Type-Safe Code Generation with Algebraic Effects and Handlers

Kanaru Isoda
University of Tsukuba
Tsukuba, Japan
isoda@logic.cs.tsukuba.ac.jp

Ayato Yokoyama
University of Tsukuba
Tsukuba, Japan
yokoyama@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
University of Tsukuba
Tsukuba, Japan
kameyama@acm.org

Abstract

Staged computation is a means to achieve maintainability and high performance simultaneously, by allowing a programmer to express domain-specific optimizations in a high-level programming language. Multi-stage programming languages such as MetaOCaml provide a static safety guarantee for generated programs by sophisticated type systems provided that program generators have no computational effects. Despite several studies, it remains a challenging problem to design a type-safe multi-stage programming language with advanced features for computational effects.

This paper introduces a two-stage programming language with algebraic effects and handlers. Based on two novel principles 'handlers as future-stage binders' and 'handlers are universal', we design a type system and prove its soundness. We also show that our language is sufficiently expressive to write various effectful staged computations including multi-level let-insertion, which is a key technique to avoid code duplication in staged computation.

CCS Concepts: • **Theory of computation → Control primitives**; **Type structures**; • **Software and its engineering → General programming languages**.

Keywords: code generation, algebraic effects and handlers, scope extrusion, type soundness

ACM Reference Format:
Kanaru Isoda, Ayato Yokoyama, and Yukiyoshi Kameyama. 2024. Type-Safe Code Generation with Algebraic Effects and Handlers. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24), October 21–22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3689484.3690731

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '24, October 21–22, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1211-1/24/10
https://doi.org/10.1145/3689484.3690731

1 Introduction

Staged computation is a means to achieve high performance and high modularity simultaneously, and has been applied to generate highly efficient code in various application domains. While other similar technologies such as partial evaluation and compilation generate programs fully or semi-automatically, staged computation allows a human programmer to control the process of program generation. This aspect has a merit that custom, domain-specific optimizations can be exploited, but also has a demerit that one can easily generate ill-structured, ill-scoped or ill-typed programs. The scope extrusion problem refers to the problem that well-scoped programs evaluate to ill-scoped (or wrongly scoped) programs. To prevent these problems, static guarantee for safety properties is strongly desired.

The pioneering works in staged computation [4, 5] gave type systems based on modal logic, and proved type soundness for them, which subsume well-scopedness and well-typedness of generated programs. These works have practical impacts, for example, the first design of MetaOCaml was based on a variant of Taha and Nielsen's type system[19] that is a descendant of these pioneering type systems. Unfortunately, all of these type systems suffer from the problem that they do not allow computational effects in program generators, while most practical multi-stage programs need computational effects to handle exceptions, memoize codes, or avoid the code-duplication problem.

Statically guaranteeing safety properties for generated programs in the presence of computations effects is recognized as a hard problem. While several solutions have been proposed [7, 10, 14, 20], all these works focused on one particular set of effect-raising primitives such as shift/reset, shift0/reset0, and reference cells, and designed dedicated type systems. They cannot be used to solve a general solution for the problem, where various computational effects are used in staged programs.

Algebraic effects and handlers [16, 17] have become a standard tool for expressing various computational effects uniformly and modularly. Invented in an algebraic theory, they have become practical, and a few main-stream programming languages including OCaml[1] have implemented them. It is natural to ask if one can design a typed calculus

[1] https://ocaml.org

for staged computation with algebraic effects and handlers, which enjoys type soundness.

This paper presents a typed two-stage programming language where algebraic effects and handlers can be used for program generation, and proves type soundness for it.

An important ingredient of our type system is a *refined environment classifier* [10] (*classifier* for short), which is a refinement of an *environment classifier* [19]. A classifier represents a lexical scope of generated variables, and the eigenvariable condition of a classifier implies well-scopedness.

Our design principles are two-fold: The first one is the "handlers as future-stage binders" principle, which means that an algebraic effect handler should be treated as a binder for a future-stage (generated) variable: when we use an algebraic handler, it may insert a code-level binder at the place of the handler, and the handled expression must be aware of this binder[2]. Hence, to prepare for this possibility, the typing rule for with-handle must introduce a new classifier corresponding to the (possibly generated) lexical scope, just like the code-level lambda abstraction does. The second one is the "handlers are universal" principle, which simply means that a handler type should be universally quantified by classifiers. This universality is necessary since handlers are first-class values (they may be stored or passed to a function) so that the definition of a handler cannot know the classifier when it is used. By virtue of these principles, our type system enjoys the type soundness property, which is, to our knowledge, the first such result for the staged calculus with algebraic effects and handlers.

Our leading example is nested let-insertion, an important technique to avoid duplicated code in generated programs. We show this technique can be implemented in our calculus. The same problem was addressed by Kiselyov [9], using a dedicated primitive that dynamically inserts let expressions.

The contribution of this paper can be summarized as follows:

- We propose two principles to soundly formulate handlers in staged calculi.
- We design a two-stage language with algebraic effects and handlers, and a type system for it, based on the two principles.
- We show that this language is sufficiently expressive by showing various programming examples.
- We prove that type soundness of our type system holds.

The rest of this paper is organized as follows: Section 2 informally gives a few programming examples in our language, and explains the issues addressed in this paper. Section 3 presents our language for staged computation with algebraic effects and handlers. Section 4 gives a type system for it, and Section 5 gives a few examples of typing. Section 6 states our main theorems and several key lemmas to ensure type soundness of our calculus. In Section 7, we compare our work with related work and Section 8 concludes the paper.

[2]Our principle has nothing to do with the lexically-bound algebraic handlers [1], which introduce lexical scopes for operations. A handler in our calculus has a *dynamic* extent.

2 Programming Examples

We will show several examples in our calculus that illustrate staged computation with algebraic effects and handlers.

2.1 Basics

Our calculus is based on the fine-grain call-by-value calculus [12], which distinguishes values from computations, for instance:

Values: $3,\ \mathsf{true},\ \lambda x.\, x + 5$

Computations: $\mathbf{return}\ 3,\ 3 + 5,\ \mathbf{let}\ x \leftarrow 3 + 5\ \mathbf{in}\ x + 5$
$\mathbf{if}\ \mathsf{true}\ \mathbf{then}\ \mathbf{return}\ 0\ \mathbf{else}\ 2 * 3$

A value V is turned into a computation $\mathbf{return}\ V$, and only values can be arguments of primitive operators such as addition.

2.2 Staged Programs

Since Lisp, most programming languages for staged computation have been using Quasi-quotation for building code expressions, while a few others use Code combinators, and we use the latter style.

The following examples perform the same computation in two styles:

Quasi-quotation: $\mathbf{let}\ x \leftarrow \langle 3 \rangle\ \mathbf{in}\ \langle {\sim}x * 5 \rangle$
Code combinator: $\mathbf{let}\ x \leftarrow \langle 3 \rangle\ \mathbf{in}\ x \mathbin{\underline{*}} \langle 5 \rangle$

The first line uses quasi-quotation where $\langle e \rangle$ is a code expression which, when executed, generates a code, and ${\sim}e'$ splices the value of e' (which should be a code value) into the surrounding code. It evaluates to the code $\langle 3 * 5 \rangle$.

The second line has no splice operators, and instead uses a code combinator $\underline{*}$, which is a *staged* variant of the multiplication operator. It takes two arguments $\langle 3 \rangle$ and $\langle 5 \rangle$, and builds a code $\langle 3 * 5 \rangle$. The code-combinator style is used in a few languages such as Scala LMS (lightweight modular staging).

The staged version of a binder is more involved:

Quasi-quotation: $\langle \lambda x.\, x * 5 \rangle$
Code combinator: $\underline{\lambda} x.\, x \mathbin{\underline{*}} \langle 5 \rangle$

On the second line, the underlined lambda is the code combinator for abstraction, and the above examples evaluates to $\langle \lambda x_1.\, x_1 * 5 \rangle$ where x_1 is a generated variable.

Our calculus uses code combinators since it is more convenient to express additional constraints about the lexical scope for the future-stage binders. This choice is not essential from the viewpoint of expressivity.

2.3 Algebraic Effects and Handlers

Algebraic effects and handlers [16, 17] are becoming popular among researchers and practitioners in programming languages. Their strength lies in their ability to express various user-defined computational effects such as state, exception, non-determinism, and resumption uniformly and modularly.

Let us define handlers H_{id}, H_{exc}, and H_{nondet} as follows:

$$H_{\mathsf{id}} \coloneqq \{\mathbf{return}\ x \mapsto \mathbf{return}\ x\}$$
$$H_{\mathsf{exc}} \coloneqq \{\mathsf{raise}\ x\ k \mapsto \mathbf{return}\ x\} \uplus H_{\mathsf{id}}$$
$$H_{\mathsf{nondet}} \coloneqq \{\mathsf{choose}\ x\ k \mapsto \mathbf{throw}\ k\ \mathsf{true}; \mathbf{throw}\ k\ \mathsf{false}\} \uplus H_{\mathsf{id}}$$

The handler H_{id} does nothing, while H_{exc} handles the operation raise and can be used as follows:

$$\mathbf{with}\ H_{\mathsf{exc}}\ \mathbf{handle}\ \mathbf{let}\ x \leftarrow \mathbf{do}\ \mathsf{raise}\ 5\ \mathbf{in}\ 3 + x$$
$$\rightsquigarrow \mathbf{return}\ 5$$

where **with** h **handle** M installs a handler h, and evaluates M while operations invoked in M will be handled by h.

H_{nondet} handles the operation choose. When choose is called, its first parameter x is bound to its actual argument, and the second parameter k is bound to the delimited continuation (the evaluation context) up to the nearest handler that handles the invoked operation. The delimited continuation bound to k is restored by **throw**. Since choose calls k twice for different arguments, it has the effect of non-deterministic choice:

```
with H_nondet handle
  if true then print 3 else print 5
with H_nondet handle
  let x ← do choose () in if x then print 3 else print 5
```

While the first code will print 3 only, the second one will print 3 and 5.

