On Cross-Stage Persistence in Multi-Stage Programming*

Yuichiro Hanada and Atsushi Igarashi

Graduate School of Informatics, Kyoto University

Abstract. We develop yet another typed multi-stage calculus $\lambda^{\triangleright\%}$. It extends Tsukada and Igarashi's $\lambda^{\triangleright}$ with cross-stage persistence and is equipped with all the key features that MetaOCaml-style multi-stage programming supports. It has an arguably simple, substitution-based full-reduction semantics and enjoys basic properties of subject reduction, confluence, and strong normalization. Progress also holds under an alternative semantics that takes staging into account and models program execution. The type system of $\lambda^{\triangleright\%}$ gives a sufficient condition when residual programs can be safely generated, making $\lambda^{\triangleright\%}$ more suitable for writing generating extensions than previous multi-stage calculi.

1 Introduction

Multi-stage programming (MSP) is a programming paradigm in which a programmer can manipulate, generate, and execute code fragments at run time. These features enhance reusability of programs and make optimizations easier by writing program specializers [1]. A number of programming languages that support multi-stage programming have been proposed [2–7], not to mention Lisp and Scheme, and provide different sets of language constructs for MSP.

Among these MSP languages, MetaOCaml provides (hygienic) quasiquotation (called brackets and escape), eval (called run) [3]. Brackets $\langle e \rangle$ are a quotation of expression e to make a code value and escape (written $\tilde{}e$) splices the value of e, which is supposed to be a quotation, into the surrounding quotation. For example, the following MetaOCaml expression[1]

$$\textbf{let}\ a = \langle 1 + 2 \rangle\ \textbf{in}\ \langle \tilde{}a * \tilde{}a \rangle$$

evaluates to $\langle (1 + 2) * (1 + 2) \rangle$. Run (written **run** e here[2]) evaluates the expression inside a given code value, and so

$$\textbf{run}\ (\textbf{let}\ a = \langle 1 + 2 \rangle\ \textbf{in}\ \langle \tilde{}a * \tilde{}a \rangle)$$

yields 9 (without brackets).

---

* Revised on May, 2019.

[1] Actually, $\langle e \rangle$ is written `.<`e`>.` and $\tilde{}e$ is written `.~`e in MetaOCaml.

[2] In MetaOCaml, `.!`e is used for **run** e.

Another interesting feature of MetaOCaml is called cross-stage persistence (CSP), which allows a computed value to be put into brackets: for example, the expression

$$\textbf{let } a = \ 1 + 2 \textbf{ in } \langle a * a \rangle$$

(without escapes on a inside the brackets) is valid in MetaOCaml and yields $\langle 3 * 3 \rangle$. Here, a is bound to the integer value 3 and CSP (implicitly applied to variable references) allows referencing a variable declared outside of the brackets. Note that, as Taha and Sheard discuss [3], CSP is *not* lifting, which converts a value into its syntactic representation (although CSP for basic values can be implemented by lifting). In fact, CSP can be applied to a variable denoting *any value*, including functions, references, or even file descriptors, which do not always have syntactic representations. CSP is a very important feature in practice, because a programmer can freely use library functions inside brackets as in $\langle \mathit{List.map}\ (\lambda x.x + 1)\ [3; 4] \rangle$.

Most type systems for MSP languages aim at ensuring safety of the code generated by multi-stage programs, as well as that of multi-stage programs themselves. A challenging issue was how to prevent **run** from executing open code (namely, code values that contain free variables), while allowing manipulation of open code, which is necessary to generate efficient code. Taha and Nielsen [8] developed a multi-stage calculus λ^{α} with all the features above and proved that its type system guaranteed safety in the above sense. A key idea in the type system of λ^{α} is the introduction of *environment classifiers* (or simply classifiers). Roughly speaking, classifiers statically keep track of information on free variables in code values and prevents code value containing free variables from being **run**. Later, its type system was adapted to ML-style type reconstruction and has become a basis of MetaOCaml [9].

Tsukada and Igarashi [10] proposed another typed MSP calculus $\lambda^{\triangleright}$, whose type system, which uses a classifier-like mechanism, can be regarded as a certain modal logic through the Curry–Howard isomorphism. Although $\lambda^{\triangleright}$ supports only brackets, escape, and **run**, its operational semantics has a more "standard flavor" than that of λ^{α} (and MetaOCaml) in that reduction can be defined in terms of (a few kinds of) substitutions.

In this paper, we present yet another multi-stage calculus $\lambda^{\triangleright\%}$, which is an extension of $\lambda^{\triangleright}$ with CSP and study its properties. We give the semantics of $\lambda^{\triangleright\%}$ in two ways: full nondeterministic reduction, which allows any redex (even inside quotations) to be reduced, and (call-by-value) staged reduction, which is a subrelation of the full reduction and allows only a certain redex at the lowest stage to be reduced. Interestingly, the semantic "delta" over $\lambda^{\triangleright\%}$ is surprisingly small, making proofs from $\lambda^{\triangleright}$ easy to extend to $\lambda^{\triangleright\%}$. Our technical contributions are summarized as follows:

- we give the formal definition of $\lambda^{\triangleright\%}$ with its syntax, type system, full reduction, and staged reduction;
- for the full reduction, we prove subject reduction, strong normalization and confluence; and
- for the staged reduction, we prove progress and a property called Type-Safe Residualization, which means that a well-typed program of code type yields a code value whose body is also a well-typed and serializable program.

We also discuss relationship between CSP and program residualization and point out a problem that, although MetaOCaml enjoys a variant of Type-Safe Residualization,

MetaOCaml is not very suitable for writing offline generators because of CSP. Our type system introduces residualizable code types to solve the problem.

1.1 Organization of the Paper

Section 2 gives an informal overview of our calculus $\lambda^{\triangleright\%}$ after a brief review of $\lambda^{\triangleright}$. Section 3 defines the syntax, type system, and full reduction of $\lambda^{\triangleright\%}$ formally and shows relevant properties. Then, Section 4 defines the staged semantics and shows Progress and Type-Safe Residualization. Finally, Section 6 discusses related work and Section 7 gives concluding remarks.

2 Informal Overview of $\lambda^{\triangleright\%}$

In this section, we give an informal overview of $\lambda^{\triangleright\%}$ after reviewing $\lambda^{\triangleright}$ [10], on which $\lambda^{\triangleright\%}$ is based.

