt{unit} \to \underline{D}) \leqslant (\mathtt{unit} \to \alpha_2\ !\ \delta_2), \alpha_3 \leqslant \mathtt{nat}, \alpha_2 \leqslant \alpha_3, \delta_2 \leqslant \delta_3, \delta_4 \leqslant \delta_3, \\ &\mathtt{nat}\ !\ \delta_1 \leqslant \underline{D}, \mathtt{nat}\ !\ \delta_3 \leqslant \underline{D}, \alpha_c \leqslant \mathtt{channel}^{\dot\rho}, \rho_{\text{in}} \leqslant \rho_{\text{out}} \uplus \dot\rho, \delta_{\text{in}} \leqslant \delta_{\text{out}}\big\}\end{aligned}$$

After unification and garbage collection, we get that the type of `count_print` is

$$\mathtt{channel}^{\rho} \xrightarrow{\delta_0} (\alpha_{\text{in}}\ !\ \{\mathtt{print} : \rho_{\text{in}} \mid \delta_{\text{in}}\} \Rightarrow \mathtt{nat}\ !\ \{\mathtt{print} : \rho_{\text{out}} \mid \delta_{\text{out}}\})$$

under the constraints $\{\delta_{\text{in}} \leqslant \delta_{\text{out}}, \rho \leqslant \dot\rho, \rho_{\text{in}} \leqslant \rho_{\text{out}} \uplus \dot\rho\}$. If we rename the parameters and use notation introduced in Section 6, we may write the type as

$$\mathtt{channel}^{\rho_1} \to (\alpha\ !\ \{\mathtt{print} : \rho_2 \mid \delta\} \Rightarrow \mathtt{nat}\ !\ \{\mathtt{print} : \rho_2 \div \rho_1 \mid \delta\})$$

or even as $\mathtt{channel}^{\rho} \to (\alpha \xrightarrow{\mathtt{print}:\div\rho} \mathtt{nat})$.

Define c to be a simple imperative computation `std#print "Hello, world!"` with the inferred type `unit` $!\ \{\mathtt{print} : \rho \mid \delta\}$ under the constraint $\mathtt{std} \in \rho$. If we use `count_print` to count the number of `std#print` calls as

$$\mathtt{let}\ h = (\mathtt{count\_print}\ \mathtt{std})\ \mathtt{in}\ (\mathtt{with}\ h\ \mathtt{handle}\ c)$$

the inferred type of the handled computation is `nat` $!\ \{\mathtt{print} : \rho \mid \delta\}$ under the constraints $\{\mathtt{std} \in \dot\rho, \mathtt{std} \in \rho \uplus \dot\rho\}$. Since $\dot\rho$ will receive no further lower bounds, we may drop the second constraint as described in Section 5.3, thus both ρ and δ are completely unconstrained and we may write the type as `nat` $!\ \emptyset$.

Conclusion

Related work. Our inference algorithm borrows heavily from existing inference algorithms: type inference is based on one for structural subtyping of Simonet [28], region inference is based on one for non-structural subtyping by Pottier [24], and dirt inference is based on Remy's row-typing [26]. Furthermore, our algorithm would be unusable in practice without garbage collection [25, 28, 32] — other lossless techniques presented in Section 5.2 merely complement it. Our decision to store inferred types in one form but display them in the other is similar to one in [24] except that to simplify display, we discard not only invariants that were necessary for inference, but inferred information as well. One point where we differ from current approaches is the use of unification for solving constraints. Though unification is simple and familiar, repeated substitution makes it very inefficient, so in practice, it would be better to replace it with a constraint-based algorithm as described in [28].

The related effect systems can be roughly divided into three groups: effect systems with no handlers, effect systems for exception handlers, and effect systems for handlers of arbitrary algebraic effects. So far, there are no approaches that offer general handlers and are not based on algebraic effects.

Most of the ongoing research considers effect systems for languages without any handlers. One of the main aims in the early work was a detailed analysis of memory allocation [16, 29, 33]. Due to the lack of handlers, there is no interest in the exact locations being accessed, only in the parts of the program that share locations from the same memory region. Hence, ordinary unification together with a simple constraint resolution is enough to infer the information, though our approach yields the same results.

Recent research acknowledges the importance of a user-friendly output (or input in prescriptive systems): Scala [27], a popular functional and object-oriented language, has a prescriptive effect system where a programmer annotates functions with simple labels such as `@pure` or `@throws[IOException]`. Unfortunately, our effect system is descriptive, so we cannot compare the two at this point.

Koka [13], a recently developed functional language, is based on a descriptive effect system that represents inferred effects with a row of labels such as `exn`, `io` or `div`. It is interesting to note that the effect system of Koka used to be based on constraints similar to ours [30], but they were dropped in favour of rows as they were found to be too complex in practice. We hope that our decision to keep constraints in the background alleviates this problem.

We can simulate the row-based approach to effect inference by assigning a single instance to each effect, though this still gives us results with too precise dirt descriptions. In order to obtain an output similar to Koka, we need to merge all related dirt parameters into a single one, just as we do for type parameters.

One effect that we do not treat, but Koka does, is divergence. We can add a dummy operation `⋆#div` and suitably modify the rule Cstr-LetRec to track divergence, but to prevent `div` from appearing in almost every practical program, we need to augment the effect system with some form of termination analysis, just like Koka does. For a practical language like Koka, this is essential, though we leave it as future work in our development.

Effect systems for exceptions and their handlers [6, 14, 35] often ignore other effects, but provide much richer information about exception flow. For example, Pessaux & Leroy [14] provide a row-based effect inference algorithm for OCaml that uses control-flow analysis in order to provide information about the values that exceptions carry as arguments. This is

because handlers in OCaml may be written so that they handle only exceptions with particular arguments, for example only `Failure "tl"`, which is raised when the tail function `tl` is applied to an empty list. We believe that in *Eff*, declaring a new exception `emptyListTail` is a cleaner solution and one for which our approach already infers all important information.

Exception arguments aside, the inference algorithm of Pessaux & Leroy produces results similar to ours (for programs using only exceptions, that is). We already saw at the beginning of Section 1 that both algorithms infer the same types of polymorphic higher-order functions. To simulate the row-based approach to exception inference, we could utilize our row-based dirt and take an *operation symbol* `exc` with a single instance $\star$ for each exception `exc`. However, this prevents us from passing around exceptions as first-class values, so it is better to represent each exception with a separate instance and obtain the same information in an unrestricted setting.

Handlers of algebraic effects are a recent discovery and so far, there are only three effect systems beside ours that employ them. First, Frank [17] is a prototype dependently-typed language with handlers and a prescriptive effect system. Next, there is a library that provides a simpler form of handlers embedded as a domain specific language (DSL) inside a dependently-typed language Idris [5]. Finally, the closest to our approach is an effect system by Kammar, Lindley & Oury [11], with safety results similar to ours, and an implementation as a DSL inside Haskell. This implementation also uses the type class mechanism of Haskell to infer effects. The main contributions we bring to the group are first-class handlers and instances (other approaches use only operations to trigger effects), and a stand-alone inference algorithm proven to be complete.

