Logical Methods in Computer Science
Volume 21, Issue 4, 2025, pp. 25:1–25:40
https://lmcs.episciences.org/

Submitted Apr. 08, 2024
Published Dec. 01, 2025

SIMPLIFYING EXPLICIT SUBTYPING COERCIONS IN A POLYMORPHIC CALCULUS WITH EFFECTS

FILIP KOPRIVEC [a,b] AND MATIJA PRETNAR [a,b]

[a] University of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, SI-1000 Ljubljana, Slovenia

[b] Institute of Mathematics, Physics and Mechanics, Jadranska 19, SI-1000 Ljubljana, Slovenia
e-mail address: filip.koprivec@fmf.uni-lj.si
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract. Algebraic effect handlers are becoming an increasingly popular way of structuring effectful computations, and their performance is often a concern. One of the proposed approaches towards efficient compilation is tracking effect information through explicit subtyping coercions. However, in the presence of polymorphism, these coercions are compiled into additional arguments of compiled functions, incurring significant overhead.

In this paper, we present a polymorphic effectful calculus, identify simplification phases needed to reduce the number of unnecessary constraints, and prove that they preserve semantics. In addition, we implement the simplification algorithm in the Eff language and evaluate its performance on a number of benchmarks. Though we do not prove the optimality of the presented simplifications, the results show that the algorithm eliminates all coercions, resulting in code as efficient as manually monomorphised one.

Introduction

Recent years have seen an increase in the number of programming languages that support algebraic effect handlers [PP03, PP13]. With a widespread usage, the need for performance is becoming ever more important. And there are two main ways of achieving it: an efficient low-level runtime [SDW+21, MGLZ24, MSSB25, MGJZ25] or an optimising compiler to a high-level language [SBO20, XL21, KKPS21], which we focus on in this paper.

Our recent work [KKPS21] has shown how an optimising compiler can take code written using the full flexibility of handlers, infer precise information about which parts of it use effects and which are pure, and produce code that matches conventional handcrafted one. The approach tracks effect information through explicit subtyping coercions [KPS+20], inserted around almost every sub-term, similar to where type constraints are introduced in Hindley-Milner type inference. In monomorphic code, most of these coercions become trivial and disappear, but for polymorphic functions, they need to be passed around as additional

Key words and phrases: Computational effects, Optimizing compilation, Algebraic effects, Polymorphic compilation, Subtyping, Denotational semantics.

This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

DOI:10.46298/LMCS-21(4:25)2025

parameters. As a result, the number of such parameters grows linearly with the size of the function body, rendering performance practically unusable.

In this paper, we propose an algorithm that soundly reduces (and often completely eliminates) redundant coercion parameters, leading to a performance comparable to that of monomorphic code. We start with an overview of the approach (Section 1) and continue with a specification of our working language (Section 2). Afterwards, we turn to our contributions, which are:

- Identifying requirements for a simplifying substitution to be correct with respect to typing (Sections 3 and 4).
- A number of progressive phases for simplification of constraints (Section 5).
- A proof that all the presented simplifications preserve the denotational semantics (Section 6).
- An implementation of the algorithm in a prototype language Eff and an evaluation of the impact it has on the code size and runtime (Section 7).

We conclude by discussing related and future work (Section 8). We assume readers to be familiar with algebraic effects and handlers, and refer them to a tutorial [Pre15] in case they are not. Unless explicitly noted, all proofs proceed by a straightforward structural induction. More details can be found in the first author's PhD thesis [Kop24].

1. Overview

1.1. **Explicit effect subtyping.** To see an example of the compilation pipeline in action, assume an operation `Random : unit -> bool` that produces random boolean values each time when triggered, and `Print : string -> unit` that prints a given string. Then, a function that randomly prints a given string is written in Eff as:

```
let print_randomly s =
  if (perform (Random ())) then (perform (Print s)) else ()
```

where the **perform** keyword triggers one of the operations.

The above function is first translated to Eff's core language CoreEff, which is a fine-grain call-by-value language [LPT03], meaning it distinguishes values and computations. For example, since **perform** `(Random ())` is a computation, and a conditional statement expects a boolean value, we must explicitly sequence the two computations and get:

$$
\begin{array}{l}
\texttt{fun}\ (s : \texttt{String}) \mapsto \\
\quad \texttt{do}\ b \leftarrow \mathit{Random}\ \texttt{unit}; \\
\quad \texttt{if}\ b\ \texttt{then} \\
\qquad \mathit{Print}\ s \\
\quad \texttt{else} \\
\qquad \texttt{return unit}
\end{array}
$$

As effects play a key role in the language, CoreEff computations are assigned types of the form $A\ !\ \Delta$, where A is the type of returned values, while a *dirt* $\Delta = \{op_1, \ldots, op_n\}$ captures the set of operations that the computation may perform.

In the above example, $\mathit{Random}\,\texttt{unit}$ is assigned the type $\texttt{Bool} \mathbin{!} \{\mathit{Random}\}$, while $\texttt{return unit}$ is assigned the type $\texttt{Unit} \mathbin{!} \emptyset$, and the fully annotated type of the function is $\texttt{String} \to \texttt{Unit} \mathbin{!} \{\mathit{Random}, \mathit{Print}\}$, which we abbreviate as $\texttt{String} \xrightarrow{\{\mathit{Random},\mathit{Print}\}} \texttt{Unit}$.

To accommodate for the differences in assigned effects, e.g. in the two conditional branches above, Eff features subtyping, which is a more flexible approach than row-based effect systems, as the latter can suffer from the *poisoning problem* [WJ99], as unification propagates effectful annotations even into pure parts of the program.

When inferring types, the inference algorithm also annotates certain terms with type coercions, which witness the compatibility of all types and effects. A fully annotated example with coercions highlighted in *gray* is:

$$
\begin{array}{l}
\texttt{fun}\ (s : \texttt{String}) \mapsto \\
\quad \texttt{do}\ b \leftarrow (\mathit{Random}\,\texttt{unit}) \rhd \langle \texttt{Bool} \rangle \mathbin{!} \gamma^1_{\texttt{d}}; \\
\quad \big(\texttt{if}\ b\ \texttt{then} \\
\qquad \mathit{Print}\, s \\
\quad \texttt{else} \\
\qquad (\texttt{return unit}) \rhd \langle \texttt{Unit} \rangle \mathbin{!} \gamma^2_{\texttt{d}} \\
\quad \big) \rhd \langle \texttt{Unit} \rangle \mathbin{!} \gamma^3_{\texttt{d}}
\end{array}
$$

Here $\gamma_{\texttt{v}} \mathbin{!} \gamma_{\texttt{d}}$ is a computation coercion that witnesses subtyping $A \mathbin{!} \Delta \leq A' \mathbin{!} \Delta'$ using $\gamma_{\texttt{v}} : A \leq A'$ for values and $\gamma_{\texttt{d}} : \Delta \leq \Delta'$ for effects. Note that even though effects are the only reason to include subtyping in Eff, we also need value coercions as higher-order functions need to accommodate arguments with differing effects. In the example, $\langle \texttt{Bool} \rangle : \texttt{Bool} \leq \texttt{Bool}$ and $\langle \texttt{Unit} \rangle : \texttt{Unit} \leq \texttt{Unit}$ are reflexive value coercions, while the dirt coercions are suitably constructed witnesses for:

$$
\begin{gathered}
\gamma^1_{\texttt{d}} : \{\mathit{Random}\} \leq \{\mathit{Random}, \mathit{Print}\} \\
\gamma^2_{\texttt{d}} : \emptyset \leq \{\mathit{Print}\} \\
\gamma^3_{\texttt{d}} : \{\mathit{Print}\} \leq \{\mathit{Random}, \mathit{Print}\}
\end{gathered}
$$

So, $\gamma^2_{\texttt{d}}$ increases the effects of the otherwise pure second branch to the effect $\{\mathit{Print}\}$ of the first branch, while $\gamma^1_{\texttt{d}}$ and $\gamma^3_{\texttt{d}}$ increase the effects of both sequenced computations to their union $\{\mathit{Random}, \mathit{Print}\}$.

1.2. **Compiling to a pure language.** The main advantage of using explicit witnesses for subtyping, rather than enforcing subtyping implicitly during inference, becomes evident at the compilation stage.

When targeting a functional language with no support for algebraic effects, effectful computations that yield a result of type A can be represented using a user-defined type $\texttt{Comp}\ A$, which typically uses one of the known encodings, such as free monads, delimited control, or continuation-passing style. While types $\texttt{Bool} \mathbin{!} \{\mathit{Random}\}$ or $\texttt{Bool} \mathbin{!} \{\mathit{Random}, \mathit{Print}\}$ can both be compiled to $\texttt{Comp Bool}$, computations of type $\texttt{Bool} \mathbin{!} \emptyset$ are pure and do not require monadic encoding. Instead, we want to compile them to (target language) computations of type $\texttt{Bool}$, gaining significant speed-up [KKPS21].

In Eff, subtyping coercions are used solely to ensure type compatibility and are discarded during evaluation. More interestingly, though, they gain computational content during translation, as according to the above efficient translation, they are mapped to injections from A to `Comp` A at the appropriate points. We do not go into the exact placement strategy here [KPS+20], but it is worth noting that this process is subtle even for first-order computations, let alone higher-order ones, which is why having explicit witnesses is crucial.

For example the computation `return` 42 : `Int` ! $\emptyset$ should be translated simply to 42 : `Int` in the target language. On the other hand, a cast (`return` 42) $\rhd$ $\gamma_{\mathtt{c}}$, where the coercion $\gamma_{\mathtt{c}}$: `Int` ! $\emptyset \leq$ `Int` ! $\{Random\}$ needs to end up in `Comp Int` and is thus translated as $return\,42$. Here, $return$: $A \to$ `Comp` A is the monadic injection in the target language (written as a variable to distinguish it from the keyword `return` in Eff). To complicate things even further, a nested cast such as ((`return` 42) $\rhd$ $\gamma_{\mathtt{c}}$) $\rhd$ $\gamma'_{\mathtt{c}}$ where $\gamma'_{\mathtt{c}}$: `Int` ! $\{Random\} \leq$ `Int` ! $\{Random, Print\}$ is still translated as $return\,42$ and not as $return\,(return\,42)$, which naive replacement of casts with $return$ would imply.

In implementation and in the following examples, we target OCaml, as its syntactic similarity to Eff makes the translation particularly straightforward. Even though version 5 of OCaml introduced native support for handlers, our compilation approach remains relevant: OCaml restricts handlers to continuations that may be resumed at most once, and many other target languages still lack native handler support. When translating Eff computations to OCaml, we target a monadic type `'a comp`, which features functions such as `performRandom : unit -> bool comp` for all the operations. The above example would thus be translated as

```
let print_randomly s =
  performRandom () >>= fun b ->
    if b then performPrint s else return ()
```

where `(>>=) : 'a comp -> ('a -> 'b comp) -> 'b comp` is the monadic bind, and `return : 'a -> 'a comp` is the monadic unit.

1.3. **Explicit polymorphism.** To see the issues that polymorphism brings to our pipeline, let us ignore effect annotations for a bit. Consider a polymorphic function that applies a given function `f` to its argument `x` only if a given predicate `p` is satisfied. In Eff, one would write it as:

```
let apply_if p f x = if p x then f x else x
```

When translating the above function to CoreEff, the inference algorithm again annotates the resulting term with types and coercions, each now featuring parameters due to the

presence of polymorphism:

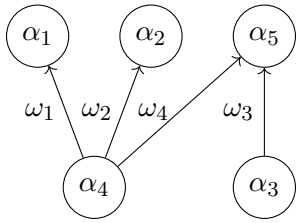
$$
\begin{array}{l}
\mathit{applyIf} = \texttt{fun}\ (p : \alpha_1 \to \texttt{Bool}) \mapsto \texttt{return}\ \big(\texttt{fun}\ (f : \alpha_2 \to \alpha_3) \mapsto \texttt{return}\ (\texttt{fun}\ (x : \alpha_4) \mapsto \\
\quad \texttt{do}\ b \leftarrow p\,(x \rhd \omega_1); \\
\quad \texttt{if}\ b\ \texttt{then} \\
\qquad (f\,(x \rhd \omega_2)) \rhd \omega_3 \\
\quad \texttt{else} \\
\qquad \texttt{return}\ (x \rhd \omega_4) \\
))
\end{array}
$$

In general, the type α_4 of x does not have to be the same as the argument type α_1 of p, it only needs to be its subtype, thus it needs to be cast by some type coercion $\omega_1 : \alpha_4 \leq \alpha_1$. A similar situation occurs with f and $\omega_2 : \alpha_4 \leq \alpha_2$. Finally, the result of f needs to be coerced to a supertype it shares with the other branch. As before, we express this by introducing a new type parameter α_5 for the output and coercions $\omega_3 : \alpha_3 \leq \alpha_5$ and $\omega_4 : \alpha_4 \leq \alpha_5$.

The inferred type of the function is then:

$$\mathit{applyIf} : (\alpha_1 \to \texttt{Bool}) \to (\alpha_2 \to \alpha_3) \to (\alpha_4 \to \alpha_5)$$

under the constraints described above. These are most easily expressed with a graph where nodes correspond to type parameters, while edges correspond to coercions between them:



When translating into OCaml, we now also need to take coercion parameters into account. Since their exact instantiations are known only at call-sites, we need to represent each parameter with an additional functional argument, so the (type-annotated) translation is:

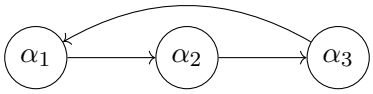
```
let apply_if (w1 : 'a4->'a1) (w2 : 'a4->'a2) (w3 : 'a3->'a5)
  (w4 : 'a4->'a5) (p : 'a1->bool) (f : 'a2->'a3) (x : 'a4) : 'a5 =

  p (x |> w1) >>= fun b ->
    if b then (f (x |> w2)) |> coer_comp w3
    else return (x |> w4)
```

where `coer_comp : ('a -> 'b) -> ('a comp -> 'b comp)` lifts value coercions to computations, and `(|>) : 'a -> ('a -> 'b) -> 'b` is the reverse application operator.

Each additional argument not only increases the function size, but also significantly impacts the runtime, especially as it soon prevents the OCaml compiler from passing function arguments in processor registers. In fact, the situation is even worse, as the above example was simplified for the presentation, while the actual inference algorithm uses additional intermediate type parameters leading to further arguments. A quick analysis of the Eff standard library shows that the number of explicit constraints produced by the current

algorithm is overwhelming. The standard `unzip` function needs 23 explicit coercions, while a straightforward implementation of the quicksort algorithm produces OCaml code with just under 200 explicit coercion parameters.

1.4. **Type constraint simplification.** Explicit type coercions are relevant during compilation, optimization and during execution, and therefore cannot be simply discarded, but it is obvious we want to reduce them to a minimum. One simple case of a simplification is when the coercions form a cycle in the graph, which often happens with recursive functions. For example, if we have three coercions:



all the parameters must be the same, and we can replace them with a single parameter α. Similarly, we can replace all coercions with the reflexive coercion $\langle\alpha\rangle : \alpha \leq \alpha$, essentially removing any casts in which they appear.

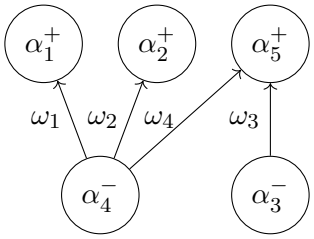
In a more general case, we have to be more cautious, though we can achieve similar results. To see how constraints impact the type of the program, we must first examine the positions at which parameters occur in types. For the sake of exposition, assume that our calculus features types $\texttt{Int} \leq \texttt{Float}$ (we are going to omit writing the coercion if we shall be interested only in the subtyping relation between the types).

Recall that subtyping of function types is *contravariant* in their domain. We have e.g. $(\texttt{Bool} \to \texttt{Int}) \leq (\texttt{Bool} \to \texttt{Float})$ because every function returning integers can also act as one returning floats. But on the other hand, we have $(\texttt{Float} \to \texttt{Bool}) \leq (\texttt{Int} \to \texttt{Bool})$ since every function accepting floats can also accept integers, and not the other way around.

Increasing an instantiation of a type parameter α in a covariant position (e.g. in $\texttt{Bool} \to \alpha$, but also in $(\alpha \to \texttt{Bool}) \to \texttt{Bool}$ where contravariance acts twice) thus increases the resulting instantiated type, while increasing an instantiation of a parameter in a contravariant position does the exact opposite. To track the impact of parameters on a type, we assign them a *polarity*: *positive* for those that appear covariantly in the type, and *negative* for those that appear contravariantly.

The running example above annotated with polarities is:

$$\mathit{applyIf} : (\alpha_1^+ \to \texttt{Bool}) \to (\alpha_2^+ \to \alpha_3^-) \to (\alpha_4^- \to \alpha_5^+)$$



As in this example, it is often the case that each parameter is either positive or negative and that constraints are of the form $X^- \leq Y^+$, however that does not hold in general!

Once parameters are assigned polarities, we can show that we lose no generality if we replace all positive parameters with their unique lower bounds, when they exist (and

conversely for negative parameters and unique upper bounds). For example, we can safely (proving that is one of the main results of this paper) replace our function with

$$\mathit{applyIf}' : (\alpha_4^{\pm} \to \texttt{Bool}) \to (\alpha_2^{+} \to \alpha_3^{-}) \to (\alpha_4^{\pm} \to \alpha_5^{+})$$

obtained by coalescing the positive parameter α_1^+ with its unique lower bound α_4^- (which now gains positive polarity) and replacing all occurrences of ω_1 in the body with $\langle \alpha_4^- \rangle$. On the surface, *applyIf′* appears less flexible than *applyIf*, but this can be amended at the call site.

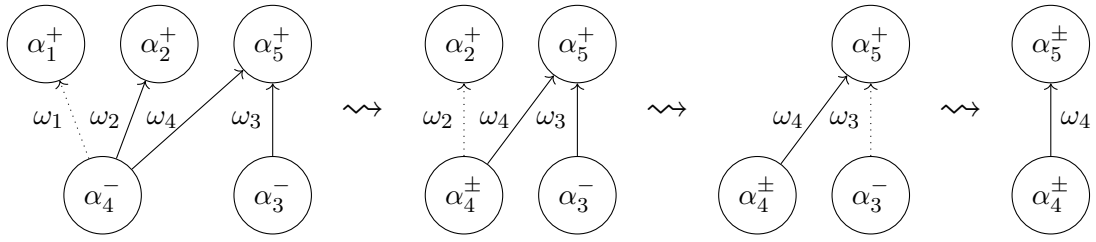
Indeed, any application $\mathit{applyIf}\ p\ f\ v$, where we instantiated all type parameters α_i with some types A_i and all coercion parameters ω_j with some coercions γ_j, can be replaced by an application $(\mathit{applyIf}' \rhd \gamma')\ p\ f\ v$ of a suitably coerced *applyIf′*, where γ' is a witness for

$$\big((A_4 \to \texttt{Bool}) \to (A_2 \to A_3) \to (A_4 \to A_5)\big) \le \big((A_1 \to \texttt{Bool}) \to (A_2 \to A_3) \to (A_4 \to A_5)\big)$$

which exists since γ_1 witnesses $A_4 \le A_1$ and α_1^+ was positive.

