Typed Equivalence of Effect Handlers and Delimited Control

Maciej Piróg
University of Wrocław, Poland
maciej.pirog@cs.uni.wroc.pl

Piotr Polesiuk
University of Wrocław, Poland
ppolesiuk@cs.uni.wroc.pl

Filip Sieczkowski
University of Wrocław, Poland
efes@cs.uni.wroc.pl

Abstract

It is folklore that effect handlers and delimited control operators are closely related: recently, this relationship has been proved in an untyped setting for deep handlers and the shift$_0$ delimited control operator. We positively resolve the conjecture that in an appropriately polymorphic type system this relationship can be extended to the level of types, by identifying the necessary forms of polymorphism, thus extending the definability result to the typed context. In the process, we identify a novel and potentially interesting type system feature for delimited control operators. Moreover, we extend these results to substantiate the folklore connection between shallow handlers and control$_0$ flavour of delimited control, both in an untyped and typed settings.

2012 ACM Subject Classification Theory of computation → Control primitives; Theory of computation → Operational semantics; Software and its engineering → Polymorphism

Keywords and phrases type-and-effect systems, algebraic effects, delimited control, macro expressibility

Digital Object Identifier 10.4230/LIPIcs.FSCD.2019.30

Supplement Material The formalisation of results presented in this paper, performed in the Coq proof assistant, can be found at `https://pl-uwr.bitbucket.io/efftrans.zip`.

Funding *Maciej Piróg*: National Science Centre, Poland, POLONEZ 3 grant "Algebraic Effects and Continuations" no. 2016/23/P/ST6/02217.
This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 665778.
Piotr Polesiuk: National Science Centre, Poland, grant no. 2014/15/B/ST6/00619.
Filip Sieczkowski: National Science Centre, Poland, grant no. 2016/23/D/ST6/01387.

Acknowledgements We would like to thank the anonymous reviewers of this paper for providing insightful comments and questions, particularly regarding presentation, as well as Dariusz Biernacki and Sam Lindley for helpful discussions of the technical content of the paper.

1 Introduction

Computational effects in programming languages are as pervasive as they are problematic. Virtually any programming language must allow its programmer some interaction with the outside world, at the very least through input and output channels, yet this concession tends to open Pandora's box of effects, built-in or user-defined, ranging from mutable state or logging mechanisms to complex nondeterminism or concurrency, and beyond. In most commonly used functional languages, such as OCaml or Scheme, control over occurrences

of such effects is lax at best: this, however, leads to loss of strong reasoning principles, like β-equivalence, and invalidates many common program optimisations by making observational equivalence a very small relation.

Two main approaches to control and contain the rise of complexity brought by the presence of effects exist: monads [17, 21], as used in Haskell, is the more common in today's practice, but type-and-effect systems [15, 20] appear rather often in many of the more experimental programming languages. Both these approaches feature means of establishing that a program fragment is *pure*, which intuitively means that all the usual reasoning principles and program transformations should apply, and both allow the programmer to control which particular effects can arise in a given program fragment. Although the original type-and-effect systems were geared towards particular computational effects, such as mutable state, they have since been used also for languages with user-defined effects. In the following, we consider two families of such user-defined effectful abstraction: delimited control operators and algebraic effect handlers.

Delimited control operators were invented, in different flavours, by Felleisen [5] and Danvy and Filinski [4], and have been used to encode effects ever since (see for instance [3]). Algebraic effects and handlers, proposed by Plotkin and Pretnar [18] form an alternative approach to user-defined effects, generalising exceptions and their handlers in a more direct fashion than the delimited control operators, which arise more naturally by studying the shape of effectful programs in continuation-passing style. Both these families of abstractions admit diverse type systems: in this work we focus on a particular family of type-and-effect systems with effect rows, which allows a large degree of control over what effects the program performs and what is their order in the computation.

In this paper, we work within the program outlined by Forster *et al.* [8]: we study local syntax-directed translations (Felleisen's *macro translations* [6]) between *mutually expressible* extensions of a base calculus, thus studying the relationships between different modes of programming with user-defined effects. In this line, we make two contributions: first, we pinpoint the precise mode of polymorphism that is required for the macro translations between effect handlers and delimited control operators to preserve typings, and in the process we discover a novel extension of a type system for delimited control operators. Furthermore, we extend these results to the case of *shallow* effect handlers and the $\mathrm{control}_0$ operator, thus substantiating the folklore claim that the shallow handlers behave in a "control-like" fashion. In contrast to Forster *et al.*, we focus solely on control operators with type-and-effect systems, leaving out the monadic approach to structuring effectful computations. The results presented in this paper are formalised using the Coq proof assistant.

2 The Common Core

In this paper, we discuss two calculi for delimited control operators (shift_0 and $\mathrm{control}_0$) and two for effect handlers (deep and shallow), all of which are given as extensions of the core calculus defined in this section. This core calculus is a polymorphic λ-calculus with only basic infrastructure for effects that is shared by all four final calculi. For readability, we keep the calculi minimal, that is, they are equipped only with the features necessary to discuss control effects and their relationships. In some examples, however, we assume some basic types and constants, but this is only for readability, as they are clearly a feature orthogonal to control.

The syntax of the core calculus is given in Figure 1. The type-level variables are denoted $\alpha, \beta, \ldots$, while the term-level variables are denoted $f, r, x, y, \ldots$ The syntax of terms consists of variables, λ-abstractions, applications, and, as a part of the infrastructure for effects,

$$
\begin{aligned}
\mathrm{TVar} &\ni \alpha, \beta, \dots && \text{(type-level variables)}\\
\mathrm{Var} &\ni f, r, x, y, \dots && \text{(term-level variables)}\\
\mathrm{Kind} &\ni \kappa ::= \mathsf{T} \mid \mathsf{E} \mid \mathsf{R} && \text{(kinds)}\\
\mathrm{Typelike} &\ni \sigma, \tau, \varepsilon, \rho ::= \alpha \mid \tau \to_\rho \tau \mid \forall\, \alpha :: \kappa \,.\, \tau \mid \iota \mid \varepsilon \cdot \rho && \text{(types and rows)}\\
\mathrm{Val} &\ni u, v ::= x \mid \boldsymbol{\lambda}\, x\mathbf{.}\ e && \text{(values)}\\
\mathrm{Exp} &\ni e ::= v \mid e\ e \mid [e] && \text{(expressions)}\\
\mathrm{ECont} &\ni E ::= \Box \mid E\ e \mid v\ E \mid [E] && \text{(evaluation contexts)}
\end{aligned}
$$

Figure 1 Syntax of the common core of the calculi.

$$
\frac{\alpha :: \kappa \in \Delta}{\Delta \vdash \alpha :: \kappa}
\qquad
\frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta \vdash \rho :: \mathsf{R} \qquad \Delta \vdash \tau_2 :: \mathsf{T}}{\Delta \vdash \tau_1 \to_\rho \tau_2 :: \mathsf{T}}
\qquad
\frac{\Delta, \alpha :: \kappa \vdash \tau :: \mathsf{T}}{\Delta \vdash \forall\, \alpha :: \kappa \,.\, \tau :: \mathsf{T}}
$$

$$
\frac{}{\Delta \vdash \iota :: \mathsf{R}}
\qquad
\frac{\Delta \vdash \varepsilon :: \mathsf{E} \qquad \Delta \vdash \rho :: \mathsf{R}}{\Delta \vdash \varepsilon \cdot \rho :: \mathsf{R}}
$$

Figure 2 Well-formedness of types and rows.

