Explicit effect subtyping

GEORGIOS KARACHALIAS

Department of Computer Science, KU Leuven, Belgium
Tweag I/O
(*e-mail:* georgios.karachalias@tweag.io)

MATIJA PRETNAR

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
(*e-mail:* matija.pretnar@fmf.uni-lj.si)

AMR HANY SALEH, STIEN VANDERHALLEN and TOM SCHRIJVERS

Department of Computer Science, KU Leuven, Belgium
(*e-mails:* ah.saleh@kuleuven.be; stienvdhallen@gmail.com; tom.schrijvers@kuleuven.be)

Abstract

As popularity of algebraic effects and handlers increases, so does a demand for their efficient execution. Eff, an ML-like language with native support for handlers, has a subtyping-based effect system on which an effect-aware optimising compiler could be built. Unfortunately, in our experience, implementing optimisations for Eff is overly error-prone because its core language is implicitly typed, making code transformations very fragile. To remedy this, we present an explicitly typed polymorphic core calculus for algebraic effect handlers with a subtyping-based type-and-effect system. It reifies appeals to subtyping in explicit casts with coercions that witness the subtyping proof, quickly exposing typing bugs in program transformations. Our typing-directed elaboration comes with a constraint-based inference algorithm that turns an implicitly typed Eff-like language into our calculus. Moreover, all coercions and effect information can be erased in a straightforward way, demonstrating that coercions have no computational content. Additionally, we present a monadic translation from our calculus into a pure language without algebraic effects or handlers, using the effect information to introduce monadic constructs only where necessary.

1 Introduction

Algebraic effect handlers (Plotkin & Power, 2003; Plotkin & Pretnar, 2013) are quickly maturing from a theoretical model to a practical language feature for user-defined computational effects. Yet, in practice, they still incur a significant performance overhead compared to native effects.

Our earlier efforts (Pretnar *et al.*, 2017) to narrow this gap with an optimising compiler from Eff (Bauer & Pretnar, 2015) to OCaml showed promising results, in some cases reaching even the performance of hand-tuned code, but were very fragile and have been postponed until a more robust solution is found. We believe the main reason behind these and other[1] problems lies in the complexity of subtyping in combination with the implicit

---

[1] See issues #11 and #16 at https://github.com/matijapretnar/eff/issues/

typing of Eff's core language, further aggravated by the "garbage collection" (Pottier, 2001) of subtyping constraints (see Section 8).

For efficient compilation, one must avoid the poisoning problem (Wansbrough & Peyton Jones, 1999), where unification forces a pure computation to take the less precise impure type of the context (e.g., a pure and an impure branch of a conditional both receive the same impure type). Since this rules out existing (and likely simpler) effect systems for handlers based on row polymorphism (Leijen, 2014; Hillerström & Lindley, 2016; Lindley *et al.*, 2017), we propose a polymorphic explicitly typed calculus based on subtyping. More specifically, our contributions are as follows:

- First, in Section 3 we present IMPEFF, a polymorphic implicitly typed calculus for algebraic effects and handlers with a subtyping-based type-and-effect system. IMPEFF is essentially a (desugared) source language as it appears in the compiler frontend of a language like Eff.
- Next, Section 4 presents EXEFF, the core calculus, which combines explicit System F-style polymorphism with explicit coercions for subtyping in the style of Breazu-Tannen *et al.* (1991). This calculus comes with a type-and-effect system, a small-step operational semantics and a proof of type safety.
- Section 5 specifies the typing-directed elaboration of IMPEFF into EXEFF and presents a type inference algorithm for IMPEFF that produces the elaborated EXEFF term as a byproduct. It also establishes that the elaboration preserves typing, and that the algorithm is sound with respect to the specification and yields principal types.
- Finally, we present two different backends for EXEFF:
  - Section 6 defines SKELEFF, which is a variant of EXEFF without effect information or coercions. SKELEFF is also representative of Multicore OCaml's support for algebraic effects and handlers (Dolan *et al.*, 2015), which is a possible compilation target of Eff. By showing that the erasure from EXEFF to SKELEFF preserves semantics, we establish that EXEFF's coercions are computationally irrelevant. To enable erasure, EXEFF annotates its types with *(type) skeletons*, which capture the erased counterpart and are, to our knowledge, a novel contribution.
  - Section 7 defines NOEFF, which is an alternative backend of EXEFF which tracks in its type system whether, but not which, effects can happen. This backend is representative of pure OCaml or Haskell code where effectful computations are represented with a free monad implementation. Because NOEFF lacks effect polymorphism, our type-preserving elaboration from EXEFF to NOEFF needs to introduce unsafe coercions, though we claim that elaborated programs never get stuck.
- Our paper comes with two software artefacts: an ongoing implementation[2] of a compiler from Eff to OCaml with EXEFF at its core, and an Abella mechanisation[3] of Theorems 4.2, 5.1, 6.2, 6.3, 7.1, 7.2 and 7.4. Remaining theorems all concern the inference algorithm, and their proofs closely follow Pretnar (2014).

---

[2] `https://github.com/matijapretnar/eff/tree/explicit-effect-subtyping`
[3] `https://github.com/matijapretnar/proofs/tree/jfp-2019/explicit-effect-subtyping`

This article is an extended version of a paper that appeared at ESOP 2018 (Saleh *et al.*, 2018). There are two main novelties. Firstly, we have altered the coercion forms available in EXEFF. Previously, it contained a range of projection forms to support an operational semantics that never matches on the coercions. Instead, we now do match on the coercions in the operational semantics, and as a consequence no longer need the projections. This not only reduces the size of the language but also has a considerable simplifying impact on the metatheory proofs in Abella. Moreover, it reduces the gap between EXEFF and NOEFF. Secondly and most importantly, Section 7, on the elaboration of EXEFF to NOEFF, is entirely new.

2 Overview

This section presents an informal overview of the EXEFF calculus, and the main issues with elaborating to and erasing from it.

2.1 Algebraic effect handlers

The main premise of algebraic effects is that impure behaviour arises from a set of *operations* such as Get and Set for mutable store, Read and Print for interactive input and output, or Raise for exceptions (Plotkin & Power, 2003). This allows generalising exception handlers to other effects, to express backtracking, cooperative multithreading and other examples in a natural way (Plotkin & Pretnar, 2013; Bauer & Pretnar, 2015).

Assume operations $\mathtt{Tick} : \mathtt{Unit} \to \mathtt{Unit}$ and $\mathtt{Tock} : \mathtt{Unit} \to \mathtt{Unit}$ that take a unit value as a parameter and yield a unit value as a result. Unlike special built-in operations, these operations have no intrinsic effectful behaviour, though we can give one through handlers. For example, the handler

$$\{\mathtt{Tick}\, x\, k \mapsto (\mathtt{Print}\text{``tick''}; k\, \mathtt{unit}),$$

$$\mathtt{Tock}\, x\, k \mapsto \mathtt{Print}\text{``tock''}\}$$

replaces all calls of Tick by printing out "tick" and similarly for Tock. But there is one significant difference between the two cases. Unlike exceptions, which always abort the evaluation, operations have a continuation waiting for their result. It is this continuation that the handler captures in the variable k and potentially uses in the handling clause. In the clause for Tick, the continuation is resumed by passing it the expected unit value, whereas in the clause for Tock, the operation is discarded. Thus, if we handle a computation emitting the two operations, it will print out "tick" until a first "tock" is printed, after which the evaluation stops. For a more thorough explanation of algebraic effect handlers, we refer the reader to Pretnar's tutorial (Pretnar, 2015), which is conveniently based on a calculus with essentially the same term-level syntax and operational semantics (but a far less involved type system).

2.2 Elaborating subtyping

Consider the computation $\mathtt{do}\, x \leftarrow \mathtt{Tick}\, \mathtt{unit}; f\, x$ and assume that f has the function type $\mathtt{Unit} \to \mathtt{Unit}\, !\, \{\mathtt{Tock}\}$, taking unit values to unit values and perhaps calling Tock operations in the process. The whole computation then has the type $\mathtt{Unit}\, !\, \{\mathtt{Tick}, \mathtt{Tock}\}$ as it returns the unit value and may call Tick and Tock.

The above typing implicitly appeals to subtyping in several places. For instance, `Tick unit` has type `Unit ! {Tick}` and $f\ x$ type `Unit ! {Tock}`. Yet, because they are sequenced with `do`, the type system expects them to have the same set of effects. The discrepancies are implicitly reconciled by the subtyping which admits both $\{\texttt{Tick}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$ and $\{\texttt{Tock}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$.

We elaborate the IMPEFF term into the explicitly typed core language EXEFF, where such implicit appeals to subtyping turn into explicit casts using *coercions*:

$$\texttt{do}\ x \leftarrow ((\texttt{Tick unit}) \rhd \gamma_1); (f\ x) \rhd \gamma_2$$

A coercion γ is a witness for a subtyping $A\ !\ \Delta \leqslant A'\ !\ \Delta'$ and can be used to cast a term c of type $A\ !\ \Delta$ to a term $c \rhd \gamma$ of type $A'\ !\ \Delta'$. In the above term, γ_1 and γ_2 respectively witness $\texttt{Unit}\ !\ \{\texttt{Tick}\} \leqslant \texttt{Unit}\ !\ \{\texttt{Tick}, \texttt{Tock}\}$ and $\texttt{Unit}\ !\ \{\texttt{Tock}\} \leqslant \texttt{Unit}\ !\ \{\texttt{Tick}, \texttt{Tock}\}$.

At this point, the reader might wonder why coercions can influence value types, and not just effect sets. This design allows us to flexibly cast types of higher-order functions and handlers which would otherwise not be possible. For example, we can use a coercion for $\delta_3 \leqslant \delta_1$ to construct value-type coercions that witnesses

$$((\alpha \to \alpha'\ !\ \delta_1) \to \alpha''\ !\ \delta_2) \leqslant ((\alpha \to \alpha'\ !\ \delta_3) \to \alpha''\ !\ \delta_2)$$

or

$$(\alpha'\ !\ \delta_1 \Rightarrow \alpha''\ !\ \delta_2) \leqslant (\alpha'\ !\ \delta_3 \Rightarrow \alpha''\ !\ \delta_2)$$

2.3 Polymorphic subtyping for types and effects

The above basic example only features monomorphic types and effects. Yet, our calculus also supports polymorphism, which makes it considerably more expressive. For instance, the type of f in `let f = (fun g ↦ g unit) in ...` is generalised to:

$$\forall \alpha, \alpha'.\forall \delta, \delta'.\alpha \leqslant \alpha' \Rightarrow \delta \leqslant \delta' \Rightarrow (\texttt{Unit} \to \alpha\ !\ \delta) \to \alpha'\ !\ \delta'$$

This polymorphic type scheme follows the qualified types convention (Jones, 1992) where the type $(\texttt{Unit} \to \alpha\ !\ \delta) \to \alpha'\ !\ \delta'$ is subjected to several qualifiers, in this case $\alpha \leqslant \alpha'$ and $\delta \leqslant \delta'$. The universal quantifiers on the outside bind the type variables α and α', and the effect set variables δ and δ'.

The elaboration of f into EXEFF introduces explicit binders for both the quantifiers and the qualifiers, as well as the explicit casts where subtyping is used.

$$\Lambda\alpha.\Lambda\alpha'.\Lambda\delta.\Lambda\delta'\Lambda(\omega : \alpha \leqslant \alpha').\Lambda(\omega' : \delta \leqslant \delta').$$
$$\texttt{fun}\ (g : \texttt{Unit} \to \alpha\ !\ \delta) \mapsto ((g\ \texttt{unit}) \rhd (\omega\ !\ \omega'))$$

Here the binders for qualifiers introduce *coercion variables* ω between pure types and ω' between operation sets, which are then combined into a *computation coercion* $\omega\ !\ \omega'$ and used for casting the function application $(g\ \texttt{unit})$ to the expected type.

Suppose that h has type $\texttt{Unit} \to \texttt{Unit}\ !\ \{\texttt{Tick}\}$ and $f\ h$ type $\texttt{Unit}\ !\ \{\texttt{Tick}, \texttt{Tock}\}$. In the EXEFF calculus, the corresponding instantiation of f is made explicit through type and coercion applications

$$f\ \texttt{Unit}\ \texttt{Unit}\ \{\texttt{Tick}\}\ \{\texttt{Tick}, \texttt{Tock}\}\ \gamma_1\ \gamma_2\ h$$

where γ_1 needs to be a witness for $\texttt{Unit} \leqslant \texttt{Unit}$ and γ_2 for $\{\texttt{Tick}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$.

2.4 Guaranteed erasure with skeletons

One of our main requirements for EXEFF is that its effect information and subtyping can be easily erased. The reason is twofold. Firstly, we want to show that neither plays a role in the runtime behaviour of EXEFF programs. Secondly and more importantly, we want to use a conventionally typed (System F-like) functional language as a backend for the Eff compiler.

At first, erasure of both effect information and subtyping seems easy: simply drop that information from types and terms. But by dropping the effect variables and subtyping constraints from the type of f, we get $\forall \alpha, \alpha'.(\text{Unit} \to \alpha) \to \alpha'$ instead of the expected type $\forall \alpha.(\text{Unit} \to \alpha) \to \alpha$. In our naive erasure attempt, we have carelessly discarded the connection between α and α'. A more appropriate approach to erasure would be to unify the types in dropped subtyping constraints. However, unifying types may reduce the number of type variables when they become instantiated, so corresponding binders need to be dropped, greatly complicating the erasure procedure and its metatheory.

Fortunately, there is an easier way by tagging all bound-type variables with *skeletons*, which are bare-bones types without effect information. For example, the skeleton of a function type $A \to B \,!\, \Delta$ is $\tau_1 \to \tau_2$, where τ_1 is the skeleton of A and τ_2 the skeleton of B. In EXEFF, every well-formed type has an associated skeleton, and any two types $A_1 \leqslant A_2$ share the same skeleton. In particular, binders for type variables are explicitly annotated with skeleton variables ς. For instance, the actual type of f is

$$\forall \varsigma.\forall(\alpha : \varsigma), (\alpha' : \varsigma).\forall \delta, \delta'.\alpha \leqslant \alpha' \Rightarrow \delta \leqslant \delta' \Rightarrow (\text{Unit} \to \alpha \,!\, \delta) \to \alpha' \,!\, \delta'$$

The skeleton quantifications and annotations also appear at the term level:

$$\Lambda \varsigma.\Lambda(\alpha : \varsigma).\Lambda(\alpha' : \varsigma).\Lambda \delta.\Lambda \delta'.\Lambda(\omega : \alpha \leqslant \alpha').\Lambda(\omega' : \delta \leqslant \delta')....$$

Now erasure is really easy: we drop not only effect and subtyping-related term formers, but also type binders and application. We do retain skeleton binders and applications, which take over the role of (plain) types in the backend language. In terms, we replace types by their skeletons. For instance, for f we get

$$\Lambda \varsigma.\text{fun}\,(g : \text{Unit} \to \varsigma) \mapsto g\,\text{unit} \;:\; \forall \varsigma.(\text{Unit} \to \varsigma) \to \varsigma$$

2.5 Elaboration into a pure language

We can drop effectful information only if the targeted language has a native implicit support for algebraic effects at any type. In a pure functional language, effectful computations that yield a result of type A are represented with a user-defined type Comp A, which typically uses one of the known encodings, such as free monads (Kammar *et al.*, 2013; Pretnar *et al.*, 2017), delimited control (Kiselyov & Sivaramakrishnan, 2016) or continuation-passing style (Leijen, 2017).

Targeting such a language requires a more careful elaboration. For example, EXEFF types Int ! {Tick} and Int ! {Tock} are both mapped to a type Comp Int. The same could be done for the type Int ! Ø, but computations of that type are pure and do not require any encoding, so it is more efficient to avoid the library overhead and map the type to the pure type Int directly (Leijen, 2017; Pretnar *et al.*, 2017). This difference is the main complicating factor in the elaboration.
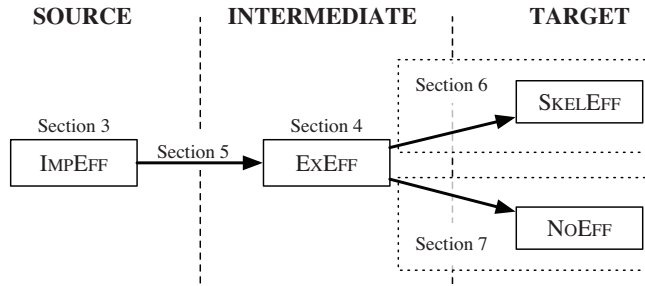
SOURCE   **INTERMEDIATE**   **TARGET**

Fig. 1. Compiler and section structure.

Since the computation $\texttt{return } 5 : \texttt{Int } ! \ \emptyset$ is pure, it should be elaborated to 5 of type $\texttt{Int}$. But if we take a witness γ for $\texttt{Int} \leqslant \texttt{Int}$ and γ_1 for $\emptyset \leqslant \{\texttt{Tick}\}$, the coerced computation $(\texttt{return } 5) \rhd (\gamma \ ! \ \gamma_1) : \texttt{Int } ! \ \{\texttt{Tick}\}$ should be elaborated to the lifted value $\texttt{return } 5 : \texttt{Comp Int}$.

However, it is not simply a matter of replacing each cast with a $\texttt{return}$. If we further take a witness γ_2 of $\{\texttt{Tick}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$, the computation

$$((\texttt{return } 5) \rhd (\gamma \ ! \ \gamma_1)) \rhd (\gamma \ ! \ \gamma_2) : \texttt{Int } ! \ \{\texttt{Tick}, \texttt{Tock}\}$$

also has to be elaborated to $\texttt{return } 5 : \texttt{Comp Int}$. We will see that this is just one of the (smaller) issues that stem from the different treatment of pure and impure computation types and show how to construct an appropriate elaboration (Section 7.4).

2.6 Outline

The remainder of this article formalises essentially a compiler pipeline for Eff. Figure 1 depicts this pipeline and annotates the different parts with the sections they are covered in.

Section 3: The starting point of the pipeline is IMPEFF, an implicitly typed calculus for algebraic effects and handlers with a subtyping-based type-and-effect system. It is the core of the desugared source language as it appears in the compiler frontend of Eff. We present its syntax and type system.

Section 4: The heart of the compiler is EXEFF, an intermediate language that is explicitly annotated with type and effect information. Its main novelty is that it also makes appeals to subtyping explicit by means of coercions. We present its syntax, type system and operational semantics.

Section 5: We explain how to elaborate IMPEFF into EXEFF and provide a type inference algorithm for IMPEFF that performs this elaboration. The algorithm is constraint-based, that is, it consists of two interleaved phases: constraint generation and constraint solving.

Section 6: Towards the end, the compiler forks to support two different compilation targets. The first compilation target is SKELEFF. This language is modelled after Multicore OCaml. In particular, it is a statically typed language with built-in support for algebraic effects, but its type system does not track effects. We provide its syntax and, in the appendix, also its type system and operational semantics. Also, we explain

Terms

$$\text{value } v ::= x \mid \text{unit} \mid \text{fun } x \mapsto c \mid h$$
$$\text{handler } h ::= \{\text{return } x \mapsto c_r, \text{Op}_1 \, x \, k \mapsto c_{\text{Op}_1}, \ldots, \text{Op}_n \, x \, k \mapsto c_{\text{Op}_n}\}$$
$$\text{computation } c ::= \text{return } v \mid \text{Op } v \, (y.c) \mid \text{do } x \leftarrow c_1; c_2$$
$$\mid \; \text{handle } c \text{ with } v \mid v_1 \, v_2 \mid \text{let } x = v \text{ in } c$$

Types & Constraints

$$\text{skeleton } \tau ::= \varsigma \mid \text{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \Rightarrow \tau_2$$

$$\text{value type } A, B ::= \alpha \mid \text{Unit} \mid A \rightarrow \underline{C} \mid \underline{C} \Rightarrow \underline{D}$$
$$\text{qualified type } K ::= A \mid \pi \Rightarrow K$$
$$\text{polytype } S ::= K \mid \forall \varsigma.S \mid \forall \alpha : \tau.S \mid \forall \delta.S$$
$$\text{computation type } \underline{C}, \underline{D} ::= A \, ! \, \Delta$$
$$\text{dirt } \Delta ::= \delta \mid \emptyset \mid \{\text{Op}\} \cup \Delta$$

$$\text{simple constraint } \pi ::= A_1 \leqslant A_2 \mid \Delta_1 \leqslant \Delta_2$$
$$\text{constraint } \rho ::= \pi \mid \underline{C} \leqslant \underline{D}$$

Fig. 2. IMPEFF syntax.

how to elaborate the intermediate EXEFF into the SKELEFF target language. Thanks to the skeleton-based set-up of EXEFF, this elaboration is a fairly straightforward erasure procedure.

Section 7: The second compilation target is NOEFF, a statically typed calculus that distinguishes in its types between pure and impure computations, but does not track which operations can happen in impure computations. This models encodings of algebraic effects in languages without native support. We present its syntax, type system and operational semantics. Finally, we show how to elaborate EXEFF into NOEFF. This is much more involved than the straightforward erasure procedure into SKELEFF. Instead of just throwing away all effect information and coercions, we have to abstract it to the presence (pure) or absence (impure) of effects. Unfortunately, polymorphism does not interact well with this abstraction process. We show how to address this problem by conservatively assuming that polymorphic code is impure and by adding unsafe coercions to obtain pure instantiations.

3 The IMPEFF language

This section presents IMPEFF, a basic functional calculus with support for algebraic effect handlers, which forms the core language of our optimising compiler.

3.1 Syntax

Figure 2 presents the syntax of the source language. There are two main kinds of terms: (pure) values v and (dirty) computations c, which may call effectful operations. Handlers

h are a subsidiary sort of values. We assume a given set of *operations* Op, such as Get and Put. We abbreviate $\text{Op}_1\, x\, k \mapsto c_{\text{Op}_1}, \ldots, \text{Op}_n\, x\, k \mapsto c_{\text{Op}_n}$ as $[\text{Op}\, x\, k \mapsto c_{\text{Op}}]_{\text{Op}\in\mathcal{O}}$ and write $\mathcal{O}$ to denote the set $\{\text{Op}_1, \ldots, \text{Op}_n\}$.

Similarly, we distinguish between two basic sorts of types: the value types A, B and the computation types $\underline{C}, \underline{D}$. There are four forms of value types: type variables α, function types $A \rightarrow \underline{C}$, handler types $\underline{C} \Rightarrow \underline{D}$ and the Unit type. Skeletons τ capture the shape of types, so, by design, their forms are identical. The computation type $A \,!\, \Delta$ is assigned to a computation returning values of type A and potentially calling operations from the *dirt* set Δ. A dirt set contains zero or more operations Op and is terminated either by an empty set or a dirt variable δ. Though we use cons-list syntax, the intended semantics of dirt sets Δ is that the order of operations Op is irrelevant. That is, $(\{\text{Op}_1\} \cup (\{\text{Op}_2\} \cup \Delta))$ denotes the same dirt as $(\{\text{Op}_2\} \cup (\{\text{Op}_1\} \cup \Delta))$. Similarly to all Hindley-Milner based systems, we discriminate between value types (or monotypes) A, qualified types K and polytypes (or type schemes) S. (Simple) subtyping constraints π denote inequalities between either value types or dirts. We also present the more general form of constraints ρ that includes inequalities between computation types (as we illustrate in Section 3.2 below, this allows for a single, uniform constraint entailment relation). Finally, polytypes consist of zero or more skeleton, type or dirt abstractions followed by a qualified type.

3.2 *Typing*

Figure 3 presents the typing rules for values and computations, along with a typing-directed elaboration into our target language ExEff. In order to simplify the presentation, in this section, we focus exclusively on typing. The parts of the rules that concern elaboration are highlighted in grey and are discussed in Section 5. In all the rules, we assume a global signature Σ that captures all defined operations along with their (well-formed) types.

Values. Typing for values takes the form $\Gamma \vdash_v v : A$, and, given a typing environment Γ, checks a value v against a value type A.

Rule TMVAR handles term variables. Given that x has type $(\forall \overline{\varsigma}.\overline{\alpha : \tau}.\forall \overline{\delta}.\overline{\pi} \Rightarrow A)$, we *appropriately* instantiate the skeleton $(\overline{\varsigma})$, type $(\overline{\alpha})$, and dirt $(\overline{\delta})$ variables, and ensure that the instantiated wanted constraints $\overline{\sigma(\pi)}$ are satisfied, via side condition $\overline{\Gamma \vdash_{co} \sigma(\pi)}$. Rule TMCASTV allows casting the type of a value v from A to B, if A is a subtype of B (upcasting). As illustrated by Rule TMTMABS, we omit freshness conditions by adopting the Barendregt convention (Barendregt, 1981). Finally, Rule TMHAND gives typing for handlers. It requires that the right-hand sides of the return clause and all operation clauses have the same computation type $(B \,!\, \Delta)$, and that all operations mentioned are part of the top-level signature Σ. The result type takes the form $A \,!\, \Delta \cup \mathcal{O} \Rightarrow B \,!\, \Delta$, capturing the intended handler semantics: given a computation of type $A \,!\, \Delta \cup \mathcal{O}$, the handler (a) produces a result of type B, (b) handles operations $\mathcal{O}$ and (c) propagates unhandled operations Δ to the output.

Computations. Typing for computations takes the form $\Gamma \vdash_c c : \underline{C}$, and, given a typing environment Γ, checks a computation c against a type $\underline{C}$.

typing environment $\Gamma ::= \varepsilon \mid \Gamma, \varsigma \mid \Gamma, \alpha : \tau \mid \Gamma, \delta \mid \Gamma, x : S \mid \Gamma, \omega : \pi$

$\boxed{\Gamma \vdash_v v : A \rightsquigarrow v'}$ **Values**

$$\frac{(x : \forall \varsigma.\forall \overline{\alpha : \tau}.\forall \overline{\delta}.\overline{\pi} \Rightarrow A) \in \Gamma \qquad \Gamma \vdash_{\mathrm{vty}} B : \tau \rightsquigarrow T \qquad \sigma = [\overline{\tau'/\varsigma}, \overline{B/\alpha}, \overline{\Delta/\delta}] \qquad \Gamma \vdash_{\mathrm{co}} \overline{\gamma : \sigma(\pi)}}{\Gamma \vdash_v x : \sigma(A) \rightsquigarrow x \; \overline{\tau'} \; \overline{T} \; \overline{\Delta} \; \overline{\gamma}} \; \textsc{TmVar}$$

$$\frac{\Gamma \vdash_v v : A \rightsquigarrow v' \qquad \Gamma \vdash_{\mathrm{co}} \gamma : A \leqslant B}{\Gamma \vdash_v v : B \rightsquigarrow v' \rhd \gamma} \; \textsc{TmCastV} \qquad\qquad \frac{}{\Gamma \vdash_v \mathtt{unit} : \mathtt{Unit} \rightsquigarrow \mathtt{unit}} \; \textsc{TmUnit}$$

$$\frac{\Gamma, x : A \vdash_c c : \underline{C} \rightsquigarrow c' \qquad \Gamma \vdash_{\mathrm{vty}} A : \tau \rightsquigarrow T}{\Gamma \vdash_v (\mathtt{fun}\; x \mapsto c) : A \to \underline{C} \rightsquigarrow \mathtt{fun}\; (x : T) \mapsto c'} \; \textsc{TmTmAbs}$$

$$\frac{\begin{array}{c} \Gamma, x : A \vdash_c c_r : B \,!\, \Delta \rightsquigarrow c'_r \qquad \Gamma \vdash_{\mathrm{vty}} A : \tau \rightsquigarrow T \\ \left[(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \quad \Gamma, x : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B \,!\, \Delta \vdash_c c_{\mathtt{Op}} : B \,!\, \Delta \rightsquigarrow c'_{\mathtt{Op}} \right]_{\mathtt{Op} \in \mathcal{O}} \\ c_{res} = \{\mathtt{return}\,(x : T) \mapsto c'_r, [\mathtt{Op}\, x\, k \mapsto c'_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} \end{array}}{\Gamma \vdash_v \{\mathtt{return}\; x \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : A \,!\, \Delta \cup \mathcal{O} \Rightarrow B \,!\, \Delta \rightsquigarrow c_{res}} \; \textsc{TmHand}$$

$\boxed{\Gamma \vdash_c c : \underline{C} \rightsquigarrow c'}$ **Computations**

$$\frac{\Gamma \vdash_c c : \underline{C}_1 \rightsquigarrow c' \qquad \Gamma \vdash_{\mathrm{co}} \gamma : \underline{C}_1 \leqslant \underline{C}_2}{\Gamma \vdash_c c : \underline{C}_2 \rightsquigarrow c' \rhd \gamma} \; \textsc{TmCastC} \qquad \frac{\begin{array}{c} \Gamma \vdash_v v_1 : A \to \underline{C} \rightsquigarrow v'_1 \\ \Gamma \vdash_v v_2 : A \rightsquigarrow v'_2 \end{array}}{\Gamma \vdash_c v_1 \, v_2 : \underline{C} \rightsquigarrow v'_1 \, v'_2} \; \textsc{TmTmApp}$$

$$\frac{S = \forall \varsigma.\overline{\alpha : \tau}.\forall \overline{\delta}.\overline{\pi} \Rightarrow A \qquad \Gamma, \overline{\varsigma}, \overline{\alpha : \tau}, \overline{\delta}, \overline{\omega : \pi} \vdash_v v : A \rightsquigarrow v' \qquad \Gamma, x : S \vdash_c c : \underline{C} \rightsquigarrow c'}{\Gamma \vdash_c \mathtt{let}\; x = v \;\mathtt{in}\; c : \underline{C} \rightsquigarrow \mathtt{let}\; x = \Lambda \overline{\varsigma}.\Lambda \overline{\alpha : \tau}.\Lambda \overline{\delta}.\Lambda (\overline{\omega : \pi}).v' \;\mathtt{in}\; c'} \; \textsc{TmLet}$$

$$\frac{\Gamma \vdash_v v : A \rightsquigarrow v'}{\Gamma \vdash_c \mathtt{return}\; v : A \,!\, \emptyset \rightsquigarrow \mathtt{return}\; v'} \; \textsc{TmReturn}$$

$$\frac{(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma}{\Gamma \vdash_v v : A_{\mathtt{Op}} \rightsquigarrow v' \quad \Gamma, y : B_{\mathtt{Op}} \vdash_c c : A \,!\, \Delta \rightsquigarrow c' \quad \Gamma \vdash_{\mathrm{vty}} B_{\mathtt{Op}} : \tau \rightsquigarrow T_{\mathtt{Op}} \quad \mathtt{Op} \in \Delta}{\Gamma \vdash_c \mathtt{Op}\; v\; (y.c) : A \,!\, \Delta \rightsquigarrow \mathtt{Op}\; v'\; (y : T_{\mathtt{Op}}.c')} \; \textsc{TmOp}$$

$$\frac{\Gamma \vdash_c c_1 : A \,!\, \Delta \rightsquigarrow c'_1 \qquad \Gamma, x : A \vdash_c c_2 : B \,!\, \Delta \rightsquigarrow c'_2}{\Gamma \vdash_c \mathtt{do}\; x \leftarrow c_1; c_2 : B \,!\, \Delta \rightsquigarrow \mathtt{do}\; x \leftarrow c'_1; c'_2} \; \textsc{TmDo}$$

$$\frac{\Gamma \vdash_v v : \underline{C} \Rightarrow \underline{D} \rightsquigarrow v' \qquad \Gamma \vdash_c c : \underline{C} \rightsquigarrow c'}{\Gamma \vdash_c \mathtt{handle}\; c \;\mathtt{with}\; v : \underline{D} \rightsquigarrow \mathtt{handle}\; c' \;\mathtt{with}\; v'} \; \textsc{TmHandle}$$

Fig. 3. IMPEFF Typing & elaboration.

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \rho}$ **Constraint Entailment**

$$\frac{(\omega : \pi) \in \Gamma}{\Gamma \vdash_{\mathsf{co}} \omega : \pi} \text{ CoVar} \qquad\qquad \frac{}{\Gamma \vdash_{\mathsf{co}} \langle \mathtt{Unit} \rangle : \mathtt{Unit} \leqslant \mathtt{Unit}} \text{ UCoRefl}$$

$$\frac{(\alpha : \tau) \in \Gamma}{\Gamma \vdash_{\mathsf{co}} \langle \alpha \rangle : \alpha \leqslant \alpha} \text{ ACoRefl} \qquad\qquad \frac{\Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\mathsf{co}} \langle \Delta \rangle : \Delta \leqslant \Delta} \text{ DCoRefl}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : B \leqslant A \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C} \leqslant \underline{D}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \to \gamma_2 : A \to \underline{C} \leqslant B \to \underline{D}} \text{ VCoArr}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \underline{C}_2 \leqslant \underline{C}_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{D}_1 \leqslant \underline{D}_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \Rightarrow \gamma_2 : \underline{C}_1 \Rightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rightarrow \underline{D}_2} \text{ VCoHand}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_1 \leqslant A_2 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \Delta_1 \leqslant \Delta_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \, ! \, \gamma_2 : A_1 \, ! \, \Delta_1 \leqslant A_2 \, ! \, \Delta_2} \text{ CCoComp} \qquad \frac{}{\Gamma \vdash_{\mathsf{co}} \emptyset_{\Delta} : \emptyset \leqslant \Delta} \text{ DCoNil}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \Delta_1 \leqslant \Delta_2 \qquad (\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma}{\Gamma \vdash_{\mathsf{co}} \{\mathtt{Op}\} \cup \gamma : \{\mathtt{Op}\} \cup \Delta_1 \leqslant \{\mathtt{Op}\} \cup \Delta_2} \text{ DCoOp}$$

Fig. 4. IMPEFF constraint entailment.

