JFP **30**, e13, 27 pages, 2020. © The Author(s) 2020. Published by Cambridge University Press
doi:10.1017/S0956796819000212

Local algebraic effect theories

ŽIGA LUKŠIČ AND MATIJA PRETNAR[†]
Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
(*e-mails:* `ziga.luksic@fmf.uni-lj.si`, `matija.pretnar@fmf.uni-lj.si`)

Abstract

Algebraic effects are computational effects that can be described with a set of basic operations and equations between them. As many interesting effect handlers do not respect these equations, most approaches assume a trivial theory, sacrificing both reasoning power and safety. We present an alternative approach where the type system tracks equations that are observed in subparts of the program, yielding a sound and flexible logic, and paving a way for practical optimisations and reasoning tools.

1 Introduction

Algebraic effects are computational effects that can be described by a *signature* of primitive operations and a collection of equations between them (Plotkin & Power, 2001, 2003), while algebraic effect *handlers* are a generalisation of exception handlers to arbitrary algebraic effects (Plotkin & Pretnar, 2009, 2013). Even though the early work considered only handlers that respect equations of the effect theory, a considerable amount of useful handlers did not, and the restriction was dropped in most—though not all (Ahman, 2017, 2018)—of the later work on handlers (Kammar *et al.*, 2013; Bauer & Pretnar, 2015; Leijen, 2017; Biernacki *et al.*, 2018), resulting in a weaker reasoning logic and imprecise specifications.

Our aim is to rectify this by reintroducing effect theories into the type system, tracking equations observed in parts of a program. On one hand, the induced logic allows us to rewrite computations into equivalent ones with respect to the effect theory, while on the other hand, the type system enforces that handlers preserve equivalences, further specifying their behaviour. After an informal overview in Section 2, we proceed as follows:

- The syntax of the working language, its operational semantics, and the typing rules are given in Section 3.
- Determining if a handler respects an effect theory is in general undecidable (Plotkin & Pretnar, 2013), so there is no canonical way of defining such a judgement. Therefore, the typing rules are given parametric to a reasoning logic, and in Section 4, we present some of the most interesting choices.

[†] This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

- Since the definition of typing judgements is intertwined with a reasoning logic, we must be careful when defining the denotation of types and terms. Thus, in Section 5, we first introduce a set-based denotational semantics that disregards effect theories and prove the expected meta-theoretic properties.
- Next, in Section 6, we extend this denotation to templates and effect theories, and describe the necessary conditions for the reasoning logic which ensure its soundness and adequacy.

We conclude by discussing related and future work in Section 7.

2 Overview

2.1 Algebraic effect handlers

We assume the reader is vaguely familiar with algebraic effects and handlers, but we will elaborate on some of the more intricate parts. For less accustomed readers, a good place to pick up the basics is the tutorial (Pretnar, 2015). The emphasis will be on the newly introduced changes to the type and effect system.

At the core of using algebraic effects is the idea that all impure behaviour arises from calls of primitive operations, for example, *print* for printing a string or *raise* for raising an exception. Any computation either returns a value or makes an operation call $op(v;\ y.c)$ where the value v is a parameter of the call, while the computation c is its *continuation*, waiting for a result of op to be bound to y.

Suppose, we wish to model non-determinism. For that, we take an operation $choose$: $\texttt{unit} \to \texttt{bool}$ that non-deterministically produces a boolean after given the unit value $(\,)$. We can recover a binary non-deterministic choice from the abbreviation:

$$c_1 \oplus c_2 \overset{\text{def}}{=} choose((\,);\ y.\texttt{if}\ y\ \texttt{then}\ c_1\ \texttt{else}\ c_2)$$

Apart from a select few built-in operations (e.g. printing out to a terminal), an operation by itself has no meaning. Their meaning is instead determined by a handler consisting of a set of operation clauses $op(x;\ k) \mapsto c_{op}$. For every called operation op, the handler replaces the call with the handling computation c_{op}, where the parameter of the call is bound to x and the continuation is captured in a function, which is recursively handled by the same handler, and bound to k.

A simple example of a non-determinism handler is

$$\begin{aligned} &pickLeft\ = \texttt{handler}\,\{ \\ &\quad \mid choose((\,);k) \mapsto k\ \texttt{true} \\ &\} \end{aligned}$$

that makes *choose* constantly pass $\texttt{true}$ to the continuation k, forcing $c_1 \oplus c_2$ to always pick c_1. Another example is the handler that collects all the results of a non-deterministic computation:

```
collectToList = handler {
  | choose((); k) ↦
       do x₁ ← k true in
       do x₂ ← k false in
       ret (x₁@x₂)
  | ret x ↦ ret [x]
}
```

If the handled computation calls the operation *choose*, the handler passes both possible outcomes to the continuation, collects the respective results into lists x_1 and x_2, and returns the concatenated list. Additionally, the handler includes a clause stating that if the handled computation returns a value x, the handler should transform it into a computation returning the singleton list $[x]$.

2.2 Effect theories

Even though handlers determine the behaviour of operations, there are nonetheless some properties we expect from effects. These are described by a collection of equations called an *effect theory*. For non-determinism, the theory consists of equations for commutativity, idempotency, and associativity of the binary choice operation:

$$z_1 \oplus z_2 \sim z_2 \oplus z_1, \tag{COMM}$$

$$z \oplus z \sim z, \tag{IDEM}$$

$$z_1 \oplus (z_2 \oplus z_3) \sim (z_1 \oplus z_2) \oplus z_3 \tag{ASSOC}$$

We quickly notice that *pickLeft* does not respect the first equation by constructing a simple counterexample, for instance, let $z_1 = \texttt{ret}\ 1$ and $z_2 = \texttt{ret}\ 2$. When handling the left side of the equation, we obtain the result 1 (the left choice) while for the right side of the equation we get the result 2. Showing that *pickLeft* respects the last two equations requires some additional tools and is done in Example 4.2. Similarly, the handler *collectToList* respects the last equation but not the first two. The above two handlers are just two examples of many computationally interesting handlers that do not respect the usually assumed equations. For this reason, most contemporary work on algebraic effect handlers (Kammar *et al.*, 2013; Bauer & Pretnar, 2015; Leijen, 2017; Biernacki *et al.*, 2018) assumes trivial effect theories that contain no equations.

Our proposed solution is to instead annotate computation types with equations that define the desired effect theory. For instance, consider the function:

$$\mathit{chooseFromList} : A\ \texttt{list} \to A!\{\mathit{choose}\}$$

that takes a list of values of type A and non-deterministically chooses an element from it. The output type captures not only the type of returned values A, but also the set $\{\mathit{choose}\}$ of operations that may get called in the process. In our proposed system, we further decorate the output type by stating the desired equations:

$$\mathit{chooseFromList} : A\ \texttt{list} \to A!\{\mathit{choose}\}/\{(\text{COMM}),\ (\text{IDEM}),\ (\text{ASSOC})\}$$

Now, by knowing only the type of *chooseFromList*, its users can use induction (see Example 4.3) to show that computations:

$$\texttt{do}\ x_1 \leftarrow \mathit{chooseFromList}\ \ell_1\ \texttt{in}\ (\texttt{do}\ x_2 \leftarrow \mathit{chooseFromList}\ \ell_2\ \texttt{in}\ c)$$

and

$$\texttt{do}\ x_2 \leftarrow \mathit{chooseFromList}\ \ell_2\ \texttt{in}\ (\texttt{do}\ x_1 \leftarrow \mathit{chooseFromList}\ \ell_1\ \texttt{in}\ c)$$

are equivalent at type $A!\{\mathit{choose}\}/\{(\textsc{comm}), (\textsc{idem}), (\textsc{assoc})\}$ (but not at $A!\{\mathit{choose}\}/\emptyset$). Function implementers also benefit from the enriched types, as they can make additional assumptions on the observed behaviour. For example, by imposing the equation (COMM) on the output type, they ensure that the order in which they process the list is insignificant, as it will not be observable on the outside.

Any handler handling a computation with assumed equations must of course respect them. For example, *collectToList* transforms a computation of type $A!\{\mathit{choose}\}$ into a pure computation of type A `list`, but respects only (ASSOC). This is reflected in its type:

$$\mathit{collectToList} : A!\{\mathit{choose}\}/\{(\textsc{assoc})\} \Rightarrow A\ \texttt{list}!\emptyset/\emptyset$$

in which the output computation calls no operations, which in turn leads to no possible equations between them. As the equations in the input type of *collectToList* do not match those in the result of *chooseFromList*, the type system prohibits us from composing them. On the other hand, we could apply a handler *collectToSet* that collects all results into a set and thus additionally respects (COMM) and (IDEM).

The proposed type system offers greater flexibility over assuming a global effect theory (Plotkin & Pretnar, 2013), as in one part of a program, we can assume arbitrary equations that the locally used handlers respect, but still use computationally interesting handlers that respect other equations in a different part.

Another flexibility that the type system offers is transforming one effect theory into another. For example, take an operation $\mathit{yield} : \texttt{int} \to \texttt{unit}$ that is used to model integer generators. Using it, we design a handler that transforms a non-deterministic integer computation into a generator of all possible results:

$$\begin{aligned}
&\mathit{yieldAll} = \texttt{handler}\,\{\\
&\quad \mid \mathit{choose}((\,);\ k) \mapsto k\,\texttt{true}; k\,\texttt{false}\\
&\quad \mid \texttt{ret}\ x \mapsto \mathit{yield}(x;\ \_.\texttt{ret}\ (\,))\\
&\}
\end{aligned}$$

In the *choose* clause, we first resume the continuation by passing it `true`, yielding all outcomes in the process, and repeat the process for `false`. Whenever a computation returns a value x, we instead call *yield* with a trivial continuation.

This handler respects none of the non-deterministic equations stated above, so its immediate type is

$$\mathit{yieldAll} : \texttt{int}!\{\mathit{choose}\}/\emptyset \Rightarrow \texttt{unit}!\{\mathit{yield}\}/\emptyset.$$

But if we assume that the order of *yield* calls does not matter:

$$\mathit{yield}(x;\ \_.\mathit{yield}(y;\ \_.c)) \sim \mathit{yield}(y;\ \_.\mathit{yield}(x;\ \_.c)) \qquad (\textsc{yieldOrder})$$

the handler respects the commutativity of *choose* and can be given a type:

$$\mathit{yieldAll} : \mathtt{int}!\{\mathit{choose}\}/\{(\textsc{comm})\} \Rightarrow \mathtt{unit}!\{\mathit{yield}\}/\{(\textsc{yieldOrder})\},$$

as we show in Example 4.3. If we next have a handler that respects (YIELDORDER), for example,

$$\mathit{sumYieldedValues} : \mathtt{unit}!\{\mathit{yield}\}/\{(\textsc{yieldOrder})\} \Rightarrow \mathtt{int}!\emptyset/\emptyset,$$

and a computation c in which we assume (COMM), we can compose them as:

$$\mathtt{with}\ \mathit{sumYieldedValues}\ \mathtt{handle}\ (\mathtt{with}\ \mathit{yieldAll}\ \mathtt{handle}\ c),$$

with the type system ensuring that equations are preserved at appropriate places. Again, we can use the Equation (COMM) and rewrite c into an equivalent computation, all by knowing only the type and not the exact definition of handlers above it.