Future work. Our inference algorithm is general, offers strong guarantees on the effectful behaviour, and presents the programmer with information that is easy to understand. There are, as always, many possible improvements.

In its current form, the presented inference algorithm infers the same types as the Hindley-Milner algorithm, except that it provides an additional description of effects. However, we can make the effect system *prescriptive* by adding constraints of the form $\rho \leqslant R$, which limit the allowed instances. Then, a programmer may ensure that a given function will not raise exceptions by simply ascribing it the type $\alpha \xrightarrow{\{\texttt{raise}:\emptyset|\delta\}} \beta$. Having such constraints allows us to lift the restrictions placed on the effect signature (discussed in Remark 3.4), but determining their satisfiability is quite involved. For example, $\rho \leqslant \emptyset$ may be satisfiable simply because there are no constraints that give a lower bound to ρ. However, a more comprehensive treatment must also consider the case when a ρ is empty because all of its instances have been removed by handlers, and this is much more difficult to determine.

Next, let-polymorphism is currently based on value restriction, so that only types of expressions are generalized. Since we already have an effect system in place, we could relax this restriction and generalize the types of all pure computations, but this again leads to the problem of determining empty regions as described in the above paragraph.

Furthermore, *Eff* allows dynamic creation of fresh instances [2]. Since instances are a crucial part of our type system, instance generation has to be represented at that level as well. One option is having dirty types of the form $\nu R.A \,!\, \Delta$, where R captures the set of created instances, bound by ν in $A \,!\, \Delta$. Then, the type inference is particularly tricky as, for example, the type of `fun f -> f (); f ()` should reflect that it creates twice as much instances as `f`. A combination with recursion is a separate problem, since programs

can then create a potentially infinite number of instances, though a wild card region $\top$ as discussed in Remark 3.4 may be used in this case.

Finally, before algebraic effects and handlers can be considered practical, we need an efficient way of evaluating programs that use them. This may be difficult to achieve in general because of the freedom that handlers allow when manipulating the continuations. At the very least, running any existing ML programs in the algebraic setting should induce a minimal overhead.

In this line, we could also pass any information inferred by the effect system to an optimizing compiler: pure computations may be postponed, and computations with disjoint effects may be exchanged or even ran in parallel [31]. This looks like a promising and valuable direction of research, especially because such optimizations have already been studied in the context of algebraic effects, though without handlers [12].

We hope that the presented work will help the ML community to recognize algebraic effects as a natural progression of their current type system, and that the Haskell community will acknowledge the additional flexibility that handlers have to offer.

Acknowledgements

I would like to thank Andrej Bauer, Chris Stone, Ohad Kammar and the anonymous referees for all their extremely helpful comments and support.

References

[1] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science — 5th International Conference, CALCO 2013, Warsaw, Poland, September 3–6, 2013. Proceedings*, volume 8089 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[2] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 2014.

[3] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer, 2000.

[4] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ML to java bytecodes. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27–29, 1998*, pages 129–140. ACM, 1998.

[5] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Morrisett and Uustalu [19], pages 133–144.

[6] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In Pascal Van Hentenryck, editor, *Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8–10, 1997, Proceedings*, volume 1302 of *Lecture Notes in Computer Science*, pages 114–126. Springer, 1997.

[7] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[8] Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.

[9] INRIA. OCaml. `http://www.ocaml.org/`.

[10] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[11] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Morrisett and Uustalu [19], pages 145–158.

[12] Ohad Kammar and Gordon D. Plotkin. Algebraic foundations for effect-dependent optimisations. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012*, pages 349–360. ACM, 2012.

[13] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP 2014, Grenoble, France, 12 April 2014*, pages 100–126, 2014.

[14] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[15] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

[16] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In Jeanne Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10–13, 1988*, pages 47–57. ACM Press, 1988.

[17] Conor McBride. Frank: An experimental programming language with typed algebraic effects. `https://hackage.haskell.org/package/Frank/`.

[18] Robin Milner. *The definition of standard ML: revised*. The MIT press, Cambridge, MA, USA, 1997.

[19] Greg Morrisett and Tarmo Uustalu, editors. *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. ACM, 2013.

[20] Benjamin C Pierce. *Types and programming languages*. The MIT press, Cambridge, MA, USA, 2002.

[21] Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 1–24. Springer, 2001.

[22] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[23] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings*, pages 80–94. Springer, 2009.

[24] François Pottier. Type inference in the presence of subtyping: from theory to practice. Technical Report RR-3483, INRIA, 1998.

[25] François Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170(2):153–183, 2001.

[26] Didier Rémy. *Type inference for records in a natural extension of ML*. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press, 1993.

[27] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In James Noble, editor, *ECOOP 2012 — Object-Oriented Programming — 26th European Conference, Beijing, China, June 11–16, 2012. Proceedings*, pages 258–282. Springer, 2012.

[28] Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In Atsushi Ohori, editor, *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27–29, 2003, Proceedings*, pages 283–302. Springer, 2003.

[29] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[30] Ross Tate and Daan Leijen. Convenient explicit effects using type inference with subeffects. Technical Report MSR-TR-2010-80, Microsoft Research, 2010.

[31] Andrew P. Tolmach. Optimizing ML using a hierarchy of monadic types. In Xavier Leroy and Atsushi Ohori, editors, *Types in Compilation, Second International Workshop, TIC '98, Kyoto, Japan, March 25–27, 1998, Proceedings*, pages 97–115. Springer, 1998.

[32] Valery Trifonov and Scott F. Smith. Subtyping constrained types. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24–26, 1996, Proceedings*, pages 349–365. Springer, 1996.

[33] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003.
[34] Keith Wansbrough and Simon L. Peyton Jones. Once upon a polymorphic type. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20–22, 1999*, pages 15–28. ACM, 1999.
[35] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in standard ML programs. *Theoretical Computer Science*, 277(1):185–217, 2002.

Appendix A. Proofs

Lemma A.1. *The following two rules are admissible:*

$$\textsc{Sub-Refl}\ \frac{}{A \leqslant A} \qquad \textsc{Sub-Trans}\ \frac{A \leqslant A' \qquad A' \leqslant A''}{A \leqslant A''}$$

Proof. The proof proceeds by an induction on the structure of types. □

The subsumption rules SubExpr and SubComp allow us to increase types of expressions and computations. Conversely, we may decrease the types in contexts.

