Staging with Control: Type-Safe Multi-stage Programming with Control Operators

Junpei Oishi[*]
Department1 of Computer Science
University of Tsukuba
Tsukuba 305-8573, Japan
oishi@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
Department of Computer Science
University of Tsukuba
Tsukuba 305-8573, Japan
kameyama@acm.org

Abstract

Staging allows a programmer to write domain-specific, custom code generators. Ideally, a programming language for staging provides all necessary features for staging, and at the same time, gives static guarantee for the safety properties of generated code including well typedness and well scopedness. We address this classic problem for the language with control operators, which allow code optimizations in a modular and compact way. Specifically, we design a staged programming language with the expressive control operators shift0 and reset0, which let us express, for instance, multi-layer let-insertion, while keeping the static guarantee of well typedness and well scopedness. For this purpose, we extend our earlier work on refined environment classifiers which were introduced for the staging language with state. We show that our language is expressive enough to write interesting code generation techniques, and that it enjoys type soundness. We also mention a type inference algorithm for our language under reasonable restriction.

CCS Concepts • **Theory of computation** → **Type theory**; • **Software and its engineering** → *Functional languages*;

Keywords code generation, language design, program transformation, generative programming, staging, type system

ACM Reference format:
Junpei Oishi and Yukiyoshi Kameyama. 2017. Staging with Control: Type-Safe Multi-stage Programming with Control Operators. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Vancouver, Canada, October 23–24, 2017 (GPCE'17),* 12 pages.
https://doi.org/10.1145/3136040.3136049

1 Introduction

Program generation is one of the leading approaches to achieve high-level abstraction and high performance in a single framework. Multi-stage Programming Languages such as MetaML [18] are statically typed languages with lexical binding for program generation, with the static safety assurance that all generated code is well scoped and well typed. Today we find several successful full-blown languages in this approach such as MetaOCaml [9, 17], where a programmer easily convert an unstaged code to a code generator and the type checker does a good job of ensuring the safety of all code possibly generated by the generator. Scala Lightweight Modular Staging (LMS) [16] is another successful full-blown language for staging in the same spirit.

Despite the success of these languages in many practical domains and applications, there still remains an issue of static safety guarantee when the language has some sort of computational effects, which are necessary to express many optimizations at the generation time. We address this classic problem for a simple calculus with lexical binding, in the presence of the very powerful control operators shift0 and reset0, and give a solution to the problem.

Taha and Nielsen introduced environment classifiers (classifiers, for short) which represent a typing context (such as $x : \mathtt{int}, y : \mathtt{bool}$) *abstractly* [17]. Compared with the pioneering type system $\lambda\Box$ [4] and $\lambda\bigcirc$ [3], the type system with classifiers successfully achieved open code manipulation while allowing the 'run' primitive in a type-safe manner. An important problem in their approach is that the underlying theory guarantees type safety only for purely functional subcalculus; no computational effects are allowed in their language. In the MetaML-style multi-stage programming, computational effects are indispensable to express, for instance, exchanging the order of loops, memoizing once-generated code, and let-insertion. It is a challenging research topic to extend the target calculus to cover necessary computational effects while keeping the static guarantee of safety.

Let us see a few examples where computational effects are needed at the code-generation time.

The first example is code motion beyond binders. Suppose we have the following generator (we use OCaml-like syntax when we write code):

```
for 0 n (fun i →
  Array.set a i <some_computation>
)
```

where the size of the array a is n+1, **for** is a constant to generate the for-expression if supplied with three arguments as the initial and final values of the for-loop, and the body of the

[*]Currently with Yahoo Japan Corporation.

GPCE'17, October 23–24, 2017, Vancouver, Canada
2017. ACM ISBN 978-1-4503-5524-7/17/10...$15.00
https://doi.org/10.1145/3136040.3136049

loop abstracted by the loop variable (i in this example). The **for** primitive generates (code for) a for-loop (with possibly unrolling the loop). The constant **Array.set** generates (code for) an **Array.set** expression, which assigns a value to an array at the specified index. These underlined primitives for code generation are called *code combinators* in this paper. The bracket expression <...> represents a code value.

The above code evaluates to the following code value:

```
<for i = 0 to n do
  a.(i) <- some_computation
done>
```

where a.(i) is the i-th element of the array a.

Suppose some_computation is code for a time-consuming computation, then we may want to generate the following code:

```
<let x = some_computation in
  for i = 0 to n do
    a.(i) <- x
  done>
```

Thus some_computation moves beyond the binder for the variable i. This is a very simple example of code motion at the code generation time, and we need some kind of computational effects:[1] we can use either mutable variables to store code, or control operators to move code fragments explicitly.

The code motion beyond binders is subtle and sometimes dangerous, as the code may contain free variables. In the above code snippet, if `some_computation` contains the variable i freely, moving it above the for-loop would generate code with an unbound variable (the *scope extrusion* problem). Things would become more complicated if we combine code motion with other optimizations such as loop exchange or another code motion.

Next, we consider the following example with nested for-loops.

```
for 0 n (fun i →
  for 0 m (fun j →
    let idx = i * m + j in
      Array.set a idx <e1>;
      Array.set b idx <e2>
  )
)
```

Suppose the variable i occurs freely in e2 but not in e1 while j does not occur freely in e1 nor e2. Then, the code we wish to generate is:

```
<let x1 = e1 in
 for i = 0 to n do
   let x2 = e2 in
     for j = 0 to m do
       let idx = i*m+j in
         a.(idx) <- x1;
         b.(idx) <- x2
     done
   done>
```

The result shows that the code <e1> moves to the topmost position, while the code <e2> moves to the intermediate place, and we cannot do it the other way around, since i occurs in e2 freely. In summary, the destination of code motion should depend on the expression to be moved.

Our third example is code sharing by let-insertion. We borrow the Gibonacci example from our earlier paper [6].

```
let rec gib n x y  =
  if n = 0 then x
  else if n = 1 then y
  else (gib (n-2) x y) + (gib (n-1) x y)
```

The Gibonacci function is a generalization of the Fibonacci function. Suppose we want to generate code for a fixed n, while leaving x and y dynamic. As in the case for the Fibonacci function, a naive code generator for Gibonacci would produce an inefficient code as follows (we assume n = 5):

```
<fun x→ fun y→ (y+(x+y))+((x+y)+(y+(x+y)))>
```

A more efficient code should be:

```
< fun x → fun y →
    let g2 = x + y in
    let g3 = y + g2 in
    let g4 = g2 + g3 in
    let g5 = g3 + g4 in
      g5 >
```

where the code contains many let expressions for sharing code. The point here is to share the results of computation between independent function calls (gib (n−2) x y) and (gib (n−1) x y), which needs the mechanism of memoization. The literature [8] shows that this particular program can be generated without moving open code across binders, but in general, we need to do it.

All the above examples need some computational effects in program generators, and each of them has been shown to be expressible in some existing work: Kameyama et al. [8] use the control operator shift and reset to cover the first and the third examples above, and Kiselyov et al. [12] use global reference cell to mimic the first and the second examples. However, to our knowledge, no existing work in the literature can solve all the problems above in a single framework. Combining global mutable cells with control operators in a multi-stage language would be daunting in the sense of guaranteeing type safety and scope safety, because of its vast complexity. Although control operators shift and reset can express dynamically-bound variables (thus, the third example is expressible), existing work on combining control operators with multi-stage language severely limit the power

[1] If we are allowed to manipulate the code *after* it is generated, code motion is easily achieved. However, code manipulation breaks semantic coherence of the multi-stage programming language. For instance, if our language allows to extract e from the code `<fun x → e>`, the variable x may appear freely, and the lexical-scope discipline (hygiene) is lost.

of control operators, to obtain type safety property, which leads to inability to express even the first example.

This paper solves all the above cases using the control operators shift0 and reset0 in a type-safe manner. To our knowledge, this paper is the first to use shift0 and reset0 in multi-stage programming. The difference between shift0/reset0 and shift/reset is subtle but important in this study, as the former can express code motion beyond nested delimiters (reset0), while the latter cannot. By using shift0 and reset0 instead of shift and reset, we can truly move possibly open code beyond binders and share its value across different branches of function calls.