While the equations look simple, they are expressive enough to state even more intricate properties. Consider integer generators with an operation $\mathit{next} : \mathtt{unit} \to \mathtt{int\ option}$. A call of the operation *next* either generates the next element `Some` n of the generator or returns `None` if the generator has finished generating the sequence. We expect that after such a call, all further calls should result in `None`. We can specify such behaviour with the equation:

$$\mathit{next}((\,);\ y.\mathtt{if}\ (y == \mathtt{None})\ \mathtt{then}\ \mathit{next}((\,);\ y'.z\,y')\ \mathtt{else}\ z\,\mathtt{None}) \sim \mathit{next}((\,);\ y.z\,\mathtt{None})$$

where the variable z stands for an arbitrary computation, dependent on a value of type `int option`. To see that this equation describes the desired behaviour, consider any ill-behaved handler that passes `None` to the continuation on the first call of *next* and `Some` n on the second one. Handling the left-hand side first passes `None` to y, leading to the second call of *next*, which is handled by passing `Some` n to y' and further on to z. Handling the right-hand side, however, resumes by passing `None` to z, resulting in a different computation. In any other case (if the handler passes `Some` n in the first call or `None` in the second one), both sides proceed by handling z `None`.

3 Language

3.1 Term syntax

Our working calculus (Figure 1) is based on the fine-grain call-by-value (Levy *et al.*, 2003) approach, which differentiates between pure *values* v and effectful *computations* c, which might return a value or call an operation. For clarity, we keep the calculus minimal, though it could easily be extended with additional value types such as integers, type sums and products, or recursive function definitions, which we discuss in Section 7.2.

For a cleaner development later on, we define an independent syntactic sort of *operation clauses* h which are joint with a *return clause* only when constructing the handler value. This deviates slightly from most of the contemporary work on handlers (Leijen, 2017; Hillerström & Lindley, 2016; Forster *et al.*, 2017; Saleh *et al.*, 2018) and is more similar to the original treatment in Plotkin & Pretnar (2009, 2013). In practice, we treat operation clauses h as a set of operations with uniquely assigned handling computations and write them as $\{op(x;\ k) \mapsto c_{op}\}_{op}$.

$$
\begin{array}{rcll}
\text{values } v & ::= & x & \text{variable} \\
 & \mid & () & \text{unit constant} \\
 & \mid & \mathtt{true} \mid \mathtt{false} & \text{boolean constants} \\
 & \mid & \mathtt{fun}\ x \mapsto c & \text{function} \\
 & \mid & \mathtt{handler}\ (\mathtt{ret}\ x \mapsto c_r ; h) & \text{handler} \\
 \\
\text{computations } c & ::= & \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 & \text{conditional} \\
 & \mid & v_1\ v_2 & \text{application} \\
 & \mid & \mathtt{ret}\ v & \text{returned value} \\
 & \mid & op(v; y.c) & \text{operation call} \\
 & \mid & \mathtt{do}\ x \leftarrow c_1\ \mathtt{in}\ c_2 & \text{sequencing} \\
 & \mid & \mathtt{with}\ v\ \mathtt{handle}\ c & \text{handling} \\
 \\
\text{operation clauses } h & ::= & \emptyset \mid h \cup \{op(x; k) \mapsto c_{op}\} &
\end{array}
$$

Fig. 1. Syntax of terms.

$$
\frac{}{\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \rightsquigarrow c_1}
\qquad
\frac{}{\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \rightsquigarrow c_2}
$$

$$
\frac{}{(\mathtt{fun}\ x \mapsto c)\ v \rightsquigarrow c[v/x]}
\qquad
\frac{c_1 \rightsquigarrow c_1'}{\mathtt{do}\ x \leftarrow c_1\ \mathtt{in}\ c_2 \rightsquigarrow \mathtt{do}\ x \leftarrow c_1'\ \mathtt{in}\ c_2}
$$

$$
\frac{}{\mathtt{do}\ x \leftarrow \mathtt{ret}\ v\ \mathtt{in}\ c \rightsquigarrow c[v/x]}
\qquad
\frac{}{\mathtt{do}\ x \leftarrow op(v; y.c_1)\ \mathtt{in}\ c_2 \rightsquigarrow op(v; y.\mathtt{do}\ x \leftarrow c_1\ \mathtt{in}\ c_2)}
$$

$$
\frac{c \rightsquigarrow c'}{\mathtt{with}\ v\ \mathtt{handle}\ c \rightsquigarrow \mathtt{with}\ v\ \mathtt{handle}\ c'}
$$

$$
\frac{}{\mathtt{with}\ (\mathtt{handler}\ (\mathtt{ret}\ x \mapsto c_r ; h))\ \mathtt{handle}\ (\mathtt{ret}\ v) \rightsquigarrow c_r[v/x]}
$$

$$
\frac{(op(x; k) \mapsto c_{op}) \in h}{\begin{array}{l}\mathtt{with}\ (\mathtt{handler}\ (\mathtt{ret}\ x \mapsto c_r ; h))\ \mathtt{handle}\ (op(v; y.c)) \\ \rightsquigarrow c_{op}[v/x, (\mathtt{fun}\ y \mapsto \mathtt{with}\ (\mathtt{handler}\ (\mathtt{ret}\ x \mapsto c_r ; h))\ \mathtt{handle}\ c)/k]\end{array}}
$$

Fig. 2. Operational semantics.

To improve readability, we sometimes use syntactic sugar. When sequencing, we replace computations of form $\mathtt{do}\ \_ \leftarrow c_1\ \mathtt{in}\ c_2$ with $c_1; c_2$. When writing handler clauses, we often use the separator $\mid$ instead of commas, to achieve the familiar ML pattern matching look.

3.2 Operational semantics

Operational semantics, given in Figure 2, remains largely the same as in our previous work (Bauer & Pretnar, 2015; Kammar & Pretnar, 2017) except for a different presentation

(value) type A,B	$::=$	`unit`	unit type
	$\mid$	`bool`	boolean type
	$\mid$	$A \to \underline{C}$	function type
	$\mid$	$\underline{C} \Rightarrow \underline{D}$	handler type
computation type $\underline{C},\underline{D}$	$::=$	$A!\Sigma/\mathcal{E}$	
signature Σ	$::=$	$\emptyset \mid \Sigma \cup \{op : A \to B\}$	
value context Γ	$::=$	$\varepsilon \mid \Gamma, x : A$	
template context Z	$::=$	$\varepsilon \mid \mathrm{Z}, z : A \to *$	
template T	$::=$	$z\ v$	applied template variable
	$\mid$	`if` v `then` T_1 `else` T_2	conditional template
	$\mid$	$op(v;\ y.T)$	operation call template
(effect) theory $\mathcal{E}$	$::=$	$\emptyset \mid \mathcal{E} \cup \{\Gamma; \mathrm{Z} \vdash T_1 \sim T_2\}$	

Fig. 3. Syntax of types.

of operation clauses. Computations continue to evaluate until they either return a value or call an operation. At that point, we propagate the operation by pushing the remaining parts of the computation inside the operation continuation. This way we ensure that the operation eventually reaches a handler, while at the same time correcting the continuation.

In line with our previous work (Bauer & Pretnar, 2015; Kammar & Pretnar, 2017), the presented handlers are *deep* (Kammar *et al.*, 2013), meaning that they continue to handle any operations called in the potentially resumed continuation. For the sake of a slightly simpler type system, we choose to switch to *closed handlers* (Kammar *et al.*, 2013), which get stuck on operation calls with no corresponding operation clauses. Our type system will prevent such cases, though it is straightforward to extend the semantics to *open* handlers, where unhandled operations are implicitly propagated outwards, or add propagating cases $op(x;\ k) \mapsto op(x;\ y.k\ y)$ that do that explicitly.

3.3 Type syntax

Continuing the separation between values and computations, the type syntax (Figure 3) distinguishes between *value types* A, B, often referred to simply as types, and *computation types* $\underline{C}, \underline{D}$. In addition to the type of the returned value A, a computation type $A!\Sigma/\mathcal{E}$ captures the signature Σ of operations that might be called during evaluation and, the main novelty of this paper, a set of equations, called a *(effect) theory* $\mathcal{E}$, which describes which computations we consider equivalent at a given type.

Operation calls exhibit answer-type polymorphism in the sense that they prescribe only the type that the continuation expects, but not its return type. Equations between operations have to be similarly polymorphic, so we describe them with a pair of *templates* T (Plotkin & Pretnar, 2013). A template pair is then instantiated to a pair of computations by replacing all template variables with appropriate function values. In particular, given a function value f_j for each free template variable z_j appearing in a template T, we recursively define the computation $T[f_j/z_j]_j$ by:

$$(z_j\ v)[f_j/z_j]_j = f_j\ v$$

$$(\texttt{if}\ v\ \texttt{then}\ T_1\ \texttt{else}\ T_2)[f_j/z_j]_j = \texttt{if}\ v\ \texttt{then}\ T_1[f_j/z_j]_j\ \texttt{else}\ T_2[f_j/z_j]_j$$

$$(op(v;\ y.T))[f_j/z_j]_j = op(v;\ y.T[f_j/z_j]_j)$$

Since templates have to be polymorphic in the result type, we have to restrict their constructors to a small answer-type polymorphic subset, though one which proves to be sufficient for many interesting and common effect theories.

3.4 Type checking

It should come as no surprise that the proposed type system is intricate. Types contain templates, which then further contain terms and types, and to top it all off, both terms and types are split into mutually dependent value and computation sorts. This forces us to mutually define judgements for:

- $\Gamma \vdash v : A$, which states that in context Γ the value v has a type A,
- $\Gamma \vdash c : \underline{C}$, which states that in context Γ the computation c has a computation type $\underline{C}$,
- $\Gamma \vdash h : \Sigma \Rightarrow \underline{D}$, which states that in context Γ clauses h cover operations listed in Σ using computations of type $\underline{D}$,
- $\Gamma \vdash h : \Sigma \Rightarrow \underline{D}\ \mathsf{respects}\ \mathcal{E}$, which states that in context Γ and with respect to the signature Σ, clauses h are well defined, meaning they handle computations equivalent under $\mathcal{E}$ into equivalent computations of type $\underline{D}$,
- $\vdash A : \mathsf{vtype}$, which states that the value type A is well formed,
- $\vdash \underline{C} : \mathsf{ctype}$, which states that the computation type $\underline{C}$ is well formed,
- $\vdash \Sigma : \mathsf{sig}$, which states that the signature Σ is well formed,
- $\vdash \Gamma : \mathsf{ctx}$, which states that the value context Γ is well formed,
- $\vdash \mathrm{Z} : \mathsf{tctx}$, which states that the template context Z is well formed,
- $\Gamma; \mathrm{Z} \vdash T : \Sigma$, which states that in contexts Γ and Z, the template T is well formed with respect to the signature Σ,
- $\vdash \mathcal{E} : \Sigma$, which states that equations $\mathcal{E}$ are well formed with respect to the signature Σ.

Even though the forthcoming rules have to be treated as a single definition, we structure them into smaller, more digestible chunks.

First, Figure 4 lists the usual typing rules for values and computations. Aside from the decoupling of operation clauses, most of the rules closely follow our previous work (Bauer & Pretnar, 2014; Pretnar, 2015; Kammar & Pretnar, 2017). The main difference is the addition of equations, which are in all but one case simply tacked onto the computation type. For example in sequencing $\texttt{do}\ x \leftarrow c_1\ \texttt{in}\ c_2$, we require that c_1 and c_2 have not only matching signatures, but also equations.