One could argue that we have replaced one coercion with an even more involved one, but this can often be optimized away due to the extra context information available at the call site. And even if we cannot completely remove γ', we have reduced the number of coercion parameters, which has a much more significant impact on performance.

We can continue the simplification by removing ω_2 and ω_3. We pick a particular order, though any other would lead to similar results:



ending up with the final type

$$(\alpha_4^{\pm} \to \texttt{Bool}) \to (\alpha_4^{\pm} \to \alpha_5^{\pm}) \to (\alpha_4^{\pm} \to \alpha_5^{\pm})$$

and the compiled OCaml function with a single remaining coercion parameter:

```
let apply_if w4 p f x =
  p x >>= fun b -> if b then f x else return (x |> w4)
```

Note that we cannot simplify the type any further because α_4 is positive and α_5 is negative.

Again, even though the simplified type seems more restricted than the original one, polarity of parameters again ensures that for any instantiation $\{(\alpha_i \mapsto A_i)_i, (\omega_j \mapsto \gamma_j)_j\}$, we have a witness

$$\big((A_4 \to \texttt{Bool}) \to (A_4 \to A_5) \to (A_4 \to A_5)\big) \le \big((A_1 \to \texttt{Bool}) \to (A_2 \to A_3) \to (A_4 \to A_5)\big)$$

constructed using coercions $\gamma_1 : A_4 \le A_1$, $\gamma_2 : A_4 \le A_2$, and $\gamma_3 : A_3 \le A_5$.

In general, the order in which we apply the simplifications affects possible future optimisations, however this is in practice more an exception than a rule. Our implementation, which chooses coercions in an arbitrary order, compiles the quicksort algorithm without any of the hundreds of additional parameters, and the same holds for the standard library of Eff, which contains around 50 of the most frequently used polymorphic functions (see Section 7.2).

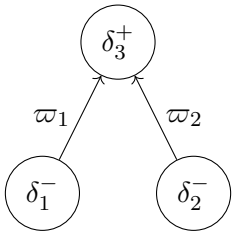
1.5. **Effect annotations.** Coming back to effect information, the annotated version of the running example becomes:

$$
\begin{array}{l}
\texttt{fun}\ (p : \alpha_1 \xrightarrow{\delta_1} \texttt{Bool}) \mapsto \texttt{return}\ (\\
\quad \texttt{fun}\ (f : \alpha_2 \xrightarrow{\delta_2} \alpha_3) \mapsto \texttt{return}\ (\\
\quad\quad \texttt{fun}\ (x : \alpha_4) \mapsto (\\
\quad\quad\quad \texttt{do}\ b \leftarrow (p\,(x \rhd \omega_1)) \rhd \langle \texttt{Bool} \rangle\ !\ \varpi_1\ ; \\
\quad\quad\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad\quad\quad (f\,(x \rhd \omega_2)) \rhd \omega_3\ !\ \varpi_2 \\
\quad\quad\quad \texttt{else} \\
\quad\quad\quad\quad (\texttt{return}\ x) \rhd \omega_4\ !\ \emptyset_{\delta_3} \\
)))
\end{array}
$$

Dirt parameters δ_1 and δ_2 capture sets of effects performed by p and f, respectively. We do not need the effects of p and f to be the same, we just need the effect of the whole computation to cover both. For that reason, we extend subtyping to dirt and capture the above by introducing a third dirt parameter δ_3 and two dirt coercion parameters $\varpi_1 : \delta_1 \leq \delta_3$ and $\varpi_2 : \delta_2 \leq \delta_3$.

In the example above, the first coercion casts the application of p to x. In it, we use ϖ_1 to cast the dirt, while for the type `Bool` of boolean values, the only option is the reflexive coercion $\langle \texttt{Bool} \rangle : \texttt{Bool} \leq \texttt{Bool}$. Next, we cast the application of f to x by extending the value coercion ω_3 from before with the dirt coercion ϖ_2. Finally, in the last cast, returned values cause no effects, so we need to cast the empty dirt to the common dirt δ_3, which we do using the empty coercion $\emptyset_{\delta_3} : \emptyset \leq \delta_3$.

Even though dirt coercions only inform the monadic structure and get erased when translating to OCaml, they are still present in the intermediate representation and other potential compilation targets, so we want to simplify them. This proceeds similarly as for type coercions. Incoming dirt parameters δ_1 and δ_2 receive negative polarity and outgoing dirt δ_3 receives the positive one. Thus, the constraint graph is of the form:



which we can simplify by replacing both δ_1 and δ_2 with δ_3 and both ϖ_1 and ϖ_2 with the reflexive coercion $\langle \delta_3 \rangle$.

With some renaming and writing $A \xrightarrow{\emptyset} B$ as $A \to B$, the final simplified type of `apply_if` is thus:

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

under the constraint $\alpha \leq \beta$. It is worth reiterating the simplified type is equivalent to the original type

$$(\alpha_1 \xrightarrow{\delta_1} \texttt{Bool}) \to (\alpha_2 \xrightarrow{\delta_2} \alpha_3) \to (\alpha_4 \xrightarrow{\delta_3} \alpha_5)$$

under the constraints $\alpha_3 \leq \alpha_5, \alpha_4 \leq \alpha_1, \alpha_4 \leq \alpha_2, \alpha_4 \leq \alpha_5, \delta_1 \leq \delta_3, \delta_2 \leq \delta_3$.

1.6. **Operations.** To account for both operations and dirt parameters, the general form of a dirt is $\mathcal{O} \cup \delta$, where δ is a *row* parameter capturing all operations not listed in $\mathcal{O}$. For example, here is a program that applies a given function to its argument with some probability:

```
let apply_randomly f x =
  if (perform Random) then f x else x
```

Simplification of type constraints removes all of them and the type of `apply_randomly` is:

$$(\alpha_1 \xrightarrow{\delta_1} \alpha_2) \to \alpha_1 \xrightarrow{\delta_2} \alpha_2$$

under the constraints $\delta_1 \leq \delta_2$ and $\{\mathit{Random}\} \leq \delta_2$. The latter constraint shows that δ_2 needs to contain Random, so we replace it with $\{\mathit{Random}\} \cup \delta_3$ for some fresh dirt parameter δ_3, ending up with a single constraint of the form

$$\delta_1 \leq \{\mathit{Random}\} \cup \delta_3$$

Furthermore, δ_1 is a negative parameter and can be safely increased to $\{\mathit{Random}\} \cup \delta_3$, leading to the final simplified type

$$(\alpha \xrightarrow{\{\mathit{Random}\} \cup \delta_3} \beta) \to \alpha \xrightarrow{\{\mathit{Random}\} \cup \delta_3} \beta$$

One may worry about possible inefficiency, since the functional argument could have been pure if δ_1 were set to $\emptyset$. But now, its expanded dirt always contains Random and is thus impure. In our current compiler, we treat all dirt parameters as potentially impure, so further increasing δ_1 has no effect. However, this is something to keep in mind when specialising effect-polymorphic functions into pure and impure variants (see Section 8.2).

While adding handlers provides additional flexibility to the programmer, it does not change the effect system significantly [BP14]. To incorporate first-class handlers, we introduce a new type constructor $A_1 \,!\, \Delta_1 \Rightarrow A_2 \,!\, \Delta_2$, which handles a computation of type $A_1 \,!\, \Delta_1$ into one of type $A_2 \,!\, \Delta_2$. For example, a handler that resolves all `Random` operations to `true` and counts the number of `Random` operations performed can be written as:

```
handler
| val x -> (x, 0)
| perform Random k -> let (res, cnt) = k true in (res, cnt + 1)
```

The type of this handler is

$$\alpha \,!\, (\{\mathit{Random}\} \cup \delta) \Rightarrow (\alpha \times \texttt{Int}) \,!\, \delta$$

So, it takes a probabilistic computation potentially performing Random and some other effects δ and producing values of type α, and handles it into a computation that performs only δ, and returns an integer in addition to the original result.

All of the above is more or less orthogonal to the rest of the work, so we will not go into further details.

2. Language

Let us now turn to a formal development of the above ideas. As our working language, we take CoreEff, a simple fine-grain call-by-value [LPT03] calculus with effects and explicit coercions. In contrast to our previous work [KPS+20], which used impredicative polymorphism, we use predicative polymorphism as it is simpler and sufficient for our use, especially as Eff generalizes only top-level let bindings [VJS10].

2.1. **Types.**

As we are working with a fine-grain call-by-value calculus, we distinguish between value and computation types, defined as:

$$\begin{array}{rr} \text{dirt} & \Delta ::= \delta \mid \emptyset \mid \{op\} \cup \Delta \\ \\ \text{value type} & A, B ::= \alpha \mid \texttt{Unit} \mid A \to \underline{C} \\ \text{computation type} & \underline{C} ::= A \,!\, \Delta \\ \\ \text{skeleton} & S ::= \varsigma \mid \texttt{Unit} \mid S_1 \to S_2 \end{array}$$

Dirt is, as before, a sequence of operations followed by either a row dirt parameter or the empty dirt. Note that the same set of operations $\mathcal{O}$ can be represented in multiple ways, all of which differ only in the ordering of operations. In this paper, we consider all such representations equivalent and treat dirt as being of the form $\mathcal{O} \cup \delta$ in case the row parameter is present, or $\mathcal{O}$ (i.e. $\mathcal{O} \cup \emptyset$) if it is not. The prototype implementation follows this convention as well, and represents dirt as an unordered set of operations and an optional dirt parameter, rather than an inductive sequence as defined by the above grammar.

For simplicity, value types are limited to type parameters, the unit type, and function types, which consist of an argument (value) type and a return (computation) type. Since functions take value arguments and perform computations, their type consists of a value type domain and a computation type codomain.

Computation types are a combination of a type of returned values with an associated dirt representing the operations that may be called. Importantly, the dirt conservatively over-approximates the set of operations that may be called.

For the rest of the paper, we assume a global signature

$$\Sigma = \{op_1 : A_1 \to B_1, op_2 : A_2 \to B_2, \ldots, op_n : A_n \to B_n\}$$

which assigns monomorphic types A_i and B_i to each operation op_i. Note that the arrow $\to$ in the operation signature is just the traditional syntax separating two value types, and should not be confused with a function type where the right-hand side is a computation type. To obtain a simpler denotational semantics (see Section 6.3), we assume the type B_i to be first-order, i.e. not a function type. Given that a similar but more involved semantics can be given for arbitrary types of returned values [BP14], we see no reason why our results would not hold in general.

In addition to types, we also introduce *skeletons* [KPS+20]. Our subtyping is concerned only with effect information and its direct impact on values via function types, but not with other forms of value subsumption, such as class inheritance or width and depth subtyping of record types. For that reason, one value can be a subtype of another only if their type constructors match, sometimes also dubbed *structural subtyping* [Pot96, Pot01, Sim03] (not to be confused with structural subtyping as the opposite of the nominal one). Skeletons concretely capture this shared structure and they behave like value types with all effect

$\Xi_s; \Xi_d; \Xi_t \vdash A : S$ assuming $\vdash_s \Xi_s$ `ctx` and $\Xi_s \vdash S$ `skel`

$$\frac{(\alpha : S) \in \Xi_t}{\Xi_s; \Xi_d; \Xi_t \vdash \alpha : S} \qquad \frac{\Xi_s; \Xi_d; \Xi_t \vdash A : S_1 \qquad \Xi_s; \Xi_d; \Xi_t \vdash \underline{C} : S_2}{\Xi_s; \Xi_d; \Xi_t \vdash A \to \underline{C} : S_1 \to S_2} \qquad \frac{}{\Xi_s; \Xi_d; \Xi_t \vdash \mathtt{Unit} : \mathtt{Unit}}$$

$\Xi_s; \Xi_d; \Xi_t \vdash \underline{C} : S$ assuming $\vdash_s \Xi_s$ `ctx` and $\Xi_s \vdash S$ `skel`

$$\frac{\Xi_s; \Xi_d; \Xi_t \vdash A : S \qquad \Xi_d \vdash \Delta \ \mathtt{dirt}}{\Xi_s; \Xi_d; \Xi_t \vdash A \,!\, \Delta : S}$$

FIGURE 1. Well-formedness rules for types

information erased. For example, both $\mathtt{Unit} \to \mathtt{Unit} \,!\, \{op\}$ and $\mathtt{Unit} \to \mathtt{Unit} \,!\, \{op, op'\}$ share the same skeleton $\mathtt{Unit} \to \mathtt{Unit}$.

Skeletons are a technical tool which becomes particularly useful in the presence of type parameters as instead of additionally tracking equivalence classes $\alpha_1 \approx \cdots \approx \alpha_n$ of comparable type parameters [Sim03, Pre14], one can simply assign them all the same skeleton S in some type parameter context $\alpha_1 : S, \ldots, \alpha_n : S$. In general, type parameter contexts and their skeleton and dirt counterparts (which are just lists of parameters), are given by the following grammar:

$$\begin{array}{rl} \text{skeleton parameter context} & \Xi_s ::= \varepsilon \mid \Xi_s, \varsigma \\ \text{dirt parameter context} & \Xi_d ::= \varepsilon \mid \Xi_d, \delta \\ \text{type parameter context} & \Xi_t ::= \varepsilon \mid \Xi_t, (\alpha : S) \end{array}$$

Well-formedness judgements of contexts, skeletons and dirts is routine and given by rules in Figure 9 in Appendix A, while the slightly more interesting rules for types are given in Figure 1.

2.2. **Coercions.** Coercions, denoted by γ are explicit witnesses for the subtyping relation. Similarly to types, coercions are split into ones for dirt, values, and computations.

$$\begin{array}{rl} \text{dirt coercion} & \gamma_{\mathtt{d}} ::= \varpi \mid \gamma^2_{\mathtt{d}} \circ \gamma^1_{\mathtt{d}} \mid \langle \delta \rangle \mid \langle \emptyset \rangle \mid \emptyset_\delta \mid \{op\} \cup \gamma_{\mathtt{d}} \mid \{op\} \cup^+ \gamma_{\mathtt{d}} \\ \text{value coercion} & \gamma_{\mathtt{v}} ::= \omega \mid \gamma^2_{\mathtt{v}} \circ \gamma^1_{\mathtt{v}} \mid \langle \alpha \rangle \mid \langle \mathtt{Unit} \rangle \mid \gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}} \\ \text{computation coercion} & \gamma_{\mathtt{c}} ::= \gamma_{\mathtt{v}} \,!\, \gamma_{\mathtt{d}} \end{array}$$

Dirt coercions can be either parameters, compositions, reflexive coercions for an arbitrary dirt parameter, reflexive empty dirt set coercion, coercion asserting that empty set is below any dirt parameter, or two extensions of coercions with operations. The first one asserts that the same operation can be added on both sides, while the second one asserts that the right hand side can be safely increased. Similarly to dirt, dirt coercions can be given with differing order of operations, which we do not distinguish.

Next, value coercions are again parameters, compositions, reflexive coercions for type parameters and the unit type, and function coercions $\gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}}$, which use $\gamma_{\mathtt{v}}$ to cast the argument and $\gamma_{\mathtt{c}}$ to cast the result. Computation coercions are just a combination of value coercion and dirt coercion.

Coercions require us to introduce two additional kinds of parameter contexts:

$$\begin{array}{rr} \text{dirt coercion parameter context} & \Xi_{dc} ::= \varepsilon \mid \Xi_{dc}, (\varpi : \Delta_1 \leq \Delta_2) \\ \text{type coercion parameter context} & \Xi_{tc} ::= \varepsilon \mid \Xi_{tc}, (\omega : A_1 \leq A_2) \end{array}$$

which are well-formed according to rules in Figure 10 in Appendix A.

For readability, we shall write the quintuple $(\Xi_s, \Xi_d, \Xi_t, \Xi_{dc}, \Xi_{tc})$ as a single, flat *parameter context* Ξ, formed as:

$$\frac{\vdash_s \Xi_s \ \mathtt{ctx} \qquad \vdash_d \Xi_d \ \mathtt{ctx} \qquad \Xi_s \vdash_t \Xi_t \ \mathtt{ctx} \qquad \Xi_d \vdash_{dc} \Xi_{dc} \ \mathtt{ctx} \qquad \Xi_s; \Xi_d; \Xi_t \vdash_{tc} \Xi_{tc} \ \mathtt{ctx}}{\vdash (\Xi_s, \Xi_d, \Xi_t, \Xi_{dc}, \Xi_{tc}) \ \mathtt{ctx}}$$

We shall use the joint context even in judgements that require only particular sub-contexts. For example, we shall write $\Xi \vdash A : S$ instead of the more verbose $\Xi_s; \Xi_d; \Xi_t \vdash A : S$ when $\Xi = (\Xi_s, \Xi_d, \Xi_t, \Xi_{dc}, \Xi_{tc})$.

Using the joint contexts, the well-formedness rules for coercions are given in Figure 2. Note that the subtyping on value and computation types is structural, i.e. both sides need to have the same skeleton.