Rule TMCASTC behaves like Rule TMCASTV, but for computation types. Rule TMLET handles polymorphic, non-recursive let-bindings. Rule TMRETURN handles return v computations. Keyword return effectively lifts a value v of type A into a computation of type $A \, ! \, \emptyset$. Rule TMOP checks operation calls. First, we ensure that v has the appropriate type, as specified by the signature of Op. Then, the continuation $(y.c)$ is checked. The side condition Op $\in \Delta$ ensures that the called operation Op is captured in the result type. Rule TMDO handles sequencing. Given that c_1 has type $A \, ! \, \Delta$, the pure part of the result of type A is bound to term variable x, which is brought in scope for checking c_2. As we mentioned in Section 2, all computations in a do-construct should have the same effect set, Δ. Rule TMHANDLE eliminates handler types, just as Rule TMTMAPP eliminates arrow types.

Constraint Entailment. The specification of constraint entailment takes the form $\Gamma \vdash_{\mathsf{co}} \rho$ and is presented in Figure 4. Notice that we use ρ instead of π, which allows us to capture subtyping between two value types, computation types or dirts, within the same relation. Subtyping can be established in several ways:

Rule COVAR handles assumptions. Rules UCOREFL, ACOREF and DCOREFL express that subtyping is reflexive, for the unit type, type variables and dirts, respectively. Notice that we do not have dedicated rules for reflexivity of arbitrary computation or value types; as we illustrate below (Section 4.1), they can both be established using the reflexivity of their subparts. Rule VCOARR establishes inequality of arrow types. As usual, the arrow

type constructor is contravariant in the argument type. Rule VCoHand is similar, but for handler types. Rule CCoComp captures the covariance of type constructor (!), establishing subtyping between two computation types if subtyping is established for their respective subparts. Finally, Rules DCoNil and DCoOp establish subtyping between dirts. Rule DCoNil captures that the empty dirty set $\emptyset$ is a subdirt of any dirt Δ and Rule DCoOp expresses that dirt subtyping preserved under extension with the same operation $\mathtt{Op}$.

Well-formedness of Types, Constraints, Dirts and Skeletons. The relations $\Gamma \vdash_{\mathtt{vty}} A : \tau$ and $\Gamma \vdash_{\mathtt{cty}} \underline{C} : \tau$ check the well-formedness of value and computation types, respectively. Similarly, relations $\Gamma \vdash_{\mathtt{ct}} \rho$ and $\Gamma \vdash_{\Delta} \Delta$ check the well-formedness of constraints and dirts, respectively. They are all defined in Appendix A.

Example 3.1. Recall the definition $\mathtt{let}\ f = (\mathtt{fun}\ g \mapsto g\ \mathtt{unit})\ \mathtt{in}\ \ldots$ of a polymorphic f from Section 2.3. Under different rule applications, f can be given different typings, including simple $(\mathtt{Unit} \to \mathtt{Unit}\,!\,\emptyset) \to \mathtt{Unit}\,!\,\emptyset$ under the typing

$$
\dfrac{
\dfrac{
\dfrac{\rule{3.5cm}{0.4pt}}{g:(\mathtt{Unit}\to\mathtt{Unit}\,!\,\emptyset)\vdash_v g:\mathtt{Unit}\to\mathtt{Unit}\,!\,\emptyset}\ \textsc{TmVar} \qquad \dfrac{\rule{2.5cm}{0.4pt}}{\Gamma\vdash_v \mathtt{unit}:\mathtt{Unit}}\ \textsc{TmUnit}
}{g:(\mathtt{Unit}\to\mathtt{Unit}\,!\,\emptyset)\vdash_c g\ \mathtt{unit}:\mathtt{Unit}\,!\,\emptyset}\ \textsc{TmTmApp}
}{\varepsilon\vdash_v (\mathtt{fun}\ g\mapsto g\ \mathtt{unit}):(\mathtt{Unit}\to\mathtt{Unit}\,!\,\emptyset)\to\mathtt{Unit}\,!\,\emptyset}\ \textsc{TmTmAbs}
$$

and the more involved polytype

$$S = \forall\varsigma.\forall\alpha:\varsigma,\alpha':\varsigma.\forall\delta,\delta'.\alpha\leqslant\alpha'\Rightarrow\delta\leqslant\delta'\Rightarrow(\mathtt{Unit}\to\alpha\,!\,\delta)\to\alpha'\,!\,\delta'$$

obtained by generalising

$$
\dfrac{
\textsc{TmTmApp}\ \dfrac{
\dfrac{\cdots}{\Gamma,g:(\mathtt{Unit}\to\alpha\,!\,\delta)\vdash_c g\ \mathtt{unit}:\alpha\,!\,\delta} \qquad
\dfrac{
\dfrac{\dfrac{}{\Gamma\vdash_{\overline{c}o}\alpha\leqslant\alpha'}\ \textsc{CoVar} \quad \dfrac{}{\Gamma\vdash_{\overline{c}o}\delta\leqslant\delta'}\ \textsc{CoVar}}{\Gamma\vdash_{\overline{c}o}\alpha\,!\,\delta\leqslant\alpha'\,!\,\delta'}\ \textsc{CCoComp}
}{}
}{
\Gamma,g:(\mathtt{Unit}\to\alpha\,!\,\delta)\vdash_c g\ \mathtt{unit}:\alpha'\,!\,\delta'
}\ \textsc{TmCastC}
}{
\underbrace{\varsigma,\alpha:\varsigma,\alpha':\varsigma,\delta,\delta',\alpha\leqslant\alpha',\delta\leqslant\delta'}_{\Gamma}\vdash_v (\mathtt{fun}\ g\mapsto g\ \mathtt{unit}):(\mathtt{Unit}\to\alpha\,!\,\delta)\to\alpha'\,!\,\delta'
}\ \textsc{TmTmAbs}
$$

Using the latter typing, f may be applied to a pure $id = \mathtt{fun}\ x\mapsto \mathtt{return}\ x$ as

$$
\dfrac{
\textsc{TmVar}\ \dfrac{
\begin{array}{c}\Gamma\vdash_{\mathtt{vty}}\mathtt{Unit}:\mathtt{Unit}\\ \Gamma\vdash_{\mathtt{co}}\emptyset\leqslant\emptyset\\ \sigma=[\mathtt{Unit}/\varsigma,\mathtt{Unit}/\alpha,\mathtt{Unit}/\alpha',\emptyset/\delta,\emptyset/\delta']\end{array}
}{\Gamma\vdash_v f:(\mathtt{Unit}\to\mathtt{Unit}\,!\,\emptyset)\to\mathtt{Unit}\,!\,\emptyset} \qquad \dfrac{\cdots}{\Gamma\vdash_v id:\mathtt{Unit}\to\mathtt{Unit}\,!\,\emptyset}\ \textsc{TmTmAbs}
}{
\underbrace{f:S}_{\Gamma}\vdash_c f\ id:\mathtt{Unit}\,!\,\emptyset
}\ \textsc{TmTmApp}
$$

We can also apply f to an impure $tick = \mathtt{fun}\ x\mapsto \mathtt{Tick}\ x\ (y.\mathtt{return}\ y)$ and even enlarge the final dirt as

$$
\dfrac{
\textsc{TmVar}\ \dfrac{
\begin{array}{c}\Gamma\vdash_{\mathtt{vty}}\mathtt{Unit}:\mathtt{Unit}\\ \Gamma\vdash_{\overline{c}o}\{\mathtt{Tick}\}\leqslant\{\mathtt{Tick},\mathtt{Tock}\}\\ \sigma=[\mathtt{Unit}/\varsigma,\mathtt{Unit}/\alpha,\mathtt{Unit}/\alpha',\{\mathtt{Tick}\}/\delta,\{\mathtt{Tick},\mathtt{Tock}\}/\delta']\end{array}
}{\Gamma\vdash_v f:(\mathtt{Unit}\to\mathtt{Unit}\,!\,\{\mathtt{Tick}\})\to\mathtt{Unit}\,!\,\{\mathtt{Tick},\mathtt{Tock}\}} \qquad \dfrac{\dfrac{\cdots}{\Gamma\vdash_v tick:\mathtt{Unit}\to\mathtt{Unit}\,!\,\{\mathtt{Tick}\}}\ \textsc{TmTmAbs}}{\vdots}
}{
\Gamma\vdash_c f\ tick:\mathtt{Unit}\,!\,\{\mathtt{Tick},\mathtt{Tock}\}
}\ \textsc{TmTmApp}
$$

Terms

$$\text{value } v ::= x \mid \texttt{unit} \mid \texttt{fun}\,(x:T) \mapsto c \mid h$$
$$\mid \; \Lambda\varsigma.v \mid v\,\tau \mid \Lambda\alpha:\tau.v \mid v\,T \mid \Lambda\delta.v \mid v\,\Delta \mid \Lambda(\omega:\pi).v \mid v\,\gamma \mid v \triangleright \gamma$$
$$\text{handler } h ::= \{\texttt{return}\,(x:T) \mapsto c_r, \texttt{Op}_1\,x\,k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\,x\,k \mapsto c_{\texttt{Op}_n}\}$$
$$\text{computation } c ::= \texttt{return}\,v \mid \texttt{Op}\,v\,(y:T.c) \mid \texttt{do}\,x \leftarrow c_1; c_2$$
$$\mid \; \texttt{handle}\,c\,\texttt{with}\,v \mid v_1\,v_2 \mid \texttt{let}\,x = v\,\texttt{in}\,c \mid c \triangleright \gamma$$

Types

$$\text{skeleton } \tau ::= \varsigma \mid \texttt{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \Rightarrow \tau_2 \mid \forall\varsigma.\tau$$

$$\text{value type } T ::= \alpha \mid \texttt{Unit} \mid T \rightarrow \underline{C} \mid \underline{C}_1 \Rightarrow \underline{C}_2 \mid \forall\varsigma.T \mid \forall\alpha:\tau.T \mid \forall\delta.T \mid \pi \Rightarrow T$$
$$\text{simple coercion type } \pi ::= T_1 \leqslant T_2 \mid \Delta_1 \leqslant \Delta_2$$
$$\text{coercion type } \rho ::= \pi \mid \underline{C}_1 \leqslant \underline{C}_2$$

$$\text{computation type } \underline{C} ::= T\,!\,\Delta$$
$$\text{dirt } \Delta ::= \delta \mid \emptyset \mid \{\texttt{Op}\} \cup \Delta$$

Coercions

$$\gamma ::= \omega \mid \langle\texttt{Unit}\rangle \mid \langle\alpha\rangle \mid \langle\Delta\rangle \mid \gamma_1 \rightarrow \gamma_2 \mid \gamma_1 \Rightarrow \gamma_2 \mid \emptyset_\Delta \mid \{\texttt{Op}\} \cup \gamma \mid \forall\varsigma.\gamma \mid \forall(\alpha:\tau)$$
$$.\gamma \mid \forall\delta.\gamma \mid \pi \Rightarrow \gamma \mid \gamma_1\,!\,\gamma_2$$

Fig. 5. EXEFF syntax.

4 The EXEFF language

4.1 Syntax

[Figure 5](#) presents EXEFF's syntax. EXEFF is a type theory akin to System F ([Girard *et al.*](#), [1989](#)), where every term encodes its own typing derivation. In essence, all abstractions and applications that are implicit in IMPEFF are made explicit in EXEFF via new syntactic forms. Additionally, EXEFF supports impredicative and higher-rank polymorphism, which is reflected in the lack of discrimination between value types, qualified types and type schemes; all non-computation types are denoted by *T*. While this design choice is not strictly required for the purpose at hand, it makes for a cleaner system.

In short, EXEFF relates to IMPEFF the same way that System F ([Girard](#), [1972](#); [Reynolds](#), [1974](#), [1983](#)) relates to the Hindley–Damas–Milner system ([Hindley](#), [1969](#); [Milner](#), [1978](#); [Damas & Milner](#), [1982](#)).

Coercions. Of particular interest is the use of explicit *subtyping coercions*, denoted by γ. EXEFF uses these to replace the implicit casts of IMPEFF (Rules TMCASTV and TMCASTC in [Figure 3](#)) with explicit casts $(v \triangleright \gamma)$ and $(c \triangleright \gamma)$. Essentially, coercions γ are explicit witnesses of subtyping derivations: each coercion form corresponds to a subtyping rule.

The first coercion form, ω, is a coercion variable, that is, a yet unknown proof of subtyping. Forms $\langle\texttt{Unit}\rangle$, $\langle\alpha\rangle$ and $\langle\Delta\rangle$ witness reflexivity for the Unit type, type variables and dirts Δ, respectively.

Most of the remaining coercion forms are simple congruences; subtyping for skeleton abstraction, type abstraction, dirt abstraction and qualification is witnessed by forms $\forall \varsigma.\gamma, \forall \alpha.\gamma, \forall \delta.\gamma$ and $\pi \Rightarrow \gamma$, respectively; similarly, syntactic forms $\gamma_1 \rightarrow \gamma_2$ and $\gamma_1 \Rightarrow \gamma_2$ capture injection for the arrow and the handler type constructor, respectively.

Subtyping for computation types is witnessed by coercion form $\gamma_1 ! \gamma_2$, which combines subtyping proofs of their components.

Finally, coercion forms $\emptyset_\Delta$ and $\{Op\} \cup \gamma$ are concerned with dirt subtyping. Form $\emptyset_\Delta$ witnesses that the empty dirt $\emptyset$ is a subdirt of any dirt Δ. Lastly, coercion form $\{Op\} \cup \gamma$ witnesses that subtyping between dirts is preserved under extension with a new operation.

A Note on Reflexivity of Arbitrary Types. In contrast to our earlier work (Saleh *et al.*, 2018), EXEFF (and the other calculi we present in the remainder of this paper) does not syntactically allow for reflexivity of arbitrary types. Nevertheless, we avoid notational burden and throughout the paper write $\langle T \rangle$ to denote the coercion that witnesses $T \leqslant T$; such a coercion can be built by traversing the structure of T (see Appendix B). A similar situation arises when applying a type substitution on a coercion, but it can be remedied in exactly the same way.

One of the problems with reflexivity of arbitrary types is that it allows for many trivially different proofs for the same constraint. The same is also true for *inversion coercions*, which are coercion formers that allow for decomposition of coercion types. For example, our earlier work (Saleh *et al.*, 2018) included a coercion former $\text{left}(\gamma)$ which is a proof of $T_2 \leqslant T_1$, if γ is a proof of $T_1 \rightarrow \underline{C}_1 \leqslant T_2 \rightarrow \underline{C}_2$.

By removing both, we have managed to greatly simplify the proofs of the metatheoretical properties of our calculi, since now there are much less proofs for any type inequality. Additionally, as we show in Section 4.3, EXEFF's operational semantics inspect the coercions so having uniqueness of proofs (coercions) is essential.

The situation is quite different when it comes to dirts. Dirts can take much less forms than types do (and so do coercions about them), and coercions regarding dirts need never be inspected during evaluation. Hence, we do not require unique coercion forms for dirt inequalities and can allow the simpler and more conventional reflexivity coercions $\langle \Delta \rangle$ for arbitrary dirts Δ.

4.2 Typing

Value and Computation Typing. Typing for EXEFF values and computations is presented in Figures 6 and 7 and is given by two mutually recursive relations of the form $\Gamma \vdash_v v : T$ (values) and $\Gamma \vdash_c c : \underline{C}$ (computations). EXEFF typing environments Γ contain bindings for variables of all sorts:

$$\Gamma ::= \varepsilon \mid \Gamma, \varsigma \mid \Gamma, \alpha : \tau \mid \Gamma, \delta \mid \Gamma, x : T \mid \Gamma, \omega : \pi$$

Typing is entirely syntax-directed. Apart from the typing rules for skeleton, type, dirt and coercion abstraction (and, subsequently, skeleton, type, dirt and coercion application), the main difference between typing for IMPEFF and EXEFF lies in the explicit cast forms, $(v \triangleright \gamma)$ and $(c \triangleright \gamma)$. Given that a value v has type T_1 and that γ is a proof that T_1 is a subtype of T_2, we can upcast v with an explicit cast operation $(v \triangleright \gamma)$. Upcasting for computations works analogously.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash_{\mathtt{v}} x : T} \qquad \overline{\Gamma \vdash_{\mathtt{v}} \mathtt{unit} : \mathtt{Unit}} \qquad \frac{\Gamma, x : T \vdash_{\mathtt{c}} c : \underline{C} \qquad \Gamma \vdash_T T : \tau}{\Gamma \vdash_{\mathtt{v}} (\mathtt{fun}\ x : T \mapsto c) : T \to \underline{C}}$$

$$\frac{\Gamma \vdash_{\mathtt{v}} v : T_1 \qquad \Gamma \vdash_{\mathtt{co}} \gamma : T_1 \leqslant T_2}{\Gamma \vdash_{\mathtt{v}} v \triangleright \gamma : T_2} \qquad \frac{\Gamma, \varsigma \vdash_{\mathtt{v}} v : T}{\Gamma \vdash_{\mathtt{v}} \Lambda \varsigma.v : \forall \varsigma.T} \qquad \frac{\Gamma, \alpha : \tau \vdash_{\mathtt{v}} v : T}{\Gamma \vdash_{\mathtt{v}} \Lambda \alpha : \tau.v : \forall \alpha : \tau.T}$$

$$\frac{\Gamma, \delta \vdash_{\mathtt{v}} v : T}{\Gamma \vdash_{\mathtt{v}} \Lambda \delta.v : \forall \delta.T} \qquad \frac{\Gamma, \omega : \pi \vdash_{\mathtt{v}} v : T \qquad \Gamma \vdash_\rho \pi}{\Gamma \vdash_{\mathtt{v}} \Lambda(\omega : \pi).v : \pi \Rightarrow T} \qquad \frac{\Gamma \vdash_{\mathtt{v}} v : \pi \Rightarrow T \qquad \Gamma \vdash_{\mathtt{co}} \gamma : \pi}{\Gamma \vdash_{\mathtt{v}} v\ \gamma : T}$$

$$\frac{\Gamma, x : T_x \vdash_{\mathtt{c}} c_r : T\ !\ \Delta \qquad [(\mathtt{Op} : T_1 \to T_2) \in \Sigma \qquad \Gamma, x : T_1, k : T_2 \to T\ !\ \Delta \vdash_{\mathtt{c}} c_{\mathtt{Op}} : T\ !\ \Delta]_{\mathtt{Op} \in \mathcal{O}}}{\Gamma \vdash_{\mathtt{v}} \{\mathtt{return}\ (x : T_x) \mapsto c_r, [\mathtt{Op}\ x\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : T_x\ !\ \Delta \cup \mathcal{O} \Rightarrow T\ !\ \Delta}$$

$$\frac{\Gamma \vdash_{\mathtt{v}} v : \forall \varsigma.T \qquad \Gamma \vdash_\tau \tau}{\Gamma \vdash_{\mathtt{v}} v\ \tau : T[\tau/\varsigma]} \qquad \frac{\Gamma \vdash_{\mathtt{v}} v : \forall \alpha : \tau.T_1 \qquad \Gamma \vdash_T T_2 : \tau}{\Gamma \vdash_{\mathtt{v}} v\ T_2 : T_1[T_2/\alpha]} \qquad \frac{\Gamma \vdash_{\mathtt{v}} v : \forall \delta.T \qquad \Gamma \vdash_\Delta \Delta}{\Gamma \vdash_{\mathtt{v}} v\ \Delta : T[\Delta/\delta]}$$

Fig. 6. EXEFF value typing.

$$\frac{\Gamma \vdash_{\mathtt{v}} v_1 : T \to \underline{C} \qquad \Gamma \vdash_{\mathtt{v}} v_2 : T}{\Gamma \vdash_{\mathtt{c}} v_1\ v_2 : \underline{C}} \qquad \frac{\Gamma \vdash_{\mathtt{v}} v : T \qquad \Gamma, x : T \vdash_{\mathtt{c}} c : \underline{C}}{\Gamma \vdash_{\mathtt{c}} \mathtt{let}\ x = v\ \mathtt{in}\ c : \underline{C}} \qquad \frac{\Gamma \vdash_{\mathtt{v}} v : T}{\Gamma \vdash_{\mathtt{c}} \mathtt{return}\ v : T\ !\ \emptyset}$$

$$\frac{\Gamma \vdash_{\mathtt{c}} c_1 : T_1\ !\ \Delta \qquad \Gamma, x : T_1 \vdash_{\mathtt{c}} c_2 : T_2\ !\ \Delta}{\Gamma \vdash_{\mathtt{c}} \mathtt{do}\ x \leftarrow c_1; c_2 : T_2\ !\ \Delta}$$

$$\frac{(\mathtt{Op} : T_1 \to T_2) \in \Sigma \qquad \Gamma \vdash_{\mathtt{v}} v : T_1 \qquad \Gamma, y : T_2 \vdash_{\mathtt{c}} c : T\ !\ \Delta \qquad \mathtt{Op} \in \Delta}{\Gamma \vdash_{\mathtt{c}} \mathtt{Op}\ v\ (y : T_2.c) : T\ !\ \Delta}$$

$$\frac{\Gamma \vdash_{\mathtt{v}} v : \underline{C}_1 \Rightarrow \underline{C}_2 \qquad \Gamma \vdash_{\mathtt{c}} c : \underline{C}_1}{\Gamma \vdash_{\mathtt{c}} \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{C}_2} \qquad \frac{\Gamma \vdash_{\mathtt{c}} c : \underline{C}_1 \qquad \Gamma \vdash_{\mathtt{co}} \gamma : \underline{C}_1 \leqslant \underline{C}_2}{\Gamma \vdash_{\mathtt{c}} c \triangleright \gamma : \underline{C}_2}$$

Fig. 7. EXEFF computation typing.

Well-formedness of Types, Constraints, Dirts and Skeletons. The definitions of the judgements that check the well-formedness of EXEFF value types ($\Gamma \vdash_T T : \tau$), computation types ($\Gamma \vdash_{\underline{C}} \underline{C} : \tau$), dirts ($\Gamma \vdash_\Delta \Delta$) and skeletons ($\Gamma \vdash_\tau \tau$) are equally straightforward as those for IMPEFF and can be found in Appendix B.

Coercion Typing. Coercion typing formalises the intuitive interpretation of coercions we gave in Section 4.1 and takes the form $\Gamma \vdash_{\mathtt{co}} \gamma : \rho$, defined in Appendix B. It is essentially an extension of the constraint entailment relation of Figure 4.

4.3 Operational semantics

Figure 8 presents selected rules of EXEFF's small-step, call-by-value operational semantics. For lack of space, we omit β-rules and other common rules and focus only on cases of interest. The complete operational semantics can be found in Appendix B.

$\boxed{v \leadsto_v v'}$ **Values**

$$\frac{v \leadsto_v v'}{v \rhd \gamma \leadsto_v v' \rhd \gamma} \text{ VCAST} \qquad \text{VPUSHUNIT} \atop v^R \rhd \langle \text{Unit} \rangle \leadsto_v v^R \qquad \text{VPUSHSKEL} \atop (v^R \rhd (\forall \varsigma. \gamma)) \tau \leadsto_v v^R \tau \rhd \gamma[\tau/\varsigma]$$

$$\text{VPUSHTY} \atop (v^R \rhd (\forall(\alpha : \tau). \gamma)) T \leadsto_v v^R T \rhd \gamma[T/\alpha] \qquad \text{VPUSHDIRT} \atop (v^R \rhd (\forall \delta. \gamma)) \Delta \leadsto_v v^R \Delta \rhd \gamma[\Delta/\delta]$$

$$\text{VPUSHQUAL} \atop (v^R \rhd (\pi \Rightarrow \gamma_1)) \gamma_2 \leadsto_v v^R \gamma_2 \rhd \gamma_1$$

$\boxed{c \leadsto_c c'}$ **Computations**

$$\frac{c \leadsto_c c'}{c \rhd \gamma \leadsto_c c' \rhd \gamma} \text{ CCAST} \qquad \text{CPUSHAPP} \atop (v^R \rhd (\gamma_1 \to \gamma_2)) v \leadsto_c (v^R (v \rhd \gamma_1)) \rhd \gamma_2$$

$$\text{CPUSHOP} \atop (\text{Op } v^R (x : T.c)) \rhd \gamma \leadsto_c \text{Op } v^R (x : T.(c \rhd \gamma))$$

$$\text{CDORET} \atop \text{do } x \leftarrow ((\text{return } v^R) \rhd (\gamma_1 \, ! \, \gamma_1') \rhd \ldots \rhd (\gamma_n \, ! \, \gamma_n')); c_2 \leadsto_c c_2[(v^R \rhd \gamma_1 \rhd \ldots \rhd \gamma_n)/x]$$

$$\text{CDOOP} \atop \text{do } x \leftarrow \text{Op } v^R (y : T.c_1); c_2 \leadsto_c \text{Op } v^R (y : T.\text{do } x \leftarrow c_1; c_2)$$

$$\text{CPUSHHANDLE} \atop \text{handle } c \text{ with } (v^R \rhd (\gamma_1 \Rightarrow \gamma_2)) \leadsto_c (\text{handle } (c \rhd \gamma_1) \text{ with } v^R) \rhd \gamma_2$$

$$\text{CHANDLERET} \atop \frac{(\text{return } x \mapsto c_r) \in h}{\text{handle } ((\text{return } v^R) \rhd (\gamma_1 \, ! \, \gamma_1') \rhd \ldots \rhd (\gamma_n \, ! \, \gamma_n')) \text{ with } h \leadsto_c c_r[(v^R \rhd \gamma_1 \rhd \ldots \rhd \gamma_n)/x]}$$

$$\frac{(\text{Op } x \, k \mapsto c_{\text{Op}}) \in h}{\text{handle } (\text{Op } v^R (y : T.c)) \text{ with } h \leadsto_c c_{\text{Op}}[v^R/x, (\text{fun } (y : T) \mapsto \text{handle } c \text{ with } h)/k]} \text{ CHANDLEOP1}$$

$$\frac{(\text{Op } x \, k \mapsto c_{\text{Op}}) \notin h}{\text{handle } (\text{Op } v^R (y : T.c)) \text{ with } h \leadsto_c \text{Op } v^R (y : T.\text{handle } c \text{ with } h)} \text{ CHANDLEOP2}$$

Fig. 8. EXEFF Operational semantics (selected rules).

Firstly, one of the non-conventional features of our system lies in the stratification of results in plain results and cast results:

$$\text{terminal value } v^T ::= \text{unit} \mid \text{fun } x : T \mapsto c \mid h \mid \Lambda \varsigma. v \mid \Lambda(\alpha : \tau). v \mid \Lambda \delta. v \mid \lambda(\omega : \pi). v$$
$$\text{value result } v^R ::= v^T \mid v^R \rhd (\gamma_1 \to \gamma_2) \mid v^R \rhd (\gamma_1 \Rightarrow \gamma_2) \mid v^R \rhd (\forall \varsigma. \gamma)$$
$$\mid v^R \rhd (\forall(\alpha : \tau). \gamma) \mid v^R \rhd (\forall \delta. \gamma) \mid v^R \rhd (\pi \Rightarrow \gamma)$$
$$\text{terminal computation } c^T ::= \text{return } v^R \mid c^T \rhd (\gamma_1 \, ! \, \gamma_2)$$
$$\text{computation result } c^R ::= c^T \mid \text{Op } v^R (y : T.c)$$

Terminal values v^T represent conventional values, and value results v^R can either be plain terminal values v^T or cast value results, where we exclude reflexivity coercions, as those can be further reduced. This stratification can also be found in Henglein's coercion calculus (Henglein, 1994), Crary's coercion calculus for inclusive subtyping (Crary, 2000), and, more recently, in System F_C (Sulzmann *et al.*, 2007).

Computations evaluate either to a returned value or an operation call. Both can be further cast, though we are able to delegate any coercion on the operation call to its continuation, leading to a slightly different stratification than in values. The same is not true for returned values. Consider for example the expression $(\texttt{return } 5 \triangleright \langle \texttt{Int}\rangle \,!\, \emptyset_{\{\texttt{Op}\}})$, of type $\texttt{Int}\,!\,\{\texttt{Op}\}$. We cannot reduce the expression further without losing effect information; removing the cast would result in computation $(\texttt{return } 5)$, of type $\texttt{Int}\,!\,\emptyset$. Even if we consider type preservation only up to subtyping, the redex may still occur as a subterm in a context that expects solely the larger type.

Secondly, we need to make sure that casts do not stand in the way of evaluation. This is captured in the so-called "push" rules, all of which appear in Figure 8.

In relation $v \rightsquigarrow_v v'$, Rule VCAST evaluates under the coercion, while the rest are push rules: whenever a redex is "blocked" due to a cast, we take the coercion apart and redistribute it (in a type-preserving manner) over the subterms, so that evaluation can progress.

Example 4.1. Consider the evaluation of $(((\Lambda\alpha.v) \triangleright (\forall\alpha.\gamma))\ T)$ (we elide skeleton annotations for clarity; they are orthogonal to the task at hand). The evaluation is "blocked" because of the type cast; in order to expose the redex $((\Lambda\alpha.v)\ T)$, we need to *push* the coercion outside the redex, which we achieve using Rule VPUSHTY:

$$((\Lambda\alpha.v) \triangleright (\forall\alpha.\gamma))\ T \rightsquigarrow_v ((\Lambda\alpha.v)\ T) \triangleright \gamma[T/\alpha]$$

Since the type cast now happens after the instantiation, we change the coercion accordingly (to $\gamma[T/\alpha]$), to ensure that the type of the expression remains the same as before (preservation). Now using Rule CCAST, we can continue with the evaluation of the redex under the cast, thus obtaining:

$$((\Lambda\alpha.v)\ T) \triangleright \gamma[T/\alpha] \rightsquigarrow_v v \triangleright \gamma[T/\alpha]$$

The rest of the push rules behave similarly.

The situation in relation $c \rightsquigarrow_c c'$ is quite similar. Rule CCAST continues evaluating the computation under the coercion. Rule CPUSHAPP is a push rule for function application. Rule CPUSHOP pushes a coercion inside an operation computation, illustrating why the syntax for c^R does not require casts on operation computations; we can always push the casts inside the continuation. Rule CDORET is a β-reduction for sequencing and performs two tasks at once. Since we know that the computation bound to x calls no operations, we (a) safely "drop" the impure part of coercions, and (b) substitute x with v^R, cast with the pure part of coercions (so that types are preserved). Rule CDOOP handles operation calls in sequencing computations. If an operation is called in a sequencing computation, evaluation is suspended and the rest of the computation is captured in the continuation.

The last four rules are concerned with effect handling. Rule CPUSHHANDLE pushes a coercion on the handler "outwards", such that the handler can be exposed and evaluation

is not stuck (similarly to the push rule for term application). Rule CHANDLERET behaves similarly to the push/beta rule for sequencing computations. Finally, the last two rules are concerned with handling of operations. Rule CHANDLEOP1 captures cases where the called operation is handled by the handler, in which case the respective clause of the handler is called. As illustrated by the rule, like Pretnar (2014), EXEFF features *deep handlers*: the continuation is also wrapped within a `with-handle` construct. Rule CHANDLEOP2 captures cases where the operation is not covered by the handler and thus remains unhandled.

We have shown that EXEFF is type-safe:

Theorem 4.2 (Type Safety)**.**

- *If $\Gamma \vdash_{\bar{v}} v : T$, then either v is a result value or $v \rightsquigarrow_v v'$ and $\Gamma \vdash_{\bar{v}} v' : T$.*
- *If $\Gamma \vdash_{\bar{c}} c : \underline{C}$, then either c is a result computation or $c \rightsquigarrow_c c'$ and $\Gamma \vdash_{\bar{c}} c' : \underline{C}$.*

5 Type inference and elaboration

This section presents the typing-directed elaboration of IMPEFF into EXEFF. This elaboration makes all the implicit type and effect information explicit and introduces explicit term-level coercions to witness the use of subtyping.

After covering the declarative specification of this elaboration, we present a constraint-based algorithm to infer IMPEFF types and at the same time elaborate into EXEFF. This algorithm alternates between two phases: (1) the syntax-directed generation of constraints from the IMPEFF term and (2) solving these constraints.

5.1 Elaboration of IMPEFF into EXEFF

The greyed parts of Figure 3 augment the typing rules for IMPEFF value and computation terms with typing-directed elaboration to corresponding EXEFF terms. The elaboration is mostly straightforward, mapping every IMPEFF construct onto its corresponding EXEFF construct while adding explicit type annotations to binders in Rules TMTMABS, TMHANDLER and TMOP. Implicit appeals to subtyping are turned into explicit casts with coercions in Rules TMCASTV and TMCASTC. Rule TMLET introduces explicit binders for skeleton, type and dirt variables, as well as for constraints. These last also introduce coercion variables ω that can be used in casts.