It should be noted that there are two proposals which provide different solutions to the same problem above.

The first one is the one by the second author of the present paper and others [7], who proposed StagedHaskell, a meta-programming library on top of Haskell. This library allows arbitrary monadic effects in multi-stage programming, but they did not give any type system (other than the implementation in Haskell) and it is rather difficult to predict when a program typechecks. We can say that the present paper formalizes a type system which corresponds to some intelligible subset of the library, but the precise relation between these two works is yet to be studied.

Another closely related work is the 'genlet' primitive [11] implemented in BER MetaOCaml, the latest version of MetaOCaml. It is a built-in facility to insert let *automatically*, and the language system automatically (and dynamically) decides an 'optimal' destination of let insertion. Although this approach is very useful in practice, there remains a problem in determining the optimal destination. Assume that we use genlet in the following code generator:

```
if <e1> then (genlet <big_computation>) + <
    e2> else <e3>
```

For this term, there are two possible destination for the let-insertion, and each gives a different result as follows:

```
(1) <if e1 then let x = big_computation in x
     + e2' else e3>
(2) <let x = big_computation in if e1 then x
     + e2' else e3'>
```

where x is a fresh variable, and e2' and e3', resp. are obtained by replacing big_computation in e2 and e3, resp. by x.

The current implementation of BER MetaOCaml chooses (2) as the result, which is sufficient for many cases, but not all. There are situations where (1) is more preferred than (2). For instance, computing the value of big_computation may be meaningful only when the condition e1 is true. This happens if e1 is the expression i >= 0 and big_computation uses the value of a.(i), for example. Then running the code (2) may raise an exception while (1) may not. Also, the performance of (1) is sometimes better than (2) when the computation of e3 does not need the value of big_computation and the test e1 is false.

This simple analysis indicates that deciding the optimal destination of let insertion is not easy if possible at all, and there is little hope for a language system to find it in the cleverest way for all cases. Staging, by definition, gives the programmers the full control over generated code, and we want our multi-stage language to let the programmer choose the best solution by herself. In this sense, each of our approach and the 'genlet' approach has its own merit. It is interesting to study the relationship between the two and how we can combine them in a single language, which is left as a future work.

This paper proposes a calculus for multi-stage programming with control operators which allows safe let-insertion in multiple levels. For this purpose, we design a type system which has a refined notion of environment classifiers, and prove its type soundness. We demonstrate relatively small examples to show how multiple-level let-insertion can be safely done in our calculus. We also briefly mention type inference for our calculus.

The contribution of this paper is summarized as follows:

- We show how the control operators shift0 and reset0 are useful in the context of multi-stage programming by examples. To the best of our knowledge, this is the first such study.
- We design a type system for our calculus and prove its soundness, which gives the static safety guarantee for all code generated by typable code generators.
- We argue the design space of the MetaML style languages with computational effects.

This paper is organized as follows: Section 2 explains the problem and our ideas using examples and informal arguments. Section 3 introduces the language of our study, a two-stage programming language with the control operators, and Section 4 introduces our type system in detail, which is the core of our study, and Section 5 gives a few examples of type derivations. Section 6 explains the type soundness property of our type system. Section 7 briefly mentions a type inference algorithm for our type system under a certain reasonable assumptions. Section 8 states conclusion and related work.

2 Our solution, informally

This section informally explains our key ideas in this study. A formal treatment is given in subsequent sections.

2.1 Control Operators shift0 and reset0

Control operators in functional programs provide control abstraction to contribute modular programming. In the literature, one of the most intensively studied control operators are shift and reset proposed by Danvy and Filinski [2] with the following reduction rule:

```
reset (E [shift k → e])
⇝ reset (e {k := fun x → reset (E[x])})
```

where E is an evaluation context[2], and e is a term. The notation {k:= ...} denotes the standard, capture-avoiding substitution. The above reduction rule shows that shift captures the evaluation context delimited by the closest reset.

Let-insertion for code motion and code sharing can be implemented by shift and reset. Suppose we generate code for E[let x = e1 in e2] while moving the fragment 'let x=e1 in' past E. We only have to add control operators to this term as follows:

```
reset (E [shift k →
          let x = e1 in throw(k,e2) ]
```

where reset represents the *destination*. The code fragment between shift and throw is moved to the destination, and the result is reset (let x = e1 in reset(E[e2])). In this example (and subsequent examples) we use the throw primitive, but readers may regard throw(k,e) as simple application k e.

The above program does not really do let insertion, since let expression is evaluated. But it can be turned into a code generator which does let insertion, as follows:

```
reset (E' [shift k →
          let x = <e1> in throw(k,<e2>) ]
```

where E' is a code-level evaluation context.

The control operators shift and reset are expressible enough for small examples, but we soon see their limitation for large examples. In the previous section, we saw a code generator for nested for-loop where we want to insert two let-expressions to two different places in code. Since shift can never escape from the inner-most reset, we can only insert a let-expression to the closest reset, and not to another reset.

In this study we use the control operators shift0 and reset0 to remedy this problem. As the name suggests, they are similar, but have slightly different semantics than shift and reset:

```
reset0 (E [shift0 k → e])
⇝ e {k:= fun x → reset0 (E[x])})
```

Notice the absence of the outermost reset. This small difference makes shift0 and reset0 more powerful than shift and reset. To see it, let us pick up a slightly complicated example:

```
reset0 (E2[ reset0 (E1 [
  shift0 k1 → shift0 k2 →
  let x = e1 in throw(k2,throw(k1,e2))])])
```

which evaluates to

```
reset0 (E2[ shift k2 →
  let x = e1 in throw(k2,reset0 (E1[e2]))])
```

and then to

```
let x = e1 in reset0 (E2[reset0 (E1[e2])])
```

[2]Precisely speaking, E must not contain another **shift** which surrounds the hole of the context.

The final program shows that we can insert let past two reset0s, by repeating the operator shift0 twice in the source term. Put differently, shift0 can reach at the second (and further) closest reset0, and thus can achieve the multi-level let-insertion.

Note that shift and reset are macro-expressible[3] by shift0 and reset0, and therefore, all the programs which use shift and reset can be written with shift0 and reset0 instead of them. The following encoding is sufficient:

```
reset e       := reset0 e
shift k → e := shift0 k → reset0 e
```

Recently, Materzok and Biernacki [13, 14] intensively studied shift0 and reset0 to give a type theory, a CPS translation and equational axiomatization. In this paper we use their type system as the basis of our type system.

2.2 Environment Classifiers and their Refinement

First, we briefly review classic environment classifiers and refined one before stating our extension.

In their original form [17], an environment classifier (classifier for short) is an abstract representation of a set of free code-level variables. For instance, the expression <x + y> has two code-level variables x and y, and by associating a classifier α to {x,y} , the expression is typed as

$$(x : int)^{\alpha}, (y : int)^{\alpha} \vdash \texttt{<}x + y\texttt{>} : \langle int \rangle^{\alpha}$$

where $(x : int)^{\alpha}$ shows that x is a code-level variable associated with the classifier α. $\langle int \rangle^{\alpha}$ is the type of integer code associated with α. The above typing judgment tells that the term <x+y> is open with respect to α, since its typing context contains a variable associated with α. The classifier α is abstract like a variable, and will never be instantiated with concrete expressions such as constants. Rather, it is a 'name' for the typing environment $x : int, y : int$, and is used to distinguish one typing environment from another.

One can derive the judgment:

$$\vdash \texttt{<}\lambda x.\lambda y.x + y\texttt{>} : \langle int \to int \to int \rangle^{\alpha}$$

which means that this term is closed with respect to α. We can run this term in the language by:

$$\vdash run(\texttt{<}\lambda x.\lambda y.x + y\texttt{>}) : int \to int \to int$$

Thus, a classifier is used to judge if a term contains no free code-level variables associated with the classifier, and after that, we can safely run the code with no risk of running open code. Later, Calcagno, Moggi and Taha [1] proposed a polymorphic calculus for classifiers λi, which forms the basis of the first version of MetaOCaml.[4]

[3]The precise definition of macro-expressivity is given by Felleisen [5].