$\Xi \vdash \gamma_{\mathtt{d}} : \Delta \leq \Delta'$ assuming $\vdash \Xi \ \mathtt{ctx}$ and $\Xi \vdash \Delta \ \mathtt{dirt}$ and $\Xi \vdash \Delta' \ \mathtt{dirt}$

$$\frac{(\varpi : \Delta \leq \Delta') \in \Xi}{\Xi \vdash \varpi : \Delta \leq \Delta'} \qquad \frac{\Xi \vdash \gamma^1_{\mathtt{d}} : \Delta \leq \Delta' \qquad \Xi \vdash \gamma^2_{\mathtt{d}} : \Delta' \leq \Delta''}{\Xi \vdash \gamma^2_{\mathtt{d}} \circ \gamma^1_{\mathtt{d}} : \Delta \leq \Delta''} \qquad \frac{\Xi \vdash \delta \ \mathtt{dirt}}{\Xi \vdash \langle \delta \rangle : \delta \leq \delta}$$

$$\frac{}{\Xi \vdash \langle \emptyset \rangle : \emptyset \leq \emptyset} \qquad \frac{}{\Xi \vdash \emptyset_\delta : \emptyset \leq \delta} \qquad \frac{op \in \Sigma \qquad \Xi \vdash \gamma_{\mathtt{d}} : \Delta \leq \Delta'}{\Xi \vdash \{op\} \cup \gamma_{\mathtt{d}} : \{op\} \cup \Delta \leq \{op\} \cup \Delta'}$$

$$\frac{op \in \Sigma \qquad \Xi \vdash \gamma_{\mathtt{d}} : \Delta \leq \Delta'}{\Xi \vdash \{op\} \cup^+ \gamma_{\mathtt{d}} : \Delta \leq \{op\} \cup \Delta'}$$

$\Xi \vdash \gamma_{\mathtt{v}} : A \leq A'$ assuming $\vdash \Xi \ \mathtt{ctx}$ and $\Xi \vdash S \ \mathtt{skel}$ and $\Xi \vdash A : S$ and $\Xi \vdash A' : S$

$$\frac{(\omega : A \leq A') \in \Xi}{\Xi \vdash \omega : A \leq A'} \qquad \frac{\Xi \vdash \gamma^1_{\mathtt{v}} : A \leq A' \qquad \Xi \vdash \gamma^2_{\mathtt{v}} : A' \leq A''}{\Xi \vdash \gamma^2_{\mathtt{v}} \circ \gamma^1_{\mathtt{v}} : A \leq A''} \qquad \frac{\Xi \vdash \alpha : S}{\Xi \vdash \langle \alpha \rangle : \alpha \leq \alpha}$$

$$\frac{}{\Xi \vdash \langle \mathtt{Unit} \rangle : \mathtt{Unit} \leq \mathtt{Unit}} \qquad \frac{\Xi \vdash \gamma_{\mathtt{v}} : A' \leq A \qquad \Xi \vdash \gamma_{\mathtt{c}} : \underline{C} \leq \underline{C'}}{\Xi \vdash \gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}} : (A \to \underline{C}) \leq (A' \to \underline{C'})}$$

$\Xi \vdash \gamma_{\mathtt{c}} : \underline{C} \leq \underline{C'}$ assuming $\vdash \Xi \ \mathtt{ctx}$ and $\Xi \vdash S \ \mathtt{skel}$ and $\Xi \vdash \underline{C} : S$ and $\Xi \vdash \underline{C'} : S$

$$\frac{\Xi \vdash \gamma_{\mathtt{v}} : A \leq A' \qquad \Xi \vdash \gamma_{\mathtt{d}} : \Delta \leq \Delta'}{\Xi \vdash \gamma_{\mathtt{v}} \ ! \ \gamma_{\mathtt{d}} : (A \ ! \ \Delta) \leq (A' \ ! \ \Delta')}$$

Figure 2. Coercion well-formedness rules

Even though we assume empty coercions only for the empty dirt and dirt parameters, they are admissible for any well-formed $\Xi \vdash \Delta\ \mathtt{dirt}$. For those, we can define the empty coercion $\Xi \vdash \emptyset_\Delta : \emptyset \leq \Delta$ by

$$\emptyset_\emptyset = \langle\emptyset\rangle \qquad \emptyset_{\{op\}\cup\Delta} = \{op\} \cup \emptyset_\Delta$$

Similarly, reflexive coercions are admissible for an arbitrary dirt $\Xi \vdash \langle\Delta\rangle : \Delta \leq \Delta$ by:

$$\langle\{op\} \cup \Delta\rangle = \{op\} \cup \langle\Delta\rangle$$

and for an arbitrary value or computation type by:

$$\langle A \to \underline{C}\rangle = \langle A\rangle \to \langle\underline{C}\rangle \qquad \langle A\ !\ \Delta\rangle = \langle A\rangle\ !\ \langle\Delta\rangle$$

Due to composition, we have multiple (semantically equivalent) coercions between two types, for example we can always compose a coercion with the reflexive one. In our previous work [KPS$^+$20], we preferred to keep the calculus minimal, but Proposition 4.7 requires compositions of arbitrary coercions, including parametric ones, therefore we add composition as an additional construct.

2.3. **Terms.** Last, and in fact least, we define the terms of the language. It turns out that it does not matter what terms we choose as long as typing judgements are preserved by substitutions (Theorem 3.4) and their denotational semantics respects subtyping (Theorem 6.2). The values are usual: variable, unit, function and also an explicit cast of a value. Computations are: return that lifts a value to a computation, operation calls, sequencing, application, and explicit casts of computations.

$$\text{typing context} \quad \Gamma ::= \varepsilon \mid \Gamma, (x : A)$$

$$\begin{aligned} \text{value} \quad & v ::= x \mid \mathtt{unit} \mid \mathtt{fun}\ (x : A) \mapsto c \mid v \rhd \gamma_{\mathtt{v}} \\ \text{computation} \quad & c ::= \mathtt{return}\ v \mid op\ v\ (y : A.c) \mid \mathtt{do}\ x \leftarrow c_1; c_2 \mid v_1\ v_2 \mid c \rhd \gamma_{\mathtt{c}} \end{aligned}$$

Typing contexts are well-formed in a parameter context according to rules in Figure 11 in Appendix A, while terms are typed according to rules in Figure 3. As typing contexts are secondary in our development, we shall always use their full name, and simply use *context* to talk about parameter contexts Ξ.

3. Substitutions

Most of our work focuses on translating types and terms in a given context into ones in a simpler context. With that purpose, we define *substitutions* σ as:

$$\text{substitution} \quad \sigma ::= \emptyset \mid \sigma, (\varsigma \mapsto S) \mid \sigma, (\delta \mapsto \Delta) \mid \sigma, (\alpha \mapsto A) \mid \sigma, (\omega \mapsto \gamma_{\mathtt{v}}) \mid \sigma, (\varpi \mapsto \gamma_{\mathtt{d}})$$

that maps skeleton parameters to skeletons, dirt parameters to dirt, and so on. We assume that each parameter is assigned at most one counterpart, so we can interpret each substitution as a function. When easier, we will use the function like notation (e.g. $\sigma(\varsigma) = S$) to denote the action of the substitution on a given parameter. We shall also use $\sigma \setminus \mathcal{P}$ to denote the substitution σ with the removal of all parameters in the set $\mathcal{P}$.

$\Xi; \Gamma \vdash v : A$ assuming $\Xi \vdash \Gamma$ `tyCtx` and $\Xi \vdash A : S$

$$\frac{x : A \in \Gamma}{\Xi; \Gamma \vdash x : A} \qquad \frac{}{\Xi; \Gamma \vdash \texttt{unit} : \texttt{Unit}} \qquad \frac{\Xi; \Gamma, x : A \vdash c : \underline{C}}{\Xi; \Gamma \vdash \texttt{fun}\ (x : A) \mapsto c : A \to \underline{C}}$$

$$\frac{\Xi; \Gamma \vdash v : A \qquad \Xi \vdash \gamma_{\mathtt{v}} : A \leq A'}{\Xi; \Gamma \vdash v \rhd \gamma_{\mathtt{v}} : A'}$$

$\Xi; \Gamma \vdash c : \underline{C}$ assuming $\Xi \vdash \Gamma$ `tyCtx` and $\Xi \vdash \underline{C} : S$

$$\frac{\Xi; \Gamma \vdash v : A}{\Xi; \Gamma \vdash \texttt{return}\ v : A\ !\ \emptyset} \qquad \frac{\Xi; \Gamma \vdash c_1 : A\ !\ \Delta \qquad \Xi; \Gamma, x : A \vdash c_2 : A'\ !\ \Delta}{\Xi; \Gamma \vdash \texttt{do}\ x \leftarrow c_1; c_2 : A'\ !\ \Delta}$$

$$\frac{\Xi; \Gamma \vdash v_1 : A \to \underline{C} \qquad \Xi; \Gamma \vdash v_2 : A}{\Xi; \Gamma \vdash v_1\ v_2 : \underline{C}}$$

$$\frac{op : A_1 \to A_2 \in \Sigma \qquad \Xi; \Gamma \vdash v : A_1 \qquad \Xi; \Gamma, y : A_2 \vdash c : A\ !\ \Delta \qquad op \in \Delta}{\Xi; \Gamma \vdash op\ v\ (y : A_2.c) : A\ !\ \Delta}$$

$$\frac{\Xi; \Gamma \vdash c : \underline{C} \qquad \Xi \vdash \gamma_{\mathtt{c}} : \underline{C} \leq \underline{C}'}{\Xi; \Gamma \vdash c \rhd \gamma_{\mathtt{c}} : \underline{C}'}$$

Figure 3. CoreEff typing rules

3.1. **Valid substitutions.** A *valid substitution* $\vdash \sigma : \Xi \Rightarrow \Xi'$ takes each parameter from a well-formed parameter context Ξ into its counterpart, which is well-formed in context Ξ', as given by rules presented in the Figure 4.

$\vdash \sigma : \Xi \Rightarrow \Xi'$ assuming $\vdash \Xi$ `ctx` and $\vdash \Xi'$ `ctx`

$$\frac{}{\vdash \emptyset : \varepsilon \Rightarrow \Xi'} \qquad \frac{\vdash \sigma : \Xi \Rightarrow \Xi' \qquad \Xi' \vdash S\ \texttt{skel}}{\vdash (\sigma, \varsigma \mapsto S) : (\Xi, \varsigma) \Rightarrow \Xi'} \qquad \frac{\vdash \sigma : \Xi \Rightarrow \Xi' \qquad \Xi' \vdash \Delta\ \texttt{dirt}}{\vdash (\sigma, \delta \mapsto \Delta) : (\Xi, \delta) \Rightarrow \Xi'}$$

$$\frac{\vdash \sigma : \Xi \Rightarrow \Xi' \qquad \Xi' \vdash A : \sigma(S)}{\vdash (\sigma, \alpha \mapsto A) : (\Xi, (\alpha : S)) \Rightarrow \Xi'} \qquad \frac{\vdash \sigma : \Xi \Rightarrow \Xi' \qquad \Xi' \vdash \gamma_{\mathtt{v}} : \sigma(A_1) \leq \sigma(A_2)}{\vdash (\sigma, \omega \mapsto \gamma_{\mathtt{v}}) : (\Xi, (\omega : A_1 \leq A_2)) \Rightarrow \Xi'}$$

$$\frac{\vdash \sigma : \Xi \Rightarrow \Xi' \qquad \Xi' \vdash \gamma_{\mathtt{d}} : \sigma(\Delta_1) \leq \sigma(\Delta_2)}{\vdash (\sigma, \varpi \mapsto \gamma_{\mathtt{d}}) : (\Xi, (\varpi : \Delta_1 \leq \Delta_2)) \Rightarrow \Xi'}$$

Figure 4. Substitution validity rules

Note that validity rules ensure that substitutions are defined on all parameters in the source context, even if they map back to themselves. To prevent unnecessary clutter, we will omit these trivial mappings.

We will mostly be interested in substitutions $\vdash \sigma : \Xi \Rightarrow \Xi'$ where Ξ' is a subset of Ξ, and σ preserves all their common parameters. In such cases, we shall call σ a *strengthening* of Ξ and denote Ξ' as $\sigma(\Xi)$. For example

$$\{\varsigma' \mapsto \mathtt{Unit}, \delta' \mapsto \delta, \alpha \mapsto \mathtt{Unit}\}$$

(where we omit $\varsigma \mapsto \varsigma, \delta \mapsto \delta$), strengthens $\Xi = \varsigma, \varsigma', \delta, \delta', (\alpha : \varsigma')$ to $\Xi' = \varsigma, \delta$.

As expected, valid substitutions preserve well-formedness and typing judgements, where the action of a substitution is extended from parameters to arbitrary types in Figure 5, arbitrary coercions in Figure 6, and arbitrary typing contexts and terms in Figure 7.

$$\sigma(\mathtt{Unit}) = \mathtt{Unit} \qquad \sigma(S_1 \to S_2) = \sigma(S_1) \to \sigma(S_2)$$

$$\sigma(\emptyset) = \emptyset \qquad \sigma(\{op\} \cup \Delta) = \{op\} \cup \sigma(\Delta)$$

$$\sigma(\mathtt{Unit}) = \mathtt{Unit} \qquad \sigma(A \to \underline{C}) = \sigma(A) \to \sigma(\underline{C})$$
$$\sigma(A \mathbin{!} \underline{C}) = \sigma(A) \mathbin{!} \sigma(\underline{C})$$

FIGURE 5. Substitution action on skeletons, dirt, and types

Proposition 3.1. *For an arbitrary valid substitution $\vdash \sigma : \Xi \Rightarrow \Xi'$, the following holds:*
- *if $\Xi \vdash S$ $\mathtt{skel}$, then $\Xi' \vdash \sigma(S)$ $\mathtt{skel}$,*
- *if $\Xi \vdash \Delta$ $\mathtt{dirt}$, then $\Xi' \vdash \sigma(\Delta)$ $\mathtt{dirt}$.*

Proposition 3.2. *For an arbitrary valid substitution $\vdash \sigma : \Xi \Rightarrow \Xi'$, the following holds:*
- *if $\Xi \vdash A : S$, then $\Xi' \vdash \sigma(A) : \sigma(S)$,*
- *if $\Xi \vdash \underline{C} : S$, then $\Xi' \vdash \sigma(\underline{C}) : \sigma(S)$.*

Proposition 3.3. *For an arbitrary valid substitution $\vdash \sigma : \Xi \Rightarrow \Xi'$, the following holds:*
- *if $\Xi \vdash \gamma_{\mathtt{v}} : A_1 \leq A_2$, then $\Xi' \vdash \sigma(\gamma_{\mathtt{v}}) : \sigma(A_1) \leq \sigma(A_2)$,*

$$\sigma(\langle\delta\rangle) = \langle\sigma(\delta)\rangle \qquad \sigma(\langle\emptyset\rangle) = \langle\emptyset\rangle$$
$$\sigma(\emptyset_\delta) = \emptyset_{\sigma(\delta)} \qquad \sigma(\{op\} \cup^+ \gamma_{\mathtt{d}}) = \{op\} \cup^+ \sigma(\gamma_{\mathtt{d}})$$
$$\sigma(\{op\} \cup \gamma_{\mathtt{d}}) = \{op\} \cup \sigma(\gamma_{\mathtt{d}})$$

$$\sigma(\langle\mathtt{Unit}\rangle) = \langle\mathtt{Unit}\rangle \qquad \sigma(\langle\alpha\rangle) = \langle\sigma(\alpha)\rangle$$
$$\sigma(\gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}}) = \sigma(\gamma_{\mathtt{v}}) \to \sigma(\gamma_{\mathtt{c}}) \qquad \sigma(\gamma_{\mathtt{v}} \mathbin{!} \gamma_{\mathtt{d}}) = \sigma(\gamma_{\mathtt{v}}) \mathbin{!} \sigma(\gamma_{\mathtt{d}})$$

FIGURE 6. Substitution action on dirt and type coercions

$$\sigma(\varepsilon) = \varepsilon \qquad \sigma(\Gamma, x : A) = \sigma(\Gamma), x : \sigma(A)$$

$$\sigma(x) = x \qquad \sigma(\mathtt{unit}) = \mathtt{unit}$$
$$\sigma(\mathtt{fun}\ x : A \mapsto c) = \mathtt{fun}\ x : \sigma(A) \mapsto \sigma(c) \qquad \sigma(v \rhd \gamma_{\mathtt{v}}) = \sigma(v) \rhd \sigma(\gamma_{\mathtt{v}})$$

$$\sigma(\mathtt{return}\ v) = \mathtt{return}\ \sigma(v) \qquad \sigma(\mathtt{do}\ x \leftarrow v; c) = \mathtt{do}\ x \leftarrow \sigma(v); \sigma(c)$$
$$\sigma(v_1\ v_2) = \sigma(v)\ \sigma(w) \qquad \sigma(op\ v\ (y : A.c)) = op\ \sigma(v)\ (y : \sigma(A).\sigma(c))$$
$$\sigma(v \rhd \gamma_{\mathtt{c}}) = \sigma(v) \rhd \sigma(\gamma_{\mathtt{c}})$$

FIGURE 7. Substitution action on typing contexts and terms

- *if* $\Xi \vdash \gamma_{\mathtt{c}} : \underline{C}_1 \leq \underline{C}_2$, *then* $\Xi' \vdash \sigma(\gamma_{\mathtt{c}}) : \sigma(\underline{C}_1) \leq \sigma(\underline{C}_2)$,
- *if* $\Xi \vdash \gamma_{\mathtt{d}} : \Delta_1 \leq \Delta_2$, *then* $\Xi' \vdash \sigma(\gamma_{\mathtt{d}}) : \sigma(\Delta_1) \leq \sigma(\Delta_2)$.

Theorem 3.4. *For an arbitrary valid substitution* $\vdash \sigma : \Xi \Rightarrow \Xi'$*, the following holds:*

- *if* $\Xi; \Gamma \vdash v : A$, *then* $\Xi'; \sigma(\Gamma) \vdash \sigma(v) : \sigma(A)$,
- *if* $\Xi; \Gamma \vdash c : \underline{C}$, *then* $\Xi'; \sigma(\Gamma) \vdash \sigma(c) : \sigma(\underline{C})$.

For substitutions σ and σ', we can define their *composition* as:

$$(\sigma' \circ \sigma)(\varsigma) = \sigma'(\sigma(\varsigma)) \qquad (\sigma' \circ \sigma)(\alpha) = \sigma'(\sigma(\alpha))$$
$$(\sigma' \circ \sigma)(\delta) = \sigma'(\sigma(\delta)) \qquad (\sigma' \circ \sigma)(\omega) = \sigma'(\sigma(\omega))$$
$$(\sigma' \circ \sigma)(\varpi) = \sigma'(\sigma(\varpi))$$

Proposition 3.5. *Let* $\vdash \sigma : \Xi \Rightarrow \Xi'$ *and* $\vdash \sigma' : \Xi' \Rightarrow \Xi''$ *be valid substitutions. Then the composition of the two substitutions is a valid substitution* $\vdash \sigma' \circ \sigma : \Xi \Rightarrow \Xi''$.

3.2. **A canonical form of contexts.** To ease further analysis, we first reduce contexts to a canonical form, following similar ideas as our previous unification algorithms [Pre14, SKPS18, KPS+20]. Recall that parameter contexts are of the form $(\Xi_s, \Xi_d, \Xi_t, \Xi_{dc}, \Xi_{tc})$, where:

- Ξ_s lists skeleton parameters ς,
- Ξ_d lists dirt parameters δ,
- Ξ_t lists type parameters and their skeletons $\alpha : S$,
- Ξ_{dc} lists dirt coercion parameters $\varpi : \Delta_1 \leq \Delta_2$, and
- Ξ_{tc} lists type coercion parameters $\omega : A_1 \leq A_2$.

