Polymorphic Multi-Stage Language with Control Effects

Yuichiro Kokaji and Yukiyoshi Kameyama

University of Tsukuba, Japan
kokaji@logic.cs.tsukuba.ac.jp,kameyama@acm.org

Abstract. Multi-stage programming (MSP) is a means for run-time code generation, and has been found promising in various fields including numerical computation and domain specific languages. An important problem in designing MSP languages is the dilemma of safety and expressivity; many foundational calculi have been proposed and proven to be type safe, yet, they are not expressive enough. Taha's MetaOCaml provides us a very expressive tool for MSP, yet, the corresponding theory covers its purely functional subset only.
In this paper, we propose a polymorphic multi-stage calculus with delimited-control operators. Kameyama, Kiselyov, and Shan proposed a multi-stage calculus with computation effects, but their calculus lacks polymorphism. In the presence of control effects, polymorphism in types is indispensable as all pure functions are polymorphic over answer types, and in MSP languages, polymorphism in stages is indispensable to write custom generators as library functions. We show that the proposed calculus satisfies type soundness and type inference. The former is the key to guarantee the absence of scope extrusion - open codes are never generated or executed. The latter is important in the ML-like programming languages. Following Calcagno, Moggi and Taha's work, we propose a Hindley-Milner style type inference algorithm to obtain principal types for given expressions (if they exist).

1 Introduction

Writing a code generator as a metaprogram is a vital means to achieve efficiency and maintainability simultaneously. Typed multi-stage (multi-level) programming languages help us write code generators easily and intuitively. The merit of typed multi-stage calculus over its untyped cousin, the quasiquote and unquote mechanism in Scheme, is the static assurance of type soundness: it subsumes not only type safety of code generators, but also that of generated codes, which in turn subsumes the absence of *scope extrusion*: a closed code generator never generates open codes (codes with free variables).

Many researchers have addressed the problem of assuring type soundness for multi-stage calculi; foundational calculi based on modal logic include $\lambda^{\Box}$ by Davies and Pfenning [5], $\lambda^{\bigcirc}$ by Davies [4], and $\lambda^{\bigcirc\Box}$ by Yuse and Igarashi[17]. More expressive calculi have been proposed such as λ^{α} by Taha and Nielsen [13], λ^{i} by Calcagno, Moggi and Taha [2], λ^{sim}_{open} by Kim, Yi and Calcagno [8], and the calculus by Tsukada and Igarashi [15].

Our goal is to extend the applicability of multi-stage programing so that one can write efficient code generators naturally, while keeping static type soundness. This is a challenging goal as efficient code generation (such as let-insertion) often needs impure

(effectful) operations, while existing theories guarantee type soundness for purely functional subcalculi only. A recent hot topic is to add computational effects into multi-stage languages [7, 10, 16]. All of them remain monomorphic setting, though.

In this paper, we introduce ML-like polymorphism into multi-stage languages with computational effects, and in particular, we propose a polymorphic multi-stage calculus with delimited-control operators which satisfies type soundness. We think polymorphism is necessary in this kind of calculi by the following reasons:

- Many useful combinators for code generation are polymorphic functions[1], and, therefore, we need polymorphism to build a useful library for code generation.
- In the presence of computational effects, a type system of MSP calculi necessarily becomes so called a type-and-effect system. Then, all the pure functions (without effects) in existing libraries should be polymorphic over effects. In the case of delimited-control operators, the effects are expressed as the answer types, and, therefore, polymorphism in effects boils down to polymorphism in (answer) types.
- In MSP calculi based on Taha and Nielsen's λ^{α} or Calcagno et al.'s λ^{i}, polymorphism in environment classifiers[2] is necessary to write code generators as libraries. The staged power function is the simplest example for MSP, which already needs polymorphism in classifiers, if written as a library function (that is, not inlined).

Introducing ML-like let-polymorphism is not as trivial as one might expect. The value restriction used in ML families is too restrictive, as we want to generate polymorphic functions as the result of code generation. Other syntactic conditions do not seem suitable, either. Our solution for this problem is to revisit the semantic notion of purity, proposed by Asai and Kameyama [1] for the unstaged polymorphic calculus with delimited-control operators. A term is called pure if it does not have computational effects observable from outside. Asai and Kameyama have shown that a pure term can be made polymorphic. Following them, we allow polymorphism only for pure terms in this paper. Surprisingly, this simple idea works: it rules out all dangerous terms, while we retain the expressivity.

The proposed calculus extends Kameyama, Kiselyov, and Shan's calculus in the sense that we add let-polymorphism and the run-construct (for code execution) to their calculus. We prove type soundness of our calculus under the purity restriction, which implies that open codes are never generated or executed. We also show Hindley-Milner's style type inference algorithm for our calculus, which gives principal types if they exist.

The rest of this paper is organized as follows: Section 2 shows several example programs using multi-stage calculi and control operators, which need polymorphism. Section 3 explains the key idea of introducing polymorphism safely. Section 4 introduces our calculus $\lambda_{let}^{DC}$and operational semantics, and Section 5 introduces its type system. Then we show several useful properties such as type soundness in Section 6 and the existence of principal types in Section 7. Section 8 states concluding remarks.

[1] We will see an example of code generation combinators in Section 2.

[2] Environment classifiers are identifiers for stages, first introduced by Taha and Nielsen [13].

2 Preliminaries

This section is an example-based introduction to MSP and delimited-control operators. A comprehensive introduction to this subject may be found in the literature [11, 12, 7]. We use MetaOCaml to write concrete programs[3] in this section. It has three constructs for code generation: brackets, escape, and run. We do not treat CSP (cross-stage persistence) in this paper.

Staged Power Function. The first, canonical example of MSP is the staged version of the power function:

```
let rec s_power n x =
  if n = 1 then x
  else < ~x * ~(s_power (n-1) x)>
```

The expression <*e*> (bracket expression) represents a code which is not executed at the present stage, but executed at the future (next) stage. We can splice a code fragment into another code by an escape expression $\sim e$. In the expression <$\sim$x$*\cdots$>, the subexpression x is executed, and its value is spliced in this code. For instance, if we evaluate `(fun x-> <~x * 2>) <3 + 4>`, we get `<(3 + 4) * 2>`. For those familiar with Scheme macros, brackets and escape, resp, correspond to quasiquote and backquote (unquote), resp.

By executing the expression `<fun x -> ~(s_power 5 <x>)>`, we get

```
<fun x_1 -> (x_1*(x_1*(x_1*(x_1*x_1))))
```

as its value. Note that the bound variable x has been renamed to `x_1` during the computation, which means that variables in codes are lexically bound unlike the template mechanisms in C++ and Haskell. We can run the resulting code internally by the run construct (`!`). The computation of the expression:

```
let power5 = ! <fun x -> ~(s_power 5 <x>)>
```

yields a function equivalent to

```
fun x_1 -> (x_1*(x_1*(x_1*(x_1*x_1))))
```

which can be used at the present stage, rather than the future stage.