2.1 $\lambda^{\triangleright}$

In $\lambda^{\triangleright}$, brackets and escapes are written "$\blacktriangleright_\alpha\, M$" and "$\blacktriangleleft_\alpha\, M$", respectively. For example, the first example in Section 1 can be represented as:

$$M_1 \stackrel{\text{def}}{=} (\lambda a : \tau.\, \blacktriangleright_\alpha(\blacktriangleleft_\alpha\, a * \blacktriangleleft_\alpha\, a))\ (\blacktriangleright_\alpha\, 1 + 2)$$

where τ is a suitable type for code values, which we discuss below. In addition to ordinary β-reduction, there is a reduction rule to cancel a pair of brackets under an escape:

$$\blacktriangleleft_\alpha(\blacktriangleright_\alpha\, M) \longrightarrow M.$$

So, M_1 reduces to $\blacktriangleright_\alpha(1 + 2) * (1 + 2)$ in three steps. The type system assigns type $\triangleright_\alpha\tau$, which means the type of code of type τ, to $\blacktriangleright_\alpha\, M$ when M is of type τ. The type system also enforces the argument to $\blacktriangleleft_\alpha$ to be of type $\triangleright_\alpha\tau$ to prevent values other than code from being spliced into a quotation.

The subscript α is called *transition variable*, which intuitively denotes how "thick" the bracket is. A transition variable can be abstracted by $\Lambda\alpha.M$ and instantiated by an application "$M\,A$". Here, A (called *transition*) is a (possibly empty) sequence of transition variables $\alpha_1 \cdots \alpha_n$. For example, $(\Lambda\alpha.(\blacktriangleright_\alpha(\lambda x : \mathbf{int}.x)))\ (\beta\gamma)$ reduces to $\blacktriangleright_{\beta\gamma}(\lambda x : \mathbf{int}.x)$, which is an abbreviation of $\blacktriangleright_\beta \blacktriangleright_\gamma(\lambda x : \mathbf{int}.x)$. A transition abstraction $\Lambda\alpha.M$ is given type $\forall\alpha.\tau$ if the type of M is τ and an application $M\,A$ is given type $\tau[\alpha := A]$ if the type of M is $\forall\alpha.\tau$. For example, $M_2 \stackrel{\text{def}}{=} \Lambda\alpha.(\blacktriangleright_\alpha(\lambda x : \mathbf{int}.x))$ is given type $\forall\alpha.\, \triangleright_\alpha\, (\mathbf{int} \to \mathbf{int})$ and $M_2\ (\beta\gamma)$ is $\triangleright_\beta\triangleright_\gamma(\mathbf{int} \to \mathbf{int})$. Transition variables are similar to environment classifiers in λ^α and the forms of terms also look like those in λ^α. One notable difference is that, in λ^α, a classifier abstraction can be applied only to a single classifier.

One pleasant effect of generalizing transition applications is that **run** M can be expressed as a derived form, rather than a dedicated construct. Namely, **run** M desugars into $M\ \varepsilon$, application to the *empty* sequence of transition variables. For example, the

second example in Section 1 can be represented as $(\Lambda\alpha.M_1)\ \varepsilon$, which first reduces to $(\Lambda\alpha.\blacktriangleright_\alpha((1+2)*(1+2)))\,\varepsilon$ (by reducing the body of $\Lambda\alpha.$) and then to $\blacktriangleright_\varepsilon(1+2)*(1+2)$, which, as we shall see later, is identified with $(1+2)*(1+2)$. Notice that $\blacktriangleright_\alpha$ standing for quotation has disappeared by substitution of ε for α. From the typing point of view, **run** takes $\forall\alpha.\ \triangleright_\alpha\ \tau$ and returns τ, representing the behavior of **run**. It is important that **run** takes $\forall$-types, because typing rules guarantee that a term of type $\forall\alpha.\ \triangleright_\alpha\ \tau$ does not contain free variables inside $\blacktriangleright_\alpha$, making it safe to remove $\blacktriangleright_\alpha$.

2.2 Adding CSP to $\lambda^{\triangleright}$

Next, we informally explain how we extend $\lambda^{\triangleright}$ with CSP to develop $\lambda^{\triangleright\%}$. Unlike MetaOCaml, where CSP is implicit, $\lambda^{\triangleright\%}$ has a dedicated construct $\%_\alpha\ M$ for CSP (as in Nielsen and Taha [8] and Benaissa et al. [11]). For example, the third example in Section 1 is represented as:

$$M_3 \stackrel{\text{def}}{=} (\lambda a : \mathbf{int}.\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ a * \%_\alpha\ a))\ (1+2).$$

Call-by-value reduction leads to $\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ 3 * \%_\alpha\ 3)$, which we consider is already a value. It may appear reasonable to allow reduction to remove $\%$ and regard $\Lambda\alpha.\blacktriangleright_\alpha\ 3*3$ as a value, but such reduction means that the run-time value 3 is converted to an integer literal and lifted into a quotation. As we mentioned already, however, lifting is not always possible, so we reject this idea.

Instead, we consider the CSP operator just *a syntactic marker waiting for* **run** *to dissolve the surrounding brackets*: for example, **run** M_3 (namely $M_3\ \varepsilon$) reduces first to $(\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ 3 * \%_\alpha\ 3))\ \varepsilon$ and then to $\blacktriangleright_\varepsilon(\%_\varepsilon\ 3 * \%_\varepsilon\ 3)$, which will be identified with $3*3$. One amusing consequence of this interpretation is that we do not even have to add reduction rules for $\%$—just extending the definition of substitution of transitions suffices.

Now we consider typing. In $\lambda^{\triangleright}$, a type judgment is of the form $\Gamma \vdash^A M : \tau$, in which transition A stands for the stage of the term, or, roughly speaking, how many brackets are surrounding M. Representative typing rules are those for $\blacktriangleright$ and $\blacktriangleleft$:

$$\frac{\Gamma \vdash^{A\alpha} M : \tau}{\Gamma \vdash^{A} \blacktriangleright_\alpha M : \triangleright_\alpha\tau}\ (\blacktriangleright) \qquad \frac{\Gamma \vdash^{A} M : \triangleright_\alpha\tau}{\Gamma \vdash^{A\alpha} \blacktriangleleft_\alpha M : \tau}\ (\blacktriangleleft)$$

The rule $\blacktriangleright$ means that a quotation is given a code type at stage A if its body is well typed at the next stage $A\alpha$ and $\blacktriangleleft$ is its converse.

Then, a straightforward rule for CSP would be something like

$$\frac{\Gamma \vdash^{A} M : \tau}{\Gamma \vdash^{A\alpha} \%_\alpha\ M : \tau}$$

It is very similar to $\blacktriangleleft$, but M can be of an arbitrary type. Actually, this rule works as far as standard type safety is concerned: a term $\%_\alpha\ M$ interacts with its surrounding context only when ε is substituted for α but then, $\%_\alpha$ disappears and yields a term of type τ, which is exactly what the context expects.