Lemma A.2. *Take contexts Γ and Γ' that bind the same variables and assume that for all $(x_i : A_i) \in \Gamma$, we have $(x_i : A_i') \in \Gamma'$ for some $A_i \leqslant A_i'$. Then, the following rules are admissible:*

$$\textsc{SubCtxExpr}\ \frac{\Gamma' \vdash e : A}{\Gamma \vdash e : A} \qquad \textsc{SubCtxComp}\ \frac{\Gamma' \vdash c : \underline{C}}{\Gamma \vdash c : \underline{C}}$$

Proof. The proof proceeds by a routine induction on the derivation of the typing judgement. □

To relate monomorphic typing judgements of core *Eff* to polymorphic inference judgements, we first need to substitute away any variables from the polymorphic context Ξ. For a polymorphic context $\Xi = (x_j : \forall F_j.\ A_j \mid \mathcal{C}_j)_{j=1}^n$ and expressions $(e_j)_{j=1}^n$, we write $\Gamma \vdash (e_j)_j \models \Xi$ if for all $j = 1, \ldots, n$, we have $\Gamma; (x_i : \forall F_i.\ A_i \mid \mathcal{C}_i)_{i=1}^{j-1} \vdash_{F_j} e_j : A_j \mid \mathcal{C}_j$. In this case, we define the expression $e[e_j/x_j]_j \stackrel{\text{def}}{=} e[e_1/x_1][e_2/x_2] \cdots [e_n/x_n]$, and the computation $c[e_j/x_j]_j \stackrel{\text{def}}{=} c[e_1/x_1][e_2/x_2] \cdots [e_n/x_n]$. Note that because polymorphic definitions may build on one another, we need to use nested, and not simultaneous substitution.

Proposition A.3 (Soundness)**.**

- *Assume that $\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}$ holds and take any expressions $(e_j)_j$ such that $\Gamma \vdash (e_j)_j \models \Xi$ holds and we have $\sigma_j(\Gamma) \vdash e_j[e_i/x_j]_{i=1}^{j-1} : \sigma_j(A_j)$ for any $\sigma_j \models \mathcal{C}_j$. Then, for any solution $\sigma \models \mathcal{C}$, we also have $\sigma(\Gamma) \vdash e[e_j/x_j]_j : \sigma(A)$.*
- *Assume that $\Gamma; \Xi \vdash_F c : \underline{C} \mid \mathcal{C}$ holds and take any expressions $(e_j)_j$ such that $\Gamma \vdash (e_j)_j \models \Xi$ holds and we have $\sigma_j(\Gamma) \vdash e_j[e_i/x_j]_{i=1}^{j-1} : \sigma_j(A_j)$ for any $\sigma_j \models \mathcal{C}_j$. Then, for any solution $\sigma \models \mathcal{C}$, we also have $\sigma(\Gamma) \vdash c[e_j/x_j]_j : \sigma(\underline{C})$.*

Proof. We proceed by a mutual induction on the derivation of inference judgements. In all cases, we use the same labels as in the considered rule. Also, as we never need the set of fresh parameters F, we do not write it. Take any suitable $(e_j)_j$ and any σ that satisfies the

set of constraints in the conclusion, and consider the case when the last rule used in the derivation is:

Cstr-PolyVar**:** The case for a variable $x_i \in \Xi$ is immediate because we can use the assumption made for $e_i = x_i[e_j/x_j]_j$.

Cstr-Inst**:** Since $\sigma \models \mathtt{ins} \in \dot{\rho}$, we have $\mathtt{ins} \in \sigma(\dot{\rho})$. Because $\mathtt{ins}[e_j/x_j]_j = \mathtt{ins}$, we can use Inst to get $\sigma(\Gamma) \vdash \mathtt{ins}[e_j/x_j]_j : \sigma(E^{\dot{\rho}})$.

Cstr-Hand**:** Let $A \,!\, \Delta = \sigma(\underline{C})$ and $B \,!\, \Delta' = \sigma(\underline{D})$. By assumption, we have $\sigma \models \mathcal{C}_v$, so we get $\sigma(\Gamma), x : \sigma(\alpha_{\text{in}}) \vdash c_v[e_j/x_j]_j : \sigma(\underline{D}_v)$ by the induction hypothesis. Since $\sigma \models \underline{D}_v \leqslant \underline{D}$, we further get $\sigma(\Gamma), x : A \vdash c_v[e_j/x_j]_j : B \,!\, \Delta'$. Similarly, we get appropriate results for each Ψ_i.

Next, for each $\mathtt{op} \in \mathcal{O}$, let $R^{\mathtt{op}}_{\text{in}} = \sigma(\rho^{\mathtt{op}}_{\text{in}})$ and $R^{\mathtt{op}}_{\text{out}} = \sigma(\rho^{\mathtt{op}}_{\text{out}})$. Now, take any $\mathtt{ins\#op} \in \Delta$. Then, if $\mathtt{op} \in \mathcal{O}$, we have $\mathtt{ins} \in R^{\mathtt{op}}_{\text{in}}$. Since $\sigma \models \rho^{\mathtt{op}}_{\text{in}} \leqslant \rho^{\mathtt{op}}_{\text{out}} \cup \bigcup_{\mathtt{op}=\mathtt{op}_i} \dot{\rho}_i$, we either have $\mathtt{ins} \in R^{\mathtt{op}}_{\text{out}}$ hence $\mathtt{ins\#op} \in \Delta'$, or $\mathtt{ins} \in \bigcup_{\mathtt{op}=\mathtt{op}_i} R_i$. If $\mathtt{op} \notin \mathcal{O}$, then $\mathtt{ins\#op}$ must be in $\sigma(\delta_{\text{in}})$, thus also in $\sigma(\delta_{\text{out}}) \subseteq \Delta'$. We can then conclude by applying Hand.

Cstr-Op**:** As we have $\sigma \models \mathcal{C}_1$, $\sigma \models \mathcal{C}_2$, and $\sigma \models \mathcal{C}$, we get $\sigma(\Gamma) \vdash e_1[e_j/x_j]_j : \sigma(A_1)$, $\sigma(\Gamma) \vdash e_2[e_j/x_j]_j : \sigma(A_2)$, and $\sigma(\Gamma), y : B^{\mathtt{op}} \vdash c[e_j/x_j]_j : \sigma(A \,!\, \Delta)$ by the induction hypothesis.

Next, because $\sigma \models A_1 \leqslant E^{\dot{\rho}}$ and $\sigma \models A_2 \leqslant A^{\mathtt{op}}$, we can use SubExpr to get $\sigma(\Gamma) \vdash e_1[e_j/x_j]_j : E^{\sigma(\dot{\rho})}$ and $\sigma(\Gamma) \vdash e_2[e_j/x_j]_j : A^{\mathtt{op}}$. Then, $\sigma \models \Delta \leqslant \{\mathtt{op} : \rho \mid \delta\}$, and by SubComp, we get $\sigma(\Gamma), y : B^{\mathtt{op}} \vdash c[e_j/x_j]_j : \sigma(A \,!\, \{\mathtt{op} : \rho \mid \delta\})$.