Binders introduced by Rule TMLET are eliminated in Rule TMVAR by means of explicit application with skeletons, types, dirts and coercions. The coercions are produced by the auxiliary judgement $\Gamma \vdash_{co} \gamma : \pi$, defined in Figure 4, which provides a coercion witness for every subtyping proof.

As a sanity check, we have shown that elaboration preserves types.

Theorem 5.1 (Type Preservation)**.**

- *If $\Gamma \vdash_v v : A \rightsquigarrow v'$, then $elab_\Gamma(\Gamma) \vdash_{\bar{v}} v' : elab_S(A)$.*
- *If $\Gamma \vdash_c c : \underline{C} \rightsquigarrow c'$, then $elab_\Gamma(\Gamma) \vdash_{\bar{c}} c' : elab_{\underline{C}}(\underline{C})$.*

Here $elab_\Gamma(\Gamma)$, $elab_S(A)$ and $elab_C(C)$ convert IMPEFF environments and types into EXEFF environments and types; they are defined in Appendix C.

Example 5.2. A valid elaboration of the polymorphic expression

$$\texttt{let } f = (\texttt{fun } g \mapsto g \texttt{ unit}) \texttt{ in } \dots$$

from Example 3.1 can be

$$
\begin{aligned}
\texttt{let } f \; &: \; (\texttt{Unit} \to \texttt{Unit} \, ! \, \emptyset) \to \texttt{Unit} \, ! \, \emptyset \\
&= \texttt{fun} \, (g : \texttt{Unit} \to \texttt{Unit} \, ! \, \emptyset) \mapsto g \texttt{ unit} \\
\texttt{in } &\dots
\end{aligned}
$$

if the simple monomorphic typing is used (we have included the signature of f for clarity). For the polymorphic variant, the elaboration features both type-level abstractions and explicit casts:

$$
\begin{aligned}
\texttt{let } f \; &: \; \forall\varsigma.\forall\alpha:\varsigma.\forall\alpha':\varsigma.\forall\delta.\forall\delta'.(\alpha\leqslant\alpha')\Rightarrow(\delta\leqslant\delta')\Rightarrow(\texttt{Unit}\to\alpha\,!\,\delta)\to\alpha'\,!\,\delta' \\
&= \Lambda\varsigma.\Lambda(\alpha:\varsigma).\Lambda(\alpha':\varsigma).\Lambda\delta.\Lambda\delta'.\Lambda(\omega:\alpha\leqslant\alpha').\Lambda(\omega':\delta\leqslant\delta'). \\
&\qquad \texttt{fun} \, (g:\texttt{Unit}\to\alpha\,!\,\delta)\mapsto((g\texttt{ unit})\rhd(\omega\,!\,\omega')) \\
\texttt{in } &\dots
\end{aligned}
$$

Here, coercion variables ω and ω' are utilised by the body of f for upcasting $(g\texttt{ unit})$ to have type $\alpha'\,!\,\delta'$.

Similarly, applications of the latter variant need to include explicit type-level applications and coercion witnesses. Elaborating the application of f to the pure function id, we get

$$f \texttt{ Unit Unit Unit } \emptyset \, \emptyset \, \langle\texttt{Unit}\rangle \, \emptyset_\emptyset \, (\texttt{fun} \, (x:\texttt{Unit})\mapsto\texttt{return } x)$$

whereas for the impure *tick* at a type $\texttt{Unit} \, ! \, \{\texttt{Tick}, \texttt{Tock}\}$, we get

$$
\begin{aligned}
f &\texttt{ Unit Unit Unit } \{\texttt{Tick}\} \, \{\texttt{Tick}, \texttt{Tock}\} \, \langle\texttt{Unit}\rangle \, (\{\texttt{Tick}\}\cup\emptyset_{\{\texttt{Tock}\}}) \\
&(\texttt{fun} \, x:\texttt{Unit} \mapsto \texttt{Tick } x \, (y:\texttt{Unit}.((\texttt{return } y)\rhd\langle\texttt{Unit}\rangle\,!\,\emptyset_{\{\texttt{Tick}\}})))
\end{aligned}
$$

where $\texttt{return}$ had to be coerced in order to match the dirt of the operation call.

5.2 Constraint generation and elaboration

Constraint generation with elaboration into EXEFF is presented in Figures 9 (values) and 10 (computations). Before going into the details of each, we first introduce the three auxiliary constructs they use.

$$
\begin{aligned}
\text{constraint set } \mathcal{P}, \mathcal{Q} &::= \bullet \mid \tau_1 = \tau_2, \mathcal{P} \mid \alpha:\tau, \mathcal{P} \mid \boxed{\omega:\pi, \mathcal{P}} \\
\text{typing environment } \Gamma &::= \varepsilon \mid \Gamma, x:S \\
\text{substitution } \sigma &::= \bullet \mid \sigma\cdot[\tau/\varsigma] \mid \sigma\cdot[A/\alpha] \mid \sigma\cdot[\Delta/\delta] \mid \sigma\cdot\boxed{[\gamma/\omega]}
\end{aligned}
$$

At the heart of our algorithm are sets $\mathcal{P}$, containing three different kinds of constraints: (a) skeleton equalities of the form $\tau_1 = \tau_2$, (b) skeleton constraints of the form $\alpha:\tau$ and (c)

$$\boxed{\mathcal{Q}; \Gamma \vdash_{\overline{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'} \quad \textbf{Values}$$

$$\frac{(x : \forall \bar{\varsigma}.\overline{\alpha : \tau}.\forall \bar{\delta}.\bar{\pi} \Rightarrow A) \in \Gamma \qquad \sigma = [\overline{\varsigma'/\varsigma}, \overline{\alpha'/\alpha}, \overline{\delta'/\delta}]}{\mathcal{Q}; \Gamma \vdash_{\overline{v}} x : \sigma(A) \mid \overline{\omega : \sigma(\pi)}, \overline{\alpha' : \sigma(\tau)}, \mathcal{Q}; \bullet \rightsquigarrow x \; \bar{\varsigma}' \; \bar{\alpha}' \; \bar{\delta}' \; \bar{\omega}} \; \text{TMVAR}$$

$$\frac{}{\mathcal{Q}; \Gamma \vdash_{\overline{v}} \texttt{unit} : \texttt{Unit} \mid \mathcal{Q}; \bullet \rightsquigarrow \texttt{unit}} \; \text{TMUNIT}$$

$$\frac{\alpha : \varsigma, \mathcal{Q}; \Gamma, x : \alpha \vdash_{\overline{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow c'}{\mathcal{Q}; \Gamma \vdash_{\overline{v}} (\texttt{fun } x \mapsto c) : \sigma(\alpha) \to \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow \texttt{fun } x : \sigma(\alpha) \mapsto c'} \; \text{TMABS}$$

$$\begin{array}{c}
\alpha_r : \varsigma_r, \mathcal{Q}; \Gamma, x : \alpha_r \vdash_{\overline{c}} c_r : B_r \mathbin{!} \Delta_r \mid \mathcal{Q}_0; \sigma_r \rightsquigarrow c'_r \qquad \sigma^i = \sigma_i \cdot \sigma_{i-1} \cdot \ldots \cdot \sigma_1 \\
\mathtt{Op}_i \in \mathcal{O}: \\
\quad (\mathtt{Op}_i : A_i \to B_i) \in \Sigma \\
\quad \alpha_i : \varsigma_i, \mathcal{Q}_{i-1}; \sigma^{i-1}(\sigma_r(\Gamma)), x : A_i, k : B_i \to \alpha_i \mathbin{!} \delta_i \vdash_{\overline{c}} c_{\mathtt{Op}_i} : B_{\mathtt{Op}_i} \mathbin{!} \Delta_{\mathtt{Op}_i} \mid \mathcal{Q}_i; \sigma_i \rightsquigarrow c'_{\mathtt{Op}_i} \\
\mathcal{Q}' = \mathcal{Q}_n, \alpha_{in} : \varsigma_{in}, \alpha_{out} : \varsigma_{out}, \\
\qquad \omega_1 : \sigma^n(B_r) \leqslant \alpha_{out}, \\
\qquad \omega_2 : \sigma^n(\Delta_r) \leqslant \delta_{out}, \\
\qquad \omega_{3_i} : \sigma^n(B_{\mathtt{Op}_i}) \leqslant \alpha_{out}, \qquad\qquad (\forall i \in [1 \ldots n]) \\
\qquad \omega_{4_i} : \sigma^n(\Delta_{\mathtt{Op}_i}) \leqslant \delta_{out}, \qquad\qquad (\forall i \in [1 \ldots n]) \\
\qquad \omega_{5_i} : B_i \to \alpha_{out} \mathbin{!} \delta_{out} \leqslant B_i \to \sigma^n(\alpha_i \mathbin{!} \delta_i), \qquad (\forall i \in [1 \ldots n]) \\
\qquad \omega_6 : \alpha_{in} \leqslant \sigma^n(\sigma_r(\alpha_r)), \\
\qquad \omega_7 : \delta_{in} \leqslant \delta_{out} \cup \mathcal{O} \\
c_{res} = \{\, \texttt{return } y : \sigma^n(\sigma_r(\alpha_r)) \mapsto \sigma^n(c'_r)[y \triangleright \omega_6/x] \triangleright \omega_1 \mathbin{!} \omega_2 \\
\qquad , \left[\mathtt{Op}_i \; x \; l \mapsto \sigma^n(c'_{\mathtt{Op}_i})[l \triangleright \omega_{5_i}/k] \triangleright \omega_{3_i} \mathbin{!} \omega_{4_i}\right]_{\mathtt{Op}_i \in \mathcal{O}} \} \triangleright (\langle \alpha_{in} \rangle \mathbin{!} \omega_7 \Rightarrow \langle \alpha_{out} \rangle \mathbin{!} \langle \delta_{out} \rangle) \\
\hline
\mathcal{Q}; \Gamma \vdash_{\overline{v}} \{\texttt{return } x \mapsto c_r, [\mathtt{Op} \; x \; k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : \alpha_{in} \mathbin{!} \delta_{in} \Rightarrow \alpha_{out} \mathbin{!} \delta_{out} \mid \mathcal{Q}'; (\sigma^n \cdot \sigma_r) \rightsquigarrow c_{res}
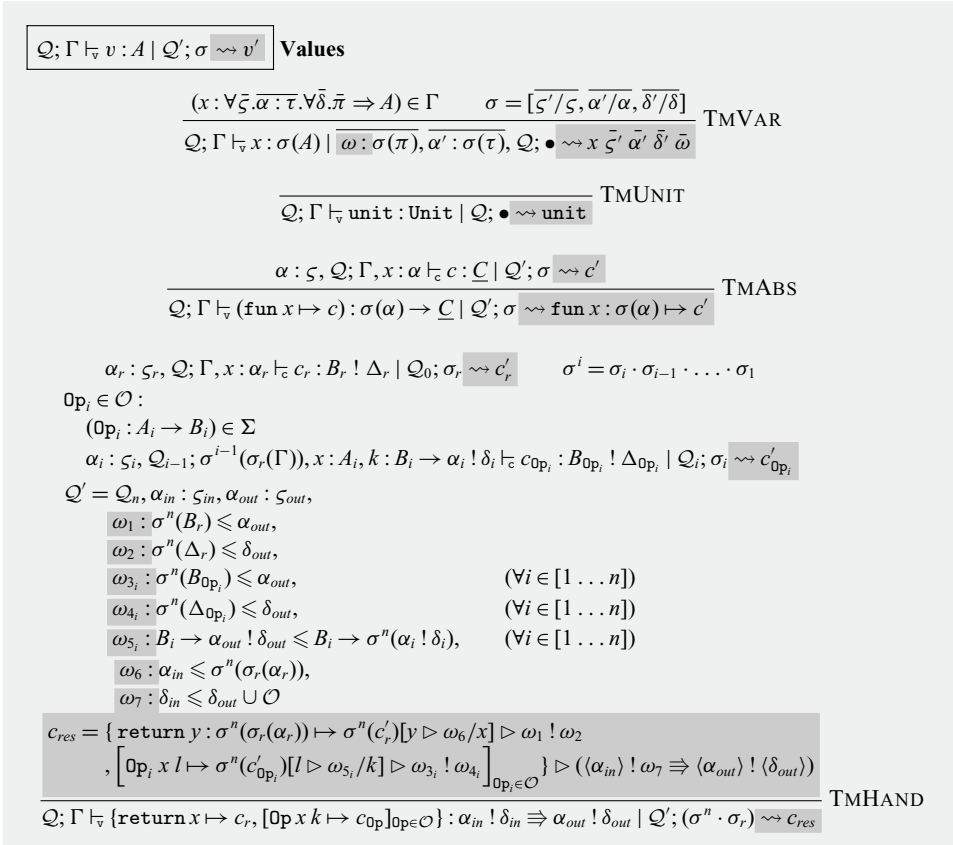\end{array} \; \text{TMHAND}$$

Fig. 9. Constraint generation with elaboration (values).

wanted subtyping constraints of the form $\omega : \pi$. The purpose of the first two becomes clear when we discuss constraint solving, in Section 5.3. Next, typing environments Γ only contain term variable bindings, while other variables represent unknowns of their sort and may end up being instantiated after constraint solving. Finally, during type inference we compute substitutions σ, for refining as of yet unknown skeletons, types, dirts and coercions. The last one is essential, since our algorithm simultaneously performs type inference and elaboration into EXEFF.

Values. Constraint generation for values takes the form $\mathcal{Q}; \Gamma \vdash_{\overline{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'$. It takes as inputs a set of wanted constraints $\mathcal{Q}$, a typing environment Γ and a IMPEFF value v and produces a value type A, a new set of wanted constraints $\mathcal{Q}'$, a substitution σ and a EXEFF value v'.

In order to support let generalisation, our inference algorithm does not keep constraint generation and solving separate. Instead, the two are interleaved, as indicated by the additional arguments of our relation: (a) constraints $\mathcal{Q}$ are passed around in a stateful manner (i.e., they are input and output), and (b) substitutions σ generated from constraint solving constitute part of the relation output.

$\boxed{\mathcal{Q}; \Gamma \vdash_{\bar{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow c'}$ **Computations**

$$\frac{\mathcal{Q}; \Gamma \vdash_{\bar{v}} v_1 : A_1 \mid \mathcal{Q}_1; \sigma_1 \rightsquigarrow v_1' \qquad \mathcal{Q}_1; \sigma_1(\Gamma) \vdash_{\bar{v}} v_2 : A_2 \mid \mathcal{Q}_2; \sigma_2 \rightsquigarrow v_2'}{\mathcal{Q}; \Gamma \vdash_{\bar{c}} v_1\, v_2 : \alpha\,!\,\delta \mid \alpha : \varsigma,\; \boxed{\omega : \sigma_2(A_1) \leqslant A_2 \rightarrow \alpha\,!\,\delta},\; \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \rightsquigarrow (\sigma_2(v_1') \triangleright \omega)\, v_2'} \text{ TMApp}$$

$$\frac{\mathcal{Q}; \Gamma \vdash_{\bar{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'}{\mathcal{Q}; \Gamma \vdash_{\bar{c}} \texttt{return } v : A\,!\,\emptyset \mid \mathcal{Q}'; \sigma \rightsquigarrow \texttt{return } v'} \text{ TMReturn}$$

$$\frac{\begin{array}{c} \mathcal{Q}; \Gamma \vdash_{\bar{v}} v : A \mid \mathcal{Q}_v; \sigma_1 \rightsquigarrow v' \\ \texttt{solve}(\bullet;\; \bullet;\; \mathcal{Q}_v) = (\sigma_1', \mathcal{Q}_v') \qquad \mathit{split}(\sigma_1'(\sigma_1(\Gamma)), \mathcal{Q}_v', \sigma_1'(A)) = \langle \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \overline{\omega : \pi}, \mathcal{Q}_1 \rangle \\ \mathcal{Q}_1; \sigma_1'(\sigma_1(\Gamma)), x : \forall \bar{\varsigma}.\forall \overline{\alpha : \tau}.\forall \bar{\delta}.\bar{\pi} \Rightarrow \sigma_1'(A) \vdash_{\bar{c}} c : \underline{C} \mid \mathcal{Q}_2; \sigma_2 \rightsquigarrow c' \\ c_{res} = \texttt{let } x = \sigma_2(\Lambda \bar{\varsigma}.\Lambda \overline{\alpha : \tau}.\Lambda \bar{\delta}.\Lambda \overline{(\omega : elab_\rho(\pi))}.v') \texttt{ in } c' \end{array}}{\mathcal{Q}; \Gamma \vdash_{\bar{c}} \texttt{let } x = v \texttt{ in } c : \underline{C} \mid \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1' \cdot \sigma_1) \rightsquigarrow c_{res}} \text{ TMLet}$$

$$\frac{\begin{array}{c} \mathcal{Q}; \Gamma \vdash_{\bar{v}} v : A_1 \mid \mathcal{Q}_1; \sigma_1 \rightsquigarrow v' \qquad \mathcal{Q}_1; \sigma_1(\Gamma), y : B_{0p} \vdash_{\bar{c}} c : A_2\,!\,\Delta_2 \mid \mathcal{Q}_2; \sigma_2 \rightsquigarrow c' \\ (\texttt{Op} : A_{0p} \rightarrow B_{0p}) \in \Sigma \qquad c_{res} = \texttt{Op } (\sigma_2(v') \triangleright \omega)\ (y : elab_s(B_{0p}).c') \end{array}}{\mathcal{Q}; \Gamma \vdash_{\bar{c}} \texttt{Op } v\ (y : B_{0p}.c) : A_2\,!\,\{\texttt{Op}\} \cup \Delta_2 \mid \boxed{\omega : \sigma_2(A_1) \leqslant A_{0p}},\; \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \rightsquigarrow c_{res}} \text{ TMOp}$$

$$\frac{\begin{array}{c} \mathcal{Q}; \Gamma \vdash_{\bar{c}} c_1 : A_1\,!\,\Delta_1 \mid \mathcal{Q}_1; \sigma_1 \rightsquigarrow c_1' \qquad \mathcal{Q}_1; \sigma_1(\Gamma), x : A_1 \vdash_{\bar{c}} c_2 : A_2\,!\,\Delta_2 \mid \mathcal{Q}_2; \sigma_2 \rightsquigarrow c_2' \\ c_{res} = \texttt{do } x \leftarrow (\sigma_2(c_1') \triangleright \langle \sigma_2(A_1) \rangle\,!\,\omega_1); (c_2' \triangleright \langle A_2 \rangle\,!\,\omega_2) \end{array}}{\mathcal{Q}; \Gamma \vdash_{\bar{c}} \texttt{do } x \leftarrow c_1; c_2 : A_2\,!\,\delta \mid \boxed{\omega_1 : \sigma_2(\Delta_1) \leqslant \delta},\; \boxed{\omega_2 : \Delta_2 \leqslant \delta},\; \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \rightsquigarrow c_{res}} \text{ TMDo}$$

$$\frac{\begin{array}{c} \mathcal{Q}; \Gamma \vdash_{\bar{v}} v : A_1 \mid \mathcal{Q}_1; \sigma_1 \rightsquigarrow v' \qquad \mathcal{Q}_1; \sigma_1(\Gamma) \vdash_{\bar{c}} c : A_2\,!\,\Delta_2 \mid \mathcal{Q}_2; \sigma_2 \rightsquigarrow c' \\ \mathcal{Q}' = \alpha_1 : \varsigma_1, \alpha_2 : \varsigma_2,\; \boxed{\omega_1 : \sigma_2(A_1) \leqslant (\alpha_1\,!\,\delta_1 \Rightarrow \alpha_2\,!\,\delta_2)},\; \boxed{\omega_2 : A_2 \leqslant \alpha_1},\; \boxed{\omega_3 : \Delta_2 \leqslant \delta_1},\; \mathcal{Q}_2 \\ c_{res} = \texttt{handle } (c' \triangleright (\omega_2\,!\,\omega_3)) \texttt{ with } (\sigma_2(v') \triangleright \omega_1) \end{array}}{\mathcal{Q}; \Gamma \vdash_{\bar{c}} \texttt{handle } c \texttt{ with } v : \alpha_2\,!\,\Delta_2 \mid \mathcal{Q}'; (\sigma_2 \cdot \sigma_1) \rightsquigarrow c_{res}} \text{ TMHandle}$$

Fig. 10. Constraint generation with elaboration (computations).

The rules are syntax-directed on the input IMPEFF value. Rule TMVAR handles term variables x: as usual for constraint-based type inference the rule instantiates the polymorphic type ($\forall \bar{\varsigma}.\overline{\alpha : \tau}.\forall \bar{\delta}.\bar{\pi} \Rightarrow A$) of x with fresh variables; these are placeholders that are determined during constraint solving. Moreover, the rule extends the wanted constraints $\mathcal{P}$ with $\bar{\pi}$, appropriately instantiated. In EXEFF, this corresponds to explicit skeleton, type, dirt and coercion applications.

More interesting is Rule TMABS, which handles term abstractions. Like in standard Hindley–Damas–Milner (Damas & Milner, 1982), it generates a fresh type variable α for the type of the abstracted term variable x. In addition, it generates a fresh skeleton variable ς, to capture the (yet unknown) shape of α.

As explained in detail in Section 5.3, the constraint solver instantiates type variables only through their skeletons annotations. Because we want to allow local constraint solving for the body c of the term abstraction the opportunity to produce a substitution σ that instantiates α, we have to pass in the annotation constraint $\alpha : \varsigma$, which hints at why we need to pass constraints in a stateful manner. We apply the resulting substitution σ to the

result type $\sigma(\alpha) \to \underline{C}$ (though σ refers to IMPEFF types, we abuse notation to save clutter and apply it directly to EXEFF entities too).

Finally, Rule TMHAND is concerned with handlers. Since it is the most complex of the rules, we discuss each of its premises separately:

Firstly, we infer a type $B_r ! \Delta_r$ for the right-hand side of the `return`-clause. Since α_r is a fresh unification variable, just like for term abstraction we require $\alpha_r : \varsigma_r$, for a fresh skeleton variable ς_r.

Secondly, we check every operation clause in $\mathcal{O}$ in order. For each clause, we generate fresh skeleton, type and dirt variables (ς_i, α_i and δ_i), to account for the (yet unknown) result type $\alpha_i ! \delta_i$ of the continuation k, while inferring type $B_{\mathtt{Op}_i} ! \Delta_{\mathtt{Op}_i}$ for the right-hand side $c_{\mathtt{Op}_i}$.

More interesting is the (final) set of wanted constraints $\mathcal{Q}'$. First, we assign to the handler the overall type

$$\alpha_{in} ! \delta_{in} \Rightarrow \alpha_{out} ! \delta_{out}$$

where $\varsigma_{in}, \alpha_{in}, \delta_{in}, \varsigma_{out}, \alpha_{out}, \delta_{out}$ are fresh variables of the respective sorts. In turn, we require that (a) the type of the return clause is a subtype of $\alpha_{out} ! \delta_{out}$ (given by the combination of ω_1 and ω_2), (b) the right-hand-side type of each operation clause is a subtype of the overall result type: $\sigma^n(B_{\mathtt{Op}_i} ! \Delta_{\mathtt{Op}_i}) \leqslant \alpha_{out} ! \delta_{out}$ (witnessed by $\omega_{3_i} ! \omega_{4_i}$), (c) the actual types of the continuations $B_i \to \alpha_{out} ! \delta_{out}$ in the operation clauses should be subtypes of their assumed types $B_i \to \sigma^n(\alpha_i ! \delta_i)$ (witnessed by ω_{5_i}), (d) the overall argument type α_{in} is a subtype of the assumed type of x: $\sigma^n(\sigma_r(\alpha_r))$ (witnessed by ω_6), and (e) the input dirt set δ_{in} is a subtype of the resulting dirt set δ_{out}, extended with the handled operations $\mathcal{O}$ (witnessed by ω_7).

All the aforementioned implicit subtyping relations become explicit in the elaborated term c_{res}, via explicit casts.

Computations. The judgement $\mathcal{Q}; \Gamma \vdash_c c : \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow c'$ generates constraints for computations.

Rule TMAPP handles term applications of the form $v_1 \, v_2$. After inferring a type for each subterm (A_1 for v_1 and A_2 for v_2), we generate the wanted constraint $\sigma_2(A_1) \leqslant A_2 \to \alpha ! \delta$, with fresh type and dirt variables α and δ, respectively. Associated coercion variable ω is then used in the elaborated term to explicitly (up)cast v_1' to the expected type $A_2 \to \alpha ! \delta$.

Rule TMRETURN handles `return`-computations and is entirely straightforward.

Rule TMLET handles polymorphic let-bindings. First, we infer a type A for v, as well as wanted constraints $\mathcal{Q}_v$. Then, we simplify wanted constraints $\mathcal{Q}_v$ by means of function `solve` (which we explain in detail in [Section 5.3](#) below), obtaining a substitution σ_1' and a set of *residual constraints* $\mathcal{Q}_v'$.

Generalisation of x's type is performed by the auxiliary function *split*, given by the following clause:

$$\bar{\varsigma} = \{\varsigma \mid (\alpha : \varsigma) \in \mathcal{Q}, \nexists \alpha'.\alpha' \notin \bar{\alpha} \wedge (\alpha' : \varsigma) \in \mathcal{Q}\}$$
$$\bar{\alpha} = fv_\alpha(\mathcal{Q}) \cup fv_\alpha(A) \setminus fv_\alpha(\Gamma) \qquad \mathcal{Q}_1 = \{(\omega : \pi) \mid (\omega : \pi) \in \mathcal{Q}, fv(\pi) \nsubseteq fv(\Gamma)\}$$
$$\bar{\delta} = fv_\delta(\mathcal{Q}) \cup fv_\delta(A) \setminus fv_\delta(\Gamma) \qquad \mathcal{Q}_2 = \mathcal{Q} - \mathcal{Q}_1$$
$$\overline{\rule{12cm}{0.4pt}}$$
$$split(\Gamma, \mathcal{Q}, A) = \langle \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \mathcal{Q}_1, \mathcal{Q}_2 \rangle$$

In essence, *split* generates the type (scheme) of x in parts. Additionally, it computes the subset Q_2 of the input constraints Q that do not depend on locally bound variables. Such constraints can be floated "upwards" and are passed as input when inferring a type for c. The remainder of the rule is self-explanatory.

Rule TMOP handles operation calls. Observe that in the elaborated term, we upcast the inferred type to match the expected type in the signature.

Rule TMDO handles sequences. The requirement that all computations in a do-construct have the same dirt set is expressed in the wanted constraints $\sigma_2(\Delta_1) \leqslant \delta$ and $\Delta_2 \leqslant \delta$ (where δ is a fresh dirt variable; the resulting dirt set), witnessed by coercion variables ω_1 and ω_2. Both coercion variables are used in the elaborated term to upcast c_1 and c_2, such that both draw effects from the same dirt set δ.

Finally, Rule TMHANDLE is concerned with effect handling. After inferring type A_1 for the handler v, we require that it takes the form of a handler type, witnessed by coercion variable $\omega_1 : \sigma_2(A_1) \leqslant (\alpha_1 ! \delta_1 \Rightarrow \alpha_2 ! \delta_2)$, for fresh $\alpha_1, \alpha_2, \delta_1, \delta_2$. To ensure that the type $A_2 ! \Delta_2$ of c matches the expected type, we require that $A_2 ! \Delta_2 \leqslant \alpha_1 ! \delta_1$. Our syntax does not include coercion variables for computation subtyping; we achieve the same effect by combining $\omega_2 : A_2 \leqslant \alpha_1$ and $\omega_3 : \Delta_2 \leqslant \delta_1$.

In the following, notation $\sigma \models Q$ denotes that the substitution σ is a *solution* of the constraint set Q, that is, when after applying σ to all constraints in Q, we get derivable judgements according to rules of Figure 4.

Theorem 5.3 (Soundness of Inference). *If $\bullet ; \Gamma \vdash_{\bar{v}} v : A \mid Q ; \sigma \rightsquigarrow v'$, then for any $\sigma' \models Q$, we have $(\sigma' \cdot \sigma)(\Gamma) \vdash_v v : \sigma'(A) \rightsquigarrow \sigma'(v')$, and analogously for computations.*

Theorem 5.4 (Completeness of Inference). *If $\Gamma \vdash_v v : A \rightsquigarrow v'$, then we have $\bullet ; \Gamma \vdash_{\bar{v}} v : A' \mid Q ; \sigma \rightsquigarrow v''$ and there exists $\sigma' \models Q$ and γ, such that $\sigma'(v'') = v'$ and $\sigma(\Gamma) \vdash_{co} \gamma : \sigma'(A') \leqslant A$. An analogous statement holds for computations.*

5.3 Constraint solving

The second phase of our inference-and-elaboration algorithm is the constraint solver. It is defined by the `solve` function signature:

$$\texttt{solve}(\sigma ; \mathcal{P} ; Q) = (\sigma', \mathcal{P}')$$

It takes three inputs: the substitution σ accumulated so far, a list of already processed constraints $\mathcal{P}$ and a queue of still to be processed constraints Q. There are two outputs: the substitution σ' that solves the constraints and the residual constraints $\mathcal{P}'$. The substitutions σ and σ' contain four kinds of mappings: $\varsigma \mapsto \tau$, $\alpha \mapsto A$, $\delta \mapsto \Delta$ and $\omega \rightarrow \gamma$ which respectively instantiate skeleton variables, type variables, dirt variables and coercion variables.

Theorem 5.5 (Correctness of Solving). *For any set Q, the call $\texttt{solve}(\bullet ; \bullet ; Q)$ either results in a failure, in which case Q has no solutions, or returns $(\sigma, \mathcal{P})$ such that for any $\sigma' \models Q$, there exists $\sigma'' \models \mathcal{P}$ such that $\sigma' = \sigma'' \cdot \sigma$.*

The solver is invoked with $\texttt{solve}(\bullet; \bullet; \mathcal{Q})$, to process the constraints $\mathcal{Q}$ generated in the first phase of the algorithm, that is, with an empty substitution and no processed constraints. The $\texttt{solve}$ function is defined by case analysis on the queue.

Empty Queue. When the queue is empty, all constraints have been processed. What remains are the residual constraints and the solving substitution σ, which are both returned as the result of the solver.

$$\texttt{solve}(\sigma; \mathcal{P}; \bullet) = (\sigma, \mathcal{P})$$

Skeleton Equalities. The next set of cases we consider are those where the queue is non-empty and its first element is an equality between skeletons $\tau_1 = \tau_2$. We consider seven possible cases based on the structure of τ_1 and τ_2 that together essentially implement conventional unification as used in Hindley–Milner type inference (Damas & Milner, 1982).

```
solve(σ; 𝒫; τ₁ = τ₂, 𝒬) = match τ₁ = τ₂ with
  | ς = ς ↦ solve(σ; 𝒫; 𝒬)
  | ς = τ ↦ if ς ∉ fvς(τ) then let σ′ = [τ/ς] in solve(σ′·σ; •; σ′(𝒬, 𝒫)) else fail
  | τ = ς ↦ if ς ∉ fvς(τ) then let σ′ = [τ/ς] in solve(σ′·σ; •; σ′(𝒬, 𝒫)) else fail
  | Unit = Unit ↦ solve(σ; 𝒫; 𝒬)
  | (τ₁ → τ₂) = (τ₃ → τ₄) ↦ solve(σ; 𝒫; τ₁ = τ₃, τ₂ = τ₄, 𝒬)
  | (τ₁ ⟹ τ₂) = (τ₃ ⟹ τ₄) ↦ solve(σ; 𝒫; τ₁ = τ₃, τ₂ = τ₄, 𝒬)
  | otherwise ↦ fail
```

The first case applies when both skeletons are the same type variable ς. Then the equality trivially holds. Hence, we drop it and proceed with solving the remaining constraints. The next two cases apply when either τ_1 or τ_2 is a skeleton variable ς. If the occurs check fails, there is no finite solution and the algorithm signals failure. Otherwise, the constraint is solved by instantiating the ς. This additional substitution is accumulated and applied to all other constraints $\mathcal{P}, \mathcal{Q}$. Because the substitution might have modified some of the already processed constraints $\mathcal{P}$, we have to revisit them. Hence, they are all pushed back onto the queue, which is processed recursively.

The next three cases consider three different ways in which the two skeletons can have the same instantiated top-level structure. In those cases, the equality is decomposed into equalities on the subterms, which are pushed onto the queue and processed recursively.

The last catch-all case deals with all ways in which the two skeletons can be instantiated to different structures. Then there is no solution.

