Understanding Algebraic Effect Handlers via Delimited Control Operators

Youyou Cong[1][0000−0003−2315−6182] and Kenichi Asai[2]

[1] Tokyo Institute of Technology
[2] Ochanomizu University
cong@c.titech.ac.jp
asai@is.ocha.ac.jp
https://prg.is.titech.ac.jp/people/cong/
http://pllab.is.ocha.ac.jp/~asai/

Abstract. Algebraic effects and handlers are a powerful and convenient abstraction for user-defined effects. We aim to understand effect handlers through the lens of control operators, a similar but more well-studied tool for expressing effects. In this paper, we establish two program transformations and a type system for effect handlers, all by reusing the existing results about control operators and their relationship to effect handlers.

Keywords: Algebraic effects and handlers · Delimited control operators · Macro translation · CPS translation · Type systems.

1 Introduction

Algebraic effects [36] and handlers [37] have become an essential element of a programmer's toolbox. Effect handlers provide a convenient interface for defining and composing effects. They also enable concise implementation of sophisticated behavior by giving the programmer access to continuations.

Over the past decade, researchers have been actively studying the theory of effect handlers. As an outcome of these studies, we have obtained various program transformations for effect handlers, which can be used to compile effect handlers into plain λ-terms [17,40,42]. There are also a variety of type systems for effect handlers, in which effects are represented as sets [3], rows [14], or capabilities [8].

We continue the study of effect handlers, but from a different point of view. Instead of directly developing the theory of effect handlers, we *derive* it from the theory of delimited control operators [10,18,29,11]. Control operators have a longer history than effect handlers, and their theory is closely connected to that of effect handlers [13,34]. We aim to improve the understanding of effect handlers by exploiting the existing results about control operators, as well as the connection between effect handlers and control operators.

In this paper, we discuss a variant of control operators known as `shift0` and dollar [30], and a variant of effect handlers that are called *deep* handlers [19]. Our goal is to answer the following research questions.

- The dollar operator extends the more traditional `reset0` operator with a return clause, and this extension is known to cause no change in the expressiveness [30]. Does this result also apply to effect handlers with and without a return clause?
- The `shift0` and dollar operators are associated with a CPS translation that can be viewed as a definitional interpreter. What CPS translation can we derive for deep effect handlers from the CPS translation for `shift0` and dollar?
- The typing of `shift0` and dollar is directed by their CPS translation [9,18,29], rather than their direct-style form. What type system can we derive for deep effect handlers using the CPS approach?

Note that we do not intend to produce results that are particularly surprising, nor do we aim to develop something that has immediate practical use. Instead, we would like to explore the possibility of transferring knowledge between different control facilities. This method can be applied for various purposes, and we feel that it is especially effective when used in a pedagogical context.

In the rest of this paper, we first define a calculus of `shift0`/dollar (Section 2) and another calculus of deep effect handlers (Section 3). We next answer the three questions one by one (Sections 4 to 6). We then discuss related work (Section 7) and conclude with future perspectives (Section 8).

2 λ_{S_0}: A Calculus of Shift0 and Dollar

As a calculus of control operators, we consider a minor variation of Forster et al.'s calculus of `shift0` and dollar [13], which we call λ_{S_0}. In Figure 1, we present the syntax and reduction rules of λ_{S_0}. The calculus differs from that of Forster et al. in that it is formalized as a fine-grain call-by-value calculus [26] instead of call-by-push-value [25]. This means (i) functions are classified as values; and (ii) computations must be explicitly sequenced using the `let` expression. The fine-grain syntax simplifies the CPS translation and type system developed in later sections.

Among the control constructs, $\mathcal{S}_0 k.\, M$ (pronounced "shift") captures a continuation surrounding itself. The other construct $\langle M \mid x.N \rangle$ (pronounced "dollar") computes the main computation M in a delimited context that ends with the continuation N[1].

There are two reduction rules for the control constructs. If the main computation of dollar evaluates to a value V, the whole expression evaluates to the ending continuation N with V bound to x (rule ($\beta_{\$}$)). If the main computation of dollar evaluates to $F[\mathcal{S}_0 k.\, M]$, where F is a pure evaluation context that has

[1] The dollar operator was originally proposed by Kiselyov and Shan [20]. In their formalization, the construct takes the form $N \,\$\, M$, where M is the main computation and N is an arbitrary expression representing an ending continuation. We use the bracket notation of Forster et al., restricting N to be an abstraction $\lambda x.\, N$. This makes it easier to compare dollar with effect handlers.

Syntax

$$
\begin{array}{lll}
V, W ::= x \mid \lambda x.\, M & & \text{Values} \\
M, N ::= \texttt{return}\ V \mid V\ V \mid \texttt{let}\ x = M\ \texttt{in}\ M \mid \mathcal{S}_0 k.\, M \mid \langle M \mid x.M \rangle & & \text{Computations}
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{lll}
E ::= [] \mid \texttt{let}\ x = E\ \texttt{in}\ M \mid \langle E \mid x.M \rangle & & \text{General Contexts} \\
F ::= [] \mid \texttt{let}\ x = F\ \texttt{in}\ M & & \text{Pure Contexts}
\end{array}
$$

Reduction

$$
\begin{array}{rll}
E[(\lambda x.\, M)\ V] \rightsquigarrow E[M[V/x]] & & (\beta_v) \\
E[\texttt{let}\ x = \texttt{return}\ V\ \texttt{in}\ M] \rightsquigarrow E[M[V/x]] & & (\zeta_v) \\
E[\langle \texttt{return}\ V \mid x.M \rangle] \rightsquigarrow E[M[V/x]] & & (\beta_{\$}) \\
E[\langle F[\mathcal{S}_0 k.\, M] \mid x.N \rangle] \rightsquigarrow E[M[\lambda y.\, \langle F[\texttt{return}\ y] \mid x.N \rangle / k]] & & (\beta_{\mathcal{S}_0})
\end{array}
$$

Fig. 1. Syntax and Reduction Rules of λ_{S_0}

no dollar surrounding a hole, the whole expression evaluates to M with k being the captured continuation $\lambda y.\, \langle F[\texttt{return}\ y] \mid x.N \rangle$ (rule $(\beta_{\mathcal{S}_0})$). Notice that the continuation includes the dollar construct that was originally surrounding the `shift0` operator. This design is shared with the `shift` operator of Danvy and Filinski [10]. Notice next that the body of `shift0` is evaluated without being surrounded by the original dollar. This differentiates `shift0` from `shift`, and allows `shift0` to capture a *meta-context*[2], i.e., a context that resides outside of the lexically closest dollar.

3 λ_h: A Calculus of Effect Handlers

As a calculus of effect handlers, we consider a restricted variant of Hillerström et al.'s calculus of deep handlers [17], which we call λ_h. In Figure 2, we present the syntax and reduction rules of λ_h. The calculus differs from Hillerström et al.'s in that it features unlabeled operations. This means handlers in λ_h can only handle a single operation. The restriction helps us concentrate on the connection to the λ_{S_0} calculus.