Biernacki *et al.*'s [2] *lift* operator $[e]$, which we discuss below. Note that the core calculus is given à la Curry, so the variable in a λ-abstraction is not labelled with a type. Moreover, we have the universal quantifier as a type former, but it is not reflected in the term-level syntax: generalisations and instantiations of the quantifier take place only in type derivations.

The core calculus is equipped with a type-and-effect system organised into three kinds: T for types, E for single effects, and R for rows of effects. Considering the well-formedness rules given in Figure 2, we read that the types are given by the mentioned universal quantifier (in which we quantify over variables of any kind), type variables, and arrows, which are decorated with rows of effects. Intuitively, a row of effects specifies what effects can happen when we evaluate an expression or call an effectful function (for a discussion on row-based type-and-effect systems see, for instance, [14]). Importantly, the specific calculi that we present in the subsequent sections do not distinguish effects by any sort of names, but by their position in the row associated to an expression. This means that the order of effects in a row is important and, intuitively, corresponds to the order of delimiters (handlers, resets) that can be placed around the expression – or, from another point of view, to the order in which a closing context introduces these effects. This is why we include the lift operator, as it allows us to manipulate the order of effects in a row and so, as discussed in Section 3.1, simulate a calculus with named effects. Formally, a row of effects can be given by a row variable, an empty row, written ι, or can be built by placing an effect in front of an existing row. In other words, each row is either *closed*, which means that it is a list of effects (in this case, we tend to omit the trailing ι when this does not lead to confusion), or *open*, which means that it is a list of effects that ends with a row variable. Note that the core calculus does not say much about single effects, except for the fact that we can quantify over effects as well as over types or rows. However, it captures how effects are organised into rows, as this aspect is common to all four final calculi.

$$\frac{}{\Delta \vdash \sigma <: \sigma} \qquad \frac{\Delta \vdash \tau_2^1 <: \tau_1^1 \qquad \Delta \vdash \rho_1 <: \rho_2 \qquad \Delta \vdash \tau_1^2 <: \tau_2^2}{\Delta \vdash \tau_1^1 \to_{\rho_1} \tau_1^2 <: \tau_2^1 \to_{\rho_2} \tau_2^2}$$

$$\frac{\Delta, \alpha :: \kappa \vdash \tau_1 <: \tau_2}{\Delta \vdash \forall\, \alpha :: \kappa \,.\, \tau_1 <: \forall\, \alpha :: \kappa \,.\, \tau_2} \qquad \frac{\Delta \vdash \rho :: \mathsf{R}}{\Delta \vdash \iota <: \rho} \qquad \frac{\Delta \vdash \rho_1 <: \rho_2}{\Delta \vdash \varepsilon \cdot \rho_1 <: \varepsilon \cdot \rho_2}$$

Figure 3 Subtyping.

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau \,/\, \iota} \qquad \frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,/\, \rho}{\Delta; \Gamma \vdash \boldsymbol{\lambda} x.\ e : \tau_1 \to_\rho \tau_2 \,/\, \iota}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to_\rho \tau_2 \,/\, \rho \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \,/\, \rho}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2 \,/\, \rho} \qquad \frac{\Delta \vdash \varepsilon :: \mathsf{E} \qquad \Delta; \Gamma \vdash e : \tau \,/\, \rho}{\Delta; \Gamma \vdash [e] : \tau \,/\, \varepsilon \cdot \rho}$$

$$\frac{\Delta, \alpha :: \kappa; \Gamma \vdash e : \tau \,/\, \iota}{\Delta; \Gamma \vdash e : \forall\, \alpha :: \kappa \,.\, \tau \,/\, \iota} \qquad \frac{\Delta \vdash \sigma :: \kappa \qquad \Delta; \Gamma \vdash e : \forall\, \alpha :: \kappa \,.\, \tau \,/\, \rho}{\Delta; \Gamma \vdash e : \tau\{\sigma \,/\, \alpha\} \,/\, \rho}$$

$$\frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta \vdash \rho_1 <: \rho_2 \qquad \Delta; \Gamma \vdash e : \tau_1 \,/\, \rho_1}{\Delta; \Gamma \vdash e : \tau_2 \,/\, \rho_2}$$

Figure 4 Core typing rules.

Figure 3 shows the subtyping rules. The purpose of subtyping is to express the idea that one expression can perform "more" effects than another. In particular, it means that the empty row is a subtype of any other row, while two functions are in the relation if they satisfy the usual "contravariant-in-argument, covariant-in-result" condition together with the appropriate subtyping of their rows of effects.

With this, we can define the typing relation $\Delta; \Gamma \vdash e : \tau \,/\, \rho$, shown in Figure 4. It uses two contexts, Δ to assign a kind to a free type-level variable, and Γ to assign a type to a free term-level variable.[1] A term e is given a type τ and a row of effects ρ, which may be performed when evaluating e. The rules for variables, λ-abstractions, and applications are rather straightforward; the only deviation from the classic formulation of Talpin and Jouvelot [20] is that in the application rule we require the same effect on all premises, which is natural, as our system includes explicit subtyping. Additionally, we have the aforementioned rules that allow us to introduce and instantiate a universal quantifier, and a rule to apply subtyping. The rule for $[e]$ reveals some intuition about the purpose of the lift operator, as it allows us to add any effect in front of the row to "mask" the actual first effect. As hinted before, this is due to the fact that order of the effects matters: thus, we might be allowed to freely pretend that a given computation has more effects "outside" the ones that are actually present, via subeffecting, but we need to be explicit, and use an operator with some runtime content, when we want to manipulate effects at the *beginning* of the row. As it is much more natural to discuss the details of the semantics of lift with some actual effects at hand, we return to this discussion in more depth in Section 3.1, where we define deep effect handlers.

[1] We treat contexts as lists of bindings, allowing ourselves to write types of the form $\forall\, \Delta \,.\, \tau$, with the natural expansion as a chain of nested quantifiers of appropriate length.

$$\boldsymbol{\lambda} x\boldsymbol{.}\ e\ v \mapsto e\{v\,/\,x\} \qquad [v] \mapsto v \qquad \frac{e_1 \mapsto e_2}{E[e_1] \to E[e_2]}$$

Figure 5 Single-step reduction and core contraction rules.

$$\frac{}{0{-}\mathrm{free}(\Box)} \qquad \frac{n{-}\mathrm{free}(E)}{n{-}\mathrm{free}(E\ e)} \qquad \frac{n{-}\mathrm{free}(E)}{n{-}\mathrm{free}(v\ E)} \qquad \frac{n{-}\mathrm{free}(E)}{n+1{-}\mathrm{free}([E])}$$

Figure 6 Freeness for core evaluation contexts.

Figure 5 together with the syntax of evaluation contexts from Figure 1 define the call-by-value operational semantics in terms of contractions, that is, single-step reductions in an evaluation context, in the standard style of Felleisen and Friedman [7]. Note that while the lift operator does not have any computational content on its own, it is used to manipulate the freeness of evaluation contexts defined in Figure 6. The freeness judgment was introduced by Biernacki et al. [2] to judgmentally match an effect handler to the operation, but it readily scales to the other systems, allowing us to move the lift to the common part of the calculus. Of course, since the core calculus does not contain any effect delimiters, the freeness can only increase, and only via the lift, but its use will become apparent when we consider particular effects.