Let us consider the type of `s_power`. Intuitively, it has type `int` $\rightarrow$ `<int>` $\rightarrow$ `<int>` where `<int>` is the type of codes for integer expressions. Taha's λ_i^{let}, the underlying calculus of MetaOCaml, assigns to each future stage an *environment classifier* (classifier for short), in order to distinguish different next stages from each other. Hence `s_power` has type `int` $\rightarrow$ `<int>`$^{\ell} \rightarrow$ `<int>`$^{\ell}$ where ℓ is a classifier. Since `s_power` is polymorphic over stages (it is not specific to any environment classifiers), its type should be polymorphic over classifiers as: $\forall \ell.$(`int` $\rightarrow$ `<int>`$^{\ell} \rightarrow$ `<int>`$^{\ell}$).

Code Generation Combinators. Combinators provide us useful patterns of generating and manipulating code fragments. One of the simplest, but widely used combinators is the following `eta`:

```
let eta f = <fun x -> ~(f <x>)>
in eta (fun y -> <fun z -> z + ~y>)
```

[3] We use slightly simplified notation for multi-stage constructs: we write <*e*> for the MetaOCaml notation `.<`*e*`>.` and we suppress type variables corresponding to environment classifiers.

which, when executed, yields `<fun x -> fun z -> z + x>`. It is easy to see that `eta` should have the polymorphic type: $\forall\ell.\forall\sigma.\forall\tau.(\texttt{<}\sigma\texttt{>}^{\ell} \to \texttt{<}\tau\texttt{>}^{\ell}) \to \texttt{<}\sigma \to \tau\texttt{>}^{\ell}$.

The staged power function may be generalized to an arbitrary binary function:

```
let rec s_iterate n f x =
  if n = 1 then x
  else < ~f ~x ~(s_iterate (n-1) f x)>
in let s_iterate5 =
  eta (fun f -> eta (s_iterate 5 f))
```

which, when executed, returns `<fun f -> fun x -> f x (f x (f x (f x x)))>`. MetaOCaml assigns a monomorphic type to this expression, but we hope to assign a polymorphic type: $\forall\ell.\forall\sigma.\texttt{<}(\sigma \to \sigma \to \sigma) \to \sigma \to \sigma\texttt{>}^{\ell}$ to this expression. Here, polymorphism is necessary both at the present stage and the future stage.

Delimited-Control Operators. Control operators in functional languages are constructs for changing the order of execution, causing computational effects. Typical control operators are catch/throw (Lisp), exception (ML, Java), and call/cc (Scheme, SML/NJ). While call/cc provides an access to *unlimited* continuations, delimited-control operators provide an access to *part* of the current continuations (*delimited* continuations).

The following examples use Danvy and Filinski's delimited-control operators shift and reset [3]:

```
1 + reset (10 + 20)                          yields  31
1 + reset (10 + (shift k -> 20))             yields  21
1 + reset (10 + (shift k -> (k (k 20))))     yields  41
```

We sometimes write shift and reset, resp., as $Sk.e$ and $\{e\}$, resp.[4] `reset` denotes a delimiter, and does nothing if there is no `shift` as shown in the first line. In the second and third lines, `shift` captures the continuation (an evaluation context) up to the nearest `reset` operator, and binds the variable `k` to it. In the example, the captured continuation is `reset (10 + [ ])` where `[ ]` is a hole. It is bound to `k` and may be used later. In the third line, `k (k 20)` evaluates to `reset (10 + (reset (10 + 20)))`.

In the presence of delimited-control operators, the type system should take into account control effects (thus becomes a type-and-effect system). If we execute the following program:

```
let f = fun x -> shift k -> (k 10) + 20
in reset (f 30 >= 40)
```

the continuation captured by `shift` is `reset ([ ] >= 40)`. Then we evaluate `(reset (10 >= 40)) + 20`, which raises a run-time type error.

The computational effect caused by `shift` and `reset` can be described by an *answer type*, the return type of a delimited continuation. In the above program, the `shift` expression expects the answer type to be `int` (since the captured continuation `k` is expected to return an integer), while the `reset` expression provides `bool` as its answer

[4] In the literature, a reset expression is denoted by $\langle e \rangle$. We write $\{e\}$ to avoid conflict with a bracket expression.

type. In the type-and-effect system, a function type takes the form $\sigma \to \sigma'/\beta$, where β represents the answer type. This change of type system makes all pure functions to be polymorphic over effects: for instance, the function `fun x -> x + 10` should have type $\texttt{int} \to \texttt{int}/\beta$ for any type β.

Delimited-control operators such as `shift` and `reset` have been found rather expressive: Filinski [6] proved that they can express any monadic effect, and Kiselyov, Shan and Sabry have shown a concise encoding of dynamic binding and local states in terms of them [9]. Kameyama, Kiselyov and Shan [7] have introduced control operators to type-safe multi-stage calculi, and shown that memoization in code generators is expressible as let-insertion in the calculus. In this paper, we proceed one step further, to introduce polymorphism into their calculus.

3 Introducing Polymorphism Safely

In this section, we investigate the problem of introducing let-polymorphism into a multi-stage calculus with effects, and informally show our ideas to solve the problem.

Value Restriction. As is well known, unrestricted combination of computational effects (such as states and control) and polymorphism leads to type unsoundness, and the value restriction is the standard solution for this problem: for an expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, e_1 must be a (syntactic) value for x to have a polymorphic type in e_2.

Unfortunately the value restriction is too restrictive in multi-stage calculi: in unstaged calculi we only define polymorphic functions, but in staged calculi we want to *generate* (the codes of) polymorphic functions, while value restriction prohibits runtime code generation of such polymorphic functions. Let us consider the following example:

```
let iterate5 = ! s_iterate5
in
  iterate5 (fun x y -> x * y) 3;
  iterate5 (fun x y -> x ^ y) "abc"
```

In this program snippet, the expression `! s_iterate5` is not a value, and, therefore, `iterate5` cannot have a polymorphic type under the value restriction.

The Problem. We need a better criterion as to which terms can be polymorphic. This is not a trivial problem as one might expect. To see the problem, let us consider the program `let y = e in <let x = ~y in e2>`. Then we have different situations depending on the expression `e`:

- if `e` evaluates to a code of a value, say, `<fun z-> z>`, then `x` can be polymorphic.
- if `e` evaluates to a code of an effectful computation (say, `<shift k -> 10>`), then `x` cannot be polymorphic.

In summary, it is not possible to decide the condition by simply looking at the expression e_1 in $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. In other words, the condition must be context sensitive.

Purity Restriction in Unstaged Calculus. Asai and Kameyama [1] have proposed a more liberal condition for let-polymorphism, called the purity restriction, for the (unstaged) calculus with the delimited-control operators `shift` and `reset`.