The rule for typing handlers is more interesting. If a handler is given a type $A!\Sigma/\mathcal{E} \Rightarrow \underline{D}$, we must first check that the return clause maps values of type A to computations of type $\underline{D}$. Next, all operations in Σ must have appropriate operation clauses that handle them with

Well-typed values $\Gamma \vdash v : A$ *(where* $\vdash \Gamma : \mathsf{ctx}$ *and* $\vdash A : \mathsf{vtype}$*)*

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}$$

$$\frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathtt{fun}\ x \mapsto c : A \to \underline{C}} \qquad \frac{\Gamma, x : A \vdash c_r : \underline{D} \qquad \Gamma \vdash h : \Sigma \Rrightarrow \underline{D}\ \mathsf{respects}\ \mathcal{E}}{\Gamma \vdash \mathtt{handler}\ (\mathtt{ret}\ x \mapsto c_r ; h) : A!\Sigma/\mathcal{E} \Rightarrow \underline{D}}$$

Well-typed computations $\Gamma \vdash c : \underline{C}$ *(where* $\vdash \Gamma : \mathsf{ctx}$ *and* $\vdash \underline{C} : \mathsf{ctype}$*)*

$$\frac{\Gamma \vdash v : \mathtt{bool} \qquad \Gamma \vdash c_1 : \underline{C} \qquad \Gamma \vdash c_2 : \underline{C}}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \underline{C}} \qquad \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\ v_2 : \underline{C}} \qquad \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathtt{ret}\ v : A!\Sigma/\mathcal{E}}$$

$$\frac{(op : A_{op} \to B_{op}) \in \Sigma \qquad \Gamma \vdash v : A_{op} \qquad \Gamma, y : B_{op} \vdash c : A!\Sigma/\mathcal{E}}{\Gamma \vdash op(v; y.c) : A!\Sigma/\mathcal{E}}$$

$$\frac{\Gamma \vdash c_1 : A!\Sigma/\mathcal{E} \qquad \Gamma, x : A \vdash c_2 : B!\Sigma/\mathcal{E}}{\Gamma \vdash \mathtt{do}\ x \leftarrow c_1\ \mathtt{in}\ c_2 : B!\Sigma/\mathcal{E}} \qquad \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathtt{with}\ v\ \mathtt{handle}\ c : \underline{D}}$$

Fig. 4. Typing judgements for terms.

Well-typed operation clauses $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}$ *(where* $\vdash \Gamma : \mathsf{ctx}$, $\vdash \Sigma : \mathsf{sig}$ *and* $\vdash \underline{D} : \mathsf{ctype}$*)*

$$\frac{}{\Gamma \vdash \emptyset : \emptyset \Rrightarrow \underline{D}} \qquad \frac{\Gamma \vdash h : \Sigma \Rrightarrow \underline{D} \qquad \Gamma, x : A_{op}, k : B_{op} \to \underline{D} \vdash c_{op} : \underline{D} \qquad op \notin \Sigma}{\Gamma \vdash h \cup \{op(x;k) \mapsto c_{op}\} : (\Sigma \cup \{op : A_{op} \to B_{op}\}) \Rrightarrow \underline{D}}$$

Well-defined operation clauses $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}\ \mathsf{respects}\ \mathcal{E}$ *(where* $\vdash \mathcal{E} : \Sigma$ *and* $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}$*)*

Given in Section 3.

Fig. 5. Typing judgements for operation clauses.

computations of type $\underline{D}$, well defined with respect to equations $\mathcal{E}$. All of this is captured by the judgement $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}$ respects $\mathcal{E}$, given in Figure 5. There are different choices one can consider in defining this judgement, though all impose the same typing rules, captured by the auxiliary judgement $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}$. We present a few interesting choices of resulting logics $\mathcal{L}$ in Section 4.

Rules that ensure well-formedness of types, signatures, and contexts are given in Figure 6 and are routine. The most interesting one is the rule for the computation type $A!\Sigma/\mathcal{E}$, which requires the theory $\mathcal{E}$ to be well formed with respect to the signature Σ.

We treat templates and equations in Figure 7. The rules for templates follow the corresponding rules for computations, while equations are well formed with respect to the signature Σ if all their templates are well formed. Template variables in template contexts Z

Well-formed value types $\vdash A : \mathsf{vtype}$

$$\frac{}{\vdash \mathtt{unit} : \mathsf{vtype}} \qquad \frac{}{\vdash \mathtt{bool} : \mathsf{vtype}} \qquad \frac{\vdash A : \mathsf{vtype} \qquad \vdash \underline{C} : \mathsf{ctype}}{\vdash A \to \underline{C} : \mathsf{vtype}} \qquad \frac{\vdash \underline{C} : \mathsf{ctype} \qquad \vdash \underline{D} : \mathsf{ctype}}{\vdash \underline{C} \Rightarrow \underline{D} : \mathsf{vtype}}$$

Well-formed computation types $\vdash \underline{C} : \mathsf{ctype}$

$$\frac{\vdash A : \mathsf{vtype} \qquad \vdash \Sigma : \mathsf{sig} \qquad \vdash \mathcal{E} : \Sigma}{\vdash A!\Sigma/\mathcal{E} : \mathsf{ctype}}$$

Well-formed signatures $\vdash \Sigma : \mathsf{sig}$

$$\frac{}{\vdash \emptyset : \mathsf{sig}} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \vdash A : \mathsf{vtype} \qquad \vdash B : \mathsf{vtype} \qquad op \notin \Sigma}{\vdash \Sigma \cup \{op : A \to B\} : \mathsf{sig}}$$

Well-formed context $\vdash \Gamma : \mathsf{ctx}$ *and template context* $\vdash \mathrm{Z} : \mathsf{tctx}$

$$\frac{}{\vdash \emptyset : \mathsf{ctx}} \qquad \frac{\vdash \Gamma : \mathsf{ctx} \qquad x \notin \Gamma \qquad \vdash A : \mathsf{vtype}}{\vdash \Gamma, x : A : \mathsf{ctx}} \qquad \frac{}{\vdash \emptyset : \mathsf{tctx}}$$

$$\frac{\vdash \mathrm{Z} : \mathsf{tctx} \qquad z \notin \mathrm{Z} \qquad \vdash A : \mathsf{vtype}}{\vdash \mathrm{Z}, z : A \to * : \mathsf{tctx}}$$

Fig. 6. Well-formedness judgements for types.

are labelled with the type $A \to *$ as they can be replaced with functions of type $A \to \underline{C}$ for an arbitrary computation type $\underline{C}$.

Theorem 3.1 (Safety).
Progress. *If* $\vdash c : A!\Sigma/\mathcal{E}$*, then either*

- *there exists a computation* c' *such that* $c \rightsquigarrow c'$*, or*
- c *is of the form* `ret` v *for some value* v*, or*
- c *is of the form* $op(v;\ k)$ *for some* $op \in \Sigma$.

Preservation. If $\vdash c : A!\Sigma/\mathcal{E}$ and $c \rightsquigarrow c'$, then $\vdash c' : A!\Sigma/\mathcal{E}$.

Proof Both parts can be shown with a routine structural induction. □

4 Logics

We now consider different possibilities for the definition of $\Gamma \vdash h : \Sigma \Rightarrow \underline{D}$ respects $\mathcal{E}$. The trivial choice is to take the empty relation, resulting in a logic $\mathcal{L}_\emptyset$ in which no handlers can be typed. At the other extreme, we can take the logic $\mathcal{L}_{\text{full}}$ with the unwieldy set of

Well-typed templates $\Gamma; \mathrm{Z} \vdash T : \Sigma$ *(where* $\vdash \Gamma : \mathsf{ctx}$, $\vdash \mathrm{Z} : \mathsf{tctx}$, *and* $\vdash \Sigma : \mathsf{sig}$*)*

$$\frac{\Gamma \vdash v : A \qquad (z : A \to *) \in \mathrm{Z}}{\Gamma; \mathrm{Z} \vdash z\, v : \Sigma} \qquad \frac{\Gamma \vdash v : \mathtt{bool} \qquad \Gamma; \mathrm{Z} \vdash T_1 : \Sigma \qquad \Gamma; \mathrm{Z} \vdash T_2 : \Sigma}{\Gamma; \mathrm{Z} \vdash \mathtt{if}\ v\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 : \Sigma}$$

$$\frac{(op : A \to B) \in \Sigma \qquad \Gamma \vdash v : A \qquad \Gamma, y : B; \mathrm{Z} \vdash T : \Sigma}{\Gamma; \mathrm{Z} \vdash op(v; y.T) : \Sigma}$$

Well-formed theories $\vdash \mathcal{E} : \Sigma$ *(where* $\vdash \Sigma : \mathsf{sig}$*)*

$$\frac{}{\vdash \emptyset : \Sigma} \qquad \frac{\vdash \mathcal{E} : \Sigma \qquad \Gamma; \mathrm{Z} \vdash T_1 : \Sigma \qquad \Gamma; \mathrm{Z} \vdash T_2 : \Sigma}{\vdash \mathcal{E} \cup \{\Gamma; \mathrm{Z} \vdash T_1 \sim T_2\} : \Sigma}$$

Fig. 7. Typing judgements for templates.

all possible combinations of $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}$ and $\vdash \mathcal{E} : \Sigma$ that yield a well-defined denotation (cf. Definition 6.4). In between, there are a few interesting logics on which we focus.

4.1 Free logic $\mathcal{L}_{free}$

The simplest one of these is the logic $\mathcal{L}_{\mathrm{free}}$ in which well-typed operation clauses respect only the empty set of equations:

$$\frac{\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}}{\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}\ \mathsf{respects}\ \emptyset}$$

This corresponds to the conventional approach to handlers in which we ignore equations and accept any well-typed handler. Note that we could replace $\emptyset$ with an arbitrary set of tautologies such as $\Gamma; \mathrm{Z} \vdash T \sim T$. The same principle applies in general, as we may replace any set of equations with an equivalent one.

4.2 Equational logic $\mathcal{L}_{eq}$

We next consider a simple equational logic $\mathcal{L}_{\mathrm{eq}}$, which allows us to prove that operation clauses indeed respect a given theory. In addition to the type rules, which are the same as in Section 3.4, $\mathcal{L}_{\mathrm{eq}}$ includes the $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}\ \mathsf{respects}\ \mathcal{E}$ relation and typed equational judgements $\Gamma \vdash v_1 \equiv_A v_2$ for values and $\Gamma \vdash c_1 \equiv_{\underline{C}} c_2$ for computations. Most of these additional rules are well known: reflexivity, symmetry, transitivity, substitution, congruences for each construct, and $\beta\eta$-equivalences. More interesting are the rules given in Figure 8.

The first rule allows us to use equations we have in computation types. Any instantiation of templates $T_1 \sim T_2$ in $\mathcal{E}$ produces an equivalence between computations at any type $A!\Sigma/\mathcal{E}$. This rule is a generalisation of the rule present in the original logic for algebraic effects (Plotkin & Pretnar, 2008), except that the effect theory is local rather than global.

When instantiating the two templates in an equation, we need to ensure that the final terms are of equal types. This is ensured by the following lemma.

Inheriting equations from the effect theory

$$\frac{\big((x_i : A_i)_i; (z_j : B_j \to *)_j \vdash T_1 \sim T_2\big) \in \mathcal{E} \qquad \Gamma \vdash v_i : A_i \qquad \Gamma \vdash f_j : B_j \to A!\Sigma/\mathcal{E}}{\Gamma \vdash (T_1[f_j/z_j]_j)[v_i/x_i]_i \equiv_{A!\Sigma/\mathcal{E}} (T_2[f_j/z_j]_j)[v_i/x_i]_i}$$

Checking when a handler respects an equational theory

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow \underline{D}}{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \emptyset}$$

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \mathcal{E} \qquad \Gamma, (x_i : A_i)_i, (f_j : B_j \to \underline{D})_j \vdash T_1^h[f_j/z_j]_j \equiv_{\underline{D}} T_2^h[f_j/z_j]_j}{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \mathcal{E} \cup \big\{(x_i : A_i)_i; (z_j : B_j \to *)_j \vdash T_1 \sim T_2\big\}}$$

where for $h = \{op(x;k) \mapsto c_{op}\}_{op}$ we define:

$$z_i(v)^h[f_j/z_j]_j = f_i\ v$$
$$(\texttt{if } v \texttt{ then } T_1 \texttt{ else } T_2)^h[f_j/z_j]_j = \texttt{if } v \texttt{ then } T_1^h[f_j/z_j]_j \texttt{ else } T_2^h[f_j/z_j]_j$$
$$op(v; y.T)^h[f_j/z_j]_j = c_{op}[v/x, (\texttt{fun } y \mapsto T^h[f_j/z_j]_j)/k]$$

Fig. 8. Non-standard rules of the logic $\mathcal{L}_{\text{eq}}$.