3 Deep Effect Handlers and Delimited Control

We now turn to study the connection between the deep handlers and the shift_0 calculus of delimited control operators. The operational semantics of the calculi and their (untyped) equivalence are a slight variation on the ones studied by Forster et al. [8]. The correspondence between the type systems is our contribution.

3.1 Deep Handler Calculus

The calculus of deep effect handlers is given in Figure 7 as an extension of the core calculus. We extend the syntax of expressions with two forms: the first one, **do** v, is an operation with a single argument. Note that while the argument is necessarily a value, this restriction is purely a matter of convenience. The second new form is a handler, **handle** e $\{x, r\boldsymbol{.}\ e\boldsymbol{;}\ x\boldsymbol{.}\ e\}$, in which the first expression is the handled computation, the $x, r\boldsymbol{.}\ e$ part is an interpretation of the operation, while $x\boldsymbol{.}\ e$ is the "return" clause, which describes the behaviour of the handler on effect-free computations. Handlers come with one new form of evaluation contexts, which allows reductions in the handled expression.

The reduction semantics for handlers is given by two contraction rules. The first one states that if we handle a 0-free context that has an operation in the evaluation position, the handler takes over, and proceeds according to the $x, r\boldsymbol{.}\ e_h$ part, where e_h is the computation that we proceed with, in which x is bound to the value of the argument of the operation, while r is bound to the resumption, which allows us to continue evaluating the handled computation with a given value in place of the operation. Note that the resumption v_c is again wrapped in the same handler, which is why such handlers are called *deep*, as opposed

$$\sigma ::= \ldots \mid \Delta\,.\,\tau_1 \Rightarrow \tau_2$$

$$e ::= \ldots \mid \textbf{do}\ v \mid \textbf{handle}\ e\ \{x, r.\ e_h;\ x.\ e_r\}$$

$$E ::= \ldots \mid \textbf{handle}\ E\ \{x, r.\ e_h;\ x.\ e_r\}$$

$$\frac{\Delta, \Delta' \vdash \tau_1 :: \mathsf{T} \qquad \Delta, \Delta' \vdash \tau_2 :: \mathsf{T}}{\Delta \vdash \Delta'\,.\,\tau_1 \Rightarrow \tau_2 :: \mathsf{E}}$$

$$\frac{n+1\text{−free}(E)}{n\text{−free}(\textbf{handle}\ E\ \{x, r.\ e_h;\ x.\ e_r\})}$$

$$\frac{0\text{−free}(E) \qquad v_c = \boldsymbol{\lambda}\, z.\ \textbf{handle}\ E[z]\ \{x, r.\ e_h;\ x.\ e_r\}}{\textbf{handle}\ E[\textbf{do}\ v]\ \{x, r.\ e_h;\ x.\ e_r\} \mapsto e_h\{v\,/\,x\}\{v_c\,/\,r\}}$$

$$\textbf{handle}\ v\ \{x, r.\ e_h;\ x.\ e_r\} \mapsto e_r\{v\,/\,x\}$$

$$\frac{\Delta; \Gamma \vdash v : \delta(\tau_1)\ /\ \iota \qquad \Delta \vdash \delta :: \Delta' \qquad \Delta \vdash \Delta'\,.\,\tau_1 \Rightarrow \tau_2 :: \mathsf{E}}{\Delta; \Gamma \vdash \textbf{do}\ v : \delta(\tau_2)\ /\ (\Delta'\,.\,\tau_1 \Rightarrow \tau_2)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \tau\ /\ (\Delta'\,.\,\tau_1 \Rightarrow \tau_2) \cdot \rho \\ \Delta, \Delta'; \Gamma, x : \tau_1, r : \tau_2 \to_\rho \tau_r \vdash e_h : \tau_r\ /\ \rho \qquad \Delta; \Gamma, x : \tau \vdash e_r : \tau_r\ /\ \rho\end{array}}{\Delta; \Gamma \vdash \textbf{handle}\ e\ \{x, r.\ e_h;\ x.\ e_r\} : \tau_r\ /\ \rho}$$

Figure 7 Calculus of deep handlers.

to shallow handlers discussed in Section 4.1. The second single-step reduction rule is used when we handle a value. In such a case, the entire handler evaluates to e_r from the $x.\ e_r$ part, in which x is bound to the handled value.

A single effect in the type system is given as $\Delta\,.\,\tau_1 \Rightarrow \tau_2$, which, intuitively, specifies the type of the **do** operation. This means that its argument is of the type τ_1, while the operation applied to an argument can be used in contexts that expect an expression of the type τ_2. Importantly, each effect may bind any number of type variables of appropriate kinds, denoted by Δ, which means that the effect $\tau_1 \Rightarrow \tau_2$ can be polymorphic. One example of why such a feature is useful is the error effect, in which the operation is "raise an exception". Since we need to be able to do it regardless of the expected type, such an effect is captured by $\alpha :: \mathsf{T}\,.\,\mathit{unit} \Rightarrow \alpha$. For our purposes, such polymorphic effects are needed for the typed translations between deep handlers and the shift$_0$/reset delimited control operators. Such polymorphic operations have been considered before, for instance by Kammar et al. [11], and indeed such an extension to the type system with effects seems rather natural.

The typing rule for the **do** operation indeed matches the shape of the effect: the argument is required to have the type τ_1 and the entire expression has type τ_2. Moreover, we may pick any *type-level substitution* δ that is well-formed in Δ, and apply it to both types, thus *instantiating* the polymorphic operation. For example, in the error effect, if we want to raise an exception in a context in which we expect a value of the type *int*, we instantiate α to *int*. Formally, we define the well-formedness of a substitution as

$$\Delta \vdash \delta :: \Delta' \triangleq \mathrm{dom}(\delta) = \mathrm{dom}(\Delta') \wedge \forall \alpha \in \mathrm{dom}(\delta), \Delta \vdash \delta(\alpha) :: \Delta'(\alpha).$$

The typing rule for **handle** $e\ \{x, r.\ e_h;\ x.\ e_r\}$ reveals that we handle the first effect in the row of the handled expression. The overall result of the handler has a type τ_r, which needs to be the type of both e_h and e_r. Importantly, the handler needs to be polymorphic in Δ', which means that for each particular occurrence of the operation, the expression e_h must be oblivious to the concrete instantiation of the variables of Δ'.

This type system is sound with respect to the operational semantics, as shown by the following result, obtained by the standard technique of progress and preservation (cf. Harper [9], Wright and Felleisen [22]:

▶ **Theorem 1.** *If* $\cdot;\cdot \vdash e : \tau \,/\, \iota$ *and* $e \to^* e' \not\to$ *in the calculus of deep handlers, then there exists a value* v *such that* $e' = v$ *and* $\cdot;\cdot \vdash v : \tau \,/\, \iota$.

One aspect that distinguishes our approach from most calculi for algebraic effects in the literature is that, for the simplicity of presentation, we have only one operation, **do**. Usually, one assumes a number of operations, often grouped in named effects, which is much more convenient in a programmer-level language, but is not necessary for our semantic considerations. This is the reason why we include the lift operator $[e]$ in our calculi, since it allows us to express programs with multiple effects. To see that this is the case, we now briefly discuss the relationship with multiple effects and multiple operations in an effect.