An expression is *pure* if there are no computational effects that are observable from outside. In the calculus with `shift` and `reset`, the only observable effect (other than termination) is the control effect caused by `shift`, so an expression is pure if all the calls to shift are captured within this expression, and is not pure otherwise. A pure expression is polymorphic in the answer types [14], and we can determine if a given expression is pure or not by tracking its answer type. However, type inference for such a type system is hard, and they replaced it by its conservative approximation as:

Definition 1 (Syntactic Purity [1]). *An (unstaged) expression is syntactically pure if it is a value or a reset expression* $\{e\}$.

The syntactic purity is a stronger (more restrictive) notion than purity, since a pure expression is not necessarily syntactically pure, for instance, *Sk.k* 10. However, there is no loss of expressivity by choosing syntactic purity, since, for any pure e, we can add a superfluous reset as $\{e\}$ while preserving typability and operational behavior. Asai and Kameyama have proven type soundness as well as other desirable properties for their calculus under the syntactic purity restriction.

Purity Restriction in Multi-Stage Calculus. We borrow their idea to formulate the notion of (syntactic) purity in multi-stage calculi, and apply it to let-polymorphism. Since a level-0 expression (present stage expression) cannot have level-1 effects (computation effects of future stage), we formulate (semantic) purity as follows:

- A level-0 expression (an expression at the present stage) is pure if and only if it does not have observable computational effects of level-0.
- A level-1 expression (an expression at the future stage) is pure if and only if it does not have observable computational effects of level-0 and level-1.

As in the case of unstaged calculus, this semantic notion of purity is hard to decide, and we replace it by syntactic purity as follows:

- For a level-0 let-expression `let` $x = e_1$ `in` e_2, the expression e_1 must be a syntactic value or in the form $\{e'_1\}$.
- For a level-1 let-expression `let` $x = e_1$ `in` e_2, the expression e_1 must be a syntactic value or in the form $\{\sim\{\texttt{<}e'_1\texttt{>}\}\}$.

The former is the same as syntactic purity in the unstaged calculus. As for the latter, for a level-1 expression e'_1, the expression $\{\sim\{\texttt{<}e'_1\texttt{>}\}\}$ introduces a level-0 reset, and then the outer most reset is of level-1. In summary, this expression has resets of both levels.

Syntactic Purity in Action. When we introduce the syntax of our calculus, we need one more twist. Rather than directly treating the (syntactically) pure expressions in the above forms, we instead use polymorphic let expression `plet` $x = e_1$ `in` e_2, which intuitively means `let` $x = \{e_1\}$ `in` e_2 for level-0, and `let` $x = \{\sim\{\texttt{<}e_1\texttt{>}\}\}$ `in` e_2 for level-1. This change of syntax greatly simplifies our formulation and it is called the *implicit-delimiter* method.

The implicit-delimiter method was also used in the literature for a different purpose; Kameyama, Kiselyov and Shan [7] regarded a level-1 binder as a delimiter for level-0, advocating that future-stage binders delimit present-stage control effects. For instance,

the level-1 expression $\lambda x.e$ is intuitively equivalent to $\lambda x.{\sim}\{\texttt{<}e\texttt{>}\}$ which has a level-0 delimiter (reset).[5]

In fact, we need both techniques in our calculus – one for ensuring syntactic purity and the other for regarding binders as delimiters. We list all the uses of implicit delimiters below.

$$
\begin{array}{lrcl}
\text{(level 0)} & \texttt{run}\, e & \equiv & \texttt{run <}\{{\sim}\{e\}\}\texttt{>} \\
 & \texttt{plet}\, x = e_1\, \texttt{in}\, e_2 & \equiv & \texttt{plet}\, x = \{e_1\}\, \texttt{in}\, e_2 \\
\text{(level-1)} & \lambda x.e & \equiv & \lambda x.{\sim}\{\texttt{<}e\texttt{>}\} \\
 & Sk.e & \equiv & Sk.{\sim}\{\texttt{<}e\texttt{>}\} \\
 & \texttt{plet}\, x = e_1\, \texttt{in}\, e_2 & \equiv & \texttt{plet}\, x = \{{\sim}\{\texttt{<}e_1\texttt{>}\}\}\, \texttt{in}\, {\sim}\{\texttt{<}e_2\texttt{>}\}
\end{array}
$$

Note that one should understand the above equivalences (denoted by $\equiv$) as informal ones. They will help us understand some reduction rules in Section 4 and the type system in Section 5, but they are not formal entities.

Summary and Discussion. We introduce the syntactic approximation of the notion of purity in the multi-stage calculus. The notion of syntactic purity meets all our needs for let-polymorphism: it is liberal so that we can generate the codes of polymorphic functions in run-time. It is safe in the sense that type soundness holds for our calculus. It is easy to decide if a given expression is pure or not.

We believe that disallowing uncaptured calls to shift in polymorphic functions is reasonable and our implicit-delimiter approach relies on this assumption. In our experience, polymorphism and computational effects in MSP languages are completely separated, and, therefore, our purity restriction is not problematic. However, this is not at all a final word, and a further study is left for future work.

4 The Calculus

This section introduces the polymorphic multi-stage calculus λ^{DC}_{let}, which is based on λ^{i} by Calcagno et al., and $\lambda^{\oslash}_{1}$ by Kameyama et al. The former has polymorphism, more than two levels, CSP, but no control operators. The latter has control operators but no polymorphism, no `run` constructs and no CSP, and is restricted to two levels. Our calculus λ^{DC}_{let} has control operators, polymorphism, `run` constructs, but currently does not have CSP, and restricted to two levels. The principles of our design are simplicity and essence: the combination of control operators, polymorphism, and `run` constructs are the usual sources of type unsoundness, thus leading to scope extrusion, while adding more than two stages and CSP seems orthogonal to these problems. It is desirable to have all these features, but in this paper, to avoid clutter, we present a minimal calculus which exposes the subtle problems in the design of multi-stage calculi. Extension to more expressive calculi are left for future work.

We restrict the computational effects to those caused by `shift` and `reset`, and their answer types be invariant through the computation (no answer-type modification). Danvy's type-safe `printf` is a typical example which needs the effect of answer-type

[5] We inserted brackets and escape to make the reset be of level 1.

$$e^0 ::= v^0 \mid e^0 e^0 \mid e^0 + e^0 \mid \texttt{plet}\ x = e^0\ \texttt{in}\ e^0 \mid \{e^0\} \mid Sk.e^0 \mid \texttt{<}e^1\texttt{>} \mid \texttt{run}\ e^0$$
$$e^1 ::= v^1 \mid e^1 e^1 \mid e^1 + e^1 \mid \texttt{plet}\ x = e^1\ \texttt{in}\ e^1 \mid \{e^1\} \mid Sk.e^1 \mid {\sim}e^0$$
$$v^0 ::= x \mid i \mid \lambda x.e^0 \mid \texttt{<}v^1\texttt{>}$$
$$v^1 ::= x \mid i \mid \lambda x.v^1 \mid v^1 v^1 \mid v^1 + v^1 \mid \texttt{plet}\ x = v^1\ \texttt{in}\ v^1 \mid \{v^1\} \mid Sk.v^1$$

Fig. 1. Syntax of Expressions

modification if written in direct style, so it is not typable in our calculus. However, this restriction is only for presentation; we can develop the calculus with answer-type modification, although it doubles the number of answer types in the judgment. We believe that our calculus provides a useful information on how to introduce polymorphism into multi-stage calculus with computational effects.