Lemma 4.1. *Suppose, we have a well-typed template* $(x_i : A_i)_i; (z_j : B_j \to *)_j \vdash T : \Sigma$ *and values* $\Gamma \vdash v_i : A_i$ *for each* i *and* $\Gamma \vdash f_j : B_j \to \underline{C}$ *for each* j*. Then, we have* $\Gamma \vdash (T[f_j/z_j]_j)[v_i/x_i]_i : \underline{C}$.

The last two rules describe when a handler respects an effect theory and are similar to the rules present in the original treatment of handlers (Plotkin & Pretnar, 2009, 2013), but adapted to local effect theories. The first of the two rules treats the empty theory as before and the second allows us to extend the theory with a single equation between templates. Lemma 4.1 again guarantees that equations in the hypotheses are well typed.

To show how rules of $\mathcal{L}_{\text{eq}}$ can be used in practice, let us return to the running examples from Section 2.

Example 4.2. Recall the definition of the handler:

$$\begin{aligned} &\mathit{pickLeft} = \texttt{handler}\ \{ \\ &\quad \mid \mathit{choose}((\,); k) \mapsto k\ \texttt{true} \\ &\} \end{aligned}$$

We wish to show that it has the type:

$$\mathit{pickLeft} : \texttt{int}!\{\mathit{choose}\}/\{(\text{IDEM}), (\text{ASSOC})\} \Rightarrow \texttt{int}!\emptyset/\emptyset$$

We first focus on the Equation (IDEM):

$$z : \mathtt{unit} \to * \vdash \mathit{choose}((\,);\ y.\mathtt{if}\ y\ \mathtt{then}\ z\ (\,)\ \mathtt{else}\ z\ (\,)) \sim z\ (\,)$$

To show that *pickLeft* respects (IDEM), we must prove that for any $f : \mathtt{unit} \to \mathtt{int}!\emptyset/\emptyset$ (recall this type is obtained by instantiating $*$ with the right-hand side of the handler type) it holds that:

$$(\mathit{choose}((\,);\ y.\mathtt{if}\ y\ \mathtt{then}\ z\ (\,)\ \mathtt{else}\ z\ (\,)))^{\mathit{pickLeft}}[f/z] \equiv_{\mathtt{int}!\emptyset/\emptyset} (z\ (\,))^{\mathit{yieldAll}}[f/z].$$

We rewrite both sides according to definition of $(\_)^{\mathit{pickLeft}}[f/z]$ to the formula:

$$(\mathtt{fun}\ y \mapsto \mathtt{if}\ y\ \mathtt{then}\ f\ (\,)\ \mathtt{else}\ f\ (\,))\ \mathtt{true} \equiv_{\mathtt{int}!\emptyset/\emptyset} f\ (\,)$$

Using β-laws, the left side first simplifies to

$$\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ f\ (\,)\ \mathtt{else}\ f\ (\,)$$

and then further to $f\ (\,)$ which concludes the proof.

The proof for (ASSOC), while requiring more space to write out, is no more difficult. For the left side, we rewrite $(z_1 \oplus (z_2 \oplus z_3))^{\mathit{pickLeft}}[f_1/z_1, f_2/z_2, f_3/z_3]$ to

$$(\mathtt{fun}\ y \mapsto \mathtt{if}\ y\ \mathtt{then}\ f_1\ (\,)\ \mathtt{else}\ (\mathtt{fun}\ y' \mapsto \mathtt{if}\ y'\ \mathtt{then}\ f_2\ (\,)\ \mathtt{else}\ f_3\ (\,))\ \mathtt{true})\ \mathtt{true}$$

which can be simplified to $f_1\ (\,)$. The same process is repeated for the right side of the equation.

Example 4.3. Recall the definition of the handler:

$$\begin{aligned}
&\mathit{yieldAll}\ =\ \mathtt{handler}\ \{\\
&\quad |\ \mathit{choose}((\,);\ k) \mapsto k\ \mathtt{true};\ k\ \mathtt{false}\\
&\quad |\ \mathtt{ret}\ x \mapsto \mathit{yield}(x;\ \_.\mathtt{ret}\ (\,))\\
&\}
\end{aligned}$$

We wish to show that it can be given the type:

$$\mathit{yieldAll} : \mathtt{int}!\{\mathit{choose}\}/\{(\text{COMM})\} \Rightarrow \mathtt{unit}!\{\mathit{yield}\}/\{(\text{YIELDORDER})\},$$

where the Equations (COMM) and (YIELDORDER) are

$$\begin{aligned}
&z_1 : \mathtt{unit} \to *, z_2 : \mathtt{unit} \to * \vdash\\
&\mathit{choose}((\,);\ y.\mathtt{if}\ y\ \mathtt{then}\ z_1\ (\,)\ \mathtt{else}\ z_2\ (\,)) \sim \mathit{choose}((\,);\ y.\mathtt{if}\ y\ \mathtt{then}\ z_2\ (\,)\ \mathtt{else}\ z_1\ (\,))
\end{aligned}$$

and

$$x : \mathtt{int}, y : \mathtt{int}; z : \mathtt{unit} \to * \vdash \mathit{yield}(x;\ \_.\mathit{yield}(y;\ \_.z\ (\,))) \sim \mathit{yield}(y;\ \_.\mathit{yield}(x;\ \_.z\ (\,)))$$

respectively.

To show that *yieldAll* respects (COMM), it is enough that we prove that for any functions $f_1, f_2 : \mathtt{unit} \to \mathtt{unit}!\{\mathit{yield}\}/\{(\text{YIELDORDER})\}$ it holds that:

$$\begin{aligned}
&(\mathit{choose}((\,);\ y.\mathtt{if}\ y\ \mathtt{then}\ z_1\ (\,)\ \mathtt{else}\ z_2\ (\,)))^{\mathit{yieldAll}}[f_1/z_1, f_2/z_2]\\
&\qquad\qquad \equiv_{\mathtt{unit}!\{\mathit{yield}\}/\{(\text{YIELDORDER})\}}\\
&\qquad\qquad\qquad (\mathit{choose}((\,);\ y.\mathtt{if}\ y\ \mathtt{then}\ z_2\ (\,)\ \mathtt{else}\ z_1\ (\,)))^{\mathit{yieldAll}}[f_1/z_1, f_2/z_2].
\end{aligned}$$

Using the definition of *yieldAll*, the left-hand side can be rewritten to the sequence of computations:

$$(\texttt{fun}\, y \mapsto \texttt{if}\, y\, \texttt{then}\, f_1\ (\,)\ \texttt{else}\, f_2\ (\,))\ \texttt{true};$$
$$(\texttt{fun}\, y \mapsto \texttt{if}\, y\, \texttt{then}\, f_1\ (\,)\ \texttt{else}\, f_2\ (\,))\ \texttt{false}.$$

which is β-equivalent to $f_1\ (\,);\ f_2\ (\,)$. We repeat the process for the right-hand side to obtain the equation:

$$f_1\ (\,); f_2\ (\,) \equiv_{\texttt{unit!}\{yield\}/\{(\textsc{YieldOrder})\}} f_2\ (\,); f_1\ (\,).$$

At this step, we postpone the remainder of the proof to Example 4.4, as $\mathcal{L}_{\mathrm{eq}}$ is unfortunately not powerful enough to finish it. A crucial piece missing is the principle of computational induction (Plotkin & Pretnar, 2008; Bauer & Pretnar, 2014), which captures the inductive structure of computation types $A!\Sigma/\mathcal{E}$.

4.3 Predicate logic with induction $\mathcal{L}_{pred}$

In order to state induction in our logic, we need to extend our judgements with hypotheses and universal quantifiers. We now extend the logic to a first-order predicate logic $\mathcal{L}_{\mathrm{pred}}$. In addition to equations, the *formulae* φ include logical connectives and quantifiers over value types:

formulae φ, ψ ::=	$v_1 \equiv_A v_2$	value equation
	$\mid c_1 \equiv_{\underline{C}} c_2$	computation equation
	$\mid \top$	truth
	$\mid \bot$	falsity
	$\mid \varphi_1 \wedge \varphi_2$	conjunction
	$\mid \varphi_1 \vee \varphi_2$	disjunction
	$\mid \varphi \Rightarrow \psi$	implication
	$\mid \forall x : A.\ \varphi$	universal quantification
	$\mid \exists x : A.\ \varphi$	existential quantification

We type formulae in a context Γ and extend judgements with hypotheses to ones of the form $\Gamma \mid \Psi \vdash \varphi$, where Ψ is a set of formulae $\psi_1, \ldots, \psi_n$. In addition to the rules of $\mathcal{L}_{\mathrm{eq}}$ (extended with hypotheses), the logic $\mathcal{L}_{\mathrm{pred}}$ includes the standard rules for logical connectives and quantifiers, which we omit (cf. Plotkin & Pretnar, 2008; Pretnar, 2010), and a principle of induction, on which we focus now.

The principle of induction states that a property holds for all computations of type $A!\Sigma/\mathcal{E}$, if it holds for all computations that return a value of type A, and for all computations that call an operation $op \in \Sigma$ under the induction hypotheses that it holds for all possible continuations. For a schema producing a formula $\varphi(c)$ for any computation c, the induction is stated as:

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash c : A!\Sigma/\mathcal{E} \qquad \Gamma, x : A \mid \Psi \vdash \varphi(\texttt{ret}\ x) \\ \left[\Gamma, x : A_{op}, k : B_{op} \to A!\Sigma/\mathcal{E} \mid \Psi, (\forall y : B_{op}.\ \varphi(k\ y)) \vdash \varphi(op(x;\ y.k\ y))\right]_{op : A_{op} \to B_{op} \in \Sigma}\end{array}}{\Gamma \mid \Psi \vdash \varphi(c)}$$

Example 4.4. Using induction, we may finally complete the proof started in Example 4.3. Recall we were left at proving:

$$f_1\ ();f_2\ () \equiv_{\underline{D}} f_2\ ();f_1\ ().$$

where we abbreviate $\underline{D} = \texttt{unit}!\{\mathit{yield}\}/\{(\textsc{YieldOrder})\}$. We wish to show with induction that in $\underline{D}$ any two computations commute. To prove this, we first show that a single call of *yield* commutes with any computation in $\underline{D}$, so we take $\varphi_1(c_1)$ to be:

$$\mathit{yield}(x;\_.c_1);\ c_2 \equiv_{\underline{D}} c_1;\ \mathit{yield}(x;\_.c_2).$$

In the proof, we include hints on what rule we used (for instance, "β for ; and *op*" means we used the β-law dealing with sequencing and operations). We first prove the base case for $c_1 = \texttt{ret}\ ()$:

$$\begin{aligned}
&\mathit{yield}(x;\_.\texttt{ret}\ ());\ c_2 && \\
\equiv_{\underline{D}}\ &\mathit{yield}(x;\_.(\texttt{ret}\ ();\ c_2)) && (\beta \text{ for ; and } op)\\
\equiv_{\underline{D}}\ &\mathit{yield}(x;\_.c_2) && (\beta \text{ for ; and } \texttt{ret}\ )\\
\equiv_{\underline{D}}\ &\texttt{ret}\ ();\ \mathit{yield}(x;\_.c_2) && (\beta \text{ for ; and } \texttt{ret}\ \text{, other direction})
\end{aligned}$$

Next, we take $c_1 = \mathit{yield}(y;\_.k\ ())$ and prove the induction step using the hypothesis:

$$\mathit{yield}(x;\_.k\ ());\ c_2 \equiv_{\underline{D}} k\ ();\ \mathit{yield}(x;\_.c_2).$$

