Explicit Effect Subtyping

Amr Hany Saleh[1], Georgios Karachalias[1], Matija Pretnar[2], and Tom Schrijvers[1]

[1] KU Leuven, Department of Computer Science, Belgium,
[2] University of Ljubljana, Faculty of Mathematics and Physics, Slovenia

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 optimizing compiler could be built. Unfortunately, in our experience, implementing optimizations 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.

1 Introduction

Algebraic effect handlers [17, 18] 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 [22] to narrow this gap with an optimising compiler from Eff [2] 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 this fragility is the complexity of subtyping in combination with the implicit typing of Eff's core language, further aggravated by the "garbage collection" of subtyping constraints (see Section 7).[3]

For efficient compilation, one must avoid the poisoning problem [26], 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 [12, 8, 14], 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.

[3] For other issues stemming from the same combination see issues #11 and #16 at https://github.com/matijapretnar/eff/issues/.

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. [3]. 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 by-product. It also establishes that the elaboration preserves typing, and that the algorithm is sound with respect to the specification and yields principal types.
- Finally, 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 [6], 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 and that, despite the existence of multiple proofs for the same subtyping, there is no coherence problem. To enable erasure, ExEff annotates its types with *(type) skeletons*, which capture the erased counterpart and are, to our knowledge, a novel contribution.
- Our paper comes with two software artefacts: an ongoing implementation[4] of a compiler from Eff to OCaml with ExEff at its core, and an Abella mechanisation[5] of Theorems 1, 2, 6, and 7. Remaining theorems all concern the inference algorithm, and their proofs closely follow [20].

The full version of this paper includes an appendix with omitted figures and can be found at http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW711.abs.html.

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 [17]. This allows generalizing exception handlers to other effects, to express backtracking, co-operative multithreading and other examples in a natural way [18, 2].

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

[4] https://github.com/matijapretnar/eff/tree/explicit-effect-subtyping

[5] https://github.com/matijapretnar/proofs/tree/master/explicit-effect-subtyping

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.

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, $\mathtt{Tick}\ \mathtt{unit}$ has type $\mathtt{Unit}\ !\ \{\mathtt{Tick}\}$ and $f\ x$ type $\mathtt{Unit}\ !\ \{\mathtt{Tock}\}$. Yet, because they are sequenced with do, the type system expects they have the same set of effects. The discrepancies are implicitly reconciled by the subtyping which admits both $\{\mathtt{Tick}\} \leqslant \{\mathtt{Tick}, \mathtt{Tock}\}$ and $\{\mathtt{Tock}\} \leqslant \{\mathtt{Tick}, \mathtt{Tock}\}$.

We elaborate the ImpEff term into the explicitly-typed core language ExEff to make those appeals to subtyping explicit by means of casts with coercions:

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

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

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 $\mathtt{let}\ f = (\mathtt{fun}\ g \mapsto g\ \mathtt{unit})\ \mathtt{in}\ \ldots$ is generalised to:

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

This polymorphic type scheme follows the qualified types convention [9] where the type $(\mathtt{Unit} \to \alpha\ !\ \delta) \to \alpha'\ !\ \delta'$ 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.

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

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 $\forall\alpha, \alpha'.(\texttt{Unit} \to \alpha) \to \alpha'$ instead of the expected type $\forall\alpha.(\texttt{Unit} \to \alpha) \to \alpha$. 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 meta-theory.

Fortunately, there is an easier way by tagging all bound type variables with *skeletons*, which are barebone 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:

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

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

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

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:

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

Terms

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

Types & Constraints

$$\text{skeleton } \tau ::= \varsigma \mid \texttt{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2$$

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

$$\begin{aligned}
\text{simple constraint } \pi &::= A_1 \leqslant A_2 \mid \mathit{\Delta}_1 \leqslant \mathit{\Delta}_2 \\
\text{constraint } \rho &::= \pi \mid \underline{C} \leqslant \underline{D}
\end{aligned}$$

Fig. 1: ImpEff Syntax

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. We describe the relevant concepts, but refer the reader to Pretnar's tutorial [21], which explains essentially the same calculus in more detail.

3.1 Syntax

Figure 1 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 $\texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n}$ as $[\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}$, and write $\mathcal{O}$ to denote the set $\{\texttt{Op}_1, \ldots, \texttt{Op}_n\}$.

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 $A \to \underline{C}$, handler types $\underline{C} \Rrightarrow \underline{D}$ and the Unit type. Skeletons τ capture the shape of types, so, by design, their forms are identical. The computation type $A \,!\, \mathit{\Delta}$ is assigned to a computation returning values of type A and potentially calling operations from the *dirt* set $\mathit{\Delta}$. 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 $\mathit{\Delta}$ is that the order of operations Op is irrelevant. Similarly to all HM-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 2 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 gray and are discussed in Section 5.

Values Typing for values takes the form $\Gamma \vdash_v v : A \rightsquigarrow v'$, 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 $\Gamma \vdash_{co} \overline{\gamma : \sigma(\pi)}$. 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 [1]. 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.[6] The result type takes the form $A \,!\, \Delta \cup \mathcal{O} \Rrightarrow B \,!\, \Delta$, 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 $\Gamma \vdash_c c : \underline{C} \rightsquigarrow c'$, and, given a typing environment Γ, checks a computation c against a type $\underline{C}$.

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 $\mathtt{Op} \in \Delta$ 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 $\Gamma \vdash_{co} \gamma : \rho$ and is presented in Figure 3. Notice that we use ρ instead of π, which

[6] We capture all defined operations along with their types in a global signature Σ.

typing environment $\mathit{\Gamma} ::= \epsilon \mid \mathit{\Gamma}, \varsigma \mid \mathit{\Gamma}, \alpha : \tau \mid \mathit{\Gamma}, \delta \mid \mathit{\Gamma}, x : S \mid \mathit{\Gamma}, \omega : \pi$

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

$$\frac{(x : \forall \bar{\varsigma}. \forall \overline{\alpha : \tau}. \forall \bar{\delta}. \bar{\pi} \Rightarrow A) \in \mathit{\Gamma} \qquad \sigma = [\overline{\tau' / \varsigma}, \overline{B / \alpha}, \overline{\mathit{\Delta} / \delta}] \qquad \overline{\mathit{\Gamma} \vdash_{\mathsf{co}} \gamma : \sigma(\pi)}}{\mathit{\Gamma} \vdash_v x : \sigma(A) \rightsquigarrow x \ \ \bar{\tau}' \ \bar{B} \ \bar{\mathit{\Delta}} \ \bar{\gamma}} \ \textsc{TmVar}$$

$$\frac{\begin{array}{c}\mathit{\Gamma} \vdash_v v : A \rightsquigarrow v' \\ \mathit{\Gamma} \vdash_{\mathsf{co}} \gamma : A \leqslant B\end{array}}{\mathit{\Gamma} \vdash_v v : B \rightsquigarrow v' \rhd \gamma} \ \textsc{TmCastV} \qquad \frac{}{\mathit{\Gamma} \vdash_v \mathtt{unit} : \mathtt{Unit} \rightsquigarrow \mathtt{unit}} \ \textsc{TmUnit}$$