Contexts Ξ_s and Ξ_d are just lists of parameters, so we leave them as is. However, due to structural subtyping, the type parameters Ξ_t can be reduced to ones of the form $\alpha : \varsigma$, so with only parameter skeletons. For example, if we have $(\alpha : S_1 \to S_2) \in \Xi_t$, the only closed types we can assign to α are function types. Thus, we can introduce two type parameters $\alpha_1 : S_1, \alpha_2 : S_2$, and replace all occurrences of α with $\alpha_1 \to \alpha_2$.

With that in mind, we define an auxiliary function $\varphi_t(\Xi_t; \Xi'_t, \Xi'_d, \sigma)$ that processes a context Ξ_t one parameter at a time and accumulates canonical type parameters Ξ'_t,

additionally generated dirt parameters Ξ'_d, and a reducing substitution σ:

$$\begin{aligned}
\varphi_t(\varepsilon; \Xi'_t, \Xi'_d, \sigma) &:= \big(\Xi'_t, \Xi'_d, \sigma\big) \\
\varphi_t((\Xi_t, \alpha : \varsigma); \Xi'_t, \Xi'_d, \sigma) &:= \varphi_t(\Xi_t; (\Xi'_t, \alpha : \varsigma), \Xi'_d, \sigma) \\
\varphi_t((\Xi_t, \alpha : \mathtt{Unit}); \Xi'_t, \Xi'_d, \sigma) &:= \varphi_t(\Xi_t; \Xi'_t, \Xi'_d, \{\alpha \mapsto \mathtt{Unit}\} \circ \sigma) \\
\varphi_t((\Xi_t, \alpha : S_1 \to S_2); \Xi'_t, \Xi'_d, \sigma) &:= \varphi_t((\Xi_t, \alpha_1 : S_1, \alpha_2 : S_2); \Xi'_t, (\Xi'_d, \delta), \{\alpha \mapsto \alpha_1 \xrightarrow{\delta} \alpha_2\} \circ \sigma) \\
&\qquad (\delta, \alpha_1, \alpha_2 \text{ fresh})
\end{aligned}$$

Note that even though the number of variables in the unreduced context Ξ_t increases in the last case, the total size of accompanying skeletons decreases, ensuring termination.

After reducing type parameters, we turn to type constraints, as reducing those may introduce further dirt constraints. Using a similar approach as for types, we reduce type constraints to ones witnessing $\alpha_1 \leq \alpha_2$, so only inequalities between type parameters. Like before, we define a function $\varphi_{tc}(\Xi_{tc}; \Xi'_{tc}, \Xi'_{dc}, \sigma)$ that processes Ξ_{tc} and accumulates reduced type constraints Ξ'_{tc}, additionally generated dirt constraints Ξ'_{dc}, and a substitution σ like before:

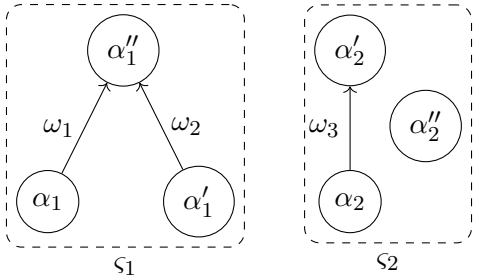
$$\begin{aligned}
\varphi_{tc}(\varepsilon; \Xi'_{tc}, \Xi'_{dc}, \sigma) &:= \big(\Xi'_{tc}, \Xi'_{dc}, \sigma\big) \\
\varphi_{tc}((\Xi_{tc}, \omega : \alpha_1 \leq \alpha_2); \Xi'_{tc}, \Xi'_{dc}, \sigma) &:= \varphi_{tc}(\Xi_{tc}; (\Xi'_{tc}, \omega : \alpha_1 \leq \alpha_2), \Xi'_{dc}, \sigma) \\
\varphi_{tc}((\Xi_{tc}, \omega : \mathtt{Unit} \leq \mathtt{Unit}); \Xi'_{tc}, \Xi'_{dc}, \sigma) &:= \varphi_{tc}(\Xi_{tc}; \Xi'_{tc}, \Xi'_{dc}, \{\omega \mapsto \langle \mathtt{Unit} \rangle\} \circ \sigma) \\
\varphi_{tc}((\Xi_{tc}, \omega : A_1 \xrightarrow{\Delta} A_2 \leq A'_1 \xrightarrow{\Delta'} A'_2); \Xi'_{tc}, \Xi'_{dc}, \sigma) &:= \\
\varphi_{tc}((\Xi_{tc}, \omega_1 : A'_1 \leq A_1, \omega_2 : A_2 \leq A'_2); \Xi'_{tc}, (\Xi'_{dc}, \varpi : \Delta \leq \Delta'), \{\omega \mapsto (\omega_1 \xrightarrow{\varpi} \omega_2)\} \circ \sigma) & \\
\text{where } \omega_1, \omega_2, \varpi \text{ fresh} &
\end{aligned}$$

Since well-formedness rules ensure that both sides of type inequalities share the same skeleton, and since the previous stage removed all type parameters with non-parameter skeletons, these four clauses cover all possible cases.

We can visualise reduced contexts with directed graphs. First, we have a graph representing type coercions. Its nodes are type parameters, and its edges are type coercion parameters between them. As we can compare only types with the same skeleton, the graph decomposes into separate subgraphs, each corresponding to a single skeleton parameter. For example, the reduced context

$$\begin{aligned}
&\varsigma_1, \varsigma_2, (\alpha_1 : \varsigma_1), (\alpha'_1 : \varsigma_1), (\alpha''_1 : \varsigma_1), (\alpha_2 : \varsigma_2), (\alpha'_2 : \varsigma_2), (\alpha''_2 : \varsigma_2) \\
&\qquad (\omega_1 : \alpha_1 \leq \alpha''_1), (\omega_2 : \alpha'_1 \leq \alpha''_1), (\omega_3 : \alpha_2 \leq \alpha'_2)
\end{aligned}$$

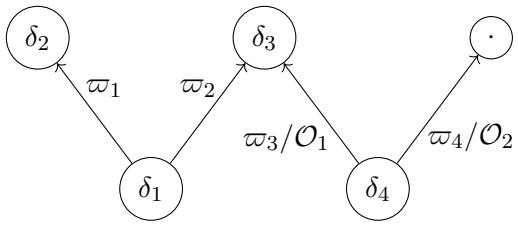
can be concisely captured with the graph



Similarly, we shall reduce dirt coercions. Unlike type coercions, we can reduce only the left-hand sides [KPS+20], ending up with inequalities of the form $\delta_1 \leq \mathcal{O}$ and $\delta_1 \leq \mathcal{O} \cup \delta_2$. Still, we end up with a graph that has dirt parameters as nodes and dirt coercion parameters as edges between them. Additionally, each edge is annotated with an operation set to account for the right-hand side of dirt inequalities, and we need an additional sink node to account for coercions with no parameter on the right. In Section 5.2, we shall explain why we can put operations on edges rather than add additional nodes for each combination $\mathcal{O} \cup \delta$. For example, the reduced context

$$\delta_1, \delta_2, \delta_3, \delta_4, (\varpi_1 : \delta_1 \leq \delta_2), (\varpi_2 : \delta_1 \leq \delta_3), (\varpi_3 : \delta_4 \leq \mathcal{O}_1 \cup \delta_3), (\varpi_4 : \delta_4 \leq \mathcal{O}_2)$$

can be captured with the graph



Reduced type parameters and type coercions involve only parameters, and are as such completely independent from dirt coercions. As a result, we can represent any reduced context with two graphs: one for type coercions, and one for dirt coercions.

We reduce dirt coercion contexts using a function $\varphi_{dc}(\Xi_{dc}; \Xi'_{dc}, \Xi'_d, \sigma)$ like before. Recall that each dirt is either of the form $\mathcal{O}$ or $\mathcal{O} \cup \delta$, so in addition to the terminal case (1), we need to consider four cases, accounting for each option on both sides of the inequality. Furthermore, each case is split into two further subcases, depending if $\mathcal{O}_1 \subseteq \mathcal{O}_2$ or not. First, we have the inequalities where the right-hand side is of the closed form $\mathcal{O}_2$:

$$\varphi_{dc}(\varepsilon; \Xi'_{dc}, \Xi'_d, \sigma) := \left(\Xi'_{dc}, \Xi'_d, \sigma\right) \tag{1}$$

$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \leq \mathcal{O}_2 \text{ for } \mathcal{O}_1 \subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \tag{2.1}$$
$$\varphi_{dc}(\Xi_{dc}; \Xi'_{dc}, \Xi'_d, \{\varpi \mapsto \mathcal{O}_1 \cup \emptyset_{\mathcal{O}_2 \setminus \mathcal{O}_1}\} \circ \sigma)$$
$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \leq \mathcal{O}_2 \text{ for } \mathcal{O}_1 \not\subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \text{fail} \tag{2.2}$$

$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \cup \delta_1 \leq \mathcal{O}_2 \text{ for } \mathcal{O}_1 \subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \tag{3.1}$$
$$\varphi_{dc}(\Xi_{dc}; (\Xi'_{dc}, \varpi' : \delta_1 \leq \mathcal{O}_2), \Xi'_d, \{\varpi \mapsto \mathcal{O}_1 \cup \varpi'\} \circ \sigma)$$
$$\text{where } \varpi' \text{ fresh}$$
$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \cup \delta_1 \leq \mathcal{O}_2 \text{ for } \mathcal{O}_1 \not\subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \text{fail} \tag{3.2}$$

followed by the inequalities where the right hand side $\mathcal{O}_2 \cup \delta_2$ features a dirt parameter:

$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \leq \mathcal{O}_2 \cup \delta_2 \text{ for } \mathcal{O}_1 \subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \varphi_{dc}(\Xi_{dc}; \Xi'_{dc}, \Xi'_d, \sigma) \tag{4.1}$$

$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \leq \mathcal{O}_2 \cup \delta_2 \text{ for } \mathcal{O}_1 \not\subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \tag{4.2}$$
$$\varphi_{dc}(\sigma'(\Xi_{dc}, \Xi'_{dc}); \varepsilon, (\Xi'_d \setminus \{\delta_2\}, \delta'_2), \sigma' \circ \sigma)$$
$$\text{where } \delta'_2 \text{ fresh}, \sigma' = \{\delta_2 \mapsto (\mathcal{O}_1 \setminus \mathcal{O}_2) \cup \delta'_2\}$$

$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \cup \delta_1 \leq \mathcal{O}_2 \cup \delta_2 \text{ for } \mathcal{O}_1 \subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \tag{5.1}$$
$$\varphi_{dc}((\Xi_{dc}; (\Xi'_{dc}, \varpi' : \delta_1 \leq (\mathcal{O}_2 \setminus \mathcal{O}_1) \cup \delta'_2); \Xi'_d, \{\varpi \mapsto \mathcal{O}_1 \cup \varpi'\} \circ \sigma)$$
$$\text{where } \varpi' \text{ fresh}$$

$$\varphi_{dc}((\Xi_{dc}, \varpi : \mathcal{O}_1 \cup \delta_1 \leq \mathcal{O}_2 \cup \delta_2 \text{ for } \mathcal{O}_1 \not\subseteq \mathcal{O}_2); \Xi'_{dc}, \Xi'_d, \sigma) := \tag{5.2}$$
$$\varphi_{dc}((\sigma'(\Xi_{dc}, \Xi'_{dc})); (\varpi' : \delta_1 \leq (\mathcal{O}_2 \setminus \mathcal{O}_1) \cup \delta'_2), (\Xi'_d \setminus \{\delta_2\}, \delta'_2), \{\varpi \mapsto \mathcal{O}_1 \cup \varpi'\} \circ \sigma' \circ \sigma)$$
$$\text{where } \delta'_2, \varpi' \text{ fresh}, \sigma' = \{\delta_2 \mapsto (\mathcal{O}_1 \setminus \mathcal{O}_2) \cup \delta'_2\}$$

In cases (2.1)–(3.2), we have only $\mathcal{O}_2$ on the right-hand side, so $\mathcal{O}_1 \subseteq \mathcal{O}_2$ must hold for an instantiation to exist. If it holds, we can factor out $\mathcal{O}_1$ from both sides of the coercion ϖ, however note that for flexibility we keep the less restrictive coercion $\varpi' : \delta \leq \mathcal{O}_2$ in Ξ'_{dc} rather than $\delta \leq \mathcal{O}_2 \setminus \mathcal{O}_1$ in case (3.1).

Cases (4.1) and (5.1) are similar to the ones before. In cases (4.2) and (5.2), however, the dirt parameter δ_2 can absorb the difference $\mathcal{O}_1 \setminus \mathcal{O}_2$. Thus, we do not fail, but replace δ_2 with $(\mathcal{O}_1 \setminus \mathcal{O}_2) \cup \delta'_2$. As δ_2 could have appeared in already reduced coercions Ξ'_{dc}, we need to reintroduce them back in the unreduced context Ξ_{dc} and start afresh. Even though in this case, the reducing context Ξ_{dc} increases in size, the set of operations that parameters in Ξ_d can absorb in the future gets smaller, ensuring termination.

Finally, we can merge all three stages into a single substitution:

$$\begin{aligned}
&\Phi_{\texttt{red}}(\Xi_s, \Xi_d, \Xi_t, \Xi_{dc}, \Xi_{tc}) = \\
&\quad \text{let } \left(\Xi'_t, \Xi'_d, \sigma_t\right) = \varphi_t(\Xi_t; \varepsilon, \Xi_d, \emptyset) \\
&\quad \text{let } \left(\Xi'_{tc}, \Xi'_{dc}, \sigma_{tc}\right) = \varphi_{tc}(\sigma_t(\Xi_{tc}); \varepsilon, \Xi_{dc}, \sigma_t) \\
&\quad \text{let } \left(\Xi''_{dc}, \Xi''_d, \sigma_{dc}\right) = \varphi_{dc}(\Xi'_{dc}; \varepsilon, \Xi'_d, \sigma_{tc}) \\
&\quad \left((\Xi_s, \Xi''_d, \Xi'_t, \Xi''_{dc}, \Xi'_{tc}), \sigma_{dc}\right)
\end{aligned}$$

Since σ_t maps only type parameters, we need to apply it only to Ξ_{tc}, while dirt constraints Ξ_{dc} can be left as is. Next, type coercion parameters do not appear anywhere else, so there is no need to apply σ_{tc}, only pass it to the next stage. Finally, even though σ_{dc} maps dirt parameters, we can assume that Ξ'_t and Ξ'_{tc} are in canonical form, so there is no need to substitute them.

Proposition 3.6. *If we have* $\Phi_{\texttt{red}}(\Xi) = (\Xi', \sigma)$*, then* $\vdash \sigma : \Xi \Rightarrow \Xi'$*.*

Proof. An unexciting step-by-step analysis of each of the three stages [Kop24, Section 6.1]. □

4. Polarity

As shown in Section 1.4, constraint simplification will not necessarily replace types with equivalent, but with compatible ones. Instead of checking the compatibility of whole types,

it is enough to check the compatibility of their free parameters. Whether a parameter can be safely increased or decreased depends on its *polarity*, which captures their impact on the whole type.

4.1. **Free parameters.**

Definition 4.1. We define a set of free type and dirt parameters F as a pair of sets (F^+, F^-) for types, dirts and typing contexts as

$$\begin{aligned} \mathtt{fp}(\alpha) &= (\{\alpha\}, \emptyset) & \mathtt{fp}(\mathtt{Unit}) &= (\emptyset, \emptyset) & \mathtt{fp}(A \to \underline{C}) &= \overline{\mathtt{fp}(A)} \cup \mathtt{fp}(\underline{C}) \\ \mathtt{fp}(\delta) &= (\{\delta\}, \emptyset) & \mathtt{fp}(\emptyset) &= (\emptyset, \emptyset) & \mathtt{fp}(\{op\} \cup \Delta) &= \mathtt{fp}(\Delta) \end{aligned}$$

$$\begin{aligned} \mathtt{fp}(A \mathbin{!} \Delta) &= \mathtt{fp}(A) \cup \mathtt{fp}(\Delta) \\ \mathtt{fp}(\varepsilon) &= (\emptyset, \emptyset) & \mathtt{fp}(\Gamma, x : A) &= \mathtt{fp}(\Gamma) \cup \mathtt{fp}(A) \end{aligned}$$

where all set operations (union, difference, subset, ...) on pairs are defined component-wise, while $\overline{(F^+, F^-)} = (F^-, F^+)$ denotes the swap of the components.

If a parameter is not present in neither F^+ nor in F^-, we call it *neutral*. If it is present in both F^+ and in F^-, we call it *bipolar*.

For example, for

$$F = \mathtt{fp}((\mathtt{Bool} \xrightarrow{\delta} \alpha) \xrightarrow{\delta'} \alpha)$$

we have $F^+ = \{\alpha, \delta'\}$ and $F^- = \{\alpha, \delta\}$.

Applying a substitution to a type changes its sets of positive and negative parameters, and the new sets can be computed directly from the original ones as

$$\sigma(F) = \bigcup_{\alpha^+ \in F^+} \mathtt{fp}(\sigma(\alpha^+)) \cup \overline{\bigcup_{\alpha^- \in F^-} \mathtt{fp}(\sigma(\alpha^-))} \cup \bigcup_{\delta^+ \in F^+} \mathtt{fp}(\sigma(\delta^+)) \cup \overline{\bigcup_{\delta^- \in F^-} \mathtt{fp}(\sigma(\delta^-))}$$

Proposition 4.2. *For an arbitrary valid substitution $\vdash \sigma : \Xi \Rightarrow \Xi'$, the following holds:*

- *if $\Xi \vdash \Gamma$ $\mathtt{tyCtx}$, then $\mathtt{fp}(\sigma(\Gamma)) = \sigma(\mathtt{fp}(\Gamma))$,*
- *if $\Xi \vdash \Gamma : A$, then $\mathtt{fp}(\sigma(A)) = \sigma(\mathtt{fp}(A))$,*
- *if $\Xi \vdash \Gamma : \underline{C}$, then $\mathtt{fp}(\sigma(\underline{C})) = \sigma(\mathtt{fp}(\underline{C}))$.*

4.2. **Coercion families.**

As discussed in Section 1.4, positive parameters can be safely decreased, and negative parameters can be safely increased. Consider two instantiations of free parameters of type $A = (\mathtt{Bool} \xrightarrow{\delta} \alpha) \xrightarrow{\delta'} \alpha$:

$$\begin{aligned} \sigma_1 &= \{\alpha \mapsto \mathtt{Int}, \delta \mapsto \{op, op'\}, \delta' \mapsto \emptyset\} \\ \sigma_2 &= \{\alpha \mapsto \mathtt{Int}, \delta \mapsto \{op\}, \delta' \mapsto \{op\}\} \end{aligned}$$

We then have

$$\sigma_1(A) = (\mathtt{Bool} \xrightarrow{\{op, op'\}} \mathtt{Int}) \to \mathtt{Int} \leq (\mathtt{Bool} \xrightarrow{\{op\}} \mathtt{Int}) \xrightarrow{\{op\}} \mathtt{Int} = \sigma_2(A)$$

witnessed by the coercion

$$(\langle \mathtt{Bool} \rangle \xrightarrow{\{op\} \cup \emptyset_{\{op'\}}} \langle \mathtt{Int} \rangle) \xrightarrow{\emptyset_{\{op\}}} \langle \mathtt{Int} \rangle$$

One can see that the structure of the coercion exactly matches the structure of the parametric type, where at leaves, we have coercions between instantiations of corresponding parameters

whose direction is determined by the polarity of parameters. Note that α is bipolar, thus it can be assigned only reflexive coercions.