We proceed as:

$$\begin{aligned}
&\mathit{yield}(x;\_.\mathit{yield}(y;\_.k\ ()));\ c_2 && \\
\equiv_{\underline{D}}\ &\mathit{yield}(y;\_.\mathit{yield}(x;\_.k\ ()));\ c_2 && ((\textsc{YieldOrder}) \text{ holds in } \underline{D})\\
\equiv_{\underline{D}}\ &\mathit{yield}(y;\_.\mathit{yield}(x;\_.k\ ());\ c_2) && (\beta \text{ for ; and } op)\\
\equiv_{\underline{D}}\ &\mathit{yield}(y;\_.k\ ();\ \mathit{yield}(x;\_.c_2)) && (\text{induction hypothesis})\\
\equiv_{\underline{D}}\ &\mathit{yield}(y;\_.k\ ());\ \mathit{yield}(x;\_.c_2) && (\beta \text{ for ; and } op\text{, other direction})
\end{aligned}$$

We now show that any two computations in $\underline{D}$ commute. For that we take $\varphi_2(c_1)$ to be

$$c_1;\ c_2 \equiv_{\underline{D}} c_2;\ c_1.$$

We again first show the base case for $c_1 = \texttt{ret}\ ()$ using the β-equivalence for sequencing and return

$$\texttt{ret}\ ();\ c_2 \equiv_{\underline{D}} c_2 \equiv_{\underline{D}} c_2;\ \texttt{ret}\ ()$$

We then show the induction step for $c_1 = \mathit{yield}(y;\_.k\ ())$ with the hypothesis $k\ ();\ c_2 \equiv_{\underline{D}} c_2;\ k\ ()$:

$$\begin{aligned}
&\mathit{yield}(x;\_.k\ ());\ c_2 && \\
\equiv_{\underline{D}}\ &\mathit{yield}(x;\_.k\ ();\ c_2) && (\beta \text{ for ; and } op)\\
\equiv_{\underline{D}}\ &\mathit{yield}(x;\_.c_2;\ k\ ()) && (\text{induction hypothesis})\\
\equiv_{\underline{D}}\ &\mathit{yield}(x;\_.c_2);\ k\ () && (\beta \text{ for ; and } op\text{, other direction})\\
\equiv_{\underline{D}}\ &c_2;\ \mathit{yield}(x;\_.k\ ()) && (\varphi_1(k\ ()) \text{ from previous proof})
\end{aligned}$$

5 Denotation of types and terms

We have presented a type system in which well-formed types depend on well-typed terms and vice versa. To avoid circularity when defining the denotational semantics of such types and terms, we proceed in two stages. First, we define a denotation of types that is independent of effect theories. This allows us to further define the denotation of well-typed terms, which similarly does not take effect theories into an account. In Section 6, we then equip each type with an equivalence relation that stems from the effect theory and show that the term denotations are well defined with respect to it.

5.1 Semantics of types

To value types A (and computation types $\underline{C}$), we assign sets $[\![A]\!]$ (and $[\![\underline{C}]\!]$) as follows:

$$[\![\texttt{unit}]\!] = \{\star\} \qquad [\![\texttt{bool}]\!] = \{\mathrm{ff}, \mathrm{tt}\}$$

$$[\![A \to \underline{C}]\!] = [\![A]\!] \to [\![\underline{C}]\!] \qquad [\![\underline{C} \Rightarrow \underline{D}]\!] = [\![\underline{C}]\!] \to [\![\underline{D}]\!]$$

$$[\![A!\Sigma/\mathcal{E}]\!] = [\![\Sigma]\!][\![A]\!]$$

where for $\Sigma = \{op : A_{op} \to B_{op}\}_{op}$, we define $[\![\Sigma]\!]$ to be the free functor mapping a set X to the inductively defined set $[\![\Sigma]\!]X$ containing:

1. $\mathrm{in}_{\texttt{ret}}(a)$ for each a in X
2. $\mathrm{in}_{op}(a; \kappa)$ for each $op : A_{op} \to B_{op} \in \Sigma$, each $a \in [\![A_{op}]\!]$ and each $\kappa \in [\![B_{op}]\!] \to [\![\Sigma]\!]X$

Note that handlers are interpreted by ordinary functions and $\mathcal{E}$ does not play a role in the denotation of $A!\Sigma/\mathcal{E}$.

Next, an *interpretation* H of a signature Σ over a set Y is a family of functions H_{op} : $[\![A_{op}]\!] \times ([\![B_{op}]\!] \to Y) \to Y$ for each $op : A_{op} \to B_{op} \in \Sigma$. We define the set $\mathsf{interp}_\Sigma(Y)$ of all interpretations by:

$$\mathsf{interp}_\Sigma(Y) = \prod_{op : A_{op} \to B_{op} \in \Sigma} [\![A_{op}]\!] \times ([\![B_{op}]\!] \to Y) \to Y$$

For any signature Σ and set X, we define a *free interpretation* $F_{X,\Sigma} \in \mathsf{interp}_\Sigma([\![\Sigma]\!]X)$ by:

$$(F_{X,\Sigma})_{op}(a)(\kappa) = \mathrm{in}_{op}(a; \kappa)$$

Next, for any interpretation $H : \mathsf{interp}_\Sigma(Y)$, we can *lift* a function $f : X \to Y$ to a function $\mathsf{lift}_H f : [\![\Sigma]\!]X \to Y$, defined recursively by:

$$\mathsf{lift}_H f(\mathrm{in}_{\texttt{ret}}(x)) = f(x),$$
$$\mathsf{lift}_H f(\mathrm{in}_{op}(x; \kappa)) = H_{op}(x; \mathsf{lift}_H f \circ \kappa).$$

5.2 Semantics of well-typed values and computations

Well-typed terms

$$\Gamma \vdash v : A \qquad \text{and} \qquad \Gamma \vdash c : \underline{C}$$

are interpreted as maps

$$[\![\Gamma \vdash v : A]\!] : [\![\Gamma]\!] \to [\![A]\!] \qquad \text{and} \qquad [\![\Gamma \vdash c : \underline{C}]\!] : [\![\Gamma]\!] \to [\![\underline{C}]\!],$$

where Γ is defined component-wise:

$$[\![\varepsilon]\!] = \{\star\}$$
$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

The definition proceeds by recursion on the derivation of the typing judgement. When no confusion can arise, we abbreviate the denotations to $[\![v]\!]$ and $[\![c]\!]$.

Given an environment $\eta \in [\![\Gamma]\!]$, the rules for base values are

$$[\![\Gamma \vdash x_i : A_i]\!]\eta = \eta_i \qquad [\![\Gamma \vdash (\,) : \texttt{unit}]\!]\eta = \star$$
$$[\![\Gamma \vdash \texttt{false} : \texttt{bool}]\!]\eta = \mathrm{ff} \qquad [\![\Gamma \vdash \texttt{true} : \texttt{bool}]\!]\eta = \mathrm{tt}$$

while for functions, we have

$$[\![\Gamma \vdash (\texttt{fun}\, x \mapsto c) : A \to \underline{C}]\!]\eta = \lambda a \in [\![A]\!] \,.\, [\![\Gamma, x : A \vdash c : \underline{C}]\!](\eta, a)$$

In order to define the denotation of handlers, we must first treat operation clauses. A set of well-typed clauses $\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}$ is defined as a map:

$$[\![\Gamma \vdash h : \Sigma \Rrightarrow \underline{D}]\!] : [\![\Gamma]\!] \to \mathsf{interp}_\Sigma([\![\underline{D}]\!])$$

where

$$[\![\Gamma \vdash \{op(x;\ k) \mapsto c_{op}\}_{op} : \Sigma \Rrightarrow \underline{D}]\!]\eta =$$
$$\{\lambda a \in [\![A_{op}]\!] \,.\, \lambda \kappa \in [\![B_{op} \to \underline{D}]\!] \,.\, [\![\Gamma, x : A_{op}, k : B_{op} \to \underline{D} \vdash c_{op} : \underline{D}]\!](\eta, a, \kappa)\}_{op : A_{op} \to B_{op} \in \Sigma}$$

A denotation of a handler is just the lifting of its return clause to the interpretation given by the operation clauses:

$$[\![\Gamma \vdash \texttt{handler}\ (\texttt{ret}\ x \mapsto c_r;\, h) : A!\Sigma/\mathcal{E} \Rrightarrow \underline{D}]\!]\eta = \mathsf{lift}_{[\![h]\!]\eta}(\lambda a \in [\![A]\!] \,.\, [\![c_r]\!](\eta, a))$$

In contrast to the original denotational semantics of handlers (Plotkin & Pretnar, 2013), effect theories do not affect the denotation of types, so all handlers receive a denotation.

The denotation of computations is more or less structural:

$$[\![\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \underline{C}]\!]\eta = \begin{cases} [\![\Gamma \vdash c_1 : \underline{C}]\!]\eta & \text{if } [\![\Gamma \vdash v : \texttt{bool}]\!]\eta = \mathrm{tt}, \\ [\![\Gamma \vdash c_2 : \underline{C}]\!]\eta & \text{if } [\![\Gamma \vdash v : \texttt{bool}]\!]\eta = \mathrm{ff} \end{cases}$$

$$[\![\Gamma \vdash v_1\, v_2 : \underline{C}]\!]\eta = ([\![\Gamma \vdash v_1 : A \to \underline{C}]\!]\eta)([\![\Gamma \vdash v_2 : A]\!]\eta)$$

Returned values and operations of type $A!\Sigma/\mathcal{E}$ are interpreted by appropriate constructors of $[\![\Sigma]\!][\![A]\!]$:

$$[\![\Gamma \vdash \texttt{ret}\ v : A!\Sigma/\mathcal{E}]\!]\eta = \mathrm{in}_{\texttt{ret}}([\![\Gamma \vdash v : A]\!]\eta)$$

$$[\![\Gamma \vdash op(v;\ y.c) : A!\Sigma/\mathcal{E}]\!]\eta =$$
$$\mathrm{in}_{op}([\![\Gamma \vdash v : A_{op}]\!]\eta;\ \lambda b \in [\![B_{op}]\!] \,.\, [\![\Gamma, y : B_{op} \vdash c : A!\Sigma/\mathcal{E}]\!](\eta, b)).$$

The denotation of sequencing is obtained by lifting the continuation of the second sequent to the free interpretation and applying it to the first sequent:

$$[\![\Gamma \vdash \mathtt{do}\ x \leftarrow c_1\ \mathtt{in}\ c_2 : B!\Sigma/\mathcal{E}]\!]\eta =$$
$$\mathsf{lift}_{F_{[\![A]\!],\Sigma}}(\lambda a \in [\![A]\!]\ .\ [\![\Gamma, x : A \vdash c_2 : B!\Sigma/\mathcal{E}]\!](\eta, a))([\![\Gamma \vdash c_1 : A!\Sigma/\mathcal{E}]\!]\eta).$$

Finally, since handlers are ordinary functions, handling is just application:

$$[\![\Gamma \vdash \mathtt{with}\ v\ \mathtt{handle}\ c : \underline{D}]\!]\eta = ([\![\Gamma \vdash v : \underline{C} \Rightarrow \underline{D}]\!]\eta)([\![\Gamma \vdash c : \underline{C}]\!]\eta)$$

5.3 Relation to operational semantics

Before we proceed, we first ensure that the presented denotational semantics is sound with respect to the operational semantics.

Proposition 5.1. *If* $\vdash c : \underline{C}$ *and* $c \rightsquigarrow c'$, *then* $\vdash c' : \underline{C}$ *and* $[\![\vdash c : \underline{C}]\!] = [\![\vdash c' : \underline{C}]\!]$.

Proof The fact that c' has the same type follows from type preservation in Theorem 3.1, while the proof of the second part proceeds by an easy induction on the derivation of $c \rightsquigarrow c'$. □

As expected, our denotational semantics is also adequate with respect to the operational semantics.