In most settings with multiple effects, each operation is associated with a particular effect [2, 13]. Then, operationally, it is assigned to the nearest enclosing handler of this effect. On the level of types, one then takes the rows $\varepsilon_1 \cdot \varepsilon_2 \cdot \rho$ and $\varepsilon_2 \cdot \varepsilon_1 \cdot \rho$ as equivalent for any two distinct effects ε_1 and ε_2, which allows freely swapping a pair of different effects in a row. Then, in a typing derivation, one needs to apply enough swaps to move the effect handled by a particular handler to the front of the row of the handled expression. However, as shown by Biernacki et al. [2], in a language with the lift operator, we can express a term-level swap function for any two positions in the row. For example, consider the expression $\mathbf{do}\ 2 + [\mathbf{do}\ ()] : \mathit{int} \,/\, (.\ \mathit{int}{\Rightarrow}\mathit{int}) \cdot (.\ \mathit{unit}{\Rightarrow}\mathit{int})$. Then, swapping the first two effects in the row would yield an expression equivalent to $[\mathbf{do}\ 2] + \mathbf{do}\ () : \mathit{int} \,/\, (.\ \mathit{unit}{\Rightarrow}\mathit{int}) \cdot (.\ \mathit{int}{\Rightarrow}\mathit{int})$. This means that our calculus can simulate a multiple-effect setting by manually placing enough lifts and swaps.

Another issue is the presence of multiple operations in an effect, which are handled together by a single handler. One can simulate this by tagging the argument of **do**, in a way that allows to distinguish between the operations. The difficulty is that different operations can have different types, depending on the tag. Thus, to simulate multiple operations in an effect, we could use GADTs, possibly encoded via equality types [23]. This aspect, however, is orthogonal to the aspects of control that we consider in this paper.

3.2 The shift$_0$ Delimited Control Calculus

The calculus of the shift$_0$ flavour of delimited control is given in Figure 8. It extends the core calculus with two new syntactic constructs: the $\mathbf{shift_0}$ operator and the reset delimiter, $\langle e | x.\ e_r \rangle$, where the expression e is the delimited computation, while $x.\ e_r$ is the "return" part.

Operationally, an expression $\mathbf{shift_0}\, k.\ e$ aborts the evaluation of the appropriate enclosing reset, replacing it with the expression e. However, the evaluation of the entire reset is not all-lost, as it is captured as a continuation bound to the variable k in e. If it happens that no $\mathbf{shift_0}$ is evaluated inside e in $\langle e | x.\ e_r \rangle$, the value of the entire reset is given by the value of e_r in which x is bound to the value of e.

The type-and-effect system is a generalisation of the calculus introduced by [8]. A single effect $\Delta\,.\ \tau \,/\, \rho$ can intuitively be seen as a specification of the type and effect of the continuation captured by any shift within the expression. Thus, if we consider the computation $\langle e | x.\ e_r \rangle$ and assume it has some type τ_r and effect row ρ_r, this type and row will be preserved as the first effect in the row associated with e. Then, in each $\mathbf{shift_0}\, k.\ e_s$ within e, the expression e_s is supposed to have this precise type and effect row – after all, the entire reset is replaced by e_s, so their types must agree. As a novel feature in our

$$\sigma ::= \ldots \mid \Delta \,.\, \tau \,/\, \rho$$
$$e ::= \ldots \mid \mathbf{shift_0}\, k.\ e \mid \langle e | x.\ e \rangle$$
$$E ::= \ldots \mid \langle E | x.\ e \rangle$$

$$\frac{\Delta, \Delta' \vdash \tau :: \mathsf{T} \qquad \Delta, \Delta' \vdash \rho :: \mathsf{R}}{\Delta \vdash \Delta' \,.\, \tau \,/\, \rho :: \mathsf{E}}$$

$$\frac{n+1\text{−free}(E)}{n\text{−free}(\langle E | x.\ e \rangle)}$$

$$\frac{0\text{−free}(E) \qquad v_c = \boldsymbol{\lambda}\, z.\ \langle E[z] | x.\ e_r \rangle}{\langle E[\mathbf{shift_0}\, k.\ e] | x.\ e_r \rangle \mapsto e\{v_c \,/\, k\}}$$

$$\langle v | x.\ e \rangle \mapsto e\{v \,/\, x\}$$

$$\frac{\begin{array}{c}\Delta, \Delta'; \Gamma, k : \tau' \to_\rho \tau \vdash e : \tau \,/\, \rho' \\ \Delta, \Delta' \vdash \rho' <: \rho \qquad \Delta \vdash \tau' :: \mathsf{T} \qquad \Delta \vdash (\Delta' \,.\, \tau \,/\, \rho) \cdot \rho' :: \mathsf{R}\end{array}}{\Delta; \Gamma \vdash \mathbf{shift_0}\, k.\ e : \tau' \,/\, (\Delta' \,.\, \tau \,/\, \rho) \cdot \rho'}$$

$$\frac{\Delta \vdash \delta :: \Delta' \qquad \Delta; \Gamma \vdash e : \tau' \,/\, (\Delta' \,.\, \tau \,/\, \rho) \cdot \delta(\rho) \qquad \Delta; \Gamma, x : \tau' \vdash e_r : \delta(\tau) \,/\, \delta(\rho)}{\Delta; \Gamma \vdash \langle e | x.\ e_r \rangle : \delta(\tau) \,/\, \delta(\rho)}$$

Figure 8 Calculus of shift$_0$/reset.

calculus, parts of the type and row captured as an effect by the reset may be abstracted: thus, both τ and ρ in an effect specification might depend on type variables bound by Δ – by the typing rule in Figure 8 we would have $\tau_r = \delta(\tau)$, and likewise for ρ. This can be intuitively understood as akin to existential quantification: the are concrete types known to a reset are abstracted within its body, but they are not revealed to the shifts within, which thus have to treat them parametrically, somewhat like an unpack operation.

This novel feature of the system should make us wonder if it is ever practical, particularly so given the long history of delimited control operators – and we believe it is. As an example, imagine a library that implements simple exceptions, with the signature given as two functions, $\mathsf{throw} : \forall\, \alpha :: \mathsf{T}\,.\ \mathit{unit} \to_\varepsilon \alpha$ and $\mathsf{try} : \forall\, \alpha :: \mathsf{T}\,.\ (\mathit{unit} \to_\varepsilon \alpha) \to \alpha \to \alpha$,[2] for some, as yet unknown, effect ε. Can we express such a library using shift$_0$ and reset and, if so, what is the definition of ε? Clearly, throw needs to capture the context and discard it, and try should delimit the context – but how can throw procure the result it should return? In fact, it cannot produce such a result directly, since at definition site the type of the result is unknown – indeed, throw can (and should) be used in multiple contexts, where the return types expected by try are different.

This is where parametricity comes in. We can have throw capture the context and return an identity function, which will return whatever value the enclosing try would feed it, giving us $\mathsf{throw}\ () \triangleq \mathbf{shift_0}\ \_.\ \boldsymbol{\lambda}\, x.\ x$, which matches the signature given $\varepsilon \triangleq \alpha :: \mathsf{T}\,.\ \alpha \to \alpha \,/\, \iota$. Similarly, we have $\mathsf{try}\ \mathit{th}\ v \triangleq \langle \mathit{th}\ () | x.\ \boldsymbol{\lambda}\, \_.\ x \rangle\ v$, which also matches its signature with the given definition of ε. This simple example shows that this novel form of polymorphism gives us a certain separation of concerns, where the programmer may write effectful library functions and eliminate them using other constructs that insert reset delimiters in the appropriate places, instantiating the polymorphic effect at the same time.