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

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

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

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

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

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

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

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

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

Fig. 2: ImpEff Typing & Elaboration

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

$$\frac{(\omega : \pi) \in \Gamma}{\Gamma \vdash_{\mathsf{co}} \omega : \pi}\ \textsc{CoVar} \qquad \frac{\Gamma \vdash_{\mathsf{vty}} A : \tau \rightsquigarrow T}{\Gamma \vdash_{\mathsf{co}} \langle T \rangle : A \leqslant A}\ \textsc{VCoRefl}$$

$$\frac{\Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\mathsf{co}} \langle \Delta \rangle : \Delta \leqslant \Delta}\ \textsc{DCoRefl} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_1 \leqslant A_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : A_2 \leqslant A_3\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : A_1 \leqslant A_3}\ \textsc{VCoTrans}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \underline{C}_1 \leqslant \underline{C}_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C}_2 \leqslant \underline{C}_3\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : \underline{C}_1 \leqslant \underline{C}_3}\ \textsc{CCoTrans} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \Delta_1 \leqslant \Delta_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \Delta_2 \leqslant \Delta_3\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : \Delta_1 \leqslant \Delta_3}\ \textsc{DCoTrans}$$

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

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : A \rightarrow \underline{C} \leqslant B \rightarrow \underline{D}}{\Gamma \vdash_{\mathsf{co}} \mathit{left}(\gamma) : B \leqslant A}\ \textsc{VCoArrL} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : A \rightarrow \underline{C} \leqslant B \rightarrow \underline{D}}{\Gamma \vdash_{\mathsf{co}} \mathit{right}(\gamma) : \underline{C} \leqslant \underline{D}}\ \textsc{CCoArrR}$$

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

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}{\Gamma \vdash_{\mathsf{co}} \mathit{left}(\gamma) : \underline{C}_2 \leqslant \underline{C}_1}\ \textsc{CCoHL} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}{\Gamma \vdash_{\mathsf{co}} \mathit{right}(\gamma) : \underline{D}_1 \leqslant \underline{D}_2}\ \textsc{CCoHR}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_1 \leqslant A_2 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \Delta_1 \leqslant \Delta_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \ !\ \gamma_2 : A_1 \ !\ \Delta_1 \leqslant A_2 \ !\ \Delta_2}\ \textsc{CCoComp}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : A_1 \ !\ \Delta_1 \leqslant A_2 \ !\ \Delta_2}{\Gamma \vdash_{\mathsf{co}} \mathit{pure}(\gamma) : A_1 \leqslant A_2}\ \textsc{VCoPure} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : A_1 \ !\ \Delta_1 \leqslant A_2 \ !\ \Delta_2}{\Gamma \vdash_{\mathsf{co}} \mathit{impure}(\gamma) : \Delta_1 \leqslant \Delta_2}\ \textsc{DCoImpure}$$

$$\frac{}{\Gamma \vdash_{\mathsf{co}} \emptyset_{\Delta} : \emptyset \leqslant \Delta}\ \textsc{DCoNil} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : \Delta_1 \leqslant \Delta_2 \qquad (\mathtt{Op} : A_{\mathtt{Op}} \rightarrow B_{\mathtt{Op}}) \in \Sigma}{\Gamma \vdash_{\mathsf{co}} \{\mathtt{Op}\} \cup \gamma : \{\mathtt{Op}\} \cup \Delta_1 \leqslant \{\mathtt{Op}\} \cup \Delta_2}\ \textsc{DCoOp}$$

Fig. 3: ImpEff Constraint Entailment

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 given assumptions. Rules VCoRefl and DCoRefl express that subtyping is reflexive, for both value types and dirts. Notice that we do not have a rule for the reflexivity of computation types since, as we illustrate below, it can be established using the reflexivity of their subparts. Rules VCo-

Trans, CCoTrans and DCoTrans express the transitivity of subtyping for value types, computation types and dirts, respectively. Rule VCoArr establishes inequality of arrow types. As usual, the arrow type constructor is contravariant in the argument type. Rules VCoArrL and CCoArrR are the inversions of Rule VCoArr, allowing us to establish the relation between the subparts of the arrow types. Rules VCoHand, CCoHL, and CCoHR work similarly, 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. Rules VCoPure and DCoImpure are its inversions. 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 Op.

Well-formedness of Types, Constraints, Dirts, and Skeletons The relations $\Gamma \vdash_{\mathsf{vty}} A : \tau \rightsquigarrow T$ and $\Gamma \vdash_{\mathsf{cty}} \underline{C} : \tau \rightsquigarrow \underline{C}$ check the well-formedness of value and computation types respectively. Similarly, relations $\Gamma \vdash_{\mathsf{ct}} \rho \rightsquigarrow \rho$ and $\Gamma \vdash_{\Delta} \Delta$ check the well-formedness of constraints and dirts, respectively.

4 The ExEff Language

4.1 Syntax

Figure 4 presents ExEff's syntax. ExEff is an intensional type theory akin to System F [7], 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 is impredicative, 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 the impredicativity is not strictly required for the purpose at hand, it makes for a cleaner system.

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 2) with explicit casts $(v \rhd \gamma)$ and $(c \rhd \gamma)$.

Essentially, coercions γ are explicit witnesses of subtyping derivations: each coercion form corresponds to a subtyping rule. Subtyping forms a partial order, which is reflected in coercion forms $\gamma_1 \gg \gamma_2$, $\langle T \rangle$, and $\langle \Delta \rangle$. Coercion form $\gamma_1 \gg \gamma_2$ captures transitivity, while forms $\langle T \rangle$ and $\langle \Delta \rangle$ capture reflexivity for value types and dirts (reflexivity for computation types can be derived from these).

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, forms $\gamma[\tau]$, $\gamma[T]$, $\gamma[\Delta]$, and $\gamma_1 @ \gamma_2$ witness subtyping of skeleton instantiation, type instantiation, dirt instantiation, and coercion application, respectively.

Syntactic forms $\gamma_1 \rightarrow \gamma_2$ and $\gamma_1 \Rrightarrow \gamma_2$ capture injection for the arrow and the handler type constructor, respectively. Similarly, inversion forms *left*(γ) and *right*(γ) capture projection, following from the injectivity of both type constructors.

Terms

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

Types

$$\begin{array}{rcl}
\text{skeleton } \tau & ::= & \varsigma \mid \texttt{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \mid \forall\varsigma.\tau \\
\\
\text{value type } T & ::= & \alpha \mid \texttt{Unit} \mid T \to \underline{C} \mid \underline{C}_1 \Rrightarrow \underline{C}_2 \mid \forall\varsigma.T \mid \forall\alpha{:}\tau.T \mid \forall\delta.T \mid \pi \Rightarrow T \\
\text{simple coercion type } \pi & ::= & T_1 \leqslant T_2 \mid \varDelta_1 \leqslant \varDelta_2 \\
\text{coercion type } \rho & ::= & \pi \mid \underline{C}_1 \leqslant \underline{C}_2 \\
\\
\text{computation type } \underline{C} & ::= & T\ !\ \varDelta \\
\text{dirt } \varDelta & ::= & \delta \mid \emptyset \mid \{\texttt{Op}\} \cup \varDelta
\end{array}$$

Coercions

$$\begin{array}{rl}
\gamma ::= & \omega \mid \gamma_1 \gg \gamma_2 \mid \langle T \rangle \mid \gamma_1 \to \gamma_2 \mid \gamma_1 \Rrightarrow \gamma_2 \mid \mathit{left}(\gamma) \mid \mathit{right}(\gamma) \mid \langle \varDelta \rangle \mid \emptyset_{\varDelta} \mid \{\texttt{Op}\} \cup \gamma \\
\mid & \forall\varsigma.\gamma \mid \gamma[\tau] \mid \forall\alpha.\gamma \mid \gamma[T] \mid \forall\delta.\gamma \mid \gamma[\varDelta] \mid \pi \Rightarrow \gamma \mid \gamma_1 @ \gamma_2 \mid \gamma_1\ !\ \gamma_2 \mid \mathit{pure}(\gamma) \mid \mathit{impure}(\gamma)
\end{array}$$

Fig. 4: ExEff Syntax

Coercion form $\gamma_1\ !\ \gamma_2$ witnesses subtyping for computation types, using proofs for their components. Inversely, syntactic forms *pure*(γ) and *impure*(γ) witness subtyping between the value- and dirt-components of a computation coercion.