Lemma 5.2 (Adequacy). *If* $[\![\vdash c : \mathtt{bool}!\emptyset/\emptyset]\!] = \mathsf{in}_{\mathtt{ret}}([\![\vdash v : \mathtt{bool}]\!])$ *then* $c \rightsquigarrow^* \mathtt{ret}\ v$.

Proof As our language features no recursion, Theorem 3.1 implies that there is some sequence of steps $c \rightsquigarrow^* \mathtt{ret}\ v'$. From Proposition 5.1, it follows that $[\![c]\!] = \mathsf{in}_{\mathtt{ret}}([\![v']\!])$, thus $[\![v]\!] = [\![v']\!]$. Because distinct boolean values receive distinct denotations, we have $v = v'$ so $c \rightsquigarrow^* \mathtt{ret}\ v$. □

If our language featured recursion, we would instead follow the adequacy proof presented in Bauer & Pretnar (2014), as our operational and denotational semantics are a simplification (we omit instances and subtyping) of the ones given there.

We define a *computation context* $\mathbb{C}$ as a computation with a number of holes $\langle\ \rangle$ (possibly under binders) into which one may plug a computation c to obtain a computation $\mathbb{C}\langle c\rangle$. A context $\mathbb{C}$ is a *ground computation context for* Γ *and* $\underline{C}$ if for all computations $\Gamma \vdash c : \underline{C}$, we have $\vdash \mathbb{C}\langle c\rangle : \mathtt{bool}!\emptyset/\emptyset$.

Computations $\Gamma \vdash c : \underline{C}$ and $\Gamma \vdash c' : \underline{C}$ are *contextually equivalent*, written as $\Gamma \vdash c \cong c' : \underline{C}$ if for all ground computation contexts $\mathbb{C}$ for Γ and $\underline{C}$ we have $\mathbb{C}\langle c\rangle \rightsquigarrow^* \mathtt{ret\ true}$ if and only if $\mathbb{C}\langle c'\rangle \rightsquigarrow^* \mathtt{ret\ true}$. We similarly define $\Gamma \vdash v \cong v' : A$ for values.

Corollary 5.3. *If* $[\![\Gamma \vdash c : \underline{C}]\!] = [\![\Gamma \vdash c' : \underline{C}]\!]$, *then* $\Gamma \vdash c \cong c' : \underline{C}$. *If* $[\![\Gamma \vdash v : A]\!] = [\![\Gamma \vdash v' : A]\!]$, *then* $\Gamma \vdash v \cong v' : A$.

Proof The proof is a folklore application of adequacy. Assume that $\mathbb{C}\langle c\rangle \rightsquigarrow^* \texttt{ret true}$. By Proposition 5.1, $[\![\mathbb{C}\langle c\rangle]\!] = \mathrm{in}_{\texttt{ret}}(\mathrm{tt})$. However, denotational semantics is structural, thus $[\![\mathbb{C}\langle c\rangle]\!] = [\![\mathbb{C}\langle c'\rangle]\!]$, which by Lemma 5.2 implies $\mathbb{C}\langle c'\rangle \rightsquigarrow^* \texttt{ret true}$. The proof for values is identical. □

6 Denotation of effect theories and logics

Having provided a sound denotational semantics that disregards effect theories, we proceed by equipping it with relations that reflect the theories.

6.1 *Semantics of templates*

We first turn our attention to templates, which are the basic building blocks of equations. As template variable contexts $\mathtt{Z}$ are polymorphic in the type of computations, we interpret them as functors $[\![\mathtt{Z}]\!]$, defined by:

$$[\![\varepsilon]\!]Y = \{\star\}$$
$$[\![\mathtt{Z}, z : A \to *]\!]Y = [\![\mathtt{Z}]\!]Y \times Y^{[\![A]\!]}$$

We overload the notation and write $[\![\Gamma]\!]$ for the constant functor that maps any set Y to the set $[\![\Gamma]\!]$ as defined in Section 5.2,

An interpretation $H : \mathsf{interp}_\Sigma(Y)$ can interpret operations of Σ as well as any template using them. For a well-typed template $\Gamma; \mathtt{Z} \vdash T : \Sigma$, we recursively define $[\![\Gamma; \mathtt{Z} \vdash T : \Sigma]\!]^H : ([\![\Gamma]\!] \times [\![\mathtt{Z}]\!])Y \to Y$ (often abbreviated to $[\![T]\!]^H$) as:

$$[\![\Gamma; \mathtt{Z} \vdash z_i(v) : \Sigma]\!]^H(\eta; \zeta) = \zeta_i([\![v]\!]\eta)$$
$$[\![\Gamma; \mathtt{Z} \vdash \texttt{if } v \texttt{ then } T_1 \texttt{ else } T_2 : \Sigma]\!]^H(\eta; \zeta) = \begin{cases} [\![T_1]\!]^H(\eta; \zeta) & ;\ \text{if } [\![v]\!]\eta = \mathrm{tt} \\ [\![T_2]\!]^H(\eta; \zeta) & ;\ \text{if } [\![v]\!]\eta = \mathrm{ff} \end{cases}$$
$$[\![\Gamma; \mathtt{Z} \vdash op(v;\ y.T) : \Sigma]\!]^H(\eta; \zeta) = H_{op}([\![v]\!]\eta, \lambda b \in [\![B_{op}]\!]\,.\,[\![T]\!]^H(\eta, b; \zeta))$$

Lemma 6.1. *Take any sets X, Y, any interpretation $H : \mathsf{interp}_\Sigma(Y)$, any function $f : X \to Y$, and define $\varphi = \mathsf{lift}_H f : [\![\Sigma]\!]X \to Y$. Then, the following diagram commutes*

$$\begin{tikzcd}
([\![\Gamma]\!] \times [\![\mathtt{Z}]\!])([\![\Sigma]\!]X) \arrow[r, "([\![\Gamma]\!]\times[\![\mathtt{Z}]\!])\varphi"] \arrow[d, "[\![T]\!]^{F_{X,\Sigma}}"] & ([\![\Gamma]\!] \times [\![\mathtt{Z}]\!])Y \arrow[d, "[\![T]\!]^H"] \\
[\![\Sigma]\!]X \arrow[r, "\varphi"] & Y
\end{tikzcd}$$

In fact, all functions $\varphi : [\![\Sigma]\!]X \to Y$ for which the diagram commutes are lifts of some function $f : X \to Y$.

From Lemma 6.1, it follows that the free interpretation $F_{X,\Sigma}$ of a template $\Gamma; \mathtt{Z} \vdash T : \Sigma$ yields a natural transformation $\alpha : ([\![\Gamma]\!] \times [\![\mathtt{Z}]\!]) \circ [\![\Sigma]\!] \Rightarrow [\![\Sigma]\!]$, given by:

$$\alpha_X = [\![\Gamma; \mathtt{Z} \vdash T : \Sigma]\!]^{F_{X,\Sigma}}$$

6.2 Semantics of theories

With denotation of templates in place, we can focus on effect theories. We remedy the absence of theories in the denotations $[\![A]\!]$ and $[\![\underline{C}]\!]$ by defining a family of relations $\sim_A$ on $[\![A]\!]$ and $\sim_{\underline{C}}$ on $[\![\underline{C}]\!]$. We also extend relations to contexts, and for $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we define the relation $\sim_\Gamma$ on $[\![\Gamma]\!]$ by:

$$(a_1, \ldots, a_n) \sim_\Gamma (a'_1, \ldots, a'_n) \iff a_1 \sim_{A_1} a'_1 \wedge \ldots \wedge a_n \sim_{A_n} a'_n$$

For value types, the relations are defined by:

- The relations on $[\![\mathtt{bool}]\!]$ and $[\![\mathtt{unit}]\!]$ are identities.
- For a function type $A \to \underline{C}$ and $f, f' : [\![A \to \underline{C}]\!]$, we define

$$f \sim_{A\to\underline{C}} f' \iff (\forall a, a' \in [\![A]\!].\ a \sim_A a' \implies f(a) \sim_{\underline{C}} f'(a'))$$

- For a handler type $\underline{C} \Rightarrow \underline{D}$ and $h, h' : [\![\underline{C} \Rightarrow \underline{D}]\!]$, we define

$$h \sim_{\underline{C}\Rightarrow\underline{D}} h' \iff (\forall t, t' \in [\![\underline{C}]\!].\ t \sim_{\underline{C}} t' \implies h(t) \sim_{\underline{D}} h'(t'))$$

For the computation type $\underline{C} = A!\Sigma/\mathcal{E}$, we define the relation $\sim_{\underline{C}}$ on the set $[\![A!\Sigma/\mathcal{E}]\!]$ to be the smallest transitive and symmetric relation closed under the following rules:

- If $a \sim_A a'$, then $\mathsf{in}_{\mathtt{ret}}(a) \sim_{A!\Sigma/\mathcal{E}} \mathsf{in}_{\mathtt{ret}}(a')$.
- For any operation $op : A_{op} \to B_{op} \in \Sigma$, if $a \sim_{A_{op}} a'$ and $f \sim_{B_{op}\to\underline{C}} f'$ then also:

$$\mathsf{in}_{op}(a; f) \sim_{\underline{C}} \mathsf{in}_{op}(a'; f').$$

- For all equations $\Gamma; \mathrm{Z} \vdash T_1 \sim T_2$ in $\mathcal{E}$, where $\Gamma = (x_i : A_i)_i$ and $\mathrm{Z} = (z_j : B_j \to *)_j$, we say that if all $a_i \sim_{A_i} a'_i$ and all $f_j \sim_{B_j\to\underline{C}} f'_j$ then:

$$[\![\Gamma; \mathrm{Z} \vdash T_1 : \Sigma]\!]^{F_{[\![A]\!],\Sigma}}((a_i)_i, (f_j)_j) \sim_{\underline{C}} [\![\Gamma; \mathrm{Z} \vdash T_2 : \Sigma]\!]^{F_{[\![A]\!],\Sigma}}((a'_i)_i, (f'_j)_j)$$

The last rule ensures that all equations in the effect theory $\mathcal{E}$ are reflected in the relation $\sim_{\underline{C}}$, while all other definitions just propagate them structurally.

Lemma 6.2. *Let $\underline{C} = A!\Sigma/\mathcal{E}$ and $\underline{D} = B!\Sigma/\mathcal{E}$. If $g \sim_{A\to\underline{D}} g'$, then for every $t \sim_{\underline{C}} t'$ it holds that:*

$$(\mathsf{lift}_{F_{[\![A]\!],\Sigma}} g)(t) \sim_{\underline{D}} (\mathsf{lift}_{F_{[\![A]\!],\Sigma}} g')(t')$$

Proof We shorten $\mathsf{lift}_{F_{[\![A]\!],\Sigma}}$ to lift and $[\![\Gamma; \mathrm{Z} \vdash T : \Sigma]\!]^{F_{[\![A]\!],\Sigma}}$ to $[\![T]\!]$ for clarity. Since $\sim_{\underline{C}}$ is defined to be the smallest relation closed under the given rules, we can proceed by induction on $t \sim_{\underline{C}} t'$:

1. First, consider $\mathsf{in}_{\mathtt{ret}}(a) \sim_{\underline{C}} \mathsf{in}_{\mathtt{ret}}(a')$ where $a \sim_A a'$. By definition of lift, we obtain $(\mathsf{lift}\, g)(\mathsf{in}_{\mathtt{ret}}(a)) = g(a)$ and $(\mathsf{lift}\, g')(\mathsf{in}_{\mathtt{ret}}(a')) = g'(a')$. Because $g \sim_{A\to\underline{D}} g'$, we know that the functions map related arguments to related images.
2. Next, take $\mathsf{in}_{op}(a; f) \sim_{\underline{C}} \mathsf{in}_{op}(a'; f')$ for $op : A_{op} \to B_{op} \in \Sigma$ where $a \sim_{A_{op}} a'$ and $f \sim_{B_{op}\to\underline{C}} f'$. We know that $(\mathsf{lift}\, g)(\mathsf{in}_{op}(a; f)) = \mathsf{in}_{op}(a, \mathsf{lift}\, g \circ f)$ (and similarly for g'). By induction, $\mathsf{lift}\, g \circ f$ is related to $\mathsf{lift}\, g' \circ f'$, and so we obtain the desired result that $\mathsf{in}_{op}(a, \mathsf{lift}\, g \circ f) \sim_{\underline{D}} \mathsf{in}_{op}(a', \mathsf{lift}\, g' \circ f')$.