[2] To see what a meta-context is, consider the following program:

$$\langle 2 * \langle 1 + \mathcal{S}_0 k_1.\, \mathcal{S}_0 k_2.\, k_2\ 3 \mid x.\texttt{return}\ x \rangle \mid x.\texttt{return}\ x \rangle \rightsquigarrow^* 6$$

The first `shift0` operator captures the delimited context $\langle 1 + [] \mid x.\texttt{return}\ x \rangle$, whereas the second one captures the meta-context $\langle 2 * [] \mid x.\texttt{return}\ x \rangle$ surrounding the innermost dollar operator.

Syntax

$$
\begin{array}{lll}
V, W ::= x \mid \lambda x.\, M & & \text{Values} \\
M, N ::= \mathtt{return}\ V \mid V\ V \mid \mathtt{let}\ x = M\ \mathtt{in}\ M & & \text{Computations} \\
\quad \mid \mathtt{do}\ V \mid \mathtt{handle}\ M\ \mathtt{with}\ \{x.\, M;\, x, k.\, M\} & &
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{lll}
E ::= [] \mid \mathtt{let}\ x = E\ \mathtt{in}\ M \mid \mathtt{handle}\ E\ \mathtt{with}\ \{x.\, M;\, x, k.\, M\} & & \text{General Contexts} \\
F ::= [] \mid \mathtt{let}\ x = F\ \mathtt{in}\ M & & \text{Pure Contexts}
\end{array}
$$

Reduction

$$
\begin{array}{rll}
E[(\lambda x.\, M)\ V] \rightsquigarrow E[M[V/x]] & & (\beta_v) \\
E[\mathtt{let}\ x = \mathtt{return}\ V\ \mathtt{in}\ M] \rightsquigarrow E[M[V/x]] & & (\zeta_v) \\
E[\mathtt{handle}\ \mathtt{return}\ V\ \mathtt{with}\ \{x.\, M_r;\, x, k.\, M_h\}] \rightsquigarrow E[M_r[V/x]] & & (\beta_h) \\
E[\mathtt{handle}\ F[\mathtt{do}\ V]\ \mathtt{with}\ \{x.\, M_r;\, x, k.\, M_h\}] \rightsquigarrow E[M_h[V/x, f/k]] & & (\beta_{do}) \\
\quad \text{where } f = \lambda y.\, \mathtt{handle}\ F[\mathtt{return}\ y]\ \mathtt{with}\ \{x.\, M_r;\, x, k.\, M_h\} & &
\end{array}
$$

Fig. 2. Syntax and Reduction Rules of λ_h

Among the effect constructs, $\mathtt{do}\ V$ performs an operation with argument V. The other construct $\mathtt{handle}\ M\ \mathtt{with}\ \{x.\, M_r;\, x, k.\, M_h\}$ computes the main computation M in a delimited context, and handles the result of M using the return clause M_r and the operation clause M_h.

There are again two reduction rules for the effect constructs. If the main computation of a handler evaluates to a value V, the whole expression evaluates to the return clause N with V bound to x (rule (β_h)). If the main computation evaluates to $F[\mathtt{do}\ V]$, where F is a pure evaluation context that has no handler surrounding a hole, the whole expression evaluates to the operation clause M_h, with x being V and k being the captured continuation (often called "resumption" in the effect handlers literature) $\lambda y.\, \mathtt{handle}\ F[\mathtt{return}\ y]\ \mathtt{with}\ \{x.\, M_r;\, x, k.\, M_h\}$ (rule (β_{do})). Notice that the continuation includes the handler that was originally surrounding the operation. This design is shared with `shift0`, and characterizes handlers in λ_h as deep ones [19]. Notice next that the operation clause is evaluated without being surrounded by the original handler. This is another similarity to `shift0`, and allows handlers to capture a metacontext.

4 Adding and Removing the Return Clause

The dollar construct $\langle M \mid x.N \rangle$ in λ_{S_0} is a generalization of the "reset" construct $\langle M \rangle$, which is more commonly found in the continuations literature. The reset construct does not have an ending continuation; it simply evaluates the body M in an empty context. As shown by Materzok and Biernacki [28], the dollar and reset constructs can macro-express [12] each other. That is, there is a pair of local translations, called *macro translations*, that add and remove the ending continuation while preserving the meaning of the program.

It is easy to see that the return clause of an effect handler plays a similar role to the ending continuation of dollar. Thus, we can naturally consider a variant of effect handlers without the return clause. Such handlers are not uncommon in formalizations [40,42,44] as they simplify the reduction and typing rules. Now the reader might wonder: Does the existence of the return clause affects the expressiveness of an effect handler calculus?

In this section, we define macro translations between handlers with and without the return clause. Among different ways of removing the return clause, we take a relatively intuitive approach that relies on a labeling mechanism. In what follows, we review the macro translations between dollar and reset (Section 4.1), then adapt the translations to effect handlers (Section 4.2), and lastly prove the correctness of the translations (Section 4.3).

4.1 Translating Between Dollar and Reset0

Materzok and Biernacki [30] define the macro translations $[\![\_]\!]_m$ between dollar and `reset0` as follows[3].

From dollar to reset

$$[\![\langle M \mid x.N \rangle]\!]_m = \langle \texttt{let}\ y = [\![M]\!]_m\ \texttt{in}\ \mathcal{S}_0 z.\, (\lambda x.\, [\![N]\!]_m)\ y \rangle$$

From reset to dollar

$$[\![\langle M \rangle]\!]_m = \langle [\![M]\!]_m \mid x.\texttt{return}\ x \rangle$$

The translation from reset to dollar is straightforward: it simply adds a trivial ending continuation. The translation from dollar to reset is more involved: it wraps the computation $[\![M]\!]_m$ around a reset, and inserts a `shift0` to remove the surrounding reset. The removal of reset is necessary for preservation of meaning (the return clause should not be evaluated under the original handler), and is realized by discarding the captured continuation z.

Note that the translation of other constructs is defined homomorphically. For instance, we have $[\![\lambda x.\, M]\!]_m = \lambda x.\, [\![M]\!]_m$.

[3] The macro translation is originally defined as:

$$(\lambda k.\, \langle (\lambda x.\, \mathcal{S}_0 z.\, k\ x)\ [\![M]\!]_m \rangle)\ [\![N]\!]_m$$

We adapted the translation by sequencing the application, and by incorporating the fact that N is always a λ-abstraction.

Syntax

$$
\begin{array}{lll}
V, W ::= x \mid \lambda x.\, M \mid \langle l, V\rangle & & \text{Values} \\
M, N ::= \mathtt{return}\ V \mid V\ V \mid \mathtt{let}\ x = M\ \mathtt{in}\ M \mid \mathtt{do}\ V & & \text{Computations} \\
\quad \mid \mathtt{case}_p\ V\ \mathtt{of}\ \{\langle l, x\rangle \to M\} \mid \mathtt{case}_l\ l\ \mathtt{of}\ \{\mathtt{ret} \to M; \mathtt{op} \to M\} & & \\
\quad \mid \mathtt{handle}\ M\ \mathtt{with}\ \{x, k.\, M\} & & \\
l ::= \mathtt{op} \mid \mathtt{ret} & & \text{Labels}
\end{array}
$$

Reduction

$$
\begin{array}{rll}
E[(\lambda x.\, M)\ V] & \rightsquigarrow E[M[V/x]] & (\beta_v) \\
E[\mathtt{let}\ x = \mathtt{return}\ V\ \mathtt{in}\ M] & \rightsquigarrow E[M[V/x]] & (\zeta_v) \\
E[\mathtt{case}_p\ \langle l, V\rangle\ \mathtt{of}\ \{\langle l', x\rangle \to M_r\}] & \rightsquigarrow E[M_r[l/l', V/x]] & (\iota_p) \\
E[\mathtt{case}_l\ \mathtt{ret}\ \mathtt{of}\ \{\mathtt{ret} \to M_r; \mathtt{op} \to M_h\}] & \rightsquigarrow E[M_r] & (\iota_{ret}) \\
E[\mathtt{case}_l\ \mathtt{op}\ \mathtt{of}\ \{\mathtt{ret} \to M_r; \mathtt{op} \to M_h\}] & \rightsquigarrow E[M_h] & (\iota_{op}) \\
E[\mathtt{handle}\ \mathtt{return}\ V\ \mathtt{with}\ \{x, k.\, M_h\}] & \rightsquigarrow E[\mathtt{return}\ V] & (\beta_h) \\
E[\mathtt{handle}\ F[\mathtt{do}\ V]\ \mathtt{with}\ \{x, k.\, M_h\}] & \rightsquigarrow E[M[V/x, f/k]] & (\beta_{do}) \\
 & \text{where } f = \lambda y.\, \mathtt{handle}\ F[y]\ \mathtt{with}\ \{x, k.\, M_h\} &
\end{array}
$$

Fig. 3. λ_h^-: A Calculus of Effect Handlers without the Return Clause

4.2 Translating Between Handlers with and without Return Clause

Guided by the translations between dollar and reset, we define macro translations between handlers with and without the return clause. We can easily imagine that adding the return clause is simple. To remove the return clause, we must somehow implement the removal of a handler. In a calculus of effect handlers, any non-local control is triggered by an operation call. This means we need to make an operation call when we wish to remove a handler. Unlike `shift0`, however, an operation call does not have an interpretation on its own. This means we need to implement the removing behavior in the surrounding handler, while distinguishing the "return operation" from regular operations.

In Figure 3, we define a calculus of effect handlers without the return clause, which we call λ_h^-. The calculus has labels l and pairs $\langle l, V\rangle$, allowing us to represent the return and regular operations as $\mathtt{do}\ \langle \mathtt{ret}, V\rangle$ and $\mathtt{do}\ \langle \mathtt{op}, V\rangle$, respectively. The calculus also has pattern matching constructs, with which we can interpret the two kinds of operations differently in a handler. Note that these facilities are introduced only for the translation purpose. That is, we assume that the user can only program with unlabeled operations; they do not have access to the shaded constructs in Figure 3.

We now define the macro translations between λ_h and λ_h^-.

From λ_h to λ_h^-

$$\begin{aligned}
[\![\texttt{do}\ V]\!]_m &= \texttt{do}\ \langle \texttt{op}, [\![V]\!]_m \rangle \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m &= \texttt{handle}\ (\texttt{let}\ y = [\![M]\!]_m \\
&\qquad\qquad \texttt{in do}\ \langle \texttt{ret}, y \rangle) \\
&\quad \texttt{with}\ \{p, k \to \texttt{case}_p\ p\ \texttt{of}\ \{\langle l, x \rangle \to \\
&\quad \{\texttt{case}_l\ l\ \texttt{of} \\
&\quad \{\texttt{ret} \to [\![M_r]\!]_m;\ \texttt{op} \to [\![M_h]\!]_m\}\}\}\}
\end{aligned}$$

From user fragment of λ_h^- to λ_h

$$\begin{aligned}
[\![\texttt{do}\ V]\!]_m &= \texttt{do}\ [\![V]\!]_m \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x, k.\, M_h\}]\!]_m &= \texttt{handle}\ [\![M]\!]_m \\
&\quad \texttt{with}\ \{x.\, \texttt{return}\ x;\ x, k.\, [\![M_h]\!]_m\}
\end{aligned}$$

The first translation[4][5] attaches a label `op` to regular operations and simulates the return clause by performing a `ret` operation, which removes the surrounding handler by discarding the continuation k. The second translation is fairly easy.

4.3 Correctness

The macro translations defined above preserve the meaning of programs. We state this property as the following theorem.

Theorem 1 (Correctness of Macro Translations). *Let $=$ be the least congruence relation that includes the reduction $\rightsquigarrow$ in λ_h and λ_h^-.*

1. *If $M = N$ in λ_h, then $[\![M]\!]_m = [\![N]\!]_m$ in λ_h^-.*
2. *If $M = N$ in the user fragment of λ_h^- (i.e., the fragment consisting of non-shaded constructs), then $[\![M]\!]_m = [\![N]\!]_m$ in λ_h.*

Proof. By cases on the reduction relation $M \rightsquigarrow N$.

[4] If we wish to apply these translations to a typed language, we would need to slightly modify the definition. The reason is that, the translation of regular handlers yields a single pattern variable x representing the arguments of the `ret` and `op` operations, which have different types in general.

[5] An anonymous reviewer suggests a different translation that does not use labels:

$$\begin{aligned}
&[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m \\
&= (\texttt{handle}\ (\texttt{let}\ y = [\![M]\!]_m\ \texttt{in return}\ \lambda_.\, (\lambda x.\, [\![M_r]\!]_m)\ y) \\
&\quad \texttt{with}\ \{x, k'.\, \texttt{return}\ \lambda_.\, [\![M_h]\!]_m[\lambda z.\, k'\ z\ 0/k]\}) \\
&\quad 0
\end{aligned}$$

where $M\ V$ is a shorthand for $\texttt{let}\ x = M\ \texttt{in}\ x\ V$ and 0 can be replaced by any constant. The translation implements the return clause by creating thunks (to exit from a handler) and passing around a dummy argument (to trigger computation). This eliminates the need for labels, but in our view, it also makes the translation slightly harder to understand.

5 Deriving a CPS Translation

The classical way of specifying the semantics of control operators is to give a translation into continuation-passing style (CPS) [35], which converts control operators into plain lambda terms. In the case of `shift0` and dollar (or reset), there exist several variants of CPS translation, differing in the representation of continuations [4,11,41,29,30].

Compared to control operators, the semantics of effect handlers seems to be less tightly tied to the CPS translation. In fact, the CPS translation of effect handlers has not been formally studied until recently [17,16]. This gives rise to a question: What would we obtain if we derive the CPS translation of effect handlers from that of control operators, which is considered definitional?

In this section, we derive the CPS translation of effect handlers by composing the following translations.

1. The macro translation from effect handlers to `shift0`/dollar [13]
2. The CPS translation of `shift0`/dollar [30]

We show that, by avoiding "administrative" constructs in the macro translation, and by being selective in the CPS translation, we obtain the same CPS translation as the unoptimized[6] translation given by Hillerström et al. [17]. In what follows, we review the existing CPS translations of `shift0`/dollar and effect handlers (Sections 5.1 and 5.2), as well as the macro translation from effect handlers to `shift0`/dollar (Section 5.3). We then compose the first and third translations to obtain the second translation (Section 5.4), thus formally relating the CPS translations of `shift0`/dollar and effect handlers.

5.1 CPS Translation of Shift0 and Dollar

In Figure 4, we present the CPS translation from λ_{S_0} to the plain λ-calculus (with no value vs. computation distinction). The definition is adopted from Hillerström et al. [17] (for the λ-terms fragment) and Materzok and Biernacki [30] (for `shift0`/dollar). We have two mutually-defined translations $[\![\_]\!]_{c'}$ and $[\![\_]\!]_c$, taking care of values and computations, respectively. The former yields a direct-style value in the λ-calculus, whereas the latter yields a continuation-taking function. To go through the cases of the control operators, the translation turns a `shift0` construct into a λ-abstraction, and a dollar construct into an application of the main computation $[\![M]\!]_c$ to an ending continuation $\lambda x.\, [\![N]\!]_c$[7].

5.2 CPS Translation of Effect Handlers

In Figure 5, we present the CPS translation from λ_h to the plain λ-calculus. The definition is adopted from Hillerström et al. [17]. We again have separate

[6] By "unoptimized translation", we mean the first-order translation in Figure 5 of Hillerström et al. [17].

[7] The CPS translation of the original dollar operator $N\,\$\,M$ is defined as $\lambda k.\, [\![N]\!]_c\ (\lambda f.\, [\![M]\!]_c\ f\ k)$.

$$
\begin{aligned}
[\![x]\!]_{c'} &= x \\
[\![\lambda x.\, M]\!]_{c'} &= \lambda x.\, [\![M]\!]_c \\
[\![\texttt{return}\ V]\!]_c &= \lambda k.\, k\ [\![V]\!]_{c'} \\
[\![V\ W]\!]_c &= [\![V]\!]_{c'}\ [\![W]\!]_{c'} \\
[\![\texttt{let}\ x = M\ \texttt{in}\ N]\!]_c &= \lambda k.\, [\![M]\!]_c\ (\lambda x.\, [\![N]\!]_c\ k) \\
[\![\mathcal{S}_0 k.\, M]\!]_c &= \lambda k.\, [\![M]\!]_c \\
[\![\langle M \mid x.N \rangle]\!]_c &= [\![M]\!]_c\ (\lambda x.\, [\![N]\!]_c)
\end{aligned}
$$

Fig. 4. CPS Translation of λ_{S_0} [17,29]

$$
\begin{aligned}
[\![\texttt{do}\ V]\!]_c &= \lambda k.\, \lambda h.\, h\ [\![V]\!]_{c'}\ (\lambda x.\, k\ x\ h) \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\, x, k.\, M_h\}]\!]_c &= [\![M]\!]_c\ (\lambda x.\, \lambda h.\, [\![M_r]\!]_c)\ (\lambda x.\, \lambda k.\, [\![M_h]\!]_c)
\end{aligned}
$$

Fig. 5. CPS Translation of λ_h [17] (cases for λ-terms are the same as Figure 4)

translations for values and computations. Among them, the computation translation yields a function that takes in two continuations: a pure continuation k for returning a value, and an effect continuation h for handling an operation. To go through the interesting cases, the translation of an operation calls the effect continuation h representing the interpretation of that operation[8]. Notice that the resumption $\lambda x.\, k\ x\ h$ passed to h includes h itself, reflecting the deep nature of handlers. The translation of a handler sets the two continuations for the handled computation $[\![M]\!]_c$. Other cases are the same as the translation of λ_{S_0}. In particular, the translation of the `return` and `let` expressions requires only one continuation argument, as the effect continuation can be removed by η-reduction.

5.3 Macro Translation from λ_h to λ_{S_0}

Having seen the CPS translations, we look at the macro translation from λ_h to λ_{S_0} (Figure 6), which is adapted from Forster et al. [13]. Roughly speaking, the translation converts operations into `shift0` and handlers into dollar. Technically, it introduces an effect-continuation-passing mechanism to make the operation clause accessible to the `shift0` operator. The result of the translation is somewhat verbose due to the presence of the `return` and `let` expressions; these are necessary for ensuring that the translation produces a well-formed expression in fine-grain call-by-value. The original translation of Forster et al. (Figure 7) is simpler, as it is defined on call-by-push-value (where the opera-

[8] The original translation of Hillerström et al. [17] packages the two arguments to h into a tuple and associates the tuple with an operation label.

$$
\begin{aligned}
[\![\texttt{do}\ V]\!]_m &= \mathcal{S}_0 k.\, \texttt{return}\ (\lambda h.\, \texttt{let}\ v_1 = h\ [\![V]\!]_m \\
&\qquad\qquad \texttt{in}\ v_1\ (\lambda x.\, \texttt{let}\ v_2 = k\ x \\
&\qquad\qquad\qquad \texttt{in}\ v_2\ h)) \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m &= \texttt{let}\ v = \langle [\![M]\!]_m \mid x.\texttt{return}\ \lambda h.\, [\![M_r]\!]_m \rangle \\
&\qquad\qquad \texttt{in}\ v\ (\lambda x.\, \texttt{return}\ (\lambda k.\, [\![M_h]\!]_m))
\end{aligned}
$$

Fig. 6. Macro Translation from λ_h to λ_{S_0}

$$
\begin{aligned}
[\![\texttt{do}\ V]\!]_m &= \mathcal{S}_0 k.\, (\lambda h.\, h\ [\![V]\!]_m\ (\lambda x.\, k\ x\ h)) \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m &= \langle [\![M]\!]_m \mid x.\lambda h.\, [\![M_r]\!]_m \rangle\ (\lambda x.\, \lambda k.\, [\![M_h]\!]_m)
\end{aligned}
$$

Fig. 7. Simpler Macro Translation from λ_h to λ_{S_0} [13]

tor of an application may be a computation). Note that, in both versions, the translation of λ-terms is defined homomorphically.

5.4 Composing Macro and CPS Translations

Now, let us derive a CPS translation of λ_h by composing the macro translation on λ_h and the CPS translation on λ_{S_0}.

Operations We begin with the case of operations. Below is the naïve composition of the macro and CPS translations.

$$
\begin{aligned}
&[\![[\![\texttt{do}\ V]\!]_m]\!]_c \\
&= [\![\mathcal{S}_0 k.\, \texttt{return}\ (\lambda h.\, \texttt{let}\ v_1 = h\ [\![V]\!]_m\ \texttt{in}\ v_1\ (\lambda x.\, \texttt{let}\ v_2 = k\ x\ \texttt{in}\ v_2\ h)]\!]_c \\
&= \lambda k.\, \lambda k_1.\, k_1\ (\lambda h.\, \lambda k_2.\, h\ [\![[\![V]\!]_m]\!]_{c'}\ (\lambda v_1.\, v_1\ (\lambda x.\, \lambda k_3.\, k\ x\ (\lambda v_2.\, v_2\ h\ k_3))\ k_2))
\end{aligned}
$$

The result has four continuation variables: k, k_1, k_2, and k_3. Among them, the latter three continuations come from the `return` and `let` expressions introduced by the macro translation. As we mentioned before, these `return` and `let` are only necessary for the well-formedness of macro-translated expressions. In other words, they do not have any computational content. What this implies is that, when our ultimate goal is to convert effect handlers into plain λ-terms, we can use a simpler macro translation that does not yield these administrative `return` and `let`. This simpler translation is exactly the original translation of Forster et al. [13].

$$
\begin{aligned}
&[\![[\![\texttt{do}\ V]\!]_m]\!]_c \\
&= [\![\mathcal{S}_0 k.\, \lambda h.\, h\ [\![V]\!]_m\ (\lambda x.\, k\ x\ h)]\!]_c
\end{aligned}
$$

The result of the simpler macro translation is not a proper expression in fine-grain CBV: the `shift0` operator has a value body, and the body has applications of a function to two values. This prevents us from directly applying the CPS translation in Figure 4. However, we can still reuse the CPS translation, under the principle "do not introduce a continuation when it is not necessary". In the case of the above expression, we simply apply the value translation to the body of `shift0` and the arguments of the applications. Such a principle reminds us of *selective* CPS translations in the literature [31,39,2].

$$\begin{aligned}&[\![\mathcal{S}_0 k.\, \lambda h.\, h\ [\![V]\!]_m\ (\lambda x.\, k\ x\ h)]\!]_c\\ =\ &\lambda k.\, \lambda h.\, h\ [\![[\![V]\!]_m]\!]_{c'}\ (\lambda x.\, k\ x\ h)\end{aligned}$$

The result of composition is identical to the existing translation of operations, which we saw in Figure 5.

Handlers We next look at the handler case. Below is the naïve composition of the two translations.

$$\begin{aligned}&[\![[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m]\!]_c\\ =\ &[\![\texttt{let}\ v = \langle [\![M]\!]_m \mid x.\texttt{return}\ \lambda h.\, [\![M_r]\!]_m\rangle\ \texttt{in}\ v\ (\lambda x.\, \texttt{return}\ \lambda k.\, [\![M_h]\!]_m)]\!]_c\\ =\ &\lambda k_1.\, ([\![[\![M]\!]_m]\!]_c\ (\lambda x.\, \lambda k_2.\, k_2\ (\lambda h.\, [\![[\![M_r]\!]_m]\!]_c)))\\ &\qquad (\lambda v.\, v\ (\lambda x.\, \lambda k_3.\, k_3\ (\lambda k.\, [\![[\![M_h]\!]_m]\!]_c))\ k_1)\end{aligned}$$

As in the case of operations, the result has three continuation variables k_1, k_2, and k_3 that originate from the macro translation. To avoid these continuation variables, we first apply the simpler macro translation, and then selectively apply the CPS traslation. This makes the composition more concise, as shown below.

$$\begin{aligned}&[\![[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m]\!]_c\\ =\ &[\![\langle [\![M]\!]_m \mid x.\lambda h.\, [\![M_r]\!]_m\rangle\ (\lambda x.\, \lambda k.\, [\![M_h]\!]_m)]\!]_c\\ =\ &[\![[\![M]\!]_m]\!]_c(\lambda x.\, \lambda h.\, [\![[\![M_r]\!]_m]\!]_c)\ (\lambda x.\, \lambda k.\, [\![[\![M_h]\!]_m]\!]_c)\end{aligned}$$

The result is again identical to the existing translation of handlers, which we saw in Figure 5.

6 Deriving a Type System from the CPS Translation

The traditional approach to designing a type system for control operators is to analyze their CPS translation [9], which exposes the typing constraints of each syntactic construct. In the case of `shift0` and dollar, the CPS translation gives rise to a stack-like representation of effects, reflecting the ability of `shift0` to capture metacontexts [29,30].

Unlike control operators, the type system of effect handlers has been designed independently of the CPS translation. Indeed, the first type system of effect

handlers [3] was proposed before the first CPS translation [17]. This brings up a question: What would we obtain if we derive the type system of effect handlers from their CPS translation?

In this section, we derive the type system of effect handlers following the recipe of Danvy and Filinski [9], which consists of two steps:

1. Annotate the CPS image of each construct in the most general way
2. Encode the identified typing constraints into the typing rules

We show that, when guided by the CPS translation, we obtain a type system with explicit answer types and a restricted form of answer-type modification [9]. In what follows, we review the existing type systems of `shift0`/dollar and effect handlers (Sections 6.1 and 6.2), design a new type system of effect handlers by applying the CPS approach (Section 6.3), and prove the soundness of the type system (Section 6.4).

6.1 Type System of Shift0 and Dollar

In Figure 8, we present the type system of λ_{S_0}. The type system is adopted from Hillerström et al [17] (for λ-terms) and Forster et al. [13] (for `shift0` and dollar). There are two classes of types: value types (A, B) and computation types (C, D). The latter take the form $A\,!\,\epsilon$, which reads: the computation returns a value of type A while possibly performing an effect represented by ϵ[9]. An effect is a list of *answer types*, representing what kind of context is needed to evaluate a computation[10]. The list is extended by (Shift0) (which requires a specific kind of context) and shrunk by (Dollar) (which supplies the required context).

6.2 Type System of Effect Handlers

In Figure 9, we present the type system of λ_h. The type system is adopted from Hillerström et al. [17]. We again have value and computation types. An effect is either the pure effect ι or an operation type $A \Rightarrow B$; the latter tells us what kind of handler is needed to evaluate a computation. Operation types are introduced by (Do) (which requires a specific kind of operation clause) and eliminated by (Handle) (which supplies the required operation clause). The typing rules for λ-terms stay the same as those of λ_{S_0}, because they do not modify effects.

6.3 Applying the CPS Approach

The type system in Figure 9 is defined on the direct-style expressions in λ_h. We now design a new type system based on the CPS translation. As we saw

[9] In the original type system of Forster et al. [13], effects are attached to the typing judgment instead of being part of a computation type.

[10] The effect representation does not allow *answer-type modification*. A more general (and involved) type system supporting answer-type modification is given by Materzok and Biernacki [30].

Syntax of Types and Effects

$$\begin{array}{rll} A, B & ::= b \mid A \to C & \text{Value Types} \\ C, D, E & ::= A \,!\, \epsilon & \text{Computation Types} \\ \epsilon & ::= [] \mid A :: \epsilon & \text{Effects} \end{array}$$

Typing Rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var}) \qquad \frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.\, M : A \to C}\ (\textsc{Abs}) \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{return}\ V : A \,!\, \epsilon}\ (\textsc{Return})$$

$$\frac{\Gamma \vdash V : A \to C \quad \Gamma \vdash W : A}{\Gamma \vdash V\ W : C}\ (\textsc{App}) \qquad \frac{\Gamma \vdash M : A \,!\, \epsilon \quad \Gamma, x : A \vdash N : B \,!\, \epsilon}{\Gamma \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N : B \,!\, \epsilon}\ (\textsc{Let})$$

$$\frac{\Gamma, k : A \to B \,!\, \epsilon \vdash M : B \,!\, \epsilon}{\Gamma \vdash \mathcal{S}_0 k.\, M : A \,!\, (B :: \epsilon)}\ (\textsc{Shift0}) \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \,!\, (B :: \epsilon) \\ \Gamma, x : A \vdash N : B \,!\, \epsilon\end{array}}{\Gamma \vdash \langle M \mid x.N \rangle : B \,!\, \epsilon}\ (\textsc{Dollar})$$

Fig. 8. Type System of λ_{S_0}

Syntax of Types and Effects

$$\begin{array}{rll} A, B & ::= b \mid A \to C & \text{Value Types} \\ C & ::= A \,!\, \epsilon & \text{Computation Types} \\ \epsilon & ::= \iota \mid A \Rightarrow B & \text{Effects} \end{array}$$

Typing Rules

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{do}\ V : B \,!\, A \Rightarrow B}\ (\textsc{Do}) \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \,!\, A' \Rightarrow B' \\ \Gamma, x : A \vdash M_r : C \\ \Gamma, x : A', k : B' \to C \vdash M_h : C\end{array}}{\Gamma \vdash \mathtt{handle}\ M\ \mathtt{with}\ \{x.\, M_r;\, x, k.\, M_h\} : C}\ (\textsc{Handle})$$

Fig. 9. Type System of λ_h (rules for λ-terms are the same as Figure 8)

in Section 5, a CPS-translated λ_h computation is a function that receives a return continuation and an effect continuation. Among the two arguments, a return continuation takes in a value and an effect continuation, whereas an effect continuation takes in an operation argument and a resumption (a continuation whose handler is already given). Hence, if we annotate the CPS image of a computation in the most general way, we obtain something like this:

$$\lambda k^{A\to(A_1\to(B_1\to C_1)\to D_1)\to E_1}.\,\lambda h^{A_2\to(B_2\to C_2)\to D_2}.\,e^{E_2}$$

These annotations are however too general. The semantics of deep handlers naturally gives rise to the following invariants.

1. The types $A_1 \to (B_1 \to C_1) \to D_1$ and $A_2 \to (B_2 \to C_2) \to D_2$ of the two effect continuations are the same. This is because a computation and its continuation are evaluated under the same handler (when handlers are deep).
2. The return type E_1 of the return continuation and the return type C_2 of the resumption are the same. This is because a resumption is constructed using a return continuation.
3. The type E_2 of the eventual answer must be either C_2 or D_2. This is because the answer is produced by the return or the operation clause of the surrounding handler.

These invariants lead to the following refined annotations.

$$\lambda k^{A\to(A'\to(B'\to C)\to D)\to C}.\,\lambda h^{A'\to(B'\to C)\to D}.\,e^{E} \qquad \text{where } E = C \text{ or } D$$

The annotations have six different types. Among them, A', B', C, D, and E specify the kind of the handler required by the computation, as well as the kind of answer to be produced by the handler. By including these types in non-empty effects, we arrive at the following effect representation.

$$\epsilon ::= \iota \mid \langle A \Rightarrow B, C, D, E\rangle \qquad \text{where } E = C \text{ or } D$$

Here, $\langle A \Rightarrow B, C, D, E\rangle$ carries the following information.

- The computation may perform an operation of type $A \Rightarrow B$.
- When evaluated under a handler whose return clause has type C and whose operation clause has type D, the computation returns an answer of type E (which must be either C or D).

Using this representation of effects, we design the typing rules for λ_h computations and values. We do this by annotating the CPS image of individual syntactic constructs and converting annotated expressions into typing derivations.

Return Expressions We begin by annotating the CPS image of a `return` expression. To make the calculation of types easier to follow, we explicitly write the effect continuation h.

$$\lambda k^{A\to(A'\to(B'\to C)\to D)\to C}.\, \lambda h^{A'\to(B'\to C)\to D}.\, (k\ [\![V]\!]^{A}_{c'}\ h)^{C}$$

The trivial use of the continuation forces the type of the eventual answer to be C (corresponding to invariant 3 mentioned above). From this annotated expression, we obtain the following typing rule.

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{return}\ V : A\,!\,\langle A' \Rightarrow B', C, D, C\rangle}\ (\textsc{Return})$$

Application We next annotate the CPS image of an application.

$$([\![V]\!]^{A\to C}_{c}\ [\![W]\!]^{A}_{c})^{C}$$

The annotations simply tell us that the effect of an application comes from the body of the function. This corresponds to the following typing rule.

$$\frac{\Gamma \vdash V : A \to C \quad \Gamma \vdash W : A}{\Gamma \vdash V\ W : C}\ (\textsc{App})$$

Let Expressions Having analyzed application, we consider the `let` expression.

$$\begin{array}{l}\lambda k^{B\to(A'\to(B'\to C)\to D)\to C}.\\ \quad [\![M]\!]^{(A\to(A'\to(B'\to C)\to D)\to C)\to(A'\to(B'\to C)\to D)\to E}_{c}\\ \quad (\lambda x^{A}.\, [\![N]\!]^{(B\to(A'\to(B'\to C)\to D)\to C)\to(A'\to(B'\to C)\to D)\to C}_{c}\ k)\end{array}$$

The sequencing of M and N forces the eventual answer type of N to be C. The application of $[\![M]\!]_c$ then forces the eventual answer type of the entire construct to be E. The corresponding typing rule is as follows.

$$\frac{\Gamma \vdash M : A\,!\,\langle A' \Rightarrow B', C, D, E\rangle \quad \Gamma, x : A \vdash N : B\,!\,\langle A' \Rightarrow B', C, D, C\rangle}{\Gamma \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N : B\,!\,\langle A' \Rightarrow B', C, D, E\rangle}\ (\textsc{Let})$$

Operations We now move on to the case of an operation call.