However, this rule does not quite work when we consider program residualization, by which we mean that a generated code can be dumped into a file, just as partial

evaluators (and generating extensions) [12] do. We expect Type-Safe Residualization, which means residual programs are type safe in the following sense:

> If $\vdash^{\varepsilon} M : \triangleright_{\alpha}\tau$ and $M \longrightarrow^{*} V$ for a value V, then $V = \blacktriangleright_{\alpha} N$ for some term N such that $\vdash^{\varepsilon} N : \tau$.

Notice that N has to be typed at stage ε in the conclusion. For example, if $V = \blacktriangleright_{\alpha}((\lambda x : \mathbf{int}.x + 4)\ 5)$, then its body is well typed at stage ε without any problem. However, if $V = \blacktriangleright_{\alpha}((\%_{\alpha}(\lambda x : \mathbf{int}.x + 4))\ 5)$, then its body $(\%_{\alpha}(\lambda x : \mathbf{int}.x + 4))\ 5$ is *not* well typed because $\%_{\alpha}$ can appear only under $\blacktriangleright_{\alpha}$. One way to sidestep this anomaly is to adjust the statement to something like "N is typeable after removing occurrences of $\%_{\alpha}$ at stage ε" so that we can consider $\blacktriangleright_{\alpha}((\lambda x : \mathbf{int}.x + 4)\ 5)$ instead of $\blacktriangleright_{\alpha}((\%_{\alpha}(\lambda x : \mathbf{int}.x + 4))\ 5)$, but it would mean that residualization requires lifting of function values, which is not feasible.

We solve this problem by distinguishing two kinds of transition variables (and two kinds of code types thereby). A transition variable of one kind can be used in CSP but cannot be used to annotate residual code, whereas the other kind can be used for residual code but not for CSP. Typing rules ensure that a transition variable of the first kind is instantiated only by the empty sequence. The property above holds only when α is of the second kind.

3 $\lambda^{\triangleright\%}$

We now present $\lambda^{\triangleright\%}$ in detail. In this section, we will define syntax, (full) reduction and type system of $\lambda^{\triangleright\%}$, and prove subject reduction, strong normalization, and confluence. In the next section, we will study call-by-value staged semantics.

3.1 Syntax

Let Σ and Π be countably infinite sets of *transition variables*, ranged over by α, β, and γ, and *variables*, ranged over by x, y, and z, respectively. A *transition*, denoted by A and B, is a finite sequence of transition variables; we write ε for the empty sequence and AB for the concatenation of A and B.

The syntax of $\lambda^{\triangleright\%}$ is defined by the following grammar.

Variables	x, y, z	$\in \Pi$
Transition variables	α, β, γ	$\in \Sigma$
Transitions	A, B	$\in \Sigma^{*}$
Types	τ, σ, ϕ	$::= b \mid \tau \to \tau \mid \triangleright_{\alpha}\tau \mid \forall\alpha.\tau \mid \forall^{\varepsilon}\alpha.\tau$
Terms	M, N	$::= x \mid \lambda x : \tau.M \mid M\ N \mid \blacktriangleright_{\alpha} M \mid \blacktriangleleft_{\alpha} M$
		$\mid \Lambda\alpha.M \mid M\ A \mid \%_{\alpha} M$

A type is a base type (ranged over by b), a function type, a code type or an α-closed type (of two kinds). A code type $\triangleright_{\alpha}\tau$, indexed by a transition variable, denotes a code fragment of a term of type τ. Two kinds $\forall\alpha.\tau$ and $\forall^{\varepsilon}\alpha.\tau$ of α-closed types (where α

is bound) correspond to the form of transition abstraction $\Lambda\alpha.M$. As we will see, the type system guarantees that the body M does not contain any free variable at any stage containing α. The type constructor $\triangleright_\alpha$ connects tighter than $\to$ and $\to$ tighter than the two forms of $\forall$: for example, $\triangleright_\alpha\tau \to \sigma$ means $(\triangleright_\alpha\tau) \to \sigma$ and $\forall\alpha.\tau \to \sigma$ means $\forall\alpha.(\tau \to \sigma)$.

In addition to the standard λ-terms, there are five more forms: $\blacktriangleright_\alpha M$, $\blacktriangleleft_\alpha M$, $\Lambda\alpha.M$, $M\,A$ and $\%_\alpha M$, as we discussed in the last section. A term of the form $\blacktriangleright_\alpha M$ represents a code fragment M, and $\blacktriangleleft_\alpha M$ unquote, or "escape." Terms $\Lambda\alpha.M$ and $M\,A$ are an abstraction and an instantiation of a transition variable, respectively. Finally, $\%_\alpha M$ is a primitive for cross-stage persistence.

The term constructors $\blacktriangleright_\alpha$, $\blacktriangleleft_\alpha$ and $\%_\alpha$ connects tighter than the two forms of applications and, as usual applications are left-associative and the two binders extends as far to the right as possible: for example, $\blacktriangleright_\alpha\, x\, y$ means $(\blacktriangleright_\alpha\, x)\, y$ and $\blacktriangleright_\alpha\, \lambda x : \tau.x\, y$ means $(\blacktriangleright_\alpha\, \lambda x : \tau.x\, y)$ and $\Lambda\alpha.\ \lambda x : \tau.x\, y$ means $\Lambda\alpha.\ (\lambda x : \tau.(x\, y))$.

As usual, the variable x is bound in $\lambda x : \tau.M$. The transition variable α is bound in $\Lambda\alpha.M$. We identify α-convertible terms and assume the names of bound variables are pairwise distinct. We write $\mathrm{FV}(M)$ and $\mathrm{FTV}(M)$ for the set of free transition variables and the set of free variables in M, respectively. We omit their straightforward definitions.

3.2 Reduction

Next, we define full reduction for $\lambda^{\triangleright\%}$. Before giving reduction rules, we need to define (capture-avoiding) substitutions for the two kinds of variables. We omit the straightforward definition of substitution $M[x := N]$ of a variable for a term but show the definition of substitution $[\alpha := A]$ of a transition variable for a transition in Figure 1. The definition is mostly straightforward. Note that, when a transition variable of $\blacktriangleleft$ and $\%$ is replaced, the order of transition variables is reversed because $\blacktriangleleft$ and $\%$ are kind of inverse to $\blacktriangleright$.

Definition 1 (Reduction). *The reduction relation $M \longrightarrow M'$ is the least relation closed under the three computation rules (β, $\blacktriangleleft\blacktriangleright$, and β_Λ) and (full) congruence rules, which we omit here.*

$$
\begin{aligned}
(\lambda x : \tau.M)\, N &\longrightarrow M[x := N] && (\beta)\\
\blacktriangleleft_\alpha \blacktriangleright_\alpha M &\longrightarrow M && (\blacktriangleleft\blacktriangleright)\\
(\Lambda\alpha.M)\, A &\longrightarrow M[\alpha := A] && (\beta_\Lambda)
\end{aligned}
$$

In addition to ordinary β-reduction, there are two new reductions. The rule $\blacktriangleleft\blacktriangleright$ means that escape cancels a quotation. The other rule β_Λ means that a transition abstraction applied to a transition reduces to the body of the abstraction, where the argument transition is substituted for the transition variable. It is interesting to see that there is no reduction rule that explicitly concerns CSP! As we have discussed already, a CSP is just a syntactic marker waiting for the indexing transition variable to disappear by substitution of the empty sequence.