Finally, coercion forms $\emptyset_{\varDelta}$ and $\{\texttt{Op}\} \cup \gamma$ are concerned with dirt subtyping. Form $\emptyset_{\varDelta}$ witnesses that the empty dirt $\emptyset$ is a subdirt of any dirt $\varDelta$. Lastly, coercion form $\{\texttt{Op}\} \cup \gamma$ witnesses that subtyping between dirts is preserved under extension with a new operation. Note that we do not have an inversion form to extract a witness for $\varDelta_1 \leqslant \varDelta_2$ from a coercion for $\{\texttt{Op}\} \cup \varDelta_1 \leqslant \{\texttt{Op}\} \cup \varDelta_2$. The reason is that dirt sets are sets and not inductive structures. For instance, for $\varDelta_1 = \{\texttt{Op}\}$ and $\varDelta_2 = \emptyset$ the latter subtyping holds, but the former does not.

4.2 Typing

Value & Computation Typing Typing for ExEff values and computations is presented in Figures 5 and 6 and is given by two mutually recursive relations of the form $\varGamma \vdash_{\mathsf{v}} v : T$ (values) and $\varGamma \vdash_{\mathsf{c}} c : \underline{C}$ (computations). ExEff typing environments $\varGamma$ contain bindings for variables of all sorts:

$$\varGamma ::= \epsilon \mid \varGamma, \varsigma \mid \varGamma, \alpha : \tau \mid \varGamma, \delta \mid \varGamma, x : T \mid \varGamma, \omega : \pi$$

Typing is entirely syntax-directed. Apart from the typing rules for skeleton, type, dirt, and coercion abstraction (and, subsequently, skeleton, type, dirt, and coercion

$$\frac{(x : T) \in \varGamma}{\varGamma \vdash_{\mathsf{v}} x : T} \qquad \frac{}{\varGamma \vdash_{\mathsf{v}} \texttt{unit} : \texttt{Unit}} \qquad \frac{\varGamma, x : T \vdash_{\mathsf{c}} c : \underline{C} \qquad \varGamma \vdash_{T} T : \tau}{\varGamma \vdash_{\mathsf{v}} (\texttt{fun}\ x : T \mapsto c) : T \to \underline{C}}$$

$$\frac{\varGamma \vdash_{\mathsf{v}} v : T_1 \qquad \varGamma \vdash_{\mathsf{co}} \gamma : T_1 \leqslant T_2}{\varGamma \vdash_{\mathsf{v}} v \rhd \gamma : T_2} \qquad \frac{\varGamma, \varsigma \vdash_{\mathsf{v}} v : T}{\varGamma \vdash_{\mathsf{v}} \varLambda\varsigma.v : \forall\varsigma.T} \qquad \frac{\varGamma, \alpha : \tau \vdash_{\mathsf{v}} v : T}{\varGamma \vdash_{\mathsf{v}} \varLambda\alpha : \tau.v : \forall\alpha : \tau.T}$$

$$\frac{\varGamma, \delta \vdash_{\mathsf{v}} v : T}{\varGamma \vdash_{\mathsf{v}} \varLambda\delta.v : \forall\delta.T} \qquad \frac{\varGamma, \omega : \pi \vdash_{\mathsf{v}} v : T \qquad \varGamma \vdash_{\rho} \pi}{\varGamma \vdash_{\mathsf{v}} \varLambda(\omega : \pi).v : \pi \Rightarrow T} \qquad \frac{\varGamma \vdash_{\mathsf{v}} v : \pi \Rightarrow T \qquad \varGamma \vdash_{\mathsf{co}} \gamma : \pi}{\varGamma \vdash_{\mathsf{v}} v\ \gamma : T}$$

$$\frac{\begin{array}{c}\varGamma, x : T_x \vdash_{\mathsf{c}} c_r : T \,!\, \varDelta \\ \left[(\texttt{Op} : T_1 \to T_2) \in \varSigma \qquad \varGamma, x : T_1, k : T_2 \to T \,!\, \varDelta \vdash_{\mathsf{c}} c_{\texttt{Op}} : T \,!\, \varDelta\right]_{\texttt{Op} \in \mathcal{O}}\end{array}}{\varGamma \vdash_{\mathsf{v}} \{\texttt{return}\ (x : T_x) \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\} : T_x \ !\ \varDelta \cup \mathcal{O} \Rrightarrow T \ !\ \varDelta}$$

$$\frac{\begin{array}{c}\varGamma \vdash_{\mathsf{v}} v : \forall\varsigma.T \\ \varGamma \vdash_{\tau} \tau\end{array}}{\varGamma \vdash_{\mathsf{v}} v\ \tau : T[\tau/\varsigma]} \qquad \frac{\begin{array}{c}\varGamma \vdash_{\mathsf{v}} v : \forall\alpha : \tau.T_1 \\ \varGamma \vdash_{T} T_2 : \tau\end{array}}{\varGamma \vdash_{\mathsf{v}} v\ T_2 : T_1[T_2/\alpha]} \qquad \frac{\begin{array}{c}\varGamma \vdash_{\mathsf{v}} v : \forall\delta.T \\ \varGamma \vdash_{\varDelta} \varDelta\end{array}}{\varGamma \vdash_{\mathsf{v}} v\ \varDelta : T[\varDelta/\delta]}$$

Fig. 5: ExEff Value Typing

application), the main difference between typing for ImpEff and ExEff lies in the explicit cast forms, $(v \rhd \gamma)$ and $(c \rhd \gamma)$. 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 $(v \rhd \gamma)$. Upcasting for computations works analogously.

Well-formedness of Types, Constraints, Dirts & Skeletons The definitions of the judgements that check the well-formedness of ExEff value types $(\varGamma \vdash_{T} T : \tau)$, computation types $(\varGamma \vdash_{\underline{C}} \underline{C} : \tau)$, dirts $(\varGamma \vdash_{\varDelta} \varDelta)$, and skeletons $(\varGamma \vdash_{\tau} \tau)$ are equally straightforward as those for ImpEff.

Coercion Typing Coercion typing formalizes the intuitive interpretation of coercions we gave in Section 4.1 and takes the form $\varGamma \vdash_{\mathsf{co}} \gamma : \rho$. It is essentially an extension of the constraint entailment relation of Figure 3.

4.3 Operational Semantics

Figure 7 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.

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

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

$$\frac{\Gamma \vdash_{\mathsf{c}} v : T}{\Gamma \vdash_{\mathsf{c}} \mathtt{return}\ v : T \mathrel{!} \emptyset} \qquad\qquad \frac{\Gamma \vdash_{\mathsf{c}} c_1 : T_1 \mathrel{!} \Delta \qquad \Gamma, x : T_1 \vdash_{\mathsf{c}} c_2 : T_2 \mathrel{!} \Delta}{\Gamma \vdash_{\mathsf{c}} \mathtt{do}\ x \leftarrow c_1 ; c_2 : T_2 \mathrel{!} \Delta}$$

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

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

Fig. 6: ExEff Computation Typing

$$\begin{array}{rrl}
\text{terminal value } v^T & ::= & \mathtt{unit} \mid h \mid \mathtt{fun}\ x : T \mapsto c \mid \Lambda\alpha : \tau . v \mid \Lambda\delta . v \mid \lambda\omega : \pi . v \\
\text{value result } v^R & ::= & v^T \mid v^T \rhd \gamma \\
\text{computation result } c^R & ::= & \mathtt{return}\ v^T \mid (\mathtt{return}\ v^T) \rhd \gamma \mid \mathtt{Op}\ v^R\ (y : T . c)
\end{array}$$

Terminal values v^T represent conventional values, and value results v^R can either be plain terminal values v^T or terminal values with a cast: $v^T \rhd \gamma$. The same applies to computation results c^R.[7]

Although unusual, this stratification can also be found in Crary's coercion calculus for inclusive subtyping [4], and, more recently, in System $\mathsf{F_C}$ [25]. Stratification is crucial for ensuring type preservation. Consider for example the expression $(\mathtt{return}\ 5 \rhd \langle \mathtt{int} \rangle \mathrel{!} \emptyset_{\{\mathtt{Op}\}})$, of type $\mathtt{int} \mathrel{!} \{\mathtt{Op}\}$. We can not reduce the expression further without losing effect information; removing the cast would result in computation $(\mathtt{return}\ 5)$, of type $\mathtt{int} \mathrel{!} \emptyset$. 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 7.