2.4 Let-Insertion

Computational effects are useful, and sometimes indispensable, to generate concise and efficient programs. Let-insertion is a technique to avoid the code-duplication problem, that can be implemented by algebraic effects and handlers:

$$H_{\mathsf{ins}} \coloneqq \{\mathsf{insert}\ x\ k \mapsto \underline{\mathbf{let}}\ y \leftarrow x\ \underline{\mathbf{in}}\ \mathbf{throw}\ k\ y\} \uplus H_{\mathsf{id}}$$

When **do** insert $\langle 3 + 5\rangle$ is evaluated, k is bound to the delimited continuation, x is bound to $\langle 3 + 5\rangle$, and the body of insert is evaluated. Then, it inserts a let-expression at the place where the handler has been installed, and restores the delimited continuation with the argument y. Hence, the surrounding evaluation context and the let-expression are swapped.

To understand how it works, consider the expression:

$$\mathbf{let}\ x \leftarrow \mathbf{return}\ \langle \mathit{fact}\ 10\rangle\ \mathbf{in}\ (\lambda y.\ y \underline{*} y)\ x$$

where *fact* computes the factorial of its argument. It evaluates to $\langle(\mathit{fact}\ 10) * (\mathit{fact}\ 10)\rangle$, which has duplicated code. We can use the above handler to avoid it:

$$\mathbf{with}\ H_{\mathsf{ins}}\ \mathbf{handle}$$
$$\mathbf{let}\ x \leftarrow \mathbf{do}\ \mathsf{insert}\ \langle \mathit{fact}\ 10\rangle\ \mathbf{in}\ (\lambda y.\ y \underline{*} y)\ x$$

This expression evaluates to a code without duplication: $\langle \mathbf{let}\ y_1 \leftarrow \mathit{fact}\ 10\ \mathbf{in}\ y_1 * y_1\rangle$.

The above example shows a potential risk for scope extrusion. Namely, if the code to be inserted (*fact* 10 in the above example) contains a locally-bound variable, let-inserting such a code would lead to an open code. For instance, the following code:

$$\mathbf{with}\ H_{\mathsf{ins}}\ \mathbf{handle}$$
$$\underline{\lambda} z.\ \mathbf{let}\ x \leftarrow \mathsf{insert}\ \langle z \underline{+} 10\rangle\ \mathbf{in}\ (\lambda y.\ y \underline{*} y)\ x$$

would evaluate to $\langle \mathbf{let}\ y_1 \leftarrow z_1 + 10\ \mathbf{in}\ \lambda z_1.\ y_1 * y_1\rangle$, which does not compile. This is the scope extrusion problem and our type system rules out such a program.

2.5 Multiple Let-Insertions

We sometimes need multiple destinations of let-insertion. Consider the following code:

$$\langle \mathbf{for}\ i = 1\ \mathbf{to}\ N_1$$
$$\mathbf{for}\ j = 1\ \mathbf{to}\ N_2$$
$$a.(idx) \leftarrow e\rangle$$

where $a.(idx)$ dereferences the idxth element of an array a. If the expression e is closed, we can let-insert it outside of the two for-loops, but if it contains i freely but no j, we should let-insert it to the place between the two for-loops.

We can easily mimic this behavior by handlers as:

$$\mathbf{with}\ H_{\mathsf{ins1}}\ \mathbf{handle}\ \underline{\mathbf{for}}\ i = 1\ \underline{\mathbf{to}}\ N_1$$
$$\mathbf{with}\ H_{\mathsf{ins2}}\ \mathbf{handle}\ \underline{\mathbf{for}}\ j = 1\ \underline{\mathbf{to}}\ N_2$$
$$\mathbf{let}\ y \leftarrow \mathbf{do}\ \mathsf{insertN}\ \langle e\rangle\ \mathbf{in}$$
$$\underline{\mathit{ArrayAssign}}(\langle a\rangle, \langle idx\rangle, y)$$

where $\underline{\mathit{ArrayAssign}}$ is a code combinator for array assignment. We assume that, for $N = 1, 2$, H_{insN} handles the operation insertN. On the third line, we set $N = 1$ if e has no i and j freely, and $N = 2$ if e has i but no j freely. It is tedious and error-prone for a human to check these constraints, and this is the point where our type system helps.

Oishi and Kameyama [14] used the control operators **shift0** and **reset0** [3] to implement nested let-insertion as:

$$\mathbf{reset0}\ (\underline{\mathbf{for}}\ i = \cdots$$
$$\mathbf{reset0}\ (\underline{\mathbf{for}}\ j = \cdots$$
$$\underline{\mathit{ArrayAssign}}(\langle a\rangle, \langle idx\rangle, M)))$$

We set M as $\mathbf{shift0}\ k_2 \rightarrow \underline{\mathbf{let}}\ y \leftarrow \langle e\rangle\ \underline{\mathbf{in}}\ \mathbf{throw}\ k_2\ \langle y\rangle$ if we insert a let expression between the two for-loops, while M

should be **shift0** $k_2 \rightarrow$ **shift0** $k_1 \rightarrow \underline{\textbf{let}}\ y \leftarrow \langle e \rangle\ \underline{\textbf{in}} \cdots$ if we insert one to the outermost place.[3] Namely, they need to iterate **shift0**s where the number of **shift0**s varies depending on the destination of let-insertion, which is clearly tedious. Even worse, the number must be known statically, while the number of **reset0**s changes dynamically. [4] On the contrary, we can write the above example easily, thanks to the *named* feature of algebraic effects and handlers.

However, going from nameless control operators (**shift0** and **reset0**) to named ones brought another technical issue. Consider the following program:

$$\begin{array}{l}\textbf{with}\ H_{\mathsf{ins1}}\ \textbf{handle}\\ \quad\textbf{with}\ H_{\mathsf{ins2}}\ \textbf{handle}\\ \quad\quad M\end{array}$$

If the body M is insert2 (insert1 C), the above term reduces to $\langle \textbf{let}\ y_1 \leftarrow C\ \textbf{in let}\ y_2 \leftarrow y_1\ \textbf{in}\ y_2 \rangle$, which is not problematic. However, if the body M is insert1 (insert2 C), the term reduces to $\langle \textbf{let}\ y_1 \leftarrow y_2\ \textbf{in let}\ y_2 \leftarrow C\ \textbf{in}\ y_2 \rangle$, which has a free variable y_2, causing a problem. Our type system must distinguish these two cases appropriately.

2.6 Other Useful Effects in Staged Programs

We can use other effects, such as non-deterministic choices and exceptions, in staged programs. For instance, we can define the handler H_{col}:

$$\begin{array}{l}H_{\mathsf{col}} \coloneqq \{\mathsf{emit}\ x\ k \mapsto\\ \qquad\textbf{let}\ y \leftarrow \textbf{throw}\ k\ x\ \textbf{in return}\ \mathsf{cons}(x, y),\\ \quad\textbf{return}\ x \mapsto \textbf{return}\ \mathsf{nil}\}\end{array}$$

where *cons* is the list-cons operation and *nil* is the empty list. We can use it to collect all emitted values into a list:

$$\begin{array}{l}\quad\textbf{with}\ H_{\mathsf{col}}\ \textbf{handle}\\ \quad\quad\textbf{do}\ \mathsf{emit}\ 1; \textbf{do}\ \mathsf{emit}\ 2; \textbf{do}\ \mathsf{emit}\ 3\\ \rightsquigarrow (1, 2, 3)\end{array}$$

By combining it with non-deterministic choices, we can generate a list of differently generated programs as follows:

$$\begin{array}{l}\quad\textbf{with}\ H_{\mathsf{col}}\ \textbf{handle}\\ \quad\textbf{with}\ H_{\mathsf{nondet}}\ \textbf{handle}\\ \quad\quad\textbf{let}\ x \leftarrow \textbf{do}\ \mathsf{choose}\ ()\ \textbf{in}\\ \quad\quad\textbf{let}\ y \leftarrow \mathit{codegen}\ x\ \textbf{in}\\ \quad\quad\textbf{do}\ \mathsf{emit}\ y\\ \rightsquigarrow (\langle \mathit{code}_1 \rangle, \langle \mathit{code}_2 \rangle)\end{array}$$

where *codegen* is a code-generating function, which generates a different code depending on its argument.

[3] By iterating **shift0**, we can reach at the outermost **reset0**.

[4] Note that the number of **reset0**s is not the de Bruijn index, since there is no built-in mechanism to adjust the number of **shift0**s when an expression moves to the scope of **reset0**.

2.7 Scope Extrusion Problem

We have already mentioned that using algebraic effects and handlers in staged programs has a risk of scope extrusion, which means that a generated code may have free variables, leading to a compiler error. This risk may show up for let-insertion, exception, collecting results, states (for memoizing open codes), and many other effects (but non-deterministic choices alone). The simplest example is shown below:

$$\begin{array}{l}\quad\textbf{with}\ H_{\mathsf{exc}}\ \textbf{handle}\ \underline{\lambda} y.\ \textbf{do}\ \mathsf{raise}\ \langle y \rangle\\ \rightsquigarrow \langle y_1 \rangle\end{array}$$

The expression before execution is closed, but it evaluates to $\langle y_1 \rangle$, a code with a free variable y_1. Avoiding the scope extrusion problem has been recognized as a hard problem in staged languages, and the subject of this paper is to solve it in the presence of algebraic effects and handlers.

3 Calculus

In this section, we present a calculus for two-stage programming with algebraic effects and handlers, leaving its type system to the next section.

3.1 Syntax

Fig. 1 defines the syntax of our calculus. It is a *mixture* of <NJ> by Kiselyov et al. [10], and algebraic effects and handlers [8, 17]. We organize them based on the fine-grain call-by-value [12], which syntactically distinguishes values from computations.