3. Finally, we have the interesting case where $t \sim_{\underline{C}} t'$ comes from an instantiation of an equation $\Gamma; \mathrm{Z} \vdash T_1 \sim T_2 \in \mathcal{E}$, where we denote $\Gamma = (x_i : A_i)_i$ and $\mathrm{Z} = (z_j : B_j \to *)_j$. That means that there exist $a_i \sim_{A_i} a'_i$ and $f_j \sim_{B_j \to \underline{C}} f'_j$ for which
$$t = [\![T_1]\!]((a_i)_i, (f_j)_j) \sim t' = [\![T_2]\!]((a'_i)_i, (f'_j)_j).$$
From Lemma 6.1, we have $(\mathsf{lift} g)([\![T_1]\!]((a_i)_i, (f_j)_j)) = [\![T_1]\!]((a_i)_i, (\mathsf{lift} g \circ f_j)_j)$ and similarly for T_2. By induction hypotheses, we get $\mathsf{lift} g \circ f_j \sim \mathsf{lift} g' \circ f'_j$ as before, thus:
$$[\![T_1]\!]((a_i)_i, (\mathsf{lift} g \circ f_j)_j) \sim [\![T_2]\!]((a'_i)_i, (\mathsf{lift} g' \circ f'_j)_j).$$
concluding the proof.

□

We can easily adapt Lemma 5.2 to consider equivalent computations, with $\sim_{\texttt{bool}!\emptyset/\emptyset}$ being the identity relation.

Lemma 6.3 (Adequacy). *If* $[\![\vdash c : \texttt{bool}!\emptyset/\emptyset]\!] \sim \mathsf{in}_{\texttt{ret}}([\![\vdash v : \texttt{bool}]\!])$*, then* $c \leadsto^* \texttt{ret}\ v$.

Generalising Corollary 5.3 is trickier, however. Recall that the typing relation depends on the information about which handlers respect which effect theories. If we state that all handlers respect all theories, we can quickly produce a counterexample of a handler that maps equivalent computations into non-equivalent ones. In order for our relations to make sense, the reasoning logic $\mathcal{L}$ needs to respect effect theories.

6.3 Soundness of a logic

Definition 6.4. *A logic* $\mathcal{L}$ *is* sound *if* $(\Gamma \vdash h : \Sigma \Rightarrow \underline{D}\ \textsf{respects}\ \mathcal{E})$ *implies that for any:*
$$(x_i : A_i)_i; (z_j : B_j \to *)_j \vdash T_1 \sim T_2 \in \mathcal{E},$$
any $\eta \sim_\Gamma \eta'$*, any* $a_i \sim_{A_i} a'_i$*, and any* $f_j \sim_{B_j \to \underline{D}} f'_j$*, we have*
$$[\![\Gamma; \mathrm{Z} \vdash T_1 : \Sigma]\!]^{[\![h]\!]\eta}((a_i)_i; (f_j)_j) \sim_{\underline{D}} [\![\Gamma; \mathrm{Z} \vdash T_2 : \Sigma]\!]^{[\![h]\!]\eta'}((a'_i)_i; (f'_j)_j)$$

Not every logic is sound. For example, take any logic that contains
$$\vdash h_{pickLeft} : \{choose\} \Rightarrow (\texttt{int}!\emptyset/\emptyset)\ \textsf{respects}\ \{(\textsc{comm})\}$$
where $h_{pickLeft}$ are the operation clauses of *pickLeft* handler, and take $f_1 = f'_1 = \lambda \star .\mathsf{in}_{\texttt{ret}}(1)$ while $f_2 = f'_2 = \lambda \star .\mathsf{in}_{\texttt{ret}}(2)$. We end up with $\mathsf{in}_{\texttt{ret}}(1)$ and $\mathsf{in}_{\texttt{ret}}(2)$ which are not equivalent under $\sim_{\texttt{int}!\emptyset/\emptyset}$.

Proposition 6.5. *If a logic* $\mathcal{L}$ *is sound, then for any* $\eta \sim_\Gamma \eta'$*, we have*

- $\Gamma \vdash v : A$ *implies* $[\![\Gamma \vdash v : A]\!]\eta \sim_A [\![\Gamma \vdash v : A]\!]\eta'$.
- $\Gamma \vdash c : \underline{C}$ *implies* $[\![\Gamma \vdash c : \underline{C}]\!]\eta \sim_{\underline{C}} [\![\Gamma \vdash c : \underline{C}]\!]\eta'$.

Proof The majority of the proof proceeds with structural induction on the typing derivation, with the only non-trivial cases being sequencing and handlers.

Recall the denotation of sequencing:

$$[\![\Gamma \vdash \mathsf{do}\ x \leftarrow c_1\ \mathtt{in}\ c_2 : \underline{D}]\!]\eta = \mathsf{lift}_{F_{[\![A]\!],\Sigma}}(\lambda a \in [\![A]\!]\,.\,[\![c_2]\!](\eta, a))([\![c_1]\!]\eta).$$

By induction, we know that $[\![c_1]\!]\eta \sim_{\underline{C}} [\![c_1]\!]\eta'$ and $[\![c_2]\!](\eta, a) \sim_{\underline{D}} [\![c_2]\!](\eta', a')$ for any $a \sim_A a'$, which further implies that $(\lambda a \in [\![A]\!]\,.\,[\![c_2]\!](\eta, a)) \sim_{A\to\underline{D}} (\lambda a \in [\![A]\!]\,.\,[\![c_2]\!](\eta', a))$. We conclude by applying Lemma 6.2, which states that lifts of related functions are related.

The other interesting case in the proof are the handlers. Recall the definition for handlers:

$$[\![\Gamma \vdash \mathtt{handler}\ (\mathtt{ret}\ x \mapsto c_r; h) : A!\Sigma/\mathcal{E} \Rightarrow \underline{D}]\!]\eta = \mathsf{lift}_{[\![h]\!](\eta)}(\lambda a \in [\![A]\!]\,.\,[\![c_r]\!](\eta, a))$$

where we define the meaning of handler cases $[\![\Gamma \vdash h : \Sigma \Rightarrow \underline{D}]\!](\eta) \in \mathsf{interp}_\Sigma([\![\underline{D}]\!])$ as the family of functions:

$$\{\lambda a \in [\![A_{op}]\!]\,.\,\lambda\kappa \in [\![B_{op} \to \underline{D}]\!]\,.\,[\![c_{op}]\!](\eta, a, \kappa)\}_{op : A_{op} \to B_{op} \in \Sigma}$$

We abbreviate the denotation of the handler as $\tilde{h}(\eta) := \mathsf{lift}_{[\![h]\!](\eta)}(\lambda a \in [\![A]\!]\,.\,[\![c_r]\!](\eta, a))$. We must prove that $\tilde{h}(\eta)$ satisfies the requirements for the relation on handler types, that is, for any $t \sim_{\underline{C}} t'$ it must hold that $\tilde{h}(\eta)(t) \sim_{\underline{D}} \tilde{h}(\eta')(t')$.

Because $\sim_{\underline{C}}$ is the smallest relation closed under the given rules, we proceed by induction on $t \sim_{\underline{C}} t'$. The cases where the relation is structural are largely the same as in proof of Lemma 6.2. The interesting case is the case of equivalence due to equations.

Suppose that the relation $t \sim_{\underline{C}} t'$ arises from the equation $\Gamma; \mathrm{Z} \vdash T_1 \sim T_2 \in \mathcal{E}$, where we denote $\Gamma = (x_i : A_i)_i$ and $\mathrm{Z} = (z_j : B_j \to *)_j$. That means that there exist $a_i \sim_{A_i} a_i'$ and $f_j \sim_{B_j \to \underline{C}} f_j'$ for which

$$t = [\![T_1]\!]((a_i)_i, (f_j)_j) \sim t' = [\![T_2]\!]((a_i')_i, (f_j')_j).$$

By Lemma 6.1, we have the equality:

$$\tilde{h}(\eta)([\![T_1]\!]_{\underline{C}}((a_i)_i, (f_j)_j)) = [\![T_1]\!]^{[\![h]\!]\eta}((a_i)_i; (\tilde{h}(\eta) \circ f_j)_j)$$

and similarly for T_2.

By induction hypotheses we get $\tilde{h}(\eta) \circ f_j \sim_{B_j \to \underline{D}} \tilde{h}(\eta') \circ f_j'$, so by the assumption on logic soundness, we have by Definition 6.4:

$$[\![\Gamma; \mathrm{Z} \vdash T_1 : \Sigma]\!]^{[\![h]\!]\eta}((a_i)_i; (\tilde{h}(\eta) \circ f_j)_j) \sim_{\underline{D}} [\![\Gamma; \mathrm{Z} \vdash T_2 : \Sigma]\!]^{[\![h]\!]\eta'}((a_i')_i; (\tilde{h}(\eta') \circ f_j')_j)$$

which concludes our proof. □

Theorem 6.6. *Assume a logic $\mathcal{L}$ is sound. Then, we have*

- *if $[\![\Gamma \vdash c : \underline{C}]\!]\eta \sim_{\underline{C}} [\![\Gamma \vdash c' : \underline{C}]\!]\eta'$ holds for any $\eta \sim_\Gamma \eta'$, then $\Gamma \vdash c \cong c' : \underline{C}$;*
- *if $[\![\Gamma \vdash v : A]\!]\eta \sim_A [\![\Gamma \vdash v' : A]\!]\eta'$ holds for any $\eta \sim_\Gamma \eta'$, then $\Gamma \vdash v \cong v' : A$.*

Proof The proof is an adaptation of Corollary 5.3. First, assume that $\mathbb{C}\langle c\rangle \rightsquigarrow^* \mathtt{ret\ true}$. By Proposition 6.5, we have $[\![\mathbb{C}\langle c\rangle]\!] \sim [\![\mathbb{C}\langle c\rangle]\!]$, and by Proposition 5.1, we have $[\![\mathbb{C}\langle c\rangle]\!] = \mathrm{in}_{\mathtt{ret}}(\mathrm{tt})$. Using a inductive proof similar to one for Proposition 6.5, we can show that the denotational semantics is structural with respect to equivalence and so $[\![\mathbb{C}\langle c\rangle]\!] \sim [\![\mathbb{C}\langle c'\rangle]\!]$. We conclude by applying Lemma 6.3. The proof for values is identical. □

6.4 Examples of sound logics

The logics $\mathcal{L}_\emptyset$, $\mathcal{L}_{\text{free}}$, and $\mathcal{L}_{\text{full}}$ presented in Section 4 are indeed sound. Next, consider the equational logic, given in 4.2. In addition to soundness, we see that all terms shown to be equivalent under the logic are also equivalent with respect to the semantics.