In relation $v \leadsto_{\mathrm{v}} v'$, the first rule groups nested casts into a single cast, by means of transitivity. The next three rules capture the essence of 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.

The situation in relation $c \leadsto_{\mathrm{c}} c'$ is quite similar. The first rule uses transitivity to group nested casts into a single cast. The second rule is a push rule for β-reduction. The third rule pushes a cast out of a `return`-computation. The fourth rule pushes a coercion inside an operation-computation, illustrating why the syntax for c^R does not require casts on operation-computations. The fifth rule is a push rule for sequencing

[7] Observe that operation values do not feature an outermost cast operation, as the coercion can always be pushed into its continuation.

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

$$(v^T \rhd \gamma_1) \rhd \gamma_2 \rightsquigarrow_{\mathrm{v}} v^T \rhd (\gamma_1 \gg \gamma_2) \qquad (v^T \rhd \gamma)\ T \rightsquigarrow_{\mathrm{v}} (v^T\ T) \rhd \gamma[T]$$

$$(v^T \rhd \gamma)\ \mathit{\Delta} \rightsquigarrow_{\mathrm{v}} (v^T\ \mathit{\Delta}) \rhd \gamma[\mathit{\Delta}] \qquad (v^T \rhd \gamma_1)\ \gamma_2 \rightsquigarrow_{\mathrm{v}} (v^T\ \gamma_2) \rhd \gamma_1 @ \gamma_2$$

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

$$(c^R \rhd \gamma_1) \rhd \gamma_2 \rightsquigarrow_{\mathrm{c}} c^R \rhd (\gamma_1 \gg \gamma_2) \qquad (v_1^T \rhd \gamma)\ v_2 \rightsquigarrow_{\mathrm{c}} (v_1^T\ (v_2 \rhd \mathit{left}(\gamma))) \rhd \mathit{right}(\gamma)$$

$$\mathtt{return}\ (v^T \rhd \gamma) \rightsquigarrow_{\mathrm{c}} (\mathtt{return}\ v^T) \rhd (\gamma\ !\ \emptyset_\emptyset)$$

$$(\mathtt{Op}\ v^R\ (y : T.c)) \rhd \gamma \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v^R\ (y : T.(c \rhd \gamma))$$

$$\mathtt{do}\ x \leftarrow ((\mathtt{return}\ v^T) \rhd \gamma); c_2 \rightsquigarrow_{\mathrm{c}} c_2[(v^T \rhd \mathit{pure}(\gamma))/x]$$

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

$$\mathtt{handle}\ c\ \mathtt{with}\ (v^T \rhd \gamma) \rightsquigarrow_{\mathrm{c}} (\mathtt{handle}\ (c \rhd \mathit{left}(\gamma))\ \mathtt{with}\ v^T) \rhd \mathit{right}(\gamma)$$

$$\mathtt{handle}\ ((\mathtt{return}\ v^T) \rhd \gamma)\ \mathtt{with}\ h \rightsquigarrow_{\mathrm{c}} c_r[v^T \rhd \mathit{pure}(\gamma)/x]$$

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

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

Fig. 7: ExEff Operational Semantics (Selected Rules)

computations 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 γ, and (b) substitute x with v^T, cast with the pure part of γ (so that types are preserved). The sixth rule 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. The first of them 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). The second rule behaves similarly to the push/beta rule for sequencing computations. Finally, the last two rules are concerned with handling of operations. The first of the two 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 [20], ExEff features *deep handlers*: the continuation is also wrapped within a `with-handle` construct. The last rule captures cases where the operation is not covered by the handler and thus remains unhandled.

We have shown that ExEff is type safe:

Theorem 1 (Type Safety).

- *If $\Gamma \vdash_{\mathsf{v}} v : T$ then either v is a result value or $v \rightsquigarrow_{\mathrm{v}} v'$ and $\Gamma \vdash_{\mathsf{v}} v' : T$.*
- *If $\Gamma \vdash_{\mathsf{c}} c : \underline{C}$ then either c is a result computation or $c \rightsquigarrow_{\mathrm{c}} c'$ and $\Gamma \vdash_{\mathsf{c}} c' : \underline{C}$.*

5 Type Inference & 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 grayed parts of Figure 2 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. The binders are eliminated in rule TmVar by means of explicit application with skeletons, types, dirts and coercions. The coercions are produced by the auxiliary judgement $\Gamma \vdash_{\mathsf{co}} \gamma : \pi$, defined in Figure 3, which provides a coercion witness for every subtyping proof.

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

Theorem 2 (Type Preservation).

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

Here $elab_\Gamma(\Gamma)$, $elab_S(A)$ and $elab_{\underline{C}}(\underline{C})$ convert ImpEff environments and types into ExEff environments and types.

5.2 Constraint Generation & Elaboration

Constraint generation with elaboration into ExEff is presented in Figures 8 (values) and 9 (computations). Before going into the details of each, we first introduce the three auxiliary constructs they use.

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

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

$$\frac{}{\mathcal{Q}; \Gamma \vdash_{\mathtt{v}} \mathtt{unit} : \mathtt{Unit} \mid \mathcal{Q}; \bullet \rightsquigarrow \mathtt{unit}}$$

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

$$\frac{\begin{array}{l}
\qquad \alpha_r : \varsigma_r, \mathcal{Q}; \Gamma, x : \alpha_r \vdash_{\mathtt{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_{\mathtt{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}' = \alpha_{in} : \varsigma_{in}, \alpha_{out} : \varsigma_{out}, \omega_1 : \sigma^n(B_r) \leqslant \alpha_{out}, \omega_2 : \sigma^n(\Delta_r) \leqslant \delta_{out}, \overline{\omega_{3_i} : \sigma^n(B_{\mathtt{Op}_i}) \leqslant \alpha_{out}}^n, \\
\quad \overline{\omega_{4_i} : \sigma^n(\Delta_{\mathtt{Op}_i}) \leqslant \delta_{out}}^n, \overline{\omega_{5_i} : B_i \to \alpha_{out} \mathbin{!} \delta_{out} \leqslant B_i \to \sigma^n(\alpha_i \mathbin{!} \delta_i)}^n, \\
\quad \omega_6 : \alpha_{in} \leqslant \sigma^n(\sigma_r(\alpha_r)), \omega_7 : \delta_{in} \leqslant \delta_{out} \cup \mathcal{O}, \mathcal{Q}_n \\
\ c_{res} = \{\mathtt{return}\ y : \sigma^n(\sigma_r(\alpha_r)) \mapsto \sigma^n(c'_r)[y \rhd \omega_6/x] \rhd \omega_1 \mathbin{!} \omega_2 \\
\qquad\quad , [\mathtt{Op}_i\ x\ l \mapsto \sigma^n(c'_{\mathtt{Op}_i})[l \rhd \omega_{5_i}/k] \rhd \omega_{3_i} \mathbin{!} \omega_{4_i}]_{\mathtt{Op}_i \in \mathcal{O}}\} \rhd (\langle \alpha_{in} \rangle \mathbin{!} \omega_7 \Rrightarrow \langle \alpha_{out} \rangle \mathbin{!} \langle \delta_{out} \rangle)
\end{array}}{\mathcal{Q}; \Gamma \vdash_{\mathtt{v}} \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : \alpha_{in} \mathbin{!} \delta_{in} \Rrightarrow \alpha_{out} \mathbin{!} \delta_{out} \mid \mathcal{Q}'; (\sigma^n \cdot \sigma_r) \rightsquigarrow c_{res}}$$

Fig. 8: Constraint Generation with Elaboration (Values)

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

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) 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.