Skeleton Annotations. The next four cases consider a skeleton annotation $\alpha : \tau$ at the head of the queue and propagate the skeleton instantiation to the type variable. The first case, where the skeleton is a variable ς, has nothing to do, moves the annotation to

the processed constraints and proceeds with the remainder of the queue. In the other three cases, the skeleton is instantiated and the solver instantiates the type variable with the corresponding structure, introducing fresh variables for any subterms, where implicitly annotate every type variable with its skeleton: α^τ. The instantiating substitution is accumulated and applied to the remaining constraints, which are processed recursively.

$$
\begin{aligned}
&\mathtt{solve}(\sigma;\ \mathcal{P};\ \alpha:\tau, \mathcal{Q}) = \mathtt{match}\ \tau\ \mathtt{with} \\
&\quad |\ \varsigma \qquad \mapsto \mathtt{solve}(\sigma;\ \mathcal{P}, \alpha:\tau;\ \mathcal{Q}) \\
&\quad |\ \mathtt{Unit} \quad \mapsto \mathtt{let}\ \sigma' = [\mathtt{Unit}/\alpha]\ \mathtt{in}\ \mathtt{solve}(\sigma'\cdot\sigma;\ \bullet;\ \sigma'(\mathcal{Q},\mathcal{P})) \\
&\quad |\ \tau_1 \to \tau_2 \mapsto \mathtt{let}\ \sigma' = [(\alpha_1^{\tau_1} \to \alpha_2^{\tau_2}\ !\ \delta)/\alpha]\ \mathtt{in}\ \mathtt{solve}(\sigma'\cdot\sigma;\ \bullet;\ \alpha_1:\tau_1, \alpha_2:\tau_2, \sigma'(\mathcal{Q},\mathcal{P})) \\
&\quad |\ \tau_1 \Rightarrow \tau_2 \mapsto \mathtt{let}\ \sigma' = [(\alpha_1^{\tau_1}\ !\ \delta_1 \Rightarrow \alpha_2^{\tau_2}\ !\ \delta_2)/\alpha]\ \mathtt{in}\ \mathtt{solve}(\sigma'\cdot\sigma;\ \bullet;\ \alpha_1:\tau_1, \alpha_2:\tau_2, \sigma'(\mathcal{Q},\mathcal{P}))
\end{aligned}
$$

Value Type Subtyping. Next are the cases where a subtyping constraint between two value types $A_1 \leqslant A_2$—evidenced by coercion variable ω—is at the head of the queue. We consider six different situations.

$$
\begin{aligned}
&\mathtt{solve}(\sigma;\ \mathcal{P};\ \omega:A_1 \leqslant A_2, \mathcal{Q}) = \mathtt{match}\ A_1 \leqslant A_2\ \mathtt{with} \\
&\quad |\ A \leqslant A \mapsto \mathtt{let}\ T = elab_S(A)\ \mathtt{in}\ \mathtt{solve}([\langle T\rangle/\omega]\cdot\sigma;\ \mathcal{P};\ \mathcal{Q}) \\
&\quad |\ \alpha^{\tau_1} \leqslant A \mapsto \mathtt{let}\ \tau_2 = skeleton(A)\ \mathtt{in}\ \mathtt{solve}(\sigma;\ \mathcal{P}, \omega:\alpha^{\tau_1} \leqslant A;\ \tau_1 = \tau_2, \mathcal{Q}) \\
&\quad |\ A \leqslant \alpha^{\tau_1} \mapsto \mathtt{let}\ \tau_2 = skeleton(A)\ \mathtt{in}\ \mathtt{solve}(\sigma;\ \mathcal{P}, \omega:A \leqslant \alpha^{\tau_1};\ \tau_2 = \tau_1, \mathcal{Q}) \\
&\quad |\ (A_1 \to B_1\ !\ \Delta_1) \leqslant (A_2 \to B_2\ !\ \Delta_2) \mapsto \mathtt{let}\ \sigma' = [(\omega_1 \to \omega_2\ !\ \omega_3)/\omega]\ \mathtt{in} \\
&\qquad\quad \mathtt{solve}(\sigma'\cdot\sigma;\ \mathcal{P};\ \omega_1:A_2 \leqslant A_1, \omega_2:B_1 \leqslant B_2, \omega_3:\Delta_1 \leqslant \Delta_2, \mathcal{Q}) \\
&\quad |\ (A_1\ !\ \Delta_1 \Rightarrow A_2\ !\ \Delta_2) \leqslant (A_3\ !\ \Delta_3 \Rightarrow A_4\ !\ \Delta_4) \mapsto \mathtt{let}\ \sigma' = [(\omega_1\ !\ \omega_2 \Rightarrow \omega_3\ !\ \omega_4)/\omega]\ \mathtt{in} \\
&\qquad\quad \mathtt{solve}(\sigma'\cdot\sigma;\ \mathcal{P};\ \omega_1:A_3 \leqslant A_1, \omega_2:\Delta_3 \leqslant \Delta_1, \omega_3:A_2 \leqslant A_4, \omega_4:\Delta_2 \leqslant \Delta_4, \mathcal{Q}) \\
&\quad |\ \mathtt{otherwise} \mapsto \mathtt{fail}
\end{aligned}
$$

If the two types are equal, the subtyping holds trivially through reflexivity. The solver thus drops the constraint and instantiates ω with the reflexivity coercion $\langle T\rangle$. Note that each coercion variable only appears in one constraint. So we only accumulate the substitution and do not have to apply it to the other constraints. In the next two cases, one of the two types is a type variable α. Then we move the constraint to the processed set. We also add an equality constraint between the skeletons to the queue, thus enforcing the invariant that only types with the same skeleton are compared. Through the skeleton equality, the type structure (if any) from the type is also transferred to the type variable. The next two cases concern two types with the same top-level instantiation. In these cases, the solver decomposes the constraint into constraints on the corresponding subterms and appropriately relates the evidence of the old constraint to the new ones. The final case catches all situations where the two types are instantiated with a different structure and thus there is no solution.

Auxiliary function *skeleton*(*A*), defined in [Appendix C](), computes the skeleton of *A*.

Dirt Subtyping. The final six cases deal with subtyping constraints between dirts.

```
solve(σ; 𝒫; ω : Δ ⩽ Δ', 𝒬) = match Δ ⩽ Δ' with
   | 𝒪 ∪ δ ⩽ 𝒪' ∪ δ' ↦ if 𝒪 ≠ ∅ then let σ' = [((𝒪\𝒪') ∪ δ'')/δ', 𝒪 ∪ ω'/ω] in
                                solve(σ' · σ; •; (ω' : δ ≤ σ'(Δ')), σ'(𝒬, 𝒫))
                          else solve(σ; 𝒫, (ω : Δ ⩽ Δ'); 𝒬)
   | ∅ ⩽ Δ' ↦ solve([∅_{Δ'}/ω] · σ; 𝒫; 𝒬)
   | δ ⩽ ∅ ↦ let σ' = [∅/δ; ∅_∅/ω] in solve(σ' · σ; •; σ'(𝒬, 𝒫))
   | 𝒪 ∪ δ ⩽ 𝒪' ↦
   if 𝒪 ⊆ 𝒪' then let σ' = [𝒪 ∪ ω'/ω] in solve(σ' · σ; 𝒫, (ω' : δ ⩽ 𝒪')); 𝒬) else fail
   | 𝒪 ⩽ 𝒪' ↦ if 𝒪 ⊆ 𝒪' then let σ' = [𝒪 ∪ ∅_{𝒪'\𝒪}/ω] in solve(σ' · σ; 𝒫; 𝒬) else fail
   | 𝒪 ⩽ 𝒪' ∪ δ' ↦ let σ' = [(𝒪\𝒪') ∪ δ''/δ'; 𝒪' ∪ ∅_{(𝒪'\𝒪)∪δ''}/ω] in solve(σ' · σ; •; σ'(𝒬, 𝒫))
```

If the two dirts are of the general form $\mathcal{O} \cup \delta$ and $\mathcal{O}' \cup \delta'$, we distinguish two sub-cases. Firstly, if $\mathcal{O}$ is empty, there is nothing to be done and we move the constraint to the processed set. Secondly, if $\mathcal{O}$ is non-empty, we partially instantiate δ' with any of the operations that appear in $\mathcal{O}$ but not in $\mathcal{O}'$. We then drop $\mathcal{O}$ from the constraint, and, after substitution, proceed with processing all constraints. For instance, for $\{\mathtt{Op}_1\} \cup \delta \leqslant \{\mathtt{Op}_2\} \cup \delta'$, we instantiate δ' to $\{\mathtt{Op}_1\} \cup \delta''$—where δ'' is a fresh dirt variable—and proceed with the simplified constraint $\delta \leqslant \{\mathtt{Op}_1, \mathtt{Op}_2\} \cup \delta''$. Note that due to the set semantics of dirts, it is not valid to simplify the above constraint to $\delta \leqslant \{\mathtt{Op}_2\} \cup \delta''$. After all the substitution, $[\delta \mapsto \{\mathtt{Op}_1\}, \delta'' \mapsto \emptyset]$ solves the former and the original constraint, but not the latter.

The second case, $\emptyset \leqslant \Delta'$, always holds and is discharged by instantiating ω to $\emptyset_{\Delta'}$. The third case, $\delta \leqslant \emptyset$, has only one solution: $\delta \mapsto \emptyset$ with coercion $\emptyset_\emptyset$. The fourth case, $\mathcal{O} \cup \delta \leqslant \mathcal{O}'$, has as many solutions as there are subsets of $\mathcal{O}'$, provided that $\mathcal{O} \subseteq \mathcal{O}'$. We then simplify the constraint to $\delta \leqslant \mathcal{O}'$, which we move to the set of processed constraints. The fifth case, $\mathcal{O} \leqslant \mathcal{O}'$, holds iff $\mathcal{O} \subseteq \mathcal{O}'$. The last case, $\mathcal{O} \leqslant \mathcal{O}' \cup \delta'$, is like the first, but without a dirt variable in the left-hand side. We can satisfy it in a similar fashion, by partially instantiating δ' with $(\mathcal{O} \setminus \mathcal{O}') \cup \delta''$—where δ'' is a fresh dirt variable. Now the constraint is satisfied and can be discarded.

6 Erasure of effect information from ExEff

Our first backend for ExEff is SkelEff, which is essentially a copy of ExEff from which all effect information Δ, type information T and coercions γ have been erased. Instead, skeletons τ play the role of plain types. Thus, SkelEff is essentially System F extended with term-level (but not type-level) support for algebraic effects.

The main point of SkelEff is to show that we can erase the effects and subtyping from ExEff to obtain types that are compatible with a System F-like language. At the term level, SkelEff also resembles a subset of Multicore OCaml (Dolan *et al.*, 2015), which provides native support for algebraic effects and handlers but features no explicit polymorphism.

Terms

value $v ::= x \mid \texttt{unit} \mid h \mid \texttt{fun}\,(x:\tau) \mapsto c \mid \Lambda\varsigma.v \mid v\,\tau$

handler $h ::= \{\texttt{return}\,(x:\tau) \mapsto c_r, \texttt{Op}_1\,x\,k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\,x\,k \mapsto c_{\texttt{Op}_n}\}$

computation $c ::= v_1\,v_2 \mid \texttt{let}\,x = v\,\texttt{in}\,c \mid \texttt{return}\,v \mid \texttt{Op}\,v\,(y:\tau.c)$

$\mid\ \texttt{do}\,x \leftarrow c_1; c_2 \mid \texttt{handle}\,c\,\texttt{with}\,v$

Types

type $\tau ::= \varsigma \mid \tau_1 \to \tau_2 \mid \tau_1 \Rightarrow \tau_2 \mid \texttt{Unit} \mid \forall\varsigma.\tau$

Fig. 11. SKELEFF syntax.

6.1 The SKELEFF language

Figure 11 defines the syntax of SKELEFF. The type system and operational semantics of SKELEFF follow from those of EXEFF and can be found in Appendix D.

6.2 Erasure

Figure 12 defines erasure functions $\varepsilon_v^\sigma(v)$, $\varepsilon_c^\sigma(c)$, $\varepsilon_V^\sigma(T)$, $\varepsilon_C^\sigma(\underline{C})$ and $\varepsilon_E^\sigma(\Gamma)$ for values, computations, value types, computation types and type environments, respectively. All five functions take a substitution σ from the free type variables α to their skeleton τ as an additional parameter.

Thanks to the skeleton-based design of EXEFF, erasure is straightforward. All types are erased to their skeletons, dropping quantifiers for type variables and all occurrences of dirt sets. Moreover, coercions are dropped from values and computations. Finally, all binders and elimination forms for type variables, dirt set variables and coercions are dropped from values and type environments.

Example 6.1. Continuing the Example 5.2, a monomorphic function

$$\texttt{let}\,f\ :\ (\texttt{Unit} \to \texttt{Unit}\,!\,\emptyset) \to \texttt{Unit}\,!\,\emptyset$$
$$= \texttt{fun}\,(g:\texttt{Unit} \to \texttt{Unit}\,!\,\emptyset) \mapsto g\,\texttt{unit}$$
$$\texttt{in}\,\ldots$$

is erased to

$$\texttt{let}\,f\ :\ (\texttt{Unit} \to \texttt{Unit}) \to \texttt{Unit}$$
$$= \texttt{fun}\,(g:\texttt{Unit} \to \texttt{Unit}) \mapsto g\,\texttt{unit}$$
$$\texttt{in}\,\ldots$$

while its polymorphic variant

$$\texttt{let}\,f\ :\ \forall\varsigma.\forall\alpha:\varsigma.\forall\alpha':\varsigma.\forall\delta.\forall\delta'.(\alpha \leqslant \alpha') \Rightarrow (\delta \leqslant \delta') \Rightarrow (\texttt{Unit} \to \alpha\,!\,\delta) \to \alpha'\,!\,\delta'$$
$$= \Lambda\varsigma.\Lambda(\alpha:\varsigma).\Lambda(\alpha':\varsigma).\Lambda\delta.\Lambda\delta'.\Lambda(\omega:\alpha \leqslant \alpha').\Lambda(\omega':\delta \leqslant \delta').$$
$$\texttt{fun}\,(g:\texttt{Unit} \to \alpha\,!\,\delta) \mapsto ((g\,\texttt{unit}) \triangleright (\omega\,!\,\omega'))$$
$$\texttt{in}\,\ldots$$

is erased to

$$\texttt{let}\,f\ :\ \forall\varsigma.(\texttt{Unit} \to \varsigma) \to \varsigma$$
$$= \Lambda\varsigma.\texttt{fun}\,(g:\texttt{Unit} \to \varsigma) \mapsto g\,\texttt{unit}$$
$$\texttt{in}\,\ldots$$

$$\varepsilon_v^\sigma(x) = x \qquad\qquad \varepsilon_v^\sigma(\Lambda\delta.v) = \varepsilon_v^\sigma(v)$$
$$\varepsilon_v^\sigma(\texttt{unit}) = \texttt{unit} \qquad\qquad \varepsilon_v^\sigma(\Lambda(\omega:\pi).v) = \varepsilon_v^\sigma(v)$$
$$\varepsilon_v^\sigma(v \triangleright \gamma) = \varepsilon_v^\sigma(v) \qquad\qquad \varepsilon_v^\sigma(v\,\tau) = \varepsilon_v^\sigma(v)\,\tau$$
$$\varepsilon_v^\sigma(\texttt{fun}\,(x:T) \mapsto c) = \texttt{fun}\,(x:\varepsilon_V^\sigma(T)) \mapsto \varepsilon_c^\sigma(c) \qquad\qquad \varepsilon_v^\sigma(v\,T) = \varepsilon_v^\sigma(v)$$
$$\varepsilon_v^\sigma(\Lambda\varsigma.v) = \Lambda\varsigma.\varepsilon_v^\sigma(v) \qquad\qquad \varepsilon_v^\sigma(v\,\Delta) = \varepsilon_v^\sigma(v)$$
$$\varepsilon_v^\sigma(\Lambda(\alpha:\tau).v) = \varepsilon_v^{\sigma\cdot\{\alpha\mapsto\tau\}}(v) \qquad\qquad \varepsilon_v^\sigma(v\,\gamma) = \varepsilon_v^\sigma(v)$$

$$\varepsilon_v^\sigma(\{\texttt{return}\,(x:T) \mapsto c_r, [\texttt{Op}\,x\,k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}) =$$
$$\{\texttt{return}\,(x:\varepsilon_V^\sigma(T)) \mapsto \varepsilon_c^\sigma(c_r), [\texttt{Op}\,x\,k \mapsto \varepsilon_c^\sigma(c_{\texttt{Op}})]_{\texttt{Op}\in\mathcal{O}}\}$$

$$\varepsilon_c^\sigma(v_1\,v_2) = \varepsilon_v^\sigma(v_1)\,\varepsilon_v^\sigma(v_2)$$
$$\varepsilon_c^\sigma(\texttt{let}\,x = v\,\texttt{in}\,c) = \texttt{let}\,x = \varepsilon_v^\sigma(v)\,\texttt{in}\,\varepsilon_c^\sigma(c)$$
$$\varepsilon_c^\sigma(\texttt{return}\,v) = \texttt{return}\,(\varepsilon_v^\sigma(v))$$
$$\varepsilon_c^\sigma(\texttt{Op}\,v\,(y:T.c)) = \texttt{Op}\,(\varepsilon_v^\sigma(v))\,(y:\varepsilon_V^\sigma(T).\varepsilon_c^\sigma(c))$$
$$\varepsilon_c^\sigma(\texttt{do}\,x \leftarrow c_1; c_2) = \texttt{do}\,x \leftarrow \varepsilon_c^\sigma(c_1); \varepsilon_c^\sigma(c_2)$$
$$\varepsilon_c^\sigma(\texttt{handle}\,c\,\texttt{with}\,v) = \texttt{handle}\,\varepsilon_c^\sigma(c)\,\texttt{with}\,\varepsilon_v^\sigma(v)$$
$$\varepsilon_c^\sigma(c \triangleright \gamma) = \varepsilon_c^\sigma(c)$$

$$\varepsilon_V^\sigma(\alpha) = \sigma(\alpha) \qquad\qquad \varepsilon_C^\sigma(T\,!\,\Delta) = \varepsilon_V^\sigma(T)$$
$$\varepsilon_V^\sigma(T \to \underline{C}) = \varepsilon_V^\sigma(T) \to \varepsilon_C^\sigma(\underline{C})$$
$$\varepsilon_V^\sigma(\underline{C_1} \Rightarrow \underline{C_2}) = \varepsilon_C^\sigma(\underline{C_1}) \Rightarrow \varepsilon_C^\sigma(\underline{C_2}) \qquad\qquad \varepsilon_E^\sigma(\varepsilon) = \varepsilon$$
$$\varepsilon_V^\sigma(\texttt{Unit}) = \texttt{Unit} \qquad\qquad \varepsilon_E^\sigma(\Gamma,\varsigma) = \varepsilon_E^\sigma(\Gamma),\varsigma$$
$$\varepsilon_V^\sigma(\pi \Rightarrow T) = \varepsilon_V^\sigma(T) \qquad\qquad \varepsilon_E^\sigma(\Gamma,\alpha:\tau) = \varepsilon_E^{\sigma\cdot\{\alpha\mapsto\tau\}}(\Gamma)$$
$$\varepsilon_V^\sigma(\forall\varsigma.T) = \forall\varsigma.\varepsilon_V^\sigma(T) \qquad\qquad \varepsilon_E^\sigma(\Gamma,\delta) = \varepsilon_E^\sigma(\Gamma)$$
$$\varepsilon_V^\sigma(\forall(\alpha:\tau).T) = \varepsilon_V^{\sigma\cdot\{\alpha\mapsto\tau\}}(T) \qquad\qquad \varepsilon_E^\sigma(\Gamma,x:T) = \varepsilon_E^\sigma(\Gamma),x:\varepsilon_V^\sigma(T)$$
$$\varepsilon_V^\sigma(\forall\delta.T) = \varepsilon_V^\sigma(T) \qquad\qquad \varepsilon_E^\sigma(\Gamma,\omega:\pi) = \varepsilon_E^\sigma(\Gamma)$$

Fig. 12. Definition of type erasure.

Note that in addition to removing all effect annotations and coercions, the erasure removed type quantifiers and abstractions and replaced α and α' with their skeleton ς.

We proceed similarly in applications, where

$$f\,\texttt{Unit}\,\texttt{Unit}\,\texttt{Unit}\,\emptyset\,\emptyset\,\langle\texttt{Unit}\rangle\,\emptyset_\emptyset\,(\texttt{fun}\,(x:\texttt{Unit}) \mapsto \texttt{return}\,x)$$

is erased simply to

$$f\,\texttt{Unit}\,(\texttt{fun}\,(x:\texttt{Unit}) \mapsto \texttt{return}\,x)$$

where only the skeleton application remains. Similarly,

$$f\,\texttt{Unit}\,\texttt{Unit}\,\texttt{Unit}\,\{\texttt{Tick}\}\,\{\texttt{Tick},\texttt{Tock}\}\,\langle\texttt{Unit}\rangle\,(\{\texttt{Tick}\}\cup\emptyset_{\{\texttt{Tock}\}})$$
$$(\texttt{fun}\,x:\texttt{Unit} \mapsto \texttt{Tick}\,x\,(y:\texttt{Unit}.((\texttt{return}\,y) \triangleright \langle\texttt{Unit}\rangle\,!\,\emptyset_{\{\texttt{Tick}\}})))$$

is erased to

$$f\,\texttt{Unit}\,(\texttt{fun}\,x:\texttt{Unit} \mapsto \texttt{Tick}\,x\,(y:\texttt{Unit}.\texttt{return}\,y))$$

showing that a polymorphic function is applied in exactly the same way to a pure or an impure function.

The expected theorems hold. Firstly, types are preserved by erasure, where typing for SKELEFF values and computations takes the obvious forms $\Gamma \vdash_{ev} v : \tau$ and $\Gamma \vdash_{ec} c : \tau$.

Theorem 6.2 (Type Preservation). *If* $\Gamma \vdash_{v} v : T$, *then* $\varepsilon_E^{\emptyset}(\Gamma) \vdash_{ev} \varepsilon_v^{\Gamma}(v) : \varepsilon_V^{\Gamma}(T)$. *If* $\Gamma \vdash_{c} c : \underline{C}$ *then* $\varepsilon_E^{\emptyset}(\Gamma) \vdash_{ec} \varepsilon_c^{\Gamma}(c) : \varepsilon_C^{\Gamma}(\underline{C})$.

Here we abuse of notation and use Γ as a substitution from type variables to skeletons used by the erasure functions.

Finally, we have that erasure preserves the operational semantics.

Theorem 6.3 (Semantic Preservation). *If* $v \rightsquigarrow_{v} v'$, *then* $\varepsilon_v^{\sigma}(v) \equiv_v^{\rightsquigarrow} \varepsilon_v^{\sigma}(v')$. *If* $c \rightsquigarrow_{c} c'$, *then* $\varepsilon_c^{\sigma}(c) \equiv_c^{\rightsquigarrow} \varepsilon_c^{\sigma}(c')$.

In both cases, $\equiv^{\rightsquigarrow}$ denotes the congruence closure of the step relation in SKELEFF, defined in [Appendix D](). The choice of substitution σ does not matter as types do not affect the behaviour. Note that because coercions are dropped during erasure, this means that also in EXEFF they do not have an essential runtime impact.

Corollary 6.4 (Coercion Irrelevance). *If* $v \rightsquigarrow_{v}^{*} v_1$ *and* $v \triangleright \gamma \rightsquigarrow_{v}^{*} v_2$, *then* $\varepsilon_v^{\sigma}(v_1) \equiv_v^{\rightsquigarrow} \varepsilon_v^{\sigma}(v_2)$. *If* $c \rightsquigarrow_{c}^{*} c_1$ *and* $c \triangleright \gamma \rightsquigarrow_{c}^{*} c_2$, *then* $\varepsilon_v^{\sigma}(c_1) \equiv_c^{\rightsquigarrow} \varepsilon_v^{\sigma}(c_2)$.

Discussion. The reason we need to use the symmetric congruence closure of the step relation in our preservation theorem is that the original EXEFF program and the resulting SKELEFF program do not necessarily operate in lockstep. Indeed, the erasure of casts with coercions, of type and coercion binders and of their applications means that the erased program does not have to step through their reductions. On the other hand, the erasure of type and coercion binders may expose applications of skeleton binders that the SKELEFF program has to reduce whereas the original EXEFF program does not.

For example, take the EXEFF term

$$c_1 = (\lambda(x : \forall \delta.\text{Unit}).\text{return}\,(\lambda(y : \text{Unit}).\text{return}\,(x\,\emptyset)))\,(\Lambda\delta.(\Lambda\varsigma.\text{unit})\,\text{Unit})$$

which β-reduces to

$$c_2 = \text{return}\,(\lambda(y : \text{Unit}).\text{return}\,((\Lambda\delta.(\Lambda\varsigma.\text{unit})\,\text{Unit})\,\emptyset))$$

When we erase c_1, we get

$$\varepsilon_c^{\sigma}(c_1) = (\lambda(x : \text{Unit}).\text{return}\,(\lambda(y : \text{Unit}).\text{return}\,x))\,((\Lambda\varsigma.\text{unit})\,\text{Unit})$$

The erasure of the $\Lambda\delta$ binder exposes a new redex that has precedence. Hence, $\varepsilon_c^{\sigma}(c_1)$ steps to

$$(\lambda(x : \text{Unit}).\text{return}\,(\lambda(y : \text{Unit}).\text{return}\,x))\,\text{unit}$$

which steps to the irreducible computation

$$\texttt{return}\,(\lambda(y:\texttt{Unit}).\texttt{return}\,\texttt{unit})$$

In contrast, c_2 erases to a different irreducible computation

$$\varepsilon_{\mathsf{c}}^{\sigma}(c_2) = \texttt{return}\,(\lambda(y:\texttt{Unit}).\texttt{return}\,((\Lambda\varsigma.\texttt{unit})\,\texttt{Unit}))$$

These two irreducible computations can be made equal by reducing under the $\lambda(y:\texttt{Unit})$ binder in $\varepsilon_{\mathsf{c}}^{\sigma}(c_2)$. The congruence closure of the step relation allows this reduction under binders. Morever, the closure is symmetric because an ExEff step may defer or block a SkelEff step that is exposed by the erasure.

Typically, when type information is erased from call-by-value languages, type binders are erased by replacing them with other (dummy) binders. For instance, the expected definition of erasure would be

$$\varepsilon_{\mathsf{v}}^{\sigma}(\Lambda(\alpha:\tau).v) = \lambda(x:\texttt{Unit}).\varepsilon_{\mathsf{v}}^{\sigma}(v)$$

This replacement is motivated by a desire to preserve the behaviour of the typed terms. By dropping binders, values might be turned into computations that trigger their side-effects immediately, rather than at the later point where the original binder was eliminated. However, there is no call for this circumspect approach in our setting, as our grammatical partition of terms in values (without side effects) and computations (with side effects) guarantees that this problem cannot happen when we erase values to values and computations to computations. Nevertheless, when adding recursion to the language, care is needed to preserve the termination behaviour of values under erasure, though we believe this is not a problem as appropriate recursive constructs are invoked only at the computation level.

7 Elaboration to a language without effects

This section considers an alternative backend for ExEff, called NoEff. In contrast to SkelEff, NoEff's types are explicit about whether or not effects can be used, but implicit about which effects in particular are used.

Given that NoEff's types track whether effects are used or not, its name may seem contradictory. Yet, the calculus is intended to model a purely functional approach to implementing handlers, for example, in the pure fragment of OCaml or in Haskell, where there is no native support for algebraic effects (thus the name NoEff). In such pure languages, algebraic effects are modelled by means of a user-defined encoding (Kammar *et al.*, 2013; Kiselyov & Sivaramakrishnan, 2016; Pretnar *et al.*, 2017; Leijen, 2017) and the type constructors used by these encodings reveal whether effectful or pure computations are encoded. Here, to keep NoEff small, we encapsulate the particular encoding details—which could be implemented in a library—and present the effect functionality as opaque primitives in NoEff.

7.1 Syntax of NoEff

Figure 13 presents the syntax of NoEff. Notably, NoEff replaces ExEff's two syntactic sorts of values and computations by a single syntactic sort of terms that combines their syntactic forms. The four absent forms are dirt and skeleton abstraction and application,
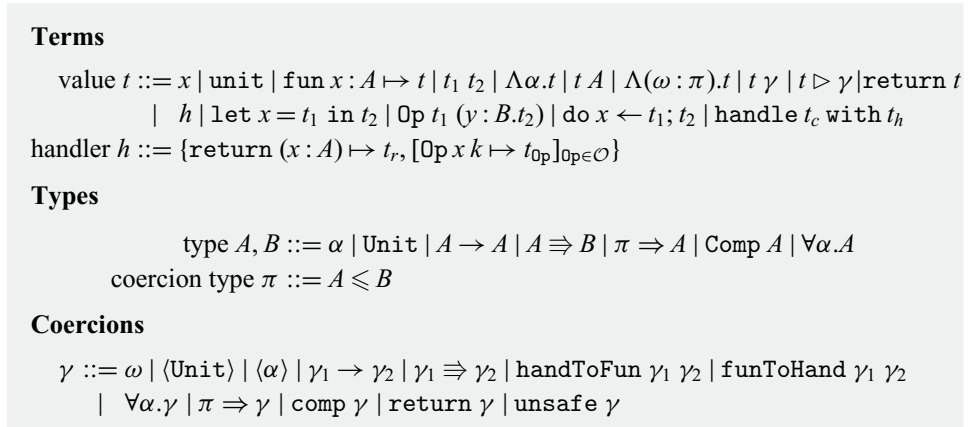
Terms

value $t ::= x \mid \text{unit} \mid \text{fun } x : A \mapsto t \mid t_1\, t_2 \mid \Lambda\alpha.t \mid t\, A \mid \Lambda(\omega : \pi).t \mid t\, \gamma \mid t \rhd \gamma \mid \text{return } t$
$\qquad \mid h \mid \text{let } x = t_1 \text{ in } t_2 \mid \text{Op } t_1\ (y : B.t_2) \mid \text{do } x \leftarrow t_1; t_2 \mid \text{handle } t_c \text{ with } t_h$

handler $h ::= \{\text{return } (x : A) \mapsto t_r, [\text{Op } x\, k \mapsto t_{\text{Op}}]_{\text{Op} \in \mathcal{O}}\}$

Types

$\qquad\qquad$ type $A, B ::= \alpha \mid \text{Unit} \mid A \rightarrow A \mid A \Rightarrow B \mid \pi \Rightarrow A \mid \text{Comp } A \mid \forall\alpha.A$
$\qquad$ coercion type $\pi ::= A \leqslant B$

Coercions

$\gamma ::= \omega \mid \langle\text{Unit}\rangle \mid \langle\alpha\rangle \mid \gamma_1 \rightarrow \gamma_2 \mid \gamma_1 \Rightarrow \gamma_2 \mid \text{handToFun } \gamma_1\, \gamma_2 \mid \text{funToHand } \gamma_1\, \gamma_2$
$\qquad \mid \forall\alpha.\gamma \mid \pi \Rightarrow \gamma \mid \text{comp } \gamma \mid \text{return } \gamma \mid \text{unsafe } \gamma$

Fig. 13. NOEFF syntax.

as NOEFF does not feature either dirt or skeletons. Similarly, EXEFF's syntactic sorts for value types T and computation types $\underline{C}$ are merged into a single sort of types A. Here EXEFF's computation types of the form $T\,!\,\Delta$ are replaced by NOEFF's computation types Comp A without dirt. The absence of dirt can also be seen in NOEFF's coercion types π, which do not feature a form for dirt subtyping.

Finally, NOEFF features adapted versions of EXEFF's type coercions. Absent are those related to dirt and skeletons, and the computation type coercion is abstracted to the form (comp γ) which does not feature a dirt coercion. There are also four new coercion forms (handToFun $\gamma_1\, \gamma_2$, funToHand $\gamma_1\, \gamma_2$, return γ and unsafe γ) which enable the elaboration from EXEFF into NOEFF; we explain their semantics when we discuss typing and their purpose when explaining the elaboration.

7.2 Typing of NOEFF

We now turn to typing of NOEFF. First, we introduce NOEFF typing environments; they are identical to those for EXEFF, modulo dirt and skeleton information:

$$\Gamma ::= \varepsilon \mid \Gamma, \alpha \mid \Gamma, x : A \mid \Gamma, \omega : \pi$$

The remainder of this section gives the typing judgements for terms (Section 7.2.1) and coercions (Section 7.2.2); uninteresting judgements like well-formedness of types ($\Gamma \vdash_A A$) and well-formedness of constraints ($\Gamma \vdash_\pi \pi$) are included in Appendix E.

7.2.1 Term typing

Typing for NOEFF terms is given by judgement $\Gamma \vdash_t t : A$, which is presented in Figure 14. The rules are similar to those of IMPEFF and EXEFF, with the exception of dirt and skeleton features, which are absent in NOEFF.

There is one subtle point: by design, type $A \Rightarrow B$ classifies handlers that handle terms of type Comp A and produce results of type Comp B. This way, we enforce that handlers always take computations to computations. If the input is not a computation, we can use a regular function instead of a handler. So this restriction matters little.