Finally, since $\sigma(\dot{\rho}) \subseteq \sigma(\rho)$, we have $\mathtt{ins\#op} \in \sigma(\{\mathtt{op} : \rho \mid \delta\})$ for any $\mathtt{ins} \in \sigma(\dot{\rho})$, so we may use Op to conclude.

Cstr-LetVal**:** By the induction hypothesis for e, we have that the terms $e_1, \ldots, e_n, e$ satisfy the conditions of the induction hypothesis for c, thus $\sigma(\Gamma) \vdash (c[e/x])[e_j/x_j]_j : \sigma(\underline{C})$ holds, and so, we have $\sigma(\Gamma) \vdash (\mathtt{let\ val}\ x = e\ \mathtt{in}\ c)[e_j/x_j]_j : \sigma(\underline{C})$ by LetVal.

Cstr-With**:** As $\sigma \models \mathcal{C}_1$ and $\sigma \models \mathcal{C}_2$, we can use induction to get $\sigma(\Gamma) \vdash e[e_j/x_j]_j : \sigma(A)$ and $\sigma(\Gamma) \vdash c[e_j/x_j]_j : \sigma(\underline{C})$. Next, we have $\sigma \models A \leqslant (\underline{C} \Rightarrow \alpha \,!\, \delta)$, thus we may use SubExpr and get $\sigma(\Gamma) \vdash e[e_j/x_j]_j : \sigma(\underline{C}) \Rightarrow \sigma(\alpha \,!\, \delta)$. So by With, we get $\sigma(\Gamma) \vdash (\mathtt{with}\ e\ \mathtt{handle}\ c)[e_j/x_j]_j : \sigma(\alpha \,!\, \delta)$.

In all other cases, the proof proceeds routinely. □

Lemma A.4 (Weakening)**.**

- *If* $\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}$ *holds, so does* $\Gamma; \Xi, (x : \forall F'.\ A' \mid \mathcal{C}') \vdash_F e : A \mid \mathcal{C}$ *for any* x *that does not appear in* Γ, Ξ *or* e.
- *If* $\Gamma; \Xi \vdash_F c : \underline{C} \mid \mathcal{C}$ *holds, so does* $\Gamma; \Xi, (x : \forall F'.\ A' \mid \mathcal{C}') \vdash_F c : \underline{C} \mid \mathcal{C}$ *for any* x *that does not appear in* Γ, Ξ *or* c.

Proof. The proof proceeds by routine induction on the derivation of inference judgements. □

Lemma A.5 (Exchange)**.**

- *If* $\Gamma; \Xi, (x_1 : \forall F_1.\ A_1 \mid \mathcal{C}_1), (x_2 : \forall F_2.\ A_2 \mid \mathcal{C}_2) \vdash_F e : A \mid \mathcal{C}$ *holds, then so does* $\Gamma; \Xi, (x_2 : \forall F_2.\ A_2 \mid \mathcal{C}_2), (x_1 : \forall F_1.\ A_1 \mid \mathcal{C}_1) \vdash_F e : A \mid \mathcal{C}$.
- *If* $\Gamma; \Xi, (x_1 : \forall F_1.\ A_1 \mid \mathcal{C}_1), (x_2 : \forall F_2.\ A_2 \mid \mathcal{C}_2) \vdash_F c : \underline{C} \mid \mathcal{C}$ *holds, then so does* $\Gamma; \Xi, (x_2 : \forall F_2.\ A_2 \mid \mathcal{C}_2), (x_1 : \forall F_1.\ A_1 \mid \mathcal{C}_1) \vdash_F c : \underline{C} \mid \mathcal{C}$.

Proof. The proof proceeds by routine induction on the derivation of inference judgements. □

Lemma A.6.

- *For any e, we have that if $\Gamma; \Xi \vdash_F e[e'/x] : A \mid \mathcal{C}$ holds for some $\Gamma; \Xi \vdash_{F'} e' : A' \mid \mathcal{C}'$, so does $\Gamma; \Xi, (x : \forall F'.\ A' \mid \mathcal{C}') \vdash_F e : A \mid \mathcal{C}$.*
- *For any c, we have that if $\Gamma; \Xi \vdash_F c[e'/x] : \underline{C} \mid \mathcal{C}$ holds for some $\Gamma; \Xi \vdash_{F'} e' : A' \mid \mathcal{C}'$, so does $\Gamma; \Xi, (x : \forall F'.\ A' \mid \mathcal{C}') \vdash_F c : \underline{C} \mid \mathcal{C}$.*

Proof. We proceed by a mutual induction on the structure of the term:

- If e is some variable $y \neq x$, we have that $e[e'/x] = e$, hence $\Gamma; \Xi \vdash_F e : A \mid \mathcal{C}$. We conclude by using Lemma A.4.
- If e is the variable x, we also have that $\Gamma; \Xi \vdash_F e' : A \mid \mathcal{C}$ holds. By induction on the derivation of inference judgements, we can show that the inferred types and constraints are unique up to renaming. Thus $(\forall F'.\ A' \mid \mathcal{C}') = (\forall F.\ A \mid \mathcal{C})$ up to α-equivalence, so $\Gamma; \Xi, (x : \forall F'.\ A' \mid \mathcal{C}') \vdash_F x : A \mid \mathcal{C}$ holds.
- If e is $\texttt{fun}\ x' \mapsto c$, the only way of obtaining the inference judgement is by using Cstr-Fun to get some $\Gamma; \Xi \vdash_{F,\alpha} \texttt{fun}\ x \mapsto c[e'/x] : \alpha \to \underline{C} \mid \mathcal{C}$. Hence, we have that $\Gamma, x : \alpha; \Xi \vdash_F c[e'/x] : \underline{C} \mid \mathcal{C}$, so we get $\Gamma, x : \alpha; \Xi, (x : \forall F'.\ A' \mid \mathcal{C}') \vdash_F c : \underline{C} \mid \mathcal{C}$ by the induction hypothesis. Using Cstr-Fun we obtain the desired conclusion $\Gamma; \Xi, (x : \forall F'.\ A' \mid \mathcal{C}') \vdash_{F,\alpha} \texttt{fun}\ x \mapsto c : \alpha \to \underline{C} \mid \mathcal{C}$.
- If e is $\texttt{let val}\ x' = e''\ \texttt{in}\ c$, the only rule that applies is Cstr-LetVal. We proceed just as in the previous case by using the induction hypothesis and reapplying the rule Cstr-LetVal, except that we also need to use Lemma A.5 to obtain the proper ordering of variables in Ξ.