$$\lambda k^{B'\to(A'\to(B'\to C)\to D)\to C}.\, \lambda h^{A'\to(B'\to C)\to D}.\, (h\ [\![V]\!]_{c'}\ (\lambda x^{A}.\, k\ x\ h))^{D}$$

The application of h forces the eventual answer type to be D (invariant 3). The application $k\ x\ h$ restricts the annotations in two ways: (i) the handler type of the whole expression and that of k must coincide (invariant 1); and (2) the return type of k and that of the resumption must coincide (invariant 2). Below is the derived typing rule.

$$\frac{\Gamma \vdash V : A'}{\Gamma \vdash \mathtt{do}\ V : B'\,!\,\langle A' \Rightarrow B', C, D, D\rangle}\ (\textsc{Do})$$

Handlers Lastly, we consider the case of a handler.

$$\begin{array}{l}[\![M]\!]_c^{(A\to(A'\to(B'\to C)\to D)\to C)\to(A'\to(B'\to C)\to D)\to E}\\(\lambda x^{A}.\,\lambda h^{A'\to(B'\to C)\to D}.\,[\![e_r]\!]_c^{C})\\(\lambda x^{A'}.\,\lambda k^{B'\to C}.\,[\![e_h]\!]_c^{D}))^{E}\end{array}$$

The construction requires the effect of the handled expression M and the type of the two clauses of the handler to be consistent. Thus we obtain the following typing rule.

$$\frac{\begin{array}{c}\Gamma \vdash M : A\,!\,\langle A' \Rightarrow B', C, D, E\rangle\\ \Gamma, x : A \vdash M_r : C\\ \Gamma, x : A', k : B' \to C \vdash M_h : D\end{array}}{\Gamma \vdash \texttt{handle}\ M\ \texttt{with}\ \{x.\,M_r;\, x, k.\,M_h\} : E}\ (\textsc{Handle})$$

Values Let us complete the development by giving the typing rules for variables and abstractions. These are derived straightforwardly from the CPS translation on values. The only thing that is non-standard is the well-formedness condition $\vdash A$ in the (Var) rule. Here, well-formedness means every function type that appears in a type takes the form $A \to B\,!\,\langle A' \Rightarrow B', C, D, C\rangle$ or $A \to B\,!\,\langle A' \Rightarrow B', C, D, D\rangle$.

$$\frac{x : A \in \Gamma \quad \vdash A}{\Gamma \vdash x : A}\ (\textsc{Var}) \qquad \frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.\,M : A \to C}\ (\textsc{Abs})$$

Through this exercise, we obtained a type system that is different from the one presented in Figure 9. First, the type system explicitly keeps track of answer types. Second, the type system allows a form of answer-type modification [9,1,22]. Answer-type modification is the ability of an effectful computation to change the return type of its delimited context. In our setting, this ability is gained by allowing the return and operation clauses of a handler to have different types.

The answer-type modification supported in this type system is however very limited. Specifically, it does not allow answer-type modification in a captured continuation. In fact, we have already seen where this restriction comes from: the `let` expression. Recall that, in the typing rule (Let), the body N of `let` (which corresponds to the continuation of M) has a type of the form $B\,!\,\langle A' \Rightarrow B', C, D, C\rangle$. This means N must be a pure computation (which is eventually handled by the return clause of type C) or it must be handled by a handler whose return and operation clauses have the same type (i.e., $C = D$).

6.4 Soundness

The type system we derived from the CPS translation is sound. We state this property as the following theorem.

Theorem 2 (Soundness of Type System). *If $\Gamma \vdash M : A\,!\,\iota$, then $M \rightsquigarrow$ `return` V, and $\Gamma \vdash V : A$.*

Proof. The statement is witnessed by a CPS interpreter[11] written in the Agda programming language [33]. The interpreter takes in a well-typed λ_h computation and returns a fully-evaluated Agda value. The signature of the interpreter corresponds exactly to the statement of soundness, and the well-typedness of the interpreter implies the validity of the statement.

7 Related Work

Variations of Effect Handler Calculi There is a discussion of the relationship between effect operations with and without a label, which is similar to our discussion of handlers with and without the return clause. The handling of a labeled operation may involve skipping of handlers that do not take care of the operation in question. To simulate labeled operations in a calculus with unlabeled operations, Biernacki et al. [5] introduce an operator [_] called "lift", which allows an operation to skip the innermost handler and be handled by an outer handler. This enables programming with multiple effects without using labels, although it requires the programmer to know what handlers are surrounding each operation in what order.

CPS Translations of Effect Handlers The CPS translations of effect handlers have been developed as a technique for compiling effect handlers into common runtime platforms. For this reason, existing CPS translations are either directed by the specific typing discipline of the source language [24,7] or tuned for the particular implementation strategy of the compiler [17,16]. We derived a CPS translation of effect handlers from the definitional CPS translation of `shift0` and dollar. As a result, we obtained a translation that is identical to Hillerström's [17] unoptimized translation, which we regard as the definitional translation of general (deep) effect handlers.

Type Systems of Effect Handlers There are different flavors of type systems for effect handlers. In the most traditional type systems [3,19], effects are represented as a set of operations, similar to the effect systems for side-effect analysis [32]. In some research languages such as Koka [23], Frank [27], and Links [14], effects are treated as row types, which were originally developed for type inference with records [38]. More recently, several languages [43,8] adopt the notion of capabilities from object-oriented programming as an approach to safe handling of effects. Unlike the type system we developed in Section 6, these type systems are all defined independent of the CPS translation, and they do not explicitly carry answer types.

Effect Handlers and Control Operators The relationship between effect handlers and control operators has been studied from several different perspectives. Forster et al. [13] and Piróg et al. [34] establish macro translations to compare

[11] The interpreter is available at `https://github.com/YouyouCong/tfp22`.

the expressiveness of the two facilities. Kiselyov and Sivaramakrishnan [21] define a similar translation to embed the Eff language into OCaml. We exploit the results of these studies to enhance our understanding of effect handlers.

8 Conclusion and Future Perspectives

In this paper, we presented three results from our study on understanding deep effect handlers via the `shift0` and dollar operators. Specifically, we defined a pair of macro translations that add and remove the return clause, derived a CPS translation for effect handlers from the CPS translation for `shift0`/dollar, and designed a type system for effect handlers from the CPS translation.

As a continuation of this work, we intend to visit those challenges that have been solved for `shift0` and dollar (or reset) but not for effect handlers. One question we would like to investigate is whether effect handlers can fully express the CPS hierarchy [10]. The other question we are interested in is how to give an equational axiomatization [28] and a reflection [6] of effect handlers. By answering these questions, we would be able to discover novel tecnhiques for implementing, optimizing, and reasoning about effect handlers.

As an extension of our approach, we plan to study shallow effect handlers [15] by leveraging their connection to the `control0` and `prompt0` operators [34]. These operators have a more complex semantics due to the absence of the delimiter surrounding captured continuations, but we conjecture that it is possible to establish similar results to what we have presented in this paper.

Acknowledgments We gratefully thank the anonymous reviewers for their constructive feedback on the presentation and technical development. This work was supported by JSPS KAKENHI under Grant No. JP18H03218, No. JP19K24339, and No. JP22H03563.

References

1. Asai, K.: On typing delimited continuations: three new solutions to the printf problem. Higher-Order and Symbolic Computation **22**(3), 275–291 (2009)
2. Asai, K., Uehara, C.: Selective CPS transformation for shift and reset. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 40–52. PEPM '18, ACM, New York, NY, USA (dec 2017). https://doi.org/10.1145/3162069
3. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. In: Heckel, R., Milius, S. (eds.) Algebra and Coalgebra in Computer Science. pp. 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
4. Biernacki, D., Danvy, O., Millikin, K.: A dynamic continuation-passing style for dynamic delimited continuations. ACM Trans. Program. Lang. Syst. **38**(1) (Oct 2015). https://doi.org/10.1145/2794078
5. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Handle with care: Relational interpretation of algebraic effects and handlers. Proc. ACM Program. Lang. **2**(POPL), 8:1–8:30 (Dec 2017). https://doi.org/10.1145/3158096

6. Biernacki, D., Pyzik, M., Sieczkowski, F.: Reflecting Stacked Continuations in a Fine-Grained Direct-Style Reduction Theory. PPDP '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3479394.3479399
7. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effect handlers for the masses. Proc. ACM Program. Lang. **2**(OOPSLA), 111:1–111:27 (Nov 2018)
8. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effects as capabilities: Effect handlers and lightweight effect polymorphism. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428194
9. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. BRICS 89/12 (Aug 1989)
10. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM conference on LISP and functional programming. pp. 151–160. ACM (1990)
11. Dyvbig, R.K., Peyton Jones, S., Sabry, A.: A monadic framework for delimited continuations. J. Funct. Program. **17**(6), 687–730 (Nov 2007). https://doi.org/10.1017/S0956796807006259
12. Felleisen, M.: On the expressive power of programming languages. In: Selected Papers from the Symposium on 3rd European Symposium on Programming. pp. 35–75. ESOP '90, Elsevier North-Holland, Inc., New York, NY, USA (1991)
13. Forster, Y., Kammar, O., Lindley, S., Pretnar, M.: On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. Journal of Functional Programming **29**, e15 (2019). https://doi.org/10.1017/S0956796819000121
14. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: Proceedings of the 1st International Workshop on Type-Driven Development. pp. 15–27. TyDe 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976022.2976033
15. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Asian Symposium on Programming Languages and Systems. pp. 415–435. APLAS '18, Springer (2018)
16. Hillerström, D., Lindley, S., Atkey, R.: Effect handlers via generalised continuations. Journal of Functional Programming **30** (2020). https://doi.org/10.1017/S0956796820000040
17. Hillerström, D., Lindley, S., Atkey, R., Sivaramakrishnan, K.: Continuation passing style for effect handlers. In: Proceedings of 2nd International Conference on Formal Structures for Computation and Deduction. pp. 18:1–18:19. FSCD '17, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017)
18. Kameyama, Y., Yonezawa, T.: Typed dynamic control operators for delimited continuations. In: International Symposium on Functional and Logic Programming. pp. 239–254. FLOPS '08, Springer (2008)
19. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 145–158. ICFP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2500365.2500590
20. Kiselyov, O., Shan, C.c.: A substructural type system for delimited continuations. In: International Conference on Typed Lambda Calculi and Applications. pp. 223–239. TLCA '07, Springer (2007)
21. Kiselyov, O., Sivaramakrishnan, K.C.: Eff directly in ocaml. In: Asai, K., Shinwell, M. (eds.) Proceedings of ML/OCaml 2016. pp. 23–58 (2018). https://doi.org/10.4204/EPTCS.285.2

22. Kobori, I., Kameyama, Y., Kiselyov, O.: Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. In: Electronic Proceedings in Theoretical Computer Science EPTCS 212 (Post-Proceedings of the Workshop on Continuations 2015). pp. 36–52 (June 2016). https://doi.org/10.4204/EPTCS.212.3
23. Leijen, D.: Koka: Programming with row polymorphic effect types. In: 5th Workshop on Mathematically Structured Functional Programming. MSFP '14 (2014). https://doi.org/10.4204/EPTCS.153.8
24. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 486–499. POPL '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009872
25. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis. Springer, Dordrecht (2003)
26. Levy, P.B., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Information and computation **185**(2), 182–210 (2003)
27. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 500–514. POPL '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009897
28. Materzok, M.: Axiomatizing subtyped delimited continuations. In: Computer Science Logic 2013. CSL 2013, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
29. Materzok, M., Biernacki, D.: Subtyping delimited continuations. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. pp. 81–93. ICFP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034786, extended version available at `http://www.tilk.eu/shift0/materzok-biernacki-HOSC13.pdf`
30. Materzok, M., Biernacki, D.: A dynamic interpretation of the CPS hierarchy. In: Asian Symposium on Programming Languages and Systems. pp. 296–311. APLAS '12, Springer (2012)
31. Nielsen, L.R.: A selective CPS transformation. Electronic Notes in Theoretical Computer Science **45**, 311–331 (2001)
32. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel). pp. 114–136. Springer-Verlag, Berlin, Heidelberg (1999)
33. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
34. Piróg, M., Polesiuk, P., Sieczkowski, F.: Typed equivalence of effect handlers and delimited control. In: 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
35. Plotkin, G.: Call-by-name, call-by-value and the λ-calculus. Theoretical computer science **1**(2), 125–159 (1975)
36. Plotkin, G., Power, J.: Algebraic operations and generic effects. Applied categorical structures **11**(1), 69–94 (2003)
37. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: European Symposium on Programming. pp. 80–94. ESOP '09, Springer (2009)
38. Rémy, D.: Type Inference for Records in Natural Extension of ML, p. 67–95. MIT Press, Cambridge, MA, USA (1994)

39. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 317–328. ICFP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1596550.1596596
40. Schuster, P., Brachthäuser, J.I., Ostermann, K.: Compiling effect handlers in capability-passing style. Proc. ACM Program. Lang. **4**(ICFP) (aug 2020). https://doi.org/10.1145/3408975
41. Shan, C.c.: A static simulation of dynamic delimited control. Higher-Order and Symbolic Computation **20**(4), 371–401 (2007)
42. Xie, N., Brachthäuser, J.I., Hillerström, D., Schuster, P., Leijen, D.: Effect handlers, evidently. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (2020). https://doi.org/10.1145/3408981
43. Zhang, Y., Myers, A.C.: Abstraction-safe effect handlers via tunneling. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290318
44. Zhang, Y., Salvaneschi, G., Myers, A.C.: Handling bidirectional control flow **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428207