A substitution σ is a solution of the set $\mathcal{P}$, written as $\sigma \models \mathcal{P}$, if we get derivable judgements after applying σ to all constraints in $\mathcal{P}$.

Values. Constraint generation for values takes the form $\mathcal{Q}; \Gamma \vdash_{\mathtt{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'$. 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'.

Unlike standard HM, 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. We discuss the reason for this interleaved approach in Section 5.4; we now focus on the algorithm.

The rules are syntax-directed on the input ImpEff value. The first rule 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 the third rule, for term abstractions. Like in standard Hindley-Damas-Milner [5], 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 $\alpha : \varsigma$.[8] We apply the resulting substitution σ to the result type $\sigma(\alpha) \to \underline{C}$.[9]

Finally, the fourth rule 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

$$\alpha_{in} \,!\, \delta_{in} \Rrightarrow \alpha_{out} \,!\, \delta_{out}$$

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

[8] This hints at why we need to pass constraints in a stateful manner.

[9] Though σ refers to ImpEff types, we abuse notation to save clutter and apply it directly to ExEff entities too.

$\boxed{\mathcal{Q}; \varGamma \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \leadsto c'}$ **Computations**

$$\frac{\mathcal{Q}; \varGamma \vdash_{\mathsf{v}} v_1 : A_1 \mid \mathcal{Q}_1; \sigma_1 \leadsto v_1' \qquad \mathcal{Q}_1; \sigma_1(\varGamma) \vdash_{\mathsf{v}} v_2 : A_2 \mid \mathcal{Q}_2; \sigma_2 \leadsto v_2'}{\mathcal{Q}; \varGamma \vdash_{\mathsf{c}} v_1\ v_2 : \alpha \,!\, \delta \mid \alpha : \varsigma, \omega : \sigma_2(A_1) \leqslant A_2 \to \alpha \,!\, \delta, \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \leadsto (\sigma_2(v_1') \rhd \omega)\ v_2'}$$

$$\frac{\mathcal{Q}; \varGamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}'; \sigma \leadsto v'}{\mathcal{Q}; \varGamma \vdash_{\mathsf{c}} \mathtt{return}\ v : A \,!\, \emptyset \mid \mathcal{Q}'; \sigma \leadsto \mathtt{return}\ v'}$$

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

$$\frac{\begin{array}{cc} \mathcal{Q}; \varGamma \vdash_{\mathsf{v}} v : A_1 \mid \mathcal{Q}_1; \sigma_1 \leadsto v' & \mathcal{Q}_1; \sigma_1(\varGamma), y : B_{\mathtt{Op}} \vdash_{\mathsf{c}} c : A_2 \,!\, \Delta_2 \mid \mathcal{Q}_2; \sigma_2 \leadsto c' \\ (\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma & c_{res} = \mathtt{Op}\ (\sigma_2(v') \rhd \omega)\ (y : \mathit{elab}_S(B_{\mathtt{Op}}).c') \end{array}}{\mathcal{Q}; \varGamma \vdash_{\mathsf{c}} \mathtt{Op}\ v\ (y : B_{\mathtt{Op}}.c) : A_2 \,!\, \{\mathtt{Op}\} \cup \Delta_2 \mid \omega : \sigma_2(A_1) \leqslant A_{\mathtt{Op}}, \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \leadsto c_{res}}$$

$$\frac{\begin{array}{c} \mathcal{Q}; \varGamma \vdash_{\mathsf{c}} c_1 : A_1 \,!\, \Delta_1 \mid \mathcal{Q}_1; \sigma_1 \leadsto c_1' \qquad \mathcal{Q}_1; \sigma_1(\varGamma), x : A_1 \vdash_{\mathsf{c}} c_2 : A_2 \,!\, \Delta_2 \mid \mathcal{Q}_2; \sigma_2 \leadsto c_2' \\ c_{res} = \mathtt{do}\ x \leftarrow (\sigma_2(c_1') \rhd \langle \sigma_2(A_1) \rangle \,!\, \omega_1); (c_2' \rhd \langle A_2 \rangle \,!\, \omega_2) \end{array}}{\mathcal{Q}; \varGamma \vdash_{\mathsf{c}} \mathtt{do}\ x \leftarrow c_1; c_2 : A_2 \,!\, \delta \mid \omega_1 : \sigma_2(\Delta_1) \leqslant \delta, \omega_2 : \Delta_2 \leqslant \delta, \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \leadsto c_{res}}$$

$$\frac{\begin{array}{c} \mathcal{Q}; \varGamma \vdash_{\mathsf{v}} v : A_1 \mid \mathcal{Q}_1; \sigma_1 \leadsto v' \qquad \mathcal{Q}_1; \sigma_1(\varGamma) \vdash_{\mathsf{c}} c : A_2 \,!\, \Delta_2 \mid \mathcal{Q}_2; \sigma_2 \leadsto c' \\ \mathcal{Q}' = \alpha_1 : \varsigma_1, \alpha_2 : \varsigma_2, \omega_1 : \sigma_2(A_1) \leqslant (\alpha_1 \,!\, \delta_1 \Rrightarrow \alpha_2 \,!\, \delta_2), \omega_2 : A_2 \leqslant \alpha_1, \omega_3 : \Delta_2 \leqslant \delta_1, \mathcal{Q}_2 \\ c_{res} = \mathtt{handle}\ (c' \rhd (\omega_2 \,!\, \omega_3))\ \mathtt{with}\ (\sigma_2(v') \rhd \omega_1) \end{array}}{\mathcal{Q}; \varGamma \vdash_{\mathsf{c}} \mathtt{handle}\ c\ \mathtt{with}\ v : \alpha_2 \,!\, \Delta_2 \mid \mathcal{Q}'; (\sigma_2 \cdot \sigma_1) \leadsto c_{res}}$$

Fig. 9: Constraint Generation with Elaboration (Computations)

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 $\mathcal{Q}; \varGamma \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \leadsto c'$ generates constraints for computations.

The first rule 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 $A_2 \to \alpha \,!\, \delta$.

The third rule 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* $\mathcal{Q}'_v$.

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

$$\frac{\begin{array}{c}\bar{\varsigma} = \{\varsigma \mid (\alpha : \varsigma) \in \mathcal{Q}, \nexists \alpha'.\alpha' \notin \bar{\alpha} \wedge (\alpha' : \varsigma) \in \mathcal{Q}\} \\ \bar{\alpha} = fv_\alpha(\mathcal{Q}) \cup fv_\alpha(A) \setminus fv_\alpha(\Gamma) \qquad \mathcal{Q}_1 = \{(\omega : \pi) \mid (\omega : \pi) \in \mathcal{Q}, fv(\pi) \not\subseteq fv(\Gamma)\} \\ \bar{\delta} = fv_\delta(\mathcal{Q}) \cup fv_\delta(A) \setminus fv_\delta(\Gamma) \qquad \mathcal{Q}_2 = \mathcal{Q} - \mathcal{Q}_1\end{array}}{split(\Gamma, \mathcal{Q}, A) = \langle \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \mathcal{Q}_1, \mathcal{Q}_2 \rangle}$$

In essence, $split$ generates the type (scheme) of x in parts. Additionally, it computes the subset $\mathcal{Q}_2$ of the input constraints $\mathcal{Q}$ 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.