[2] The second argument of try denotes the result of the entire expression if the exception was raised; this example could clearly be generalised, but as an illustration it is sufficient.

$$[\![\mathbf{shift_0}\ k.\ e]\!]^{\mathrm{DH}} \triangleq \mathbf{do}\ (\boldsymbol{\lambda}\, k.\ [\![e]\!]^{\mathrm{DH}})$$

$$[\![\langle e|x.\ e_r\rangle]\!]^{\mathrm{DH}} \triangleq \mathbf{handle}\ [\![e]\!]^{\mathrm{DH}}\ \{x, r.\ x\ r;\ x.\ [\![e_r]\!]^{\mathrm{DH}}\}$$

$$[\![\Delta\,.\,\tau\ /\ \rho]\!]^{\mathrm{DH}} \triangleq \alpha :: \mathsf{T}\,.\,\big(\forall\,\Delta\,.\,(\alpha{\to}_{[\![\rho]\!]^{\mathrm{DH}}}[\![\tau]\!]^{\mathrm{DH}}){\to}_{[\![\rho]\!]^{\mathrm{DH}}}[\![\tau]\!]^{\mathrm{DH}}\big){\Rightarrow}\alpha$$

Figure 9 Shift$_0$ as deep handlers.

Note that our calculus has additional constructs over the usual presentations of shift$_0$ and reset. In particular, we have the return clause in resets and the lift operator inherited from the core calculus. However, these constructs can be both macro-expressed in the standard setting for well-typed programs: the former was shown by Materzok and Biernacki [16], while the latter is definable as $[e] \triangleq \mathbf{shift_0}\ k.\ k\ e$.

Like with the deep handlers, we use progress and preservation lemmas to obtain the following soundness theorem:

▶ **Theorem 2.** *If $\cdot;\cdot \vdash e : \tau\ /\ \iota$ and $e \to^* e' \not\to$ in the calculus of shift$_0$ and reset, then there exists a value v such that $e' = v$ and $\cdot;\cdot \vdash v : \tau\ /\ \iota$.*

3.3 Typed Correspondence

We show the typed correspondence of the calculi presented in previous sections by presenting two local syntax-directed translations between calculi and showing that they preserve types and semantics. The translations of expressions are simple adaptations of those of Forster et al. [8], but the novel parts are translations of types and identifying which kind of polymorphism is required to obtain type preservation.

The translation from the calculus of shift$_0$ to deep handlers is shown in Figure 9. We only show translations of those parts of the language which need to be "macro expanded", i.e., the control constructs $\mathbf{shift_0}\ k.\ e$ and $\langle e|x.\ e_r\rangle$ at the level of expressions and the single effects at the level of types. For other constructs the translation behaves homomorphically: λ-abstraction is translated to a λ-abstraction, application to an application, etc.

The control operator $\mathbf{shift_0}$ performs en effect, so it is translated to **do** operation. The body of $\mathbf{shift_0}$ operator is expressed as a λ-abstraction and then passed as an argument to the operation, while a handler is responsible for providing the captured continuation. Indeed, the translation of reset follows this protocol: reset is turned into a handler which applies an argument of operation directly to the resumption.

In order to explain the translation of single effects, consider the following example. The expression $\langle E[\mathbf{shift_0}\ k.\ e]|x.\ e_r\rangle$ of type τ_0 where E is a 0-free evaluation context, will be translated into

$$\mathbf{handle}\ ([\![E]\!]^{\mathrm{DH}}[\mathbf{do}\ (\boldsymbol{\lambda}\, k.\ [\![e]\!]^{\mathrm{DH}})])\ \{x, r.\ x\ r;\ x.\ [\![e_r]\!]^{\mathrm{DH}}\},$$

so the effect handled by this handler should have the shape $((\tau'{\to}_\rho\tau){\to}_\rho\tau){\Rightarrow}\tau'$ where $\tau = [\![\tau_0]\!]^{\mathrm{DH}}$ and τ' is a translation of the type of $\mathbf{shift_0}\ k.\ e$ expression. The main difficulty of assigning types to this translation is that type τ' is not known by the handler, and can be different for any of the $\mathbf{shift_0}$ constructs, potentially even within the same reset: type τ' in a typing rule of $\mathbf{shift_0}$ does not occur in the effect. To solve this problem we use polymorphic effects and quantify over all possible τ' in the translated effect: thus, the result has the shape of $\alpha :: \mathsf{T}\,.\,((\alpha{\to}_\rho\tau){\to}_\rho\tau){\Rightarrow}\alpha$.

$$[\![\textbf{do}\ v]\!]^{\mathrm{DD}} \triangleq \textbf{shift}_0\ k.\ \boldsymbol{\lambda} h.\ h\ [\![v]\!]^{\mathrm{DD}}\ (\boldsymbol{\lambda} x.\ k\ x\ h)$$

$$[\![\textbf{handle}\ e\ \{x, r.\ e_h;\ x.\ e_r\}]\!]^{\mathrm{DD}} \triangleq \langle [\![e]\!]^{\mathrm{DD}} | x.\ \boldsymbol{\lambda} h.\ [\![e_r]\!]^{\mathrm{DD}} \rangle\ (\boldsymbol{\lambda} x.\ \boldsymbol{\lambda} r.\ [\![e_h]\!]^{\mathrm{DD}})$$

$$[\![\Delta\,.\,\tau_1 \Rightarrow \tau_2]\!]^{\mathrm{DD}} \triangleq \alpha :: \mathsf{T}, \beta :: \mathsf{R}\,.\,\left((\forall\,\Delta\,.\,[\![\tau_1]\!]^{\mathrm{DD}} \to ([\![\tau_2]\!]^{\mathrm{DD}} \to_\beta \alpha) \to_\beta \alpha) \to_\beta \alpha\right)\ /\ \beta$$

Figure 10 Deep handlers as shift$_0$.

Polymorphic variables of the delimited-control effect (bound by Δ), behave, intuitively, as if they were existentially quantified, so they cannot be directly translated as polymorphic variables of the algebraic effect, which behave akin to universal quantification. Therefore, we express them as a chain of universal quantifiers on the left-hand-side of an arrow type: thus, the body of **shift$_0$** is polymorphic in Δ.

The translation preserves types and semantics, which is expressed by the following theorems.

▶ **Theorem 3.** *If $\Delta; \Gamma \vdash e : \tau\ /\ \rho$ in the calculus of shift$_0$, then $\Delta; [\![\Gamma]\!]^{\mathrm{DH}} \vdash [\![e]\!]^{\mathrm{DH}} : [\![\tau]\!]^{\mathrm{DH}}\ /\ [\![\rho]\!]^{\mathrm{DH}}$ in the calculus of deep handlers.*

▶ **Theorem 4** (after Forster et al.)**.** *If $e \to e'$ in the calculus of shift$_0$, then in the calculus of deep handlers we have $[\![e]\!]^{\mathrm{DH}} \to^+ [\![e']\!]^{\mathrm{DH}}$.*

The translation in the opposite direction is shown in Figure 10 and is slightly more complex. When the **do** v construct is reduced in the source language, the control is passed to the handler. After the translation, the **shift$_0$** operator needs to somehow obtain the code that "handles" the operation: thus, the body of **shift$_0$** immediately returns a λ-abstraction which expects the translated code of the handler as an argument. Therefore, the translation of the handler is itself more involved: it is translated into a reset *applied* to the body of the handler, itself expressed as a function.