Syntax. We define the syntax of λ^{DC}_{let}. We assume to have an infinite number of environment classifiers (or classifiers) $\ell, \ell_1, \ell_2, \cdots$. They are abstract entities; variables for classifiers are quantified by $\forall$, but there are no constants for classifiers. The stage-level L is either 0 (for the present stage), or a single classifier ℓ (for the next, or future stage). In general a level is a finite sequence of classifiers, but we restrict the number of levels to two, so the maximum length of levels is 1. When the names of classifiers do not matter, all stage-levels ℓ_i are simply called "level 1".

Fig. 1 defines the type-free expressions where e^n and v^n, resp., are a level-n expression and a level-n value, resp., for $n = 0, 1$.

A level-0 expression e^0 is either a variable x, an integer literal i, λ-abstraction $\lambda x.e^0$, addition $e^0 + e^0$, a polymorphic let expression $\texttt{plet}\ x = e^0\ \texttt{in}\ e^0$, a reset expression $\{e^0\}$, a shift expression $Sk.e^0$, a bracket expression $\texttt{<}e^1\texttt{>}$, or a run expression $\texttt{run}\ e^0$. Note that, a level-1 expression e^1 should come inside a bracket expression.

A level-1 expression e^1 contains an escape expression ${\sim}e^0$, but since the maximum level is one, there are no expressions like $\texttt{<}e^2\texttt{>}$ or $\texttt{run}\ e^1$.

A level-0 value v^0 is standard except that a bracket expression $\texttt{<}v^1\texttt{>}$ constitutes a code value. Note that we will introduce call-by-value operational semantics. The definition of a level-1 value v^1 contains all kinds of expressions except an escape expression.

The variable x in $\lambda x.e$ and $\texttt{plet}\ x = e'\ \texttt{in}\ e$, k in $Sk.e$ are bound in each e. We identify α-equivalent expressions as usual, and $\texttt{FV}(e)$ denotes the set of free variables in e. Given an expression e, a variable x and a value v of the same level as x, $e[v/x]$ denotes the result of substitution of v for x in e.

Operational Semantics. We define the call-by-value operational semantics. Fig. 2 defines evaluation contexts of various levels. E^{ij} denotes an evaluation context such that its hole (denoted by $\bullet$) will be filled by a level-j expression, and then the whole context will become a level-i expression. An interesting one is $E^{01}[\lambda x.\bullet]$, which means that we evaluate under lambda abstraction.

We also define a pure evaluation context F^{0j} for $j = 0, 1$. Intuitively, this context does not have resets which enclose the hole. In our calculus, certain expressions have implicit resets, and they cannot constitute pure evaluation contexts.

$$
\begin{aligned}
E^{00} ::= {} & \bullet \mid E^{00}[\bullet e^0] \mid E^{00}[v^0 \bullet] \mid E^{00}[\bullet + e^0] \mid E^{00}[v^0 + \bullet] \\
& \mid E^{00}[\mathtt{plet}\ x = \bullet\ \mathtt{in}\ e^0] \mid E^{00}[\{\bullet\}] \mid E^{00}[\mathtt{run}\ \bullet] \mid E^{01}[\sim\bullet] \\
E^{01} ::= {} & E^{01}[\bullet e^1] \mid E^{01}[v^1 \bullet] \mid E^{01}[\bullet + e^1] \mid E^{01}[v^1 + \bullet] \mid E^{01}[\lambda x.\bullet] \\
& \mid E^{01}[\mathtt{plet}\ x = \bullet\ \mathtt{in}\ e^1] \mid E^{01}[\mathtt{plet}\ x = v^1\ \mathtt{in}\ \bullet] \\
& \mid E^{01}[Sk.\bullet] \mid E^{01}[\{\bullet\}] \mid E^{00}[<\bullet>] \mid E^{01}[\mathtt{run}\ \bullet] \\
F^{00} ::= {} & \bullet \mid F^{00}[\bullet e^0] \mid F^{00}[v^0 \bullet] \mid F^{00}[\bullet + e^0] \mid F^{00}[v^0 + \bullet] \mid F^{01}[\sim\bullet] \\
F^{01} ::= {} & F^{01}[\bullet e^1] \mid F^{01}[v^1 \bullet] \mid F^{01}[\bullet + e^1] \mid F^{01}[v^1 + \bullet] \mid F^{00}[<\bullet>]
\end{aligned}
$$

Fig. 2. Evaluation Contexts

$$
\begin{aligned}
E^{00}[i+j] &\rightsquigarrow E^{00}[m] \ \text{ if } \ i+j=m & \qquad E^{00}[\{v^0\}] &\rightsquigarrow E^{00}[v^0] \\
E^{00}[(\lambda x.e^0)v^0] &\rightsquigarrow E^{00}[e^0[v^0/x]] & \qquad E^{01}[\sim<v^1>] &\rightsquigarrow E^{01}[v^1] \\
E^{00}[\mathtt{plet}\ x = v^0\ \mathtt{in}\ e^0] &\rightsquigarrow E^{00}[e^0[v^0/x]] & \qquad E^{00}[\mathtt{run}\ <v^1>] &\rightsquigarrow E^{00}[v^1] \\
E^{00}[\{F^{00}[Sk.e^0]\}] &\rightsquigarrow E^{00}[\{e^0[\lambda x.\{F^{00}[x]\}/k]\}]
\end{aligned}
$$

Fig. 3. Reduction Rules

Fig. 3 gives the reduction rules in the evaluation-context style.

The first three rules are integer addition, β reduction in call-by-value, and let-reduction as usual. The next two rules are the ones for control operators. shift captures the continuation delimited by the nearest delimiter. In the rule, F^{00} is an evaluation context which does not have resets around the hole, which means that the reset displayed in the rule is the nearest one. After capturing the delimited context $\{F^{00}\}$, we convert it to a functional form $\lambda x.\{F^{00}[x]\}$, and bind k to it, and continue the evaluation. For the next rule, if the body of a reset expression is a level-0 value, the delimiter is simply discarded.