Theorem 6.7. *The equational logic $\mathcal{L}_{eq}$ is sound and we have*

- *If $\Gamma \vdash v \equiv_A v'$ then $[\![\Gamma \vdash v : A]\!]\eta \sim_A [\![\Gamma \vdash v' : A]\!]\eta'$.*
- *If $\Gamma \vdash c \equiv_{\underline{C}} c'$ then $[\![\Gamma \vdash c : \underline{C}]\!]\eta \sim_{\underline{C}} [\![\Gamma \vdash c' : \underline{C}]\!]\eta'$.*

Proof The proof proceeds by induction on the derivation of typing and equality judgements. Note that we cannot simply apply Proposition 6.5 because of the mutual dependence between typing judgements and equality judgements; however, we can follow the structure of its proof. Most cases are immediate apart from inheritance from the effect theory, which amounts exactly to the definition of $\sim_{A!\Sigma\mathcal{E}}$, and handler validation, which follows from the assumption that $\mathcal{L}_{\text{eq}}$ is sound. □

Composing Theorems 6.6 and 6.7, we then get

Corollary 6.8. *If $\Gamma \vdash v_1 \equiv_A v_2$, then $\Gamma \vdash v_1 \cong v_2 : A$. If $\Gamma \vdash c_1 \equiv_{\underline{C}} c_2$, then $\Gamma \vdash c_1 \cong c_2 : \underline{C}$.*

In order to show that $\mathcal{L}_{\text{pred}}$ is sound, we need to define the denotation of formulae. We interpret each formula φ in a context Γ as a relation on $[\![\Gamma]\!]$, defined by:

$$\begin{aligned}
\eta[\![v_1 \equiv_A v_2]\!]\eta' &\iff [\![\Gamma \vdash v_1 : A]\!]\eta \sim_{\underline{C}} [\![\Gamma \vdash v_2 : A]\!]\eta' \\
\eta[\![c_1 \equiv_{\underline{C}} c_2]\!]\eta' &\iff [\![\Gamma \vdash c_1 : \underline{C}]\!]\eta \sim_{\underline{C}} [\![\Gamma \vdash c_2 : \underline{C}]\!]\eta' \\
\eta[\![\top]\!]\eta' &\iff \eta \sim_\Gamma \eta' \\
\eta[\![\bot]\!]\eta' &\iff \bot \\
\eta[\![\varphi_1 \wedge \varphi_2]\!]\eta' &\iff (\eta[\![\varphi_1]\!]\eta') \wedge (\eta[\![\varphi_2]\!]\eta') \\
\eta[\![\varphi_1 \vee \varphi_2]\!]\eta' &\iff (\eta[\![\varphi_1]\!]\eta') \vee (\eta[\![\varphi_2]\!]\eta') \\
\eta[\![\varphi \Rightarrow \psi]\!]\eta' &\iff (\eta[\![\varphi]\!]\eta') \Rightarrow (\eta[\![\psi]\!]\eta') \\
\eta[\![\forall x : A.\ \varphi]\!]\eta' &\iff \forall a \sim_A a'.\ (\eta, a)[\![\varphi]\!](\eta', a') \\
\eta[\![\exists x : A.\ \varphi]\!]\eta' &\iff \exists a \sim_A a'.\ (\eta, a)[\![\varphi]\!](\eta', a')
\end{aligned}$$

Like in proof of Theorem 6.7, we proceed by an induction on the judgement derivation and show soundness of $\mathcal{L}_{\text{pred}}$. Most cases are identical to $\mathcal{L}_{\text{eq}}$ or follow from the defining properties of logical connectives and quantifiers. Soundness of induction follows straight from the inductive structure of computation types (Plotkin & Pretnar, 2008; Pretnar, 2010; Bauer & Pretnar, 2014).

Theorem 6.9. *The equational logic $\mathcal{L}_{pred}$ is sound and if $\Gamma \mid \psi_1, \ldots, \psi_n \vdash \varphi$ holds, then for any η and η' such that $\eta[\![\psi_1 \wedge \cdots \wedge \psi_n]\!]\eta'$, we have $\eta[\![\varphi]\!]\eta'$.*

7 Conclusion

7.1 Related work

Equational reasoning about monadic effects. Simple equational reasoning is one of the better properties that pure functional programs enjoy. As suggested in the seminal paper on monads (Moggi, 1991), this approach can be extended to functional programs that use monadic effects. Further examples of such reasoning can be found in Gibbons & Hinze (2011). In addition to equations over pure programs, reasoning about effectful programs employs equations describing propagation of operation calls, monadic axioms, and equations describing primitive effectful operations. All three kinds of equations are fully supported in our approach (the first is an axiom, the second is derivable with induction, and the third corresponds to inheritance from effect theories), allowing us to use the same techniques with the additional flexibility of locally varying the effect theory.

Algebraic effects and dependent types. Another approach to reason about effectful programs through richer type annotations is to employ dependent types (Brady, 2013; Ahman, 2017, 2018). In fact, Ahman (2018) is the only research work on handlers besides the original one that considers non-trivial effect theories. Its aim is similar to ours, though the implementation differs. The biggest difference is in the representation of handlers: dependent typing allows the operation clauses to be encoded in types, so any algebra for the theory has a matching type, whereas in our approach, only the free ones do.

Program optimisations. Effect theories allow us to rewrite programs into equivalent but more efficient ones. For example, idempotence of a non-deterministic choice operation allows us to skip repeated computations, while commutativity allows us to change the order of evaluation, enabling further optimisations. A survey of such transformations can be found in Kammar & Plotkin (2012). Even though the development was done in the context of a global effect theory, its results are easily adapted to our work. Though we understand that our work is far from reaching practical type-driven optimisations, we hope that it might prove to be useful in further development.

7.2 Future work

Language extensions. For the sake of a clearer presentation, we have limited our working calculus to the smallest possible fragment that already exhibits the novel features of the type system. Extending the language with additional value types such as sums or products is simple, one just needs to take care to extend the template syntax with additional value destructors. Similarly, one can extend the language with recursive functions, though in this case one must switch the denotational semantics from the category of sets to one of domains and adapt the logics to divergence (Bauer & Pretnar, 2014).

Subtyping. A sensible extension is the addition of structural subtyping as in Bauer & Pretnar (2014), Saleh *et al.* (2018). We only need to modify the rules for computation types since the addition of equations only affects computations. The simplest version is to allow $A!\Sigma/\mathcal{E} \leq A'!\Sigma'/\mathcal{E}'$ whenever $A \leq A'$, every operation in Σ appears in Σ' with a greater type, and when the theory $\mathcal{E}'$ entails all the equations in $\mathcal{E}$ (we may always consider more computations to be equivalent). The exact logic that allows such reasoning can be considered in future work. A simpler variant is to check that every equation in $\mathcal{E}$ appears in $\mathcal{E}'$ as well, though this approach is limited due to non-canonicity of the set of equations.

Formalisation. Similar to our previous work (Bauer & Pretnar, 2014; Kammar & Pretnar, 2017; Forster *et al.*, 2017; Saleh *et al.*, 2018), we wish to mechanise our formalisation in a proof assistant. From the past experience, we expect the proof of Theorem 3.1 to proceed smoothly, but expect bigger problems with proofs that depend on denotational semantics (e.g. Corollary 6.8). For that reason, we plan to look at purely syntactical treatments of contextual equivalence (McLaughlin *et al.*, 2018) that are more amenable to mechanisation.

Practical implementation. We plan on implementing the proposed system in the Eff programming language (Bauer & Pretnar, 2015). The user would annotate computation types with the desired equations, and the system would ensure they are respected. There are multiple interesting implementations to consider, such as dispatching the proofs to an SMT solver (de Moura & Bjørner, 2008), generating proof assistant templates that a user must complete with proofs, or using a QuickCheck (Claessen & Hughes, 2000) like tool, used to detect errors by running the given handler on random examples generated from equations and comparing the results on both sides.

Polymorphism. Another aspect to consider in the practical implementation is the interaction with the currently implemented polymorphic core language (Saleh *et al.*, 2018) where one may consider functions that are polymorphic both in the type and signature of an operation, for example,

$$map : \forall\alpha, \beta, \sigma.(\alpha \to \beta!\sigma) \to (\alpha\ \mathtt{list} \to \beta\ \mathtt{list}!\sigma)!\emptyset$$

Since we want such functions to preserve the equivalences between computations, it would be natural to consider polymorphism in all the components of a computation type, which would allow us to assign a type such as:

$$map : \forall\alpha, \beta, \sigma, \varepsilon.(\alpha \to \beta!\sigma/\varepsilon) \to (\alpha\ \mathtt{list} \to \beta\ \mathtt{list}!\sigma/\varepsilon)!\emptyset$$

Additional examples. All of the examples currently presented are kept minimal for clarity. By implementing the system, we plan on producing larger and more complex examples, showcasing the usefulness of our proposed system. Extending the system with some form of subtyping will also make examples more composable.

Acknowledgments

We would like to thank Danel Ahman, Bob Atkey, Andrej Bauer, Jeremy Gibbons, Ohad Kammar, Oleg Kiselyov, Gordon Plotkin, Alex Simpson, Tom Schrijvers, Nicolas Wu, all other participants of Dagstuhl seminars 16112 Bauer *et al.* (2016) and 18172 Chandrasekaran *et al.* (2018), and the anonymous referees for all their helpful insights and suggestions.

Conflicts of Interest

None.

References

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

Ahman, D. (2018) Handling fibred algebraic effects. *PACMPL* **2**(POPL), 7:1–7:29.

Bauer, A. & Pretnar, M. (2014) An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.* **10**(4), 1–29.

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

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

Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2018) Handle with care: Relational interpretation of algebraic effects and handlers. *PACMPL* **2**(POPL), 8:1–8:30.

Brady, E. (2013) Programming and reasoning with algebraic effects and dependent types. In Morrisett, G. & Uustalu, T. (eds), ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA, September 25–27, 2013. ACM, pp. 133–144.

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

Claessen, K. & Hughes, J. (2000) Quickcheck: A lightweight tool for random testing of haskell programs. In Odersky, M. & Wadler, P. (eds), Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, Canada, September 18–21, 2000. ACM, pp. 268–279.

de Moura, L. M. & Bjørner, N. (2008) Z3: An efficient SMT solver. In *TACAS*. Lecture Notes in Computer Science, vol. 4963. Springer, pp. 337–340.

Forster, Y., Kammar, O., Lindley, S. & Pretnar, M. (2017) On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *PACMPL* **1**(ICFP), 13:1–13:29.

Gibbons, J. & Hinze, R. (2011) Just do it: Simple monadic equational reasoning. In Chakravarty, M. M. T., Hu, Z. & Danvy, O. (eds), Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19–21, 2011. ACM, pp. 2–14.

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

Kammar, O. & Plotkin, G. D. (2012). Algebraic foundations for effect-dependent optimisations. In Field, J. & Hicks, M. (eds), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012. ACM, pp. 349–360.

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

Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. In Morrisett, G. & Uustalu, T. (eds), ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA, September 25–27, 2013. ACM, pp. 145–158.

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

Levy, P. B., Power, J. & Thielecke, H. (2003) Modelling environments in call-by-value programming languages. *Inf. Comput.* **185**(2), 182–210.

McLaughlin, C., McKinna, J. & Stark, I. (2018) Triangulating context lemmas. In Andronick, J. & Felty, A. P. (eds), Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8–9, 2018. ACM, pp. 102–114.

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

Plotkin, G. D. & Power, J. (2001) Adequacy for algebraic effects. In Honsell, F. & Miculan, M. (eds), Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2030. Springer, pp. 1–24.

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

Plotkin, G. D. & Pretnar, M. (2008) A logic for algebraic effects. In Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24–27 June 2008, Pittsburgh, PA, USA. IEEE Computer Society, pp. 118–129.

Plotkin, G. D. & Pretnar, M. (2009) Handlers of algebraic effects. In Castagna, G. (ed), Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5502. Springer, pp. 80–94.

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

Pretnar, M. (2010) *Logic and Handling of Algebraic Effects*. Ph.D. thesis, University of Edinburgh, UK.

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

Saleh, A. H., Karachalias, G., Pretnar, M. & Schrijvers, T. (2018) Explicit effect subtyping. In Ahmed, A. (ed), Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801. Springer, pp. 327–354.