Since the translation of a handler is not just a reset, but a reset applied to an argument, we have to take care about this argument in two places. First, the return clause of a handler is translated into a λ-abstraction which throws away the handler code. Secondly, the continuation k captured by the **shift$_0$** in the translation of **do** contains the reset, but does not contain the handler code. So the resumption passed to the h is not just k, but $\boldsymbol{\lambda} x.\ k\ x\ h$.

The type part of the translation of an effect $\Delta\,.\,\tau_1 \Rightarrow \tau_2$ is the type of a reset in the translation of a handler. This reset is a function which expects the translated handler clause of type $\forall\,\Delta\,.\,[\![\tau_1]\!]^{\mathrm{DD}} \to ([\![\tau_2]\!]^{\mathrm{DD}} \to_\beta \alpha) \to_\beta \alpha$ and returns α, where α is a type of the entire translated handler expression. The type α and the effect β of translated handler expression are not known at the site where **do** is performed, so we quantify over them in the translated effect, similarly to the previous translation. Now we can show that the translation preserves types.

▶ **Theorem 5.** *If $\Delta; \Gamma \vdash e : \tau\ /\ \rho$ holds in the calculus of deep handlers, then $\Delta; [\![\Gamma]\!]^{\mathrm{DD}} \vdash [\![e]\!]^{\mathrm{DD}} : [\![\tau]\!]^{\mathrm{DD}}\ /\ [\![\rho]\!]^{\mathrm{DD}}$ holds in the calculus of shift$_0$.*

As noted by Forster et al., we cannot obtain a direct analogue of Theorem 4. Instead, we let $\to_i$ be a relation on expressions in target calculus, defined by the following rule

$$\frac{e_1 \mapsto e_2}{C[e_1] \to_i C[e_2]},$$

where C is a general context with one hole, not necessarily in the evaluation position, and obtain the following theorem.

▶ **Theorem 6** (after Forster et al.). *If $e \to e'$ holds in the calculus of deep handlers, then $[\![e]\!]^{\mathrm{DD}} \to_i^+ [\![e']\!]^{\mathrm{DD}}$.*

4 Shallow Effect Handlers and Delimited Control

We now turn to the calculi of shallow handlers and control$_0$. While these flavours of operators are arguably less popular than those studied in the previous section, they are nonetheless an interesting part of the spectrum of user-defined control operators. In the following, we formally capture the inter-expressibility of these two forms of delimited control, and comment on the differences with respect to the translations and results obtained in the case of deep handlers and shift$_0$.

4.1 Shallow Handler Calculus

Much like in the case of deep handlers, the calculus of shallow handlers is given in Figure 11. These handlers do not differ syntactically from their deep counterparts: the only difference lies within their semantics.

The operational semantics proceeds in a fashion that is very similar to the deep handlers: a **do** operation matches an enclosing handler using a freeness judgment, and both the value, and the captured continuation are passed to the handler. The only difference is that the captured continuation does *not* contain the handler itself – rather, only the evaluation context *within* the handler is captured. This lack of replication is often intuitively explained via the analogy to the case analysis and recursors, with the deep handlers' replication of the context making them behave in a more fold-like fashion [10].

Much like in the case of deep handlers, the calculus has a single effect constructor that denotes the type of the **do** operation. However, in addition to the polymorphic quantification that we have already seen in the previous section, this effect is *recursive*: it is prefixed with a form $\mu\alpha$, which binds α in the body of the effect as an effect-kinded variable. In both typing rules, when we access the "input" and "output" types of the effect, we substitute the entire effect for α in their body, thus effectively performing a single unfolding of the recursive effect. The only other difference with respect to the type system for the deep handlers lies in the type of the resumption in the rule for **handle**: since the resumption does not contain the handler, its return type is τ rather than τ_r, and it may still perform the same effect ε.

While recursive effect declarations abound in the literature, we are unaware of a prior formulation where the recursion is entirely confined to the type level, without making any appearance at the expression level. Note also that, much like polymorphic effects, the recursive effects are not necessary to establish soundness of the type system: we use them to establish inter-expressibility with the calculus of control$_0$.

As with the previous calculi, we prove type soundness via progress and preservation:

▶ **Theorem 7.** *If $\cdot;\cdot \vdash e : \tau \,/\, \iota$ and $e \to^* e' \not\to$ in the calculus of shallow handlers, then there exists a value v such that $e' = v$ and $\cdot;\cdot \vdash v : \tau \,/\, \iota$.*

4.2 The control$_0$ Delimited Control Calculus

Finally, we turn to the last of our calculi, the delimited control calculus of control$_0$, presented in Figure 12. Much like the difference between deep and shallow handlers, the differences between various delimited control operators are rather subtle. The common intuition refers to

$$\sigma ::= \ldots \mid \mu\,\alpha\,.\,\Delta\,.\,\tau_1 \Rightarrow \tau_2$$

$$e ::= \ldots \mid \mathbf{do}\ v \mid \mathbf{handle}\ e\ \{x, r.\ e;\ x.\ e\}$$

$$E ::= \ldots \mid \mathbf{handle}\ E\ \{x, r.\ e;\ x.\ e\}$$

$$\frac{\Delta, \alpha :: \mathsf{E}, \Delta' \vdash \tau_1 :: \mathsf{T} \qquad \Delta, \alpha :: \mathsf{E}, \Delta' \vdash \tau_2 :: \mathsf{T}}{\Delta \vdash \mu\,\alpha\,.\,\Delta'\,.\,\tau_1 \Rightarrow \tau_2 :: \mathsf{E}}$$

$$\frac{n + 1\text{−free}(E)}{n\text{−free}(\mathbf{handle}\ E\ \{x, r.\ e_h;\ x.\ e_r\})}$$

$$\frac{0\text{−free}(E)}{\mathbf{handle}\ E[\mathbf{do}\ v]\ \{x, r.\ e_h;\ x.\ e_r\} \mapsto e_h\{v\,/\,x\}\{\boldsymbol{\lambda}\, z.\ E[z]\,/\,r\}}$$

$$\mathbf{handle}\ v\ \{x, r.\ e_h;\ x.\ e_r\} \mapsto e_r\{v\,/\,x\}$$

$$\frac{\varepsilon = \mu\,\alpha\,.\,\Delta'\,.\,\tau_1 \Rightarrow \tau_2 \qquad \delta' = \delta[\alpha \mapsto \varepsilon] \qquad \Delta; \Gamma \vdash v : \delta'(\tau_1)\ /\ \iota \qquad \Delta \vdash \delta :: \Delta' \qquad \Delta \vdash \varepsilon :: \mathsf{E}}{\Delta; \Gamma \vdash \mathbf{do}\ v : \delta'(\tau_2)\ /\ \varepsilon}$$

$$\frac{\varepsilon = \mu\,\alpha\,.\,\Delta'\,.\,\tau_1 \Rightarrow \tau_2 \qquad \Delta; \Gamma \vdash e : \tau\ /\ \varepsilon \cdot \rho \qquad \Delta; \Gamma, x : \tau \vdash e_r : \tau_r\ /\ \rho \qquad \Delta, \Delta'; \Gamma, x : \tau_1\{\varepsilon\,/\,\alpha\}, r : \tau_2\{\varepsilon\,/\,\alpha\} \to_{\varepsilon \cdot \rho} \tau \vdash e_h : \tau_r\ /\ \rho}{\Delta; \Gamma \vdash \mathbf{handle}\ e\ \{x, r.\ e_h;\ x.\ e_r\} : \tau_r\ /\ \rho}$$

Figure 11 Calculus of shallow handlers with recursive effects.

the treatment of the reset delimiters – see the account of Shan [19] for the complete picture, including shift and control. In our formulation, we keep the same shape of delimiters equipped with "return" clauses that we used with shift_0, and the control operator has an analogous form, while the operational semantics changes in the standard way, by not wrapping the captured continuation with delimiter.