The last two rules are the reduction rules for multi-stage constructs. In the second last rule, the evaluation context E^{01} signifies that the hole in E^{01} is of level-1, which means that there are brackets enclosing the hole. Hence the subexpression $\sim<v^1>$ appears in a code, and thus we are splicing the code v^1 into the code. Then it is easy to understand the reduction rule. The last rule defines the code execution. If the body of the run expression is $<v^1>$, then we extract the content v^1 of the code expression, and start evaluating v^1. In the right-hand side of this rule, a level-1 value v^1 is plugged in to the level-0 hole in E^{00}.

Reductions for Implicit Delimiter. One may notice that reduction rules Fig. 3 are too weak. In fact, there are closed, non-value expressions that may not be reduced by any reduction rules. For instance, $\{<\lambda x.\sim(Sk.e)>\}$ gets stuck, since the obvious candidate for reduction is $E^{00}[\{F^{00}[Sk.e^0]\}] \rightsquigarrow \cdots$, but the definition of F^{00} does not allow level-1 abstraction $\lambda x.e$. Our calculus rules out some of such kinds of expressions, but not all of

Auxiliary definition for pure evaluation contexts:

$$F^{10} ::= F^{10}[\bullet e^0] \mid F^{10}[v^0 \bullet] \mid F^{10}[\bullet + e^0] \mid F^{10}[v^0 + \bullet] \mid F^{11}[\sim \bullet]$$
$$F^{11} ::= \bullet \mid F^{11}[\bullet e^1] \mid F^{11}[v^1 \bullet] \mid F^{11}[\bullet + e^1] \mid F^{11}[v^1 + \bullet] \mid F^{10}[\texttt{<} \bullet \texttt{>}]$$

Additional reduction for level-0 implicit resets:

$$G[F^{00}[Sk.e^0]] \rightsquigarrow G[\sim\{e^0[\lambda x.\{F^{00}[x]\}/k]\}] \qquad \text{where } G ::= E^{00}[\texttt{plet } x = \bullet \texttt{ in } e^0]$$
$$G'[F^{00}[Sk.e^0]] \rightsquigarrow G'[\{e^0[\lambda x.\{\texttt{<}\{\sim F^{00}[x]\}\texttt{>}\}/k]\}] \qquad \text{where } G' ::= E^{00}[\texttt{run } \bullet]$$

Additional reduction for level-1 implicit resets:

$$H[F^{10}[Sk.e^0]] \rightsquigarrow H[\sim\{e^0[\lambda x.\{\texttt{<}F^{10}[x]\texttt{>}\}/k]\}]$$
$$\text{where } H ::= E^{01}[\lambda x.\bullet] \mid E^{01}[Sk'.\bullet] \mid E^{01}[\texttt{plet } x = \bullet \texttt{ in } e^1] \mid E^{01}[\texttt{plet } x = e^1 \texttt{ in } \bullet]$$

Fig. 4. Reduction rules for implicit resets

them. Some expressions in the above form typecheck in our type system, and thus, we need additional reduction rules for those safe patterns, to take into account the implicit delimiters.

Fig. 4 gives the reduction rules corresponding to implicit resets.

Although the reduction rules look complicated, they are in fact simply derived from the informal reading for implicit resets, stated earlier.

5 Type System

In this section, we define a polymorphic type system for the calculus λ^{DC}_{let}. We first define types:

$$\sigma, \tau, \alpha, \beta ::= t \mid \texttt{int} \mid \sigma \to \tau/\beta \mid \texttt{<}\sigma/\beta\texttt{>}^{\ell} \qquad \text{monomorphic type}$$
$$T ::= \sigma \mid \forall t.T \mid \forall \ell.T \qquad \text{polymorphic type}$$

where t is a type variable, and $\texttt{int}$ is the type for integers. The type $\sigma \to \tau/\beta$ is the function type with effects, which are determined by the answer type β. The type $\texttt{<}\sigma/\beta\texttt{>}^{\ell}$ is the type for codes of level ℓ where σ is the type of the code, and β is a level-1 answer type.

The polymorphic type T is a monomorphic type with universal quantification. Following Calcagno et al. [2], we have two kinds of quantification: $\forall t.T$ represents universal quantification over types, and $\forall \ell.T$ universal quantification over environment classifiers. We sometimes write $\forall \overline{t}.\forall \overline{\ell}.\sigma$ for the type σ quantified over sequences of type variables $t_1, \cdots, t_n$ and environment classifiers $\ell_1, \cdots, \ell_m$.

For a type T, $\texttt{FC}(T)$ and $\texttt{FTV}(T)$, resp., are the set of free classifiers in T, and the set of free type variables in T, resp. In the following, we sometimes write $\texttt{FV}(\Gamma)$ and so on, which has obvious meaning.

A general form of a judgment is $\Gamma \vdash^L e : \sigma \;\; ; \beta_0 \;\; ; \beta_1$ where the type context Γ is a (possibly empty) finite sequence of the form $(x : T)^L$ where T is a polymorphic type

and L is a level. The level L in $(x:T)^L$ means that the variable x can be used in level L. The above judgment means that, under the type context Γ, e is a level-L expression of type σ with the level-0 answer type β_0 and the level-1 answer type β_1. When $L=0$, the level-1 answer type β_1 is not significant (a level-0 expression cannot contain level-1 effects), and therefore we often write $-$ for β_1.

Typing rules of λ^{DC}_{let} are defined as follows.

$$\frac{(i \text{ is an integer constant})}{\Gamma \vdash^L i : \mathtt{int}\ ;\beta_0\ ;\beta_1}\ \text{int} \qquad \frac{(\tau\ \leq T)}{\Gamma,(x:T)^L \vdash^L x:\tau\ ;\beta_0\ ;\beta_1}\ \text{var}$$

$$\frac{\Gamma \vdash^L e_1 : \mathtt{int}\ ;\beta_0\ ;\beta_1 \quad \Gamma \vdash^L e_2 : \mathtt{int}\ ;\beta_0\ ;\beta_1}{\Gamma \vdash^L e_1 + e_2 : \mathtt{int}\ ;\beta_0\ ;\beta_1}\ \text{plus}$$

$$\frac{\Gamma,(x:\sigma)^0 \vdash^0 e:\tau\ ;\beta_0\ ;-}{\Gamma \vdash^0 \lambda x.e : \sigma \to \tau/\beta_0\ ;\alpha_0\ ;-}\ \lambda^0 \qquad \frac{\Gamma,(x:\sigma)^\ell \vdash^\ell e:\tau\ ;<\tau/\beta_1>\ ;\beta_1}{\Gamma \vdash^\ell \lambda x.e : \sigma \to \tau/\beta_1\ ;\alpha_0\ ;\alpha_1}\ \lambda^1$$