We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

$$
\begin{aligned}
(A\alpha)[\alpha := B] &= (A[\alpha := B])B \\
(A\alpha)[\beta := B] &= (A[\beta := B])\alpha \quad (\text{if } \alpha \neq \beta)
\end{aligned}
$$

$$
\begin{aligned}
b[\alpha := A] &= b \\
(\tau \to \sigma)[\alpha := A] &= (\tau[\alpha := A]) \to (\sigma[\alpha := A]) \\
(\triangleright_\alpha \tau)[\alpha := A] &= \triangleright_A (\tau[\alpha := A]) \\
(\triangleright_\beta \tau)[\alpha := A] &= \triangleright_\beta (\tau[\alpha := A]) \quad (\text{if } \alpha \neq \beta) \\
(\forall \alpha.\tau)[\beta := A] &= \forall \alpha.(\tau[\beta := A]) \quad (\text{if } \alpha \neq \beta \text{ and } \alpha \notin A) \\
(\forall^\varepsilon \alpha.\tau)[\beta := A] &= \forall^\varepsilon \alpha.(\tau[\beta := A]) \quad (\text{if } \alpha \neq \beta \text{ and } \alpha \notin A)
\end{aligned}
$$

$$
\begin{aligned}
x[\alpha := A] &= x \\
(\lambda x : \tau.M)[\alpha := A] &= \lambda x : (\tau[\alpha := A]).(M[\alpha := A]) \\
(M\ N)[\alpha := A] &= (M[\alpha := A])\ (N[\alpha := A]) \\
(\blacktriangleright_\beta M)[\alpha := A] &= \blacktriangleright_{\beta[\alpha:=A]} (M[\alpha := A]) \\
(\blacktriangleleft_\beta M)[\alpha := A] &= \blacktriangleleft_{\beta[\alpha:=A]} (M[\alpha := A]) \\
(\%_\beta M)[\alpha := A] &= \%_{\beta[\alpha:=A]} (M[\alpha := A]) \\
(\Lambda\beta.M)[\alpha := A] &= \Lambda\beta.(M[\alpha := A]) \quad (\text{if } \beta \neq \alpha \text{ and } \beta \notin A) \\
(M\ B)[\alpha := A] &= (M[\alpha := A])\ (B[\alpha := A])
\end{aligned}
$$

Here, $\triangleright_A \tau$, $\blacktriangleright_A M$, $\blacktriangleleft_A M$ and $\%_A M$ (where $A = \alpha_1 \alpha_2 \cdots \alpha_n$) denote:

$$
\begin{aligned}
\triangleright_A \tau &= \triangleright_{\alpha_1} \triangleright_{\alpha_2} \cdots \triangleright_{\alpha_n} \tau \\
\blacktriangleright_A M &= \blacktriangleright_{\alpha_1} \blacktriangleright_{\alpha_2} \cdots \blacktriangleright_{\alpha_n} M \\
\blacktriangleleft_A M &= \blacktriangleleft_{\alpha_n} \blacktriangleleft_{\alpha_{n-1}} \cdots \blacktriangleleft_{\alpha_1} M \\
\%_A M &= \%_{\alpha_n} \%_{\alpha_{n-1}} \cdots \%_{\alpha_1} M.
\end{aligned}
$$

In particular, $\blacktriangleright_\varepsilon M = \blacktriangleleft_\varepsilon M = \%_\varepsilon M = M$.

Fig. 1. Transition Substitution.

Using integer constants, arithmetic operations, the type of integers, and **let**, we show an example reduction sequence below (where the underlines show the redexes):

$$
\begin{aligned}
&\textbf{let } f = \lambda x : \textbf{int}.x * 2 \textbf{ in} \\
&\quad (\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ \underline{(f\ 1)} + (\%_\alpha\ f)\ (1 + 2)))\ \varepsilon && (\beta) \\
&\longrightarrow^* (\Lambda\alpha.\blacktriangleright_\alpha(\underline{\%_\alpha\ 2} + (\%_\alpha(\lambda x : \textbf{int}.x * 2))\ \underline{(1 + 2)}))\ \varepsilon \\
&\longrightarrow \underline{(\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ 2 + (\%_\alpha(\lambda x : \textbf{int}.x * 2))\ 3))\ \varepsilon} && (\beta_\Lambda) \\
&\longrightarrow 2 + ((\lambda x : \textbf{int}.x * 2)\ 3) \\
&\longrightarrow^* 6
\end{aligned}
$$

Since the reduction is full, there are other reduction sequences as well. The sequence above is not *staged* in the sense that only redexes at the lowest stage are reduced (notice that $1 + 2$ appears under a quotation). We will give staged reduction in the next section.

$$\frac{}{\forall\alpha.\tau <: \forall^{\varepsilon}\alpha.\tau} \qquad \frac{}{\tau <: \tau} \qquad \frac{\tau <: \sigma \qquad \sigma <: \phi}{\tau <: \phi}$$

$$\frac{\tau_1 <: \sigma_1 \qquad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} \qquad \frac{\tau <: \sigma}{\triangleright_\alpha \tau <: \triangleright_\alpha \sigma} \qquad \frac{\tau <: \sigma}{\forall\alpha.\tau <: \forall\alpha.\sigma} \qquad \frac{\tau <: \sigma}{\forall^{\varepsilon}\alpha.\tau <: \forall^{\varepsilon}\alpha.\sigma}$$

Fig. 2. Subtyping Rules.

3.3 Type system

Next, we develop the type system of $\lambda^{\triangleright\%}$. As discussed in Section 2, we distinguish two kinds of transition variables and have two forms of types $\forall\alpha.\tau$ and $\forall^{\varepsilon}\alpha.\tau$ for $\Lambda\alpha.M$. The former can be applied to any transitions but M cannot contain $\%_\alpha$; the latter allows $\%_\alpha$ but can be applied only to ε. For programming convenience, we introduce subtyping between two kinds of $\forall$ types to allow promotion from the former type to the latter.

Subtyping. We first give the subtyping relation.

Definition 2 (Subtyping). *The subtyping relation $\tau <: \sigma$ is the least relation closed under the rules in Figure 2.*

The only interesting rule is the first one, which means that a Λ-abstraction that can be applied to any transitions can also be used in a restricted context where only applications to the empty transition are allowed. The other rules mean that subtyping is reflexive and transitive and that type constructors are covariant except for function types, which are contravariant in argument types.