In all other cases that do not touch Ξ, the proof proceeds routinely just like for functions. □

Proposition A.7 (Completeness)**.**

- *For any polymorphic context Ξ, closed substitution σ, expressions $\sigma(\Gamma) \vdash (e_j)_j \models \Xi$ and any $\sigma(\Gamma) \vdash e[e_j/x_j]_j : A$, we have $\Xi; \Gamma \vdash_F e : A' \mid \mathcal{C}$ and there exists a solution $\sigma' \models \mathcal{C}$, which extends σ to F, such that $\sigma(A') \leqslant A$.*
- *For any polymorphic context Ξ, closed substitution σ, expressions $\sigma(\Gamma) \vdash (e_j)_j \models \Xi$ and any $\sigma(\Gamma) \vdash c[e_j/x_j]_j : \underline{C}$, we have $\Xi; \Gamma \vdash_F c : \underline{C'} \mid \mathcal{C}$ and there exists a solution $\sigma' \models \mathcal{C}$, which extends σ to F, such that $\sigma(\underline{C'}) \leqslant \underline{C}$.*

Proof. We again proceed by a mutual induction, this time on the derivation of typing judgements. In all cases, we again use the same labels as in the considered rule.

Note that no parameters from F can appear in Γ, and we may safely assume that σ is undefined on F. Furthermore, we may safely compose substitutions that are defined on disjoint sets of parameters. For example, by taking σ_1 and σ_2 that extend σ to disjoint sets F_1 and F_2, respectively, we may uniquely define $\sigma' = \sigma \cup \sigma_1 \cup \sigma_2$, which extends σ to $F_1 \cup F_2$.

Consider the case when the last rule used in the derivation is:

Inst**:** Assume that $\sigma(\Gamma) \vdash \texttt{ins} : E^R$ for some $\texttt{ins} \in R$. By Cstr-Inst, we first get $\Gamma \vdash_{\dot\rho} \texttt{ins} : E^{\dot\rho} \mid \texttt{ins} \in \dot\rho$. Next, we extend σ to $\dot\rho$ by defining $\sigma' = \sigma \cup \{\dot\rho \mapsto R\}$. By assumption, we have $\texttt{ins} \in R$, so $\sigma' \models \texttt{ins} \in \dot\rho$ by definition. Finally, we get $\sigma'(E^{\dot\rho}) \leqslant E^R$ by Sub-Refl.

Hand**:** From the value case and operation cases, we get the necessary premises Ψ_v and $(\Psi_i)_i$ of Cstr-Hand, and solutions σ_v, $(\sigma_i)_i$ and $(\sigma'_i)_i$ that satisfy the required properties. We

then define

$$\sigma = \sigma_v \cup \bigcup_i \sigma_i \cup \bigcup_i \sigma_i' \cup \Big\{ \alpha_{\text{in}} \mapsto A, \alpha_{\text{out}} \mapsto B, \delta_{\text{in}} \mapsto \{\texttt{ins\#op} \in \Delta \mid \texttt{op} \notin \mathcal{O}\}, \\ \delta_{\text{out}} \mapsto \{\texttt{ins\#op} \in \Delta' \mid \texttt{op} \notin \mathcal{O}\}, (\dot\rho_i \mapsto R_i)_i, \\ (\rho_{\text{in}}^{\texttt{op}} \mapsto \{\texttt{ins} \mid \texttt{ins\#op} \in \Delta\})_{\texttt{op} \in \mathcal{O}}, \\ (\rho_{\text{out}}^{\texttt{op}} \mapsto \{\texttt{ins} \mid \texttt{ins\#op} \in \Delta'\})_{\texttt{op} \in \mathcal{O}} \Big\}$$

We routinely check that σ satisfies all the necessary conditions and that we can apply Cstr-Hand to obtain the expected result.

SubExpr**:** By assumption, we have $\sigma(\Gamma) \vdash e[e_j/x_j]_j : A'$ for some $\sigma(\Gamma) \vdash e[e_j/x_j]_j : A$ and $A \leqslant A'$. By induction hypothesis, we have $\Gamma \vdash_F e : A'' \mid \mathcal{C}$ and $\sigma' \models \mathcal{C}$ that extends σ to F such that $\sigma'(A'') \leqslant A$. We may then use Sub-Trans to get $\sigma'(A'') \leqslant A'$.

Op**:** Assume $\sigma(\Gamma) \vdash (e_1 \texttt{\#op}\, e_2\, (y.\, c))[e_j/x_j]_j : A \mathbin{!} \Delta$ for some $\sigma(\Gamma) \vdash e_1[e_j/x_j]_j : E^R$, $\sigma(\Gamma) \vdash e_2[e_j/x_j]_j : A^{\texttt{op}}$, and $\sigma(\Gamma), y : B^{\texttt{op}} \vdash c[e_j/x_j]_j : A \mathbin{!} \Delta$. By induction hypothesis, we get $\Gamma \vdash_{F_1} e_1 : A_1 \mid \mathcal{C}_1$, $\Gamma \vdash_{F_2} e_2 : A_2 \mid \mathcal{C}_2$, and $\Gamma, y : B^{\texttt{op}} \vdash_F c : \underline{C} \mid \mathcal{C}$, together with σ_1, σ_2, and σ' that satisfy the required properties. Using Cstr-Op, we get

$$\Gamma \vdash_{F_1,F_2,F,\rho,\dot\rho,\delta} e_1 \texttt{\#op}\, e_2\, (y.\, c) : A \mathbin{!} \{\texttt{op} : \rho \mid \delta\} \\ \mid \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}, A_1 \leqslant E^{\dot\rho}, A_2 \leqslant A^{\texttt{op}}, \dot\rho \leqslant \rho, \Delta \leqslant \{\texttt{op} : \rho \mid \delta\}$$

We extend σ to all the fresh parameters by

$$\sigma'' = \sigma_1 \cup \sigma_2 \cup \sigma' \cup \{\rho \mapsto R, \dot\rho \mapsto R, \delta \mapsto \{\texttt{ins\#op}' \in \Delta \mid \texttt{op}' \neq \texttt{op}\}\}$$

We conclude the proof by observing that σ'' satisfies all the necessary conditions.

LetVal**:** As $\sigma(\Gamma) \vdash (\texttt{let val}\ x = e\ \texttt{in}\ c)[e_j/x_j]_j : \underline{C}$, we get some $\sigma(\Gamma) \vdash (c[e/x])[e_j/x_j]_j : \underline{C}$. By induction hypothesis, we get some $\Gamma; \Xi \vdash_{F_1} e : A \mid \mathcal{C}_1$ and $\Gamma; \Xi \vdash_{F_2} c[e/x] : \underline{C'} \mid \mathcal{C}_2$ together with $\sigma \models \mathcal{C}_2$ such that $\sigma(\underline{C'}) \leqslant \underline{C}$. Then, by Lemma A.6, we have that $\Gamma; \Xi, (x : \forall F_1.\ A \mid \mathcal{C}_1) \vdash_{F_2} c : \underline{C'} \mid \mathcal{C}_2$, so we may conclude by using Cstr-LetVal.