$$\frac{\Gamma \vdash^0 e_1 : \sigma \to \tau/\alpha_0\ ;\alpha_0\ ;- \quad \Gamma \vdash^0 e_2 : \sigma\ ;\alpha_0\ ;-}{\Gamma \vdash^0 e_1 e_2 : \tau\ ;\alpha_0\ ;-}\ \text{app}^0 \qquad \frac{\Gamma \vdash^\ell e_1 : \sigma \to \tau/\alpha_1\ ;\alpha_0\ ;\alpha_1 \quad \Gamma \vdash^\ell e_2 : \sigma\ ;\alpha_0\ ;\alpha_1}{\Gamma \vdash^\ell e_1 e_2 : \tau\ ;\alpha_0\ ;\alpha_1}\ \text{app}^1$$

$$\frac{\Gamma \vdash^0 e_1 : \sigma\ ;\sigma\ ;- \quad \bar{t} \subseteq \mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma), \bar{\ell} \subseteq \mathtt{FC}(\sigma) - \mathtt{FC}(\Gamma) \quad \Gamma,(x:\forall \bar{t}.\forall \bar{\ell}.\sigma)^0 \vdash^0 e_2 : \tau\ ;\beta_0\ ;-}{\Gamma \vdash^0 \mathtt{plet}\ x = e_1\ \mathtt{in}\ e_2 : \tau\ ;\beta_0\ ;-}\ \text{let}^0$$

$$\frac{\Gamma \vdash^\ell e_1 : \sigma\ ;<\sigma/\sigma>^\ell\ ;\sigma \quad \bar{t} \subseteq \mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma) \quad \Gamma,(x:\forall \bar{t}.\sigma)^\ell \vdash^\ell e_2 : \tau\ ;<\tau/\beta_1>^\ell\ ;\beta_1}{\Gamma \vdash^\ell \mathtt{plet}\ x = e_1\ \mathtt{in}\ e_2 : \tau\ ;<\tau/\beta_1>^\ell\ ;\beta_1}\ \text{let}^1$$

$$\frac{\Gamma,(k:\forall t.(\sigma \to \beta_0/t))^0 \vdash^0 e:\beta_0\ ;\beta_0\ ;-}{\Gamma \vdash^0 Sk.e : \sigma\ ;\beta_0\ ;-}\ \text{shift}^0 \qquad \frac{\Gamma \vdash^0 e:\beta_0\ ;\beta_0\ ;-}{\Gamma \vdash^0 \{e\} : \beta_0\ ;\alpha_0\ ;-}\ \text{reset}^0$$

$$\frac{\Gamma,(k:\forall t.(\sigma \to \beta_1/t))^\ell \vdash^\ell e:\beta_1\ ;<\beta_1/\beta_1>^\ell\ ;\beta_1}{\Gamma \vdash^\ell Sk.e : \sigma\ ;\alpha_0\ ;\beta_1}\ \text{shift}^1 \qquad \frac{\Gamma \vdash^\ell e:\beta_1\ ;\beta_0\ ;\beta_1}{\Gamma \vdash^\ell \{e\} : \beta_1\ ;\beta_0\ ;\alpha_1}\ \text{reset}^1$$

$$\frac{\Gamma \vdash^\ell e:\sigma\ ;\beta_0\ ;\beta_1}{\Gamma \vdash^0 <e> : <\sigma/\beta_1>^\ell\ ;\beta_0\ ;-}\ \text{brackets}^0 \qquad \frac{\Gamma \vdash^0 e : <\sigma/\beta_1>^\ell\ ;\beta_0\ ;-}{\Gamma \vdash^\ell \sim e : \sigma\ ;\beta_0\ ;\beta_1}\ \text{escape}^1$$

$$\frac{\Gamma \vdash^0 e : <\sigma/\sigma>^\ell\ ;<\sigma/\sigma>^\ell\ ;- \quad (\ell \notin \mathtt{FC}(\Gamma,\sigma))}{\Gamma \vdash^0 \mathtt{run}\ e : \sigma\ ;\alpha_0\ ;-}\ \text{run}^0$$

Let us explain the typing rules briefly.

The var rule is the standard one in polymorphic type systems where $\tau \leq T$ means that τ is an instance of polymorphic type T. More precisely, if $T = \forall \overline{t}.\forall \overline{\ell}.\sigma$ for a monomorphic type σ, then $\tau = \sigma[\overline{\alpha}/\overline{t}][\overline{\ell'}/\overline{\ell}]$ for some $\alpha_1, \cdots, \alpha_n, \ell'_1, \cdots, \ell'_k$.

The λ^0 rule is also standard except that it retrieves the level-0 answer type β_0 into the function type $\sigma \to \tau/\beta_0$. Since a function does not have computational effects, its level-0 answer type can be an arbitrary type α_0. The λ^1 rule is the key to avoid scope extrusion as studied by Kameyama, Kiselyov and Shan [7]. As we explained in Section 3, the level-1 expression $\lambda x.e$ has an implicit level-0 reset beneath this lambda, namely, its intuitive meaning is $\lambda x.\sim\{\texttt{<}e\texttt{>}\}$. In order to type this expression, we need to require the level-0 answer type for e be $\texttt{<}\tau/\beta_1\texttt{>}^\ell$. The app rules can be understood easily.

The level-0 polymorphic let expression $\texttt{plet}\ x = e_1\ \texttt{in}\ e_2$ should be understood as $\texttt{let}\ x = \{e_1\}\ \texttt{in}\ e_2$, hence the answer type of e_1 must be the same as the type of e_1 itself. (See also the reset rule below.) The level-1 polymorphic let expression $\texttt{plet}\, x = e_1\ \texttt{in}\ e_2$ is slightly more complex. First, due to the purity restriction for e_1, it is understood as $\texttt{let}\, x = \{\sim\{\texttt{<}e_1\texttt{>}\}\}\ \texttt{in}\ e_2$. But since this level-1 let expression is a binder for e_2, it must not have level-0 effects, and therefore, $\texttt{let}\ x = \{\sim\{\texttt{<}e_1\texttt{>}\}\}\ \texttt{in}\ \sim\{\texttt{<}e_2\texttt{>}\}$ is the final meaning of polymorphic let expression. The type rule reflects this reading.

The shift rules are adaptation from the standard typing rules for them in the literature of delimited continuations. For instance, level-0 shift captures a delimited context whose answer type is β_0. Thus the type of k must be a function type whose return type is β_0. Since the delimited continuation is a pure function (no control effects are involved), it is polymorphic in the answer type, and thus we quantify t in $\sigma \to \beta_0/t$. The shift^1 rule is more complex, as it binds level-1 variables k, and again we have an implicit level-0 reset. (Note the level-0 answer type for e is $\texttt{<}\beta_1/\beta_1\texttt{>}^\ell$.)

The reset rules are also adaptation of the type rule in the literature. For a level-0 expression $\{e\}$, its type must be the same type as that of e and also the level-0 answer type of e. Since $\{e\}$ has no observable control effects, its level-0 answer type can be arbitrary type α_0. The level-1 reset rule can be understood similarly.