Values

$$V, W ::= x \mid {}^0K \mid \lambda x.\, M \mid \kappa x.\, M \mid \langle M^1 \rangle$$

Computations

$$\begin{array}{rcl}M, N &::=& {}^nK\ V_1 \cdots V_n \mid V\ W \mid \textbf{throw}\ V\ W\\ &\mid& \textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N \mid \textbf{return}\ V\\ &\mid& \textbf{let}\ x \leftarrow M\ \textbf{in}\ N \mid \textbf{do}\ \mathit{op}\ V\\ &\mid& \textbf{with}\ H\ \textbf{handle}\ M \mid \underline{\lambda} x.\, M \mid \underline{\underline{\lambda}} x.\, M\end{array}$$

Handlers

$$H ::= \{\textbf{return}\ x \mapsto M\} \mid H \uplus \{\mathit{op}\ x\ k \mapsto M\}$$

Level-1 Values

$$V^1, W^1 ::= x \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.\, M^1$$

Level-1 Computations

$$\begin{array}{rcl}M^1, N^1 &::=& V^1\ W^1 \mid \textbf{if}\ V^1\ \textbf{then}\ M^1\ \textbf{else}\ N^1 \mid \textbf{return}\ V^1\\ &\mid& \textbf{let}\ x \leftarrow M^1\ \textbf{in}\ N^1\end{array}$$

Figure 1. Syntax

Values are either a variable, a constant ${}^0K$, lambda abstraction, a continuation, or a code value. The continuation $\kappa x.\, M$ is similar to lambda abstraction, but we need a special

treatment for continuations. The superscript 1 in the code value $\langle M^1 \rangle$ indicates that the subexpression is a future-stage (or level-1) computation. Computations are mostly standard in the standard lambda calculus except those for algebraic effects and handlers and for staged programs, which we will explain shortly.

For staged programs, our calculus uses *code combinators*, which are expressed as underlined constants given later and the underlined lambda abstraction $\underline{\lambda} x.\, M$. The latter builds a code for a lambda abstraction. The form $\underline{\underline{\lambda}} y.\, N$ is used as an intermediate term only, and should not be used in the source term. It will be explained in the operational semantics. Level-1 values and computations are those used at the future stage. For simplicity, we restrict that level-1 terms should not contain code combinators or effectful computations. Thus, the set of level-1 values (computations, resp.) forms a proper subset of values (computations, resp.).

For algebraic effects and handlers, we have two major constructs: the first one is a computation **do** *op* V, which calls an operation *op* with an argument V, and the second is a handler H. Each handler contains exactly one *value handler* $\{\textbf{return}\ x \mapsto M\}$, that specifies the return value when the handled computation returns without calling an operation. A handler can specify the behavior of an arbitrary number of operations $\{op\ x\ k \mapsto N\}$, where we assume that the operation names *op* are mutually distinct. When an operation is called (namely, an effect is raised), its argument is bound to x, the continuation[5] up to the handler is captured and bound to k, and the corresponding behavior of the operation specified by the handler is invoked.

For technical reasons, we need special treatment for a captured continuation syntactically: $\kappa x.\, M$ is a continuation, and **throw** $V\ W$ is a resumption of a continuation V against an argument W.

The rest of the syntax is mostly standard. As our calculus is based on the fine-grain call-by-value calculus, we have **return** V, which turns a value V to an atomic computation, and **let** $x \leftarrow M$ **in** N, which represents a sequential computation of M and N where x is bound to the value of M.

The constants ${}^aK$ of arity a are given by:

$${}^0K ::= \mathsf{true} \mid \mathsf{false} \quad {}^1K ::= \underline{\mathsf{cbool}} \quad {}^2K ::= \underline{@} \quad {}^3K ::= \underline{\mathsf{cif}}$$

where underlined constants are code combinators. We extend the set of constants if necessary.

3.2 Operational Semantics

Fig. 2 defines the small-step operational semantics of the calculus. The relation $U; M \rightsquigarrow U'; N$ represents a one-step reduction from a computation M to N. The additional information U is a *name heap*, which tracks the use of future-stage variables, namely, variables in a generated code.

[5]Since the continuation captured by an algebraic effect is delimited to the corresponding occurrence of the handlers, it is called a *delimited* continuation. In this paper, we often call it a continuation when there is no confusion.

Most rules are standard if we ignore name heaps, which do not change except the rule E-CAbs.

The rules E-Ret and E-Do reduce the with-handle term **with** H **handle** M. The former is used when the handled term M is a value, namely, no operations have been called during the evaluation of M. Then, the value handler in H is called. The latter is used when an operation defined in H is called during the evaluation of M. The rule is the standard one for *deep* handlers.

The rules E-CAbs and E-IAbs describe how the code-combinator for lambda abstraction is reduced. The former is used when $\underline{\lambda} x.\, M$ is evaluated. We generate a future-stage variable y and substitute $\langle \textbf{return}\ y \rangle$ for the present-stage variable x, and evaluate its body where the intermediate term is represented by doubly-underlined abstraction. The name store U is used to ensure the freshness of y. The rule E-IAbs is used when the evaluation of the body of abstraction is finished with the value $\langle M^1 \rangle$. Then, it builds a code of an abstraction and returns it.

Fig. 3. defines the semantics of constants.

Below we show a concrete examples of a reduction sequence which begins with an empty name heap (denoted by $\cdot$) and a computation:

$$\begin{aligned}
&\cdot; \underline{\lambda} x.\, \underline{\mathsf{cif}}\ x\ \langle \mathsf{false} \rangle\ \langle \mathsf{true} \rangle \\
\rightsquigarrow\ &y; \underline{\underline{\lambda}} y.\, \underline{\mathsf{cif}}\ \langle \textbf{return}\ y \rangle\ \langle \mathsf{false} \rangle\ \langle \mathsf{true} \rangle \\
\rightsquigarrow\ &y; \underline{\underline{\lambda}} y.\, \textbf{return}\ \langle \textbf{let}\ x \leftarrow \textbf{return}\ y\ \textbf{in} \\
&\qquad\qquad \textbf{if}\ x\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true} \rangle \\
\rightsquigarrow\ &y; \textbf{return}\ \langle \lambda y.\, \textbf{let}\ x \leftarrow \textbf{return}\ y\ \textbf{in} \\
&\qquad\qquad \textbf{if}\ x\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true} \rangle
\end{aligned}$$

At the second step in the sequence, a future-stage variable y has been generated, and the final result is the code value $\langle \lambda y. \cdots \rangle$.

4 Type System

We present a type system for our calculus with the explanation of our design principles.

4.1 Classifiers

We will extensively use a *classifier* to track future-stage variables to prevent scope extrusion. The notion of classifiers was originally proposed by Taha and Nielsen [19] as an abstract representation of a set of future-stage variables[6]. Kiselyov et al. [10] refined it, and found that the eigen-variable condition in mathematical logic can be used to express the lexical scope of a future-stage variable, hence, scope extrusion is detected as a violation. The set of classifiers is equipped with a partial

[6]Since scope extrusion may occur only for generated future-stage variables, we do not have to tack ordinary variables (variables used at the present stage).

$\boxed{U; M \rightsquigarrow U'; N}$

$$
\begin{array}{lll}
\textsc{E-ConstApp} & U; {}^{n}K\ V_1 \cdots V_n \rightsquigarrow U; M, & \text{where } {}^{n}K\ V_1 \cdots V_n \rightsquigarrow_{\mathsf{const}} M \\
\textsc{E-True} & U; \textbf{if true then } M \textbf{ else } N \rightsquigarrow U; M & \\
\textsc{E-False} & U; \textbf{if false then } M \textbf{ else } N \rightsquigarrow U; N & \\
\textsc{E-App} & U; (\lambda x.\, M)\ W \rightsquigarrow U; M[W/x] & \\
\textsc{E-Throw} & U; \textbf{throw } (\kappa x.\, M)\ W \rightsquigarrow U; M[W/x] & \\
\textsc{E-Seq} & U; \textbf{let } x \leftarrow \textbf{return } V \textbf{ in } N \rightsquigarrow U; N[V/x] & \\
\textsc{E-Ret} & U; \textbf{with } H \textbf{ handle return } V \rightsquigarrow U; M[V/x], & \text{where } (\textbf{return } x \mapsto M) \in H \\
\textsc{E-Do} & U; \textbf{with } H \textbf{ handle } \mathcal{E}[\textbf{do } op\ V] \rightsquigarrow U; M[V/x, W/k], & \\
 & \quad \text{where } op \notin \mathsf{bl}(\mathcal{E}),\ (op\ x\ k \mapsto M) \in H, \text{ and } W = \kappa w.\, \textbf{with } H \textbf{ handle } \mathcal{E}[\textbf{return } w] & \\
\textsc{E-CAbs} & U; \underline{\lambda} x.\, M \rightsquigarrow U, y; \underline{\underline{\lambda}} y.\, M[\langle \textbf{return } y \rangle / x], & \text{where } y \notin U \\
\textsc{E-IAbs} & U; \underline{\lambda} y.\, \textbf{return } \langle M^1 \rangle \rightsquigarrow U; \textbf{return } \langle \lambda y.\, M^1 \rangle & \\
\textsc{E-Lift} & U; \mathcal{E}[M] \rightsquigarrow U'; \mathcal{E}[N], & \text{where } U; M \rightsquigarrow U'; N
\end{array}
$$

Evaluation contexts $\mathcal{E} ::= [\,] \mid \textbf{let } x \leftarrow \mathcal{E} \textbf{ in } M \mid \textbf{with } V \textbf{ handle } \mathcal{E} \mid \underline{\underline{\lambda}} y.\, \mathcal{E}$