With**:** As $\sigma(\Gamma) \vdash (\texttt{with}\ e\ \texttt{handle}\ c)[e_j/x_j]_j : A \mathbin{!} \Delta$ we get $\sigma(\Gamma) \vdash e[e_j/x_j]_j : \underline{C} \Rightarrow A \mathbin{!} \Delta$ and $\sigma(\Gamma) \vdash c[e_j/x_j]_j : \underline{C}$. Next, by induction hypothesis, we get $\Gamma \vdash_{F_1} e : A' \mid \mathcal{C}_1$ together with $\sigma_1 \models \mathcal{C}_1$ that extends σ to F_1, such that $\sigma_1(A') \leqslant \underline{C} \Rightarrow A \mathbin{!} \Delta$. Similarly, we get $\Gamma \vdash_{F_2} c : \underline{C'} \mid \mathcal{C}_2$ together with $\sigma_2 \models \mathcal{C}_2$ that extends σ to F_2, such that $\sigma_2(\underline{C'}) \leqslant \underline{C}$.

By Cstr-With, we get

$$\Gamma \vdash_{F_1,F_2,\alpha,\delta} \texttt{with}\ e\ \texttt{handle}\ c : \alpha \mathbin{!} \delta \mid \mathcal{C}_1, \mathcal{C}_2, A' \leqslant (\underline{C'} \Rightarrow \alpha \mathbin{!} \delta)$$

We define

$$\sigma' = \sigma_1 \cup \sigma_2 \cup \{\alpha \mapsto A, \delta \mapsto \Delta\}$$

which extends σ to all the fresh parameters. Since σ' extends σ_1 and σ_2, we have $\sigma' \models \mathcal{C}_1$ and $\sigma' \models \mathcal{C}_2$. Furthermore,

$$\sigma'(A') = \sigma_1(A') \leqslant \underline{C} \Rightarrow A \mathbin{!} \Delta \leqslant \sigma_2(\underline{C'}) \Rightarrow A \mathbin{!} \Delta = \sigma'(\underline{C'} \Rightarrow \alpha \mathbin{!} \delta)$$

hence σ' satisfies all the necessary conditions. Finally, we have $\sigma'(\alpha \mathbin{!} \delta) \leqslant A \mathbin{!} \Delta$ by Sub-Refl.

SubComp**:** The proof proceeds in the same way as in the case of SubExpr.

In all other cases, the proof again proceeds routinely. □

Proof of Theorem 3.3. Take any $\Gamma \vdash e : A'$. We may then use Proposition A.7 for the empty context Ξ and the identical substitution to obtain $\Gamma \vdash e : A \mid \mathcal{C}$ such that $A' \in [\![A \mid \mathcal{C}]\!]$. Since A and $\mathcal{C}$ are uniquely determined up to a renaming, we may use the same reasoning for any A'' and get $\{A'' \mid (\Gamma \vdash e : A'')\} \subseteq [\![A \mid \mathcal{C}]\!]$. The converse follows from Proposition A.3 while the case for computations is exactly the same. □

Proof of Lemma 4.2. We can define a solution σ of a unified set of constraints $\mathcal{C}$ as follows. For all type parameters α and all dirt parameters δ, we define $\sigma(\alpha) = \texttt{unit}$ and $\sigma(\delta) = \emptyset$. Then, σ trivially satisfies all type and dirt constraints in $\mathcal{C}$. In a similar way, we also show that $\sigma(\alpha) \approx \sigma(\alpha')$ for any $\alpha \approx_{\mathcal{C}} \alpha'$. In fact, we get a solution whenever we replace all type parameters in the same skeleton with the same type.

For region parameters, we define $\sigma(\rho) = \{\texttt{ins} \mid (\texttt{ins} \in \rho \cup \bigcup\!\!\bullet_i \dot{\rho}_i) \in \mathcal{C}\}$. Then, σ satisfies all constraints $(\texttt{ins} \in \rho \cup \bigcup\!\!\bullet_i \dot{\rho}_i) \in \mathcal{C}$. Next, take a constraint $(\rho \leqslant \rho' \cup \bigcup\!\!\bullet_{i \in I} \dot{\rho}_i)$. Then $(\texttt{ins} \in \rho \cup \bigcup\!\!\bullet_{i \in J} \dot{\rho}_i) \in \mathcal{C}$ implies $(\texttt{ins} \leqslant \rho' \cup \bigcup\!\!\bullet_{i \in I \cup J} \dot{\rho}_i) \in \mathcal{C}$ because $\mathcal{C}$ is unified. Thus, if $\texttt{ins} \in \sigma(\rho)$ then $\texttt{ins} \in \sigma(\rho')$, and so $\sigma \models \rho \leqslant \rho' \cup \bigcup\!\!\bullet_i \dot{\rho}_i$.

There are, of course, countless other ways of choosing σ. □

Proof of Proposition 4.4. We see that `unify` always terminates because at each recursive call, we strictly decrease the degree, defined lexicographically according to:

(1) the number of skeletons of $\approx_{\mathcal{C}}$,
(2) the number of all type constructors in $\mathcal{Q}$,
(3) the total *potential* of all dirt parameters, where the potential of a dirt parameter $\delta^{\mathcal{O}}$ is the number of all operation symbols mentioned in $\mathcal{Q}$ that do not appear in $\mathcal{O}$,
(4) the number of constraints in $\mathcal{Q}$.

Because of occur check, (1) decreases each time we unify any constraint of the form $\alpha \leqslant A$ or $A \leqslant \alpha$. When decomposing constraints between types of the same form, we keep (1) as it is, but decrease (2). When we unify a constraint $\alpha \leqslant \alpha'$, (1) decreases if α was not in the same skeleton as α'. If it was, (1–3) remain the same, but (4) decreases. A similar situation occurs when unifying region constraints or dirt constraints with matching operations. If we unify dirt with non-matching annotations, we keep type constraints and thus (1–2) as they are, but ensure that (3) decreases.

To show that unification preserves solutions, we observe that at each recursive call $\texttt{unify}(\sigma_k; \mathcal{C}_k; \mathcal{Q}_k) = \texttt{unify}(\sigma_{k+1}; \mathcal{C}_{k+1}; \mathcal{Q}_{k+1})$, and for each $\sigma'_k \models \mathcal{C}_k \cup \mathcal{Q}_k$, there exists $\sigma'_{k+1} \models \mathcal{C}_{k+1} \cup \mathcal{Q}_{k+1}$ such that $\sigma'_k \circ \sigma_k = \sigma'_{k+1} \circ \sigma_{k+1}$. Similarly, we observe that if $\texttt{unify}(\sigma; \mathcal{C}; \mathcal{Q})$ results in a failure, then $\mathcal{C} \cup \mathcal{Q}$ does not have any solutions.