$\boxed{\Gamma \vdash_{\mathsf{t}} t : A}$ **Term Typing**

$$\dfrac{(x:A) \in \Gamma}{\Gamma \vdash_{\mathsf{t}} x : A} \qquad \dfrac{}{\Gamma \vdash_{\mathsf{t}} \mathtt{unit} : \mathtt{Unit}} \qquad \dfrac{\Gamma \vdash_{\mathbb{A}} A \qquad \Gamma, x:A \vdash_{\mathsf{t}} t : B}{\Gamma \vdash_{\mathsf{t}} (\mathtt{fun}\ x:A \mapsto t) : A \to B} \qquad \dfrac{\Gamma, \alpha \vdash_{\mathsf{t}} t : A}{\Gamma \vdash_{\mathsf{t}} \Lambda \alpha. t : \forall \alpha. A}$$

$$\dfrac{\Gamma \vdash_{\mathbb{A}} A \qquad \Gamma \vdash_{\mathsf{t}} t : \forall \alpha. B}{\Gamma \vdash_{\mathsf{t}} t\ A : B[A/\alpha]} \qquad \dfrac{\Gamma \vdash_{\mathsf{t}} t : A \qquad \Gamma \vdash_{\mathsf{co}} \gamma : A \leqslant B}{\Gamma \vdash_{\mathsf{t}} t \triangleright \gamma : B}$$

$$\dfrac{\Gamma, x:A \vdash_{\mathsf{t}} t_r : \mathtt{Comp}\ B \quad \left[(\mathtt{Op} : A_1 \to A_2) \in \Sigma \quad \Gamma, x:A_1, k:A_2 \to \mathtt{Comp}\ B \vdash_{\mathsf{t}} t_{\mathtt{Op}} : \mathtt{Comp}\ B \right]_{\mathtt{Op} \in \mathcal{O}}}{\Gamma \vdash_{\mathsf{t}} \{\mathtt{return}\ (x:A) \mapsto t_r, [\mathtt{Op}\ x\ k \mapsto t_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : A \Rightarrow B}$$

$$\dfrac{\Gamma \vdash_{\overline{\pi}} \pi \qquad \Gamma, \omega : \pi \vdash_{\mathsf{t}} t : A}{\Gamma \vdash_{\mathsf{t}} \Lambda(\omega : \pi). t : \pi \Rightarrow A} \qquad \dfrac{\Gamma \vdash_{\mathsf{t}} t : \pi \Rightarrow A \qquad \Gamma \vdash_{\mathsf{co}} \gamma : \pi}{\Gamma \vdash_{\mathsf{t}} t\ \gamma : A} \qquad \dfrac{\Gamma \vdash_{\mathsf{t}} t_1 : A \to B \qquad \Gamma \vdash_{\mathsf{t}} t_2 : A}{\Gamma \vdash_{\mathsf{t}} t_1\ t_2 : B}$$

$$\dfrac{\Gamma \vdash_{\mathsf{t}} t_1 : A \qquad \Gamma, x:A \vdash_{\mathsf{t}} t_2 : B}{\Gamma \vdash_{\mathsf{t}} \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : B} \qquad \dfrac{\Gamma \vdash_{\mathsf{t}} t : A}{\Gamma \vdash_{\mathsf{t}} \mathtt{return}\ t : \mathtt{Comp}\ A}$$

$$\dfrac{(\mathtt{Op} : A_1 \to A_2) \in \Sigma \qquad \Gamma \vdash_{\mathsf{t}} t_1 : A_1 \qquad \Gamma, y:A_2 \vdash_{\mathsf{t}} t_2 : \mathtt{Comp}\ B}{\Gamma \vdash_{\mathsf{t}} \mathtt{Op}\ t_1\ (y:A_2. t_2) : \mathtt{Comp}\ B}$$

$$\dfrac{\Gamma \vdash_{\mathsf{t}} t_1 : \mathtt{Comp}\ A \qquad \Gamma, x:A \vdash_{\mathsf{t}} t_2 : \mathtt{Comp}\ B}{\Gamma \vdash_{\mathsf{t}} \mathtt{do}\ x \leftarrow t_1; t_2 : \mathtt{Comp}\ B} \qquad \dfrac{\Gamma \vdash_{\mathsf{t}} t_h : A \Rightarrow B \qquad \Gamma \vdash_{\mathsf{t}} t_c : \mathtt{Comp}\ A}{\Gamma \vdash_{\mathsf{t}} \mathtt{handle}\ t_c\ \mathtt{with}\ t_h : \mathtt{Comp}\ B}$$

Fig. 14. NOEFF term typing.

More importantly, by forcing the output to be a computation, we avoid a potential source of unsoundness in NOEFF. Indeed, because the type system does not track which operations are performed in the input computation, we cannot tell whether or not they will all be handled. Of course, we do want any operation that is not handled to be forwarded to the output, just like in EXEFF. Hence, because we cannot statically tell in NOEFF whether any operations will be forwarded, to remain on the safe side, we have to assume that there may be some. Thus, with forwarded operations, the output must be a computation. We will see that this causes additional difficulties in the elaboration from EXEFF to NOEFF.

7.2.2 Coercion typing

Coercion typing is given by judgement $\Gamma \vdash_{\mathsf{co}} \gamma : \pi$, presented in Figure 15. Most of the rules are straightforward so we only focus on the four new coercion forms.

The first new coercion form ($\mathtt{handToFun}\ \gamma_1\ \gamma_2$) concerns the issue of handler typing above. It converts a handler, which expects a computation as input, into a function, which can be applied to a non-computation. The next coercion form ($\mathtt{funToHand}\ \gamma_1\ \gamma_2$) is its dual; it turns a function into a handler that only specifies how to handle the $\mathtt{return}$ case and forwards all operations.

The third new coercion form ($\mathtt{return}\ \gamma$) promotes a value t of any type A to a computation $\mathtt{return}\ t$ of type $\mathtt{Comp}\ A$. The last new coercion form ($\mathtt{unsafe}\ \gamma$) is the dual of the previous form. It forces a value of computation type $\mathtt{Comp}\ A$ to a value of type A. This only

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \pi}$ **Coercion Typing**

$$\frac{(\omega : \pi) \in \Gamma}{\Gamma \vdash_{\mathsf{co}} \omega : \pi} \qquad \frac{}{\Gamma \vdash_{\mathsf{co}} \langle \mathtt{Unit} \rangle : \mathtt{Unit} \leqslant \mathtt{Unit}} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash_{\mathsf{co}} \langle \alpha \rangle : \alpha \leqslant \alpha}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_2 \leqslant A_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : B_1 \leqslant B_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \to \gamma_2 : (A_1 \to B_1) \leqslant (A_2 \to B_2)}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \mathtt{Comp}\ A_2 \leqslant \mathtt{Comp}\ A_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \mathtt{Comp}\ B_1 \leqslant \mathtt{Comp}\ B_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \Rightarrow \gamma_2 : (A_1 \Rightarrow B_1) \leqslant (A_2 \Rightarrow B_2)}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_2 \leqslant A_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \mathtt{Comp}\ B_1 \leqslant B_2}{\Gamma \vdash_{\mathsf{co}} \mathtt{handToFun}\ \gamma_1\ \gamma_2 : (A_1 \Rightarrow B_1) \leqslant (A_2 \to B_2)}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_2 \leqslant A_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : B_1 \leqslant \mathtt{Comp}\ B_2}{\Gamma \vdash_{\mathsf{co}} \mathtt{funToHand}\ \gamma_1\ \gamma_2 : \leqslant (A_1 \to B_1) \leqslant (A_2 \Rightarrow B_2)} \qquad \frac{\Gamma, \alpha \vdash_{\mathsf{co}} \gamma : A \leqslant B}{\Gamma \vdash_{\mathsf{co}} \forall \alpha . \gamma : \forall \alpha . A \leqslant \forall \alpha . B}$$

$$\frac{\Gamma \vdash_{\overline{\pi}} \pi \qquad \Gamma \vdash_{\mathsf{co}} \gamma : A \leqslant B}{\Gamma \vdash_{\mathsf{co}} \pi \Rightarrow \gamma : \pi \Rightarrow A \leqslant \pi \Rightarrow B} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : A_1 \leqslant A_2}{\Gamma \vdash_{\mathsf{co}} \mathtt{comp}\ \gamma : \mathtt{Comp}\ A_1 \leqslant \mathtt{Comp}\ A_2}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : A_1 \leqslant A_2}{\Gamma \vdash_{\mathsf{co}} \mathtt{return}\ \gamma : A_1 \leqslant \mathtt{Comp}\ A_2} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : A_1 \leqslant A_2}{\Gamma \vdash_{\mathsf{co}} \mathtt{unsafe}\ \gamma : \mathtt{Comp}\ A_1 \leqslant A_2}$$

Fig. 15. NOEFF coercion typing.

works when the value is of the form `return` t and in that case yields t. If the computation is of the form $\mathtt{Op}\ t_1\ (y : B.t_2)$, the cast gets stuck; hence its name. We will see that this is the single source of type unsafety in NOEFF, though we claim that programs elaborated from EXEFF into NOEFF only use this coercion in a safe way and never get stuck.

7.3 Operational semantics of NOEFF

Figure 16 presents selected rules of NOEFF's small-step, call-by-value operational semantics. We omit other rules as they closely follow the rules for EXEFF, except being adjusted for the amalgamation of values and computations. The complete operational semantics can be found in Appendix E.

The first rule pushes the cast onto the returned value; in contrast to EXEFF, there is no effect information to lose, making this reduction type-preserving. This allows the second and third rule which are simplified variants of the ones for NOEFF: because all the coercions can be pushed into t^R, there is no need to extract their pure parts before substituting t^R for a variable. The remaining five rules capture the semantics of the newly introduced coercion forms, exactly as described in Section 7.2.2.

$$\text{value } t^R ::= \text{unit} \mid h \mid \text{fun } x : A \mapsto t \mid \Lambda\alpha.t \mid \Lambda(\omega : \pi).t \mid t^R \rhd (\gamma_1 \to \gamma_2) \mid t^R \rhd (\gamma_1 \Rightarrow \gamma_2)$$
$$\mid \ t^R \rhd (\text{handToFun } \gamma_1 \ \gamma_2) \mid t^R \rhd (\text{funToHand } \gamma_1 \ \gamma_2) \mid t^R \rhd \forall\alpha.\gamma \mid t^R \rhd (\pi \Rightarrow \gamma)$$
$$\mid \ \text{return } t^R \mid \text{Op } t^R \ (y : A.t)$$

$\boxed{t \rightsquigarrow t'}$ **Operational Semantics**

$$(\text{return } t^R) \rhd (\text{comp } \gamma) \rightsquigarrow \text{return } (t^R \rhd \gamma)$$

$$\text{do } x \leftarrow \text{return } t^R; t \rightsquigarrow t[t^R/x] \qquad\qquad \text{handle } (\text{return } t^R) \text{ with } h \rightsquigarrow t_r[t^R/x]$$

$$(t_1^R \rhd (\text{handToFun } \gamma_1 \ \gamma_2)) \ t_2^R \rightsquigarrow (\text{handle } (\text{return } (t_2^R \rhd \gamma_1)) \text{ with } t_1^R) \rhd \gamma_2$$

$$\text{handle } (\text{return } t_1^R) \text{ with } (t_2^R \rhd (\text{funToHand } \gamma_1 \ \gamma_2)) \rightsquigarrow (t_2^R \ (t_1^R \rhd \gamma_1)) \rhd \gamma_2$$

$$\text{handle } (\text{Op } t_1^R \ (y : B.t)) \text{ with } (t_2^R \rhd (\text{funToHand } \gamma_1 \ \gamma_2))$$
$$\rightsquigarrow \text{Op } t_1^R \ (y : B.\text{handle } t \text{ with } (t_2^R \rhd (\text{funToHand } \gamma_1 \ \gamma_2)))$$

$$t^R \rhd \text{return } \gamma \rightsquigarrow \text{return } (t^R \rhd \gamma) \qquad (\text{return } t^R) \rhd (\text{unsafe } \gamma) \rightsquigarrow t^R \rhd \gamma$$

Fig. 16. NOEFF operational semantics (selected rules).

The NOEFF Metatheory. We have proven a weak form of type safety for NOEFF in terms of type preservation and (partial) progress theorems. The latter characterises the way in which well-typed terms can get stuck.

Theorem 7.1 (Preservation). *If $\Gamma \vdash_{\text{t}} t : A$ and $t \rightsquigarrow t'$, then $\Gamma \vdash_{\text{t}} t' : A$.*

Theorem 7.2 (Partial Progress). *If $\Gamma \vdash_{\text{t}} t : A$, then either (a) t is a value, (b) $t \rightsquigarrow t'$, or (c) t is "stuck".*

Stuck terms are defined as follows:

$$t^S ::= \text{Op } t^R \ (y : A.t) \rhd \text{unsafe } \gamma \mid t^S \ A \mid t^S \rhd \gamma \mid t^S \ \gamma \mid t^S \ t \mid t^R \ t^S \mid \text{let } x = t^S \text{ in } t \mid \text{return } t^S$$
$$\mid \ \text{Op } t^S \ (y : A.t) \mid \text{do } x \leftarrow t^S; t \mid \text{handle } t_c \text{ with } t^S \mid \text{handle } t^S \text{ with } t^R$$

The first case is the essential one, while the remaining ones just provide an evaluation context around it. Hence, terms only get stuck when an unsafe coercion is applied to an operation. As we have already indicated, we claim that elaborated NOEFF programs never end up in this situation.

7.4 Elaboration of EXEFF *to* NOEFF

7.4.1 Type elaboration

Figure 17 presents the elaboration of value types ($\Gamma \vdash_T T : \tau \rightsquigarrow A$) and computation types ($\Gamma \vdash_C \underline{C} : \underline{\tau} \rightsquigarrow \underline{A}$). The latter captures the main idea of the whole elaboration: when the dirt

$\boxed{nonEmpty(\Delta)}$ **Conservative Non-Empty Dirt**

$$\frac{}{nonEmpty(\delta)} \qquad\qquad \frac{}{nonEmpty(\{\mathtt{Op}\} \cup \Delta)}$$

$\boxed{\Gamma \vdash_T T : \tau \rightsquigarrow A}$ **Value Type Elaboration**

$$\frac{(\alpha : \tau) \in \Gamma}{\Gamma \vdash_T \alpha : \tau \rightsquigarrow \alpha} \qquad \frac{}{\Gamma \vdash_T \mathtt{Unit} : \mathtt{Unit} \rightsquigarrow \mathtt{Unit}} \qquad \frac{\Gamma \vdash_T T : \tau_1 \rightsquigarrow A \qquad \Gamma \vdash_{\underline{C}} \underline{C} : \tau_2 \rightsquigarrow B}{\Gamma \vdash_T T \to \underline{C} : \tau_1 \to \tau_2 \rightsquigarrow A \to B}$$

$$\frac{\Gamma \vdash_T T : \tau_1 \rightsquigarrow A \qquad \Gamma \vdash_{\underline{C}} \underline{C} : \tau_2 \rightsquigarrow B}{\Gamma \vdash_T T \,!\, \emptyset \Rightarrow \underline{C} : \tau_1 \Rightarrow \tau_2 \rightsquigarrow A \to B}$$

$$\frac{\Gamma \vdash_T T_1 : \tau_1 \rightsquigarrow A \qquad \Gamma \vdash_T T_2 : \tau_2 \rightsquigarrow B \qquad nonEmpty(\Delta_1)}{\Gamma \vdash_T (T_1 \,!\, \Delta_1 \Rightarrow T_2 \,!\, \Delta_2) : \tau_1 \Rightarrow \tau_2 \rightsquigarrow A \Rightarrow B} \qquad \frac{\Gamma, \varsigma \vdash_T T : \tau \rightsquigarrow A}{\Gamma \vdash_T \forall \varsigma.T : \forall \varsigma.\tau \rightsquigarrow A}$$

$$\frac{\Gamma, \alpha : \tau_1 \vdash_T T : \tau_2 \rightsquigarrow A}{\Gamma \vdash_T \forall (\alpha : \tau_1).T : \tau_2 \rightsquigarrow \forall \alpha.A} \qquad \frac{\Gamma, \delta \vdash_T T : \tau \rightsquigarrow A}{\Gamma \vdash_T \forall \delta.T : \tau \rightsquigarrow A} \qquad \frac{\Gamma \vdash_T T : \tau \rightsquigarrow A}{\Gamma \vdash_T (\Delta_1 \leqslant \Delta_2) \Rightarrow T : \tau \rightsquigarrow A}$$

$$\frac{\Gamma \vdash_T T_1 : \tau_1 \rightsquigarrow B_1 \qquad \Gamma \vdash_T T_2 : \tau_1 \rightsquigarrow B_2 \qquad \Gamma \vdash_T T : \tau \rightsquigarrow A}{\Gamma \vdash_T (T_1 \leqslant T_2) \Rightarrow T : \tau \rightsquigarrow (B_1 \leqslant B_2) \Rightarrow A}$$

$$\frac{\Gamma \vdash_{\underline{C}} \underline{C}_1 : \tau_1 \rightsquigarrow B_1 \qquad \Gamma \vdash_{\underline{C}} \underline{C}_2 : \tau_1 \rightsquigarrow B_2 \qquad \Gamma \vdash_T T : \tau \rightsquigarrow A}{\Gamma \vdash_T (\underline{C}_1 \leqslant \underline{C}_2) \Rightarrow T : \tau \rightsquigarrow (B_1 \leqslant B_2) \Rightarrow A}$$

$\boxed{\Gamma \vdash_{\underline{C}} \underline{C} : \tau \rightsquigarrow A}$ **Computation Type Elaboration**

$$\frac{\Gamma \vdash_T T : \tau \rightsquigarrow A}{\Gamma \vdash_{\underline{C}} T \,!\, \emptyset : \tau \rightsquigarrow A} \qquad \frac{nonEmpty(\Delta) \qquad \Gamma \vdash_T T : \tau \rightsquigarrow A}{\Gamma \vdash_{\underline{C}} T \,!\, \Delta : \tau \rightsquigarrow \mathtt{Comp}\ A}$$

Fig. 17. Elaboration of EXEFF types to NOEFF types.

Δ of a computation type is empty, the elaboration of the computation type $T \,!\, \Delta$ is just the elaboration of the value type T. If it is non-empty, the elaborated value type A is wrapped in a computation type, $\mathtt{Comp}\ A$. We cannot always tell whether Δ is empty or not, namely in case it is a dirt variable δ. Our conservative solution is to assume that dirt variables are also non-empty. This works because we can always represent a term t of type A in terms of a trivial computation $\mathtt{return}\ t$ of type $\mathtt{Comp}\ A$.

Most cases for value types are straightforward, but a few are worth mentioning. Firstly, to respect the particularities of NOEFF handler types explained in Section 7.2.1, we distinguish two different cases for elaborating EXEFF. Recall that if a computation type has an empty dirt, it is elaborated to some pure type A, not a computation type $\mathtt{Comp}\ A$ that handlers expect. Correspondingly, handler types with empty input dirts are elaborated into NOEFF function types. If the dirt is non-empty, we unavoidably elaborate to a NOEFF handler type. Note that in the latter case, we ignore whether or not the output computation type has an empty dirt; the NOEFF handler type always implicitly assumes an output computation type.

Secondly, since dirts and skeletons are absent from NoEff, the elaboration drops universal quantification over skeletons and dirts, as well as dirt subtyping qualifiers.

Coercion Elaboration. We now turn to the elaboration of ExEff coercions to NoEff coercions. Most cases are straightforward and either copy a ExEff coercion to its NoEff counterpart, or drop a dirt- or skeleton-related ExEff construct that is not present in NoEff. Hence, we only discuss the interesting cases here; the complete definition can be found in Appendix F.

Two groups of rules do deserve additional explanation. The first group concerns the elaboration of handler coercions. If we compare the input dirts of the source and target handler types of the coercion, there are three different cases: either both are empty, both are non-empty, or the source input dirt is non-empty and the target input dirt is empty. The fourth combination is not possible due to the monotonicity of subtyping and the contravariance in the input argument.

In the first case, both the source and the target ExEff type elaborate to NoEff function types, and thus the coercion is elaborated to a function coercion:

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : T_2 \,!\, \emptyset \leqslant T_1 \,!\, \emptyset \rightsquigarrow \gamma_1' \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C}_1 \leqslant \underline{C}_2 \rightsquigarrow \gamma_2'}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \Rightarrow \gamma_2 : (T_1 \,!\, \emptyset \Rightarrow \underline{C}_1) \leqslant (T_2 \,!\, \emptyset \Rightarrow \underline{C}_2) \rightsquigarrow \gamma_1' \rightarrow \gamma_2'}$$

In the second case, both types elaborate to NoEff handler types, and thus the whole coercion is elaborated to a NoEff handler coercion:

$$\frac{nonEmpty(\Delta_1) \qquad nonEmpty(\Delta_2)}{\Gamma \vdash_{\mathsf{co}} \gamma_1 : (T_2 \,!\, \Delta_2 \leqslant T_1 \,!\, \Delta_1) \rightsquigarrow \gamma_1' \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : T_1' \leqslant T_2' \rightsquigarrow \gamma_2' \qquad \Gamma \vdash_{\mathsf{co}} \gamma_3 : \Delta_1' \leqslant \Delta_2'}{\Gamma \vdash_{\mathsf{co}} (\gamma_1 \Rightarrow (\gamma_2 \,!\, \gamma_3)) : ((T_1 \,!\, \Delta_1) \Rightarrow (T_1' \,!\, \Delta_1')) \leqslant ((T_2 \,!\, \Delta_2) \Rightarrow (T_2' \,!\, \Delta_2')) \rightsquigarrow \gamma_1' \Rightarrow \mathtt{comp}\ \gamma_2'}$$

In the third case, the elaborated source type is a handler type and the target type is a function type. Here, we use the `handToFun` coercion to bridge between the two. There are two subcases to consider though, depending on whether the source output dirt is empty or not:

$$\frac{nonEmpty(\Delta_1)}{\Gamma \vdash_{\mathsf{co}} \gamma_1 : T_2 \leqslant T_1 \rightsquigarrow \gamma_1' \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : (T_1' \leqslant T_2') \rightsquigarrow \gamma_2' \quad \Gamma \vdash_{\mathsf{co}} \gamma_3 : \emptyset \leqslant \Delta_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma_4 : \emptyset \leqslant \Delta_2'}{\Gamma \vdash_{\mathsf{co}} (\gamma_1 \,!\, \gamma_3 \Rightarrow \gamma_2 \,!\, \gamma_4) : ((T_1 \,!\, \Delta_1 \Rightarrow T_1' \,!\, \emptyset) \leqslant (T_2 \,!\, \emptyset \Rightarrow T_2' \,!\, \Delta_2')) \rightsquigarrow \mathtt{handToFun}\ \gamma_1'\ (\mathtt{unsafe}\ \gamma_2')}$$

$$\frac{nonEmpty(\Delta_1) \qquad nonEmpty(\Delta_1')}{\Gamma \vdash_{\mathsf{co}} \gamma_1 : T_2 \leqslant T_1 \rightsquigarrow \gamma_1' \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : (T_1' \leqslant T_2') \rightsquigarrow \gamma_2' \quad \Gamma \vdash_{\mathsf{co}} \gamma_3 : \emptyset \leqslant \Delta_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma_4 : \Delta_1' \leqslant \Delta_2'}{\Gamma \vdash_{\mathsf{co}} (\gamma_1 \,!\, \gamma_3 \Rightarrow \gamma_2 \,!\, \gamma_4) : ((T_1 \,!\, \Delta_1 \Rightarrow T_1' \,!\, \Delta_1') \leqslant (T_2 \,!\, \emptyset \Rightarrow T_2' \,!\, \Delta_2')) \rightsquigarrow \mathtt{handToFun}\ \gamma_1'\ \gamma_2'}$$

In the former case, NoEff does not respect the emptiness in the elaborated handler type, but does respect it in the elaboration of γ_2. To bridge the discrepancy that arises here, we insert an `unsafe` coercion. In the latter case, no discrepancy arises, and no `unsafe` coercion is needed.

The second group of interest concerns the elaboration of computation type coercions. Again we distinguish three different cases based on the source and target dirt. If both are empty, the computation type coercion is elaborated like the underlying value type coercion γ_1:

$$\frac{\Gamma \vdash_{\texttt{co}} \gamma_1 : T_1 \leqslant T_2 \leadsto \gamma_1' \qquad \Gamma \vdash_{\texttt{co}} \gamma_2 : \emptyset \leqslant \emptyset}{\Gamma \vdash_{\texttt{co}} (\gamma_1 \mathbin{!} \gamma_2) : (T_1 \mathbin{!} \emptyset \leqslant T_2 \mathbin{!} \emptyset) \leadsto \gamma_1'}$$

If both are non-empty, we elaborate to a NoEff computation type coercion $\texttt{comp}\ \gamma_1'$:

$$\frac{\Gamma \vdash_{\texttt{co}} \gamma_1 : T_1 \leqslant T_2 \leadsto \gamma_1' \qquad \Gamma \vdash_{\texttt{co}} \gamma_2 : \emptyset \leqslant \Delta_2 \qquad nonEmpty(\Delta_2)}{\Gamma \vdash_{\texttt{co}} (\gamma_1 \mathbin{!} \gamma_2) : (T_1 \mathbin{!} \emptyset \leqslant T_2 \mathbin{!} \Delta_2) \leadsto \texttt{return}\ \gamma_1'}$$

In the third case, there is a mismatch because the source is pure and the target is impure; we bridge this with a $\texttt{return}$ coercion:

$$\frac{\Gamma \vdash_{\texttt{co}} \gamma_1 : T_1 \leqslant T_2 \leadsto \gamma_1' \qquad \Gamma \vdash_{\texttt{co}} \gamma_2 : \Delta_1 \leqslant \Delta_2 \qquad nonEmpty(\Delta_1) \qquad nonEmpty(\Delta_2)}{\Gamma \vdash_{\texttt{co}} (\gamma_1 \mathbin{!} \gamma_2) : (T_1 \mathbin{!} \Delta_1 \leqslant T_2 \mathbin{!} \Delta_2) \leadsto \texttt{comp}\ \gamma_1'}$$

7.4.2 Value elaboration

Again, the elaboration of ExEff values into NoEff terms is mostly straightforward, so we only discuss the interesting cases here; the complete definition can be found in Appendix F. There are two cases of interest: handlers and dirt applications.

Handlers. We have three rules describing different cases of elaborating handlers of type $T_x \mathbin{!} \mathcal{O} \Rightarrow T \mathbin{!} \Delta$. Recall from Section 7.4.1 that if $\mathcal{O} = \emptyset$, handlers need to be elaborated into functions, which is described by the first of these three rules:

$$\frac{\Gamma \vdash_T T : \tau \leadsto A \qquad \Gamma, x : T \vdash_{\texttt{c}} c_r : C \leadsto t}{\Gamma \vdash_{\texttt{v}} \{\texttt{return}\ (x : T) \mapsto c_r\} : T \mathbin{!} \emptyset \Rightarrow C \leadsto \texttt{fun}\ (x : A) \mapsto t}$$

The second rule describes the case where $\mathcal{O}$ is non-empty, but Δ is empty:

$$\frac{\begin{array}{c} nonEmpty(\mathcal{O}) \qquad \Gamma \vdash_T T_x : \tau \leadsto A \qquad \Gamma, x : T_x \vdash_{\texttt{c}} c_r : T \mathbin{!} \emptyset \leadsto t_r \\ \left[(\texttt{Op} : T_1^{\texttt{Op}} \to T_2^{\texttt{Op}}) \in \Sigma \quad \vdash_T T_i^{\texttt{Op}} : \tau_i^{\texttt{Op}} \leadsto A_i^{\texttt{Op}} \quad \Gamma, x : T_1^{\texttt{Op}}, k : T_2^{\texttt{Op}} \to T \mathbin{!} \emptyset \vdash_{\texttt{c}} c_{\texttt{Op}} : T \mathbin{!} \emptyset \leadsto t_{\texttt{Op}} \right]_{\texttt{Op} \in \mathcal{O}} \end{array}}{\begin{array}{c} \Gamma \vdash_{\texttt{v}} \{\texttt{return}\ (x : T_x) \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} : T_x \mathbin{!} \mathcal{O} \Rightarrow T \mathbin{!} \emptyset \\ \leadsto \{\texttt{return}\ (x : A) \mapsto \texttt{return}\ t_r, [\texttt{Op}\ x\ k \mapsto \texttt{return}\ t_{\texttt{Op}}[k \triangleright \langle A_1^{\texttt{Op}} \rangle \to \texttt{unsafe}\ \langle A_2^{\texttt{Op}} \rangle / k]]_{\texttt{Op} \in \mathcal{O}}\} \end{array}}$$

Since $\mathcal{O}$ is non-empty, we do elaborate a handler into a handler, but there is an important caveat. Recall from Section 7.2.1 that to ensure safe forwarding of unhandled operations, handlers take computations to computations. But as Δ is empty, handler clauses of type $T \mathbin{!} \emptyset$ are elaborated to terms of type A (the elaboration of T), not $\texttt{Comp}\ A$ as expected. We amend this by wrapping them with a $\texttt{return}$. However, the handled continuations now include an extraneous $\texttt{return}$, which we remove with an $\texttt{unsafe}$ coercion before plugging them into the operation clause that expects k to result in A, not $\texttt{Comp}\ A$.

In the third rule, both $\mathcal{O}$ and Δ are non-empty, and the elaboration is structural:

$$\frac{\begin{array}{c} nonEmpty(\mathcal{O}) \qquad nonEmpty(\Delta) \qquad \Gamma \vdash_T T_x : \tau \leadsto A \qquad \Gamma, x : T_x \vdash_{\texttt{c}} c_r : T \mathbin{!} \Delta \leadsto t_r \\ \left[(\texttt{Op} : T_1^{\texttt{Op}} \to T_2^{\texttt{Op}}) \in \Sigma \qquad \Gamma, x : T_1^{\texttt{Op}}, k : T_2^{\texttt{Op}} \to T \mathbin{!} \Delta \vdash_{\texttt{c}} c_{\texttt{Op}} : T \mathbin{!} \Delta \leadsto t_{\texttt{Op}} \right]_{\texttt{Op} \in \mathcal{O}} \end{array}}{\begin{array}{c} \Gamma \vdash_{\texttt{v}} \{\texttt{return}\ (x : T_x) \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} : T_x \mathbin{!} \mathcal{O} \Rightarrow T \mathbin{!} \Delta \\ \leadsto \{\texttt{return}\ (x : A) \mapsto t_r, [\texttt{Op}\ x\ k \mapsto t_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} \end{array}}$$

Dirt Applications. The elaboration of dirt applications possibly needs to bridge between an impure and a pure type. Consider for instance a EXEFF value v of type $\forall\delta.\text{Unit} \to \text{Unit}\,!\,\delta$ which is applied to the empty dirt; thus the type of the dirt application is $\text{Unit} \to \text{Unit}\,!\,\emptyset$. The elaboration of the former type is $\text{Unit} \to \text{Comp Unit}$, while the latter is $\text{Unit} \to \text{Unit}$.

Such elaborations are handled by the following rule:

$$\frac{\Gamma \vdash_{\bar{v}} v : \forall\delta.T \rightsquigarrow t \qquad \delta \mapsto \Delta; \Gamma \vdash_{\bar{v}} T \rightsquigarrow \gamma}{\Gamma \vdash_{\bar{v}} v\,\Delta : T[\Delta/\delta] \rightsquigarrow t \triangleright \gamma}$$

where for a given v of type $\forall\delta.T$, we need a coercion γ from the elaboration of T (recall this is done under the assumption $nonEmpty(\delta)$) to the elaboration of $T[\Delta/\delta]$. Such coercion is produced by a judgement $\delta \mapsto \Delta; \Gamma \vdash_{\bar{v}} T \rightsquigarrow \gamma$, driven by the structure of T. This judgement is defined in Figure 18 alongside with the judgement $\delta \mapsto \Delta; \Gamma \vdash_{\bar{c}} C \rightsquigarrow \gamma$ for computation types. In addition, there are two dual judgements $\Delta \mapsto \delta; \Gamma \vdash_{\bar{v}} T \rightsquigarrow \gamma$ and $\Delta \mapsto \delta; \Gamma \vdash_{\bar{c}} C \rightsquigarrow \gamma$ for the opposite coercions, which are used on types in contravariant positions. We have omitted their definitions because they are obtained by flipping the sides of all $\mapsto$ arrows, and replacing unsafe with return and handToFun with funToHand. Most rules of these judgements are straightforward congruences.

The main rule of interest is the one that produces an unsafe coercion where the dirt variable δ in a computation type $T\,!\,\delta$ is instantiated to the empty dirt $\emptyset$ (Rule FICMP2). In that case, the elaboration of the polymorphic abstraction conservatively assumes the computation is impure, while the elaboration of its instantiation accurately knows it is pure.

A further case that deserves attention is that of the handler type, where four different rules (Rules FIHAND1, FIHAND2, FIHAND3 and FIHAND4) cover the possible scenarios related to elaboration into handler and function types.

Note that in Rule FICOABSTY, we have restricted the case of $T_1 \leqslant T_2 \Rightarrow T$ to situations where T_1 and T_2 are both types variables. This is not a severe restriction because subtyping constraints can be simplified to this form; this simplification is precisely what our type inference algorithm does. Moreover, there is a good reason to impose the syntactic restriction. Consider the trivial reflexive subtyping constraint $(\text{Unit} \to \text{Unit}\,!\,\delta) \leqslant (\text{Unit} \to \text{Unit}\,!\,\delta)$. If we conservatively assume that δ is non-empty, the constraint is elaborated to $(\text{Unit} \to \text{Comp Unit}) \leqslant (\text{Unit} \to \text{Comp Unit})$, whereas, if δ is instantiated to $\emptyset$, the constraint is elaborated to $(\text{Unit} \to \text{Unit}) \leqslant (\text{Unit} \to \text{Unit})$. Hence, we would need to be able to coerce a coercion for the former constraint to a coercion for the latter, and vice versa. This would require a complication of the NOEFF language with additional coercion forms to accomplish this coercion of coercions, which, happily, the above syntactic restriction allows us to avoid.