The rules for brackets and escapes are the same as those in λ^α and λ^i except that we need to memoize the level-0 effect (β_1) in the code type as $\texttt{<}\sigma/\beta_1\texttt{>}^\ell$. Brackets turn a level-1 expression to a level-0 expression, and escape does the converse.

Finally, the run rule is one of the most interesting ones. The run construct is to execute the code (inside brackets) at the present stage, and thus it is important to ensure it is a closed code. The calculi λ^α and λ^i ensure this closedness condition in terms of the eigen-variable condition for the classifier ℓ: it must not appear at any other place in the judgment for e. If the condition is met, e does not depend on the stage ℓ, so we can run (and compile) it at the present stage. In addition to this condition, we need to rule out, for instance, an expression like `run <shift k -> ..>`. In general we need to ensure level-0 and level-1 purity of e in this rule. Thus we implicitly introduce resets of both levels.

The following figure shows a type derivation for $\lambda x.\sim(Sk.\texttt{<}x+10\texttt{>})$ where $\Gamma = (x : \texttt{int})^\ell, (k : \forall t.\texttt{<int/int>}^\ell \to \texttt{<int/int>}^\ell/t)^0$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash^{\ell} x+10 : \mathtt{int}\ ;\mathtt{<int/int>}^{\ell}\ ;\mathtt{int}}{\Gamma \vdash^{0} \mathtt{<}x+10\mathtt{>} : \mathtt{<int/int>}^{\ell}\ ;\mathtt{<int/int>}^{\ell}\ ;-}}{(x:\mathtt{int})^{\ell} \vdash^{0} Sk.\mathtt{<}x+10\mathtt{>} : \mathtt{<int/int>}^{\ell}\ ;\mathtt{<int/int>}^{\ell}\ ;-}}{(x:\mathtt{int})^{\ell} \vdash^{\ell} \sim(Sk.\mathtt{<}x+10\mathtt{>}) : \mathtt{int}\ ;\mathtt{<int/int>}^{\ell}\ ;\mathtt{int}}}{\vdash^{\ell} \lambda x.\sim(Sk.\mathtt{<}x+10\mathtt{>}) : (\mathtt{int} \to \mathtt{int/int})\ ;\beta_1\ ;\beta_2}$$

6 Type Soundness

In this section we prove type soundness of λ^{DC}_{let}, which consists of the subject reduction property (type preservation) and the progress property. Due to lack of space, we do not give complete proofs, and, instead state important lemmas in this paper.

Lemma 1 (Values do not have effects). *If $\Gamma \vdash^{0} v^{0} : \sigma\ ;\beta_0\ ;-$ is derivable, then $\Gamma \vdash^{0} v^{0} : \sigma\ ;\alpha\ ;-$ is derivable for any α. If $\Gamma \vdash^{\ell} v^{1} : \sigma\ ;\beta_0\ ;\beta_1$ is derivable, then $\Gamma \vdash^{\ell} v^{1} : \sigma\ ;\alpha\ ;\beta_1$ is derivable for any α.*

This lemma can be proven immediately. We then state two lemmas about substitution, which are used in the proof of subject reduction.

Lemma 2 (Substitution for monomorphic variable). *If $\Gamma_1 \vdash^{0} v : \sigma\ ;\alpha_0\ ;-$ and $\Gamma_1, \Gamma_2, (x:\sigma)^{0} \vdash^{L} e : \tau\ ;\beta_0\ ;\beta_1$ are derivable, $\Gamma_1, \Gamma_2 \vdash^{L} e[v/x] : \tau\ ;\beta_0\ ;\beta_1$ is derivable.*

Lemma 3 (Substitution for polymorphic variable). *If $\Gamma_1 \vdash^{0} v : \sigma;\ [\,]$ and $\Gamma_1, \Gamma_2, (x : \forall t.\forall \ell.\sigma)^{0} \vdash^{L} e : \tau;\ ;\beta_0\ ;\beta_1$ are derivable, $t_1, \cdots, t_n \in FTV(\sigma) - FTV(\Gamma)$ and $\ell_1, \cdots, \ell_k \in FC(\sigma) - FC(\Gamma)$, then $\Gamma_1, \Gamma_2 \vdash^{L} e[v/x] : \tau;\ ;\beta_0\ ;\beta_1$ is derivable.*

These lemmas can be proven by structural induction on the second derivation, resp. The next lemma is necessary to prove subject reduction for the case of the run construct. First, we introduce an auxiliary definition $\_ \Downarrow_{\ell}$ for elements of typing contexts, defined as:

$$(x:\sigma)^{0} \Downarrow_{\ell} \stackrel{\text{def}}{=} (x:\sigma)^{0}$$
$$(x:\upsilon)^{\ell} \Downarrow_{\ell} \stackrel{\text{def}}{=} (x:\upsilon)^{0}$$
$$(x:\upsilon)^{\ell'} \Downarrow_{\ell} \stackrel{\text{def}}{=} (x:\upsilon)^{\ell'} \quad \text{if } \ell' \neq \ell$$

The definition extends to $\Gamma \Downarrow_{\ell}$ straightforwardly.

Lemma 4. *Suppose $\Gamma_1, \Gamma_2 \vdash^{\ell} v^{1} : \sigma\ ;\beta_0\ ;\beta_1$ is derivable such that $\ell \notin FC(\Gamma_1, \sigma, \beta_0, \beta_1)$, and Γ_2 consists of the form $(x_i : \tau_i)^{\ell}$ such that $\ell \notin FC(\tau_i)$. Then we can derive $\Gamma_1, \Gamma_2 \Downarrow_{\ell} \vdash^{0} v^{1} : \sigma;\ ;\beta_1\ ;-$.*

This lemma is proven by induction on the derivation of $\Gamma_1, \Gamma_2 \vdash^{\ell} v^{1} : \sigma\ ;\beta_0\ ;\beta_1$. As its corollary, we obtain:

Corollary 1. *Suppose* $\Gamma \vdash^{\ell} v^1 : \sigma \ ;\beta_0 \ ;\beta_1$ *is derivable such that* $\ell \notin FV(\Gamma, \sigma, \beta_0, \beta_1)$*, then we can derive* $\Gamma \vdash^0 v^1 : \sigma \ ;\beta_1 \ ;-$.

Finally, we state the subject reduction property.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash^0 e : \tau \ ;\beta_0 \ ;-$ *is derivable and* $e \rightsquigarrow^* e'$*, then* $\Gamma \vdash^0 e' : \tau \ ;\beta_0 \ ;-$ *is derivable.*

Proof. (very brief sketch) We prove the theorem by induction on the number of reduction steps, and case analysis of reductions.

For the reduction rules of shift (when shift captures a continuation up to the nearest reset), we have the important observation: for any type derivation of a pure context F^{ij} $(i, j = 0, 1)$, there are no (explicit or implicit) level-0 resets and no level-0 binders which enclose the hole. Moreover, there are no binders of the same level. Then by induction on F^{ij}, we can prove that the level-0 answer type does not change through this derivation. By using this fact, we can prove the subject reduction property for this case.