[4]The latest version of MetaOCaml is Kiselyov's BER MetaOCaml [10] which does not use environment classifiers, and does dynamic checking of free variables in generated code. He eliminated classifiers from the language in order to make its maintenance easier.

Recently, Kiselyov et al. proposed *refined environment classifiers* to give a type system for a two-stage language with mutable cells [12]. Their study introduced a finer structure for classifiers than Taha and Nielsen's. The set of classifiers in their calculus is a partially ordered set, and the lexical scope (no scope extrusion) is enforced by the rule similar to the eigen-variable condition in the natural deduction style logic.

Consider the example <$\lambda x.\lambda y.x + y$> again. Kiselyov et al.'s calculus associates one classifier to each code-level variable; x is associated with a classifier γ_1 and y with γ_2. The partial order is determined by the lexical scope of variables with the reverse inclusion order; here x has a larger lexical scope than y does, hence we write $\gamma_2 \geq \gamma_1$.[5] The following typing rule for code-level abstraction is a simplified version of their typing rule:

$$\frac{\Gamma,\ \gamma_1 \geq \gamma,\ x : \langle t_1 \rangle^{\gamma_1} \vdash e : \langle t_2 \rangle^{\gamma_1}}{\Gamma \vdash \underline{\lambda} x.e : \langle t_1 \to t_2 \rangle^{\gamma}}\ (\gamma_1 \text{ is eigen variable})$$

The classifier outside of the term is γ and the classifier corresponding to this x is γ_1. The assumption $\gamma_1 \geq \gamma$ means that the scope of γ has a larger lexical scope than γ_1. Furthermore, the side condition says that γ_1 must not appear in the conclusion of this rule. Kiselyov et al. have successfully designed a type system for a two-level language with global mutable cells using the above rule.

In this paper, we extend the refined environment classifiers to accommodate control operators. The problem to be solved is that control operators can move code, and the lexical scope of generated code is not preserved during the code generation.

Let us illustrate the problem by a simple example as follows:

$$\begin{array}{l} \underline{\textbf{let}}\ u = \cdots\ \underline{\textbf{in}} <e_0> \underline{+} \\ \quad \textbf{reset0}\ (\underline{\textbf{let}}\ x = \cdots\ \underline{\textbf{in}} <e_1> \underline{+} \\ \qquad \textbf{shift0}\ k\ \to\ \underline{\textbf{let}}\ y = \cdots\ \underline{\textbf{in}} <e_2> \underline{+} \\ \qquad\qquad \textbf{throw}(k, <e_3>)) \end{array}$$

which evaluates to the following code value:

$$\begin{array}{l} \langle let\ u = \cdots\ in\ e_0 + \\ \ (let\ y = \cdots\ in\ e_2 + \\ \ (let\ x = \cdots\ in\ e_1 + e_3))\rangle \end{array}$$

In this example we used the code combinator $\underline{\textbf{let}}$ to generate code of a let-expression. $\underline{\textbf{let}}\ x = e_1\ \underline{\textbf{in}}\ e_2$ is macro-defined by $(\underline{\lambda} x.e_2)\ \underline{@}\ e_1$ where $\underline{@}$ is a code combinator for generating application. Assuming that all bound variables have different names, we have that (0) e_0 may not contain x, y free and (0') e_1 may not contain y free. (We ignored the variable k since it is a present-stage variable.)

If e_2 contains the variable x free, the result of the above evaluation would produce an ill-scoped code; the notorious scope extrusion problem. To avoid it, we require that (1) e_2 may contain u and y free, but not x, and (2) e_3 may contain u, x, and y.

Fig. 1 illustrates the set of free variables for each term under the conditions (0), (0'),(1), and (2) where arrows show the set inclusion order. (The classifiers γ_i in the figure shall be explained shortly.)

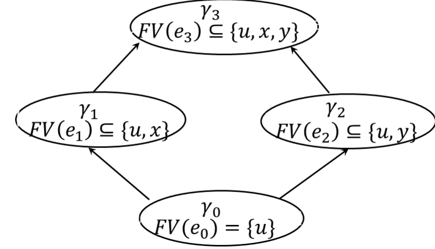

Figure 1. Lattice of environment classifiers

We want to represent the above conditions (1) and (2) in terms of environment classifiers. For this purpose, we associate the environment classifiers $\gamma_1, \gamma_2, \gamma_3$, resp., to the code variables u, x, and y, resp. and let γ_0 be the classifier associated with the top level of the above term. Then the four classifiers must form the partially order set, or lattice, shown in Fig. 1. The important point here is that, even though the scope for x is larger than y, we should not have $\gamma_2 \geq \gamma_1$ because of the condition (1). Also, the scope of γ_3 must be smaller than other classifiers, namely, $\gamma_3 \geq \gamma_i$ should hold for $i = 0, 1, 2$ because of the condition (2). In other words, γ_3 is an upper bound of γ_1 and γ_2, and since there are no other classifiers between them, we can actually regard γ_3 as the least upper bound of γ_1 and γ_2 (or their join). This is contrast to Kiselyov et al.'s work [12] where the set of environment classifiers forms a tree, reflecting the inclusion order of scopes.

In general, we do not need the lattice structure, but an upper semi-lattice[6] suffices; To represent the semi-lattice structure, we introduce the join operator $\cup$ to the classifiers, which is our key innovation in this paper. We write the join of γ_1 and γ_2 as $\gamma_1 \cup \gamma_2$ to emphasize the intuition of set union.

Adding the join operator to the set of classifiers is a simple extension, but it turned out to be very powerful. In the subsequent sections, we will show that a type-safe calculus for the two-stage language with shfit0 and reset0 can be given based on this semi-lattice structures of classifiers.

[5]For historical reasons, we write $\gamma_2 \geq \gamma_1$ if γ_1 has a larger lexical scope.

[6] An upper semi-lattice is a partially order set where the least upper bound (join) of two elements always exist.

3 Language

In this section, we present a calculus for two-stage programming with the control operators shift0 and reset0. A type system for the calculus will be given in the next section.

3.1 Syntax

Our language is the simply typed lambda calculus with the primitives for integer arithmetic, conditional, code generation and control operators.

$$
\begin{aligned}
c &::= i \mid b \mid \underline{\textbf{int}} \mid + \mid \underline{@} \mid \underline{+} \mid \underline{\texttt{if}} \\
v &::= x \mid c \mid \lambda x.e^0 \mid <e^1> \\
e^0 &::= v \mid e^0\ e^0 \mid \textbf{if}\ e^0\ \textbf{then}\ e^0\ \textbf{else}\ e^0 \\
&\quad \mid \underline{\lambda} x.e^0 \mid \underline{\underline{\lambda}} u.e^0 \\
&\quad \mid \textbf{reset0}\ e^0 \mid \textbf{shift0}\ k \to e^0 \mid \textbf{throw}(k, v) \\
e^1 &::= u \mid c \mid \lambda u.e^1 \mid e^1\ e^1 \mid \textbf{if}\ e^1\ \textbf{then}\ e^1\ \textbf{else}\ e^1
\end{aligned}
$$

Figure 2. Constants, values, and level-0 and level-1 terms

Figure 2 defines the syntax of the language. Constants (c) include basic constants of integer (i) and boolean (b), primitive operators (+), and code combinators such as $\underline{\textbf{int}}$ and $\underline{@}$. Code combinators are primitives to generate a code value. For instance, $\underline{\textbf{int}}$ 5 evaluates to the code value $<5>$. We sometimes write %e instead of $\underline{\textbf{int}}\ e$. Although we did not include the let-expression, the fixpoint operator and the corresponding code combinators, they can be added to our language in the standard way.

A value (v) is either a variable (x), a constant (c), a lambda abstraction, or a code value $<e^1>$. Unlike the calculi in the literature such as $\lambda\alpha$ [17], an expression in the form $<e^1>$ in our calculus is always a value, since we do not have the 'escape' primitive for splicing. (Recall that we are using the code-combinator style rather than the quasi-quotation style.)