Typing. A typing context in $\lambda^{\triangleright\%}$ keeps track of not only types of variables but also transitions, which represent which stage it is declared at.

Definition 3 (Typing Context). *A typing context Γ is a finite mapping from variables to pairs of a type and a transition.*

We often write $\Gamma, x : \tau@A$ for the typing context Γ' such that $dom(\Gamma') = dom(\Gamma) \cup \{x\}$ and $\Gamma'(x) = (\tau, A)$ and $\Gamma'(y) = \Gamma(y)$ if $x \neq y$. $\Gamma(x) = (\tau, A)$ means "the variable x at the stage A has the type τ." We write $\mathrm{FTV}(\Gamma)$ for the set of free transition variables in Γ, defined as $\bigcup_{x \in dom(\Gamma)} \{\, \mathrm{FTV}(\tau) \cup \mathrm{FTV}(A) \mid (\tau, A) = \Gamma(x) \,\}$

A type judgment is of the form $\Gamma; \Delta \vdash^A M : \tau$, read "the term M is given type τ under the context Γ and Δ at stage A." Here, Δ is a set of transition variables and records $\forall^{\varepsilon}$-bound transition variables. Intuitively, transition variables in Δ denote the empty sequence and cross-stage persistence is allowed only for them. Conversely, a code type $\triangleright_\alpha \tau$ is residualizable if $\alpha \notin \Delta$.

Definition 4 (Typing). *The typing relation $\Gamma; \Delta \vdash^A M : \tau$ is the least relation closed under the rules in Figure 3.*

$$\frac{}{\Gamma, x : \tau@A; \Delta \vdash^A x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, x : \tau@A; \Delta \vdash^A M : \sigma}{\Gamma; \Delta \vdash^A \lambda x : \tau.M : \tau \to \sigma}\ (\textsc{Abs})$$

$$\frac{\Gamma; \Delta \vdash^A M : \tau \to \sigma \qquad \Gamma; \Delta \vdash^A N : \tau}{\Gamma; \Delta \vdash^A M\, N : \sigma}\ (\textsc{App})$$

$$\frac{\Gamma; \Delta \vdash^{A\alpha} M : \tau}{\Gamma; \Delta \vdash^A \blacktriangleright_\alpha M : \triangleright_\alpha \tau}\ (\blacktriangleright) \qquad \frac{\Gamma; \Delta \vdash^A M : \triangleright_\alpha \tau}{\Gamma; \Delta \vdash^{A\alpha} \blacktriangleleft_\alpha M : \tau}\ (\blacktriangleleft)$$

$$\frac{\Gamma; \Delta \vdash^A M : \tau \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(A) \cup \Delta}{\Gamma; \Delta \vdash^A \Lambda\alpha.M : \forall\alpha.\tau}\ (\textsc{Gen}) \qquad \frac{\Gamma; \Delta \vdash^A M : \forall\alpha.\tau}{\Gamma; \Delta \vdash^A M\, B : \tau[\alpha := B]}\ (\textsc{Ins})$$

$$\frac{\Gamma;\ \Delta \cup \{\alpha\} \vdash^A M : \tau \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(A) \cup \Delta}{\Gamma; \Delta \vdash^A \Lambda\alpha.M : \forall^\varepsilon\alpha.\tau}\ (\textsc{GenE})$$

$$\frac{\Gamma;\ \Delta \vdash^A M : \forall^\varepsilon\alpha.\tau \qquad \beta \in \Delta \text{ whenever } \beta \in B}{\Gamma;\ \Delta \vdash^A M\, B : \tau[\alpha := B]}\ (\textsc{InsE}) \qquad \frac{\Gamma; \Delta \vdash^A M : \tau \qquad \alpha \in \Delta}{\Gamma; \Delta \vdash^{A\alpha} \%_\alpha M : \tau}\ (\%)$$

$$\frac{\Gamma; \Delta \vdash^A M : \tau \qquad \tau <: \sigma}{\Gamma; \Delta \vdash^A M : \sigma}\ (\textsc{Sub})$$

Fig. 3. Typing Rules.

The rules Var, Abs and App are mostly same as those in the simply typed lambda calculus, except for stage annotations. The rule Var means that a variable can appear only at the stage in which it is declared; the rule Abs requires the parameter and the body to be at the same stage; similarly, the rule App requires M and N to be typeable at the same stage. The following four rules $\blacktriangleright$, $\blacktriangleleft$, Gen and Ins are essentially the same as those of $\lambda^{\triangleright}$, except that Δ is added to typing judgments. The rule $\blacktriangleright$ means that, if M is of type τ at stage $A\alpha$, $\blacktriangleright_\alpha A$ is code of type τ at stage A; the rule $\blacktriangleleft$ is its converse. The rules Gen and Ins are the introduction and elimination of $\forall$ types, respectively. The side condition of the rule (Gen) guarantees α-closedness of M, which means M has no free variable which has a transition variable α in its type or its stage.

The next two rules GenE and InsE for $\forall^\varepsilon$ are very similar to Gen and Ins, respectively, but there are two important differences. In GenE, the transition variable α must be in the second component $\Delta \cup \{\alpha\}$ of the premise, so that CSP with α is possible. In InsE, the argument B has to consist only of transition variables from Δ—B is virtually the empty sequence. The next rule % is for CSP, which is allowed only when the indexing transition variable is in Δ.

The last rule stands for ordinary subsumption.

3.4 Properties

We show three basic properties of the calculus: subject reduction, strong normalization and confluence.

Subject Reduction. The key lemma to prove subject reduction is Substitution Lemma as usual. We show that transition substitution $[\alpha := A]$ preserves subtyping and typing; and that term substitution $[x := M]$ preserves typing. There are two separate statements for transition substitution and typing because a transition variable in Δ can be replaced only with the "virtually empty" transitions.

Lemma 1 (Substitution Lemma).

1. *If* $\tau <: \sigma$*, then* $\tau[\alpha := B] <: \sigma[\alpha := B]$
2. *If* $\Gamma, x : \tau@B; \Delta \vdash^A M : \tau$ *and* $\Gamma : \Delta \vdash^B N : \tau$*, then* $\Gamma; \Delta \vdash^A M[x := N] : \tau$
3. *If* $\alpha \notin \Delta$ *and* $\Gamma; \Delta \vdash^A M : \tau$*, then* $\Gamma[\alpha := B]; \Delta \vdash^{A[\alpha:=B]} M[\alpha := B] : \tau[\alpha := B]$
4. *If* $\alpha \in \Delta$ *and* $\Gamma; \Delta \vdash^A M : \tau$ *and* $\beta \in \Delta$ *for any* $\beta \in B$*, then* $\Gamma[\alpha := B]; (\Delta \setminus \{\alpha\} \cup \mathrm{FTV}(B)) \vdash^{A[\alpha:=B]} M[\alpha := B] : \tau[\alpha := B]$

Proof. Straightforward induction on subtyping and typing derivations.