7.4.3 Computation elaboration

Finally, Figure 19 defines how EXEFF computations are elaborated into NOEFF terms. There are a number of interesting cases.

Firstly, because (return v) has an empty dirt, its elaborated form drops the return (Rule CRET). Secondly, do- computations are translated to either let- or do- terms, depending on whether the dirt is empty or not (Rules CDO1 and CDO2, respectively).

$$\boxed{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma} \;\; \textbf{Value Type Coercion from Impure Dirt Instantiation}$$

$$\frac{}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} \text{Unit} \rightsquigarrow \langle \text{Unit} \rangle} \;\text{FiUnit} \qquad \frac{\Delta \mapsto \delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma_1 \qquad \delta \mapsto \Delta; \Gamma \vdash_{\overline{c}} \underline{C} \rightsquigarrow \gamma_2}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightarrow \underline{C} \rightsquigarrow \gamma_1 \rightarrow \gamma_2} \;\text{FiArr}$$

$$\frac{\Delta \mapsto \delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma_1 \qquad \delta \mapsto \Delta; \Gamma \vdash_{\overline{c}} \underline{C} \rightsquigarrow \gamma_2}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T\,!\,\emptyset \Rightarrow \underline{C} \rightsquigarrow \gamma_1 \rightarrow \gamma_2} \;\text{FiHand1}$$

$$\frac{\Delta_2[\emptyset/\delta] = \emptyset \qquad \emptyset \mapsto \delta; \Gamma \vdash_{\overline{v}} T_1 \rightsquigarrow \gamma_1 \qquad \delta \mapsto \emptyset; \Gamma \vdash_{\overline{v}} T_2 \rightsquigarrow \gamma_2}{\delta \mapsto \emptyset; \Gamma \vdash_{\overline{v}} T_1\,!\,\delta \Rightarrow T_2\,!\,\Delta_2 \rightsquigarrow \text{handToFun } \gamma_1 \,(\text{unsafe } \gamma_2)} \;\text{FiHand2}$$

$$\frac{nonEmpty(\Delta_2[\emptyset/\delta]) \qquad \emptyset \mapsto \delta; \Gamma \vdash_{\overline{v}} T_1 \rightsquigarrow \gamma_1 \qquad \delta \mapsto \emptyset; \Gamma \vdash_{\overline{v}} T_2 \rightsquigarrow \gamma_2}{\delta \mapsto \emptyset; \Gamma \vdash_{\overline{v}} T_1\,!\,\delta \Rightarrow T_2\,!\,\Delta_2 \rightsquigarrow \text{handToFun } \gamma_1 \,(\text{comp } \gamma_2)} \;\text{FiHand3}$$

$$\frac{\begin{array}{c} nonEmpty(\Delta_1[\Delta/\delta]) \\ \Delta \mapsto \delta; \Gamma \vdash_{\overline{c}} T_1\,!\,\Delta_1 \rightsquigarrow \gamma_1 \qquad \delta \mapsto \Delta; \Gamma, \delta' \vdash_{\overline{c}} T_2\,!\,\delta' \rightsquigarrow \gamma_2 \qquad \text{fresh } \delta' \end{array}}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T_1\,!\,\Delta_1 \Rightarrow T_2\,!\,\Delta_2 \rightsquigarrow \gamma_1 \Rightarrow \gamma_2} \;\text{FiHand4}$$

$$\frac{\delta \mapsto \Delta; \Gamma, \varsigma \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} \forall \varsigma.T \rightsquigarrow \gamma} \;\text{FiSkelAbs} \qquad \frac{\delta \mapsto \Delta; \Gamma, \alpha:\tau \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} \forall \alpha:\tau.T \rightsquigarrow \forall \alpha.\gamma} \;\text{FiTyAbs}$$

$$\frac{\delta \mapsto \Delta; \Gamma, \delta' \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} \forall \delta'.T \rightsquigarrow \gamma} \;\text{FiDirtAbs} \qquad \frac{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} \alpha_1 \leqslant \alpha_2 \Rightarrow T \rightsquigarrow \alpha_1 \leqslant \alpha_2 \Rightarrow \gamma} \;\text{FiCoAbsTy}$$

$$\frac{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} \Delta_1 \leqslant \Delta_2 \Rightarrow T \rightsquigarrow \gamma} \;\text{FiCoAbsDirt}$$

$$\boxed{\delta \mapsto \Delta; \Gamma \vdash_{\overline{c}} \underline{C} \rightsquigarrow \gamma} \;\; \textbf{Computation Type Coercion from Impure Dirt Instantiation}$$

$$\frac{\delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{c}} T\,!\,\emptyset \rightsquigarrow \gamma} \;\text{FiCmp1} \qquad \frac{\delta \mapsto \emptyset; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \emptyset; \Gamma \vdash_{\overline{c}} T\,!\,\delta \rightsquigarrow \text{unsafe } \gamma} \;\text{FiCmp2}$$

$$\frac{nonEmpty(\Delta'[\Delta/\delta]) \qquad \delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma}{\delta \mapsto \Delta; \Gamma \vdash_{\overline{c}} T\,!\,\Delta' \rightsquigarrow \text{comp } \gamma} \;\text{FiCmp3}$$

$$\boxed{\Delta \mapsto \delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma} \;\; \textbf{Value Type Coercion to Impure Dirt Instantiation}$$

$$\text{defined dually to } \delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma$$

$$\boxed{\Delta \mapsto \delta; \Gamma \vdash_{\overline{c}} \underline{C} \rightsquigarrow \gamma} \;\; \textbf{Computation Type Coercion to Impure Dirt Instantiation}$$

$$\text{defined dually to } \delta \mapsto \Delta; \Gamma \vdash_{\overline{c}} T \rightsquigarrow \gamma$$

Fig. 18. Type coercions from and to an impure dirt instantiation.

Thirdly, handler applications are elaborated in three possible ways. If the input dirt of the handler is empty, it is elaborated as a function and thus the handler application too should be elaborated as function application (Rule CHANDLE1). Otherwise, a handler application is indeed elaborated as a handler application. If the output dirt is empty, the translation is

$\boxed{\Gamma \vdash_{\text{c}} c : \underline{C} \rightsquigarrow t}$ **Computation Elaboration**

$$\frac{\Gamma \vdash_{\text{v}} v_1 : T \rightarrow \underline{C} \rightsquigarrow t_1 \qquad \Gamma \vdash_{\text{v}} v_2 : T \rightsquigarrow t_2}{\Gamma \vdash_{\text{c}} v_1 \ v_2 : \underline{C} \rightsquigarrow t_1 \ t_2} \ \text{CAPP}$$

$$\frac{\Gamma \vdash_{\text{v}} v : T \rightsquigarrow t_1 \qquad \Gamma, x : T \vdash_{\text{c}} c : \underline{C} \rightsquigarrow t_2}{\Gamma \vdash_{\text{c}} \text{let } x = v \text{ in } c : \underline{C} \rightsquigarrow \text{let } x = t_1 \text{ in } t_2} \ \text{CLET} \qquad \frac{\Gamma \vdash_{\text{v}} v : T \rightsquigarrow t}{\Gamma \vdash_{\text{c}} \text{return } v : T \, ! \, \emptyset \rightsquigarrow t} \ \text{CRET}$$

$$\frac{\begin{array}{c} (\text{Op} : T_1 \rightarrow T_2) \in \Sigma \qquad \Gamma \vdash_{\text{T}} T_1 : \tau_1 \rightsquigarrow A \\ \Gamma \vdash_{\text{T}} T_2 : \tau_2 \rightsquigarrow B \qquad \Gamma \vdash_{\text{v}} v : T_1 \rightsquigarrow t_v \qquad \Gamma, x : T_2 \vdash_{\text{c}} c : T \, ! \, \Delta \rightsquigarrow t_c \qquad \text{Op} \in \Delta \end{array}}{\Gamma \vdash_{\text{c}} \text{Op } v \ (y : T_2.c) : T \, ! \, \Delta \rightsquigarrow \text{Op } t_v \ (y : B.t_c)} \ \text{COP}$$

$$\frac{\Gamma \vdash_{\text{c}} c_1 : T_1 \, ! \, \emptyset \rightsquigarrow t_1 \qquad \Gamma, x : T_1 \vdash_{\text{c}} c_2 : T_2 \, ! \, \emptyset \rightsquigarrow t_2}{\Gamma \vdash_{\text{c}} (\text{do } x \leftarrow c_1; c_2) : T_2 \, ! \, \emptyset \rightsquigarrow \text{let } x = t_1 \text{ in } t_2} \ \text{CDo1}$$

$$\frac{nonEmpty(\Delta) \qquad \Gamma \vdash_{\text{c}} c_1 : (T_1 \, ! \, \Delta) \rightsquigarrow t_1 \qquad \Gamma, x : T_1 \vdash_{\text{c}} c_2 : (T_2 \, ! \, \Delta) \rightsquigarrow t_2}{\Gamma \vdash_{\text{c}} \text{do } x \leftarrow c_1; c_2 : (T_2 \, ! \, \Delta) \rightsquigarrow \text{do } x \leftarrow t_1; t_2} \ \text{CDo2}$$

$$\frac{\Gamma \vdash_{\text{c}} c : T \, ! \, \emptyset \rightsquigarrow t_2 \qquad \Gamma \vdash_{\text{v}} v : (T \, ! \, \emptyset \Rightarrow \underline{C}) \rightsquigarrow t_1}{\Gamma \vdash_{\text{c}} (\text{handle } c \text{ with } v) : \underline{C} \rightsquigarrow t_1 \ t_2} \ \text{CHANDLE1}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\text{c}} c : T_1 \, ! \, \Delta_1 \rightsquigarrow t_c \\ nonEmpty(\Delta_1) \qquad \Gamma \vdash_{\text{v}} v : (T_1 \, ! \, \Delta_1 \Rightarrow T_2 \, ! \, \emptyset) \rightsquigarrow t_v \qquad \Gamma \vdash_{\text{T}} T_2 : \tau \rightsquigarrow A \end{array}}{\Gamma \vdash_{\text{c}} (\text{handle } c \text{ with } v) : T_2 \, ! \, \emptyset \rightsquigarrow (\text{handle } t_c \text{ with } t_v) \triangleright \text{unsafe } \langle A \rangle} \ \text{CHANDLE2}$$

$$\frac{\begin{array}{c} nonEmpty(\Delta_2) \\ \Gamma \vdash_{\text{c}} c : T_1 \, ! \, \Delta_1 \rightsquigarrow t_c \qquad \Gamma \vdash_{\text{v}} v : (T_1 \, ! \, \Delta_1 \Rightarrow T_2 \, ! \, \Delta_2) \rightsquigarrow t_v \qquad nonEmpty(\Delta_1) \end{array}}{\Gamma \vdash_{\text{c}} (\text{handle } c \text{ with } v) : T_2 \, ! \, \Delta_2 \rightsquigarrow \text{handle } t_c \text{ with } t_v} \ \text{CHANDLE3}$$

$$\frac{\Gamma \vdash_{\text{c}} c : \underline{C}_1 \rightsquigarrow t \qquad \Gamma \vdash_{\text{co}} \gamma : \underline{C}_1 \leqslant \underline{C}_2 \rightsquigarrow \gamma'}{\Gamma \vdash_{\text{c}} c \triangleright \gamma : \underline{C}_2 \rightsquigarrow t \triangleright \gamma'} \ \text{CCAST}$$

Fig. 19. Elaboration of EXEFF computations to NOEFF terms.

straightforward (Rule CHANDLE3). However, if the output dirt is empty, then the elaboration of the handler still produces a computation where none is expected. Hence, we insert an $\texttt{unsafe}$ coercion to bridge the gap (Rule CHANDLE2).

Example 7.3. Elaboration of terms to NOEFF again depends on the type of a EXEFF term. A monomorphic function

$$\begin{array}{l} \texttt{let } f \ : \ (\texttt{Unit} \rightarrow \texttt{Unit} \, ! \, \emptyset) \rightarrow \texttt{Unit} \, ! \, \emptyset \\ \qquad = \texttt{fun } (g : \texttt{Unit} \rightarrow \texttt{Unit} \, ! \, \emptyset) \mapsto g \ \texttt{unit} \\ \texttt{in } \ldots \end{array}$$

is erased to

$$\begin{aligned}
\texttt{let } f \;:\; & (\texttt{Unit} \to \texttt{Unit}) \to \texttt{Unit} \\
& = \texttt{fun}\,(g : \texttt{Unit} \to \texttt{Unit}) \mapsto g\,\texttt{unit} \\
\texttt{in}\,\ldots
\end{aligned}$$

as before, while its polymorphic variant

$$\begin{aligned}
\texttt{let } f \;:\; & \forall\varsigma.\forall\alpha : \varsigma.\forall\alpha' : \varsigma.\forall\delta.\forall\delta'.(\alpha \leqslant \alpha') \Rightarrow (\delta \leqslant \delta') \Rightarrow (\texttt{Unit} \to \alpha\,!\,\delta) \to \alpha'\,!\,\delta' \\
& = \Lambda\varsigma.\Lambda(\alpha : \varsigma).\Lambda(\alpha' : \varsigma).\Lambda\delta.\Lambda\delta'.\Lambda(\omega : \alpha \leqslant \alpha').\Lambda(\omega' : \delta \leqslant \delta'). \\
& \qquad \texttt{fun}\,(g : \texttt{Unit} \to \alpha\,!\,\delta) \mapsto ((g\,\texttt{unit}) \rhd (\omega\,!\,\omega')) \\
\texttt{in}\,\ldots
\end{aligned}$$

is conservatively elaborated to an impure

$$\begin{aligned}
\texttt{let } f \;:\; & \forall\alpha.\forall\alpha'.(\alpha \leqslant \alpha') \Rightarrow (\texttt{Unit} \to \texttt{Comp}\,\alpha) \to \texttt{Comp}\,\alpha' \\
& = \Lambda\alpha.\Lambda\alpha'.\Lambda(\omega : \alpha \leqslant \alpha'). \\
& \qquad \texttt{fun}\,(g : \texttt{Unit} \to \texttt{Comp}\,\alpha) \mapsto ((g\,\texttt{unit}) \rhd \omega) \\
\texttt{in}\,\ldots
\end{aligned}$$

Note that in contrast to the erasure to SKELEFF, we keep type variables α and α', while removing any mention of their skeleton ς. As before, we remove any effect annotations, conservatively assuming that dirt variables are impure, but keep an explicit coercion ω between types.

Recall that in EXEFF, the application $f\ id$ was pure, and so must be its elaboration. However, since f itself was conservatively assumed to be impure, the application must be suitably coerced. In particular, the elaboration of

$$f\ \texttt{Unit Unit Unit}\ \emptyset\ \emptyset\ \langle\texttt{Unit}\rangle\ \emptyset_\emptyset\ (\texttt{fun}\,(x : \texttt{Unit}) \mapsto \texttt{return}\,x)$$

is

$$(((f\ \texttt{Unit Unit}) \rhd \gamma_1) \rhd \gamma_2)\ \langle\texttt{Unit}\rangle\ (\texttt{fun}\,(x : \texttt{Unit}) \mapsto x)$$

where for $\pi = \texttt{Unit} \leqslant \texttt{Unit}$, the coercion γ_1, which lifts a pure function into one that returns a computation, is given by

$$\pi \Rightarrow (\langle\texttt{Unit}\rangle \to \texttt{return}\,\langle\texttt{Unit}\rangle) \to \texttt{comp}\,\langle\texttt{Unit}\rangle$$
$$: (\pi \Rightarrow (\texttt{Unit} \to \texttt{Comp Unit}) \to \texttt{Comp Unit}) \leqslant (\pi \Rightarrow (\texttt{Unit} \to \texttt{Unit}) \to \texttt{Comp Unit})$$

while γ_2, which extracts back the value from a pure computation is

$$\pi \Rightarrow (\langle\texttt{Unit}\rangle \to \langle\texttt{Unit}\rangle) \to \texttt{unsafe}\,\langle\texttt{Unit}\rangle$$
$$: (\pi \Rightarrow (\texttt{Unit} \to \texttt{Unit}) \to \texttt{Comp Unit}) \leqslant (\pi \Rightarrow (\texttt{Unit} \to \texttt{Unit}) \to \texttt{Unit})$$

On a side note, observe the removal of `return` in the identity as its elaboration is a pure function. For an impure function

$$f\ \texttt{Unit Unit Unit}\ \{\texttt{Tick}\}\ \{\texttt{Tick}, \texttt{Tock}\}\ \langle\texttt{Unit}\rangle\ (\{\texttt{Tick}\} \cup \emptyset_{\{\texttt{Tock}\}})$$
$$(\texttt{fun}\,x : \texttt{Unit} \mapsto \texttt{Tick}\,x\,(y : \texttt{Unit}.((\texttt{return}\,y) \rhd \langle\texttt{Unit}\rangle\,!\,\emptyset_{\{\texttt{Tick}\}})))$$

the elaboration

$$(((f\ \texttt{Unit Unit}) \rhd \gamma_1') \rhd \gamma_2')\ \langle\texttt{Unit}\rangle$$
$$(\texttt{fun}\,x : \texttt{Unit} \mapsto \texttt{Tick}\,x\,(y : \texttt{Unit}.(y \rhd \texttt{return}\,\langle\texttt{Unit}\rangle)))$$

is similar, except that the coercions $\gamma_1' = \gamma_2'$ are both trivial:

$\pi \Rightarrow (\langle \text{Unit} \rangle \rightarrow \text{comp}\,\langle \text{Unit} \rangle) \rightarrow \text{comp}\,\langle \text{Unit} \rangle$

$: (\pi \Rightarrow (\text{Unit} \rightarrow \text{Comp Unit}) \rightarrow \text{Comp Unit}) \leqslant (\pi \Rightarrow (\text{Unit} \rightarrow \text{Comp Unit}) \rightarrow \text{Comp Unit})$

and may be removed by an optimiser. Also note that just as in *id*, the `return` vanishes in the elaboration, though in this case it is reintroduced by the elaboration of the coercion $\emptyset_{\{\text{Tick}\}}$.

7.4.4 Metatheory of elaboration

We have proven in Abella that the elaboration of EXEFF values and computations into NOEFF terms preserves typing.

Theorem 7.4 (Type Preservation)**.**

- *If $\Gamma \vdash_{\bar{v}} v : T \rightsquigarrow t$ and $\vdash_{\bar{\Gamma}} \Gamma \rightsquigarrow \Gamma'$ then $\Gamma \vdash_{\bar{T}} T : \tau \rightsquigarrow A$ and $\Gamma' \vdash_{\bar{t}} t : A$.*
- *If $\Gamma \vdash_{\bar{c}} c : \underline{C} \rightsquigarrow t$ and $\vdash_{\bar{\Gamma}} \Gamma \rightsquigarrow \Gamma'$ then $\Gamma \vdash_{\underline{c}} \underline{C} : \tau \rightsquigarrow B$ and $\Gamma' \vdash_{\bar{t}} t : B$.*

A key lemma in the theorem's proof establishes the appropriate typing of the coercion produced by the $\delta \mapsto \Delta; \Gamma \vdash_{\bar{v}} T \rightsquigarrow \gamma$ judgement.

Lemma 7.5 (From Impure Coercion Typing)**.** *If $\delta \mapsto \Delta; \Gamma \vdash_{\bar{v}} T \rightsquigarrow \gamma$ and $\Gamma, \delta \vdash_T T : \tau \rightsquigarrow A$ then there exists a B such that $\Gamma \vdash_T T[\Delta / \delta] : \tau \rightsquigarrow B$ and $\Gamma \vdash_{\text{co}} \gamma : A \leqslant B$.*

Semantic preservation for the elaboration from EXEFF to NOEFF turns out to be much more complicated than for the elaboration to SKELEFF. Indeed, the congruence closure of the step relation is not sufficient in the case of NOEFF.

For instance, consider the following EXEFF evaluation step:

$$(\Lambda \delta.\text{fun}\, x : \text{Unit} \mapsto v)\,\emptyset \rightsquigarrow_{\text{v}} \text{fun}\, x : \text{Unit} \mapsto v[\emptyset / \delta]$$

where the dirt abstraction $(\Lambda \delta.\text{fun}\, x : \text{Unit} \mapsto v)$ has type $\forall \delta.\text{Unit} \rightarrow \text{Unit}\,!\,\{\text{Op}\} \cup \delta$ and its application to $\emptyset$ has type $\text{Unit} \rightarrow \text{Unit}\,!\,\{\text{Op}\} \cup \emptyset$. Suppose that the right-hand-side elaborates to the NOEFF term $\text{fun}\, x : \text{Unit} \mapsto v'$. Then the left-hand side elaborates to $\text{fun}\,(x : \text{Unit} \mapsto v') \triangleright (\langle \text{Unit} \rangle \rightarrow \text{comp}\,\langle \text{Unit} \rangle)$; observe that the function coercion is nothing more than a reflexivity coercion. Neither of these two elaborated NOEFF terms is reducible. In particular, we cannot eliminate the reflexivity coercion by reduction and thus the two terms are not related by a congruence closure of the step relation.

Instead, we believe that a semantic notion of equivalence is needed: contextual equivalence (Morris, 1969). Informally, this notion expresses that two terms are equivalent iff, when placed in any "appropriate" program context, the resulting programs reduce to normal forms that are equivalent under some other, simpler notion of equivalence such as syntactic equality.

The precise formal definition depends on the particular setting it is used in. In our setting, there are a number of complicating factors that need to be taken into account.

- Firstly, we are dealing with two mutually recursive syntactic sorts for terms, values and computations. This calls for four different mutually recursive sorts of program contexts: ones that take a value/computation and yield a value/computation.
- Secondly, we need to consider what simpler notion of equivalence to use and how to restrict program contexts so that we can use it. A common approach is to consider only contexts that have some atomic type as a result, such as naturals or integers, where syntactic equality is appropriate. We believe that approach works here too. Indeed, we can expect that an appropriate computation context handles all operations and yields a pure program.
- Thirdly, we do not want to admit all possible NOEFF contexts. In particular, we do not want to admit those that get stuck because of an inappropriate use of an `unsafe` coercion. Hence, we want to restrict ourselves to those that are the image of a EXEFF program context.

We leave working out the precise formal definition of contextual equivalence and proving semantic preservation on top of it a substantial open challenge. Yet, we point to the work of (Bi *et al.*, 2018) as an important source of inspiration. They also deal with an elaboration-based setting, for disjoint intersection types, and use logical relations as the basis of their proofs.

8 Related work and conclusion

Eff's Implicit Type System. The most closely related work is that of Pretnar (2014) on inferring algebraic effects for Eff, which is the basis for our implicitly typed IMPEFF calculus, its type system and the type inference algorithm. There are three major differences with Pretnar's inference algorithm.

Firstly, our work introduces an explicitly typed calculus. For this reason, we have extended the constraint generation phase with the elaboration into EXEFF and the constraint solving phase with the construction of coercions.

Secondly, we add skeletons to guarantee erasure. Skeletons also allow us to use the standard occurs check during unification. In contrast, unification in Pretnar's algorithm performs the occurs check up to the equivalence closure of the subtyping relation (Fuh & Mishra, 1990; Simonet, 2003) and needs to take care of appropriately instantiating all variables in an equivalence class (also called a skeleton). As these classes turn out to be surrogates for the underlying skeleton types, we have decided to keep the name. Traytel *et al.* (2011) propose an alternative approach and first perform a weak unification algorithm, which is unification with the standard occurs check on what are essentially skeletons, although this is not an explicit concept in their work.

Finally, Pretnar incorporates garbage collection of constraints (Pottier, 2001). The aim of this approach is to obtain unique and simple type schemes by eliminating redundant constraints. Garbage collection is not suitable for our use as type variables and coercions witnessing subtyping constraints cannot simply be dropped, but must be instantiated in a suitable manner, which cannot be done in general.

Consider for instance a situation with type variables $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ and α_5 where $\alpha_1 \leqslant \alpha_3$, $\alpha_2 \leqslant \alpha_3$, $\alpha_3 \leqslant \alpha_4$ and $\alpha_3 \leqslant \alpha_5$. Suppose that α_3 does not appear in the type, then garbage

collection would eliminate it and replace the constraints by $\alpha_1 \leqslant \alpha_4$, $\alpha_2 \leqslant \alpha_4$, $\alpha_1 \leqslant \alpha_5$, and $\alpha_2 \leqslant \alpha_5$. While garbage collection guarantees that for any ground instantiation of the remaining type variables, there exists a valid ground instantiation for α_3, ExEff would need to be extended with joins (or meets) to express a generically valid instantiation like $\alpha_1 \sqcup \alpha_2$. Moreover, we would need additional coercion formers to establish $\alpha_1 \leqslant (\alpha_1 \sqcup \alpha_2)$ or $(\alpha_1 \sqcup \alpha_2) \leqslant \alpha_4$.

As these additional constructs considerably complicate the calculus, we propose a simpler solution, especially as we have experienced no blow-up in inference times during our initial experiments. We use ExEff as it is for internal purposes, but display types to programmers in their garbage-collected form.

Calculi with Explicit Coercions. The notion of explicit coercions is not new; Mitchell (1984) introduced the idea of inserting coercions during type inference for ML-based languages, as a means for explicit casting between different numeric types.

Breazu-Tannen *et al.* (1991) also present a translation of languages with inheritance polymorphism into System F, extended with coercions. Although their coercion combinators are very similar to our coercion forms, Breazu-Tannen *et al.*'s coercions are terms and thus cannot be erased.

Much closer to ExEff is Crary's coercion calculus for inclusive subtyping (Crary, 2000), from which we borrowed the stratification of value results. Though the coercion calculus does not support coercion abstraction and other coercion forms that we need for supporting effects, coercions in Crary's system are also erasable so they have no runtime effect.

System F_C (Sulzmann *et al.*, 2007) uses explicit type-equality coercions to encode complex language features (e.g., GADTs Peyton Jones *et al.*, 2006 or type families Schrijvers *et al.*, 2008). Though ExEff's coercions are proofs of subtyping rather than type equality, our system has a lot in common with it, and in particular the "push" rules. A difference between the two lies in the presence of inversion coercions (i.e., coercions that allow for decomposition of type inequalities), which System F_C (and our own earlier work (Saleh *et al.*, 2018)) includes.

The NoEff `unsafe` coercion shows similarities with downcasts in object-oriented languages and calculi like Featherweight Java (Igarashi *et al.*, 2001), which get stuck when the object is from the wrong class. A difference to Featherweight Java is that, when successful, `unsafe` also destructures a value. This shares similarities with the explicit coercions in the recent backend calculi for (disjoint) intersection types (Oliveira *et al.*, 2016; Bi *et al.*, 2018), which also extract relevant components from composite values.

Future Work. Our plans focus on resuming the postponed work on efficient compilation of handlers. First, we intend to adjust program transformations to the explicit type information. We hope that this will not only make the optimiser more robust, but also expose new optimisation opportunities. Next, we plan to write compilers to both Multicore OCaml and standard OCaml. Finally, once the compiler shows promising preliminary results, we plan to extend it to other Eff features such as user-defined types or recursion, allowing us to benchmark it on more realistic programs.

Acknowledgements

Conflicts of Interest

None.

References

Barendregt, H. (1981) *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland.

Bauer, A., Hofmann, M., Pretnar, M. & Yallop, J. (2016) From theory to practice of algebraic effects and handlers (dagstuhl seminar 16112). *Dagstuhl Reports* **6**(3), 44–58.

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

Bi, X., Oliveira, B. C. d. S. & Schrijvers, T. (2018) The essence of nested composition. In 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16–21, 2018, Amsterdam, The Netherlands, Millstein, T. D. (ed). LIPIcs, vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 22:1–22:33.

Breazu-Tannen, V., Coquand, T., Gunter, C. A. & Scedrov, A. (1991) Inheritance as implicit coercion. *Inf. Comput.* **93**, 172–221.

Chandrasekaran, S. K., Leijen, D., Pretnar, M. & Schrijvers, T. (2018). Algebraic effect handlers go mainstream (dagstuhl seminar 18172). *Dagstuhl Reports* **8**(4), 104–125.

Crary, K. (2000). Typed compilation of inclusive subtyping. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. ICFP'00. NY, USA: ACM, pp. 68–81.

Damas, L. & Milner, R. (1982) Principal type-schemes for functional programs. In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'82. NY, USA: ACM, pp. 207–212.

Dolan, S., White, L., Sivaramakrishnan, K. C., Yallop, J. & Madhavapeddy, A. (2015) Effective concurrency through algebraic effects. In OCaml Workshop.

Fuh, Y.-C. & Mishra, P. (1990) Type inference with subtypes. *Theor. Comput. Sci.* **73**(2), 155–175.

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures de larithmétique dordre supérieur*. PhD thesis, Université Paris VII.

Girard, J.-Y., Taylor, P. & Lafont, Y. (1989) *Proofs and Types*. Cambridge University.

Henglein, F. (1994) Dynamic typing: Syntax and proof theory. In Selected Papers of the Symposium on Fourth European Symposium on Programming. ESOP'92. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., pp. 197–230.

Hillerström, D. & Lindley, S. (2016) Liberating effects with rows and handlers. In Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016, Chapman, J. & Swierstra, W. (eds). ACM, pp. 15–27.

Hindley, R. (1969) The principal type-scheme of an object in combinatory logic. *Trans. Am. Math. Soc.* **146**, 29–60.

Igarashi, A., Pierce, B. C. & Wadler, P. (2001) Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.* **23**(3), 396–450.

Jones, M. P. (1992) A theory of qualified types. In ESOP'92, 4th European Symposium on Programming, Rennes, France, February 26–28, 1992, Proceedings, Krieg-Brückner, B. (ed). Lecture Notes in Computer Science, vol. 582. Springer, pp. 287–306.

Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming. ICFP'14. ACM, pp. 145–158.

Kiselyov, O. & Sivaramakrishnan, K. C. (2016) Eff directly in OCaml. In OCaml Workshop.

Leijen, D. (2014) Koka: Programming with row polymorphic effect types. In Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014, Levy, P. & Krishnaswami, N. (eds). EPTCS, vol. 153, pp. 100–126.

Leijen, D. (2017) Type directed compilation of row-typed algebraic effects. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017, Castagna, G. & Gordon, A. D. (eds). ACM, pp. 486–499.

Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017, Castagna, G. & Gordon, A. D. (eds). ACM, pp. 500–514.

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

Mitchell, J. C. (1984) Coercion and type inference. In Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL'84. New York, NY, USA: ACM, pp. 175–185.

Morris Jr., J. H. (1969) *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology.

Oliveira, B. C. d. S., Shi, Z. & Alpuim, J. (2016) Disjoint intersection types. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016, Garrigue, J., Keller, G. & Sumii, E. (eds). ACM, pp. 364–377.

Peyton Jones, S., Vytiniotis, D., Weirich, S. & Washburn, G. (2006) Simple unification-based type inference for GADTs. In ICFP'06.

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

Plotkin, G. D. & Pretnar, M. (2013) Handling algebraic effects. *Log. Methods Comput. Sci* **9**(4), 1–36.

Pottier, F. (2001) Simplifying subtyping constraints: A theory. *Inf. Comput.* **170**(2), 153–183.

Pretnar, M. (2014) Inferring algebraic effects. *Log. Methods Comput. Sci.* **10**(3), 1–43.

Pretnar, M. (2015) An introduction to algebraic effects and handlers, invited tutorial. *Electron. Notes Theor. Comput. Sci.* **319**, 19–35.

Pretnar, M., Saleh, A. H., Faes, A. & Schrijvers, T. (2017) *Efficient Compilation of Algebraic Effects and Handlers*. Technical report. CW 708. KU Leuven Department of Computer Science.

Reynolds, J. C. (1974) Towards a theory of type structure. In Programming Symposium, Proceedings Colloque Sur La Programmation. London, UK: Springer-Verlag, pp. 408–423.

Reynolds, J. C. (1983) Types, abstraction, and parametric polymorphism. In *Information Processing*, Mason, R.E.A. (ed), vol. 83, pp. 513–523.

Saleh, A. H., Karachalias, G., Pretnar, M. & Schrijvers, T. (2018) Explicit effect subtyping. In ESOP. Lecture Notes in Computer Science, vol. 10801. Springer, pp. 327–354.

Schrijvers, T., Peyton Jones, S., Chakravarty, M. & Sulzmann, M. (2008) Type checking with open type functions. In ICFP'08. ACM, pp. 51–62.

Simonet, V. (2003) Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27–29, 2003, Proceedings, Ohori, A. (ed), pp. 283–302. Springer.

Sulzmann, M., Chakravarty, M. M. T., Peyton Jones, S. & Donnelly, K. (2007) System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. TLDI'07. New York, NY, USA: ACM, pp. 53–66.

Traytel, D., Berghofer, S. & Nipkow, T. (2011) Extending hindley-milner type inference with coercive structural subtyping. In *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5–7, 2011. Proceedings*, Yang, H. (ed). Lecture Notes in Computer Science, vol. 7078. Springer, pp. 89–104.

Wansbrough, K. & Peyton Jones, S. L. (1999) Once upon a polymorphic type. In *POPL*. ACM, pp. 15–28.

Appendix A IMPEFF Additional Judgements

Type Well-formedness and Elaboration. Since our system discriminates between value types and computation types, well-formedness of types is checked using two mutually recursive relations: $\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T$ (values) and $\Gamma \vdash_{\text{cty}} \underline{C} : \tau \rightsquigarrow \underline{C}$ (computations). We discuss each one separately below.

Well-formedness for value types is given by the following rules:

$$\frac{\alpha : \tau \in \Gamma}{\Gamma \vdash_{\text{vty}} \alpha : \tau \rightsquigarrow \alpha} \qquad \frac{\Gamma \vdash_{\text{vty}} A : \tau_1 \rightsquigarrow T \qquad \Gamma \vdash_{\text{cty}} \underline{C} : \tau_2 \rightsquigarrow \underline{C}}{\Gamma \vdash_{\text{vty}} A \rightarrow \underline{C} : \tau_1 \rightarrow \tau_2 \rightsquigarrow T \rightarrow \underline{C}}$$

$$\frac{\Gamma \vdash_{\text{cty}} \underline{C} : \tau_1 \rightsquigarrow \underline{C}_1 \qquad \Gamma \vdash_{\text{cty}} \underline{D} : \tau_2 \rightsquigarrow \underline{C}_2}{\Gamma \vdash_{\text{vty}} \underline{C} \Rightarrow \underline{D} : \tau_1 \Rightarrow \tau_2 \rightsquigarrow \underline{C}_1 \Rightarrow \underline{C}_2} \qquad \frac{}{\Gamma \vdash_{\text{vty}} \text{Unit} : \text{Unit} \rightsquigarrow \text{Unit}}$$

$$\frac{\Gamma \vdash_{\text{ct}} \pi \rightsquigarrow \pi \qquad \Gamma \vdash_{\text{vty}} K : \tau \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \pi \Rightarrow K : \tau \rightsquigarrow \pi \Rightarrow T} \qquad \frac{\Gamma, \alpha : \tau_1 \vdash_{\text{vty}} S : \tau_2 \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \forall \alpha : \tau_1.S : \tau_2 \rightsquigarrow \forall \alpha : \tau_1.T}$$

$$\frac{\Gamma, \delta \vdash_{\text{vty}} S : \tau \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \forall \delta.S : \tau \rightsquigarrow \forall \delta.T} \qquad \frac{\Gamma, \varsigma \vdash_{\text{vty}} S : \tau \rightsquigarrow T}{\Gamma \vdash_{\text{vty}} \forall \varsigma.S : \forall \varsigma.\tau \rightsquigarrow \forall \varsigma.T}$$

The judgement is syntax-directed on the structure of types; each rule corresponds to a value type syntactic form. Since EXEFF types are a superset of IMPEFF types, the elaboration part (highlighted in grey) is the identity transformation. Hence, the essence of the judgement is to check the well-scopedness of source types.

Well-formedness for computation types is given by the following rule:

$$\frac{\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T \qquad \Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\text{cty}} A \,!\, \Delta : \tau \rightsquigarrow T \,!\, \Delta}$$

We ensure that both parts of a computation type (the value type and the dirt) are well scoped under Γ, while elaborating the value type into a proper EXEFF representation.

Constraint Well-formedness and Elaboration. Well-formedness for constraints is given by judgement $\Gamma \vdash_{\text{ct}} \rho \rightsquigarrow \rho$, given by the following rules:

$$\frac{\Gamma \vdash_{\text{vty}} A : \tau \rightsquigarrow T_1 \qquad \Gamma \vdash_{\text{vty}} B : \tau \rightsquigarrow T_2}{\Gamma \vdash_{\text{ct}} A \leqslant B \rightsquigarrow T_1 \leqslant T_2} \qquad \frac{\Gamma \vdash_{\text{cty}} \underline{C} : \tau \rightsquigarrow \underline{C}_1 \qquad \Gamma \vdash_{\text{cty}} \underline{D} : \tau \rightsquigarrow \underline{C}_2}{\Gamma \vdash_{\text{ct}} \underline{C} \leqslant \underline{D} \rightsquigarrow \underline{C}_1 \leqslant \underline{C}_2}$$

$$\frac{\Gamma \vdash_{\Delta} \Delta_1 \qquad \Gamma \vdash_{\Delta} \Delta_2}{\Gamma \vdash_{\text{ct}} \Delta_1 \leqslant \Delta_2 \rightsquigarrow \Delta_1 \leqslant \Delta_2}$$

Since the dirt syntax is shared between IMPEFF and EXEFF, all three rules check the constraint components for well-scopedness, but only the type-related constraints are elaborated: the elaboration of a dirt constraint is the identity.

Dirt Well-formedness. Judgement $\Gamma \vdash_{\Delta} \Delta$ checks dirt well-formedness and is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Delta} \emptyset} \qquad \frac{\delta \in \Gamma}{\Gamma \vdash_{\Delta} \delta} \qquad \frac{(\text{Op} : A_{\text{Op}} \to B_{\text{Op}}) \in \Sigma \qquad \Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\Delta} \{\text{Op}\} \cup \Delta}$$

In addition to checking that the dirt is well scoped (illustrated by the second rule), we also make sure that all operations in a dirt set are already defined, by looking them up in the globally visible signature Σ (third rule).

Skeleton Well-formedness. Finally, skeleton well-formedness is performed by judgement $\Gamma \vdash_{\tau} \tau$, as given by the following rules:

$$\frac{\varsigma \in \Gamma}{\Gamma \vdash_{\tau} \varsigma} \qquad \frac{}{\Gamma \vdash_{\tau} \text{Unit}} \qquad \frac{\Gamma \vdash_{\tau} \tau_1 \qquad \Gamma \vdash_{\tau} \tau_2}{\Gamma \vdash_{\tau} \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_{\tau} \tau_1 \qquad \Gamma \vdash_{\tau} \tau_2}{\Gamma \vdash_{\tau} \tau_1 \Rightarrow \tau_2}$$

Since skeletons are uni-kinded, this relation is entirely straightforward and is in fact identical to the well-formedness of System F simple types.

Appendix B EXEFF Additional Judgements

Type Well-formedness. Again, preserving the separation between value and computation types, EXEFF comes with two mutually recursive relation for checking the well-formedness of types: $\Gamma \vdash_{T} T : \tau$ (values) and $\Gamma \vdash_{\underline{C}} \underline{C} : \tau$ (computations). We discuss each one separately.

Well-formedness for value types is given by the following rules:

$$\frac{(\alpha : \tau) \in \Gamma}{\Gamma \vdash_{T} \alpha : \tau} \qquad \frac{\Gamma \vdash_{T} T : \tau_1 \qquad \Gamma \vdash_{\underline{C}} \underline{C} : \tau_2}{\Gamma \vdash_{T} T \to \underline{C} : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_{\underline{C}} \underline{C}_1 : \tau_1 \qquad \Gamma \vdash_{\underline{C}} \underline{C}_2 : \tau_2}{\Gamma \vdash_{T} \underline{C}_1 \Rightarrow \underline{C}_2 : \tau_1 \Rightarrow \tau_2}$$

$$\frac{}{\Gamma \vdash_{T} \text{Unit} : \text{Unit}} \qquad \frac{\Gamma \vdash_{\rho} \pi \qquad \Gamma \vdash_{T} T : \tau}{\Gamma \vdash_{T} \pi \Rightarrow T : \tau} \qquad \frac{\Gamma, \varsigma \vdash_{T} T : \tau}{\Gamma \vdash_{T} \forall \varsigma.T : \forall \varsigma.\tau} \qquad \frac{\Gamma, \alpha : \tau_1 \vdash_{T} T : \tau_2}{\Gamma \vdash_{T} \forall \alpha : \tau_1.T : \tau_2}$$

$$\frac{\Gamma, \delta \vdash_{T} T : \tau}{\Gamma \vdash_{T} \forall \delta.T : \tau}$$

The relation is almost identical to the corresponding one for IMPEFF value types. The only difference between the two lies in the ExEFF's impredicative polymorphism and higher-rank types.

Similarly, well-formedness of computation types is checked via relation $\Gamma \vdash_{\underline{C}} \underline{C} : \tau$, given by a single rule, which is identical to the corresponding one of IMPEFF:

$$\frac{\Gamma \vdash_T T : \tau \qquad \Gamma \vdash_\Delta \Delta}{\Gamma \vdash_{\underline{C}} T \,!\, \Delta : \tau}$$

The only difference, again, is that instead of a monotype A, computation types are allowed to refer to arbitrary System F types T.

Constraint Well-formedness. Well-formedness for constraints is given by judgement $\Gamma \vdash_\rho \rho$:

$$\frac{\Gamma \vdash_T T_1 : \tau \qquad \Gamma \vdash_T T_2 : \tau}{\Gamma \vdash_\rho T_1 \leqslant T_2} \qquad \frac{\Gamma \vdash_{\underline{C}} \underline{C}_1 : \tau \qquad \Gamma \vdash_{\underline{C}} \underline{C}_2 : \tau}{\Gamma \vdash_\rho \underline{C}_1 \leqslant \underline{C}_2} \qquad \frac{\Gamma \vdash_\Delta \Delta_1 \qquad \Gamma \vdash_\Delta \Delta_2}{\Gamma \vdash_\rho \Delta_1 \leqslant \Delta_2}$$

Dirt Well-formedness. Dirt well-formedness takes the form $\Gamma \vdash_\Delta \Delta$ and is given by the following rules:

$$\frac{}{\Gamma \vdash_\Delta \emptyset} \qquad \frac{\delta \in \Gamma}{\Gamma \vdash_\Delta \delta} \qquad \frac{(\mathrm{Op} : T_1 \to T_2) \in \Sigma \qquad \Gamma \vdash_\Delta \Delta}{\Gamma \vdash_\Delta \{\mathrm{Op}\} \cup \Delta}$$

The only difference with the corresponding relation for IMPEFF is that instead of operations Op having IMPEFF types, they now have ExEFF types. We abuse notation and use Σ for both the IMPEFF and the ExEFF top-level signature set.

Skeleton Well-formedness. Skeleton well-formedness is checked via relation $\Gamma \vdash_\tau \tau$, given by the following rules:

$$\frac{\varsigma \in \Gamma}{\Gamma \vdash_\tau \varsigma} \qquad \frac{}{\Gamma \vdash_\tau \mathrm{Unit}} \qquad \frac{\Gamma \vdash_\tau \tau_1 \qquad \Gamma \vdash_\tau \tau_2}{\Gamma \vdash_\tau \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\tau \tau_1 \qquad \Gamma \vdash_\tau \tau_2}{\Gamma \vdash_\tau \tau_1 \Rightarrow \tau_2} \qquad \frac{\Gamma, \varsigma \vdash_\tau \tau}{\Gamma \vdash_\tau \forall \varsigma . \tau}$$

The only noticeable difference between this judgement and the corresponding for IMPEFF skeletons, is captured in the last rule. We have opted for a System F-based skeleton structure, thus this relation is identical to the well-formedness of System F types.

Coercion Typing. Coercion typing is presented in Figure B1 and formalises the intuitive interpretation of coercions we gave in Section 4.1.

Reflexivity of Arbitrary Types. Function *reflOf* $(\cdot)$ below shows how to create a reflexivity coercion for an arbitrary value type, computation type or dirt:

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \rho}$ **Coercion Typing**

$$\frac{(\omega:\pi)\in\Gamma}{\Gamma\vdash_{\mathsf{co}}\omega:\pi} \qquad \frac{\Gamma\vdash_{T}\alpha:\tau}{\Gamma\vdash_{\mathsf{co}}\langle\alpha\rangle:\alpha\leqslant\alpha} \qquad \frac{\Gamma\vdash_{\Delta}\Delta}{\Gamma\vdash_{\mathsf{co}}\langle\Delta\rangle:\Delta\leqslant\Delta} \qquad \frac{}{\Gamma\vdash_{\mathsf{co}}\langle\mathtt{Unit}\rangle:\mathtt{Unit}\leqslant\mathtt{Unit}}$$

$$\frac{\Gamma\vdash_{\mathsf{co}}\gamma_1:T_2\leqslant T_1 \quad \Gamma\vdash_{\mathsf{co}}\gamma_2:\underline{C}_1\leqslant\underline{C}_2}{\Gamma\vdash_{\mathsf{co}}\gamma_1\to\gamma_2:T_1\to\underline{C}_1\leqslant T_2\to\underline{C}_2} \qquad \frac{\Gamma\vdash_{\mathsf{co}}\gamma_1:\underline{C}_3\leqslant\underline{C}_1 \quad \Gamma\vdash_{\mathsf{co}}\gamma_2:\underline{C}_2\leqslant\underline{C}_4}{\Gamma\vdash_{\mathsf{co}}\gamma_1\Rightarrow\gamma_2:\underline{C}_1\Rightarrow\underline{C}_2\leqslant\underline{C}_3\Rightarrow\underline{C}_4}$$

$$\frac{\Gamma,\varsigma\vdash_{\mathsf{co}}\gamma:T_1\leqslant T_2}{\Gamma\vdash_{\mathsf{co}}\forall\varsigma.\gamma:\forall\varsigma.T_1\leqslant\forall\varsigma.T_2} \qquad \frac{\Gamma,\alpha:\tau\vdash_{\mathsf{co}}\gamma:T_1\leqslant T_2}{\Gamma\vdash_{\mathsf{co}}\forall\alpha:\tau.\gamma:\forall\alpha:\tau.T_1\leqslant\forall\alpha:\tau.T_2}$$

$$\frac{\Gamma,\delta\vdash_{\mathsf{co}}\gamma:T_1\leqslant T_2}{\Gamma\vdash_{\mathsf{co}}\forall\delta.\gamma:\forall\delta.T_1\leqslant\forall\delta.T_2} \qquad \frac{\Gamma\vdash_{\mathsf{co}}\gamma:T_1\leqslant T_2 \quad \Gamma\vdash_{\rho}\pi}{\Gamma\vdash_{\mathsf{co}}\pi\Rightarrow\gamma:\pi\Rightarrow T_1\leqslant\pi\Rightarrow T_2} \qquad \frac{\Gamma\vdash_{\Delta}\Delta}{\Gamma\vdash_{\mathsf{co}}\emptyset_{\Delta}:\emptyset\leqslant\Delta}$$

$$\frac{\Gamma\vdash_{\mathsf{co}}\gamma_1:T_1\leqslant T_2 \quad \Gamma\vdash_{\mathsf{co}}\gamma_2:\Delta_1\leqslant\Delta_2}{\Gamma\vdash_{\mathsf{co}}\gamma_1\,!\,\gamma_2:T_1\,!\,\Delta_1\leqslant T_2\,!\,\Delta_2} \qquad \frac{\Gamma\vdash_{\mathsf{co}}\gamma:\Delta_1\leqslant\Delta_2 \quad (\mathtt{Op}:T_1\to T_2)\in\Sigma}{\Gamma\vdash_{\mathsf{co}}\{\mathtt{Op}\}\cup\gamma:\{\mathtt{Op}\}\cup\Delta_1\leqslant\{\mathtt{Op}\}\cup\Delta_2}$$

Fig. B1. EXEFF coercion typing.

$\boxed{v\leadsto_{\mathsf{v}} v'}$ **Values++**

$$\frac{v\leadsto_{\mathsf{v}} v'}{v\triangleright\gamma\leadsto_{\mathsf{v}} v'\triangleright\gamma} \qquad v^R\triangleright\langle\mathtt{Unit}\rangle\leadsto_{\mathsf{v}} v^R \qquad \frac{v\leadsto_{\mathsf{v}} v'}{v\,\tau\leadsto_{\mathsf{v}} v'\,\tau} \qquad \frac{v\leadsto_{\mathsf{v}} v'}{v\,T\leadsto_{\mathsf{v}} v'\,T}$$

$$\frac{v\leadsto_{\mathsf{v}} v'}{v\,\Delta\leadsto_{\mathsf{v}} v'\,\Delta} \qquad \frac{v\leadsto_{\mathsf{v}} v'}{v\,\gamma\leadsto_{\mathsf{v}} v'\,\gamma} \qquad (v^R\triangleright(\forall\varsigma.\gamma))\,\tau\leadsto_{\mathsf{v}} v^R\,\tau\triangleright\gamma[\tau/\varsigma]$$

$$(v^R\triangleright(\forall(\alpha:\tau).\gamma))\,T\leadsto_{\mathsf{v}} v^R\,T\triangleright\gamma[T/\alpha] \qquad (v^R\triangleright(\forall\delta.\gamma))\,\Delta\leadsto_{\mathsf{v}} v^R\,\Delta\triangleright\gamma[\Delta/\delta]$$

$$(v^R\triangleright(\pi\Rightarrow\gamma_1))\,\gamma_2\leadsto_{\mathsf{v}} v^R\,\gamma_2\triangleright\gamma_1 \qquad (\Lambda\varsigma.v)\,\tau\leadsto_{\mathsf{v}} v[\tau/\varsigma] \qquad (\Lambda\alpha:\tau.v)\,T\leadsto_{\mathsf{v}} v[T/\alpha]$$

$$(\Lambda\delta.v)\,\Delta\leadsto_{\mathsf{v}} v[\Delta/\delta] \qquad (\Lambda(\omega:\pi).v)\,\gamma\leadsto_{\mathsf{v}} v[\gamma/\omega]$$

Fig. B2. EXEFF operational semantics (Values).

$$\begin{aligned}
reflOf(\alpha) &= \langle\alpha\rangle & reflOf(T\,!\,\Delta) &= reflOf(T)\,!\,reflOf(\Delta)\\
reflOf(\mathtt{Unit}) &= \langle\mathtt{Unit}\rangle\\
reflOf(T\to\underline{C}) &= reflOf(T)\to reflOf(\underline{C})\\
reflOf(\underline{C}_1\Rightarrow\underline{C}_2) &= reflOf(\underline{C}_1)\Rightarrow reflOf(\underline{C}_2)\\
reflOf(\forall\varsigma.T) &= \forall\varsigma.reflOf(T)\\
reflOf(\forall\alpha:\tau.T) &= \forall\alpha:\tau.reflOf(T) & reflOf(\delta) &= \langle\delta\rangle\\
reflOf(\forall\delta.T) &= \forall\delta.reflOf(T) & reflOf(\emptyset) &= \emptyset_{\emptyset}\\
reflOf(\pi\Rightarrow T) &= \pi\Rightarrow reflOf(T) & reflOf(\{\mathtt{Op}\}\cup\Delta) &= \{\mathtt{Op}\}\cup reflOf(\Delta)
\end{aligned}$$

Operational Semantics. The complete small-step, call-by-value operational semantics for EXEFF can be found in Figures B2 (values) and B3 (computations).

$\boxed{c \leadsto_c c'}$ **Computations**

$$\frac{c \leadsto_c c'}{c \rhd \gamma \leadsto_c c' \rhd \gamma} \qquad \frac{v_1 \leadsto_v v_1'}{v_1\, v_2 \leadsto_c v_1'\, v_2} \qquad (v^R \rhd (\gamma_1 \to \gamma_2))\, v \leadsto_c (v^R\, (v \rhd \gamma_1)) \rhd \gamma_2$$

$$\frac{v_2 \leadsto_v v_2'}{v^T\, v_2 \leadsto_c v^T\, v_2'} \qquad (\mathtt{fun}\, x : T \mapsto c)\, v^R \leadsto_c c[v^R/x] \qquad \frac{v \leadsto_v v'}{\mathtt{let}\, x = v\, \mathtt{in}\, c \leadsto_c \mathtt{let}\, x = v'\, \mathtt{in}\, c}$$

$$\mathtt{let}\, x = v^R\, \mathtt{in}\, c \leadsto_c c[v^R/x] \qquad \frac{v \leadsto_v v'}{\mathtt{return}\, v \leadsto_c \mathtt{return}\, v'} \qquad \frac{v \leadsto_v v'}{\mathtt{Op}\, v\, (y : T.c) \leadsto_c \mathtt{Op}\, v'\, (y : T.c)}$$

$$(\mathtt{Op}\, v^R\, (x : T.c)) \rhd \gamma \leadsto_c \mathtt{Op}\, v^R\, (x : T.(c \rhd \gamma)) \qquad \frac{c_1 \leadsto_c c_1'}{\mathtt{do}\, x \leftarrow c_1; c_2 \leadsto_c \mathtt{do}\, x \leftarrow c_1'; c_2}$$

$$\mathtt{do}\, x \leftarrow ((\mathtt{return}\, v^R) \rhd (\gamma_1\, !\, \gamma_1') \rhd \ldots \rhd (\gamma_n\, !\, \gamma_n')); c_2 \leadsto_c c_2[(v^R \rhd \gamma_1 \rhd \ldots \rhd \gamma_n)/x]$$

$$\mathtt{do}\, x \leftarrow \mathtt{Op}\, v^R\, (y : T.c_1); c_2 \leadsto_c \mathtt{Op}\, v^R\, (y : T.\mathtt{do}\, x \leftarrow c_1; c_2)$$

$$\frac{v \leadsto_v v'}{\mathtt{handle}\, c\, \mathtt{with}\, v \leadsto_c \mathtt{handle}\, c\, \mathtt{with}\, v'}$$

$$\mathtt{handle}\, c\, \mathtt{with}\, (v^R \rhd (\gamma_1 \Rightarrow \gamma_2)) \leadsto_c (\mathtt{handle}\, (c \rhd \gamma_1)\, \mathtt{with}\, v^R) \rhd \gamma_2$$

$$\frac{c \leadsto_c c'}{\mathtt{handle}\, c\, \mathtt{with}\, v^T \leadsto_c \mathtt{handle}\, c'\, \mathtt{with}\, v^T}$$

$$\mathtt{handle}\, ((\mathtt{return}\, v^R) \rhd (\gamma_1\, !\, \gamma_1') \rhd \ldots \rhd (\gamma_n\, !\, \gamma_n'))\, \mathtt{with}\, h \leadsto_c c_r[(v^R \rhd \gamma_1 \rhd \ldots \rhd \gamma_n)/x]$$

$$\mathtt{handle}\, (\mathtt{Op}\, v^R\, (y : T.c))\, \mathtt{with}\, h \leadsto_c c_{\mathtt{Op}}[v^R/x, (\mathtt{fun}\, (y : T) \mapsto \mathtt{handle}\, c\, \mathtt{with}\, h)/k]$$

$$\mathtt{handle}\, (\mathtt{Op}\, v^R\, (y : T.c))\, \mathtt{with}\, h \leadsto_c \mathtt{Op}\, v^R\, (y : T.\mathtt{handle}\, c\, \mathtt{with}\, h)$$

Fig. B3. EXEFF operational semantics (Computations).

Appendix C   Type Inference and Elaboration: Additional Judgements

Elaboration of Types, Constraints and Typing Environments. Below we give the definitions of elaboration functions $elab_S(S)$, $elab_{\underline{C}}(\underline{C})$, $elab_\rho(\rho)$ and $elab_\Gamma(\Gamma)$, for value types, computation types, constraints and typing environments.

$$
\begin{aligned}
elab_S(\alpha) &= \alpha & elab_{\underline{C}}(A\, !\, \Delta) &= elab_S(A)\, !\, \Delta \\
elab_S(A \to \underline{C}) &= elab_S(A) \to elab_{\underline{C}}(\underline{C}) & & \\
elab_S(\underline{C} \Rightarrow \underline{D}) &= elab_{\underline{C}}(\underline{C}) \Rightarrow elab_{\underline{C}}(\underline{D}) & elab_\Gamma(\varepsilon) &= \varepsilon \\
elab_S(\mathtt{Unit}) &= \mathtt{Unit} & elab_\Gamma(\Gamma, \varsigma) &= elab_\Gamma(\Gamma), \varsigma \\
elab_S(\forall \varsigma.S) &= \forall \varsigma.elab_S(S) & elab_\Gamma(\Gamma, \alpha : \tau) &= elab_\Gamma(\Gamma), \alpha : \tau \\
elab_S(\forall \alpha : \tau.S) &= \forall \alpha : \tau.elab_S(S) & elab_\Gamma(\Gamma, \delta) &= elab_\Gamma(\Gamma), \delta \\
elab_S(\forall \delta.S) &= \forall \delta.elab_S(S) & elab_\Gamma(\Gamma, x : S) &= elab_\Gamma(\Gamma), x : elab_S(S) \\
elab_S(\pi \Rightarrow K) &= elab_\rho(\pi) \Rightarrow elab_S(K) & elab_\Gamma(\Gamma, \omega : \rho) &= elab_\Gamma(\Gamma), \omega : elab_\rho(\rho)
\end{aligned}
$$

$$
\begin{aligned}
elab_\rho(A \leqslant B) &= elab_S(A) \leqslant elab_S(B) \\
elab_\rho(\underline{C} \leqslant \underline{D}) &= elab_{\underline{C}}(\underline{C}) \leqslant elab_{\underline{C}}(\underline{D}) \\
elab_\rho(\Delta_1 \leqslant \Delta_2) &= \Delta_1 \leqslant \Delta_2
\end{aligned}
$$

$$\text{typing environment } \Gamma ::= \varepsilon \mid \Gamma, \varsigma \mid \Gamma, x : \tau$$

$\boxed{\Gamma \vdash_{\text{ev}} v : \tau}$ **Values**

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{\text{ev}} x : \tau} \qquad \frac{}{\Gamma \vdash_{\text{ev}} \text{unit} : \text{Unit}} \qquad \frac{\Gamma, x : \tau_1 \vdash_{\text{ec}} c : \tau_2 \qquad \Gamma \vdash_{\overline{\tau}} \tau_1}{\Gamma \vdash_{\text{ev}} (\text{fun } x : \tau_1 \mapsto c) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash_{\text{ev}} v : \forall \varsigma . \tau_1 \qquad \Gamma \vdash_{\overline{\tau}} \tau_2}{\Gamma \vdash_{\text{ev}} v \; \tau_2 : \tau_1 [\tau_2 / \varsigma]} \qquad \frac{\Gamma, \varsigma \vdash_{\text{ev}} v : \tau}{\Gamma \vdash_{\text{ev}} \Lambda \varsigma . v : \forall \varsigma . \tau}$$

$$\frac{\Gamma, x : \tau_x \vdash_{\text{ec}} c_r : \tau \qquad [(\text{Op} : \tau_1 \to \tau_2) \in \Sigma \qquad \Gamma, x : \tau_1, k : \tau_2 \to \tau \vdash_{\text{ec}} c_{\text{Op}} : \tau]_{\text{Op} \in \mathcal{O}}}{\Gamma \vdash_{\text{ev}} \{\text{return } (x : \tau_x) \mapsto c_r, [\text{Op } x \, k \mapsto c_{\text{Op}}]_{\text{Op} \in \mathcal{O}}\} : \tau_x \Rrightarrow \tau}$$

$\boxed{\Gamma \vdash_{\text{ec}} c : \tau}$ **Computations**

$$\frac{\Gamma \vdash_{\text{ev}} v_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\text{ev}} v_2 : \tau_1}{\Gamma \vdash_{\text{ec}} v_1 \; v_2 : \tau_2} \qquad \frac{\Gamma \vdash_{\text{ev}} v : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\text{ec}} c : \tau_2}{\Gamma \vdash_{\text{ec}} \text{let } x = v \text{ in } c : \tau_2} \qquad \frac{\Gamma \vdash_{\text{ev}} v : \tau}{\Gamma \vdash_{\text{ec}} \text{return } v : \tau}$$

$$\frac{(\text{Op} : \tau_1 \to \tau_2) \in \Sigma \qquad \Gamma \vdash_{\text{ev}} v : \tau_1 \qquad \Gamma, y : \tau_2 \vdash_{\text{ec}} c : \tau}{\Gamma \vdash_{\text{ec}} \text{Op } v \; (y : \tau_2 . c) : \tau} \qquad \frac{\Gamma \vdash_{\text{ec}} c_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\text{ec}} c_2 : \tau_2}{\Gamma \vdash_{\text{ec}} \text{do } x \leftarrow c_1 ; c_2 : \tau_2}$$

$$\frac{\Gamma \vdash_{\text{ev}} v : \tau_1 \Rrightarrow \tau_2 \qquad \Gamma \vdash_{\text{ec}} c : \tau_1}{\Gamma \vdash_{\text{ec}} \text{handle } c \text{ with } v : \tau_2}$$

Fig. D1. SKELEFF typing.

All four are entirely straightforward and essentially traverse each sort, so that IMPEFF value types A are replaced with EXEFF value types T.

Skeleton Extraction. In Section 5, we made use of function *skeleton*(A), which computes the skeleton of a type. Its formal definition is given below:

$$
\begin{aligned}
skeleton(\alpha^\tau) &= \tau \\
skeleton(\text{Unit}) &= \text{Unit} \\
skeleton(A \to B \,!\, \Delta) &= skeleton(A) \to skeleton(B) \\
skeleton(A \,!\, \Delta_1 \Rrightarrow B \,!\, \Delta_2) &= skeleton(A) \Rrightarrow skeleton(B)
\end{aligned}
$$

A skeleton of a type captures its structure (modulo the dirt information), which is directly expressed in clauses 2, 3 and 4. Hence, in order to capture the whole skeleton of a type, the only missing piece of information is the skeleton of all type variables appearing in the type.

As we mentioned in passing in Section 5.3, each type variable is implicitly annotated with its skeleton, which allows for the complete determination of the skeleton of a type (clause 1).

Appendix D SKELEFF Additional Judgements

Typing. Typing for SKELEFF values and computations is given is Figure D1. As illustrated by the rules, SKELEFF is essentially System F extended with term-level (but not type-level) support for algebraic effects.

$$\text{value result } v^R ::= \texttt{unit} \mid h \mid \texttt{fun } (x : \tau) \mapsto c \mid \Lambda\varsigma.v$$
$$\text{computation result } c^R ::= \texttt{return } v^R \mid \texttt{Op } v^R \ (y.c)$$

$\boxed{v \rightsquigarrow_{\mathrm{v}} v'}$ **Values**

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{v \ \tau \rightsquigarrow_{\mathrm{v}} v' \ \tau} \qquad\qquad (\Lambda\varsigma.v) \ \tau \rightsquigarrow_{\mathrm{v}} v[\tau/\varsigma]$$

$\boxed{c \rightsquigarrow_{\mathrm{c}} c'}$ **Computations**

$$\frac{v_1 \rightsquigarrow_{\mathrm{v}} v_1'}{v_1 \ v_2 \rightsquigarrow_{\mathrm{c}} v_1' \ v_2} \qquad \frac{v_2 \rightsquigarrow_{\mathrm{v}} v_2'}{v_1^R \ v_2 \rightsquigarrow_{\mathrm{c}} v_1^R \ v_2'} \qquad (\texttt{fun } (x : \tau) \mapsto c) \ v^R \rightsquigarrow_{\mathrm{c}} c[v^R/x]$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{let } x = v \texttt{ in } c \rightsquigarrow_{\mathrm{c}} \texttt{let } x = v' \texttt{ in } c} \qquad \texttt{let } x = v^R \texttt{ in } c \rightsquigarrow_{\mathrm{c}} c[v^R/x] \qquad \frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{return } v \rightsquigarrow_{\mathrm{c}} \texttt{return } v'}$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{Op } v \ (y : \tau.c) \rightsquigarrow_{\mathrm{c}} \texttt{Op } v' \ (y : \tau.c)} \qquad \frac{c_1 \rightsquigarrow_{\mathrm{c}} c_1'}{\texttt{do } x \leftarrow c_1; c_2 \rightsquigarrow_{\mathrm{c}} \texttt{do } x \leftarrow c_1'; c_2}$$

$$\texttt{do } x \leftarrow \texttt{return } v^R; c_2 \rightsquigarrow_{\mathrm{c}} c_2[v^R/x] \qquad \texttt{do } x \leftarrow \texttt{Op } v^R \ (y : \tau.c_1); c_2 \rightsquigarrow_{\mathrm{c}} \texttt{Op } v^R \ (y : \tau.\texttt{do } x \leftarrow c_1; c_2)$$

$$\frac{v \rightsquigarrow_{\mathrm{v}} v'}{\texttt{handle } c \texttt{ with } v \rightsquigarrow_{\mathrm{c}} \texttt{handle } c \texttt{ with } v'} \qquad \frac{c \rightsquigarrow_{\mathrm{c}} c'}{\texttt{handle } c \texttt{ with } v^R \rightsquigarrow_{\mathrm{c}} \texttt{handle } c' \texttt{ with } v^R}$$

$$\texttt{handle } (\texttt{return } v^R) \texttt{ with } h \rightsquigarrow_{\mathrm{c}} c_r[v^R/x]$$

$$\texttt{handle } (\texttt{Op } v^R \ (y : \tau.c)) \texttt{ with } h \rightsquigarrow_{\mathrm{c}} c_{\texttt{Op}}[v^R/x, (\texttt{fun } (y : \tau) \mapsto \texttt{handle } c \texttt{ with } h)/k]$$

$$\texttt{handle } (\texttt{Op } v^R \ (y : \tau.c)) \texttt{ with } h \rightsquigarrow_{\mathrm{c}} \texttt{Op } v^R \ (y : \tau.\texttt{handle } c \texttt{ with } h)$$

Fig. D2. SKELEFF operational semantics.

Operational Semantics. Figure D2 presents the small-step, call-by-value operational semantics of SKELEFF, and Figure D3 gives the congruence closure of the step relations as used in Theorem 6.3.

Appendix E NOEFF Additional Judgements

Type Well-formedness. Well-formedness for NOEFF types is given by judgement $\Gamma \vdash_{\bar{A}} A$, which is given by the following rules:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash_{\bar{A}} \alpha} \qquad \frac{}{\Gamma \vdash_{\bar{A}} \texttt{Unit}} \qquad \frac{\Gamma \vdash_{\bar{A}} A \quad \Gamma \vdash_{\bar{A}} B}{\Gamma \vdash_{\bar{A}} A \to B} \qquad \frac{\Gamma, \alpha \vdash_{\bar{A}} A}{\Gamma \vdash_{\bar{A}} \forall \alpha.A} \qquad \frac{\Gamma \vdash_{\bar{A}} A \quad \Gamma \vdash_{\bar{A}} B}{\Gamma \vdash_{\bar{A}} A \Rightarrow B} \qquad \frac{\Gamma \vdash_{\bar{\pi}} \pi \quad \Gamma \vdash_{\bar{A}} A}{\Gamma \vdash_{\bar{A}} \pi \Rightarrow A}$$

Since NOEFF is uni-kinded, the rules simply ensure that types are well scoped.

Terms with holes

$$\text{value-holed value } V^{\text{v}} ::= [\,] \mid x \mid \text{unit} \mid H^{\text{v}} \mid \text{fun } (x:\tau) \mapsto C^{\text{v}} \mid \Lambda\varsigma.V^{\text{v}} \mid V^{\text{v}}\,\tau$$

$$\text{value-holed handler } H^{\text{v}} ::= \{\text{return } (x:\tau) \mapsto C_r^{\text{v}}, [\text{Op } x\,k \mapsto C_{\text{Op}}^{\text{v}}]_{\text{Op} \in \mathcal{O}}\}$$

$$\text{value-holed computation } C^{\text{v}} ::= V_1^{\text{v}}\,V_2^{\text{v}} \mid \text{let } x = V^{\text{v}} \text{ in } C^{\text{v}} \mid \text{return } V^{\text{v}}$$
$$\mid \ \text{Op } V^{\text{v}}\,(y:\tau.C^{\text{v}}) \mid \text{do } x \leftarrow C_1^{\text{v}}; C_2^{\text{v}}$$
$$\mid \ \text{handle } C^{\text{v}} \text{ with } V^{\text{v}}$$

$$\text{computation-holed value } V^{\text{c}} ::= x \mid \text{unit} \mid H^{\text{c}} \mid \text{fun } (x:\tau) \mapsto C^{\text{c}} \mid \Lambda\varsigma.V^{\text{c}} \mid V^{\text{c}}\,\tau$$

$$\text{computation-holed handler } H^{\text{c}} ::= \{\text{return } (x:\tau) \mapsto C_r^{\text{c}}, [\text{Op } x\,k \mapsto C_{\text{Op}}^{\text{c}}]_{\text{Op} \in \mathcal{O}}\}$$

$$\text{computation-holed computation } C^{\text{c}} ::= [\,] \mid V_1^{\text{c}}\,V_2^{\text{c}} \mid \text{let } x = V^{\text{c}} \text{ in } C^{\text{c}} \mid \text{return } V^{\text{c}}$$
$$\mid \ \text{Op } V^{\text{c}}\,(y:\tau.C^{\text{c}}) \mid \text{do } x \leftarrow C_1^{\text{c}}; C_2^{\text{c}}$$
$$\mid \ \text{handle } C^{\text{c}} \text{ with } V^{\text{c}}$$

We define values $V^{\text{v}}[v]$, $V^{\text{c}}[c]$, and computations $C^{\text{v}}[v]$, $C^{\text{c}}[c]$ in the obvious way.

$\boxed{v \equiv_{\text{v}}^{\rightsquigarrow} v'}$ **Values**

$$\frac{v \rightsquigarrow_{\text{v}} v'}{v \equiv_{\text{v}}^{\rightsquigarrow} v'} \qquad v \equiv_{\text{v}}^{\rightsquigarrow} v \qquad \frac{v \equiv_{\text{v}}^{\rightsquigarrow} v'}{v' \equiv_{\text{v}}^{\rightsquigarrow} v} \qquad \frac{v \equiv_{\text{v}}^{\rightsquigarrow} v' \qquad v' \equiv_{\text{v}}^{\rightsquigarrow} v''}{v \equiv_{\text{v}}^{\rightsquigarrow} v''}$$

$$\frac{v \equiv_{\text{v}}^{\rightsquigarrow} v'}{V^{\text{v}}[v] \equiv_{\text{v}}^{\rightsquigarrow} V^{\text{v}}[v']} \qquad\qquad \frac{c \equiv_{\text{c}}^{\rightsquigarrow} c'}{V^{\text{c}}[c] \equiv_{\text{v}}^{\rightsquigarrow} V^{\text{c}}[c']}$$

$\boxed{c \equiv_{\text{c}}^{\rightsquigarrow} c'}$ **Computations**

$$\frac{c \rightsquigarrow_{\text{c}} c'}{c \equiv_{\text{c}}^{\rightsquigarrow} c'} \qquad c \equiv_{\text{c}}^{\rightsquigarrow} c \qquad \frac{c \equiv_{\text{c}}^{\rightsquigarrow} c'}{c' \equiv_{\text{c}}^{\rightsquigarrow} c} \qquad \frac{c \equiv_{\text{c}}^{\rightsquigarrow} c' \qquad c' \equiv_{\text{c}}^{\rightsquigarrow} c''}{c \equiv_{\text{c}}^{\rightsquigarrow} c''}$$

$$\frac{v \equiv_{\text{v}}^{\rightsquigarrow} v'}{C^{\text{v}}[v] \equiv_{\text{c}}^{\rightsquigarrow} C^{\text{v}}[v']} \qquad\qquad \frac{c \equiv_{\text{c}}^{\rightsquigarrow} c'}{C^{\text{c}}[c] \equiv_{\text{c}}^{\rightsquigarrow} C^{\text{c}}[c']}$$

Fig. D3. Congruence closures of the step relations.

Constraint Well-formedness. Well-formedness for NOEFF constraints takes the form $\Gamma \vdash_{\pi} \pi$ and is given by the following rule:

$$\frac{\Gamma \vdash_A A \qquad \Gamma \vdash_A B}{\Gamma \vdash_{\pi} A \leqslant B}$$

Though very similar to the corresponding one for EXEFF, since NOEFF features no skeletons (or kinds), the above rule simply ensures that the types appearing in the constraint are both well scoped.

Operational Semantics. The complete small-step operational semantics for NOEFF are presented in Figure E1.

$\boxed{t \leadsto t'}$ **Operational Semantics**

$$\frac{t_1 \leadsto t_1'}{t_1\, t_2 \leadsto t_1'\, t_2} \qquad \frac{t \leadsto t'}{t^R\, t \leadsto t^R\, t'} \qquad (\mathtt{fun}\, x:A \mapsto t)\, t^R \leadsto t[t^R/x] \qquad \frac{t \leadsto t'}{t\, A \leadsto t'\, A}$$

$$(\Lambda\alpha.t)\, A \leadsto t[A/\alpha] \qquad \frac{t \leadsto t'}{t\, \gamma \leadsto t'\, \gamma} \qquad (\Lambda(\omega:\pi).t)\, \gamma \leadsto t[\gamma/\omega]$$

$$\frac{t_1 \leadsto t_1'}{\mathtt{let}\, x = t_1\, \mathtt{in}\, t_2 \leadsto \mathtt{let}\, x = t_1'\, \mathtt{in}\, t_2} \qquad \mathtt{let}\, x = t^R\, \mathtt{in}\, t \leadsto t[t^R/x] \qquad \frac{t \leadsto t'}{\mathtt{return}\, t \leadsto \mathtt{return}\, t'}$$

$$\frac{t_1 \leadsto t_1'}{\mathtt{Op}\, t_1\, (y:B.t_2) \leadsto \mathtt{Op}\, t_1'\, (y:B.t_2)} \qquad \frac{t_1 \leadsto t_1'}{\mathtt{do}\, x \leftarrow t_1; t_2 \leadsto \mathtt{do}\, x \leftarrow t_1'; t_2}$$

$$\mathtt{do}\, x \leftarrow \mathtt{return}\, t^R; t \leadsto t[t^R/x] \qquad \mathtt{do}\, x \leftarrow (\mathtt{Op}\, t^R\, (y:A.t_1)); t_2 \leadsto \mathtt{Op}\, t^R\, (y:A.\mathtt{do}\, x \leftarrow t_1; t_2)$$

$$\frac{t_h \leadsto t_h'}{\mathtt{handle}\, t_c\, \mathtt{with}\, t_h \leadsto \mathtt{handle}\, t_c\, \mathtt{with}\, t_h'} \qquad \frac{t_c \leadsto t_c'}{\mathtt{handle}\, t_c\, \mathtt{with}\, t_h^R \leadsto \mathtt{handle}\, t_c'\, \mathtt{with}\, t_h^R}$$

$$\mathtt{handle}\, (\mathtt{return}\, t^R)\, \mathtt{with}\, h \leadsto t_r[t^R/x]$$

$$\frac{(\mathtt{Op}\, x\, k \mapsto t_{\mathtt{Op}}) \in h}{\mathtt{handle}\, (\mathtt{Op}\, t^R\, (y:B.t))\, \mathtt{with}\, h \leadsto t_{\mathtt{Op}}[t^R/x, (\mathtt{fun}\, (y:B) \mapsto \mathtt{handle}\, t\, \mathtt{with}\, h)/k]}$$

$$\frac{(\mathtt{Op}\, x\, k \mapsto t_{\mathtt{Op}}) \notin h}{\mathtt{handle}\, (\mathtt{Op}\, t^R\, (y:B.t))\, \mathtt{with}\, h \leadsto \mathtt{Op}\, t^R\, (y:B.\mathtt{handle}\, t\, \mathtt{with}\, h)} \qquad \frac{t \leadsto t'}{t \triangleright \gamma \leadsto t' \triangleright \gamma}$$

$$t^R \triangleright \langle \mathtt{Unit} \rangle \leadsto t^R \qquad (\mathtt{return}\, t^R) \triangleright (\mathtt{comp}\, \gamma) \leadsto \mathtt{return}\, (t^R \triangleright \gamma)$$

$$(\mathtt{Op}\, t^R\, (y:B.t)) \triangleright (\mathtt{comp}\, \gamma) \leadsto \mathtt{Op}\, t^R\, (y:B.(t \triangleright (\mathtt{comp}\, \gamma)))$$

$$t^R \triangleright \mathtt{return}\, \gamma \leadsto \mathtt{return}\, (t^R \triangleright \gamma) \qquad (\mathtt{return}\, t^R) \triangleright (\mathtt{unsafe}\, \gamma) \leadsto t^R \triangleright \gamma$$

$$(t^R \triangleright (\gamma_1 \rightarrow \gamma_2))\, t \leadsto (t^R\, (t \triangleright \gamma_1)) \triangleright \gamma_2$$

$$\mathtt{handle}\, t_1^R\, \mathtt{with}\, (t_2^R \triangleright (\gamma_1 \Rightarrow \gamma_2)) \leadsto (\mathtt{handle}\, (t_1^R \triangleright \gamma_1)\, \mathtt{with}\, t_2^R) \triangleright \gamma_2$$

$$(t_1^R \triangleright (\mathtt{handToFun}\, \gamma_1\, \gamma_2))\, t_2^R \leadsto (\mathtt{handle}\, (\mathtt{return}\, (t_2^R \triangleright \gamma_1))\, \mathtt{with}\, t_1^R) \triangleright \gamma_2$$

$$\mathtt{handle}\, (\mathtt{Op}\, t_1^R\, (y:B.t))\, \mathtt{with}\, (t_2^R \triangleright (\mathtt{funToHand}\, \gamma_1\, \gamma_2))$$
$$\leadsto \mathtt{Op}\, t_1^R\, (y:B.\mathtt{handle}\, t\, \mathtt{with}\, (t_2^R \triangleright (\mathtt{funToHand}\, \gamma_1\, \gamma_2)))$$

$$\mathtt{handle}\, (\mathtt{return}\, t_1^R)\, \mathtt{with}\, (t_2^R \triangleright (\mathtt{funToHand}\, \gamma_1\, \gamma_2)) \leadsto (t_2^R\, (t_1^R \triangleright \gamma_1)) \triangleright \gamma_2$$

$$(t^R \triangleright \forall\alpha.\gamma)\, A \leadsto (t^R\, A) \triangleright \gamma[A/\alpha] \qquad (t^R \triangleright (\pi \Rightarrow \gamma_1))\, \gamma_2 \leadsto (t^R\, \gamma_2) \triangleright \gamma_1$$

Fig. E1. NOEFF Operational semantics.
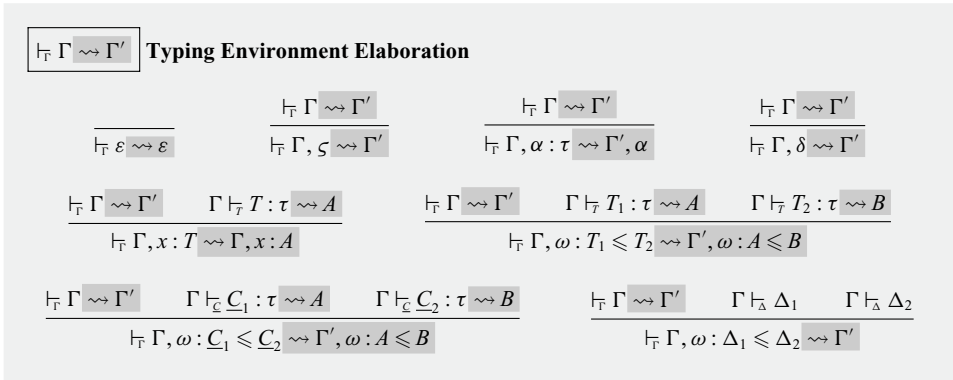
Fig. F1. Elaboration of EXEFF typing environments to NOEFF typing environments.

Appendix F EXEFF to NOEFF: Additional Judgements

Typing Environment Elaboration. Elaboration of typing environments is given in Figure F1. Essentially, the judgement removes all dirt and skeleton information is removed (including dirt inequalities).

Elaboration of EXEFF coercions to NOEFF coercions is given in Figure F2.

Figure F3 shows the elaboration of EXEFF values into NOEFF terms.

$\boxed{\Gamma \vdash_{\overline{co}} \gamma : \pi \rightsquigarrow \gamma'}$ **Coercion Elaboration**

$$\frac{(\omega : \pi) \in \Gamma}{\Gamma \vdash_{\overline{co}} \omega : \pi \rightsquigarrow \omega} \qquad \frac{}{\Gamma \vdash_{\overline{co}} \langle \texttt{Unit} \rangle : \texttt{Unit} \leqslant \texttt{Unit} \rightsquigarrow \langle \texttt{Unit} \rangle} \qquad \frac{(\alpha : \tau) \in \Gamma}{\Gamma \vdash_{\overline{co}} \langle \alpha \rangle : \alpha \leqslant \alpha \rightsquigarrow \langle \alpha \rangle}$$

$$\frac{\Gamma \vdash_{\overline{co}} \gamma_1 : T_2 \leqslant T_1 \rightsquigarrow \gamma_1' \qquad \Gamma \vdash_{\overline{co}} \gamma_2 : \underline{C}_1 \leqslant \underline{C}_2 \rightsquigarrow \gamma_2'}{\Gamma \vdash_{\overline{co}} \gamma_1 \rightarrow \gamma_2 : (T_1 \rightarrow \underline{C}_1) \leqslant (T_2 \rightarrow \underline{C}_2) \rightsquigarrow \gamma_1' \rightarrow \gamma_2'}$$

$$\frac{\Gamma \vdash_{\overline{co}} \gamma_1 : T_2 \,!\, \emptyset \leqslant T_1 \,!\, \emptyset \rightsquigarrow \gamma_1' \qquad \Gamma \vdash_{\overline{co}} \gamma_2 : \underline{C}_1 \leqslant \underline{C}_2 \rightsquigarrow \gamma_2'}{\Gamma \vdash_{\overline{co}} \gamma_1 \Rightarrow \gamma_2 : (T_1 \,!\, \emptyset \Rightarrow \underline{C}_1) \leqslant (T_2 \,!\, \emptyset \Rightarrow \underline{C}_2) \rightsquigarrow \gamma_1' \rightarrow \gamma_2'}$$

$$\frac{\begin{array}{ccc} nonEmpty(\Delta_1) & nonEmpty(\Delta_2) \\ \Gamma \vdash_{\overline{co}} \gamma_1 : (T_2 \,!\, \Delta_2 \leqslant T_1 \,!\, \Delta_1) \rightsquigarrow \gamma_1' & \Gamma \vdash_{\overline{co}} \gamma_2 : T_1' \leqslant T_2' \rightsquigarrow \gamma_2' & \Gamma \vdash_{\overline{co}} \gamma_3 : \Delta_1' \leqslant \Delta_2' \end{array}}{\Gamma \vdash_{\overline{co}} (\gamma_1 \Rightarrow (\gamma_2 \,!\, \gamma_3)) : ((T_1 \,!\, \Delta_1) \Rightarrow (T_1' \,!\, \Delta_1')) \leqslant ((T_2 \,!\, \Delta_2) \Rightarrow (T_2' \,!\, \Delta_2')) \rightsquigarrow \gamma_1' \Rightarrow \texttt{comp}\, \gamma_2'}$$

$$\frac{\begin{array}{cccc} & nonEmpty(\Delta_1) & & \\ \Gamma \vdash_{\overline{co}} \gamma_1 : T_2 \leqslant T_1 \rightsquigarrow \gamma_1' & \Gamma \vdash_{\overline{co}} \gamma_2 : T_1' \leqslant T_2' \rightsquigarrow \gamma_2' & \Gamma \vdash_{\overline{co}} \gamma_3 : \emptyset \leqslant \Delta_1 & \Gamma \vdash_{\overline{co}} \gamma_4 : \emptyset \leqslant \Delta_2' \end{array}}{\Gamma \vdash_{\overline{co}} (\gamma_1 \,!\, \gamma_3 \Rightarrow \gamma_2 \,!\, \gamma_4) : ((T_1 \,!\, \Delta_1 \Rightarrow T_1' \,!\, \emptyset) \leqslant (T_2 \,!\, \emptyset \Rightarrow T_2' \,!\, \Delta_2')) \rightsquigarrow \texttt{handToFun}\, \gamma_1' \, (\texttt{unsafe}\, \gamma_2')}$$

$$\frac{\begin{array}{cccc} & nonEmpty(\Delta_1) & nonEmpty(\Delta_1') & \\ \Gamma \vdash_{\overline{co}} \gamma_1 : T_2 \leqslant T_1 \rightsquigarrow \gamma_1' & \Gamma \vdash_{\overline{co}} \gamma_2 : (T_1' \leqslant T_2') \rightsquigarrow \gamma_2' & \Gamma \vdash_{\overline{co}} \gamma_3 : \emptyset \leqslant \Delta_1 & \Gamma \vdash_{\overline{co}} \gamma_4 : \Delta_1' \leqslant \Delta_2' \end{array}}{\Gamma \vdash_{\overline{co}} (\gamma_1 \,!\, \gamma_3 \Rightarrow \gamma_2 \,!\, \gamma_4) : ((T_1 \,!\, \Delta_1 \Rightarrow T_1' \,!\, \Delta_1') \leqslant (T_2 \,!\, \emptyset \Rightarrow T_2' \,!\, \Delta_2')) \rightsquigarrow \texttt{handToFun}\, \gamma_1' \, \gamma_2'}$$

$$\frac{\Gamma, \varsigma \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma'}{\Gamma \vdash_{\overline{co}} \forall \varsigma.\gamma : \forall \varsigma.T_1 \leqslant \forall \varsigma.T_2 \rightsquigarrow \gamma'} \qquad \frac{\Gamma \vdash_{\overline{\tau}} \tau \qquad \Gamma, \alpha : \tau \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma'}{\Gamma \vdash_{\overline{co}} \forall(\alpha : \tau).\gamma : \forall(\alpha : \tau).T_1 \leqslant \forall(\alpha : \tau).T_2 \rightsquigarrow \forall\alpha.\gamma'}$$

$$\frac{\Gamma, \delta \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma'}{\Gamma \vdash_{\overline{co}} \forall \delta.\gamma : \forall \delta.T_1 \leqslant \forall \delta.T_2 \rightsquigarrow \gamma'}$$

$$\frac{\Gamma \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma' \qquad \Gamma \vdash_{\overline{\tau}} T_3 : \tau \rightsquigarrow A_1 \qquad \Gamma \vdash_{\overline{\tau}} T_4 : \tau \rightsquigarrow A_2}{\Gamma \vdash_{\overline{co}} (T_3 \leqslant T_4) \Rightarrow \gamma : ((T_3 \leqslant T_4) \Rightarrow T_1) \leqslant ((T_3 \leqslant T_4) \Rightarrow T_2) \rightsquigarrow (A_1 \leqslant A_2) \Rightarrow \gamma'}$$

$$\frac{\Gamma \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma' \qquad \Gamma \vdash_{\overline{\underline{c}}} \underline{C}_1 : \tau \rightsquigarrow B_1 \qquad \Gamma \vdash_{\overline{\underline{c}}} \underline{C}_2 : \tau \rightsquigarrow B_2}{\Gamma \vdash_{\overline{co}} (\underline{C}_1 \leqslant \underline{C}_2) \Rightarrow \gamma : ((\underline{C}_1 \leqslant \underline{C}_2) \Rightarrow T_1) \leqslant ((\underline{C}_1 \leqslant \underline{C}_2) \Rightarrow T_2) \rightsquigarrow (B_1 \leqslant B_2) \Rightarrow \gamma'}$$

$$\frac{\Gamma \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma'}{\Gamma \vdash_{\overline{co}} (\Delta_1 \leqslant \Delta_2) \Rightarrow \gamma : (\Delta_1 \leqslant \Delta_2) \Rightarrow T_1 \leqslant (\Delta_1 \leqslant \Delta_2) \Rightarrow T_2 \rightsquigarrow \gamma'}$$

$$\frac{\Gamma \vdash_{\overline{co}} \gamma_1 : T_1 \leqslant T_2 \rightsquigarrow \gamma_1' \qquad \Gamma \vdash_{\overline{co}} \gamma_2 : \emptyset \leqslant \emptyset}{\Gamma \vdash_{\overline{co}} (\gamma_1 \,!\, \gamma_2) : (T_1 \,!\, \emptyset \leqslant T_2 \,!\, \emptyset) \rightsquigarrow \gamma_1'} \qquad \frac{\begin{array}{cc} \Gamma \vdash_{\overline{co}} \gamma_1 : T_1 \leqslant T_2 \rightsquigarrow \gamma_1' \\ \Gamma \vdash_{\overline{co}} \gamma_2 : \emptyset \leqslant \Delta_2 \quad nonEmpty(\Delta_2) \end{array}}{\Gamma \vdash_{\overline{co}} (\gamma_1 \,!\, \gamma_2) : (T_1 \,!\, \emptyset \leqslant T_2 \,!\, \Delta_2) \rightsquigarrow \texttt{return}\, \gamma_1'}$$

$$\frac{\Gamma \vdash_{\overline{co}} \gamma_1 : T_1 \leqslant T_2 \rightsquigarrow \gamma_1' \qquad \Gamma \vdash_{\overline{co}} \gamma_2 : \Delta_1 \leqslant \Delta_2 \qquad nonEmpty(\Delta_1) \qquad nonEmpty(\Delta_2)}{\Gamma \vdash_{\overline{co}} (\gamma_1 \,!\, \gamma_2) : (T_1 \,!\, \Delta_1 \leqslant T_2 \,!\, \Delta_2) \rightsquigarrow \texttt{comp}\, \gamma_1'}$$

Fig. F2. Elaboration of EXEFF coercions to NOEFF coercions.

$\boxed{\Gamma \vdash_{\overline{v}} v : T \rightsquigarrow t}$ **Value Elaboration**

$$\Gamma \vdash_{\overline{v}} \mathtt{unit} : \mathtt{Unit} \rightsquigarrow \mathtt{unit}$$

$$\frac{\Gamma \vdash_T T : \tau \rightsquigarrow A \qquad \Gamma, x:T \vdash_{\overline{c}} c : \underline{C} \rightsquigarrow t}{\Gamma \vdash_{\overline{v}} \mathtt{fun}\,(x:T) \mapsto c : T \to \underline{C} \rightsquigarrow \mathtt{fun}\,(x:A) \mapsto t}$$

$$\frac{\Gamma \vdash_T T : \tau \rightsquigarrow A \qquad \Gamma, x:T \vdash_{\overline{c}} c_r : \underline{C} \rightsquigarrow t}{\Gamma \vdash_{\overline{v}} \{\mathtt{return}\,(x:T) \mapsto c_r\} : T\,!\,\emptyset \Rrightarrow \underline{C} \rightsquigarrow \mathtt{fun}\,(x:A) \mapsto t}$$

$$\frac{\begin{array}{c} nonEmpty(\mathcal{O}) \qquad \Gamma \vdash_T T_x : \tau \rightsquigarrow A \qquad \Gamma, x:T_x \vdash_{\overline{c}} c_r : T\,!\,\emptyset \rightsquigarrow t_r \\ \left[(\mathtt{Op}:T_1^{\mathtt{Op}} \to T_2^{\mathtt{Op}}) \in \Sigma \quad \vdash_T T_i^{\mathtt{Op}} : \tau_i^{\mathtt{Op}} \rightsquigarrow A_i^{\mathtt{Op}} \quad \Gamma, x:T_1^{\mathtt{Op}}, k:T_2^{\mathtt{Op}} \to T\,!\,\emptyset \vdash_{\overline{c}} c_{\mathtt{Op}} : T\,!\,\emptyset \rightsquigarrow t_{\mathtt{Op}}\right]_{\mathtt{Op}\in\mathcal{O}} \end{array}}{\begin{array}{c} \Gamma \vdash_{\overline{v}} \{\mathtt{return}\,(x:T_x) \mapsto c_r, [\mathtt{Op}\,x\,k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op}\in\mathcal{O}}\} : T_x\,!\,\mathcal{O} \Rrightarrow T\,!\,\emptyset \\ \rightsquigarrow \{\mathtt{return}\,(x:A) \mapsto \mathtt{return}\,t_r, \left[\mathtt{Op}\,x\,k \mapsto \mathtt{return}\,t_{\mathtt{Op}}[k \triangleright \langle A_1^{\mathtt{Op}}\rangle \to \mathtt{unsafe}\,\langle A_2^{\mathtt{Op}}\rangle/k]\right]_{\mathtt{Op}\in\mathcal{O}}\} \end{array}}$$

$$\frac{\begin{array}{c} nonEmpty(\mathcal{O}) \qquad nonEmpty(\Delta) \qquad \Gamma \vdash_T T_x : \tau \rightsquigarrow A \qquad \Gamma, x:T_x \vdash_{\overline{c}} c_r : T\,!\,\Delta \rightsquigarrow t_r \\ \left[(\mathtt{Op}:T_1^{\mathtt{Op}} \to T_2^{\mathtt{Op}}) \in \Sigma \qquad \Gamma, x:T_1^{\mathtt{Op}}, k:T_2^{\mathtt{Op}} \to T\,!\,\Delta \vdash_{\overline{c}} c_{\mathtt{Op}} : T\,!\,\Delta \rightsquigarrow t_{\mathtt{Op}}\right]_{\mathtt{Op}\in\mathcal{O}} \end{array}}{\Gamma \vdash_{\overline{v}} \{\mathtt{return}\,(x:T_x) \mapsto c_r, [\mathtt{Op}\,x\,k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op}\in\mathcal{O}}\} : T_x\,!\,\mathcal{O} \Rrightarrow T\,!\,\Delta \rightsquigarrow \{\mathtt{return}\,(x:A) \mapsto t_r, [\mathtt{Op}\,x\,k \mapsto t_{\mathtt{Op}}]_{\mathtt{Op}\in\mathcal{O}}\}}$$

$$\frac{\Gamma, \varsigma \vdash_{\overline{v}} v : T \rightsquigarrow t}{\Gamma \vdash_{\overline{v}} \Lambda\varsigma.v : \forall\varsigma.T \rightsquigarrow t} \qquad \frac{\Gamma \vdash_{\overline{v}} v : \forall\varsigma.T \rightsquigarrow t}{\Gamma \vdash_{\overline{v}} v\,\tau : T[\tau/\varsigma] \rightsquigarrow t} \qquad \frac{\Gamma, \alpha:\tau \vdash_{\overline{v}} v : T \rightsquigarrow t}{\Gamma \vdash_{\overline{v}} \Lambda(\alpha:\tau).v : \forall(\alpha:\tau).T \rightsquigarrow \Lambda\alpha.t}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\overline{v}} v : \forall(\alpha:\tau).T \rightsquigarrow t \\ \Gamma \vdash_T T_1 : \tau \rightsquigarrow A \end{array}}{\Gamma \vdash_{\overline{v}} v\,T_1 : T[T_1/\alpha] \rightsquigarrow t\,A} \qquad \frac{\Gamma, \delta \vdash_{\overline{v}} v : T \rightsquigarrow t}{\Gamma \vdash_{\overline{v}} \Lambda\delta.v : \forall\delta.T \rightsquigarrow t} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\overline{v}} v : \forall\delta.T \rightsquigarrow t \\ \delta \mapsto \Delta; \Gamma \vdash_{\overline{v}} T \rightsquigarrow \gamma \end{array}}{\Gamma \vdash_{\overline{v}} v\,\Delta : T[\Delta/\delta] \rightsquigarrow t \triangleright \gamma}$$

$$\frac{\Gamma, \omega:T_1 \leqslant T_2 \vdash_{\overline{v}} v : T \rightsquigarrow t \qquad \Gamma \vdash_T T_1 : \tau \rightsquigarrow A \qquad \Gamma \vdash_T T_2 : \tau \rightsquigarrow B}{\Gamma \vdash_{\overline{v}} \Lambda(\omega:T_1 \leqslant T_2).v : (T_1 \leqslant T_2 \Rightarrow T) \rightsquigarrow \Lambda(\omega:A \leqslant B).t}$$

$$\frac{\Gamma, \omega:\underline{C}_1 \leqslant \underline{C}_2 \vdash_{\overline{v}} v : T \rightsquigarrow t \qquad \Gamma \vdash_{\underline{C}} \underline{C}_1 : \tau \rightsquigarrow A \qquad \Gamma \vdash_{\underline{C}} \underline{C}_2 : \tau \rightsquigarrow B}{\Gamma \vdash_{\overline{v}} \Lambda(\omega:\underline{C}_1 \leqslant \underline{C}_2).v : (\underline{C}_1 \leqslant \underline{C}_2 \Rightarrow T) \rightsquigarrow \Lambda(\omega:A \leqslant B).t}$$

$$\frac{\Gamma, \omega:\Delta_1 \leqslant \Delta_2 \vdash_{\overline{v}} v : T \rightsquigarrow t}{\Gamma \vdash_{\overline{v}} \Lambda(\omega:\Delta_1 \leqslant \Delta_2).v : (\Delta_1 \leqslant \Delta_2 \Rightarrow T) \rightsquigarrow t} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\overline{v}} v : (T_1 \leqslant T_2) \Rightarrow T \rightsquigarrow t \\ \Gamma \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma' \end{array}}{\Gamma \vdash_{\overline{v}} v\,\gamma : T \rightsquigarrow t\,\gamma'}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\overline{v}} v : (\underline{C}_1 \leqslant \underline{C}_2) \Rightarrow T \rightsquigarrow t \\ \Gamma \vdash_{\overline{co}} \gamma : \underline{C}_1 \leqslant \underline{C}_2 \rightsquigarrow \gamma' \end{array}}{\Gamma \vdash_{\overline{v}} v\,\gamma : T \rightsquigarrow t\,\gamma'} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\overline{v}} v : (\Delta_1 \leqslant \Delta_2) \Rightarrow T \rightsquigarrow t \\ \Gamma \vdash_{\overline{co}} \gamma : \Delta_1 \leqslant \Delta_2 \end{array}}{\Gamma \vdash_{\overline{v}} v\,\gamma : T \rightsquigarrow t} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\overline{v}} v : T_1 \rightsquigarrow t \\ \Gamma \vdash_{\overline{co}} \gamma : T_1 \leqslant T_2 \rightsquigarrow \gamma' \end{array}}{\Gamma \vdash_{\overline{v}} v \triangleright \gamma : T_2 \rightsquigarrow t \triangleright \gamma'}$$

Fig. F3. Elaboration of ExEff values to NoEff terms.