Terms are classified into level-0 terms (e^0) and level-1 terms (e^1). We use u for level-1 variables (code-level variables). We have restricted our attention to two-level calculi. This is an important restriction from $\lambda\alpha$, which allows an arbitrarily high level, thus code like <<1+2>> can be written in $\lambda\alpha$ but not in our calculus. Since level-2 or higher terms rarely appear in practical code generators, and higher-level terms would complicate the formulation, we restrict our attention in this paper to two-level calculi. We anticipate that relaxing this restriction is not too difficult, but it is left for future work. A level-0 term (e^0) is either a value v, application $e^0\ e^0$, conditional, terms built with code combinators $\underline{\lambda} x.e^0$, $\underline{\underline{\lambda}} u.e^0$, or terms built with control operators $\textbf{reset0}\ e^0$, $\textbf{shift0}\ k \to e^0$, $\textbf{throw}(k, v)$. We have restricted the argument of the throw expression to be a value v, but this is not significant, as we can represent $\textbf{throw}(k, e)$ (for an arbitrary expression e) by $\textbf{let}\ x = e\ \textbf{in}\ \textbf{throw}(k, x)$.

Level-0 variables (x) are bound by lambda-abstraction $\lambda x.\cdots$ and code-level lambda abstraction $\underline{\lambda} x.\cdots$, and level-1 variables (u) are bound by another code-level abstraction $\underline{\underline{\lambda}} u.\cdots$, which will be explained later. Continuation variables (k) are bound by shift0. We identify α-equivalent terms in the usual sense, and rename bound variables if necessary. Note that our language has the standard lexical scopes.

3.2 Operational Semantics

We define the call-by-value small-step operational semantics $\leadsto$ in the evaluation-context style by $E[r] \leadsto E[l]$ if $r \to l$, where E is an evaluation context defined in Fig. 3, and the reduction relation $\to$ is defined in Fig. 4.

$$
\begin{aligned}
E &::= [\ ] \mid E\ e^0 \mid v\ E \\
&\quad \mid \textbf{if}\ E\ \textbf{then}\ e^0\ \textbf{else}\ e^0 \mid \textbf{reset0}\ E \mid \underline{\underline{\lambda}} u.E
\end{aligned}
$$

Figure 3. Evaluation Context

$$
\begin{aligned}
(\lambda x.e)\ v &\to e\{x := v\} \\
\textbf{if}\ \mathit{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 &\to e_1 \\
\textbf{if}\ \mathit{else}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 &\to e_2 \\
\underline{\lambda} x.e &\to \underline{\underline{\lambda}} u.(e\{x := <u>\}) \\
\underline{\underline{\lambda}} u.<e> &\to <\lambda u.e> \\
\textbf{reset0}\ v &\to v \\
\textbf{reset0}(E[\textbf{shift0}\ k \to e]) &\to e\{k \Leftarrow E\}
\end{aligned}
$$

where u is fresh in the fourth rule and E must not capture shift0 in the last rule.

Figure 4. Reduction Rules

Let us explain the reduction rules in Fig. 4. The first rule is the call-by-value β rule where $e\{x := v\}$ denotes the capture-avoiding substitution for the variable x in e. The second and third rules are the standard one for conditional expressions. The next two rules define the behavior of code-level abstraction $\underline{\lambda} x.e$. Given this term, we first change the level-0 variable x by $<u>$ where u is a level-1 variable (code variable). Note that a code variable is bound by doubly underlined lambda as $\underline{\underline{\lambda}} u.e$. Then we evaluate its body e and when it gets to a code value $<e'>$, we replace the whole expression by $<\lambda u.e'>$.

The last two rules are the rules for evaluating an expression reset0 e. If e is already a value, then the delimiter reset0 is discarded (the second to last rule). If e takes the form $E[\textbf{shift0}\ k \to e]$ then this **shift0** captures the continuation

up to **reset0**, namely E, and substitutes E for the continuation variable k. The substitution for a continuation variable is defined in Fig. 5 below.

$$(\textbf{throw}(k,v))\{k \Leftarrow E\} \equiv \textbf{reset0}(E[v])$$
$$(\textbf{throw}(k',v))\{k \Leftarrow E\} \equiv \textbf{throw}(k',(v\{k \Leftarrow E\})) \quad \text{if } k \neq k'$$

Figure 5. Substitution for continuation variables

The expression $e\{k \Leftarrow E\}$ denotes the result of substituting an evaluation context E for a continuation variable k in a term e. The substitution commutes with most constructors and the only interesting case is when $e \equiv \textbf{throw}(k,v)$, which is shown on the first line above. In this case, the continuation variable k is replaced by the captured context E, but during this computation, we need to add one **reset0** above E. This may be seen as a strange behavior, but it is essential to obtain the semantics of the delimited-control operator shift0 (and also shift). See the papers on control operators [2, 13] for details. The substitution is capture avoiding in the sense that $(\lambda x.e)\{k \Leftarrow E\}$ is defined only when x does not occur free in E.

Finally we give reduction rules for constants. Reductions for the standard constants such as integer-addition are given in the standard way, and omitted. Fig. 6 gives the reduction rules for code combinators, which work at the present stage, but manipulate code values.

$$\underline{\textbf{int}}\ n \to <n>$$
$$<e_1>\ \underline{@}\ <e_2> \to <e_1\ e_2>$$
$$<e_1>\ \underline{+}\ <e_2> \to <e_1 + e_2>$$
$$\underline{\textbf{if}}\ <e_1>\ <e_2>\ <e_3> \to <\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3>$$

Figure 6. Evaluation Rules for Code Combinators

3.3 Examples

We give a simple example of evaluation in our calculus.

Let e_1 be the following term:

$$\begin{aligned} e_1 = {} & \textbf{reset0}\ (\underline{\textbf{let}}\ x = \%3\ \underline{\textbf{in}} \\ & \textbf{shift0}\ k \ \to\ \underline{\textbf{let}}\ y = \%7\ \underline{\textbf{in}} \\ & \textbf{throw}(k, x \mathbin{\underline{+}} y))) \end{aligned}$$

Then its evaluation under the above semantics goes in the following way:

$$\begin{aligned} e_1 = {} & \textbf{reset0}\ ((\underline{\lambda} x. \\ & \textbf{shift0}\ k \ \to\ \underline{\textbf{let}}\ y = \%7\ \underline{\textbf{in}} \\ & \textbf{throw}(k,\ x \mathbin{\underline{+}} y))\ \underline{@}\ \%3) \\ \rightsquigarrow^* & \textbf{reset0}\ ((\underline{\underline{\lambda}} x'. \\ & \textbf{shift0}\ k \ \to\ \underline{\textbf{let}}\ y = \%7\ \underline{\textbf{in}} \\ & \textbf{throw}(k, (<x'> \mathbin{\underline{+}} y))\ \underline{@}\ \%3) \\ \rightsquigarrow & \underline{\textbf{let}}\ y = \%7\ \underline{\textbf{in}} \\ & \textbf{reset0}\ ((\underline{\underline{\lambda}} x'.<x'> \mathbin{\underline{+}} y)\ \underline{@}\ <3>) \\ = & (\underline{\lambda} y.\textbf{reset0}\ ((\underline{\underline{\lambda}} x'.<x'> \mathbin{\underline{+}} y)\ \underline{@}\ <3>))\ \underline{@}\ \%7 \\ \rightsquigarrow & (\underline{\underline{\lambda}} y'.\textbf{reset0}\ ((\underline{\underline{\lambda}} x'.<x'> \mathbin{\underline{+}} <y'>)\ \underline{@}\ <3>))\ \underline{@}\ \%7 \\ \rightsquigarrow & (\underline{\underline{\lambda}} y'.\textbf{reset0}\ ((\underline{\underline{\lambda}} x'.<x' + y'>)\ \underline{@}\ <3>))\ \underline{@}\ \%7 \\ \rightsquigarrow & (\underline{\underline{\lambda}} y'.\textbf{reset0}\ (<(\lambda x'.x' + y')\ 3>))\underline{@}\ \%7 \\ \rightsquigarrow^* & <(\lambda y'.(\lambda x'.x' + y')\ 3)\ 7> \end{aligned}$$

The result is equivalent to

$$<\textbf{let}\ y' = 7\ \textbf{in}\ \textbf{let}\ x' = 3\ \textbf{in}\ x' + y'>$$

as desired. The let-binding for y' (which was y in the source term) moves upward then the let-binding for x' (which was x in the source term).