The fourth rule handles operation calls. Observe that in the elaborated term, we upcast the inferred type to match the expected type in the signature.

The fifth rule 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, the sixth rule 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 $\omega_1 : \sigma_2(A_1) \leqslant (\alpha_1 \,!\, \delta_1 \Rrightarrow \alpha_2 \,!\, \delta_2)$, 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 $\omega_3 : \Delta_2 \leqslant \delta_1$.

Theorem 3 (Soundness of Inference). *If $\bullet; \Gamma \vdash_{\mathrm{v}} v : A \mid \mathcal{Q}; \sigma \rightsquigarrow v'$ then for any $\sigma' \models \mathcal{Q}$, we have $(\sigma' \cdot \sigma)(\Gamma) \vdash_v v : \sigma'(A) \rightsquigarrow \sigma'(v')$, and analogously for computations.*

Theorem 4 (Completeness of Inference). *If $\Gamma \vdash_v v : A \rightsquigarrow v'$ then we have $\bullet; \Gamma \vdash_{\mathrm{v}} v : A' \mid \mathcal{Q}; \sigma \rightsquigarrow v''$ and there exists $\sigma' \models \mathcal{Q}$ and γ, such that $\sigma'(v'') = v'$ and $\sigma(\Gamma) \vdash_{\mathrm{co}} \gamma : \sigma'(A') \leqslant A$. 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:

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

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 $\mathcal{Q}$. 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 $\omega \to \gamma$ which instantiate respectively skeleton variables, type variables, dirt variables and coercion variables.

Theorem 5 (Correctness of Solving). *For any set $\mathcal{Q}$, the call* $\texttt{solve}(\bullet; \bullet; \mathcal{Q})$ *either results in a failure, in which case $\mathcal{Q}$ has no solutions, or returns $(\sigma, \mathcal{P})$ such that for any $\sigma' \models \mathcal{Q}$, 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, i.e., with an empty substitution and no processed constraints. The `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 [5].

```
$\texttt{solve}(\sigma;\ \mathcal{P}; \tau_1 = \tau_2, \mathcal{Q}) =$
  $\texttt{match}\ \tau_1 = \tau_2\ \texttt{with}$
   $\mid \varsigma = \varsigma \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \mathcal{Q})$
   $\mid \varsigma = \tau \mapsto \texttt{if}\ \varsigma \notin \mathit{fv}_\varsigma(\tau)\ \texttt{then let}\ \sigma' = [\tau/\varsigma]\ \texttt{in solve}(\sigma' \cdot \sigma;\ \bullet; \sigma'(\mathcal{Q}, \mathcal{P}))\ \texttt{else fail}$
   $\mid \tau = \varsigma \mapsto \texttt{if}\ \varsigma \notin \mathit{fv}_\varsigma(\tau)\ \texttt{then let}\ \sigma' = [\tau/\varsigma]\ \texttt{in solve}(\sigma' \cdot \sigma;\ \bullet; \sigma'(\mathcal{Q}, \mathcal{P}))\ \texttt{else fail}$
   $\mid \texttt{Unit} = \texttt{Unit} \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \mathcal{Q})$
   $\mid (\tau_1 \to \tau_2) = (\tau_3 \to \tau_4) \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \tau_1 = \tau_3, \tau_2 = \tau_4, \mathcal{Q})$
   $\mid (\tau_1 \Rrightarrow \tau_2) = (\tau_3 \Rrightarrow \tau_4) \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \tau_1 = \tau_3, \tau_2 = \tau_4, \mathcal{Q})$
   $\mid \texttt{otherwise} \mapsto \texttt{fail}$
```

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. The instantiating substitution is accumulated and applied to the remaining constraints, which are processed recursively.

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

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

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

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[10] to the queue. This enforces 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. The solver then 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 $skeleton(A)$ computes the skeleton of A.

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

```
solve($\sigma;\ \mathcal{P}; \omega : \Delta \leqslant \Delta', \mathcal{Q}$) =
 match $\Delta \leqslant \Delta'$ with
  | $\mathcal{O} \cup \delta \leqslant \mathcal{O}' \cup \delta' \mapsto$ if $\mathcal{O} \neq \emptyset$ then let $\sigma' = [((\mathcal{O} \backslash \mathcal{O}') \cup \delta'')/\delta', \mathcal{O} \cup \omega'/\omega]$ in
                                           solve($\sigma' \cdot \sigma;\ \bullet; (\omega' : \delta \leq \sigma'(\Delta')), \sigma'(\mathcal{Q}, \mathcal{P})$)
                                    else solve($\sigma;\ \mathcal{P}, (\omega : \Delta \leqslant \Delta');\ \mathcal{Q}$)
  | $\emptyset \leqslant \Delta' \mapsto$ solve($[\emptyset_{\Delta'}/\omega] \cdot \sigma;\ \mathcal{P};\ \mathcal{Q}$)
  | $\delta \leqslant \emptyset \mapsto$ let $\sigma' = [\emptyset/\delta;\ \emptyset_\emptyset/\omega]$ in solve($\sigma' \cdot \sigma;\ \bullet;\ \sigma'(\mathcal{Q}, \mathcal{P})$)
  | $\mathcal{O} \cup \delta \leqslant \mathcal{O}' \mapsto$
     if $\mathcal{O} \subseteq \mathcal{O}'$ then let $\sigma' = [\mathcal{O} \cup \omega'/\omega]$ in solve($\sigma' \cdot \sigma;\ \mathcal{P}, (\omega' : \delta \leqslant \mathcal{O}');\ \mathcal{Q}$) else fail
  | $\mathcal{O} \leqslant \mathcal{O}' \mapsto$ if $\mathcal{O} \subseteq \mathcal{O}'$ then let $\sigma' = [\mathcal{O} \cup \emptyset_{\mathcal{O}' \backslash \mathcal{O}}/\omega]$ in solve($\sigma' \cdot \sigma;\ \mathcal{P};\ \mathcal{Q}$) else fail
  | $\mathcal{O} \leqslant \mathcal{O}' \cup \delta' \mapsto$ let $\sigma' = [(\mathcal{O} \backslash \mathcal{O}') \cup \delta''/\delta';\ \mathcal{O}' \cup \emptyset_{(\mathcal{O}' \backslash \mathcal{O}) \cup \delta''}/\omega]$ in
                    solve($\sigma' \cdot \sigma;\ \bullet;\ \sigma'(\mathcal{Q}, \mathcal{P})$)
```

If the two dirts are of the general form $\mathcal{O} \cup \delta$ and $\mathcal{O}' \cup \delta'$, we distinguish two subcases. 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,

[10] We implicitly annotate every type variable with its skeleton: α^τ.

Terms

$$\begin{array}{rrl}
\text{value } v & ::= & x \mid \texttt{unit} \mid h \mid \texttt{fun}\ (x : \tau) \mapsto c \mid \varLambda \varsigma . v \mid v\ \tau \\
\text{handler } h & ::= & \{\texttt{return}\ (x : \tau) \mapsto c_r, \texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \dots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n}\} \\
\text{computation } c & ::= & v_1\ v_2 \mid \texttt{let}\ x = v\ \texttt{in}\ c \mid \texttt{return}\ v \mid \texttt{Op}\ v\ (y : \tau . c) \\
 & \mid & \texttt{do}\ x \leftarrow c_1 ; c_2 \mid \texttt{handle}\, c\, \texttt{with}\, v
\end{array}$$

Types

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

Fig. 10: SkelEff Syntax

$\mathcal{O} \leqslant \mathcal{O}' \cup \delta'$, 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.