Theorem 1 (Subject Reduction). *If* $\Gamma, \Delta \vdash^A M : \tau$ *and* $M \longrightarrow M'$ *then* $\Gamma, \Delta \vdash^A M' : \tau$.

Proof. By induction on the derivation of $M \longrightarrow M'$.

Strong Normalization. Well-typed terms are strongly normalizing:

Theorem 2 (Strong Normalization). *If a term* M *is typeable, there is no infinite reduction sequence* $M \longrightarrow M' \longrightarrow M'' \longrightarrow \cdots$ *starting with* M.

Proof. First we define translation from $\lambda^{\triangleright\%}$-terms to simply typed λ-terms; the translation just removes all staging annotations. Then, it is easy to show that the translation preserves typeability and one-step β reduction. It is also easy to see that an infinite reduction sequence in $\lambda^{\triangleright\%}$, which necessarily contains infinite β-reduction steps, can be translated to an infinite reduction sequence in the simply typed λ-calculus, contradicting strong normalization of the simply typed λ-calculus.

Confluence. We prove confluence by using the standard technique of parallel reduction and complete development [13]. We omit the proof since it is entirely standard.

Theorem 3 (Confluence). *For any term* M*, if* $M \longrightarrow^* M_1$ *and* $M \longrightarrow^* M_2$*, there exists* M_3 *that satisfies* $M_1 \longrightarrow^* M_3$ *and* $M_2 \longrightarrow^* M_3$.

4 Staged Semantics

The reduction relation given in the last section is full reduction, where an arbitrary subterm can be reduced nondeterministically, and it is not clear if computation can be properly staged in the sense that code generation can be completed without computing inside quotation.

In this section, we will define a deterministic call-by-value *staged* semantics, which can be easily seen as program execution, and show the standard progress property. We obtain the new semantics by allowing reduction at the lowest possible stages (and fixing

the evaluation order). As a result, the rules β and β_Λ are allowed only at the stage ε and the rule $\blacktriangleleft\blacktriangleright$ only at a stage α. (Notice that a redex $\blacktriangleleft_\alpha \blacktriangleright_\alpha M$ is supposed to appear under a quotation in a well-typed term.)

We begin with the definitions of values and redexes.

Definition 5 (Values and Redexes). *The family V^A of sets of values, ranged over by v^A and the sets of ε-redexes (ranged over by R^ε) and α-redexes (ranged over by R^α) are defined by the following grammar. In the grammar, A is nonempty.*

$$
\begin{array}{lll}
\textit{Values} & v^\varepsilon \in V^\varepsilon ::= \lambda x : \tau.M \mid \blacktriangleright_\alpha v^\alpha \mid \Lambda\alpha.v^\varepsilon \\
& v^A \in V^A ::= x \mid \lambda x : \tau.v^A \mid v^A\, v^A \mid \blacktriangleright_\alpha v^{A\alpha} \\
& \qquad\quad \mid \Lambda\alpha.v^A \mid v^A\, B \\
& \qquad\quad \mid \blacktriangleleft_\alpha v^{A'} (\textit{if } A'\alpha = A \textit{ and } A' \neq \varepsilon) \\
& \qquad\quad \mid \%_\alpha v^{A'} (\textit{if } A'\alpha = A) \\
\textit{Redexes} & R^\varepsilon ::= (\lambda x : \tau.M)\, v^\varepsilon \mid (\Lambda\alpha.v^\varepsilon)\, A \\
& R^\alpha ::= \blacktriangleleft_\alpha \blacktriangleright_\alpha M
\end{array}
$$

Values at stage ε consist of abstractions and quotations. The body of a λ-abstraction can be any term, whereas the body of a transition abstraction must be a value. It means that the bodies of transition abstractions are reduced before transition application is reduced. The body of a quotation is a value at a higher stage. Since evaluation at higher stages are not performed during code generation, values at higher stages contain all forms of terms.[3] Redexes are classified into two, according to the stage where they appear.

Then, we define evaluation contexts, which are indexed by two stages and written E^A_B. Intuitively, A stands for that of the whole context when the stage of the hole is B.

Definition 6 (Evaluation Contexts). *The family of sets $ECtx^A_B$ of evaluation contexts, ranged over by E^A_B, is defined by the grammar below. In the grammar, A is nonempty (whereas B, A' and B' can be empty).*

$$
\begin{array}{l}
E^\varepsilon_B \in ECtx^\varepsilon_B ::= \Box\ (\textit{if } B = \varepsilon) \mid E^\varepsilon_B\, M \mid v^\varepsilon\, E^\varepsilon_B \mid \blacktriangleright_\alpha E^\alpha_B \mid \Lambda\alpha.E^\varepsilon_B \mid E^\varepsilon_B\, A' \\
E^A_B \in ECtx^A_B ::= \Box\ (\textit{if } A = B) \mid \lambda x : \tau.E^A_B \mid E^A_B\, M \mid v^A\, E^A_B \mid \blacktriangleright_\alpha E^{A\alpha}_B \\
\qquad\qquad \mid \blacktriangleleft_\alpha E^{A'}_B\ (\textit{where } A'\alpha = A) \mid \Lambda\alpha.E^A_B \mid E^A_B\, B' \mid \%_\alpha E^{A'}_B\ (\textit{where } A'\alpha = A)
\end{array}
$$

We show a few examples of evaluation contexts below.

$$
\begin{array}{r}
\Box\ (\lambda x : \tau.x) \in ECtx^\varepsilon_\varepsilon \\
(\lambda x : \tau.x)\ (\blacktriangleright_\alpha \Box) \in ECtx^\varepsilon_\alpha \\
\blacktriangleright_\beta \blacktriangleleft_\alpha \blacktriangleright_\alpha \blacktriangleright_\gamma \Box \in ECtx^\varepsilon_{\beta\gamma}
\end{array}
$$

We write $E^A_B[M]$ for a term obtained by filling the hole in E^A_B with M.

Definition 7 (Staged Reduction). *The staged reduction relation, written $M \longrightarrow_s M'$, is defined by the least relation closed under the rules in Figure 4.*

$$E^A_\varepsilon[(\lambda x : \tau.M)\ v^\varepsilon] \longrightarrow_s E^A_\varepsilon[M[x := v^\varepsilon]] \qquad (\beta_v)$$
$$E^A_\varepsilon[(\Lambda\alpha.v^\varepsilon)\ B] \longrightarrow_s E^A_\varepsilon[v^\varepsilon[\alpha := B]] \qquad (\beta_\Lambda)$$
$$E^A_\alpha[\blacktriangleleft_\alpha \blacktriangleright_\alpha v^\alpha] \longrightarrow_s E^A_\alpha[v^\alpha] \qquad (\blacktriangleleft\blacktriangleright)$$

Fig. 4. Staged Reduction.