The effect constructor for this calculus is the most complex of the ones we consider. In addition to the quantified variables (three, in this case: two types and a row) and the effect being recursive, like the one introduced for shallow effects, the underlying structure also contains an additional type. This is due to the mismatch between the return type of the continuation and the type of the expression under the control_0 operator: mirroring the case of shallow effects, this is caused by the fact that the captured continuation discards the return clause of the delimiter. Thus, leaving out the polymorphic quantifiers, in an effect $\tau_1 \Rightarrow \tau_2\ /\ \rho$ we have τ_1 as the type of the expression within the delimiter *and* the return type of the captured continuation, while τ_2 as the type of the return clause in the delimiter *and* the type of the expression under the control_0 operator.

As with the previous calculi, we prove type soundness via progress and preservation:

▶ **Theorem 8.** *If* $\cdot; \cdot \vdash e : \tau\ /\ \iota$ *and* $e \to^* e' \not\to$ *in the calculus of* control_0 *and reset, then there exists a value* v *such that* $e' = v$ *and* $\cdot; \cdot \vdash v : \tau\ /\ \iota$.

4.3 Typed Correspondence

The translation from the control_0 calculus to shallow handlers is presented in Figure 13. Note that the translation on expressions is precisely analogous to the case of shift_0 and deep handlers.

$$\sigma ::= \ldots \mid \mu\,\alpha\,.\,\Delta\,.\,\tau \Rightarrow \tau\ /\ \rho$$
$$e ::= \ldots \mid \mathbf{control_0}\ k.\ e \mid \langle e|x.\ e\rangle$$
$$E ::= \ldots \mid \langle E|x.\ e\rangle$$

$$\frac{\Delta, \alpha :: \mathsf{E}, \Delta' \vdash \tau_1 :: \mathsf{T} \quad \Delta, \alpha :: \mathsf{E}, \Delta' \vdash \tau_2 :: \mathsf{T} \quad \Delta, \alpha :: \mathsf{E}, \Delta' \vdash \rho :: \mathsf{R}}{\Delta \vdash \mu\,\alpha\,.\,\Delta'\,.\,\tau_1 \Rightarrow \tau_2\ /\ \rho :: \mathsf{E}}$$

$$\frac{n+1\text{−free}(E)}{n\text{−free}(\langle E|x.\ e\rangle)}$$

$$\frac{0\text{−free}(E)}{\langle E[\mathbf{control_0}\ k.\ e]|x.\ e_r\rangle \mapsto e\{\boldsymbol{\lambda}\, z.\ E[z]\ /\ k\}} \qquad \langle v|x.\ e\rangle \mapsto e\{v\ /\ x\}$$

$$\frac{\begin{array}{c}\varepsilon = \mu\,\alpha\,.\,\Delta'\,.\,\tau_1 \Rightarrow \tau_2\ /\ \rho \qquad \Delta, \Delta'; \Gamma, k : \tau' \to_{\varepsilon\cdot\rho\{\varepsilon\ /\ \alpha\}} \tau_1\{\varepsilon\ /\ \alpha\} \vdash e : \tau_2\{\varepsilon\ /\ \alpha\}\ /\ \rho' \\ \Delta, \Delta' \vdash \rho' <: \rho \qquad \Delta \vdash \tau' :: \mathsf{T} \qquad \Delta \vdash \varepsilon \cdot \rho' :: \mathsf{R}\end{array}}{\Delta; \Gamma \vdash \mathbf{control_0}\ k.\ e : \tau'\ /\ \varepsilon \cdot \rho'}$$

$$\frac{\begin{array}{c}\varepsilon = \mu\,\alpha\,.\,\Delta'\,.\,\tau_1 \Rightarrow \tau_2\ /\ \rho \qquad \delta' = \delta[\alpha \mapsto \varepsilon] \qquad \tau_e = \delta'(\tau_1) \qquad \tau_r = \delta'(\tau_2) \\ \rho_r = \delta'(\rho) \qquad \Delta \vdash \delta :: \Delta' \qquad \Delta; \Gamma \vdash e : \tau_e\ /\ \varepsilon \cdot \rho_r \qquad \Delta; \Gamma, x : \tau_e \vdash e_r : \tau_r\ /\ \rho_r\end{array}}{\Delta; \Gamma \vdash \langle e|x.\ e_r\rangle : \tau_r\ /\ \rho_r}$$

Figure 12 Calculus of control$_0$ with recursive and polymorphic effects.

The new elements appear in the translation of the effect constructor. Like before, an effect $\tau_1 \Rightarrow \tau_2\ /\ \rho$ after translation has the shape $\beta :: \mathsf{T}\,.\,((\beta \to_{\rho_c} \tau_1') \to_{\rho'} \tau_2') \Rightarrow \beta$, where the ρ_c is a translated row of effects of the captured continuation. However, in this case the captured continuation does not contain the delimiter, so ρ_c is a nonempty row that contains the entire translated effect in its head position: this is why we need recursive effects to type-check the translated expressions. Now, the translated effect has the form $\mu\,\alpha\,.\,\beta :: \mathsf{T}\,.\,((\beta \to_{\alpha\cdot\rho'} \tau_1') \to_{\rho'} \tau_2') \Rightarrow \beta$. The translation of polymorphic variables of an effect is analogous to what we have seen in the case of "deep" control, with variables quantifiers of the effect turning into universal quantifiers in proper types.

We show the following theorems, which say that the translation preserves both types and semantics.

▶ **Theorem 9.** *If $\Delta; \Gamma \vdash e : \tau\ /\ \rho$ in the calculus of control$_0$, then $\Delta; [\![\Gamma]\!]^{\mathrm{SH}} \vdash [\![e]\!]^{\mathrm{SH}} : [\![\tau]\!]^{\mathrm{DH}}\ /\ [\![\rho]\!]^{\mathrm{SH}}$ in the calculus of shallow handlers.*

▶ **Theorem 10.** *If $e \to e'$ in the calculus of control$_0$, then in the calculus of shallow handlers we have $[\![e]\!]^{\mathrm{SH}} \to^{+} [\![e']\!]^{\mathrm{SH}}$.*

The translation from shallow handlers to the control$_0$ calculus is shown in Figure 14. The translation for expressions is almost the same as for the deep case with one minor difference: the continuation captured by $\mathbf{control_0}$ does not contain a delimiter, so it can be directly passed as a resumption to a handler code. The translation also requires recursive effects, for the same reasons as the translation in the other direction.