5.4 Discussion

At first glance, the constraint generation algorithm of Section 5.2 might seem needlessly complex, due to eager constraint solving for let-generalization. Yet, we want to generalize at local `let`-bound values over both type and skeleton variables,[11] which means that we must solve all equations between skeletons before generalizing. In turn, since skeleton constraints are generated when solving subtyping constraints (Section 5.3), all skeleton annotations should be available during constraint solving. This can not be achieved unless the generated constraints are propagated statefully.

6 Erasure of Effect Information from ExEff

6.1 The SkelEff Language

The target of the erasure is SkelEff, which is essentially a copy of ExEff from which all effect information $\varDelta$, type information T and coercions γ have been removed. 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. Figure 10 defines the syntax of SkelEff. The type system and operational semantics of SkelEff follow from those of ExEff.

Discussion 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 [6], which provides native support for algebraic effects and handlers but features no explicit polymorphism. Moreover, SkelEff can also serve as a staging area for further elaboration into System F-like languages without support for algebraic effects and handlers (e.g., Haskell or regular OCaml). In those cases, computation terms can

[11] As will become apparent in Section 6, if we only generalize at the top over skeleton variables, the erasure does not yield local polymorphism.

$$\begin{aligned}
\epsilon_{\mathrm{v}}^{\sigma}(x) &= x & \epsilon_{\mathrm{v}}^{\sigma}(\varLambda\delta.v) &= \epsilon_{\mathrm{v}}^{\sigma}(v) \\
\epsilon_{\mathrm{v}}^{\sigma}(\mathtt{unit}) &= \mathtt{unit} & \epsilon_{\mathrm{v}}^{\sigma}(\varLambda(\omega:\pi).v) &= \epsilon_{\mathrm{v}}^{\sigma}(v) \\
\epsilon_{\mathrm{v}}^{\sigma}(v \rhd \gamma) &= \epsilon_{\mathrm{v}}^{\sigma}(v) & \epsilon_{\mathrm{v}}^{\sigma}(v\ \tau) &= \epsilon_{\mathrm{v}}^{\sigma}(v)\ \tau \\
\epsilon_{\mathrm{v}}^{\sigma}(\mathtt{fun}\ (x:T) \mapsto c) &= \mathtt{fun}\ (x:\epsilon_{\mathrm{V}}^{\sigma}(T)) \mapsto \epsilon_{\mathrm{c}}^{\sigma}(c) & \epsilon_{\mathrm{v}}^{\sigma}(v\ T) &= \epsilon_{\mathrm{v}}^{\sigma}(v) \\
\epsilon_{\mathrm{v}}^{\sigma}(\varLambda\varsigma.v) &= \varLambda\varsigma.\epsilon_{\mathrm{v}}^{\sigma}(v) & \epsilon_{\mathrm{v}}^{\sigma}(v\ \varDelta) &= \epsilon_{\mathrm{v}}^{\sigma}(v) \\
\epsilon_{\mathrm{v}}^{\sigma}(\varLambda(\alpha:\tau).v) &= \epsilon_{\mathrm{v}}^{\sigma\cdot\{\alpha\mapsto\tau\}}(v) & \epsilon_{\mathrm{v}}^{\sigma}(v\ \gamma) &= \epsilon_{\mathrm{v}}^{\sigma}(v)
\end{aligned}$$

$$\begin{aligned}
&\epsilon_{\mathrm{v}}^{\sigma}(\{\mathtt{return}\ (x:T) \mapsto c_r, [\mathtt{Op}\,x\,k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op}\in\mathcal{O}}\}) = \\
&\quad \{\mathtt{return}\ (x:\epsilon_{\mathrm{V}}^{\sigma}(T)) \mapsto \epsilon_{\mathrm{c}}^{\sigma}(c_r), [\mathtt{Op}\,x\,k \mapsto \epsilon_{\mathrm{c}}^{\sigma}(c_{\mathtt{Op}})]_{\mathtt{Op}\in\mathcal{O}}\}
\end{aligned}$$

$$\begin{aligned}
\epsilon_{\mathrm{c}}^{\sigma}(v_1\ v_2) &= \epsilon_{\mathrm{v}}^{\sigma}(v_1)\ \epsilon_{\mathrm{v}}^{\sigma}(v_2) \\
\epsilon_{\mathrm{c}}^{\sigma}(\mathtt{let}\ x = v\ \mathtt{in}\ c) &= \mathtt{let}\ x = \epsilon_{\mathrm{v}}^{\sigma}(v)\ \mathtt{in}\ \epsilon_{\mathrm{c}}^{\sigma}(c) \\
\epsilon_{\mathrm{c}}^{\sigma}(\mathtt{return}\ v) &= \mathtt{return}\ (\epsilon_{\mathrm{v}}^{\sigma}(v)) \\
\epsilon_{\mathrm{c}}^{\sigma}(\mathtt{Op}\ v\ (y:T.c)) &= \mathtt{Op}\ (\epsilon_{\mathrm{v}}^{\sigma}(v))\ (y:\epsilon_{\mathrm{V}}^{\sigma}(T).\epsilon_{\mathrm{c}}^{\sigma}(c)) \\
\epsilon_{\mathrm{c}}^{\sigma}(\mathtt{do}\ x \leftarrow c_1; c_2) &= \mathtt{do}\ x \leftarrow \epsilon_{\mathrm{c}}^{\sigma}(c_1); \epsilon_{\mathrm{c}}^{\sigma}(c_2) \\
\epsilon_{\mathrm{c}}^{\sigma}(\mathtt{handle}\ c\ \mathtt{with}\ v) &= \mathtt{handle}\ \epsilon_{\mathrm{c}}^{\sigma}(c)\ \mathtt{with}\ \epsilon_{\mathrm{v}}^{\sigma}(v) \\
\epsilon_{\mathrm{c}}^{\sigma}(c \rhd \gamma) &= \epsilon_{\mathrm{c}}^{\sigma}(c)
\end{aligned}$$

$$\begin{aligned}
\epsilon_{\mathrm{V}}^{\sigma}(\alpha) &= \sigma(\alpha) & \epsilon_{\mathrm{C}}^{\sigma}(T\ !\ \varDelta) &= \epsilon_{\mathrm{V}}^{\sigma}(T) \\
\epsilon_{\mathrm{V}}^{\sigma}(T \to \underline{C}) &= \epsilon_{\mathrm{V}}^{\sigma}(T) \to \epsilon_{\mathrm{C}}^{\sigma}(\underline{C}) & & \\
\epsilon_{\mathrm{V}}^{\sigma}(\underline{C}_1 \Rrightarrow \underline{C}_2) &= \epsilon_{\mathrm{C}}^{\sigma}(\underline{C}_1) \Rrightarrow \epsilon_{\mathrm{C}}^{\sigma}(\underline{C}_2) & \epsilon_{\mathrm{E}}^{\sigma}(\epsilon) &= \epsilon \\
\epsilon_{\mathrm{V}}^{\sigma}(\mathtt{Unit}) &= \mathtt{Unit} & \epsilon_{\mathrm{E}}^{\sigma}(\varGamma, \varsigma) &= \epsilon_{\mathrm{E}}^{\sigma}(\varGamma), \varsigma \\
\epsilon_{\mathrm{V}}^{\sigma}(\pi \Rightarrow T) &= \epsilon_{\mathrm{V}}^{\sigma}(T) & \epsilon_{\mathrm{E}}^{\sigma}(\varGamma, \alpha:\tau) &= \epsilon_{\mathrm{E}}^{\sigma\cdot\{\alpha\mapsto\tau\}}(\varGamma) \\
\epsilon_{\mathrm{V}}^{\sigma}(\forall\varsigma.T) &= \forall\varsigma.\epsilon_{\mathrm{V}}^{\sigma}(T) & \epsilon_{\mathrm{E}}^{\sigma}(\varGamma, \delta) &= \epsilon_{\mathrm{E}}^{\sigma}(\varGamma) \\
\epsilon_{\mathrm{V}}^{\sigma}(\forall(\alpha:\tau).T) &= \epsilon_{\mathrm{V}}^{\sigma\cdot\{\alpha\mapsto\tau\}}(T) & \epsilon_{\mathrm{E}}^{\sigma}(\varGamma, x:T) &= \epsilon_{\mathrm{E}}^{\sigma}(\varGamma), x:\epsilon_{\mathrm{V}}^{\sigma}(T) \\
\epsilon_{\mathrm{V}}^{\sigma}(\forall\delta.T) &= \epsilon_{\mathrm{V}}^{\sigma}(T) & \epsilon_{\mathrm{E}}^{\sigma}(\varGamma, \omega:\pi) &= \epsilon_{\mathrm{E}}^{\sigma}(\varGamma)
\end{aligned}$$