Since we start with $\sigma_0 = \mathrm{id}$, $\mathcal{C}_0 = \emptyset$ and $\mathcal{Q}_0 = \mathcal{C}$ and end with some $\sigma_n = \sigma'$, $\mathcal{C}_n = \mathcal{C}'$ and $\mathcal{Q}_n = \emptyset$, then for each $\sigma \models \mathcal{C}$, we have some $\sigma'' \models \mathcal{C}'$ such that $\sigma = \sigma'' \circ \sigma'$. □

Proof of Corollary 4.5. Let us show the soundness of rule Unify-Expr, because the proof for Unify-Comp is exactly the same. First, take any $\sigma' \models \mathcal{C}'$. By Proposition 4.4, we know that $\sigma' \circ \sigma \models \mathcal{C}$, so by the induction hypothesis, we get $(\sigma' \circ \sigma)(\Gamma) \vdash e : (\sigma' \circ \sigma)(A)$, which is exactly what we wanted to prove.

For the other direction, take any $\sigma'' \models \mathcal{C}$. Again by Proposition 4.4, there must exist some $\sigma' \models \mathcal{C}'$ such that $\sigma'' = \sigma' \circ \sigma$. We then get $\sigma'(\sigma(\Gamma)) \vdash e : \sigma'(\sigma(A))$ or, equivalently, $\sigma''(\Gamma) \vdash e : \sigma''(A)$ by the induction hypothesis. □

Proof of Proposition 5.3. Again, we shall prove only the soundness of rule GC-Expr, because the proof for GC-Comp is identical. The soundness of the reverse rule is trivial:

any solution $\sigma \models \mathcal{C}$ is also a solution of $\mathtt{gc}_{P,N}(\mathcal{C})$ so by induction hypothesis, we get $\sigma(\Gamma) \vdash e : \sigma(A)$.

For the forward direction, the reasoning follows the one sketched in the beginning of Section 5.1. Take any solution $\sigma' \models \mathtt{gc}_{P,N}(\mathcal{C})$. Following [28], we are going to construct a solution $\sigma \models \mathcal{C}$ such that $\sigma(\alpha^+) \leqslant \sigma'(\alpha^+)$ holds for all $\alpha^+ \in P$ and $\sigma'(\alpha^-) \leqslant \sigma(\alpha^-)$ holds for all $\alpha^- \in N$ (and similarly for region and dirt parameters). Then, by the induction hypothesis, we get $\sigma(\Gamma) \vdash e : \sigma(A)$. Since P contains $\mathtt{pos}(A)$ and N contains $\mathtt{neg}(A)$, we have $\sigma(A) \leqslant \sigma'(A)$, thus by SubExpr we get $\sigma(\Gamma) \vdash e : \sigma'(A)$. Conversely, for all $(x_i : A_i) \in \Gamma$, the set P contains $\mathtt{neg}(A_i)$ and N contains $\mathtt{pos}(A_i)$, so we get $\sigma'(A_i) \leqslant \sigma(A_i)$. We may thus use SubCtxExpr and get $\sigma'(\Gamma) \vdash e : \sigma'(A)$.

We construct $\sigma \models \mathcal{C}$ as follows. Let us start with the simplest case of dirt parameters. We define

$$\sigma(\delta) \stackrel{\text{def}}{=} \bigcup_{\substack{(\delta^- \leqslant \delta) \in \mathcal{C} \\ \delta^- \in N}} \sigma'(\delta^-)$$

First, for any $\delta^- \in N$, we implicitly have $(\delta^- \leqslant \delta^-) \in \mathcal{C}$, so $\sigma'(\delta^-) \subseteq \sigma(\delta^-)$. Next, for any $\delta^+ \in P$, if have $(\delta^- \leqslant \delta^+) \in \mathcal{C}$, we also have $\delta^- \leqslant \delta^+ \in \mathtt{gc}_{P,N}(\mathcal{C})$ since $\delta^- \in N$. Because $\sigma' \models \mathtt{gc}_{P,N}(\mathcal{C})$, we have $\sigma'(\delta^-) \subseteq \sigma'(\delta^+)$, so

$$\sigma(\delta^+) = \bigcup_{\delta^- \leqslant \delta^+} \sigma'(\delta^-) \subseteq \bigcup_{\delta^- \leqslant \delta^+} \sigma'(\delta^+) = \sigma'(\delta^+)$$

Finally, we need to show that σ satisfies all dirt constraints in $\mathcal{C}$. Take some $(\delta \leqslant \delta') \in \mathcal{C}$. Now, if $(\delta^- \leqslant \delta) \in \mathcal{C}$ holds for some $\delta^- \in N$, then $(\delta^- \leqslant \delta') \in \mathcal{C}$ holds as well because $\mathcal{C}$ is unified and therefore closed under logical implication. Thus $\sigma(\delta')$ is defined as a union over a bigger index set than $\sigma(\delta)$, so $\sigma(\delta) \subseteq \sigma(\delta')$ and $\sigma \models \delta \leqslant \delta'$.

For type parameters α, we proceed similarly and define

$$\sigma(\alpha) \stackrel{\text{def}}{=} \bigcup_{\substack{(\alpha^- \leqslant \alpha) \in \mathcal{C} \\ \alpha^- \in N}} \sigma'(\alpha^-)$$

where we define union over closed types as:

$$\begin{aligned}
\mathtt{bool} \cup \mathtt{bool} &= \mathtt{bool} & (A_1 \to \underline{C}_1) \cup (A_2 \to \underline{C}_2) &= (A_1 \cap A_2) \to (\underline{C}_1 \cup \underline{C}_2) \\
\mathtt{nat} \cup \mathtt{nat} &= \mathtt{nat} & E^{R_1} \cup E^{R_2} &= E^{R_1 \cup R_2} \\
\mathtt{unit} \cup \mathtt{unit} &= \mathtt{unit} & (\underline{C}_1 \Rightarrow \underline{D}_1) \cup (\underline{C}_2 \Rightarrow \underline{D}_2) &= (\underline{C}_1 \cap \underline{C}_2) \Rightarrow (\underline{D}_1 \cup \underline{D}_2) \\
\mathtt{empty} \cup \mathtt{empty} &= \mathtt{empty} & (A_1 \,!\, \Delta_1) \cup (A_2 \,!\, \Delta_2) &= (A_1 \cup A_2) \,!\, (\Delta_1 \cup \Delta_2)
\end{aligned}$$

The intersection is defined dually. We can see that the union is well defined because for all $(\alpha^- \leqslant \alpha) \in \mathcal{C}$, we have $\alpha^- \approx_{\mathcal{C}} \alpha$, hence all such α^- belong to the same skeleton of $\approx_{\mathcal{C}}$, and so also of $\approx_{\mathtt{gc}_{P,N}(\mathcal{C})}$. Since σ' is a solution of $\mathtt{gc}_{P,N}(\mathcal{C})$, all types $\sigma'(\alpha^-)$ are of the same shape.