The rules are rather straightforward adaptations of the reduction rules for $\longrightarrow$. Note that, in the first two rules, the lower index of the evaluation context is ε, which means the redex appears at stage ε and that, in the third rule, it is α, which means the redex appears inside a (single) quotation.

For example, the reduction sequence for the example shown before is as follows:

$$\begin{array}{ll}
\textbf{let}\ f = \lambda x : \textbf{int}.x * 2\ \textbf{in} & \\
\quad (\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ \underline{(f\ 1)} + (\%_\alpha\ f)\ (1 + 2)))\ \varepsilon & (\beta) \\
\longrightarrow_s^* (\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ \underline{(1 * 2)} + (\%_\alpha(\lambda x : \textbf{int}.x * 2))\ (1 + 2)))\ \varepsilon & \\
\longrightarrow_s \underline{(\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha\ 2 + (\%_\alpha(\lambda x : \textbf{int}.x * 2))\ (1 + 2)))\ \varepsilon} & (\beta_\Lambda) \\
\longrightarrow_s 2 + ((\lambda x : \textbf{int}.x * 2)\ (1 + 2)) & \\
\longrightarrow_s^* 8. &
\end{array}$$

4.1 Properties of Staged Reduction

First, it is easy to see that $\longrightarrow_s$ is a subrelation of $\longrightarrow$. So, the relation $\longrightarrow_s$ has strong normalization and subject reduction.

Theorem 4. $\longrightarrow_s \subseteq \longrightarrow$.

Proof. By case analysis of the rules of $\longrightarrow_s$.

Every well-typed term can be either a value or decomposed into an evaluation context and a redex uniquely. Thanks to this theorem, we know that $\longrightarrow_s$ is deterministic.

Theorem 5 (Unique Decomposition). *If Γ does not have any variable declared at stage ε and $\Gamma;\Delta \vdash^A M : \tau$, then either (1) $M \in V^A$, or (2) there exists a unique pair (E^A_B, R^B) such that $M = E^A_B[R^B]$ for some B, which is either ε or a transition variable β.*

Proof. By induction on the derivation of $\Gamma;\Delta \vdash^A M : \tau$.

Unique Decomposition usually states that a term M is either a value or there is another term that it reduces to, if M is a *closed* well-typed term. In $\lambda^{\triangleright\%}$, free variables at higher-stages can be considered symbols, so we can relax the closedness condition in stating the property.

Thanks to Unique Decomposition, Progress is easy to show.

[3] The only exception is that an escape cannot appear at stage α because the term of the form $\blacktriangleleft_\alpha v^\varepsilon$ is a redex (if it is well typed).)

Theorem 6 (Progress). *If Γ does not have any variable declared at stage ε and $\Gamma; \Delta \vdash^A M : \tau$, then $M \in V^A$ or there exists M' such that $M \longrightarrow_s M'$.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash^A M : \tau$.

The last property we show is Type-Safe Residualization, which we have discussed in Section 2. It states that if a program of a code type is well typed under the assumption that Δ is empty, i.e., CSP (indexed by free transition variables) is not used, then the result (if any) is certainly a quotation and its body is also typeable at stage ε.

In the statement of the theorem, we use the notation $\Gamma^{-\alpha}$, defined by; $\Gamma^{-\alpha} = \{x : \tau@B \mid x : \tau@\alpha B \in \Gamma\}$.

Theorem 7 (Type-Safe Residualization). *If Γ does not have any variable declared at stage ε and $\Gamma; \emptyset \vdash^\varepsilon M : \triangleright_\alpha \tau$ is derivable then there exists $v^\varepsilon = \blacktriangleright_\alpha N \in V^\varepsilon$, $M \longrightarrow_s^* v^\varepsilon$ and $\Gamma^{-\alpha}; \emptyset \vdash^\varepsilon N : \tau$ is derivable.*

Proof. We show this theorem by two parts. First, we show the existence of $v = \blacktriangleright_\alpha N$, which is reduced from M. Next, we show that $\Gamma^{-\alpha}; \emptyset \vdash^\varepsilon N : \tau$ is derivable.

The first part is proved by case analysis on the form of M. By the first part and the typing rule $\blacktriangleright$, we have a derivation of $\Gamma; \emptyset \vdash^\alpha N : \tau$. So, all we need to show the second part is that if $\Gamma; \emptyset \vdash^\alpha N : \tau$ then $\Gamma^{-\alpha}; \emptyset \vdash^\varepsilon N : \tau$, and we can prove this by induction on the derivation of $\Gamma; \emptyset \vdash^\alpha N : \tau$.

5 Discussion

In this section, we investigate differences between $\lambda^{\triangleright\%}$ and BER MetaOCaml[4] in more detail. We also discuss the relationship between CSP and program residualization in $\lambda^{\triangleright\%}$.

5.1 CSP in MetaOCaml

In MetaOCaml, CSP is implicitly applied to the occurrences of value identifiers (variables and references to module members such as List.map) declared outside brackets. The behavior of CSP in MetaOCaml is, however, subtly different from that of $\lambda^{\triangleright\%}$; actually, it depends on where the identifier is declared.

First, CSP for a variable declared in the same compilation unit works (almost) the same as in $\lambda^{\triangleright\%}$. In the implementation, a code value is represented as an AST and there is a special node that contains a pointer to the value of a variable under CSP[5]. This pointer is dereferenced while the surrounding code is evaluated. In contrast to that, CSP for an identifier in another compilation unit is represented by an AST node that contains the identifier name, which is resolved while the surrounding code is evaluated. The following program (run by BER MetaOCaml version N 101) demonstrates the difference:

[4] A (re)implementation of the original MetaOCaml by Oleg Kiselyov.

[5] For ground values such as integers, this node is replaced with an AST node for a constant.

```
# let f = List.map in .< (f, List.map) >.;;
- : ...
 = .<(((* cross-stage persistent value (id: f) *)), List.map)>.
```

The result is a quoted pair consisting of a pointer to a closure (which is the value of `List.map`) and a module member reference to be resolved later. This lazy name resolution does not affect the result of program execution, because (1) variable reference is a side-effect free operation and (2) resolving the same module name at code-generation time and at code-evaluation time results in the same module implementation.

5.2 CSP and Program Residualization

As already discussed, in $\lambda^{\triangleright\%}$, CSP with a transition variable α can be applied only if α is bound by $\Lambda\alpha$ which has a $\forall^{\varepsilon}\alpha$ type. Due to this restriction, it is impossible to use the same code value both for running and residualization if it contains a reference to a library function, (which can be considered a free variable at stage ε).