The translation preserves types and reduction semantics, as we show in the following theorems.

$$[\![\mathbf{control_0}\ k.\ e]\!]^{\mathrm{SH}} \triangleq \mathbf{do}\ \boldsymbol{\lambda} k.\ [\![e]\!]^{\mathrm{SH}}$$
$$[\![\langle e | x.\ e_r \rangle]\!]^{\mathrm{SH}} \triangleq \mathbf{handle}\ [\![e]\!]^{\mathrm{SH}}\ \{x, r.\ x\ r;\ x.\ [\![e_r]\!]^{\mathrm{SH}}\}$$

$$[\![\mu\,\alpha\,.\,\Delta\,.\,\tau_1 \Rightarrow \tau_2\ /\ \rho]\!]^{\mathrm{SH}} \triangleq \mu\,\alpha\,.\,\beta :: \mathsf{T}\,.\,\big(\forall\,\Delta\,.\,(\beta \to_{\alpha\cdot[\![\rho]\!]^{\mathrm{SH}}} [\![\tau_1]\!]^{\mathrm{SH}}) \to_{[\![\rho]\!]^{\mathrm{SH}}} [\![\tau_2]\!]^{\mathrm{SH}}\big) \Rightarrow \beta$$

Figure 13 The control$_0$ calculus via shallow handlers.

$$[\![\mathbf{do}\ v]\!]^{\mathrm{SD}} \triangleq \mathbf{control_0}\ k.\ \boldsymbol{\lambda} h.\ h\ [\![v]\!]^{\mathrm{SD}}\ k$$
$$[\![\mathbf{handle}\ e\ \{x, r.\ e_h;\ x.\ e_r\}]\!]^{\mathrm{SD}} \triangleq \langle [\![e]\!]^{\mathrm{SD}} | x.\ \boldsymbol{\lambda} h.\ [\![e_r]\!]^{\mathrm{SD}} \rangle\ (\boldsymbol{\lambda}\, x, r.\ [\![e_h]\!]^{\mathrm{SD}})$$

$$[\![\mu\,\alpha\,.\,\Delta\,.\,\tau_1 \Rightarrow \tau_2]\!]^{\mathrm{SD}} \triangleq$$
$$\mu\,\alpha\,.\,\beta_1 :: \mathsf{T}, \beta_2 :: \mathsf{T}, \gamma :: \mathsf{R}\,.\,\beta_1 \Rightarrow \big(\forall\,\Delta\,.\,[\![\tau_1]\!]^{\mathrm{SD}} \to ([\![\tau_2]\!]^{\mathrm{SD}} \to_{\alpha\cdot\gamma} \beta_1) \to_\gamma \beta_2\big) \to_\gamma \beta_2\ /\ \gamma$$

Figure 14 Shallow handlers as shallow delimited control.

▶ **Theorem 11.** *If* $\Delta; \Gamma \vdash e : \tau\ /\ \rho$ *in the calculus of shallow handlers, then* $\Delta; [\![\Gamma]\!]^{\mathrm{SD}} \vdash [\![e]\!]^{\mathrm{SD}} : [\![\tau]\!]^{\mathrm{SD}}\ /\ [\![\rho]\!]^{\mathrm{SD}}$ *in the calculus of control*$_0$.

▶ **Theorem 12.** *If* $e \to e'$ *in the calculus of shallow handlers, then in the calculus of control*$_0$ *we have* $[\![e]\!]^{\mathrm{SD}} \to^+ [\![e']\!]^{\mathrm{SD}}$.

Since we do not modify the captured continuation before passing it as a resumption, we are able to prove Theorem 12 for a reduction in evaluation contexts instead of general contexts.

5 Discussion and Further Work

We have shown how the untyped correspondence between deep effect handlers and shift$_0$, known from prior work can be extended to the typed setting, given an appropriately expressive type system. In the process, we have identified a novel type system for the shift$_0$/reset calculus, in which the shift expressions are parametric in the type and effect. To our knowledge, such a system has not been considered before in the extensive literature on delimited control operators, although further work is necessary to explore the relation of our system to those presently found in literature – one aspect to consider would certainly be answer-type modification.

At the same time, we believe it is useful to contrast the parametric shift$_0$, which remains rather difficult to grasp, to the apparently natural move to polymorphic effect handlers, which indeed have been considered before. We feel that the fact that the effect handlers give an interpretation to the control effect at the delimiter, rather than at the capture point – which causes the translations studied in this paper to have to "invert" the direction of control – may be the reason behind the recent surge in popularity of effect handlers as the main control abstraction provided by a language, while delimited control operators have been, throughout their history, somewhat of a niche interest.

We have also adapted the translations, for both terms and types, to the shallow handlers and a control$_0$/reset calculus. While the translation for terms is barely different from the case for deep handlers and shift$_0$, the type systems that are required in order to achieve inter-expressibility evidence the folklore notion that shallow handlers are "case-like" (while the deep variant behaves "fold-like"): thus the need for explicit recursion within the effects.

Finally, since we work within a common type-and-effect system with effect rows, we managed to extend the calculi with both a simple notion of subeffecting and a variant of Biernacki *et al.*'s lift operator that allows "masking" effects at the front of the effect row without any effect on the complexity of the translation.

While this work resolves a conjecture of Forster *et al.* [8], there remains a wide array of topics for future work. The most obvious line of work that we chose not to pursue at this point is answer-type modification [1, 12], which is a common feature of type systems for delimited control operators. Whether it can be reconciled with the polymorphic effects of our calculi remains to be investigated: if so, it is certainly interesting to see how it would translate to the effect handler calculi. Relating our type systems to the monadic representation is another aspect that would require further attention.

References

1 Kenichi Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 22(3):275–291, 2009. doi:10.1007/s10990-009-9049-5.

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

3 Olivier Danvy. An analytical approach to programs as data objects. *DSc thesis, Department of Computer Science, Aarhus University*, 11, 2006.

4 Olivier Danvy and Andrzej Filinski. Abstracting Control. In *LISP and Functional Programming*, pages 151–160, 1990. doi:10.1145/91556.91622.

5 Matthias Felleisen. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 180–190, 1988. doi:10.1145/73560.73576.

6 Matthias Felleisen. On the Expressive Power of Programming Languages. *Sci. Comput. Program.*, 17(1-3):35–75, 1991. doi:10.1016/0167-6423(91)90036-W.

7 Matthias Felleisen and Daniel P. Friedman. A Reduction Semantics for Imperative Higher-Order Languages. In *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*, pages 206–223, 1987. doi:10.1007/3-540-17945-3_12.

8 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *PACMPL*, 1(ICFP):13:1–13:29, 2017. doi:10.1145/3110257.

9 Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016.

10 Daniel Hillerström and Sam Lindley. Shallow Effect Handlers. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, pages 415–435, 2018. doi:10.1007/978-3-030-02768-1_22.

11 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158, 2013. doi:10.1145/2500365.2500590.

12 Ikuo Kobori, Yukiyoshi Kameyama, and Oleg Kiselyov. Answer-Type Modification without Tears: Prompt-Passing Style Translation for Typed Delimited-Control Operators. In *Proceedings of the Workshop on Continuations, WoC 2016, London, UK, April 12th 2015.*, pages 36–52, 2015. doi:10.4204/EPTCS.212.3.

13 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499, 2017. URL: http://dl.acm.org/citation.cfm?id=3009872.

14 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 91–102, 2012. doi:10.1145/2103786.2103798.

15 John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 47–57, 1988. doi:10.1145/73560.73564.

16 Marek Materzok and Dariusz Biernacki. A Dynamic Interpretation of the CPS Hierarchy. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, pages 296–311, 2012. doi:10.1007/978-3-642-35182-2_21.

17 Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23, 1989. doi:10.1109/LICS.1989.39155.

18 Gordon D. Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.

19 Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007. doi:10.1007/s10990-007-9010-4.

20 Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Inf. Comput.*, 111(2):245–296, 1994. doi:10.1006/inco.1994.1046.

21 Philip Wadler. Comprehending Monads. In *LISP and Functional Programming*, pages 61–78, 1990. doi:10.1145/91556.91592.

22 Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

23 Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235, 2003. doi:10.1145/640128.604150.