We also need to consider the case when the union is empty because there are no $\alpha^- \in N$ such that $(\alpha^- \leqslant \alpha) \in \mathcal{C}$. In this case, we set $\sigma(\alpha)$ to be the intersection of $\sigma'(\alpha')$ for all $\alpha \approx_{\mathcal{C}} \alpha'$. Then, $\sigma(\alpha) \leqslant \sigma'(\alpha)$ holds in case $\alpha \in P$, and σ satisfies all constraints $(\alpha \leqslant \alpha') \in \mathcal{C}$. If we have a constraint $(\alpha' \leqslant \alpha) \in \mathcal{C}$, then α' also cannot have any negative lower bounds because $\mathcal{C}$ is closed, hence $\sigma(\alpha') = \sigma(\alpha)$. Finally, the case $\alpha \in N$ cannot occur.

For region parameters ρ, we need to take both instances and handled regions into an account. So, we define

$$\sigma(\rho) \stackrel{\text{def}}{=} \Big(\bigcup_{\substack{(\rho^- \leqslant \rho \cup \bigcup\!\!\bullet_i \dot\rho_i) \in \mathcal{C} \\ \rho^- \in N}} \big(\sigma'(\rho^-) - \bigcup\!\!\bullet_i \sigma'(\dot\rho_i)\big) \Big) \cup \Big(\bigcup_{(\mathtt{ins} \in \rho \cup \bigcup\!\!\bullet_i \dot\rho_i) \in \mathcal{C}} \big(\{\mathtt{ins}\} - \bigcup\!\!\bullet_i \sigma'(\dot\rho_i)\big) \Big)$$

As before, we get $\sigma'(\rho^-) \leqslant \sigma(\rho^-)$ for all $\rho^- \in N$ and $\sigma(\rho^+) \leqslant \sigma'(\rho^+)$ for all $\rho^+ \in P$. Next, let us show that σ satisfies all region constraints in $\mathcal{C}$. Let us consider only ones of the form $(\rho \leqslant \rho' \cup \bigcup\!\!\bullet_{i \in I} \dot\rho_i) \in \mathcal{C}$ because the proof for ones with instances is similar.

For any $(\rho^- \leqslant \rho \cup \bigcup\!\!\bullet_{i \in J} \dot\rho_i) \in \mathcal{C}$ contributing to $\sigma(\rho)$, we have $(\rho^- \leqslant \rho' \cup \bigcup\!\!\bullet_{i \in I \cup J} \dot\rho_i) \in \mathcal{C}$ because $\mathcal{C}$ is unified. Thus, $\sigma'(\rho^-) - \bigcup\!\!\bullet_{i \in I \cup J} \sigma'(\dot\rho_i) \subseteq \sigma(\rho')$ by the definition of $\sigma(\rho')$. If we add $\bigcup\!\!\bullet_{i \in I} \sigma'(\dot\rho_i)$ to both sides, we get

$$\big(\sigma'(\rho^-) - \bigcup\!\!\bullet_{i \in J} \sigma'(\dot\rho_i)\big) \subseteq \sigma(\rho') \cup \bigcup\!\!\bullet_{i \in I} \sigma'(\dot\rho_i)$$

We similarly get

$$\big(\{\mathtt{ins}\} - \bigcup\!\!\bullet_{i \in J} \sigma'(\dot\rho_i)\big) \subseteq \sigma(\rho') \cup \bigcup\!\!\bullet_{i \in I} \sigma'(\dot\rho_i)$$

for each $(\mathtt{ins} \in \rho \cup \bigcup\!\!\bullet_{i \in J} \dot\rho_i) \in \mathcal{C}$ that contributes to $\sigma(\rho)$. These are all contributions to $\sigma(\rho)$, thus

$$\sigma(\rho) \subseteq \sigma(\rho') \cup \bigcup\!\!\bullet_{i \in I} \sigma'(\dot\rho_i)$$

Now comes the critical step: since $\dot\rho_i \in P$, we have $\sigma(\dot\rho_i) \subseteq \sigma'(\dot\rho_i)$. However, $\sigma(\dot\rho_i)$ was non-empty and if we increase it, it contributes less to the singleton union, therefore

$$\sigma(\rho) \subseteq \sigma(\rho') \cup \bigcup\!\!\bullet_{i \in I} \sigma(\dot\rho_i)$$

and $\sigma \models (\rho \leqslant \rho' \cup \bigcup\!\!\bullet_{i \in I} \dot\rho_i)$.

A careful reader might have observed that not all parameters $\dot\rho_i$ occur in P, only those that appear in constraints $\rho^- \leqslant \rho^+ \cup \bigcup\!\!\bullet_i \dot\rho_i$ where $\rho^- \in N$ and $\rho^+ \in P_0$ (and similar ones for instances). However, this is easy to fix.

First, take P' to be P_0 and the set of all parameters $\dot\rho_i$ that appear in *any* singleton union in *any* constraint in $\mathcal{C}$. In this case, the above reasoning is valid and the set of constraints $\mathtt{gc}_{P',N}(\mathcal{C})$ is equivalent to $\mathcal{C}$. Now, repeat the whole process and take P'' to be P_0 and all parameters that appear in any singleton union in any constraint in $\mathtt{gc}_{P',N}(\mathcal{C})$. Again, $\mathtt{gc}_{P'',N}(\mathcal{C})$ is equivalent to $\mathtt{gc}_{P',N}(\mathcal{C})$ and so also to $\mathcal{C}$. However, the only region constraints left in $\mathtt{gc}_{P',N}(\mathcal{C})$ are ones of the form $\rho^- \leqslant \rho^+ \cup \bigcup\!\!\bullet_i \dot\rho_i$ with $\rho^- \in N$ and $\rho^+ \in P_0$, thus $P'' = P$.

In the end, let us show that $\mathtt{gc}_{P,N}(\mathcal{C})$ is unified if $\mathcal{C}$ is. We can consider only the case for type constraints as for other cases the proof is almost exactly the same. Take constraints $\alpha_1 \leqslant \alpha_2$ and $\alpha_2 \leqslant \alpha_3$ in $\mathtt{gc}_{P,N}(\mathcal{C})$. Since $\mathtt{gc}_{P,N}(\mathcal{C}) \subseteq \mathcal{C}$, these two constraints must also be in $\mathcal{C}$, which is unified, hence $\alpha_1 \leqslant \alpha_3$ is in $\mathcal{C}$ as well. From our assumption we have $\alpha_1, \alpha_2 \in N$ and $\alpha_2, \alpha_3 \in P$, therefore $(\alpha_1 \leqslant \alpha_3) \in \mathtt{gc}_{P,N}(\mathcal{C})$. We can similarly show that if we have $\alpha \leqslant \alpha' \in \mathtt{gc}_{P,N}(\mathcal{C})$, then $\alpha \approx_{\mathtt{gc}_{P,N}(\mathcal{C})} \alpha'$. □