Similarly, let e_2 be the following term:

$$\begin{aligned} e_2 = {} & \textbf{reset0}\ (\underline{\textbf{let}}\ x_1 = \%3\ \underline{\textbf{in}} \\ & \textbf{reset0}\ (\underline{\textbf{let}}\ x_2 = \%5\ \underline{\textbf{in}} \\ & \textbf{shift0}\ k_2 \ \to\ \textbf{shift0}\ k_1 \ \to\ \underline{\textbf{let}}\ y = \%7\ \underline{\textbf{in}} \\ & \textbf{throw}(k_1\ ,\textbf{throw}(k_2, x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)))) \end{aligned}$$

Then we can evaluate it to the code:

$$e_2 \rightsquigarrow <(\lambda y'.(\lambda x_1'.(\lambda x_2'.x_1' + x_2' + y')\ 5)\ 3)\ 7>$$

which is equivalent to

$$<\textbf{let}\ y' = 7\ \textbf{in}\ \textbf{let}\ x_1' = 3\ \textbf{in}\ \textbf{let}\ x_2'\ \textbf{in}\ x_1' + x_2' + y'>$$

as desired.

These examples show that our calculus is capable of inserting let to multiple points (multi-layer let-insertion).

We do not show the examples of expressing code sharing, as it is well studied in the literature [7, 8] in depth, using shift and reset. As we have seen in Section 2, every program expressible with shift and reset is expressible with shift0 and reset0.

4 Type System

We introduce a type system for two-level staged language with control operators shift0 and reset0. Our type system is based on roughly two source of type systems with some new ideas. First, it is based on the type system with refined environment classifiers while we extended to include the join operator. Second, our type system is based on effect typing

for shift0 and reset0, which is a simplified type system of Materzok and Biernacki's.

4.1 Our Type System

We define basic type (b) and environment classifier γ as follows:

$$b ::= \texttt{int} \mid \texttt{bool}$$
$$\gamma ::= \gamma_x \mid \gamma \cup \gamma$$
$$L ::= \cdot \mid \gamma$$

Figure 7. Basic Type, Classifier, and Level

In this figure, γ_x is a classifier-variable, and $\cup$ is the join operator for two classifiers. We use γ for classifiers.

The level $\cdot$ denotes the present stage, and the level γ denotes the future (next) stage corresponding to the classifier γ. (It is abuse of notation to use γ for a level, too, but there should be no confusion.) For instance, $\Gamma \vdash^L e : t \,;\, \sigma$ is a present-stage judgment if $L = \cdot$, and a future-stage judgment if $L = \gamma$. The symbol for the present stage $\cdot$ is often omitted.

Fig. 8 defines a level-0 type t^0, a level-1 type t^1, a level-0 type sequence σ, and a level-0 continuation type κ .

$$t^0 ::= b \mid t^0 \xrightarrow{\sigma} t^0 \mid \langle t^1 \rangle^\gamma$$
$$t^1 ::= b \mid t^1 \to t^1$$
$$\sigma ::= \epsilon \mid t^0, \sigma$$
$$\kappa^0 ::= \langle t^1 \rangle^\gamma \overset{\sigma}{\Rightarrow} \langle t^1 \rangle^\gamma$$

Figure 8. Type, Effect Type and Continuation Type

In the definition of σ, the first item ϵ denotes the empty sequence. A level-0 function type $t^0 \xrightarrow{\sigma} t^0$ is accompanied with a sequence σ, which represents the types for computational effects during the computation of the function body. More concretely, it is the sequence of *answer types* for shift0. As we explained earlier, shift0 can access beyond multiple occurrences of reset0, the type system must be aware of n answer types where n is the number of layers reachable from the term being typed. Thus the effect type in the typing judgment needs n answer types and we write it as a sequence σ. Unlike Materzok and Biernacki's type system, σ in our calculus is a simple list of types, since we do not need answer type modification in our calculus. Interested readers are encouraged to refer to their work [13].

Control operators in our calculus do not appear in the code value, and thus, level-1 function types do not have the effect types. As we only use shift0 and reset0 for code manipulation, the type for continuations is the function type from a code type to a code type. We treat continuation types differently from ordinary function types for technical reasons.

$$\frac{}{\Gamma \models \gamma_1 \geq \gamma_1} \qquad \frac{}{\Gamma, \gamma_1 \geq \gamma_2 \models \gamma_1 \geq \gamma_2}$$

$$\frac{\Gamma \models \gamma_1 \geq \gamma_2 \quad \Gamma \models \gamma_2 \geq \gamma_3}{\Gamma \models \gamma_1 \geq \gamma_3}$$

$$\frac{}{\Gamma \models \gamma_1 \cup \gamma_2 \geq \gamma_1} \qquad \frac{}{\Gamma \models \gamma_1 \cup \gamma_2 \geq \gamma_2}$$

$$\frac{\Gamma \models \gamma_3 \geq \gamma_1 \quad \Gamma \models \gamma_3 \geq \gamma_2}{\Gamma \models \gamma_3 \geq \gamma_1 \cup \gamma_2}$$

Figure 9. Rules for Classifier Ordering

Typing judgments take either of the following two forms:

$$\Gamma \vdash^L e : t;\ \sigma$$
$$\Gamma \models \gamma \geq \gamma$$

where a typing context Γ is defined as follows:

$$\Gamma ::= \emptyset \mid \Gamma, (\gamma \geq \gamma) \mid \Gamma, (x : t) \mid \Gamma, (u : t)^\gamma$$

We now introduce typing rules. First, we give the rules for the typing judgment $\Gamma \models \gamma \geq \gamma$ in Fig. 9.

The derivation rules determine the semi-lattice structure of classifiers with $\cup$ being the join operator. The ordering of classifiers are either derived by the context Γ, or the semi-lattice rules[7].

We then give the type derivation rules for $\Gamma \vdash^L e : t;\ \sigma$. We first give simple cases of the level-0 rules in Fig. 10.

All rules in Fig. 10 are standard except that we have levels, and function spaces are annotated with effect types σ, and also the judgments are annotated similarly. The rule for constants (c) assumes that their types (t^c) are already given in some way. For instance, $t^{17} = \texttt{int}$ and $t^{\underline{+}} = \langle\texttt{int}\rangle^\gamma \xrightarrow{\sigma} \langle\texttt{int}\rangle^\gamma \xrightarrow{\sigma} \langle\texttt{int}\rangle^\gamma$. The codo-combinator for application has a complicated type: $t^{\underline{@}} = \langle t_1 \to t_2 \rangle^\gamma \xrightarrow{\sigma} \langle t_1 \rangle^\gamma \xrightarrow{\sigma} \langle t_2 \rangle^\gamma$.

$$\frac{\Gamma \vdash^\gamma e : t^1; \sigma}{\Gamma \vdash \texttt{<}e\texttt{>} : \langle t^1 \rangle^\gamma; \sigma}$$

$$\frac{\Gamma,\ \gamma_1 \geq \gamma,\ x : \langle t_1 \rangle^{\gamma_1} \vdash e : \langle t_2 \rangle^{\gamma_1}; \sigma}{\Gamma \vdash \underline{\lambda} x.e : \langle t_1 \to t_2 \rangle^\gamma\ ;\ \sigma}\ (\gamma_1 \text{ is eigen variable})$$

Figure 11. Typing rule for code

Fig. 11 lists the rules for code-value and code-level abstraction. The first rule is the standard code construction rule found in the literature [17]. The second rule in Fig. 11 is

[7] We omit the anti-symmetric law, which is not necessary to derive $\gamma_1 \geq \gamma_2$.