Consider the following term (of $\lambda^{\triangleright\%}$ extended with pairs):

$$M = \textbf{let } c = \Lambda\alpha.\blacktriangleright_{\alpha}(1+2) \textbf{ in let } d = c\,\varepsilon \textbf{ in } \blacktriangleright_{\beta}(\blacktriangleleft_{\beta}(c\,\beta), \%_{\beta}\,d)$$

The intention behind this term is to construct a code value representing $1 + 2$, evaluate it to 3, and construct another code value representing $((1 + 2), 3)$ to be residualized. If $+$ is a language primitive (just as numbers), which can be used at any stage, then this term can be given type $\triangleright_{\beta}(\textbf{int} \times \textbf{int})$. However, if $+$ is a free variable at stage ε, the subterm $\Lambda\alpha.\blacktriangleright_{\alpha}(1 + 2)$ is ill typed. One may apply CSP to $+$ to make this subterm well typed but the only type given to this term is $\forall^{\varepsilon}\alpha.\, \triangleright_{\alpha} \textbf{int}$, making another subterm $c\,\beta$ ill typed (here, β cannot be in Δ in the type derivation because the generated code is to be residualized).

Although this may sound very unfortunate because one may expect $+$ is available everywhere, we believe that it is reasonable for the type system to reject this term, because, in general, a library function that is available during code generation may or may not be available when the generated code is executed later. In other words, using the same name at different levels may result in different values.

6 Related Work

Although many multi-stage calculi are studied in the literature, few of them are equipped with all the combination of quasiquotation, run and CSP.

Davies' λ° [14], which can model multi-level generating extensions [15], has quasiquotation but neither run nor CSP. Due to the absence of CSP, Type-Safe Residualization naturally follows.

Davies and Pfenning have proposed modal λ-calculi, whose type systems can be seen as (intuitionistic) S4 modal logic [16]. They do not model CSP but a code fragment can be embedded inside arbitrarily nested quotations. In this sense, code types can cross stages. Such a limited support of CSP is found in other calculi [17, 10].

Taha et al. [18] and Moggi et al. [19] have proposed MSP calculi with quasiquotation, run, and CSP. In these calculi, CSP is implicit as in MetaOCaml and limited to variable references. They satisfy a property similar to Type-Safe Residualization but, unfortunately, the distinction between lifting and CSP is not very clear from its semantics because a variable under implicit CSP is just replaced with a value, e.g., $(\lambda f.\langle f\ 42\rangle)(\lambda x.x+x)$ evaluates to $\langle(\lambda x.x+x)\ 42\rangle$, which looks as if the function $\lambda x.x+x$ were lifted.

Benaissa et al. [11] have presented λ^{BN}, which has an explicit CSP operator up that can be applied to any expressions, as well as quasiquotation and (a limited support for) **run**. Although there is a certain typing restriction on the use of up, this operator can be used for any kind of values, including functions; lifting and CSP are confused here, too.

As we already mentioned, Taha and Nielsen [8] have introduced the notion of environment classifiers to λ^{α}, which has quasiquotation, run, and CSP. In λ^{α}, CSP is explicit (in fact, we borrow the symbol $\%$ from λ^{α}) and can be applied to any expression and λ^{α}-term $\langle \%_{\alpha}\, 3 * \%_{\alpha}\, 3\rangle^{\alpha}$, which would correspond to $\blacktriangleright_{\alpha}(\%_{\alpha}\, 3 * \%_{\alpha}\, 3)$, is also considered a value. Since environment classifiers in λ^{α} cannot be instantiated by the empty sequence, the semantics of **run** is formalized as a reduction step which removes the outermost pair of brackets and adjusts occurrences of $\%$ by a complicated meta-level operation called demotion. For example, **run** $(\alpha)\langle(\%_{\alpha}\, 3 * \%_{\alpha}\, 3)\rangle^{\alpha}$ (where $(\alpha)M$ is a binder of a classifier) reduces to $(\alpha)(3 * 3)$. In the implementation (both the original one [4] and BER MetaOCaml [6] by Kiselyov), a code value is represented by an AST tree, in which CSP is a special node that points to a run-time value; when a quotation is **run** and compiled, a CSP node is compiled to an instruction to dereference the pointer to the value. This implementation scheme matches the intuition that CSP is a syntactic marker that waits for the surrounding code to start **run**ning. Lifting is not needed to implement CSP[7], as far as **run** is concerned, but dumping code values into a file is not generally possible because a CSP node might point to a nonserializable object. We think λ^{α} is a suitable model only of MSP languages without support of generating residual code because the type system does not distinguish code that can/cannot be residualized.

Kim et al. [5] have proposed another multi-stage calculus λ^{sim}_{open}, which is equipped with lifting of arbitrary values so that any value can be embedded into a quotation. So, it seems also difficult to support residualization.

7 Conclusions

We have given the formal definition of $\lambda^{\triangleright\%}$ with its syntax, type system, full reduction and staged reduction. A key idea here is to view CSP as a syntactic marker waiting for **run** to dissolve the surrounding brackets. For the full reduction, where an arbitrary subterm can be reduced nondeterministically, we have proven subject reduction, strong normalization and confluence. For staged reduction, which is a deterministic call-by-value operational semantics, we have proven Progress, Type-Safe Residualization and that staged reduction is a subrelation of the full reduction.

[6] `http://okmij.org/ftp/ML/MetaOCaml.html`

[7] Basic values such as numbers or strings under CSP are converted to literals.

We have also discussed interactions between CSP and program residualization and pointed out a problem that residualization for a value which is put into a bracket by CSP requires lifting that is always not feasible. In this sense, MetaOCaml is not very suitable for writing offline generators. Our type system for $\lambda^{\triangleright\%}$ solves this problem by distinguishing two kinds of transition variables.

Type inference for $\lambda^{\triangleright\%}$ would not be possible as it is for the same reason as λ^{α} [8] and $\lambda^{\triangleright}$ [10], but we would be able to identify a subset of $\lambda^{\triangleright\%}$ in which type inference is possible by a similar approach to Calgano, Moggi and Taha [9].

Acknowledgements. We thank Kenichi Asai and Yukiyoshi Kameyama for valuable comments. We also thank three anonymous reviewers for their helpful comments (in particular, one reviewer for describing how CSP is implemented in MetaOCaml).

References

1. Taha, W.: A gentle introduction to multi-stage programming. In Lengauer, C., Batory, D., Consel, C., Odersky, M., eds.: Domain-Specific Program Generation. Volume 3016 of Lecture Notes in Computer Science. (2004) 30–50
2. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of Haskell Workshop (Haskell'02). 60–75
3. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science **248** (2000) 211–242
4. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: Proceedings of International Conference on Generative Programming and Component Engineering (GPCE'03). (2003) 57–76
5. Kim, I.S., Yi, K., