Temporal Algebraic Effect Handlers

Yulong Liu Taro Sekiyama

Algebraic effect handlers make control flow modular, but they also complicate the temporal behavior of a computation: a handler clause may discard, resume, or duplicate the continuation captured at an operation. To bridge this gap, we present a type-and-effect system whose effects are expressed as higher-order recursion schemes (HORS). The choice of HORS over less expressive formalisms such as regular expressions is justified with practical examples involving resource protocols where effect handlers can break regularity. In particular, we formalize a program syntax and endow it with a polymorphic type-and-effect system. To describe program effects as HORS, we define the relevant HORS constructions for sequencing, branching, and handling. We also elaborate on subeffecting by suggesting that trace inclusion loses branching information observable by multi-shot handlers and that a branch-sensitive definition is needed instead.

1 Introduction

Many program properties concern not only the value that a computation returns, but also how it interacts with its environment along the way. Resource usage, communication protocols, and event-driven computation impose constraints on the order and branching structure of observable operations. *Type-and-effect systems* make such behavior available to static reasoning, while *temporal* effect systems refine a mere collection of possible operations with information about their sequencing and repetition [11]. Recursive and higher-order programs make this behavior potentially infinite, thus requiring a useful effect language to describe infinite families of executions with a finite representation.

To make the control flow of effectful programs more modular, *algebraic effects and handlers* have been adopted in practical programming-language ecosystems, with native support in OCaml 5 and library embeddings in Haskell [13][15]. An algebraic effect is invoked by primitive operations that separate a request for an effectful computation from its implementation, while a *handler* supplies that implementation through return and operation clauses [7][8]. Moreover, the operation clause may discard the continuation, invoke it once, or invoke it multiple times, and may perform other operations before, between, or after those invocations. Handlers can therefore express exceptions, ordinary interpretation, and multi-shot control patterns such as backtracking.

A concrete example is the following program:

$$
\begin{aligned}
M_{\mathrm{walk}} := \mathsf{letrec}\ \mathit{walk} = (\lambda d.\ & \\
& \mathsf{let}\ d' \leftarrow \mathbf{descend}\ d\ \mathsf{in} \\
& \mathsf{if}\ d' \neq \texttt{""}\ \mathsf{then}\ \mathit{walk}\ d'\ \mathsf{else}\ \mathbf{return}\ d \\
)\ \mathsf{in}\ \mathit{walk}\ & \texttt{"."}.
\end{aligned}
$$

The program M_{walk} describes traversal down one path of an arbitrarily deep directory tree from the current directory. The operation **descend** descends the tree by one level, and returns the name of an arbitrary subdirectory, if one exists; otherwise returns the empty string. Using *temporal regular effects* [11], we can reason that this program's terminating *effect traces* conform to the regular expression $\mathbf{descend} \cdot \mathbf{descend}^*$.

With effect handlers, we can implement the operation **descend** with more primitive ones, reminiscent of system calls to an operating system:

This is an unreferred paper. Copyrights belong to the authors.

Yulong Liu, Taro Sekiyama, 国立情報学研究所, National Institute of Informatics.

```
M_walk := handle {
            return x ↦ return x,
            descend x k ↦
              let _ ← enter x in
              let d′ ← getChild () in
              let x ← k d′ in
              let _ ← leave () in
              return x
          } M_walk.
```

The handler implements each descent by entering the given directory via **enter**, looking for an arbitrary subdirectory using **getChild**, running the remaining traversal by calling the continuation k while keeping ancestor directories open, and leaving that directory afterward via **leave**.

Since the effect traces of the unhandled program M_{walk} are of the form $\mathbf{descend}^n$ for $n \geq 1$, one would reasonably propose that the handled program $\underline{M}_{\text{walk}}$ has the (finite) traces

$$\{(\mathbf{enter} \cdot \mathbf{getChild})^n \cdot \mathbf{leave}^n : n \geq 1\}. \quad (1)$$

However, the traces described above no longer form a regular language. Hence, even handling a regular source effect need not produce a regular effect; analogous examples can exceed context-free effect languages. To resolve this issue, we study higher-order recursion schemes (HORS) [6], a class of tree grammars that generate possibly infinite labeled trees. We propose that the class of HORS-definable effects is closed under effect handlers.

More specifically, this paper makes the following contributions.

1. We present a polymorphic calculus with recursive functions and algebraic effect handlers.
2. We endow our calculus with a type-and-effect system à la Hindley-Milner in which effects are expressed as higher-order recursion schemes (HORS).
3. We define corresponding HORS constructions for sequential composition, branching, and handling of effectful programs.
4. We discuss decidability, subeffecting, and future directions including shallow handlers, effect polymorphism, and refinement types.

Section 2 develops the calculus, its operational semantics, HORS effect descriptions, and the Hindley–Milner type-and-effect system. It then defines HORS constructions for branching, sequencing, and handling. Section 3 records the outstanding well-typedness, soundness, preservation, type-safety, and decidability obligations. Section 4 discusses future work on shallow handlers, branch-sensitive subeffecting, effect polymorphism, and refinements, while Section 5 compares the approach with related work on effect handlers, HORS verification, and temporal type systems for delimited control operators.

2 Temporal Effect System with HORS

In this section, we present a higher-order language with recursive functions, effectful operations, and effect handlers. We then equip our language with a polymorphic type-and-effect system, and provide the semantics of effects as higher-order recursion schemes.

2.1 Syntax and Semantics

Syntax. The syntax of our language is shown in Figure 1. Values, ranged over by V, consist of variables x; boolean and unit constants; and lambda abstractions $\lambda x.\, M$. Computations, ranged over by M and N, are pure computations $\mathbf{return}\, V$; lambda applications $V_1\ V_2$; applications of effectful operations $\mathbf{op}\, V$; if-expressions $\mathbf{if}\, V\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2$; let-expressions $\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N$; lambda-restricted recursive let-expressions $\mathbf{letrec}\, g = V\, \mathbf{in}\, N$; and handled computations $\mathbf{handle}\, h\, M$. A handler h consists of a return clause $\mathbf{return}\, x \mapsto M$ and an operation clause $\mathbf{op}\, x\, k \mapsto M$. In both clauses, x is bound to the value V in $\mathbf{return}\, V$ and $\mathbf{op}\, V$ respectively in the handled computation. In the operation clause, k will bind to the delimited continuation from the **op** call. Evaluation contexts are defined as usual.

Types consist of base types Unit and Bool, type variables α, and function types $A \to_E B$ where E is an effect. The type $A \to_E B$ describes computations that, when supplied a value of type A, return a value of type B while producing effects described by E. Type schemes P are types whose variables are bound by zero or more quantifiers.

The heart of this work is to introduce effects as higher-order recursion schemes, but for syntactic brevity, we formalize them as lambda terms with fix operators, simply typed over a unique base kind o. The constants are $\bullet : o$ for pure computations,

$\mathbf{op} : o \to o$ for each effectful operation $\mathbf{op}$, and $+ : o \to o \to o$ for branching. This presentation is as expressive as higher-order recursion schemes in the sense that both generate the same class of labeled trees [14] [1]. Nevertheless, the rest of the paper works with effects as recursion schemes for the sake of intuition.

Semantics. Figure 2 defines the operational semantics of the language. Function application $(\lambda x.\, M)\, V$ and let-expression for pure computations $\mathbf{let}\, x \leftarrow \mathbf{return}\, V\, \mathbf{in}\, M$ reduce to $M[V/x]$ as usual. A recursive computation $\mathbf{letrec}\, g = (\lambda x.\, M)\, \mathbf{in}\, N$ reduces to N with each occurrence of g substituted by $\lambda x.\, \mathbf{letrec}\, g = (\lambda x.\, M)\, \mathbf{in}\, M$. If-expressions have their standard reduction rules, and the last two rules are for handled computations. If a handled expression is pure $\mathbf{return}\, V$, then the R-Return rule will reduce $\mathbf{handle}\, h\, (\mathbf{return}\, V)$ to $M[V/x]$ where $\mathsf{return}\, x \mapsto M \in h$ is the return clause of the handler h. Otherwise, the expression to reduce takes the form $\mathbf{handle}\, h\, \mathcal{E}[\mathbf{op}\, V]$ where $\mathcal{E}$ contains no handler for the operation $\mathbf{op}$. When $\mathbf{op}\, x\, k \mapsto M$ is a clause of h, rule R-Op substitutes V for x and the continuation $\lambda y.\, \mathbf{handle}\, h\, \mathcal{E}[\mathbf{return}\, y]$ for k, so every invocation of the captured continuation reinstalls the same handler.

Finally, the evaluation proceeds according to E-Eval. Under this rule, a program is first decomposed into an evaluation context $\mathcal{E}$ whose hole is filled by a reducible expression M; the resulting term then steps by replacing M with its reduction M'.

2.2 Effects and Higher-Order Recursion Schemes

As mentioned above, while effects are syntactically recursive lambda terms, our primary objective is to express them as *higher-order recursion schemes* (HORS). Formally, a HORS $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ consists of

- a *ranked alphabet of terminals* Σ where we write $a : n \in \Sigma$ to denote a terminal a with the assigned kind $\underbrace{o \to \cdots}_{n} \to o$;
- a set $\mathcal{N}$ of typed *non-terminals*, with $S : o \in \mathcal{N}$ being a distinguished *start symbol* of kind o;
- a finite set $\mathcal{R}$ of *rewrite rules*, each of the form
$$F\, x_1 \cdots x_n \to t$$
where $F : \kappa_1 \to \cdots \to \kappa_n \to o$, each $x_i : \kappa_i$, and $t : o$ is an applicative term over $\Sigma \cup \mathcal{N} \cup \{x_1, \cdots, x_n\}$.

As in the syntax described in Figure 1, our terminals consist exactly of a unique 0-ary $\bullet$ denoting the end of a computation, a unary $\mathbf{op}$ for each effectful operation, and a unique binary $+$ for branching over if-statements. For example, we can assign to M_{walk} from the previous section the following effect:

$$S \to \mathbf{descend}\, (+\, S\, \bullet).$$

We can also describe, as follows, the effect of the handled program shown in 1:

$$S \to F\, \bullet$$
$$F\, x \to \mathbf{enter}\, (\mathbf{getChild}\, (+\, (F\, (\mathbf{leave}\, x))\, (\mathbf{leave}\, x))).$$

Here, the argument x of F serves as a binder for the rest of a branch, consisting of a sequence of $\mathbf{leave}$s followed by $\bullet$ as the leaf.

2.3 Type-and-Effect System

To statically assign appropriate types and effects to programs, we define a type-and-effect system with typing judgments of the form $\Delta\, ;\, \Gamma \vdash V : A$ and $\Delta\, ;\, \Gamma \vdash M : A \mid E$, where Δ kinds effect variables and Γ types term variables. The purpose of Δ is to type handlers. Since a single operation clause of a handler may be used to handle different continuations, the operation clause is meant to be well-typed over an arbitrary effect $p : o \to o$ representing the captured continuation.

Figure 3 defines the typing rules for values, computations, and handlers. The typing system is based on Hindley-Milner typing systems for algebraic effect handlers [3] with the addition of recursive functions. Notable rules are T_LET, T_IF, and T_HANDLER. They involve new notations that compose effects through sequencing $E \cdot F$, branching $E + F$, and handling $\underline{E}^{R}_{\lambda \xi^{o \to o}.\, O}$, respectively. These are HORS constructions whose value trees correspond to the behavior of the composed programs.

In particular, the third argument of the handling construction has kind $(o \to o) \to o$. This is because we require the effect $O : o$ for the operation clause to be well-kinded under a fresh variable

$$
\begin{array}{rcl}
\textbf{Constants} & c & ::= () \mid \mathsf{true} \mid \mathsf{false} \\
\textbf{Values} & V & ::= x \mid c \mid \lambda x.\, M \\
\textbf{Computations} & M, N & ::= \mathbf{return}\ V \mid V_1\ V_2 \mid \mathbf{op}\ V \mid \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \\
 & & \quad \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \mathbf{letrec}\ g = V\ \mathbf{in}\ N \mid \mathbf{handle}\ h\ M \\
\textbf{Handlers} & h & ::= \{\mathbf{return}\ x \mapsto M, \mathbf{op}\ x\ k \mapsto N\} \\
\textbf{Base Types} & T & ::= \mathsf{Unit} \mid \mathsf{Bool} \\
\textbf{Types} & A, B, C, D & ::= T \mid \alpha \mid A \to_E B \\
\textbf{Type Schemes} & P & ::= A \mid \forall \boldsymbol{\alpha}.\, A \\
\textbf{Effect Kinds} & \kappa & ::= o \mid \kappa_1 \to \kappa_2 \\
\textbf{Effects} & E, F, R, O & ::= p \mid \bullet \mid \mathbf{op}\ E \mid +\ E\ F \mid E\ F \mid \lambda p^{\kappa}.\, E \mid \mathsf{fix}\, p^{\kappa}.\, E \\
\textbf{Evaluation Contexts} & \mathcal{E} & ::= \Box \mid \mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N \mid \mathbf{handle}\ h\ \mathcal{E}
\end{array}
$$

Fig. 1 Syntax

Reduction rules $\boxed{M \rightsquigarrow N}$

$$
\begin{array}{rcll}
(\lambda x.\, M)\ V & \rightsquigarrow & M[V/x] & \text{(R-Beta)} \\
\mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ M & \rightsquigarrow & M[V/x] & \text{(R-Let)} \\
\mathbf{letrec}\ g = (\lambda x.\, M)\ \mathbf{in}\ N & \rightsquigarrow & N[(\lambda x.\, \mathbf{letrec}\ g = (\lambda x.\, M)\ \mathbf{in}\ M)/g] & \text{(R-Letrec)} \\
\mathbf{if}\ \mathsf{true}\ \mathbf{then}\ M\ \mathbf{else}\ N & \rightsquigarrow & M & \text{(R-IfTrue)} \\
\mathbf{if}\ \mathsf{false}\ \mathbf{then}\ M\ \mathbf{else}\ N & \rightsquigarrow & N & \text{(R-IfFalse)} \\
\mathbf{handle}\ h\ (\mathbf{return}\ V) & \rightsquigarrow & M[V/x] & \text{(R-Return)} \\
 & & \text{where } \mathsf{return}\ x \mapsto M \in h & \\
\mathbf{handle}\ h\ \mathcal{E}[\mathbf{op}\ V] & \rightsquigarrow & M[V/x][(\lambda y.\, \mathbf{handle}\ h\ \mathcal{E}[\mathbf{return}\ y])/k] & \text{(R-Op)} \\
 & & \text{where } \mathbf{op} : A \to B, \quad \mathbf{op}\ x\ k \mapsto M \in h, \quad \mathbf{op} \notin \mathcal{E} &
\end{array}
$$

Evaluation rule $\boxed{M \longrightarrow N}$

$$
\frac{M \rightsquigarrow M'}{\mathcal{E}[M] \longrightarrow \mathcal{E}[M']} \quad \text{(E-Eval)}
$$

Fig. 2 Semantics

$\xi : o \to o$[†1]. The variable is used to type N under the operation argument $x : A$ and continuation $k : B \to_{\xi\, \bullet} D$. Hence, N may surround the resumption with more effects:

$$(\mathbf{enter}\ (\xi\ \bullet)) \cdot \mathbf{leave}\ \bullet = \mathbf{enter}\ (\xi\ (\mathbf{leave}\ \bullet))$$

Note that according to the typing rule, ξ cannot escape into the handler's domain or result type. It is a scoped parameter of this rule, and in Section 2. 4. 1 we instantiate it at each handled operation.

2. 4 HORS Constructions

Write recursion schemes $G_i = \langle \Sigma, \mathcal{N}_i, \mathcal{R}_i, S_i \rangle$ where we assume, without loss of generality, that the sets of non-terminals $\mathcal{N}_i$ are disjoint. Define

$$
\begin{aligned}
\mathcal{N}_+ &= \mathcal{N}_1 \uplus \mathcal{N}_2 \uplus \{S\}, \\
\mathcal{R}_+ &= \mathcal{R}_1 \uplus \mathcal{R}_2 \uplus \{S \to +\, S_1\, S_2\}, \\
G_1 + G_2 &= \langle \Sigma, \mathcal{N}_+, \mathcal{R}_+, S \rangle, \\
\mathcal{N}_\cdot &= \mathcal{N}_1 \uplus \mathcal{N}_2, \\
\mathcal{R}_\cdot &= \mathcal{R}'_1 \uplus \mathcal{R}_2, \\
G_1 \cdot G_2 &= \langle \Sigma, \mathcal{N}_\cdot, \mathcal{R}_\cdot, S_1 \rangle,
\end{aligned}
$$

where $\mathcal{R}'_1$ replaces every occurrence of $\bullet$ in every right-hand side of $\mathcal{R}_1$ by S_2. Since $S_1, S_2, \bullet$ all have type o, the constructions produce well-typed recursion schemes without increasing the order.

2. 4. 1 Construction for Handlers

To assign an effect to a handled computation, we first expose the effects of the handler clauses. Given a handler h with return clause $\mathbf{return}\ x \mapsto M_R$ and operation clause $\mathbf{op}\ x\ k \mapsto M_O$, let $G_R = \langle \Sigma, \mathcal{N}_R, \mathcal{R}_R, S_R \rangle$ describe M_R and let $G_O = \langle \Sigma, \mathcal{N}_O, \mathcal{R}_O, S_O \rangle$ be a parameterized clause representing the abstraction $\lambda \xi^{o \to o}.O$ assigned to M_O by T_HANDLER. Thus $S_O : (o \to o) \to o$ and occurrences of ξ record invocations of the continuation k in the handler's clause.

The construction is split into two stages. First,

†1 We use ξ instead of p to distinguish its purpose in typing handlers.

Typing values $\boxed{\Delta\,;\,\Gamma \vdash V : A}$

$$\frac{x : P \in \Gamma \quad \Delta\,;\,\Gamma \vdash P \preceq A}{\Delta\,;\,\Gamma \vdash x : A}\ \text{T_VAR} \qquad \frac{}{\Delta\,;\,\Gamma \vdash () : \mathsf{Unit}}\ \text{T_UNIT} \qquad \frac{\Delta\,;\,\Gamma, x : A \vdash M : B \mid E}{\Delta\,;\,\Gamma \vdash \lambda x.\, M : A \to_E B}\ \text{T_ABS}$$

$$\frac{}{\Delta\,;\,\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T_TRUE} \qquad \frac{}{\Delta\,;\,\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ \text{T_FALSE}$$

Typing Computations $\boxed{\Delta\,;\,\Gamma \vdash M : A \mid E}$

$$\frac{\Delta\,;\,\Gamma \vdash V : A}{\Delta\,;\,\Gamma \vdash \mathbf{return}\ V : A \mid \bullet}\ \text{T_RETURN} \qquad \frac{\mathbf{op} : A \to B \quad \Delta\,;\,\Gamma \vdash V : A}{\Delta\,;\,\Gamma \vdash \mathbf{op}\ V : B \mid \mathbf{op}\,\bullet}\ \text{T_OP} \qquad \frac{\Delta\,;\,\Gamma \vdash V_1 : A \to_E B \quad \Delta\,;\,\Gamma \vdash V_2 : A}{\Delta\,;\,\Gamma \vdash V_1\ V_2 : B \mid E}\ \text{T_APP}$$

$$\frac{\Delta\,;\,\Gamma, \boldsymbol{\alpha} \vdash M : A \mid E \quad \Delta\,;\,\Gamma, x : \forall \boldsymbol{\alpha}.\, A \vdash N : B \mid F}{\Delta\,;\,\Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : B \mid E \cdot F}\ \text{T_LET} \qquad \frac{\Delta\,;\,\Gamma, \boldsymbol{\alpha}, g : A \to_E B, x : A \vdash M : B \mid E \quad \Delta\,;\,\Gamma, g : \forall \boldsymbol{\alpha}.\, A \to_E B \vdash N : C \mid F}{\Delta\,;\,\Gamma \vdash \mathbf{letrec}\ g = (\lambda x.\, M)\ \mathbf{in}\ N : C \mid F}\ \text{T_LETREC}$$

$$\frac{\Delta\,;\,\Gamma \vdash V : \mathsf{Bool} \quad \Delta\,;\,\Gamma \vdash M_1 : A \mid E_1 \quad \Delta\,;\,\Gamma \vdash M_2 : A \mid E_2}{\Delta\,;\,\Gamma \vdash \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : A \mid E_1 + E_2}\ \text{T_IF} \qquad \frac{\Delta\,;\,\Gamma \vdash h : A \mid E \Rightarrow B \mid F \quad \Delta\,;\,\Gamma \vdash M : A \mid E}{\Delta\,;\,\Gamma \vdash \mathbf{handle}\ h\ M : B \mid F}\ \text{T_HANDLE}$$

Typing Handlers $\boxed{\Delta\,;\,\Gamma \vdash h : A \mid E \Rightarrow B \mid F}$

$$\frac{\begin{array}{l}\mathbf{op} : A \to B \\ \Delta\,;\,\Gamma, x : C \vdash M : D \mid R \\ \xi \# \Delta \\ \Delta, \xi : o \to o\,;\,\Gamma, x : A, k : B \to_{(\xi\,\bullet)} D \vdash N : D \mid O\end{array}}{\Delta\,;\,\Gamma \vdash \{\mathbf{return}\ x \mapsto M, \mathbf{op}\ x\ k \mapsto N\} : C \mid E \Rightarrow D \mid \underline{E}^{R}_{(\lambda \xi^{o \to o}.\, O)}}\ \text{T_HANDLER}$$

Fig. 3 Typing rules

we parameterize G_R and G_O so the leaves can be arbitrary trees $q : o$ instead of $\bullet : o$. Second, given a program M with effect G, we construct the handled HORS $\underline{G}$ from G_R and G_O as the effect for $\mathbf{handle}\, h\, M$.

Parameterization. We first parameterize G_R. The idea is to build non-terminals $\mathcal{N}_R^{\uparrow}$ and rewrite rules $\mathcal{R}_R^{\uparrow}$ containing $S_R^{\uparrow} : o \to o$ as a parameterized start symbol. For any continuation tree $q : o$, the tree generated by $S_R^{\uparrow}\, q$ can be viewed as replacing every terminating $\bullet$ of the return-clause effect (i.e. generated by S_R) by q.

The construction is as follows. Let $q : o$ be a fresh variable, and introduce $F^{\uparrow} : o \to \kappa_1 \to \cdots \to \kappa_n \to o$ for every $F : \kappa_1 \to \cdots \to \kappa_n \to o \in \mathcal{R}_R$. Define $\mathcal{C}_q$ structurally on terms:

$$\begin{aligned}\mathcal{C}_q(\bullet) &= q,\\ \mathcal{C}_q(F) &= F^{\uparrow}\, q,\\ \mathcal{C}_q(a) &= a \qquad (a \neq \bullet),\\ \mathcal{C}_q(x) &= x,\\ \mathcal{C}_q(t\,u) &= \mathcal{C}_q(t)\,\mathcal{C}_q(u)\end{aligned}$$

for non-terminals F, terminals a except $\bullet$, and variables x. Every rule $F\, x_1 \cdots x_n \to e$ is replaced by

$$F^{\uparrow}\, q\, x_1 \cdots x_n \to \mathcal{C}_q(e).$$

Thus the transformation replaces every occurrence of $\bullet$ by q, adds q as the first argument of every non-terminal occurrence, and adds the same argument to every rule's left-hand side.

Likewise, let $\mathcal{N}_O^{\uparrow}, \mathcal{R}_O^{\uparrow}$ be the parameterization for G_O. In particular, a start rule $S_O\, \xi \to e$ becomes

$$S_O^{\uparrow}\, q\, \xi \to \mathcal{C}_q(e).$$

In other words, $S_O^{\uparrow}\, q\, x$ instantiates the scoped parameter ξ with x and replaces each ordinary termination by q.

Ultimately, the parameterized recursion schemes have the following start symbols:

$$S_R^{\uparrow} : o \to o, \qquad S_O^{\uparrow} : o \to (o \to o) \to o.$$

Handling. Now let $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ describe the computation to be handled. Define a lifting of simple types by

$$o^{\dagger} = o \to o, \qquad (\kappa_1 \to \kappa_2)^{\dagger} = \kappa_1^{\dagger} \to \kappa_2^{\dagger}.$$

Thus a ground effect with $\bullet$ at every leaf is translated to a parameterized one that accepts the effect following it. For every variable $x : \kappa$ and non-terminal $F : \kappa$, introduce a translated copy $x^{\dagger} : \kappa^{\dagger}$ and $F^{\dagger} : \kappa^{\dagger}$. The type-directed translation $\langle t \rangle$, homomorphic with respect to application, is defined by

$$\begin{aligned}&\langle \bullet \rangle = S_R^{\uparrow}, && \langle + \rangle = \mathrm{Plus}, && \langle \mathbf{op} \rangle = \mathrm{Op},\\ &\langle a \rangle = A \quad (a \neq \mathbf{op}), && \langle x \rangle = x^{\dagger}, && \langle F \rangle = F^{\dagger},\\ &\langle t\,u \rangle = \langle t \rangle\, \langle u \rangle\end{aligned}$$

with the following auxiliary rules:

$$\begin{aligned} \mathrm{Plus}\, x\, y\, q &\to +\,(x\, q)\,(y\, q), \\ \mathrm{Op}\, x\, q &\to S_O^{\uparrow}\, q\, x, \\ A\, x\, q &\to a\,(x\, q) \qquad (a \neq \mathbf{op}). \end{aligned}$$

Next, for every rule $F\, x_1 \cdots x_m \to t$ in $\mathcal{R}$, add

$$F^{\dagger}\, x_1^{\dagger} \cdots x_m^{\dagger}\, q \to \langle t \rangle\, q.$$

Let $\underline{\mathcal{N}}$ contain $\underline{S}$, the clause non-terminals $\mathcal{N}_R^{\uparrow} \uplus \mathcal{N}_O^{\uparrow}$, Plus, Op, and every $F^{\dagger}$ and A. Let $\underline{\mathcal{R}}$ contain $\mathcal{R}_R^{\uparrow} \uplus \mathcal{R}_O^{\uparrow}$, the translated source rules, the auxiliary rules, and the start rule below. The handled scheme $\underline{G}_O^R = \langle \Sigma, \underline{\mathcal{N}}, \underline{\mathcal{R}}, \underline{S} \rangle$ starts with

$$\underline{S} \to S^{\dagger}\ \bullet\,.$$

When the translation reaches **op**, the variable x describes the remainder of the source computation under the handler. The rule for Op supplies that same x as the instantiation of ξ. Since each source-level invocation of k contributes an occurrence of ξ to O, any number of invocations are handled. When a source branch instead reaches $\bullet$, the non-terminal $S_R^{\uparrow}$ invokes the return clause. If the source tree contains no **op**, the construction consequently agrees with $G \cdot G_R$. The translation raises the order of the involved simple types by at most one.

For the directory traversal example, write **d** for **descend**, **e** for **enter**, **c** for **getChild**, and **l** for **leave**. The operation clause has the effect $O = \mathbf{e}\,(\mathbf{c}\,(\xi\,(\mathbf{l}\,\bullet)))$ under $\xi : o \to o$. After parameterizing the two clause schemes, the relevant rules are

$$\begin{aligned} S &\to \mathbf{d}\,(+\,S\,\bullet) && \text{for } G, \\ S_R^{\uparrow}\, q &\to q && \text{for } G_R, \\ S_O^{\uparrow}\, q\, \xi &\to \mathbf{e}\,(\mathbf{c}\,(\xi\,(\mathbf{l}\, q))) && \text{for } G_O. \end{aligned}$$

By the construction above, we have that the HORS $\underline{G}_O^R$ has the following rules:

$$\begin{aligned} \underline{S} &\to S^{\dagger}\,\bullet, \\ S^{\dagger}\, q &\to \mathbf{D}\,(\mathrm{Plus}\, S^{\dagger}\, S_R^{\uparrow})\, q, \\ \mathbf{D}\, x\, q &\to S_O^{\uparrow}\, q\, x, \\ \mathrm{Plus}\, x\, y\, q &\to +\,(x\, q)\,(y\, q), \\ S_R^{\uparrow}\, q &\to q, \\ S_O^{\uparrow}\, q\, \xi &\to \mathbf{e}\,(\mathbf{c}\,(\xi\,(\mathbf{l}\, q))). \end{aligned}$$

Its terminating branches have traces $\{(\mathbf{ec})^n \mathbf{l}^n : n \geq 1\}$, exactly the nested resource behavior shown in 1.

3 Discussion

This section discusses the limitations and outstanding proof obligations of the present work.

Well-typedness of HORS constructions and temporal effect soundness. It remains to show that union $G_1 + G_2$, concatenation $G_1 \cdot G_2$, and handling $\underline{G}_O^R$ are well-typed recursion schemes and faithful to the corresponding program semantics.

Temporal effect soundness can be stated via the operational semantics. We can define $\mathrm{Tr}\, M \subseteq \Sigma^{\leq \omega}$ as the set of finite and infinite traces obtained by executing M according to its semantics. Let $\mathrm{Tr}[\![E]\!]$ be the finite and infinite traces of the value tree generated by HORS E, ignoring branch nodes $+$ and the terminal marker $\bullet$. If $\varnothing; \Gamma \vdash M : A \mid E$, temporal effect soundness is the statement

$$\mathrm{Tr}\, M \subseteq \mathrm{Tr}[\![E]\!].$$

Progress lemma. If $\varnothing; \varnothing \vdash M : A \mid E$, then one of the following applies: M is **return** V for some V; M is $\mathcal{E}[\mathbf{op}\, V]$ with $\mathbf{op} \notin \mathcal{E}$; or $M \longrightarrow M'$ for some M'. The second form exposes an operation not handled by any enclosing matching handler. No other closed, well-typed computation is stuck.

Preservation with residual effects. Reduction preserves the result type. If $\Gamma \vdash M : A \mid E$ and $M \longrightarrow M'$, then there is an E' such that $\Gamma \vdash M' : A \mid E'$, and the tree denoted by E' is an admissible residual of the tree denoted by E:

$$\mathrm{Tree}(E') \preceq_{\mathrm{res}} \mathrm{Tree}(E).$$

More precisely, the residual relation is defined from the operational step and the evaluation context. Ordinary same-effect preservation would be false for the exact system: for example, selecting one arm of a conditional changes its effect from $E + F$ to the effect of the selected arm. Alternatively, one can define subeffecting $E' \leq E$ and formulate the theorem as preservation of both types and effects.

Type safety corollary. Progress and residual preservation imply that every reduct of a closed, well-typed computation has the same result type and an admissible residual effect. Consequently, every maximal finite execution ends in either a returned value or an unhandled operation, never in another stuck form. Combined with temporal effect soundness, the operation trace of every finite or infinite execution is represented by a branch of the assigned HORS.

Decidability of type checking. We expect type checking to be decidable, following the Hindley–Milner system for algebraic effects and handlers of Kammar and Pretnar [3], but this still requires an algorithmic presentation of instantiation, generalization, unification, and recursive

bindings. Model checking a fixed HORS against an appropriate logical or automata specification is decidable [6]. Nonetheless, arbitrary language equivalence and inclusion between HORS are known to be undecidable in general—HORS generalize context-free grammars, for which both equivalence and inclusion are undecidable. A subeffect relation therefore cannot rely on deciding arbitrary HORS language inclusion.

4 Future Directions

The immediate goal is to finish the current core calculus: prove progress and preservation, type the union, concatenation, and handler transformations, and establish effect soundness using HORS. Several further directions remain.

Shallow handlers. The core calculus uses *deep* handlers, which implicitly reinstall themselves when a captured continuation is invoked. Shallow handlers [2] provide more flexibility by making this rehandling policy explicit. Their HORS constructions are more complex because they distinguish behavior that is still under the handler from behavior after the handler has been discharged. With recursive functions, a program can recover deep handlers by explicitly applying a shallow handler again when it invokes the continuation.

Trace-based versus tree-based subeffecting. While traditional trace-based temporal effect systems [11] define subeffecting as trace inclusion $\mathrm{Tr}[\![E]\!] \subseteq \mathrm{Tr}[\![E']\!]$, two tree-structured effect descriptions can exhibit different semantics despite having the same traces. Consider

$$P = \mathbf{enter}\,(\mathbf{read}\,\bullet + \mathbf{write}\,\bullet)$$
$$Q = \mathbf{enter}\,(\mathbf{read}\,\bullet) + \mathbf{enter}\,(\mathbf{write}\,\bullet).$$

Their trace sets are equal, but the choice in P occurs after **enter**, whereas the choice in Q occurs before it. A handler for **enter** that invokes the captured continuation twice can produce all four read/write pairs from P, but only the two matching pairs from Q. Hence trace equivalence is not preserved by handling. This counterexample shows that forgetting the branching structure breaks the semantics under handlers, and a tree-based definition of subeffecting is required instead.

Effect polymorphism. Effect polymorphism is a later extension. Combining polymorphic effects, let polymorphism, and handlers is known to require care in order to preserve decidability: existing approaches restrict the interaction of resumptions [9] or restrict operation signatures [10].

Refinement types and effects. The current type-and-effect system can be extended with *refinement types* and *value-indexed effects* to enable more accurate reasoning about program behavior. Answer refinement modification tracks value-sensitive changes caused by handlers [4], while answer effect modification tracks temporal effects around delimited control [12]. These techniques suggest how to enrich the result and continuation types. A finite-domain first step, such as Boolean-indexed effects approximated by explicit branching, may avoid prematurely requiring a HORS to inspect arbitrary infinite-domain values.

Regularity-preserving fragments. Our example program deliberately uses the continuation non-tail-recursively: the **leave** operation follows the invocation of k. It remains to characterize restrictions under which handling preserves a lower-order effect class. One candidate is the tail-resumptive fragment in which the only occurrence of the continuation ξ is of the form $\xi\,\bullet$.

5 Related Work

Plotkin and Power characterize algebraic operations and generic effects [7]; Plotkin and Pretnar develop handlers as modular interpretations of those operations [8]. Our operational contribution is not a new notion of handler, but a temporal account of standard handling semantics. The polymorphic discipline builds on the observation that algebraic effects and handlers do not require a value restriction [3].

HORS provide finite generators for infinite higher-order trees and support powerful model-checking techniques [6] [5]. We use them as effect descriptions and transform the generator itself to model a handler's manipulation of a continuation.

Algebraic temporal effects give an abstract account of safety and liveness for higher-order programs, with instances based on regular and fixpoint effects [11]; they do not target algebraic handlers. Answer-effect modification does address continuation manipulation, but for delimited control operators and with value-dependent temporal verifica-

tion [12]. A handler selects clauses by operation name, whereas a delimited-control capture selects a surrounding delimiter; both lines of work nevertheless show that the effect of a captured continuation must be represented explicitly. Answer refinement modification develops the corresponding refinement-type perspective for algebraic handlers [4].

6 Conclusion

We developed a type-and-effect system for algebraic effect handlers in a polymorphic calculus with recursion. In this system, we use HORS as finite descriptions of possibly infinite, branching temporal effects. Structural constructions model conditional choice, sequencing, effect handling. We illustrate why this extra expressiveness is needed via examples, and also argue why trace sets alone lose observable branching information. The present contribution is the calculus and these HORS constructions; proving their well-typedness, preservation, type safety, and temporal soundness is the immediate next step.

References

[1] Clairambault, P. and Murawski, A. S.: Böhm Trees as Higher-Order Recursive Schemes, Vol. 24, pp. 91–102. Artwork Size: 12 pages, 643636 bytes ISBN: 9783939897644 Medium: application/pdf.

[2] Hillerström, D. and Lindley, S.: Shallow Effect Handlers, *Programming Languages and Systems*, Ryu, S.(ed.), Vol. 11275, Springer International Publishing, pp. 415–435. Series Title: Lecture Notes in Computer Science.

[3] Kammar, O. and Pretnar, M.: No value restriction is needed for algebraic effects and handlers, Vol. 27, pp. e7.

[4] Kawamata, F., Unno, H., Sekiyama, T., and Terauchi, T.: Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers.

[5] Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs, *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 416–428.

[6] Ong, L.: Higher-Order Model Checking: An Overview, *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, IEEE, pp. 1–15.

[7] Plotkin, G. and Power, J.: Algebraic Operations and Generic Effects, Vol. 11, No. 1, pp. 69–94.

[8] Plotkin, G. and Pretnar, M.: Handlers of Algebraic Effects, *Programming Languages and Systems*, Castagna, G.(ed.), Vol. 5502, Springer Berlin Heidelberg, pp. 80–94. Series Title: Lecture Notes in Computer Science.

[9] Sekiyama, T. and Igarashi, A.: Handling polymorphic algebraic effects, Vol. 11423, pp. 353–380.

[10] Sekiyama, T., Tsukada, T., and Igarashi, A.: Signature Restriction for Polymorphic Algebraic Effects, Vol. 34, pp. e7.

[11] Sekiyama, T. and Unno, H.: Algebraic Temporal Effects: Temporal Verification of Recursively Typed Higher-Order Programs, Vol. 9, pp. 2306–2336.

[12] Sekiyama, T. and Unno, H.: Temporal Verification with Answer-Effect Modification.

[13] Sivaramakrishnan, K. C., Dolan, S., White, L., Kelly, T., Jaffer, S., and Madhavapeddy, A.: Retrofitting Effect Handlers onto OCaml, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, New York, NY, USA, Association for Computing Machinery, 2021, pp. 206–221.

[14] Statman, R.: On the λ Y calculus, Vol. 130, No. 1, pp. 325–337.

[15] Xie, N. and Leijen, D.: Effect handlers in Haskell, evidently, *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, Haskell 2020, New York, NY, USA, Association for Computing Machinery, 2020, pp. 95–108.