$$\frac{}{\Gamma, x : t \vdash x : t \ ; \ \sigma} \quad \frac{}{\Gamma, (u : t)^{\gamma} \vdash^{\gamma} u : t \ ; \ \epsilon}$$

$$\frac{}{\Gamma \vdash^{L} c : t^{c} \ ; \ \sigma}$$

$$\frac{\Gamma \vdash^{\gamma} e_1 : t_2 \to t_1; \ \epsilon \quad \Gamma \vdash^{\gamma} e_2 : t_2 \ ; \ \epsilon}{\Gamma \vdash^{\gamma} e_1 \, e_2 : t_1; \ \epsilon}$$

$$\frac{\Gamma \vdash e_1 : t_2 \xrightarrow{\sigma} t_2; \ \sigma \quad \Gamma \vdash e_2 : t_2; \ \sigma}{\Gamma \vdash e_1 \, e_2 : t_1; \ \sigma}$$

$$\frac{\Gamma, \ x : t_1 \vdash e : t_2 \ ; \ \sigma}{\Gamma \vdash \lambda x.e : t_1 \xrightarrow{\sigma} t_2 \ ; \ \sigma'} \quad \frac{\Gamma, \ (u : t_1)^{\gamma} \vdash^{\gamma} e : t_2 \ ; \ \epsilon}{\Gamma \vdash^{\gamma} \lambda u.e : t_1 \to t_2 \ ; \ \epsilon}$$

$$\frac{\Gamma \vdash^{L} e_1 : \texttt{bool}; \ \sigma \quad \Gamma \vdash^{L} e_2 : t; \sigma \quad \Gamma \vdash^{L} e_3 : t; \sigma}{\Gamma \vdash^{L} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t \ ; \ \sigma}$$

Figure 10. Rules for $\Gamma \vdash^{L} e : t; \ \sigma$ (simple cases)

the key rule in Kiselyov et al.'s work [12], and we use it in our type system. The side-condition means that γ_1 must not appear in the conclusion $\Gamma \vdash \underline{\lambda} x.e : \langle t_1 \to t_2 \rangle^{\gamma} \ ; \ \sigma$.

$$\frac{\Gamma \vdash e : \langle t \rangle^{\gamma} \ ; \ \langle t \rangle^{\gamma}, \sigma}{\Gamma \vdash \textbf{reset0} \ e : \langle t \rangle^{\gamma} \ ; \ \sigma}$$

$$\frac{\Gamma, \ k : \langle t_1 \rangle^{\gamma_1} \overset{\sigma}{\Rightarrow} \langle t_0 \rangle^{\gamma_0} \vdash e : \langle t_0 \rangle^{\gamma_0}; \sigma \quad \Gamma \models \gamma_1 \geq \gamma_0}{\Gamma \vdash \textbf{shift0} \ k \to e : \langle t_1 \rangle^{\gamma_1} \ ; \ \langle t_0 \rangle^{\gamma_0}, \sigma}$$

$$\frac{\Gamma \models \gamma_2 \geq \gamma_0 \qquad \Gamma, \ k : \langle t_1 \rangle^{\gamma_1} \overset{\sigma}{\Rightarrow} \langle t_0 \rangle^{\gamma_0} \vdash v : \langle t_1 \rangle^{\gamma_1 \cup \gamma_2}; \ \sigma}{\Gamma, \ k : \langle t_1 \rangle^{\gamma_1} \overset{\sigma}{\Rightarrow} \langle t_0 \rangle^{\gamma_0} \vdash \textbf{throw}(k, v) : \langle t_0 \rangle^{\gamma_2}; \ \sigma}$$

Figure 12. Rules for Control Operators

Fig. 12 introduces the rules for control operators. The reset0 rule is the standard one (see the literature [2, 14]) except that the argument of reset0 must be of code type. Namely, in our calculus control operators are restricted to code combinators which receive code values, and return code values, and may not be used in ordinary computation or in the generated code. We stress that this is not a severe restriction; we can introduce the 'standard' control operators whose arguments are not necessarily of code types, in which case we do not have to worry about classifiers. Such typing rules are standard and well studied, so we did not include them to avoid clutter.

$$\frac{\Gamma \vdash e : \langle t \rangle^{\gamma_1}; \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash e : \langle t \rangle^{\gamma_2}; \sigma}$$

$$\frac{\Gamma \vdash^{\gamma_1} e : t \ ; \ \epsilon \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash^{\gamma_2} e : t \ ; \ \epsilon}$$

$$\frac{\Gamma \vdash e : t; \ \langle t' \rangle^{\gamma_1}, \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash e : t; \ \langle t' \rangle^{\gamma_2}, \sigma}$$

$$\frac{\Gamma \vdash^{L} e : t_1; \ \sigma}{\Gamma \vdash^{L} e : t_1; \ \sigma, t_2}$$

Figure 13. Subsumption rules

The shift0 rule is complicated, but by ignoring code types, it is the standard typing rule in [14]. The noticeable point here is that the body of the shift0 expression e in the rule has the code type of level γ_0 such that $\gamma_1 \geq \gamma_0$. While all other rules enforce that a subterm's classifier must be larger than the superterm's classifier, this single rule enforces an opposite order.

The throw rule is also complicated, and we do not give detailed account here, but the point is that the body of throw expression v has the code type with the classifier $\gamma_1 \cup \gamma_2$, indicating that this is a join point of two separated (by the shift0 rule) lines of classifier ordering.

As auxiliary rules, we have the rules for subsumption in Fig. 13. The first three rules can be understood in the following way: $\gamma_2 \geq \gamma_1$ means that the context (the set of available free variables) of γ_2 is a superset of that of γ_1, hence all terms typable at γ_1 are also typable at γ_2. The last rule is a subsumption for effect-types; intuitively, it means a non-effectful term may be regarded as an effectful term, which is natural and standard.

5 Examples of Type Derivation

Here we show a few examples of type derivation.

5.1 Single-layer Let-insertion

Consider the following term which performs let-insertion using shift0 and reset0.

$$\begin{aligned} e = \ & \underline{\textbf{let}} \ x_1 = e_1 \ \underline{\textbf{in}} \\ & \textbf{reset0}(\ \underline{\textbf{let}} \ x_2 = e_2 \ \underline{\textbf{in}} \\ & \textbf{shift0} \ k \ \to \ \underline{\textbf{let}} \ y = \Box \ \underline{\textbf{in}} \\ & \textbf{throw}(k, y)) \end{aligned}$$

where $\Box$ is to be filled with some term. The expression e inserts the let-expression with e_2 to the destination specified by reset0, hence it should be typable if $\Box = \underline{\textbf{int}} \ 7$ or $\Box = x_1$, but not for $\Box = x_2$. Fig. 14 shows a type derivation for e.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1,\ k : \langle t\rangle^{\gamma_2} \overset{\epsilon}{\Rightarrow} \langle t\rangle^{\gamma_1}, \gamma_3 \geq \gamma_2,\ y : \langle t\rangle^{\gamma_3} \vdash y : \langle t\rangle^{\gamma_2 \cup \gamma_3};\ \epsilon}}{\Gamma_1,\ k : \langle t\rangle^{\gamma_2} \overset{\epsilon}{\Rightarrow} \langle t\rangle^{\gamma_1}, \gamma_3 \geq \gamma_2,\ y : \langle t\rangle^{\gamma_3} \vdash \mathbf{throw}(k, y) : \langle t\rangle^{\gamma_3};\ \epsilon}\ (*1) \qquad \dfrac{}{\Gamma_1,\ k : \langle t\rangle^{\gamma_2} \overset{\epsilon}{\Rightarrow} \langle t\rangle^{\gamma_1} \vdash \Box : \langle t\rangle^{\gamma_1};\ \epsilon}\ (*2)}{\Gamma_1,\ k : \langle t\rangle^{\gamma_2} \overset{\epsilon}{\Rightarrow} \langle t\rangle^{\gamma_1} \vdash \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}}\ \mathbf{throw}(k, y) : \langle t\rangle^{\gamma_1};\ \epsilon}}{\Gamma_1 = \gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1},\ \gamma_2 \geq \gamma_1,\ x_2 : \langle t\rangle^{\gamma_2} \vdash \mathbf{shift0}\ k \to \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}}\ \mathbf{throw}(k, y) : \langle t\rangle^{\gamma_2};\ \langle t\rangle^{\gamma_1}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \mathbf{shift0}\ k \to \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}}\ \mathbf{throw}(k, y) : \langle t\rangle^{\gamma_1};\ \langle t\rangle^{\gamma_1}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \mathbf{shift0}\ k \to \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}}\ \mathbf{throw}(k, y)) : \langle t\rangle^{\gamma_1};\ \epsilon}}{\vdash e = \underline{\mathbf{let}}\ x_1 = e_1\ \underline{\mathbf{in}}\ \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \mathbf{shift0}\ k \to \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}}\ \mathbf{throw}(k, y)) : \langle t\rangle^{\gamma_0};\ \epsilon}$$

Figure 14. Example of Type Derivation (Single-layer let-insertion)

Let us check the inference marked with (*1). The typing rule for throw requires $\gamma_3 \geq \gamma_1$, but it is derivable from the context $\Gamma_1,\ k : \langle t\rangle^{\gamma_2} \overset{\epsilon}{\Rightarrow} \langle t\rangle^{\gamma_1}, \gamma_3 \geq \gamma_2$. We also have to show $y : \langle t\rangle^{\gamma_3} \vdash y : \langle t\rangle^{\gamma_2 \cup \gamma_3};\ \epsilon$, but it is also derivable.

In the inference marked with (*2), $\Box$ must have the type $\langle t\rangle^{\gamma_1}$, which means that $\Box$ may have the variables associated with γ_1 or smaller, but not other variables. We analyse by cases:

When $\Box = \underline{\mathbf{int}}\ 7$
: $\underline{\mathbf{int}}\ 7$ is typed under any classifiers, so the term is typable.

When $\Box = x_1$
: $x_1 : \langle t\rangle^{\gamma_1} \vdash x_1 : \langle t\rangle^{\gamma_1}$ holds and typable.

When $\Box = x_2$
: The classifier associated with x_2 is γ_2, and we cannot derive $\gamma_1 \geq \gamma_2$ from the context, so e is not typable.

5.2 Multi-layer Let-insertion

Let us consider a more interesting example for multi-layer let-insertion as follows:

$$\begin{aligned} e' = {} & \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_1 = e_1\ \underline{\mathbf{in}} \\ & \quad \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}} \\ & \quad \mathbf{shift0}\ k_2\ \to\ \mathbf{shift0}\ k_1\ \to\ \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}} \\ & \quad \mathbf{throw}(k_1, \mathbf{reset0}\ (\mathbf{throw}(k_2, y))))) \end{aligned}$$

To achieve two-layer let-insertion, the term e' has two occurrences of shift0 and throw, and the outer shift0 captures the continuation up to the inner reset0, and the inner shift captures the continuation up to the outer reset0. Iterating throw0 twice is necessary to install the two captured continuation. Here we need an extra occurrence of reset0 between two throws, which is superfluous in the computation, but is necessary in our calculus due to the restriction of our simple type system. It is annoying but may be hidden by providing macros for representing let insertion.

When we run the term e', let for the variable y is inserted at the topmost position of e'. Hence it should be typable if $\Box = \underline{\mathbf{int}}7$, but not be typable if $\Box = x_2$ or $\Box = x_1$.

Let us see the inference marked with (#1) in Fig. 15. The type of k_2 is $\langle t\rangle^{\gamma_2} \overset{\langle t\rangle^{\gamma_0}}{\Rightarrow} \langle t\rangle^{\gamma_1}$ and the effect type (the σ-part) of the whole expression is $\langle t\rangle^{\gamma_1 \cup \gamma_3}$, and by applying the **throw**-rule, we can derive $\Gamma_3 \models \gamma_1 \cup \gamma_3 \geq \gamma_0$.

The inference marked with (#2) shows that the term filled in to $\Box$ should have type $\langle t\rangle^{\gamma_0}$ under the context $\Gamma_2 = \gamma_2 \geq \gamma_1,\ x_2 : \langle t\rangle^{\gamma_2},\ \gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1}, k_2 : \langle t\rangle^{\gamma_2} \overset{\langle t\rangle^{\gamma_0}}{\Rightarrow} \langle t\rangle^{\gamma_1},\ k_1 : \langle t\rangle^{\gamma_1} \overset{\epsilon}{\Rightarrow} \langle t\rangle^{\gamma_0}$. By the same reasoning as before, we conclude that x_1 and x_2 cannot be filled into this $\Box$ and the term is not typeable for these cases.

Thus our type system can distinguish a safe expression from dangerous ones where scope extrusion may occur.

6 Type Soundness

A fundamental property of a typed system (or a typed calculus) is type soundness; type preservation and progress.

Theorem 6.1. *If $\Gamma \vdash e : t\ ;\ \sigma$ is derivable and $e \rightsquigarrow e'$ holds, then we can derive $\Gamma \vdash e' : t\ ;\ \sigma$.*

This theorem can be proved in the standard way. We first need a few lemmas.

Lemma 6.2. *Suppose $\Gamma_1, \gamma_2 \geq \gamma_1 \vdash e : t_1\ ;\ \sigma$ is derivable and γ_2 does not appear in Γ_1, e, t_1, σ. Then $\Gamma_1 \vdash e : t_1\ ;\ \sigma$ is derivable.*

Lemma 6.3. *Suppose $\Gamma_1 \vdash v : t_1\ ;\ \sigma$ is derivable. Then so is $\Gamma_1 \vdash v : t_1\ ;\ \sigma'$.*

Lemma 6.4 (Substitution lemma). *If $\Gamma_1, \Gamma_2, x : t_1 \vdash e : t_2\ ;\ \sigma$ and $\Gamma_1 \vdash v : t_1\ ;\ \sigma$ are derivable, then so is $\Gamma_1, \Gamma_2 \vdash e\{x := v\} : t_2\ ;\ \sigma$.*

We need one more technical lemma as follows.

Lemma 6.5. *Let E be an evaluation context which does not have reset0 surrounding the hole, x be a variable, $\Gamma = (u_1 : t_1)^{\gamma_1}, \cdots, (u_n : t_n)^{\gamma_n}$ and $\Gamma \models \gamma_0 \geq \gamma_i$ is derivable for $i = 1, \cdots, n$. If $\Gamma, x : \langle t_0\rangle^{\gamma'} \vdash E[x] : \langle t_1\rangle^{\gamma_0}\ ;\ \sigma$ then we can derive $\Gamma, x : \langle t_0\rangle^{\gamma' \cup \gamma} \vdash E[x] : \langle t_1\rangle^{\gamma_0 \cup \gamma}\ ;\ \sigma$ for a fresh γ.*

Using these lemmas, we can prove the subject reduction property (Theorem 6.1).

We also have the progress property.

Theorem 6.6 (Progress). *If $\vdash e : t;\ \epsilon$ is derivable, namely, e is a typable term and does not have free variables and free*