Name heaps $U ::= \cdot \mid U, x$

Bound lables $\mathsf{bl}([\,]) = \emptyset, \quad \mathsf{bl}(\textbf{let } x \leftarrow \mathcal{E} \textbf{ in } M) = \mathsf{bl}(\mathcal{E}), \quad \mathsf{bl}(\underline{\underline{\lambda}} y.\, \mathcal{E}) = \mathsf{bl}(\mathcal{E}),$
$\mathsf{bl}(\textbf{with } H \textbf{ handle } \mathcal{E}) = \mathsf{bl}(\mathcal{E}) \cup \mathsf{dom}(H)$

Domains of handlers $\mathsf{dom}(\{\textbf{return } x \mapsto M\}) = \emptyset, \quad \mathsf{dom}(H \uplus \{op\ x\ k \mapsto M\}) = \mathsf{dom}(H) \cup \{op\}$

Figure 2. Small-step Operational Semantics

$$
\begin{array}{rll}
\underline{\mathsf{cbool}}\ L \rightsquigarrow_{\mathsf{const}} \textbf{return } \langle \textbf{return } L \rangle & & (L = \mathsf{true}, \mathsf{false}) \\
\underline{\mathsf{cif}}\ \langle M_1^1 \rangle\ \langle M_2^1 \rangle\ \langle M_3^1 \rangle \rightsquigarrow_{\mathsf{const}} \textbf{return } \langle \textbf{let } x \leftarrow M_1^1 \textbf{ in if } x \textbf{ then } M_2^1 \textbf{ else } M_3^1 \rangle & & \\
\langle M_1^1 \rangle\ \underline{@}\ \langle M_2^1 \rangle \rightsquigarrow_{\mathsf{const}} \textbf{return } \langle \textbf{let } x_1 \leftarrow M_1^1 \textbf{ in let } x_2 \leftarrow M_2^1 \textbf{ in } x_1\ x_2 \rangle & &
\end{array}
$$

Figure 3. Constant Applications

Classifiers $\gamma ::= \gamma_i \mid \gamma \cup \gamma'$

Figure 4. Classifiers

$\boxed{\Delta; \Gamma \vDash \gamma_1 \succeq \gamma_2}$

$$
\frac{}{\Delta; \Gamma \vDash \gamma \succeq \gamma} \qquad \frac{(\gamma_1 \succeq \gamma_2) \in \Delta \uplus \Gamma}{\Delta; \Gamma \vDash \gamma_1 \succeq \gamma_2}
$$

$$
\frac{\Delta; \Gamma \vDash \gamma_1 \succeq \gamma_2 \qquad \Delta; \Gamma \vDash \gamma_2 \succeq \gamma_3}{\Delta; \Gamma \vDash \gamma_1 \succeq \gamma_3} \qquad \frac{}{\Delta; \Gamma \vDash \gamma_1 \cup \gamma_2 \succeq \gamma_1}
$$

$$
\frac{}{\Delta; \Gamma \vDash \gamma_1 \cup \gamma_2 \succeq \gamma_2} \qquad \frac{\Delta; \Gamma \vDash \gamma_1 \succeq \gamma_2 \qquad \Delta; \Gamma \vDash \gamma_1 \succeq \gamma_3}{\Delta; \Gamma \vDash \gamma_1 \succeq \gamma_2 \cup \gamma_3}
$$

Figure 5. Classifier Rules

order, as lexical scopes are partially ordered by inclusion. Oishi and Kameyama [14] added a semi-lattice structure to classifiers. We use refined environment classifiers equipped with an upper semi-lattice structure and simply call them *classifiers*.

Figure 4 defines the syntax of classifiers where γ_i is a classifier variable, and $\gamma \cup \gamma'$ is the join of γ and γ'. Figure 5 specifies the partial order $\succeq$, and the join operation. Typing contexts Δ and Γ in $\Delta; \Gamma \vDash \gamma_1 \succeq \gamma_2$ will be explained later.

There are no classifier constants. For concrete examples, we use an initial classifier γ_0 to designate the empty set of future-stage variables.

4.2 Types

$$
\begin{array}{lrl}
\text{Levels} & L & ::= \cdot \mid \gamma \\
\text{Value types} & A, B & ::= \mathsf{bool} \mid A \to C \mid A \rightarrowtail C \\
 & & \mid \langle A^1 \rangle^{\gamma} \\
\text{Computation types} & C, D & ::= A \,!\, E \\
\text{Effects} & E & ::= \cdot \mid E \uplus \{op : A \twoheadrightarrow B\} \\
\text{Handler types} & F & ::= \forall \gamma.\, C \rightrightarrows D \\
\text{Level-1 types} & A^1, B^1 & ::= \mathsf{bool} \mid A^1 \to B^1
\end{array}
$$

Figure 6. Types

$$
\begin{array}{rl}
\mathsf{true} & : \mathsf{bool} \\
\mathsf{false} & : \mathsf{bool} \\
\underline{\mathsf{cbool}} & : \mathsf{bool} \to \langle \mathsf{bool} \rangle^{\gamma} \\
\underline{\mathsf{cif}} & : \langle \mathsf{bool} \rangle^{\gamma} \to \langle A^1 \rangle^{\gamma} \to \langle A^1 \rangle^{\gamma} \to \langle A^1 \rangle^{\gamma} \\
\underline{@} & : \langle A^1 \to B^1 \rangle^{\gamma} \to \langle A^1 \rangle^{\gamma} \to \langle B^1 \rangle^{\gamma}
\end{array}
$$

Figure 7. Types of Constants

Figure 6 defines types and related terms. Levels are either $\cdot$, which represents the present stage (level-0), or a future stage (level-1) represented by a classifier γ.

Value types and computation types, resp., are the types for values and computations, resp. Computations may involve an effect; hence, a computation type $A \,!\, E$ has the effect type E, which specifies the type of operations that may be called during the computation. We simply write A for $A \,!\, \cdot$. We distinguish the type for the continuation $A \rightarrowtail C$ from the ordinary function type $A \to C$ for technical reasons.

$\langle A^1 \rangle^{\gamma}$ is the type for a code value whose content has type A^1 where the code is in the lexical scope associated with γ. As is the case with values and computations, level-1 (future stage) types A^1 are more restricted than the ordinary (present-stage) types.

Finally, a handler type is assigned to a handler; consider a handler H for which **with** H **handle** M is typable under a suitable context. If the type of M is C, and the type of the whole term (the return type of the value handler in H) is D, H has the type $\forall \gamma.\, C \rightrightarrows D$. In this case, γ is not used in C and D, so the quantifier is meaningless; later, we will see concrete examples where γ has a significant role.

4.3 Typing Judgments

A typing judgment for a computation takes the form $\Delta; \Gamma \vdash^{L} M : C$, which is a five-place relation. (Those for a value V and a handler H are similar.) L, M, and C, resp. are levels, computations, and computation types, resp. The contexts Δ and Γ are well-formed sequences of an individual typing $(x : A)^{L}$, a classifier γ, and an ordering $\gamma_1 \succeq \gamma_2$, where well-formedness is defined by Figure 8.

We will also need a judgment for a name heap U, denoted by $\Delta \vdash U$, which is defined by Figure 9.

$\boxed{\vdash \Delta \ \mathsf{wf}}$

$$
\frac{}{\vdash \cdot \ \mathsf{wf}} \qquad \frac{\vdash \Delta \ \mathsf{wf} \qquad \gamma \notin \Delta}{\vdash \Delta, \gamma \ \mathsf{wf}}
$$

$$
\frac{\vdash \Delta \ \mathsf{wf} \qquad \gamma_1 \in \Delta \qquad \gamma_2 \in \Delta}{\vdash \Delta, (\gamma_1 \succeq \gamma_2) \ \mathsf{wf}}
$$

$$
\frac{\vdash \Delta \ \mathsf{wf} \qquad \gamma \in \Delta \qquad x \notin \Delta \qquad \gamma' \neq \gamma \text{ for all } (y : B)^{\gamma'} \in \Delta}{\vdash \Delta, (x : A^1)^{\gamma} \ \mathsf{wf}}
$$

$\boxed{\Delta \vdash \Gamma \ \mathsf{wf}}$

$$
\frac{}{\Delta \vdash \cdot \ \mathsf{wf}} \qquad \frac{\Delta \vdash \Gamma \ \mathsf{wf} \qquad \gamma \notin \Delta \uplus \Gamma}{\Delta \vdash \Gamma, \gamma \ \mathsf{wf}}
$$

$$
\frac{\Delta \vdash \Gamma \ \mathsf{wf} \qquad \gamma_1 \in \Delta \uplus \Gamma \qquad \gamma_2 \in \Delta \uplus \Gamma}{\Delta \vdash \Gamma, (\gamma_1 \succeq \gamma_2) \ \mathsf{wf}}
$$

$$
\frac{\Delta \vdash \Gamma \ \mathsf{wf}}{\Delta \vdash \Gamma, (x : A) \ \mathsf{wf}} \qquad \frac{\Delta \vdash \Gamma \ \mathsf{wf} \qquad \gamma \in \Delta \uplus \Gamma}{\Delta \vdash \Gamma, (x : \langle A^1 \rangle^{\gamma}) \ \mathsf{wf}}
$$

$$
\frac{\Delta \vdash \Gamma \ \mathsf{wf} \qquad \gamma \in \Delta \uplus \Gamma}{\Delta \vdash \Gamma, (x : A^1)^{\gamma} \ \mathsf{wf}}
$$

Figure 8. Well-formedness of Judgments

$\boxed{\Delta \vdash U}$

$$
\frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash U \qquad (x : A^1)^{\gamma} \in \Delta}{\Delta \vdash U, x}
$$

Figure 9. Typing Rules for Name Heaps

4.4 Design Principles

As we explained in Section 1, we made two important observations on our calculus, both of which are reflected in the typing rule for handlers (see the rule T-Hdl in Figure 12).