The cases for β and let reductions are handled by Lemmas 2 and 3. Other cases are proven straightforwardly.

The progress property states that a closed well-typed express does not get stuck.

Theorem 2 (Progress). *If* $\vdash^0 e : \tau \ ;\tau \ ;-$ *is derivable, there exists an expression* e_1 *such that* $\{e\} \rightsquigarrow e_1$.

Proof. (sketch) We first prove by induction that, for any typable expression e, it is a value, a reducible expression, or a stuck expression $E^{00}[Sk.e']$.

Then, we can prove that $\{e\}$ is always a redex.

7 Principal Type and Type Inference

Type inference is an important feature for ML-like languages, and it is even more important for our calculus, since we need to keep track of the effects of an expression as type annotation. In this section we briefly mention the type inference algorithm for λ^{DC}_{let}.

Calcagno, Moggi and Taha [2] proposed the calculus λ^i and its polymorphic version λ^i_{let}, which is slightly less expressive than the earlier calculus λ^{α} by Taha and Nielsen, but has principal typing (or principal type for λ^i_{let}). λ^i is suitable as the foundation of multi-stage programming languages, as demonstrated by the success of the MetaOCaml language.

Our calculus λ^{DC}_{let} also has the principal type property, and a sound and complete type inference algorithm similar to the algorithm W.

Given an expression e, a type context Γ and a level L, we say $(\theta, \sigma, \alpha, \beta)$ is a solution for these data if and only if θ is a substitution for type variables and classifiers, σ, α, β are types, and $\Gamma \vdash^L e : \sigma \ ;\alpha \ ;\beta$ is derivable.

Proposition 1 (Principal Type). *Suppose there is a solution for an expression e, a type context* Γ*, and a level L. Then there exists a principal solution* $(\theta_0, \sigma_0, \alpha_0, \beta_0)$ *for them, namely, for any solution* $(\theta_1, \sigma_1, \alpha_1, \beta_1)$ *for the same input, there exists a substitution* ϕ *such that* $\theta_1 = \theta_0\phi$*,* $\sigma_1 = \sigma_0\phi$*,* $\alpha_1 = \alpha_0\phi$ *and* $\beta_1 = \beta_0\phi$.

Here we slightly generalized the statement of the theorem than the standard form so that the inductive proof goes through.

Proof. (sketch) We can construct a Hindley-Milner's style type inference algorithm for λ_{let}^{DC}. This is due to the "implicit reset" approach, since we no longer have to infer the (semantic) purity in the process of type inference.

The only difficulty in constructing the algorithm is the case for the run rule, which has the negative side condition $\ell \notin \mathtt{FV}(\Gamma, \sigma)$. To see how our algorithm handle this case, let us consider the expression `run` e under the context Γ and the level L. We first infer the type of e under Γ and L. Suppose we get $(\theta_0, \sigma_0, \alpha_0, \beta_0)$ as the answer. Then we unify σ_0 with $<\sigma'/\sigma'>^{\ell}$ for a fresh σ' and so on. We check if the classifier ℓ appears in the results or not. The algorithm fails to infer a type if ℓ appears in a wrong place of the results, and continues otherwise. In the latter case, the future process of type inference does not unify ℓ to other classifiers, so the side condition $\ell \notin \mathtt{FV}(\Gamma, \sigma)$ will remain valid.

Hence we can build a W-like type inference algorithm. The proof of soundness and completeness of the algorithm is as standard.

8 Conclusion

We have designed a polymorphic type system for multi-stage calculus with delimited-control operators where polymorphism in types and that in classifiers are expressible. The calculus in this paper extends Kameyama, Kiselyov and Shan's calculus in that (besides polymorphism) we have the run construct. The key idea of integrating these conflicting concepts into one calculus is to relax the value restriction to the syntactic purity restriction, and then introduce the notion of implicit resets. We have proven type soundness and the existence of principal types, both of which are essential to make our language usable.

The success of this combination strongly owes to the local nature of computational effects by shift and reset: we can represent, in particular, mutable variables in terms of shift and reset, but their scope is local to a certain block in a program. By placing a sufficiently many resets (on the borders of polymorphism and run), we have obtained a type-safe polymorphic calculus.

We mention other approaches to design type-safe multi-stage calculi with control effects. Kim, Yi and Calcagno [8] proposed a polymorphic modal type system for Lisp-like languages. Since their calculus has quite different flavors in nature (for instance, α-equivalence is not admissible in their calculus, while it is built in our calculus), it is left for future work to compare these two different lines of works.

Recently Westbrook et al. [16] designed a multi-stage programming language Mint as an extension of Java. To ensure type safety, they introduced the notion of weak separability. Despite the difference of underlying languages, it will be interesting to compare their conditions with ours so that we can build an even more powerful, and type-safe calculus.

Acknowledgments

We would like to thank Kenichi Asai, Atsushi Igarashi, Oleg Kiselyov, Chung-chieh Shan, and Kwangkeun Yi. We also thank anonymous reviewers for constructive comments. The second author is supported in part by JSPS Grant-in-Aid for Scientific Research (B) 21300005.

References

1. K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *Proc. Asian Programming Languages and Systems, LNCS 4807*, pages 239–254, Nov-Dec 2007.
2. Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Ml-like inference for classifiers. In *ESOP*, pages 79–93, 2004.
3. Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.
4. Rowan Davies. A temporal logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996.
5. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.
6. Andrzej Filinski. Representing Monads. In *Proc. 21st Symposium on Principles of Programming Languages*, pages 446–457, 1994.
7. Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: staging with delimited control. In *PEPM*, pages 111–120, 2009.
8. Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL*, pages 257–268, 2006.
9. Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In John H. Reppy and Julia L. Lawall, editors, *ICFP*, pages 26–37. ACM, 2006.
10. Keisuke Sugiura and Yukiyoshi Kameyama. Multi-stage language with control effect and code execution (in Japanese). *Computer Software (Journal of Japan Society of Software Science and Technology)*, 28(1):217–229, 2011.
11. Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.
12. Walid Taha. A Gentle Introduction to Multi-stage Programming, Part II. In *GTTSE*, pages 260–290, 2007.
13. Walid Taha and Michael F. Nielsen. Environment classifiers. In *POPL*, pages 26–37, 2003.
14. Hayo Thielecke. From control effects to typed continuation passing. In *POPL*, pages 139–149, New York, January 2003. ACM Press.
15. Takeshi Tsukada and Atsushi Igarashi. A logical foundation for environment classifiers. *Logical Methods in Computer Science*, 6(4:9):1–43, 2010.
16. Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *PLDI*, pages 400–411, 2010.
17. Yosihiro Yuse and Atsushi Igarashi. A modal type system for multi-level generating extensions with persistent code. In *PPDP*, pages 201–212, 2006.