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma_3 \vdash y : \langle t\rangle^{\gamma_2\cup\gamma_1\cup\gamma_3};\ \epsilon} \quad \overline{\Gamma_3 \models \gamma_1 \cup \gamma_3 \geq \gamma_0}}{\Gamma_3 \vdash \mathbf{throw}(k_2, y) : \langle t\rangle^{\gamma_1\cup\gamma_3};\ \langle t\rangle^{\gamma_1\cup\gamma_3}}\ (\#1)}{\Gamma_3 \vdash \mathbf{reset0}(\mathbf{throw}(k_2, y)) : \langle t\rangle^{\gamma_1\cup\gamma_3};\ \epsilon}}{\Gamma_3 = \Gamma_2,\ \gamma_3 \geq \gamma_0,\ y : \langle t\rangle^{\gamma_3} \vdash \mathbf{throw}(k_1, (\mathbf{reset0}(\mathbf{throw}(k_2, y)))) : \langle t\rangle^{\gamma_3};\ \epsilon} \quad \overline{\Gamma_2 \vdash \Box : \langle t\rangle^{\gamma_0};\ \epsilon}}{\Gamma_2 = \Gamma_1,\ k_2 : \langle t\rangle^{\gamma_2} \overset{\langle t\rangle^{\gamma_0}}{\Rightarrow} \langle t\rangle^{\gamma_1},\ k_1 : \langle t\rangle^{\gamma_1} \overset{\epsilon}{\Rightarrow} \langle t\rangle^{\gamma_0} \vdash \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_0};\ \epsilon}\ (\#2)}{\Gamma_1,\ k_2 : \langle t\rangle^{\gamma_2} \overset{\langle t\rangle^{\gamma_0}}{\Rightarrow} \langle t\rangle^{\gamma_1} \vdash \mathbf{shift0}\ k_1 \to \cdots : \langle t\rangle^{\gamma_1};\ \langle t\rangle^{\gamma_0}}}{\Gamma_1 = \gamma_2 \geq \gamma_1,\ x_2 : \langle t\rangle^{\gamma_2},\ \gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \mathbf{shift0}\ k_2 \to \mathbf{shift0}\ k_1 \to \cdots : \langle t\rangle^{\gamma_2};\ \langle t\rangle^{\gamma_1}, \langle t\rangle^{\gamma_0}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_1};\ \langle t\rangle^{\gamma_1}, \langle t\rangle^{\gamma_0}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \mathbf{reset0}\ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_1};\ \langle t\rangle^{\gamma_0}}}{\vdash \underline{\mathbf{let}}\ x_1 = e_1\ \underline{\mathbf{in}}\ \mathbf{reset0}\ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_0};\ \langle t\rangle^{\gamma_0}}}{\vdash e' : \langle t\rangle^{\gamma_0};\ \epsilon}
$$

Figure 15. Example of Type Derivation (Multi-layer let-insertion)

effects, then e is either a value or there exists a term e' such that $e \rightsquigarrow e'$.

The proof is straightforward and so omitted.

By subject reduction and progress, we conclude that our calculus is type sound.

7 Type Inference

Our type system is admittedly very complicated to be used as a type system for programming languages without type inference. To remedy this problem, we have developed a type inference algorithm for our type system under the following assumption: we do not allow implicit effect subtyping.

Let us explain it. Suppose $\Gamma \vdash e : \tau;\ \cdot$ is derivable, namely, e is a pure term with no free effects (which intuitively means that all shift0's in e are surrounded by some reset0). Then we want to use e in an effectful context, too, namely, we expect to have $\Gamma \vdash e : \tau;\ \sigma$ for an arbitrary σ. This is what we call (implicit) effect subtyping. This is semantically admissible, and in fact it is allowed in the type system by Materzok and Biernaki.

However, we found that if the length of the effect part σ can silently change by implicit effect-subtyping, our type inference algorithm would have a trouble in performing unification for the effect part. At present, we do not know if there is a decision procedure for this unification problem, and we have excluded the implicit effect-subtyping, and programmers are asked to explicitly annotate the term with this subtyping, namely, when we the length of the effect part increases.

Under this restriction, type inference for our type system boils down to a satisfiability problem of a boolean combination of inequalities for classifiers. Since the semi-lattice of classifiers has finitely many elements, the satisfiability problem is decidable. The final answer of a successful type inference will be a type with a constraint represented by boolean combination of inequalities for classifiers.

8 Conclusion

We have proposed a core calculus of two-stage programming language in the MetaML style which has control operators shfit0 and reset0 to express sophisticated manipulation of generated code, while keeping the static safety guarantee of the code, Our work is based on environment classifiers [17] and refined environment classifiers [12]. We have extended them to include a semi-lattice structure to the set of classifiers to reflect the behavior of control operators. As far as we know, this is the first study to use shift0 and reset0 [13] in the area of program generation. They allow multi-layer let-insertion in a modular and compact way, without having to introduce more involved control operators such as multi-prompt control operators.

We briefly compare our work to other approaches to representing let insertion in a more general context.

The first approach is to use a CPS translation to eliminate control operators. This approach works if we are only concerned with efficiency and can ignore well typedness and well scopedness, since by CPS translating code generators, we may possibly obtain open code, hence guaranteeing well scopedness in this approach would be much more difficult than our case.

The second one is to do let-insertion automatically such as Keiselyov's 'genlet' primitive discussed in Section 1 of the present paper. Let-insertion was originally studied in partial evaluation, where let-insertion is done automatically. The pros and cons of this approach in the context of staging is mostly the same as those of 'genlet'; if we insert let aggressively (beyond conditional statements), we may unexpectedly change the semantics of the program, and if we insert let conservatively (not beyond conditional statements), we would obtain less efficient code than expected. We think that, in multi-stage programming, programmers want to have a full control over the strategy of let-insertion, and we want to provide a language where various control strategies are expressible in a single framework.

Scala LMS [16] also takes an automation approach for let-insertion, and has made a great success in providing high-performance code with very small burden for human intervention for staging. Although it shares the spirit (abstraction without guilt or regret) with the MetaML-style languages, it has the same pros and cons as the partial-evaluation approach to let-insertion.

Scope graphs [15] are an expressive formalization for specifying binding structures in programming languages. It is a general framework and provides theoretical basis and useful algorithms for name resolution facility for various languages, hence it is interesting to see if our language may be expressible using scope graphs.

Finally, we mention several future works. The first one is a more thorough investigation of type inference algorithm in particular effect subtyping, as it is unknown to us if the type system without restriction has a decidable type inference algorithm or not. We also want to study how the result of type inference (which ought to contain constraints on classifiers) may be shown in an intelligible way. Also our language should be extended to cover let-polymorphism, which would need another investigation on type inference algorithm. Another direction of study is the relationship with the 'genlet' primitive. Clearly 'genlet' is easier to use for programmers than ours but we think that ours may be useful in several cases, so combining them in a single language is an interesting topic. Finally, we should test if our language is usable for a large, practical application of code generation, and if our type system is effective in the sense that all or most typical errors in code generators may be detected by type inference.

Acknowledgments

We would like to thank Oleg Kiselyov for discussion on genlet, the attendees of the IFIP WG 2.11 meeting at Koblenz for suggestions. We deeply thank anonymous reviewers for careful reading and constructive comments. The second author is supported in part by JSPS Kakenhi No. 15K12007.

References

[1] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2004. ML-Like Inference for Classifiers. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings.* 79–93. https://doi.org/10.1007/978-3-540-24725-8_7

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

[3] R. Davies. 1996. A Temporal-logic Approach to Binding-time Analysis. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96).* IEEE Computer Society, Washington, DC, USA, 184–. http://dl.acm.org/citation.cfm?id=788018.788825

[4] 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

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

[6] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2015. Combinators for impure yet hygienic code generation. *Sci. Comput. Program.* 112 (2015), 120–144. https://doi.org/10.1016/j.scico.2015.08.007

[7] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2015. Combinators for impure yet hygienic code generation. *Sci. Comput. Program.* 112 (2015), 120–144. https://doi.org/10.1016/j.scico.2015.08.007

[8] 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 (PEPM '09).* ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1480945.1480962

[9] Oleg Kiselyov. 2014. The design and implementation of BER MetaOCaml. In *International Symposium on Functional and Logic Programming.* Springer, 86–102.

[10] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings.* 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[11] Oleg Kiselyov. 2017. Let-insertion as a primitive. (2017). http://okmij.org/ftp/ML/MetaOCaml.html#genlet

[12] 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

[13] Marek Materzok and Dariusz Biernacki. 2011. Subtyping Delimited Continuations. *ICFP 2011* 46, 9 (Sept. 2011), 81–93. https://doi.org/10.1145/2034574.2034786

[14] Marek Materzok and Dariusz Biernacki. 2012. A Dynamic Interpretation of the CPS Hierarchy. In *APLAS 2012*, Ranjit Jhala and Atsushi Igarashi (Eds.). Lecture Notes in Computer Science, Vol. 7705. Springer Berlin Heidelberg, 296–311. https://doi.org/10.1007/978-3-642-35182-2_21

[15] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[16] Tiark Rompf. 2016. Lightweight modular staging (LMS): generate all the things! (keynote). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016.* 1. https://doi.org/10.1145/2993236.2993237

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

[18] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0