The "handlers as future-stage binders" principle means that, in the computation **with** H **handle** $\Box$, the handler H

acts like a binder at the future stage $\underline{\lambda}x.\,\square$. consider the handler H defined by:

$$H_{\mathsf{ins}} \coloneqq \{\mathbf{return}\ x \mapsto \mathbf{return}\ x,\\ \mathsf{ins}\ x\ k \mapsto \underline{\mathbf{let}}\ y \leftarrow \mathbf{return}\ x\ \underline{\mathbf{in}}\ \mathbf{throw}\ k\ y\}$$

and the following evaluation:

$$\begin{aligned} &\mathbf{with}\ H_{\mathsf{ins}}\ \mathbf{handle}\ \mathcal{E}[\mathbf{do}\ \mathsf{ins}\ V] && (1)\\ \rightsquigarrow\ &\underline{\mathbf{let}}\ y \leftarrow \mathbf{return}\ V\ \underline{\mathbf{in}} && (2)\\ &\mathbf{throw}\ (\kappa z.\,\mathbf{with}\ H_{\mathsf{ins}}\ \mathbf{handle}\ \mathcal{E}[\mathbf{return}\ z])\ y \end{aligned}$$

Before the evaluation (1), $\mathcal{E}[\mathbf{do}\ \mathsf{ins}\ V]$ is not in the scope of any future-stage binders. After the evaluation (2), $\mathcal{E}[\mathbf{return}\ z]$ in the result (where y will be substituted for z) is bound by a generated let-expression $\underline{\mathbf{let}}\ y \leftarrow \mathbf{return}\ V\ \underline{\mathbf{in}}\ \square$. To cope with this phenomenon soundly, the term $\mathbf{with}\ H_{\mathsf{ins}}\ \mathbf{handle}\ \square$ in (1) should be regarded as a binder.

The second principle, "handlers are universal", simply means the universality of handlers; since a handler is a first-class citizen in our calculus, it may be used at any place in a program. Hence, it must be universal in classifiers which will be instantiated to various classifiers when the handler is used by the with-handle expression.

4.5 Typing Rules

We give typing rules of our type system in groups.

Figure 10 defines those for basic primitives. T-Contd introduces the typing for a constant ${}^0K$ if its type is given as A. Similarly for the rule T-Nonstop where ${}^nK$ is a n-ary primitive operator. T-Var, T-Abs, T-App, and T-Cond are standard, except that the level of the judgments and the level of the variable must coincide. T-Cont is similar to T-Abs, but uses a different syntax to represent a continuation. T-Ret and T-Seq give types for computations in the fine-grain call-by-value calculus.

Figure 11 defines those for staging primitives. T-CAbs assigns a type to a code generator for lambda abstraction. Namely, $\underline{\lambda}x.\,M$ introduces a new lexical scope for a future-stage variable (which will be generated during the computation), hence, this rule introduces a new classifier γ_1 which has an equal or smaller lexical scope than γ, which is associated with the whole expression $\underline{\lambda}x.\,M$. Note that we write $\gamma_1 \succeq \gamma$ when γ_1's scope is smaller than γ's. To ensure the freshness of a classifier, γ_1 must obey the eigen-variable condition, which is the key innovation of Kiselyov et al.'s refined environment classifiers. T-IAbs is similar to T-CAbs, except that the present-stage variable x is replaced by a future-stage variable y. T-Code describes the meaning of a level γ; it is typed inside a bracket of the classifier γ. T-Sub0 and T-Sub1 are subsumption rules according to our intuition that classifiers are partially ordered.

$$\boxed{\Delta;\Gamma \vdash^L V : A} \quad \boxed{\Delta;\Gamma \vdash^L M : C} \quad \boxed{\Delta;\Gamma \vdash H : F}$$

$$\text{T-Const}\ \frac{}{\Delta;\Gamma \vdash^L {}^0K : A} \qquad \text{T-Var}\ \frac{(x : A)^L \in \Gamma}{\Delta;\Gamma \vdash^L x : A}$$

$$\text{T-Abs}\ \frac{\Delta;\Gamma,(x : A)^L \vdash^L M : C}{\Delta;\Gamma \vdash^L \lambda x.\,M : A \to C} \qquad \text{T-Cont}\ \frac{\Delta;\Gamma,(x : A) \vdash M : C}{\Delta;\Gamma \vdash \kappa x.\,M : A \rightarrowtail C}$$

$$\text{T-ConstApp}\ \frac{n \geq 1 \qquad {}^nK : A_1 \to \cdots \to A_{n+1} \qquad \left[\Delta;\Gamma \vdash^L V_i : A_i\right]_{1\leq i\leq n}}{\Delta;\Gamma \vdash^L {}^nK\ V_1 \cdots V_n : A_{n+1}\,!\,E} \qquad \text{T-App}\ \frac{\Delta;\Gamma \vdash^L V : A \to C \qquad \Delta;\Gamma \vdash^L W : A}{\Delta;\Gamma \vdash^L V\ W : C}$$

$$\text{T-Cond}\ \frac{\Delta;\Gamma \vdash^L V : \mathsf{bool} \qquad \Delta;\Gamma \vdash^L M : C \qquad \Delta;\Gamma \vdash^L N : C}{\Delta;\Gamma \vdash^L \mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N : C}$$

$$\text{T-Ret}\ \frac{\Delta;\Gamma \vdash^L V : A}{\Delta;\Gamma \vdash^L \mathbf{return}\ V : A\,!\,E}$$

$$\text{T-Seq}\ \frac{\Delta;\Gamma \vdash^L M : A\,!\,E \qquad \Delta;\Gamma,(x : A)^L \vdash^L N : B\,!\,E}{\Delta;\Gamma \vdash^L \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : B\,!\,E}$$

Figure 10. Type System (1)

$$\text{T-CAbs}\ \frac{\gamma \in \Delta \uplus \Gamma \qquad \gamma_1 \notin \Delta \uplus \Gamma \qquad \Delta;\Gamma,\gamma_1,(\gamma_1 \succeq \gamma),(x : \langle A^1\rangle^{\gamma_1}) \vdash M : \langle B^1\rangle^{\gamma_1}}{\Delta;\Gamma \vdash \underline{\lambda}x.\,M : \langle A^1 \to B^1\rangle^{\gamma}}$$

$$\text{T-IAbs}\ \frac{\Delta = \Delta',\gamma_1,(\gamma_1 \succeq \gamma),(y : A^1)^{\gamma_1} \qquad \Delta;\cdot \vdash M : \langle B^1\rangle^{\gamma_1}}{\Delta;\cdot \vdash \underline{\lambda}y.\,M : \langle A^1 \to B^1\rangle^{\gamma}} \qquad \text{T-Code}\ \frac{\Delta;\Gamma \vdash^{\gamma} M^1 : A^1}{\Delta;\Gamma \vdash \langle M^1\rangle : \langle A^1\rangle^{\gamma}}$$

$$\text{T-Sub0}\ \frac{\Delta;\Gamma \vdash M : \langle A^1\rangle^{\gamma_1} \qquad \Delta;\Gamma \vDash \gamma_2 \succeq \gamma_1}{\Delta;\Gamma \vdash M : \langle A^1\rangle^{\gamma_2}} \qquad \text{T-Sub1}\ \frac{\Delta;\Gamma \vdash^{\gamma_1} M : A^1 \qquad \Delta;\Gamma \vDash \gamma_2 \succeq \gamma_1}{\Delta;\Gamma \vdash^{\gamma_2} M : A^1}$$

Figure 11. Type System (2)

Figure 12 gives the most interesting typing rules, namely, those for algebraic effects and handlers in conjunction with staging primitives.[7]

[7] Due to lack of space, we omitted the typing rules for algebraic effects and handlers which do not interact with staging primitives. They can be added to our type system straightforwardly while preserving type safety.

$$\textsc{T-Throw}\quad \frac{\Delta;\Gamma \vdash V : \langle A^1\rangle^{\gamma_1} \rightarrowtail \langle B^1\rangle^{\gamma_0}\,!\,E \qquad \Delta;\Gamma \vdash W : \langle A^1\rangle^{\gamma_1 \cup \gamma_2} \qquad \Delta;\Gamma \vDash \gamma_2 \succeq \gamma_0}{\Delta;\Gamma \vdash \mathbf{throw}\ V\ W : \langle B^1\rangle^{\gamma_2}\,!\,E}$$

$$\textsc{T-Do}\quad \frac{E = \{op : A_{\mathrm{op}} \twoheadrightarrow \langle B^1_{\mathrm{op}}\rangle^{\gamma}\} \uplus E' \qquad \Delta;\Gamma \vdash V : A_{\mathrm{op}}}{\Delta;\Gamma \vdash \mathbf{do}\ op\ V : \langle B^1_{\mathrm{op}}\rangle^{\gamma}\,!\,E}$$

$$\textsc{T-Hdl}\quad \frac{\gamma \in \Delta \uplus \Gamma \qquad \gamma_1 \notin \Delta \uplus \Gamma \qquad \Delta;\Gamma \vdash H : \forall\gamma'.\,\langle A^1\rangle^{\gamma'}\,!\,E \rightrightarrows \langle B^1\rangle^{\gamma'}\,!\,E' \qquad \Delta;\Gamma,\gamma_1,(\gamma_1 \succeq \gamma) \vdash M : \langle A^1\rangle^{\gamma_1}\,!\,E[\gamma_1/\gamma']}{\Delta;\Gamma \vdash \mathbf{with}\ H\ \mathbf{handle}\ M : \langle B^1\rangle^{\gamma}\,!\,E'[\gamma/\gamma']}$$

$$\textsc{T-Hdlr}\quad \frac{H = \{\mathbf{return}\ x \mapsto M\} \uplus \{op_i\ x\ k \mapsto N_i\}_i \qquad E = \{op_i : A_{\mathrm{op}_i} \twoheadrightarrow \langle B^1_{\mathrm{op}_i}\rangle^{\gamma}\}_i \uplus E' \qquad \Gamma'_i = \Gamma,\gamma,(x : A_{\mathrm{op}_i}),(k : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma} \rightarrowtail \langle B^1\rangle^{\gamma}\,!\,E') \qquad \gamma \notin \Delta \uplus \Gamma \qquad \Delta;\Gamma,\gamma,(x : \langle A^1\rangle^{\gamma}) \vdash M : \langle B^1\rangle^{\gamma}\,!\,E' \qquad \left[\Delta;\Gamma'_i \vdash N_i : \langle B^1\rangle^{\gamma}\,!\,E'\right]_i}{\Delta;\Gamma \vdash H : \forall\gamma.\,\langle A^1\rangle^{\gamma}\,!\,E \rightrightarrows \langle B^1\rangle^{\gamma}\,!\,E'}$$

Figure 12. Type System (3)

T-Hdlr is a specialized typing rule for staged programs with handlers. To type a handler H, we need to type its value handler (the judgment $\Delta;\Gamma,\gamma,(x : \langle A^1\rangle^{\gamma}) \vdash M : \langle B^1\rangle^{\gamma}\,!\,E'$) and all operations op_i ($\cdots \vdash N_i : \langle B^1\rangle^{\gamma}\,!\,E'$). Assuming that related types of these operations are code types (for instance, $\langle B^1\rangle^{\gamma}$), the two assumptions in T-Hdlr are not difficult to understand. The only important feature is that the classifier γ must be fresh ($\gamma \notin \Delta \uplus \Gamma$), in other words, it should follow the eigen-variable condition of this typing rule. The universal quantification for γ in the handler type (the conclusion of this rule) means that this handler may be used in any lexical scope.

Its counterpart is T-Hdl. According to our second principle "handlers are universal", we introduce a new lexical scope by creating a new classifier γ_1. T-Do assigns a type to an operation call.

T-Throw is interesting; it intuitively means that a type of a continuation $\gamma_1 \rightarrowtail \gamma_0$ can be turned into $\gamma_1 \cup \gamma_2 \rightarrowtail \gamma_0 \cup \gamma_2$ (we are looking at only classifiers for simplicity). To see it concretely, consider the following example:

$$\underline{\mathbf{let}}\ z \leftarrow N_1\ \underline{\mathbf{in}}^{\gamma_z}$$
$$\mathbf{with}\ H_{\mathrm{op}}\ \mathbf{handle}\ \underline{\mathbf{let}}\ w \leftarrow N_2\ \underline{\mathbf{in}}^{\gamma_w}\ \mathbf{do}\ \mathrm{op}\ W$$

where

$$H_{\mathrm{op}} = \{\mathbf{return}\ x \mapsto \mathbf{return}\ x,\ \mathrm{op}\ x\ k \mapsto \underline{\mathbf{let}}\ y \leftarrow M\ \underline{\mathbf{in}}^{\gamma_y}\ C[\mathbf{throw}\ k\ V]\}.$$

Note that we added notations for a classifier introduced at each future-stage let-binding. As we explained before, a classifier can be considered as a set of available future-stage variables. From that perspective, we can consider that γ_z denotes $\{z\}$, γ_w denotes $\{z, w\}$, and γ_y denotes $\{z, y\}$. Here the continuation k has a type $\gamma_w \rightarrowtail \gamma_z$. Since the future-stage variable y will be bound outside of w's binder, it is reasonable to allow V to contain y. However, since we have no subtyping relation between γ_y and γ_w, some special treatment is necessary to achieve this. That is the reason why we regard $k : \gamma_w \rightarrowtail \gamma_z$ as $k : \gamma_w \cup \gamma_y \rightarrowtail \gamma_y$ under $\gamma_y \succeq \gamma_z$.

The intuition of T-Throw described above means that a continuation is polymorphic over classifiers, which is expressed more naturally by the following type:

$$\forall\gamma_2.\ (\gamma_1 \cup \gamma_2 \rightarrowtail \gamma_0 \cup \gamma_2)$$

We could have given such a type to a continuation, but we did not do so, since we want to keep the monomorphic nature of our type system. Extending our calculus to a type-and-classifier-polymorphic calculus is left for future work.

5 Examples of Typing Derivation

In this section, we show some examples of typing derivations.

5.1 Multiple Let-insertions

As an example of code generation with a complex combination of multiple effects, we demonstrate typing of the following program.

$$\mathbf{with}\ H_1\ \mathbf{handle}\ \mathbf{with}\ H_2\ \mathbf{handle}\ \mathbf{let}\ x_1 \leftarrow M\ \mathbf{in}\ \mathbf{let}\ x_2 \leftarrow \mathbf{do}\ \mathrm{ins}_1\ x_1\ \mathbf{in}\ \mathbf{do}\ \mathrm{ins}_2\ x_2,$$

where

$$M = \underline{\mathrm{cbool}}\ \mathrm{true},$$
$$E_i = \{\mathrm{ins}_i : \langle \mathrm{b}\rangle^{\gamma_i} \rightarrow \langle \mathrm{b}\rangle^{\gamma_i}\},\ \text{and}$$
$$H_i = \{\mathrm{ins}_i\ x\ k \mapsto \mathbf{let}\ f \leftarrow \underline{\lambda} y.\ \mathbf{throw}\ k\ y\ \mathbf{in}\ f\ \underline{@}\ x,\ \mathbf{return}\ x \mapsto \mathbf{return}\ x\}.$$

This program is eventually evaluated to the following well-scoped code:

$$\mathbf{return}\ \langle(\lambda y_1.\ (\lambda y_2.\ \mathbf{return}\ y_2)\ y_1)\ \mathrm{true}\rangle$$

hence it is expected to be well-typed, and indeed it is. We show its type derivation in Fig. 13. Note that we abbreviate bool as b and omit the same part of a type environment as its conclusion in the figure.

$$\dfrac{\dfrac{}{\cdots \vdash k : \langle \mathsf{b}\rangle^{\gamma'} \rightarrowtail \langle \mathsf{b}\rangle^{\gamma'}}\ \text{T-Var} \quad \dfrac{\dfrac{}{\cdots \vdash y : \langle \mathsf{b}\rangle^{\gamma_1'}}\ \text{T-Var} \quad \dfrac{}{\cdots \vdash \gamma' \cup \gamma_1' \succeq \gamma_1'}}{\cdots \vdash y : \langle \mathsf{b}\rangle^{\gamma' \cup \gamma_1'}}\ \text{T-Sub1} \quad \dfrac{}{\cdots \vdash \gamma_1' \succeq \gamma'}}{\cdots, \gamma_1', (\gamma_1' \succeq \gamma'), (y : \langle \mathsf{b}\rangle^{\gamma_1'}) \vdash \mathbf{throw}\ k\ y : \langle \mathsf{b}\rangle^{\gamma_1'}\ !\ E_1}\ \text{T-Throw}$$
$$\cdots \vdash \underline{\lambda} y.\, \mathbf{throw}\ k\ y : \langle \mathsf{b} \to \mathsf{b}\rangle^{\gamma'}\ !\ E_1\ (\mathcal{D}_4)\ \text{T-CAbs}$$

$$\dfrac{\dfrac{}{\cdots \vdash x : \langle \mathsf{b}\rangle^{\gamma'}}\ \text{T-Var}}{\gamma_0; \gamma_1, (\gamma_1 \succeq \gamma_0), \gamma', (x : \langle \mathsf{b}\rangle^{\gamma'}) \vdash \mathbf{return}\ x : \langle \mathsf{b}\rangle^{\gamma'}\ !\ E_1\ (\mathcal{D}_3)}\ \text{T-Ret}$$

$$\dfrac{\mathcal{D}_3 \qquad \dfrac{\mathcal{D}_4 \qquad \dfrac{\dfrac{}{\cdots \vdash f : \langle \mathsf{b} \to \mathsf{b}\rangle^{\gamma'}}\ \text{T-Var} \quad \dfrac{}{\cdots \vdash x : \langle \mathsf{b}\rangle^{\gamma'}}\ \text{T-Var}}{\cdots, (f : \langle \mathsf{b} \to \mathsf{b}\rangle^{\gamma'}) \vdash f\ \underline{@}\ x : \langle \mathsf{b}\rangle^{\gamma'}\ !\ E_1}\ \text{T-ConstApp}}{\cdots, \gamma', (x : \langle \mathsf{b}\rangle^{\gamma'}), (k : \langle \mathsf{b}\rangle^{\gamma'} \rightarrowtail \langle \mathsf{b}\rangle^{\gamma'}) \vdash \mathbf{let}\ f \leftarrow \underline{\lambda} y.\, \mathbf{throw}\ k\ y\ \mathbf{in}\ f\ \underline{@}\ x : \langle \mathsf{b}\rangle^{\gamma'}\ !\ E_1}\ \text{T-Seq}}{\gamma_0; \gamma_1, (\gamma_1 \succeq \gamma_0) \vdash H_2 : \forall \gamma'.\, \langle \mathsf{b}\rangle^{\gamma'}\ !\ (E_1 \uplus \{\mathsf{ins}_2 : \langle \mathsf{b}\rangle^{\gamma'} \to \langle \mathsf{b}\rangle^{\gamma'}\}) \Rightarrow \langle \mathsf{b}\rangle^{\gamma'}\ !\ E_1\ (\mathcal{D}_2)}\ \text{T-Hdlr}$$

($\mathcal{D}_1$ is similar to $\mathcal{D}_2$ and is therefore omitted)

$$\dfrac{\mathcal{D}_1 \qquad \dfrac{\mathcal{D}_2 \qquad \dfrac{\vdots \qquad \dfrac{\dfrac{}{\cdots \vdash x_1 : \langle \mathsf{b}\rangle^{\gamma_1}}\ \text{T-Var}}{\cdots \vdash \mathbf{do}\ \mathsf{ins}_1\ x_1 : \langle \mathsf{b}\rangle^{\gamma_1}\ !\ (E_1 \uplus E_2)}\ \text{T-Do} \quad \dfrac{\dfrac{\dfrac{}{\cdots \vdash x_2 : \langle \mathsf{b}\rangle^{\gamma_1}}\ \text{T-Var} \quad \dfrac{}{\cdots \vdash \gamma_2 \succeq \gamma_1}}{\cdots \vdash x_2 : \langle \mathsf{b}\rangle^{\gamma_2}}\ \text{T-Sub0}}{\cdots, (x_2 : \langle \mathsf{b}\rangle^{\gamma_1}) \vdash \mathbf{do}\ \mathsf{ins}_2\ x_2 : \langle \mathsf{b}\rangle^{\gamma_2}\ !\ (E_1 \uplus E_2)}\ \text{T-Do}}{\cdots, (x_1 : \langle \mathsf{b}\rangle^{\gamma_1}) \vdash \mathbf{let}\ x_2 \leftarrow \mathbf{do}\ \mathsf{ins}_1\ x_1\ \mathbf{in}\ \mathbf{do}\ \mathsf{ins}_2\ x_2 : \langle \mathsf{b}\rangle^{\gamma_2}\ !\ (E_1 \uplus E_2)}\ \text{T-Seq}}{\gamma_0; \gamma_1, (\gamma_1 \succeq \gamma_0), \gamma_2, (\gamma_2 \succeq \gamma_1) \vdash \mathbf{let}\ x_1 \leftarrow M\ \mathbf{in}\ \mathbf{let}\ x_2 \leftarrow \mathbf{do}\ \mathsf{ins}_1\ x_1\ \mathbf{in}\ \mathbf{do}\ \mathsf{ins}_2\ x_2 : \langle \mathsf{b}\rangle^{\gamma_2}\ !\ (E_1 \uplus E_2)}\ \text{T-Seq}}{\gamma_0; \gamma_1, (\gamma_1 \succeq \gamma_0) \vdash \mathbf{with}\ H_2\ \mathbf{handle}\ \mathbf{let}\ x_1 \leftarrow M\ \mathbf{in}\ \mathbf{let}\ x_2 \leftarrow \mathbf{do}\ \mathsf{ins}_1\ x_1\ \mathbf{in}\ \mathbf{do}\ \mathsf{ins}_2\ x_2 : \langle \mathsf{b}\rangle^{\gamma_1}\ !\ E_1}\ \text{T-Hdl}}{\gamma_0; \cdot \vdash \mathbf{with}\ H_1\ \mathbf{handle}\ \mathbf{with}\ H_2\ \mathbf{handle}\ \mathbf{let}\ x_1 \leftarrow M\ \mathbf{in}\ \mathbf{let}\ x_2 \leftarrow \mathbf{do}\ \mathsf{ins}_1\ x_1\ \mathbf{in}\ \mathbf{do}\ \mathsf{ins}_2\ x_2 : \langle \mathsf{b}\rangle^{\gamma_0}}\ \text{T-Hdl}$$

Figure 13. Example of Typing Derivation for Let-insertion

6 Properties of the Calculus

In this section, we show the key steps of the proof of type soundness for our calculus. Type soundness is one of the most important properties for typed calculi. In staged calculi, type soundness has stronger implication than unstaged calculi does, since it ensures the absence of run-time type errors not only for the programs in the present stage, but that for all the generated programs. The latter subsumes the absence of scope extrusion.

Definition 6.1 (Normal Form of Computation)**.** We say a computation M is a normal form with respect to E if M is either of the form $M = \mathbf{return}\ V$, or $M = \mathcal{E}[\mathbf{do}\ op\ W]$ where $op \in E$ and $op \notin \mathsf{bl}(\mathcal{E})$.

We write $\overline{\Gamma}$ for an environment with a special condition: it contains only either a classifier or their constraint ($\gamma_2 \succeq \gamma_1$).

Theorem 6.2 (Progress)**.** *If $\Delta; \overline{\Gamma} \vdash M : A\,!\,E$, then M is either a normal form with respect to E, or there exists N such that $U; M \rightsquigarrow U'; N$.*

It can be proved straightforwardly by induction on a derivation with the standard canonical-forms lemma.

To prove the subject reduction, we first state a key lemma that allows us to replace a newly introduced classifier with an arbitrary pre-defined classifier in an environment.

Lemma 6.3 (Classifier Specialization)**.**

1. *If $\Delta, \gamma_2, \Delta'; \Gamma \vdash^L M : C$, then $\Delta, \Delta'[\gamma_2/\gamma_1]; \Gamma[\gamma_2/\gamma_1] \vdash^{L[\gamma_2/\gamma_1]} M : C[\gamma_2/\gamma_1]$ for all $\gamma_1 \in \Delta$.*
2. *If $\Delta; \Gamma, \gamma_2, \Gamma' \vdash M : C$, then $\Delta; \Gamma, \Gamma'[\gamma_2/\gamma_1] \vdash M : C[\gamma_2/\gamma_1]$ for all $\gamma_1 \in \Delta \uplus \Gamma$.*

We also have some standard substitution lemmas proved straightforwardly.

Lemma 6.4 (Substitution)**.** *Suppose $\Delta; \Gamma \vdash W : B$.*

1. *If $\Delta; \Gamma, (x : B) \vdash V : A$, then $\Delta; \Gamma \vdash V[W/x] : A$.*
2. *If $\Delta; \Gamma, (x : B) \vdash M : C$, then $\Delta; \Gamma \vdash M[W/x] : C$.*
3. *If $\Delta; \Gamma, (x : B) \vdash H : F$, then $\Delta; \Gamma \vdash H[W/x] : F$.*

Lemma 6.5 (Substitution to Evaluation Context)**.** *If $\Delta; \Gamma \vdash \mathcal{E}[M] : C$, $\Delta; \Gamma \vdash M : D$, and $\Delta, \Delta'; \Gamma, \Gamma' \vdash N : D$, then $\Delta, \Delta'; \Gamma, \Gamma' \vdash \mathcal{E}[N] : C$*

One of the main results is the subject reduction. We state it as follows.

Theorem 6.6 (Subject Reduction). *If $\Delta; \overline{\Gamma} \vdash M : C$, $\Delta \vdash U$, and $U; M \rightsquigarrow U'; N$, then $\Delta, \Delta'; \overline{\Gamma} \vdash N : C$ and $\Delta, \Delta' \vdash U'$ for some Δ'.*

Proof. By induction on a typing derivation. Instead of putting the full proof, we will focus on one specific reduction as a non-trivial case:

$$U; \mathbf{with}\ H\ \mathbf{handle}\ \mathcal{E}[\mathbf{do}\ op_i\ V] \rightsquigarrow U; M_{\mathrm{op}_i}[V/x, W/k]$$

by E-Do, where $W = \kappa y.\, \mathbf{with}\ H\ \mathbf{handle}\ \mathcal{E}[\mathbf{return}\ y]$ and $op_i \notin \mathrm{bl}(\mathcal{E})$. Inversion of T-Hdlr and Lemma 6.3 gives $\Delta; \overline{\Gamma}, (x : A_{\mathrm{op}_i}[\gamma/\gamma']), (k : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma} \rightarrowtail \langle B^1\rangle^{\gamma} \,!\, E'[\gamma/\gamma']) \vdash M_{\mathrm{op}_i} : \langle B^1\rangle^{\gamma} \,!\, E'[\gamma/\gamma']$ (1). We have the following subderivation where $(op_i : A_{\mathrm{op}_i}[\gamma_1/\gamma'] \twoheadrightarrow \langle B^1_{\mathrm{op}_i}\rangle^{\gamma_1}) \in E''$.

$$\frac{\Delta; \overline{\Gamma}, \gamma_1, (\gamma_1 \succeq \gamma) \vdash V : A_{\mathrm{op}_i}[\gamma_1/\gamma']}{\Delta; \overline{\Gamma}, \gamma_1, (\gamma_1 \succeq \gamma) \vdash \mathbf{do}\ op_i\ V : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma_1} \,!\, E''}\ \text{T-Do}$$

Applying Lemma 6.3 to the premise above gives $\Delta; \overline{\Gamma} \vdash V : A_{\mathrm{op}_i}[\gamma/\gamma']$ (3). Let $\Gamma = \overline{\Gamma}, (y : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma})$. We construct a typing derivation for W as follows.

$$\dfrac{\dfrac{\Delta; \Gamma \vdash H : \forall \gamma'.\, \langle A^1\rangle^{\gamma'} \,!\, E \rightrightarrows \langle B^1\rangle^{\gamma'} \,!\, E' \text{ (a)} \quad \Delta; \Gamma, \gamma_1, (\gamma_1 \succeq \gamma) \vdash \mathcal{E}[\mathbf{return}\ y] : \langle A^1\rangle^{\gamma_1} \,!\, E[\gamma_1/\gamma'] \text{ (b)}}{\Delta; \Gamma \vdash \mathbf{with}\ H\ \mathbf{handle}\ \mathcal{E}[\mathbf{return}\ y] : \langle B^1\rangle^{\gamma} \,!\, E'[\gamma/\gamma']}}{\Delta; \overline{\Gamma} \vdash W : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma} \rightarrowtail \langle B^1\rangle^{\gamma} \,!\, E'[\gamma/\gamma']\ (2)}$$

(a) can be obtained from the original typing of H by weakening. Let $\Gamma' = \overline{\Gamma}, \gamma_1, (\gamma_1 \succeq \gamma), (y : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma})$. For deriving (b), we consider a typing of $\mathbf{return}\ y$ as follows.

$$\dfrac{\dfrac{\dfrac{}{\Delta; \Gamma' \vdash y : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma}}\ \text{T-Var} \qquad \dfrac{}{\Delta; \Gamma' \vdash \gamma_1 \succeq \gamma}}{\Delta; \Gamma' \vdash y : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma_1}}\ \text{T-Sub0}}{\Delta; \Gamma' \vdash \mathbf{return}\ y : \langle B^1_{\mathrm{op}_i}\rangle^{\gamma_1} \,!\, E''}\ \text{T-Ret}$$

We obtain $\Delta; \Gamma' \vdash \mathcal{E}[\mathbf{return}\ y] : \langle A^1\rangle^{\gamma_1} \,!\, E[\gamma_1/\gamma']$ by substitution, thus we obtain (b) by harmless exchange in the environment. Applying substitution to (1), (2), and (3) gives a desired result $\Delta; \overline{\Gamma'} \vdash M_{\mathrm{op}_i}[V/x, W/k] : \langle B^1\rangle^{\gamma} \,!\, E'[\gamma/\gamma']$. □

As a consequence of the subject reduction and a brief discussion, we conclude the following important property that a generated code never causes scope extrusion.

Corollary 6.7 (Absence of Scope Extrusion). *If $\gamma_0; \cdot \vdash M : \langle A^1\rangle^{\gamma_0}$ and $\cdot; M \rightsquigarrow^* U; \mathbf{return}\ V$, then $V = \langle N^1\rangle$ and $\gamma_0; \cdot \vdash^{\gamma_0} N^1 : A^1$ for some N^1.*

7 Related Work

One of the oldest studies on safe imperative multi-stage programming was given by Calcagno et al. [2]. They obtained safety for the language by restricting only closed codes can be manipulated (stored in mutable cells, etc.), which is a severe restriction on the viewpoint of the practical application of staged computation.

Westbrook et al. [20] introduced a multi-stage extension of Java-like languages and proved its type safety. They proposed the weak separability property, which essentially prohibits any open code to be stored in mutable cells,

Kameyama et al. [7] designed a staged language with control operators *shift* and *reset*, which are capable of representing various monadic effects, including states, exceptions, and non-determinism. To achieve type soundness, they put the restriction that any control effects may not go across the future-stage binders and showed that 'assertion-insertion' and a few other useful idioms can be expressed in their language. However, more advanced techniques such as let-insertion are not expressible under their restriction.

These classic works designed relatively simple type systems and, therefore cannot express the lexical scope of future-stage binders of variables, which seems to be essential to gain more expressiveness.

The seminal work by Taha and Nielsen [19] introduced the notion of environment classifier, which is a proxy of a typing context, thus enabling a type can mention a typing context. They presented a safe multi-stage language that can handle open codes, and has the *run* primitive. Although their type system is a basis of the first version of MetaOCaml, type safety is guaranteed for purely functional sublanguage only; we can use effectful computations in MetaOCaml, but then there is no guarantee of type safety.

Kiselyov et al. [10] proposed refined environment classifiers (REC), which gives a more refined representation of Taha and Nielsen's environment classifier. The salient discovery in their design is that the well-known eigen-variable condition is sufficient to guarantee well-scopedness of future-stage binders. They designed a two-stage language with REC and ML-style reference cells and proved type soundness.

Oishi and Kameyama [14] imported REC to the two-stage typed language with the control operators *shift0* and *reset0*. Thanks to REC and a sophisticated restriction to avoid type-unsoundness, their language allows one to express nested let-insertion for open codes. Their work and our work in this paper share the motivation, and we have already compared their work and our work in Section 2.

Realistic multi-stage extensions of mainstream programming languages often ignore the type-unsoundness problem in the presence of computational effects. Template Haskell, MetaOCaml, and Scala, for instance, do not give a static guarantee of well-typedness and well-scopedness of generated codes if a programmer uses computational effects.

An important exception is the genlet primitive in BER MetaOCaml [9], which performs (nested) let-insertion. The salient feature of genlet is that a program that uses genlet does not have to specify the destination, and the language system determines, on the fly, where to insert the expression

into the code. More precisely, the destination is chosen as the top-most point where no scope extrusion occurs, hence, (provided the scope-checking is correct) scope extrusion will never occur. This primitive is particularly useful, but its semantics is not easy to understand nor formally documented. This is problematic, since genlet can jump over arbitrary expressions such as conditionals and lambda-abstraction, but a programmer cannot know the destination of let-insertion statically. Compared with their work, our work covers not only let-insertions but arbitrary computational effects expressible by algebraic effects and handlers, which distinguishes our work from their work. A possible demerit of our language is that a programmer needs to specify the destination of let-insertion, but we think this is not a big problem in practice since a programmer of code generators surely knows the place where the code memoizes open codes. Combining algebraic effects and handlers with genlet in a multi-stage language would be an interesting future research topic.

While we extensively used environment classifiers as a proxy of typing contexts, there is another line of work on formalizing type-sound staged calculi based on explicit representation of typing contexts in types.

Nanevski and Pfenning [13] proposed Contextual Modal Type Theory, which gives a logical foundation for staged computation. While they gave a finer type system than those based on environment classifiers, their calculus is more verbose and complicated than those based on the latter.

Rhiger [18] gave a similar, but different, calculus for staging based on explicit representation of typing contexts in types, which covered mutable state. While the goal of his paper is similar to ours, it is unclear whether his type system can be used to guarantee type safety in the presence of control operators including algebraic effects and handlers.

Formal calculi based on the contextual representation of code types have been studied by several groups of researchers [6, 15], both of which have the feature of analyzing (pattern matching) generated code. We have taken the purely generative approach where code can be generated, but generated code cannot be decomposed.

Recent work by Kovacs [11] gave a two-level type theory as the foundational system for staged compilation with dependent type theory.

It is interesting to study an extension of our calculus where code analysis or dependent types is allowed while preserving type safety.

8 Conclusion

This paper presented a two-stage language for effectful staged computation and its type system, and proved type soundness, leading to the static guarantee of well-scopedness and well-typedness for any code generated by well-typed code generators. As far as we know, this is the first such result in the presence of algebraic effects and handlers, which are quickly becoming a standard construct to express various computational effects uniformly.

Compared with previous studies, our calculus allows nested named[8] operations, which allows one to express arbitrarily nested let-insertion precisely and concisely. We believe this is a great benefit, as the code-duplication problem is one of the central issues for program generation to be a practically applicable technique.

Our type system is based on a refined environment classifier, which is a proxy of a lexical scope of a code-level variable, but two new ideas were needed. The first key idea is the "handlers as binders" principle, which means that a with-handle expression introduces a new lexical scope for the future-stage variable, although there are no future-stage variables being involved in handlers. The second key idea is the "handlers are universal" principle, which means that the handler types should be polymorphic over scopes. By using these two principles, we designed a type system that enjoys the type soundness property.

We believe that this paper demonstrates the usefulness of classifiers, in connection with the eigen-variable condition in natural-deduction style logic. Even though it is a first step, we believe that our calculus based on classifiers can be a basis for studying other interesting program generators which need different patterns of computational effects.

Future work: this paper gave a small core calculus that has two stages and algebraic effects and handlers. Extending our results to a full-blown programming language and implementing it are interesting next steps. To make our calculus practical in OCaml or MetaOCaml, we should consider a type inference algorithm. We expect that the universal quantification of handler types can be reduced to ML-style let-polymorphism, and that the principal-type property holds for our calculus under some suitable restriction. For the foundational aspect, there is no reason to stop at two stages. It is an interesting next step to design a truly multi-stage calculus with algebraic effects and handlers, for which type soundness is statically guaranteed.

Acknowledgments

We would like to thank anonymous reviewers of GPCE'24 for their careful reading and critical comments. The authors are supported in part by JSPS Grant-in-Aid for Scientific Research (B) 22H03563.

References

[1] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. https://doi.org/10.1145/3527320

[8] In some literature, delimited-control operators that are tagged by names are called *multi-prompt* control operators.

[2] Cristiano Calcagno and Eugenio Moggi. 2000. Multi-Stage Imperative Languages: A Conservative Extension Result. In *Semantics, Applications, and Implementation of Program Generation, International Workshop SAIG 2000, Montreal, Canada, September 20, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1924)*, Walid Taha (Ed.). Springer, 92–107. https://doi.org/10.1007/3-540-45350-4_9

[3] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/91556.91622

[4] Rowan Davies. 1996. A Temporal-Logic Approach to Binding-Time Analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 184–195. https://doi.org/10.1109/LICS.1996.561317

[5] Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *Conference Record of The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 258–270. https://doi.org/10.1145/237721.237788

[6] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. https://doi.org/10.1145/3498700

[7] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the Stage: Staging with Delimited Control. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Savannah, GA, USA) *(PEPM '09)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1480945.1480962

[8] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *ACM SIGPLAN International Conference on Functional Programming*. 145–158.

[9] Oleg Kiselyov. 2024. MetaOCaml: Ten Years Later - System Description. In *Functional and Logic Programming - 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15-17, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14659)*, Jeremy Gibbons and Dale Miller (Eds.). Springer, 219–236. https://doi.org/10.1007/978-981-97-2300-3_12

[10] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*. 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

[11] András Kovács. 2022. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. https://doi.org/10.1145/3547641

[12] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

[13] Aleksandar Nanevski and Frank Pfenning. 2005. Staged computation with names and necessity. *J. Funct. Program.* 15, 5 (2005), 893–939. https://doi.org/10.1017/S095679680500568X

[14] Junpei Oishi and Yukiyoshi Kameyama. 2017. Staging with control: type-safe multi-stage programming with control operators. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 29–40. https://doi.org/10.1145/3136040.3136049

[15] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* 2, POPL (2018), 13:1–13:33. https://doi.org/10.1145/3158101

[16] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

[17] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[18] Morten Rhiger. 2012. Staged Computation with Staged Lexical Scope. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 559–578. https://doi.org/10.1007/978-3-642-28869-2_28

[19] Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604131.604134

[20] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java Multi-stage Programming Using Weak Separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 400–411. https://doi.org/10.1145/1806596.1806642

Received 2024-06-18; accepted 2024-08-15