Fig. 11: Definition of type erasure.

be compiled to one of the known encodings in the literature, such as a free monad representation [10, 22], with delimited control [11], or using continuation-passing style [13], while values can typically be carried over as they are.

6.2 Erasure

Figure 11 defines erasure functions $\epsilon_{\mathrm{v}}^{\sigma}(v)$, $\epsilon_{\mathrm{c}}^{\sigma}(c)$, $\epsilon_{\mathrm{V}}^{\sigma}(T)$, $\epsilon_{\mathrm{C}}^{\sigma}(\underline{C})$ and $\epsilon_{\mathrm{E}}^{\sigma}(\varGamma)$ 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.

The expected theorems hold. Firstly, types are preserved by erasure.[12]

[12] Typing for SkelEff values and computations take the form $\varGamma \vdash_{\mathsf{ev}} v : \tau$ and $\varGamma \vdash_{\mathsf{ec}} c : \tau$.

Theorem 6 (Type Preservation). *If* $\mathit{\Gamma} \vdash_{\mathsf{v}} v : T$ *then* $\epsilon^{\emptyset}_{\mathrm{E}}(\mathit{\Gamma}) \vdash_{\mathsf{ev}} \epsilon^{\mathit{\Gamma}}_{\mathrm{v}}(v) : \epsilon^{\mathit{\Gamma}}_{\mathrm{V}}(T)$*. If* $\mathit{\Gamma} \vdash_{\mathsf{c}} c : \underline{C}$ *then* $\epsilon^{\emptyset}_{\mathrm{E}}(\mathit{\Gamma}) \vdash_{\mathsf{ec}} \epsilon^{\mathit{\Gamma}}_{\mathrm{c}}(c) : \epsilon^{\mathit{\Gamma}}_{\mathrm{C}}(\underline{C})$*.*

Here we abuse of notation and use $\mathit{\Gamma}$ as a substitution from type variables to skeletons used by the erasure functions.

Finally, we have that erasure preserves the operational semantics.

Theorem 7 (Semantic Preservation). *If* $v \rightsquigarrow_{\mathrm{v}} v'$ *then* $\epsilon^{\sigma}_{\mathrm{v}}(v) \equiv^{\rightsquigarrow}_{\mathrm{v}} \epsilon^{\sigma}_{\mathrm{v}}(v')$*. If* $c \rightsquigarrow_{\mathrm{c}} c'$ *then* $\epsilon^{\sigma}_{\mathrm{c}}(c) \equiv^{\rightsquigarrow}_{\mathrm{c}} \epsilon^{\sigma}_{\mathrm{c}}(c')$*.*

In both cases, $\equiv^{\rightsquigarrow}$ denotes the congruence closure of the step relation in SkelEff. The choice of substitution σ does not matter as types do not affect the behaviour.

Discussion 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:

$$\epsilon^{\sigma}_{\mathrm{v}}(\mathit{\Lambda}(\alpha : \tau).v) = \lambda(x : \texttt{Unit}).\epsilon^{\sigma}_{\mathrm{v}}(v)$$

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.

7 Related Work & Conclusion

Eff's Implicit Type System The most closely related work is that of Pretnar [20] 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 standard occurs-check during unification. In contrast, unification in Pretnar's algorithm is inspired by Simonet [24] and performs the occurs-check up to the equivalence closure of the subtyping relation. In order to maintain invariants, all variables in an equivalence class (also called a skeleton) must be instantiated simultaneously, whereas we can process one constraint at a time. As these classes turn out to be surrogates for the underlying skeleton types, we have decided to keep the name.

Finally, Pretnar incorporates garbage collection of constraints [19]. 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 α_1, α_2, α_3, α_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. 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 [15] 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. [3] 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, they do not include inversion forms, which are crucial for the proof of type safety for our system. Moreover, Breazu-Tannen et al.'s coercions are terms, and thus can not be erased.

Much closer to ExEff is Crary's coercion calculus for inclusive subtyping [4], from which we borrowed the stratification of value results. Crary's system supports neither coercion abstraction nor coercion inversion forms.

System $\mathsf{F_C}$ [25] uses explicit type-equality coercions to encode complex language features (e.g. GADTs [16] or type families [23]). Though ExEff's coercions are proofs of subtyping rather than type equality, our system has a lot in common with it, including the inversion coercion forms and the "push" rules.

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 optimizer more robust, but also expose new optimization opportunities. Next, we plan to write compilers to both Multicore OCaml and standard OCaml, though for the latter, we must first adapt the notion of erasure to a target calculus without algebraic effect handlers. 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 We would like to thank the anonymous reviewers for careful reading and insightful comments. Part of this work is funded by the Flemish Fund for Scientific Research (FWO). This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

Bibliography

[1] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics, volume 103 of Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logic and Algebraic Programming*, 84(1):108–123, 2015.

[3] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation vol*, 93:172–221, 1991.

[4] K. Crary. Typed compilation of inclusive subtyping. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 68–81, NY, USA, 2000. ACM.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, NY, USA, 1982. ACM.

[6] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, 2015.

[7] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[8] D. Hillerström and S. Lindley. Liberating effects with rows and handlers. In J. Chapman and W. Swierstra, editors, *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pages 15–27. ACM, 2016.

[9] M. P. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 1992.

[10] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '14, pages 145–158. ACM, 2013.

[11] O. Kiselyov and K. Sivaramakrishnan. Eff directly in ocaml. In *OCaml Workshop*, 2016.

[12] D. Leijen. Koka: Programming with row polymorphic effect types. In P. Levy and N. Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.*, volume 153 of *EPTCS*, pages 100–126, 2014.

[13] D. Leijen. Type directed compilation of row-typed algebraic effects. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499. ACM, 2017.

[14] S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 500–514. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009897.

[15] J. C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 175–185, New York, NY, USA, 1984. ACM.

[16] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *ICFP '06*, 2006.

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

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

[19] F. Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170(2):153–183, 2001. doi: 10.1006/inco.2001.2963. URL http://dx.doi.org/10.1006/inco.2001.2963.

[20] M. Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014.

[21] M. Pretnar. An introduction to algebraic effects and handlers, invited tutorial. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015.

[22] M. Pretnar, A. H. Saleh, A. Faes, and T. Schrijvers. Efficient compilation of algebraic effects and handlers. Technical Report CW 708, KU Leuven Department of Computer Science, 2017.

[23] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08*, pages 51–62. ACM, 2008.

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

[25] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[26] K. Wansbrough and S. L. Peyton Jones. Once upon a polymorphic type. In *POPL*, pages 15–28. ACM, 1999.