We consolidate all these coercions into a *coercion family* Y, which assigns a type coercion $Y(\alpha)$ to each type parameter α, and a dirt coercion $Y(\delta)$ to each dirt parameter δ. The coercion corresponding to the above example would thus be:

$$Y = \{\alpha \mapsto \langle \texttt{Int} \rangle, \delta \mapsto (\{op\} \cup \emptyset_{\{op'\}}), \delta' \mapsto \emptyset_{\{op\}}\}$$

Definition 4.3. Take substitutions $\vdash \sigma_1 : \Xi \Rightarrow \Xi'$, $\vdash \sigma_2 : \Xi \Rightarrow \Xi'$ and a set of free parameters F. Then, a family Y as described above is *a coercion between σ_1 and σ_2 relative to F*, written as:

$$\Xi' \vdash Y : \sigma_1 \leq_F \sigma_2$$

if:

- for all $\alpha \in F^+$, we have $\Xi' \vdash Y(\alpha) : \sigma_1(\alpha) \leq \sigma_2(\alpha)$
- for all $\alpha \in F^-$, we have $\Xi' \vdash Y(\alpha) : \sigma_2(\alpha) \leq \sigma_1(\alpha)$
- for all $\delta \in F^+$, we have $\Xi' \vdash Y(\delta) : \sigma_1(\delta) \leq \sigma_2(\delta)$
- for all $\delta \in F^-$, we have $\Xi' \vdash Y(\delta) : \sigma_2(\delta) \leq \sigma_1(\delta)$

Given a coercion family Y, we can extend dirt coercions $Y(\Delta)$ to arbitrary dirts Δ as long as $\texttt{fp}(\Delta) \subseteq F$ by:

$$Y(\emptyset) = \langle \emptyset \rangle \qquad\qquad Y(\{op\} \cup \Delta) = \{op\} \cup Y(\Delta)$$

As expected, coercions on parameters give rise to a coercion on the whole dirt.

Proposition 4.4. *For any valid substitutions $\vdash \sigma_1 : \Xi \Rightarrow \Xi'$,$\vdash \sigma_2 : \Xi \Rightarrow \Xi'$, such that $\Xi' \vdash Y : \sigma_1 \leq_F \sigma_2$, and an arbitrary dirt $\Xi \vdash \Delta$ $\texttt{dirt}$ such that $\texttt{fp}(\Delta) \subseteq F$, we have*

$$\Xi' \vdash Y(\Delta) : \sigma_1(\Delta) \leq \sigma_2(\Delta)$$

Similarly, we can define a value coercion $Y(A)$ and a computation coercion $Y(\underline{C})$ for arbitrary $A, \underline{C}$ such that $\texttt{fp}(A) \subseteq F$ or $\texttt{fp}(\underline{C}) \subseteq F$:

$$Y(\texttt{Unit}) = \langle \texttt{Unit} \rangle \qquad Y(A \to \underline{C}) = Y(A) \to Y(\underline{C}) \qquad Y(A \,!\, \Delta) = Y(A) \,!\, Y(\Delta)$$

Proposition 4.5. *For any valid substitutions $\vdash \sigma_1 : \Xi \Rightarrow \Xi'$,$\vdash \sigma_2 : \Xi \Rightarrow \Xi'$, such that $\Xi' \vdash Y : \sigma_1 \leq_F \sigma_2$, for an arbitrary value type $\Xi \vdash A : S$ such that $\texttt{fp}(A) \subseteq F$, and for an arbitrary computation type $\Xi \vdash \underline{C} : S$ such that $\texttt{fp}(\underline{C}) \subseteq F$, we have*

$$\Xi' \vdash Y(A) : \sigma_1(A) \leq \sigma_2(A) \qquad \textit{and} \qquad \Xi' \vdash Y(\underline{C}) : \sigma_1(\underline{C}) \leq \sigma_2(\underline{C})$$

In addition, coercion families satisfy a number of meta-theoretical properties that will prove useful when establishing soundness of simplifications.

Proposition 4.6. *Let $\vdash \sigma_1 : \Xi \Rightarrow \Xi'$,$\vdash \sigma_2 : \Xi \Rightarrow \Xi'$ be valid substitutions and F a set of free parameters such that $\Xi' \vdash Y : \sigma_1 \leq_F \sigma_2$, then $\Xi' \vdash Y : \sigma_2 \leq_{\overline{F}} \sigma_1$.*

Proposition 4.7. *Take substitutions $\vdash \sigma_1 : \Xi \Rightarrow \Xi'$, $\vdash \sigma_2 : \Xi \Rightarrow \Xi'$, $\vdash \sigma_3 : \Xi \Rightarrow \Xi'$, and coercion families $\Xi' \vdash Y_1 : \sigma_1 \leq_F \sigma_2$ and $\Xi' \vdash Y_2 : \sigma_2 \leq_F \sigma_3$. Then $\Xi' \vdash (Y_2 \circ Y_1) : \sigma_1 \leq_F \sigma_3$, where $(Y_2 \circ Y_1)(\alpha) := Y_2(\alpha) \circ Y_1(\alpha)$ and $(Y_2 \circ Y_1)(\delta) := Y_2(\delta) \circ Y_1(\delta)$.*

Proof. Take an arbitrary positive type parameter $\alpha \in F^+$. We immediately get coercions $\Xi' \vdash Y_1(\alpha) : \sigma_1(\alpha) \leq \sigma_2(\alpha)$ and $\Xi' \vdash Y_2(\alpha) : \sigma_2(\alpha) \leq \sigma_3(\alpha)$, thus $\Xi' \vdash Y_2(\alpha) \circ Y_1(\alpha) : \sigma_1(\alpha) \leq \sigma_3(\alpha)$ as required. The proof for positive dirt parameters, and for negative type and dirt parameters is analogous. □

Proposition 4.8. *Take substitutions* $\vdash \sigma : \Xi_1 \Rightarrow \Xi_2$, $\vdash \sigma_1 : \Xi_2 \Rightarrow \Xi'$, $\vdash \sigma_2 : \Xi_2 \Rightarrow \Xi'$, *and a coercion family* $\Xi' \vdash Y : \sigma_1 \leq_{\sigma(F)} \sigma_2$, *then* $\Xi' \vdash Y \circ \sigma : (\sigma_1 \circ \sigma) \leq_F (\sigma_2 \circ \sigma)$, *where* $(Y \circ \sigma)(\alpha) := Y(\sigma(\alpha))$ *and* $(Y \circ \sigma)(\delta) := Y(\sigma(\delta))$.

Proof. Take a positive type parameter $\alpha \in F^+$. We have $\mathtt{fp}(\sigma(\alpha)) \subseteq \sigma(F)$ by definition of $\sigma(F)$, thus from Proposition 4.5, we get $\Xi' \vdash Y(\sigma(\alpha)) : \sigma_1(\sigma(\alpha)) \leq_F \sigma_2(\sigma(\alpha))$ or equivalently $\Xi' \vdash Y(\sigma(\alpha)) : (\sigma_1 \circ \sigma)(\alpha) \leq_F (\sigma_2 \circ \sigma)(\alpha)$. The proof for other kinds of parameters is analogous. □

5. Simplifications

Finally, we have all the ingredients required for our constraint simplification algorithm. We split the algorithm into multiple independent phases focusing on specific patterns often present in the inferred constraints. In Section 5.1, we shall discuss general properties of such phases, and afterwards examine their particular instances.

As we have seen in Section 3.2, arbitrary contexts can be transformed into equivalent ones that are representable as graphs, with parameters as vertices and coercions as edges. We continue this natural direction in Section 5.2 by removing loops and parallel edges to obtain simple graphs, and in Section 5.3 by contracting strongly connected components to obtain directed acyclic graphs (DAGs). Finally, we end up with two heuristics that significantly simplify constraint graphs that occur in practice: in Section 5.4 we attempt to remove as much of redundant intermediate type constraints as possible, and in Section 5.5 we do the same with dirt parameters.

5.1. Simplification phases.

A *phase* Φ takes as inputs a parameter context Ξ and a set of free parameters F, denoting the polarity of parameters in the type being simplified, and produces a simplified parameter context Ξ' together with a substitution $\vdash \sigma : \Xi \Rightarrow \Xi'$ that can be used to replace the removed parameters.

We shall compare contexts in terms of all their possible *instantiations* in some other context, say Ξ_{use}, which we fix throughout the rest of the paper. Take a polymorphic value v with parameters in context Ξ. Whenever this value is used in Ξ_{use}, we need to instantiate all its parameters to their counterparts (types, coercions, etc.) that need to be valid in Ξ_{use}. Instantiating a polymorphic value thus amounts to a substitution $\vdash \eta : \Xi \Rightarrow \Xi_{\text{use}}$, which we shall label with η to highlight its role.

From Proposition 3.5, it immediately follows that any instantiation $\vdash \eta' : \Xi' \Rightarrow \Xi_{\text{use}}$ of the simplified context also produces an instantiation $\vdash \eta' \circ \sigma : \Xi \Rightarrow \Xi_{\text{use}}$ of the original context. The converse does not hold: not every instantiation of the original context can be reconstructed from one of the simplified context. This is for the simple reason that simplifications remove parameters and hence decrease the degrees of freedom. However, we can recover the lost freedom using subtyping.

Definition 5.1. Phase Φ is *complete* if for any $\Phi(\Xi, F) = (\Xi', \sigma)$ and any instantiation $\vdash \eta : \Xi \Rightarrow \Xi_{\text{use}}$, there exists an instantiation $\vdash \eta' : \Xi' \Rightarrow \Xi_{\text{use}}$ and a coercion family $\Xi_{\text{use}} \vdash Y_\eta : (\eta' \circ \sigma) \leq_F \eta$.

Definition 5.2. Let Φ_1 and Φ_2 be two phases. The *composition of phases* $\Phi_2 \circ \Phi_1$ is defined as follows:

$$(\Phi_2 \circ \Phi_1)(\Xi, F) = (\Xi'', \sigma_2 \circ \sigma_1) \text{ where } \Phi_1(\Xi, F) = \big(\Xi', \sigma_1\big) \text{ and } \Phi_2(\Xi', \sigma_1(F)) = \big(\Xi'', \sigma_2\big)$$

Proposition 3.5 implies that the composition of two phases is also a phase.

Proposition 5.3. *Let Φ_1 and Φ_2 be two phases. If Φ_1 and Φ_2 are complete, then $\Phi_2 \circ \Phi_1$ is also complete.*

Proof. Let Φ_1 and Φ_2 be two complete phases and $\vdash \eta : \Xi \Rightarrow \Xi_{\text{use}}$ be a valid instantiation. We need to show that there exists an instantiation $\vdash \eta'' : \Xi'' \Rightarrow \Xi_{\text{use}}$ and a coercion family $\Xi_{\text{use}} \vdash Y : (\eta'' \circ \sigma_2 \circ \sigma_1) \leq_F \eta$.

Since Φ_1 is complete, there exists $\vdash \eta' : \Xi' \Rightarrow \Xi_{\text{use}}$ and $\Xi_{\text{use}} \vdash Y_\eta : (\eta' \circ \sigma_1) \leq_F \eta$. Furthermore, by completeness of Φ_2, there exists an instantiation $\vdash \eta'' : \Xi'' \Rightarrow \Xi_{\text{use}}$ and a coercion family $\Xi_{\text{use}} \vdash Y_{\eta'} : (\eta'' \circ \sigma_2) \leq_{\sigma_1(F)} \eta'$. Proposition 4.8 implies that we have $\Xi_{\text{use}} \vdash Y_{\eta'} \circ \sigma_1 : (\eta'' \circ \sigma_2 \circ \sigma_1) \leq_F (\eta' \circ \sigma_1)$. Finally, Proposition 4.7 implies that, as required, $\Xi_{\text{use}} \vdash (Y_{\eta'} \circ \sigma_1 \circ Y_\eta) : (\eta'' \circ \sigma_2 \circ \sigma_1) \leq_F \eta$. □

5.2. **Removing loops and parallel edges.** An observant reader might have noticed that our graphs might have loops if the context contains coercion parameters of the form $\omega : \alpha \leq \alpha$ or $\varpi : \delta \leq \mathcal{O} \cup \delta$. However, these are easily disposed by a strengthening

$$\sigma_l = \{\omega \mapsto \langle\alpha\rangle \mid (\omega : \alpha \leq \alpha) \in \Xi\} \circ \{\varpi \mapsto \langle\delta\rangle \cup^+ \mathcal{O} \mid (\varpi : \delta \leq \mathcal{O} \cup \delta) \in \Xi\}$$

which is also the reason behind including $\cup^+$ amongst our coercion constructors. Note that the sink node cannot have loops as coercions $\varpi : \mathcal{O}_1 \leq \mathcal{O}_2$ are already removed during context reduction.

Furthermore, our graph may have multiple parallel edges. Let us first consider type coercions. If we already have $\omega : \alpha_1 \leq \alpha_2$, an additional parameter $\omega' : \alpha_1 \leq \alpha_2$ between the same two nodes offers no additional flexibility, so we can simply replace it with ω. In general, we define a strengthening

$$\sigma_t = \{\omega_{ijk} \mapsto \omega_{ij0} \mid (\omega_{ijk} : \alpha_i \leq \alpha_j) \in \Xi\}$$

where ω_{ij0} is the chosen representative of all coercions between α_i and α_j.

For dirt coercions, the situation is more interesting due to the presence of operation sets. Still, if we have both $\varpi : \delta_1 \leq \mathcal{O} \cup \delta_2$ and $\varpi' : \delta_1 \leq \mathcal{O}' \cup \delta_2$, then the upper bound for δ_1 can be refined to $(\mathcal{O} \cap \mathcal{O}') \cup \delta_2$, and $\mathcal{O} \cap \mathcal{O}'$ can be factored out of both coercions. Again, for a general context, we define

$$\sigma_d = \{\varpi_{ijk} \mapsto (\mathcal{O}_{ijk} \setminus \bigcap_l \mathcal{O}_{ijl}) \cup^+ \varpi_{ij} \mid (\varpi_{ijk} : \delta_i \leq \mathcal{O}_{ijk} \cup \delta_j) \in \Xi\}$$

for some fresh $\varpi_{ij} : \delta_i \leq (\bigcap_l \mathcal{O}_{ijl}) \cup \delta_j$ replacing all coercions between δ_i and δ_j. We can apply a similar approach for the sink node and define

$$\sigma_s = \{\varpi_{ij} \mapsto (\mathcal{O}_{ij} \setminus \bigcap_k \mathcal{O}_{ik}) \cup^+ \varpi_i \mid (\varpi_{ij} : \delta_i \leq \mathcal{O}_{ij}) \in \Xi\}$$

coalescing all edges $\varpi_{ij} : \delta_i \leq \mathcal{O}_{ij}$ between δ_i and the sink node into a fresh $\varpi_i : \delta_i \leq \bigcap_k \mathcal{O}_{ik}$.

Even though σ_d and σ_s are not strengthenings due to the introduction of additional dirt coercions, it is easy to check that

$$\Phi_{\texttt{loopPar}}(\Xi, \_) = ((\sigma_s \circ \sigma_d \circ \sigma_t \circ \sigma_l)(\Xi), (\sigma_s \circ \sigma_d \circ \sigma_t \circ \sigma_l))$$

is a valid and complete simplification phase.

5.3. **Contracting strongly connected components.** Having obtained a simple directed graph, we can simplify it even further.

Firstly, we can easily see that any strongly connected component (which in practice occurs for recursive functions) in a graph requires all the type parameters to be equal. In terms of coercions, if Ξ contains type coercions

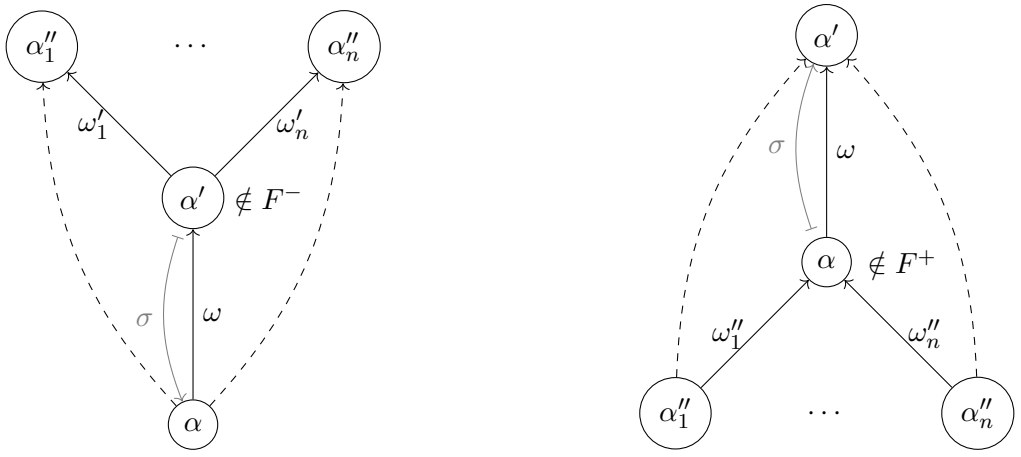
$$\omega_1 : \alpha_1 \leq \alpha_2, \omega_2 : \alpha_2 \leq \alpha_3, \ldots \omega_n : \alpha_n \leq \alpha_1$$

we can unify all type parameters with a substitution $\{\alpha_i \mapsto \alpha_1\}_{i=2}^{k}$. If we wish, we can also replace all ω_i with $\langle \alpha_1 \rangle$, or we can simply weed them out by re-running $\Phi_{\texttt{loopPar}}$.

The same can be done for cycles in dirt coercions in case no additional operations $\mathcal{O}$ are present, which is often the case in higher-order functions.

We skip writing out the formal definition of the substitution σ as it would offer way more verbosity than insight. Still, σ is a strengthening, and we can again define a complete phase $\Phi_{\texttt{scc}}(\Xi, \_) = (\sigma(\Xi), \sigma)$, after which we end with a directed acyclic graph (DAG).

5.4. **Compressing bridges.** Secondly, intermediate type parameters $\omega : \alpha \leq \alpha'$ can be merged if no other type parameters are dependent on their specific relationship, in particular if the edge is a *bridge*, i.e. the only incoming edge of α', or the only outgoing edge of α (Figure 8).



(A) Exactly one incoming coercion

(B) Exactly one outgoing coercion

Figure 8. Two cases for type parameter contraction

First, consider the case when $\omega : \alpha \leq \alpha'$ is the only incoming edge of α'. In that case, we can safely replace α' with α as the only affected constraints are ones of the form $\alpha' \leq \alpha''_i$,

which continue to hold in the new form $\alpha \leq \alpha_i''$ due to transitivity. To ensure completeness, we must furthermore ensure that α' is not negative.

Formally, if α' has only one incoming edge and $\alpha' \notin F^-$, we can apply the phase

$$\Phi_{\texttt{bridgeIn}}(\Xi, F) = (\sigma(\Xi), \sigma) \text{ where } \sigma = \{\alpha' \mapsto \alpha\}$$

Conversely, if α has only outgoing edge and $\alpha \notin F^+$, we can apply the phase

$$\Phi_{\texttt{bridgeOut}}(\Xi, F) = (\sigma(\Xi), \sigma) \text{ where } \sigma = \{\alpha \mapsto \alpha'\}$$

In both cases, the substitution σ is a strengthening, thus we get valid simplification phases.

Proposition 5.4. *Simplification phases* $\Phi_{\texttt{bridgeIn}}$ *and* $\Phi_{\texttt{bridgeOut}}$ *are complete.*

Proof. Since the cases are symmetric, we can consider only $\Phi_{\texttt{bridgeIn}}$. Take an arbitrary $\vdash \eta : \Xi \Rightarrow \Xi_{\text{use}}$ and let us define $\vdash \eta' : \sigma(\Xi) \Rightarrow \Xi_{\text{use}}$

Since σ affects only type and type coercion parameters, we define $\eta'(\varsigma) = \eta(\varsigma)$ for all skeleton parameters ς, and similar for dirt parameters δ and dirt coercion parameters ϖ. For all type parameters $\alpha'' \neq \alpha'$ (including α) we similarly define $\eta'(\alpha'') = \eta(\alpha'')$, while on α', we leave η' undefined.

Finally, the only affected type coercion parameters are ones that contained α'. By assumption, there is only one where α' appears on the right-hand side: $\omega : \alpha \leq \alpha'$, and this maps to $(\omega : \alpha \leq \alpha) \in \sigma(\Xi)$, which we satisfy by defining $\eta'(\omega) = \langle \eta'(\alpha) \rangle$. All other coercion parameters have α' on the left, so are of the form $\omega_i' : \alpha' \leq \alpha_i''$ for some α_i'', and map to $(\omega_i' : \alpha \leq \alpha_i'') \in \sigma(\Xi)$. For those, we define $\eta'(\omega_i') = \eta(\omega_i') \circ \eta(\omega)$. As we had $\vdash \eta(\omega) : \eta(\alpha) \leq \eta(\alpha')$ and $\vdash \eta(\omega_i') : \eta(\alpha') \leq \eta(\alpha_i'')$, and as $\eta'(\alpha) = \eta(\alpha)$ and $\eta'(\alpha_i'') = \eta(\alpha_i'')$, we indeed have $\vdash \eta'(\omega_i') : \eta'(\alpha) \leq \eta'(\alpha_i'')$.

Having checked that $\vdash \eta' : \sigma(\Xi) \Rightarrow \Xi_{\text{use}}$ let us find a coercion family Y such that $\Xi_{\text{use}} \vdash Y : (\eta' \circ \sigma) \leq_F \eta$. The only parameter we need to check is α', as on all others, instantiations $\eta' \circ \sigma$ and η are the same. By assumption, we have $\alpha' \notin F^-$, so we only need to consider the case $\alpha' \in F^+$. However, $\eta'(\sigma(\alpha')) = \eta'(\alpha) = \eta(\alpha)$, thus $\vdash \eta(\omega) : \eta(\alpha) \leq \eta(\alpha')$ is the required coercion $\vdash Y(\alpha') : \eta'(\sigma(\alpha')) \leq \eta(\alpha')$. □

If there are multiple bridges, we can repeat the process, though we must make keep in mind that the polarity of the remaining parameter is the union of both polarities, so the polarity condition has to be re-evaluated after each compression. As before, the same compression can be done for bridges in dirt coercions as long as the edge has no additional operations $\mathcal{O}$ on the right-hand side.

5.5. **Minimizing operation sets.** Recall that dirt constraint parameters in a typing context Ξ are of the form $\varpi : \delta \leq \delta' \cup \mathcal{O}$ or $\varpi : \delta \leq \mathcal{O}$. We have already seen that we can apply $\Phi_{\texttt{scc}}$, $\Phi_{\texttt{bridgeIn}}$, and $\Phi_{\texttt{bridgeOut}}$ in case the sets of operations $\mathcal{O}$ are empty, for example with higher-order functions that are effectful without mentioning any particular operation *op*.

In contrast to types, dirts have the least element $\emptyset$, which is often desirable as it signifies lack of effects, leading to efficient compilation [KKPS21]. Furthermore, any dirt parameter appearing only on the left-hand side of dirt inequalities can be safely set to $\emptyset$, as long as it is not negative. Since setting $\delta \mapsto \emptyset$ trivializes inequalities of the form $\varpi : \delta \leq \delta' \cup \mathcal{O}$ where δ' appeared on the right-hand side, this can cause a chain-reaction of further removals.

More generally, we can simultaneously remove any set of non-negative dirt parameters $\mathcal{D} = \{\delta \mid \delta \in \Xi, \delta \notin F^-\}$ with no incoming edges, i.e. no coercion parameters $\varpi : \delta' \leq \delta \cup \mathcal{O}$ with $\delta \notin \mathcal{D}$. For such a set $\mathcal{D}$, we define a substitution σ by $\sigma(\delta) = \emptyset$ for each $\delta \in \Xi$, and by

$\sigma(\varpi) = \emptyset_{\sigma(\Delta)}$ for each $(\varpi : \delta \leq \Delta) \in \mathcal{D}$ where $\delta \in \mathcal{D}$. As σ is a strengthening, we can define a simplification phase

$$\Phi_{\texttt{emptyDirt}}(\Xi, F) = (\sigma(\Xi), \sigma)$$

Proposition 5.5. $\Phi_{\texttt{emptyDirt}}$ *is complete.*

Proof. Take an arbitrary $\vdash \eta : \Xi \Rightarrow \Xi_{\text{use}}$. The required $\vdash \eta' : \sigma(\Xi) \Rightarrow \Xi_{\text{use}}$ is thus

$$\eta' = (\eta \setminus \mathcal{D}) \setminus \{\varpi : \delta \leq \Delta \mid \delta \in \mathcal{D}\}$$

As by assumption, parameters from $\mathcal{D}$ did not appear on right-hand sides of dirt coercions, no other parameters apart from the removed ones have been impacted by the substitution. Thus, to obtain $\Xi_{\text{use}} \vdash Y : (\eta' \circ \sigma) \leq_F \eta$, we only need to check how Y is defined on $\delta \in \mathcal{D}$. By assumption, $\delta \notin F^-$, so we only need to consider the case $\delta \in F^+$. But since $(\eta' \circ \sigma)(\delta) = \eta'(\sigma(\delta)) = \emptyset$, we can define $Y(\delta)$ to be the empty coercion $\emptyset_{\eta(\delta)}$. □

Like $\Phi_{\texttt{bridgein}}$ and $\Phi_{\texttt{bridgeOut}}$, the phase $\Phi_{\texttt{dirtEmpty}}$ can be repeated iteratively until there is no set satisfying the requirements. In practice, this can be done efficiently by traversing the quotient graph of strongly connected components in topological ordering. Dually, there is a greatest dirt Σ, and we can apply a dual phase $\Phi_{\texttt{fullDirt}}$ for sets of non-positive dirt parameters with no incoming edges. However, as discussed in Section 7, this does not bring significant practical benefits, so we refrain from doing it in the implementation.

6. Denotational semantics

To show that simplification phases preserve the semantics of terms, we turn to denotational semantics.

6.1. **Free monad.**

As it is standard in the setting of algebraic effects, we are going to interpret effectful computations with the help of a *free monad*. For any assignment $\mathcal{S} = \{op_i : U_i \to V_i\}_i$, that maps an operation op_i to sets U_i and V_i, we can define a monad $T_{\mathcal{S}}$ that takes a set X to the set $T_{\mathcal{S}}X$, defined inductively as the smallest set, containing:

- $\text{in}_{\texttt{return}}(a)$ for each $a \in X$,
- $\text{in}_{op_i}(u; \kappa)$ for each $op_i : U_i \to V_i \in \mathcal{S}$, each $u \in U_i$, and each $\kappa \in V_i \to T_{\mathcal{S}}X$

The inclusion $\text{in}_{\texttt{return}}$ acts as the unit of the monad, while for a map $f : X \to T_{\mathcal{S}}Y$, its lifting $f^\dagger : T_{\mathcal{S}}X \to T_{\mathcal{S}}Y$ is defined recursively as

$$f^\dagger(\text{in}_{\texttt{return}}(a)) = f(a)$$
$$f^\dagger(\text{in}_{op}(u; \kappa)) = \text{in}_{op}(u; f^\dagger \circ \kappa)$$

6.2. **Skeletal semantics.** Since the presented simplifications preserve types only up to subtyping, we first introduce a *skeletal* semantics, which disregards effect information, giving us a unified platform on which to compare such changes.

For a parameter context Ξ, we shall take its interpretation $[\![\Xi]\!]$ to be a set of all valid assignments ξ that map parameters to their interpretations. Due to dependencies between parameters, we shall expose ξ in turn, starting with skeleton parameters $\varsigma \in \Xi$, which are mapped to arbitrary sets $\xi(\varsigma)$.

Then, for any $\xi \in [\![\Xi]\!]$, we can interpret each skeleton $\Xi \vdash S$ `skel` with a set $[\![S]\!]_\xi$, defined as:

$$[\![\varsigma]\!]_\xi = \xi(\varsigma) \qquad [\![\mathtt{Unit}]\!]_\xi = \{\star\} \qquad [\![S_1 \to S_2]\!]_\xi = [\![S_1]\!]_\xi \to T_{[\![\Sigma]\!]}([\![S_2]\!]_\xi)$$

where for a global signature $\Sigma = \{op_i : A_i \to B_i\}_i$, we define $[\![\Sigma]\!] = \{op_i : [\![S_i]\!] \to [\![S_i']\!]\}_i$, where $\vdash A_i : S_i$ and $\vdash B_i : S_i'$ are monomorphic, thus ξ does not play a role in their interpretation.

Next, for each $\Xi \vdash A : S$, we define its *skeletal semantics* $(\!|A|\!)_\xi = [\![S]\!]_\xi$ and similarly $(\!|\underline{C}|\!)_\xi = [\![S]\!]_\xi$ for $\Xi \vdash \underline{C} : S$. Since interpretations of types ignore all effect information, types with matching skeletons have the same interpretation. For this reason, we also do not define any skeletal interpretations of coercions.

Given an assignment ξ, skeletal denotational semantics of well-typed terms are defined as maps from interpretations of typing contexts to interpretations of types:

$$(\!|\Xi; \Gamma \vdash v : A|\!)_\xi : (\!|\Gamma|\!)_\xi \to (\!|A|\!)_\xi \qquad (\!|\Xi; \Gamma \vdash c : \underline{C}|\!)_\xi : (\!|\Gamma|\!)_\xi \to (\!|\underline{C}|\!)_\xi$$

where interpretation of typing contexts $(\!|\Gamma|\!)_\xi$ is defined component wise:

$$(\!|x_1 : A_1, \ldots, x_n : A_n|\!)_\xi := (\!|A_1|\!)_\xi \times \cdots \times (\!|A_n|\!)_\xi$$

Interpretations are given in the standard way [BP14]. For a tuple $(a_1, \ldots, a_n) = \vec{a} \in (\!|\Gamma|\!)_\xi$ interpretations of values are defined as:

$$\begin{aligned}
(\!|\Xi; \Gamma \vdash x_i : A_i|\!)_\xi(\vec{a}) &= a_i \\
(\!|\Xi; \Gamma \vdash \mathtt{unit} : \mathtt{Unit}|\!)_\xi(\vec{a}) &= \star \\
(\!|\Xi; \Gamma \vdash \mathtt{fun}\ (x : A) \mapsto c : A \to \underline{C}|\!)_\xi(\vec{a}) &= a' \in (\!|A|\!)_\xi \mapsto (\!|\Xi; \Gamma, x : A \vdash c : \underline{C}|\!)_\xi(\vec{a}, a') \\
(\!|\Xi; \Gamma \vdash v \rhd \gamma_{\mathtt{v}} : A'|\!)_\xi(\vec{a}) &= (\!|\Xi; \Gamma \vdash v : A|\!)_\xi(\vec{a})
\end{aligned}$$

where in the last line, we can safely discard $\Xi \vdash \gamma_{\mathtt{v}} : A \le A'$ since $(\!|A|\!)_\xi = (\!|A'|\!)_\xi$. Similarly, interpretations of computations are defined as:

$$\begin{aligned}
(\!|\Xi; \Gamma \vdash \mathtt{return}\ v : A\ !\ \Delta|\!)_\xi(\vec{a}) &= \mathrm{in}_{\mathtt{return}}((\!|\Xi; \Gamma \vdash v : A|\!)_\xi(\vec{a})) \\
(\!|\Xi; \Gamma \vdash op\ v\ (y : A_{op}.c) : \underline{C}|\!)_\xi(\vec{a}) &= \mathrm{in}_{op}\big((\!|\Xi; \Gamma \vdash v : A_{op}|\!)_\xi(\vec{a}); \\
&\qquad b \in (\!|B_{op}|\!)_\xi \mapsto (\!|\Xi; \Gamma, y : B_{op} \vdash c : \underline{C}|\!)_\xi(\vec{a}, b)\big) \\
(\!|\Xi; \Gamma \vdash \mathtt{do}\ x \leftarrow c_1; c_2 : A_2\ !\ \Delta|\!)_\xi(\vec{a}) &= \big(a' \in (\!|A_1|\!)_\xi \mapsto (\!|\Xi; \Gamma, x : A_1 \vdash c_2 : A_2\ !\ \Delta|\!)_\xi(\vec{a}, a')\big)^\dagger \\
&\qquad ((\!|\Xi; \Gamma \vdash c_1 : A_1\ !\ \Delta|\!)_\xi(\vec{a})) \\
(\!|\Xi; \Gamma \vdash v_1\ v_2 : \underline{C}|\!)_\xi(\vec{a}) &= ((\!|\Xi; \Gamma \vdash v_1 : A \to \underline{C}|\!)_\xi(\vec{a}))((\!|\Xi; \Gamma \vdash v_2 : A|\!)_\xi(\vec{a})) \\
(\!|\Xi; \Gamma \vdash c \rhd \gamma_{\mathtt{c}} : \underline{C}'|\!)_\xi(\vec{a}) &= (\!|\Xi; \Gamma \vdash c : \underline{C}|\!)_\xi(\vec{a})
\end{aligned}$$

where again in the last line, we again discard the coercion $\Xi \vdash \gamma_{\mathtt{c}} : \underline{C} \le \underline{C}'$ since $(\!|\underline{C}|\!)_\xi = (\!|\underline{C}'|\!)_\xi$.

6.3. **Effectful semantics.** We refine the skeletal semantics into a more precise *effectful* semantics, which also reflects the operations, tracked by the effect system.

To do so, an assignment $\xi \in [\![\Xi]\!]$ maps each dirt parameter $\delta \in \Xi$ to some set of operations $\xi(\delta)$, all of which must appear in the global signature Σ. Then, each dirt $\Xi \vdash \Delta$ `dirt` can similarly be interpreted with a set of operations $[\![\Delta]\!]_\xi$, given by:

$$[\![\delta]\!]_\xi = \xi(\delta) \qquad [\![\emptyset]\!]_\xi = \emptyset \qquad [\![\{op\} \cup \Delta]\!]_\xi = \{op\} \cup [\![\Delta]\!]_\xi$$

Interpreting dirt allows us to also interpret value and computation types. For each value type $\Xi \vdash A : S$ and each computation type $\Xi \vdash \underline{C} : S$ we define sets $[\![A]\!]_\xi$ and $[\![\underline{C}]\!]_\xi$ together with injections into the skeletal semantics. To interpret types, any assignment $\xi \in [\![\Xi]\!]$ must map each $(\alpha : S) \in \Xi$ to some set $\xi(\alpha)$ together with an injection $\iota_{[\![\alpha]\!]_\xi} : \xi(\alpha) \hookrightarrow [\![S]\!]_\xi$. Then, interpretations are extended to arbitrary value and computation types by:

$$\begin{aligned}
[\![\texttt{Unit}]\!]_\xi &= \{\star\} \\
[\![A \mathbin{!} \Delta]\!]_\xi &= T_{[\![\Sigma]\!]|_{[\![\Delta]\!]_\xi}}([\![A]\!]_\xi) \\
[\![A \to \underline{C}]\!]_\xi &= \left\{ (f, f') : (\!|A|\!)_\xi \to (\!|\underline{C}|\!)_\xi) \times ([\![A]\!]_\xi \to [\![\underline{C}]\!]_\xi) \;\middle|\; f \circ \iota_{[\![A]\!]_\xi} = \iota_{[\![\underline{C}]\!]_\xi} \circ f' \right\}
\end{aligned}$$

where $[\![\Sigma]\!]|_{[\![\Delta]\!]_\xi}$ is the restriction of $[\![\Sigma]\!]$ to operations from $[\![\Delta]\!]_\xi$, ensuring that any valid computation can trigger only operations listed in Δ.

The injections $\iota_{[\![A]\!]_\xi} : [\![A]\!]_\xi \hookrightarrow (\!|A|\!)_\xi$ and $\iota_{[\![\underline{C}]\!]_\xi} : [\![\underline{C}]\!]_\xi \hookrightarrow (\!|\underline{C}|\!)_\xi$ are defined recursively by:

$$\begin{aligned}
\iota_{[\![\texttt{Unit}]\!]_\xi}(\star) &= \star \\
\iota_{[\![A \to \underline{C}]\!]_\xi}(f, f') &= f \\
\iota_{[\![A \mathbin{!} \Delta]\!]_\xi}(\mathrm{in}_{\texttt{return}}(a)) &= \mathrm{in}_{\texttt{return}}(a) \\
\iota_{[\![A \mathbin{!} \Delta]\!]_\xi}(\mathrm{in}_{op}(u; \kappa)) &= \mathrm{in}_{op}(\iota_{[\![A_i]\!]_\xi}(u); \iota_{[\![A \mathbin{!} \Delta]\!]_\xi} \circ \kappa)
\end{aligned}$$

where we crucially use the fact that B_i is monomorphic and first-order, and so $(\!|B_i|\!)_\xi = [\![B_i]\!]_\xi$.

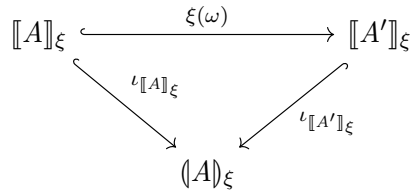
More interestingly, functions are interpreted with pairs of functions, the first acting on skeletal and the second on effectful semantics. Interpreting values of $A \to \underline{C}$ with functions $[\![A]\!]_\xi \to [\![\underline{C}]\!]_\xi$ is not sufficient because there is no simple way to extend their domain from $[\![A]\!]_\xi$ to $(\!|A|\!)_\xi$ in order to include them in $(\!|A|\!)_\xi \to (\!|\underline{C}|\!)_\xi$. Even though $\iota_{[\![A \to \underline{C}]\!]_\xi}$ projects on the first component, it is still injective. Indeed, if $\iota_{[\![A \to \underline{C}]\!]_\xi}(f_1, f_1') = \iota_{[\![A \to \underline{C}]\!]_\xi}(f_2, f_2')$, we first get $f_1 = f_2$ and furthermore $\iota_{[\![\underline{C}]\!]_\xi} \circ f_1' = \iota_{[\![\underline{C}]\!]_\xi} \circ f_2'$ which implies $f_1' = f_2'$ since $\iota_{[\![\underline{C}]\!]_\xi}$ is injective.

Next, we turn to previously ignored coercions. As dirt is interpreted with sets of operations, dirt coercions are interpreted as injections between them. Thus, an assignment $\xi \in [\![\Xi]\!]$ maps each parameter $(\varpi : \Delta \leq \Delta') \in \Xi$ to an injection $\xi(\varpi) : [\![\Delta]\!]_\xi \hookrightarrow [\![\Delta']\!]_\xi$. Then, a dirt coercion $\Xi \vdash \gamma_{\mathtt{d}} : \Delta \leq \Delta'$, can be interpreted with an injection $[\![\gamma_{\mathtt{d}}]\!]_\xi : [\![\Delta]\!]_\xi \hookrightarrow [\![\Delta']\!]_\xi$, defined recursively in the expected way.

Similarly, we use injections to interpret value and computation type coercions. For the fifth and final kind of parameters, an assignment $\xi \in [\![\Xi]\!]$ maps type coercion parameters $(\omega : A \leq A') \in \Xi$ to injections $\xi(\omega) : [\![A]\!]_\xi \hookrightarrow [\![A']\!]_\xi$. In addition, we require these injections to commute with injections into the shared skeletal semantics:

$$\iota_{[\![A]\!]_\xi} = \iota_{[\![A']\!]_\xi} \circ \xi(\omega)$$

which can be summed up with the following commutative diagram



Then, we can interpret arbitrary value and computation type coercions with injections

$$[\![\Xi \vdash \gamma_{\mathtt{v}} : A \leq A']\!]_\xi : [\![A]\!]_\xi \hookrightarrow [\![A']\!]_\xi \qquad [\![\Xi \vdash \gamma_{\mathtt{c}} : \underline{C} \leq \underline{C}']\!]_\xi : [\![\underline{C}]\!]_\xi \hookrightarrow [\![\underline{C}']\!]_\xi$$

that again commute with injections into the skeletal semantics:

$$\iota_{[\![A]\!]_\xi} = \iota_{[\![A']\!]_\xi} \circ [\![\gamma_{\mathtt{v}}]\!]_\xi \qquad \iota_{[\![\underline{C}]\!]_\xi} = \iota_{[\![\underline{C}']\!]_\xi} \circ [\![\gamma_{\mathtt{c}}]\!]_\xi$$

The interpretations are defined as follows:

$$\begin{aligned}
[\![\langle \mathtt{Unit} \rangle]\!]_\xi(\star) &= \star \\
[\![\gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}}]\!]_\xi((f, f')) &= (f, [\![\gamma_{\mathtt{c}}]\!]_\xi \circ f' \circ [\![\gamma_{\mathtt{v}}]\!]_\xi) \\
[\![\gamma_{\mathtt{v}} \mathbin{!} \gamma_{\mathtt{d}}]\!]_\xi(\mathrm{in}_{\mathtt{return}}(a)) &= \mathrm{in}_{\mathtt{return}}([\![\gamma_{\mathtt{v}}]\!]_\xi(a)) \\
[\![\gamma_{\mathtt{v}} \mathbin{!} \gamma_{\mathtt{d}}]\!]_\xi(\mathrm{in}_{op}(u; k)) &= \mathrm{in}_{[\![\gamma_{\mathtt{d}}]\!]_\xi(op)}(u; [\![\gamma_{\mathtt{v}} \mathbin{!} \gamma_{\mathtt{d}}]\!]_\xi \circ k)
\end{aligned}$$

The only non-trivial requirement is checking that the function coercion is valid. Take injections $[\![\gamma_{\mathtt{v}}]\!]_\xi : [\![A']\!]_\xi \hookrightarrow [\![A]\!]_\xi$ and $[\![\gamma_{\mathtt{c}}]\!]_\xi : [\![\underline{C}]\!]_\xi \hookrightarrow [\![\underline{C}']\!]_\xi$, and take an arbitrary $(f, f') \in [\![A \to \underline{C}]\!]_\xi$, i.e. $f \circ \iota_{[\![A]\!]_\xi} = \iota_{[\![\underline{C}]\!]_\xi} \circ f'$. Let us first show that $[\![\gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}}]\!]_\xi(f, f')$ lies in $[\![A' \to \underline{C}']\!]_\xi$, which follows from

$$f \circ \iota_{[\![A']\!]_\xi} = f \circ \iota_{[\![A]\!]_\xi} \circ [\![\gamma_{\mathtt{v}}]\!]_\xi = \iota_{[\![\underline{C}]\!]_\xi} \circ f' \circ [\![\gamma_{\mathtt{v}}]\!]_\xi = \iota_{[\![\underline{C}']\!]_\xi} \circ [\![\gamma_{\mathtt{c}}]\!]_\xi \circ f' \circ [\![\gamma_{\mathtt{v}}]\!]_\xi$$

Showing that $\iota_{[\![A' \to \underline{C}']\!]_\xi} \circ [\![\gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}}]\!]_\xi = \iota_{[\![A \to \underline{C}]\!]_\xi}$ is trivial since the injections just project the first component, while $[\![\gamma_{\mathtt{v}} \to \gamma_{\mathtt{c}}]\!]_\xi$ preserves it. As coercions commute with injections into skeletons, their interpretation is uniquely determined.

Proposition 6.1. *For an arbitrary $\xi \in [\![\Xi]\!]$, the following holds:*

- *If $\Xi \vdash \gamma_{\mathtt{d}} : \Delta_1 \leq \Delta_2$ and $\Xi \vdash \gamma'_{\mathtt{d}} : \Delta_1 \leq \Delta_2$, then $[\![\gamma_{\mathtt{d}}]\!]_\xi = [\![\gamma'_{\mathtt{d}}]\!]_\xi$.*
- *If $\Xi \vdash \gamma_{\mathtt{v}} : A_1 \leq A_2$ and $\Xi \vdash \gamma'_{\mathtt{v}} : A_1 \leq A_2$, then $[\![\gamma_{\mathtt{v}}]\!]_\xi = [\![\gamma'_{\mathtt{v}}]\!]_\xi$.*
- *If $\Xi \vdash \gamma_{\mathtt{c}} : \underline{C}_1 \leq \underline{C}_2$ and $\Xi \vdash \gamma'_{\mathtt{c}} : \underline{C}_1 \leq \underline{C}_2$, then $[\![\gamma_{\mathtt{c}}]\!]_\xi = [\![\gamma'_{\mathtt{c}}]\!]_\xi$.*

Interpretation for typing contexts is again defined component wise.

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!]_\xi := [\![A_1]\!]_\xi \times \cdots \times [\![A_n]\!]_\xi$$

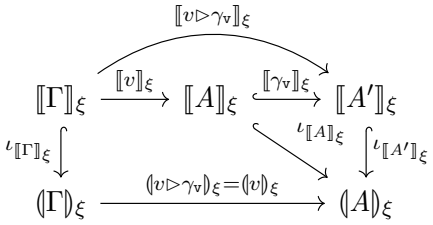
Applying injections for each component also gives us $\iota_{[\![\Gamma]\!]_\xi} : [\![\Gamma]\!]_\xi \hookrightarrow (\!|\Gamma|\!)_\xi$.

Theorem 6.2. *For any $\Xi; \Gamma \vdash v : A$ or $\Xi; \Gamma \vdash c : \underline{C}$, and for any $\xi \in [\![\Xi]\!]$, there exist unique maps $[\![\Xi; \Gamma \vdash v : A]\!]_\xi : [\![\Gamma]\!]_\xi \to [\![A]\!]_\xi$ or $[\![\Xi; \Gamma \vdash c : \underline{C}]\!]_\xi : [\![\Gamma]\!]_\xi \to [\![\underline{C}]\!]_\xi$ such that the following diagrams commute:*

$$\begin{array}{ccc}
[\![\Gamma]\!]_\xi & \xrightarrow{[\![v]\!]_\xi} & [\![A]\!]_\xi \\
{\scriptstyle \iota_{[\![\Gamma]\!]_\xi}}\downarrow & & \downarrow{\scriptstyle \iota_{[\![A]\!]_\xi}} \\
(\!|\Gamma|\!)_\xi & \xrightarrow{(\!|v|\!)_\xi} & (\!|A|\!)_\xi
\end{array} \qquad \begin{array}{ccc}
[\![\Gamma]\!]_\xi & \xrightarrow{[\![c]\!]_\xi} & [\![\underline{C}]\!]_\xi \\
{\scriptstyle \iota_{[\![\Gamma]\!]_\xi}}\downarrow & & \downarrow{\scriptstyle \iota_{[\![\underline{C}]\!]_\xi}} \\
(\!|\Gamma|\!)_\xi & \xrightarrow{(\!|c|\!)_\xi} & (\!|\underline{C}|\!)_\xi
\end{array}$$

Proof. Uniqueness follows from the fact that the connecting mappings are injections. The existence is proven by a routine induction on the typing derivation.

For example, take the case for $\Xi; \Gamma \vdash v \rhd \gamma_{\mathtt{v}} : A'$, where $\Xi; \Gamma \vdash v : A$ and $\Xi \vdash \gamma_{\mathtt{v}} : A \leq A'$. Then, we see that defining $[\![\Xi; \Gamma \vdash v \rhd \gamma_{\mathtt{v}} : A']\!]_\xi = [\![\Xi \vdash \gamma_{\mathtt{v}} : A \leq A']\!]_\xi \circ [\![\Xi; \Gamma \vdash v : A]\!]_\xi$ makes the following diagram commute, where the left square commutes by induction hypothesis, and the right triangle commutes by definition of $[\![\gamma_{\mathtt{v}}]\!]_\xi$.



□

Recall that apart from Theorem 3.4, which shows that valid substitutions preserve typing judgements, Theorem 6.2 is the only requirement we require from our terms. Both being very natural conditions gives us strong confidence that results can be applied to a number of language extensions.

6.4. **Preservation.** We now turn to the crucial soundness result: a coercion family between two substitutions gives us injections between interpretations of substituted terms.

Theorem 6.3. *Take substitutions $\vdash \sigma_1 : \Xi \Rightarrow \Xi'$ and $\vdash \sigma_2 : \Xi \Rightarrow \Xi'$ together with a coercion family $\Xi' \vdash Y : \sigma_1 \leq_F \sigma_2$. Then, for any $\Xi; \Gamma \vdash v : A$ such that $\mathtt{fp}(A) \cup \overline{\mathtt{fp}(\Gamma)} \subseteq F$, and any $\xi \in [\![\Xi']\!]$, we have*

$$[\![\Xi'; \sigma_1(\Gamma) \vdash \sigma_1(v) : \sigma_1(A)]\!]_\xi = [\![Y(A)]\!]_\xi \circ [\![\Xi'; \sigma_2(\Gamma) \vdash \sigma_2(v) : \sigma_2(A)]\!]_\xi \circ [\![Y(\Gamma)]\!]_\xi$$

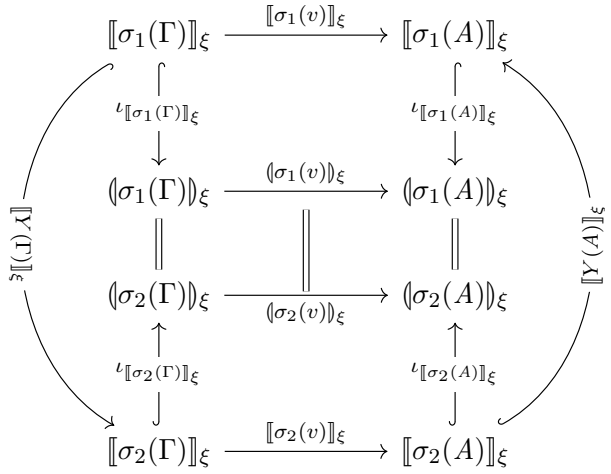
Similarly, for any $\Xi; \Gamma \vdash c : \underline{C}$ such that $\mathtt{fp}(\underline{C}) \cup \overline{\mathtt{fp}(\Gamma)} \subseteq F$, we have

$$[\![\Xi'; \sigma_1(\Gamma) \vdash \sigma_1(c) : \sigma_1(\underline{C})]\!]_\xi = [\![Y(\underline{C})]\!]_\xi \circ [\![\Xi'; \sigma_2(\Gamma) \vdash \sigma_2(c) : \sigma_2(\underline{C})]\!]_\xi \circ [\![Y(\Gamma)]\!]_\xi$$

where $Y(\Gamma)$ is defined component-wise as:

$$Y(x_1 : A_1, \ldots, x_n : A_n) := Y(A_1) \times \cdots \times Y(A_n)$$

Proof. Let us consider only the case for values, as the one for computations proceeds analogously. The proof is nicely summed up with the commutative diagram:



Here, the top and bottom square commute due to the definition of effectful semantics, the middle square commutes because the skeletal semantics is identical, and the side triangles commute because Y produces well-typed coercions, which commute with injections into skeletons. We conclude by noticing that conclusion follows from the fact that $\iota_{[\![\sigma_1(A)]\!]_\xi}$ is an injection and can be cancelled from the left. □

From this, it immediately follows that complete phases preserve the semantics up to a coercion. In particular, every instantiation of a polymorphic value v, the scenario we are interested in, can be replaced with a suitably coerced instantiation of a simplified value $\sigma(v)$.

Corollary 6.4. *Let $\Xi; \cdot \vdash v : A$ be a well-typed closed value, Φ a complete phase such that $\Phi(\Xi, \mathtt{fp}(A)) = (\Xi', \sigma)$. Then, for any instantiation $\vdash \eta : \Xi \Rightarrow \Xi_{\text{use}}$, there exists an instantiation $\vdash \eta' : \Xi' \Rightarrow \Xi_{\text{use}}$ and a coercion $\Xi_{\text{use}} \vdash \gamma_{\mathtt{v}} : \eta'(\sigma(A)) \leq \eta(A)$ such that*

$$[\![\Xi_{\text{use}}; \cdot \vdash \eta(v) : \eta(A)]\!]_\xi = [\![\Xi_{\text{use}}; \cdot \vdash \eta'(\sigma(v)) \rhd \gamma_{\mathtt{v}} : \eta(A)]\!]_\xi$$

for any assignment ξ of parameters in Ξ_{use}.

Proof. Since Φ is complete, we get $\Xi_{\text{use}} \vdash Y : (\eta' \circ \sigma) \leq \eta$, and $\Xi_{\text{use}} \vdash Y(A) : \eta'(\sigma(A)) \leq \eta(A)$ is the required coercion $\gamma_{\mathtt{v}}$. From Theorem 6.3, we get

$$\begin{aligned}[\![\Xi_{\text{use}}; \cdot \vdash \eta(v) : \eta(A)]\!]_\xi &= [\![Y(A)]\!]_\xi \circ [\![\Xi_{\text{use}}; \cdot \vdash \eta'(\sigma(v)) : \eta'(\sigma(A))]\!]_\xi \\ &= [\![\Xi_{\text{use}}; \cdot \vdash \eta'(\sigma(v)) \rhd Y(A) : \eta(A)]\!]_\xi\end{aligned}$$

with the second equation being exactly the one as in the proof of Theorem 6.2. □

7. Implementation

7.1. **Simplification pipeline.** We have implemented the simplification phases described in Section 5 in the existing Eff compiler as a default post-processing step after top-level type inference [KPS+20]. Type inference produces explicitly typed terms in a core calculus, similar to one described in Section 2 in a given parameter context. Note that we generalize only top-level values [VJS10]. From the type of those values, we compute the polarities of

parameters, apply all simplification phases, and map the resulting substitution to both the type and the term. Only after the simplification, we generalize the term and store it for further use. Due to the Corollary 6.4, each time the original top-level value is referenced, a simplified value can be used in its place, with both the instantiation and cast automatically inferred by type inference. In practice, the casts are usually trivial and optimized away by other source-level transformations [KKPS21].

The implementation of phases is a slightly different from the outline in the Section 5. As constraints are represented with simple directed graphs, phases $\Phi_{\texttt{red}}$ and $\Phi_{\texttt{loopPar}}$ are applied implicitly. To make the implementation simpler, we do not construct substitutions, but only collect equality constraints that we pass to the existing unification engine. Also, since type and type constraint parameters belonging to different skeletons are independent of each other, we perform all the simplifications separately on smaller graphs, significantly reducing their cost.

As mentioned in Section 5.5, we do not apply the phase $\Phi_{\texttt{fullDirt}}$, that would map all dirt parameters with no outgoing edges to the whole signature Σ. First, users can define operations at later time, so Σ can grow after the simplifications have been applied. Next, while removal slightly simplifies dirt coercions, those already disappear when compiling to OCaml, so unlike $\Phi_{\texttt{emptyDirt}}$, there is no additional efficiency benefit. Finally, we have not encountered such cases in practice.

7.2. **Results.** To evaluate the true impact of our simplifications, we repeated the experiments on the benchmark suite previously used to assess the performance of our optimizing compiler [KKPS21]. This time, we compared manually monomorphised code against both unsimplified and simplified polymorphic code. The newly added simplification phases successfully reproduced the exact same code as the manually monomorphised version, except that now standard lists are used instead of custom monomorphic ones tailored to specific cases. For example, the n-queens benchmark previously used

```
type solution = SolutionNil | SolutionCons of (int * int) * solution
type solutions = SolutionsNil | SolutionsCons of solution * solutions
```

instead of `(int * int) list list`, as the use of latter would produce polymorphic functions.

The performance results, shown in Table 1, reflect this equivalence. All benchmarks were using Bechamel micro-benchmarking tool on OCaml branch `4.12.0+domains+effects`. We repeated the experiments across a range of problem sizes, but the results were consistent, so we report only the largest instance for each benchmark.

As expected, the runtimes of the simplified code are roughly on par with those of monomorphic code, while unsimplified polymorphic code can be one or two orders of magnitude slower, depending on the degree of polymorphism used in the benchmarks. Since the only actual difference between the monomorphic and simplified polymorphic code lies in the types used, any performance variation can be attributed to measurement error or to the trade-off between using built-in polymorphic lists, which may benefit from compiler and runtime optimisations, and user-defined monomorphic lists, which may gain from reduced overhead due to their fixed types.

To measure the impact of particular simplification phases, we also compiled the Eff standard library, which features multiple recursive higher-order polymorphic functions, mostly on lists. Without simplifications, the compiled version of the standard library functions

	unsimplified	simplified
one solution of n-queens	24.5	0.92
all solutions of n-queens	489.3	0.93
stateful counter	2.48	1.21
list of generator values	2.31	1.01
stateful sum of generator values[1]	1.03	0.94
exceptional arithmetic	1.92	1.07
stateful arithmetic[1]	0.86	0.86
pure tree traversal	48.85	1.21
reader tree traversal	42.32	0.95
stateful tree traversal	32.96	0.99

TABLE 1. Slowdown factors (lower is better) of polymorphic code in comparison to monomorphic one.

featured 447 explicit coercions, all of which were removed after applying the phases. Table 2 shows the information about graph sizes of simplified constraints sets with different simplification settings (recall that phases $\Phi_{\texttt{red}}$ and $\Phi_{\texttt{loopPar}}$ are performed implicitly). The number of edges in type graphs is shown in **bold** as these amount to additional coercion parameters in the compiled code. Additionally, we show the number of generated monadic artifacts that have significant impact on the final runtime efficiency.

simplification level	**dirt nodes/edges**	**type nodes/edges**	`>>=`	`return`
no simplification	632 / 644	435 / **447**	78	29
only $\Phi_{\texttt{scc}}$ (dirt and types)	47 / 53	242 / **193**	19	24
only dirt simplifications	0 / 0	435 / **447**	27	24
only type simplifications	632 / 644	0 / **0**	77	52
all simplifications	0 / 0	0 / **0**	17	31

TABLE 2. Impact of different simplification phases on graph sizes and generated artifacts when compiling EFF standard library.

Similarly, we compiled polymorphic versions of all the above benchmarks in the suite, with the total results shown in Table 3. As with the standard library, the number of nodes, edges and monadic artifacts decreases with each simplification phase. The final result, where all simplifications are applied, is the same as the one obtained by manually monomorphising the benchmarks.

One should notice the marked difference in the sizes of dirt and type constraints between the standard library and benchmarks. This is because the benchmarks are heavily skewed towards effects and handlers to showcase optimizations when inlining handlers, while the

[1]Unfortunately, the two marked benchmarks are affected by an unresolved compiler bug that prevents inlining of a handler, resulting in approximately 700× worse performance compared to hand-written OCaml code without handlers. This regression is most likely due to the extensive refactoring that occurred during the transition to a polymorphic compiler. Previously, the handler was successfully inlined, yielding comparable performance. Due to the excessive runtime, coercions contribute little to the overall cost, and measurement error is amplified.

standard library is geared towards general purpose polymorphic functions (like `map` and `fold`).

simplification level	dirt nodes/edges	type nodes/edges	>>=	return
no simplification	4351 / 4832	89 / **98**	485	253
only $\Phi_{\texttt{scc}}$ (dirt and types)	6 / 8	29 / **20**	84	211
only dirt simplifications	0 / 0	89 / **98**	87	211
only type simplifications	4351 / 4832	0 / **0**	485	265
all simplifications	0 / 0	0 / **0**	84	213

Table 3. Impact of different simplification phases on graph sizes and generated artifacts when compiling polymorphic versions of Eff benchmark suite.

We can clearly see how progressive simplification phases reduce the size of the graphs down to zero. Dirt simplification phases do not reduce coercion parameters as those are irrelevant when compiling to OCaml anyway. However, they do reduce monadic artifacts as they allow purity based optimizations [KKPS21] to kick in and reduce the number of generated binds by more than 60% in favour of more efficient let bindings. As expected by Corollary 6.4, additional casts increase the number of monadic returns as more terminal values are recognized as pure and need to be lifted to be used in computations that remain monadic. Note that all those returns were present before, but in form of additional arguments to functions with coercion parameters.

8. Conclusion

In this work, we have established a framework in which to explore semantics-preserving simplifications of coercions. Even though we mentioned effect handlers only briefly, they are our main motivation, as the massive overhead of polymorphic explicit coercions restricted programmers to using monomorphic functions. With the removal of practically all unnecessary coercion parameters, our work makes the overhead negligible and allows efficient compilation of significantly larger and more modular programs, which is likely to reveal further challenges.

8.1. **Related work.** There are of course multiple other approaches to the efficient evaluation of effect handlers. From OCaml 5 [SDW+21] onwards, effect handlers are available natively through an efficient runtime representation. As the main purpose for introducing handlers into OCaml was the support for various concurrency scheduling strategies, the implementation is focused on efficiency of single-shot handlers, i.e. ones where a continuation is called at most once. In this respect, our approach is agnostic and supports efficient compilation of arbitrary handlers.

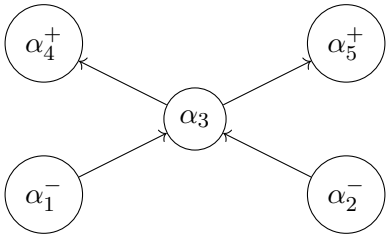
Effect-aware compilation is also featured in Koka [Lei14] that, in addition to an efficient runtime, uses a row-based effect system to determine which functions are pure and do not require a CPS translation [Lei17]. In order to speed up the search for the appropriate encompassing handler, Koka uses *evidence passing*, where a vector of active handlers is passed around to operations at runtime [XL21]. Similar evidence is determined at compile time in Effekt [BSO20], which enforces lexical scoping of handlers [SBO20, SBMO22] at the slight cost of losing full polymorphism. It would also be interesting if this approach, which

features *subregion evidence* [MSS+23], a form of explicit subeffecting witnesses, fits into our simplification framework and could benefit from it in any way.

Another promising venue on the topic of algebraic effects is that of *lift coercions* [BPPS18, BPPS19, XCIL22, YSI24], which enable modular code by explicitly marking operations that should not be handled, thereby preventing their unintentional capture by external handlers. Similar to our subtyping coercions, lift coercions are explicit coercions between different effects, and it would be interesting to see how much they impact the performance and if our approach could simplify them.

Looking at constraint simplification more generally, the existing literature is vast [FM90, TS96, HR95], and uses similar techniques as we do: canonising constraints into a graph, followed by graph reduction and context-specific simplifications. It would be interesting to explore whether bridge compression or our framework of substitutions, coercion families, and complete phases offers any new insights.

One general approach worth mentioning is *garbage collection* [Pot96, Pot01], a canonical simplification algorithm, which performs just as well as, or better than, a collection of heuristics. Here, one first transitively closes the constraints, and then keeps only those of the form $\alpha_1^- \leq \alpha_2^+$. Even though our earlier work [Pre14] was based on garbage collection, this fails to transfer over to explicit coercions. For example, consider the set of type constraints represented by the following graph:



Even if we add additional coercions from α_1 and α_2 to α_4 and α_5 to the graph, the removal of α_3 requires its instantiation, and there is no type we can assign it without changing the meaning.

One could consider adding additional type constructors like $\alpha_1 \sqcup \alpha_2$ or $\alpha_4 \sqcap \alpha_5$ as done in algebraic subtyping [DM17]. However, recall from Section 1.3 that in OCaml, type coercions are translated into polymorphic functions. Translating $\alpha_4 \sqcap \alpha_5$ therefore requires a type constructor `'a4 /\ 'a5`, along with coercions `'a4 /\ 'a5 -> 'a4` and `'a4 /\ 'a5 -> 'a5`, for arbitrary type parameters `'a4` and `'a5`. Type products might serve this purpose, but we have not yet explored this option or the potential performance implications of using (boxed) tuples.

8.2. **Future work.** Even though the results in Section 7.2 indicate there is not a lot left to simplify, we would like to determine when a given set of constraints cannot be reduced further, like it is done for garbage collection [Pot01] or algebraic subtyping [DM17].

Similarly, we would like to explore the role that ordering of simplification phases plays in the end result. Outside of the framework of simplifying contexts, constraints can be reduced even further through transformations of terms, for example, with suitable rules that rearrange the coercions in terms [KKPS21], changing their local meaning whilst preserving the semantics of the whole program.

Another direction in which polymorphism can be combined with the optimizing compiler [KKPS21] is polymorphic function specialisation. For example, a polymorphic list map of type

$$(\alpha_1 \xrightarrow{\delta} \alpha_2) \to (\alpha_1\,\texttt{list} \xrightarrow{\delta} \alpha_2\,\texttt{list})$$

needs to be compiled using the monadic embedding as the given function can be effectful. Since this is inefficient for pure functions, it is beneficial to compile an additional variant [Lei17] with the assumption $\delta = \emptyset$. The number of possible variants is exponential in the number of dirt parameters, so the presented algorithm already helps in this regard. It remains to be explored, but it is likely that in practice only two variants are sufficient, a general one and one with all dirt parameters set to $\emptyset$.

As the focus of our work was practical — how to best reduce the number of coercion parameters — we kept the denotational semantics as simple as necessary to still guarantee soundness of simplifications. However, the interplay between various constructs hints towards a 2-category with parameter contexts as objects, substitutions as morphisms, and coercion families as 2-morphisms, though the role of polarity remains to be explored.

Acknowledgments

For their suggestions and comments, we would like to sincerely thank the anonymous reviewers, Danel Ahman, Andrej Bauer, George Karachalias, Sam Lindley, Tom Schrijvers and the participants of IFIP WG 2.1 meeting in Oxford, especially Fritz Henglein.

References

[BP14] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.*, 10(4), 2014. doi:10.2168/LMCS-10(4:9)2014.

[BPPS18] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL):8:1–8:30, 2018. doi:10.1145/3158096.

[BPPS19] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proc. ACM Program. Lang.*, 3(POPL):6:1–6:28, 2019. doi:10.1145/3290319.

[BSO20] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA):126:1–126:30, 2020. doi:10.1145/3428194.

[DM17] Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsub. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 60–72. ACM, 2017. doi:10.1145/3009837.3009882.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theor. Comput. Sci.*, 73(2):155–175, 1990. doi:10.1016/0304-3975(90)90144-7.

[HR95] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for scheme: Translating scheme to ML. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 192–203. ACM, 1995. doi:10.1145/224164.224203.

[KKPS21] Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. Efficient compilation of algebraic effect handlers. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021. doi:10.1145/3485479.

[Kop24] Filip Koprivec. *Efficient multishot algebraic effect handlers*. PhD thesis, University of Ljubljana, Faculty of Mathematics and Physics, 2024. URL: https://repozitorij.uni-lj.si/IzpisGradiva.php?lang=eng&id=162658.

[KPS+20] Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. Explicit effect subtyping. *J. Funct. Program.*, 30:e15, 2020. doi:10.1017/S0956796820000131.

[Lei14] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014. doi:10.4204/EPTCS.153.8.

[Lei17] Daan Leijen. Type directed compilation of row-typed algebraic effects. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499. ACM, 2017. doi:10.1145/3009837.3009872.

[LPT03] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

[MGJZ25] Cong Ma, Zhaoyi Ge, Max Jung, and Yizhou Zhang. Zero-overhead lexical effect handlers. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025. doi:10.1145/3763177.

[MGLZ24] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. Lexical effect handlers, directly. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1670–1698, 2024. doi:10.1145/3689770.

[MSS+23] Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. From capabilities to regions: Enabling efficient compilation of lexical effect handlers. *Proc. ACM Program. Lang.*, 7(OOPSLA2):941–970, 2023. doi:10.1145/3622831.

[MSSB25] Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. Multiple resumptions and local mutable state, directly. *Proc. ACM Program. Lang.*, 9(ICFP), August 2025. doi:10.1145/3747529.

[Pot96] François Pottier. Simplifying subtyping constraints. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, pages 122–133. ACM, 1996. doi:10.1145/232627.232642.

[Pot01] François Pottier. Simplifying subtyping constraints: A theory. *Inf. Comput.*, 170(2):153–183, 2001. URL: https://doi.org/10.1006/inco.2001.2963, doi:10.1006/INCO.2001.2963.

[PP03] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

[PP13] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.

[Pre14] Matija Pretnar. Inferring algebraic effects. *Log. Methods Comput. Sci.*, 10(3), 2014. doi:10.2168/LMCS-10(3:21)2014.

[Pre15] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, 2015. URL: https://doi.org/10.1016/j.entcs.2015.12.003, doi:10.1016/J.ENTCS.2015.12.003.

[SBMO22] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. A typed continuation-passing translation for lexical effect handlers. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 566–579. ACM, 2022. doi:10.1145/3519939.3523710.

[SBO20] Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.*, 4(ICFP):93:1–93:28, 2020. doi:10.1145/3408975.

[SDW+21] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto ocaml. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 206–221. ACM, 2021. doi:10.1145/3453483.3454039.

[Sim03] 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*, volume 2895 of *Lecture Notes in Computer Science*, pages 283–302. Springer, 2003. doi:10.1007/978-3-540-40018-9_19.

[SKPS18] Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. Explicit effect subtyping. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 327–354. Springer, 2018. doi:10.1007/978-3-319-89884-1_12.

[TS96] 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*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 1996. doi:10.1007/3-540-61739-6_52.

[VJS10] Dimitrios Vytiniotis, Simon L. Peyton Jones, and Tom Schrijvers. Let should not be generalized. In Andrew Kennedy and Nick Benton, editors, *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, pages 39–50. ACM, 2010. doi:10.1145/1708016.1708023.

[WJ99] 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. doi:10.1145/292540.292545.

[XCIL22] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. First-class names for effect handlers. *Proc. ACM Program. Lang.*, 6(OOPSLA2):30–59, 2022. doi:10.1145/3563289.

[XL21] Ningning Xie and Daan Leijen. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473576.

[YSI24] Takuma Yoshioka, Taro Sekiyama, and Atsushi Igarashi. Abstracting effect systems for algebraic effect handlers. *Proc. ACM Program. Lang.*, 8(ICFP):455–484, 2024. doi:10.1145/3674641.

Appendix A. Well-formedness rules

$\vdash_s \Xi_s$ ctx

$$\frac{}{\vdash_s \varepsilon \text{ ctx}} \qquad \frac{\vdash_s \Xi_s \text{ ctx}}{\vdash_s \Xi_s, \varsigma \text{ ctx}}$$

$\Xi_s \vdash S$ skel assuming $\vdash_s \Xi_s$ ctx

$$\frac{\varsigma \in \Xi_s}{\Xi_s \vdash \varsigma \text{ skel}} \qquad \frac{}{\Xi_s \vdash \texttt{Unit} \text{ skel}} \qquad \frac{\Xi_s \vdash S_1 \text{ skel} \qquad \Xi_s \vdash S_2 \text{ skel}}{\Xi_s \vdash S_1 \to S_2 \text{ skel}}$$

$\vdash_d \Xi_d$ ctx

$$\frac{}{\vdash_d \varepsilon \text{ ctx}} \qquad \frac{\vdash_d \Xi_d \text{ ctx}}{\vdash_d \Xi_d, \delta \text{ ctx}}$$

$\Xi_d \vdash \Delta$ dirt assuming $\vdash_d \Xi_d$ ctx

$$\frac{\delta \in \Xi_d}{\Xi_d \vdash \delta \text{ dirt}} \qquad \frac{}{\Xi_d \vdash \emptyset \text{ dirt}} \qquad \frac{(op : A_1 \to A_2) \in \Sigma \qquad \Xi_d \vdash \Delta \text{ dirt}}{\Xi_d \vdash \{op\} \cup \Delta \text{ dirt}}$$

$\Xi_s \vdash_t \Xi_t$ ctx

$$\Xi_s \vdash_t \varepsilon \text{ ctx} \qquad \frac{\Xi_s \vdash_t \Xi_t \text{ ctx} \qquad \Xi_s \vdash S \text{ skel}}{\Xi_s \vdash_t \Xi_t, (\alpha : S) \text{ ctx}}$$

Figure 9. Well-formedness rules for contexts, skeletons, and dirts

$\Xi_d \vdash_{dc} \Xi_{dc}\ \texttt{ctx}$ assuming $\vdash_d \Xi_d\ \texttt{ctx}$

$$\frac{\Xi_d \vdash_{dc} \Xi_{dc}\ \texttt{ctx} \qquad \Xi_d \vdash \Delta\ \texttt{dirt} \qquad \Xi_d \vdash \Delta'\ \texttt{dirt}}{\Xi_d \vdash_{dc} \Xi_{dc}, (\varpi : \Delta \leq \Delta')\ \texttt{ctx}}$$

$\Xi_s; \Xi_d; \Xi_t \vdash_{tc} \Xi_{tc}\ \texttt{ctx}$ assuming $\vdash_t \Xi_t\ \texttt{ctx}$, $\vdash_d \Xi_d\ \texttt{ctx}$ and $\Xi_s \vdash_d \Xi_d\ \texttt{ctx}$

$$\frac{\Xi_s; \Xi_d; \Xi_t \vdash_{tc} \Xi_{tc}\ \texttt{ctx} \qquad \Xi_s; \Xi_d; \Xi_t \vdash A : S \qquad \Xi_s; \Xi_d; \Xi_t \vdash A' : S}{\Xi_s; \Xi_d; \Xi_t \vdash_{tc} \Xi_{tc}, (\omega : A \leq A')\ \texttt{ctx}}$$

Figure 10. Well-formedness rules for coercion contexts

$\Xi \vdash \Gamma\ \texttt{tyCtx}$ assuming $\vdash \Xi\ \texttt{ctx}$

$$\frac{}{\Xi \vdash \varepsilon\ \texttt{tyCtx}} \qquad \frac{\Xi \vdash \Gamma\ \texttt{tyCtx} \qquad \Xi \vdash A : S}{\Xi \vdash \Gamma, (x : A)\ \texttt{tyCtx}}$$

Figure 11. Typing context well-formedness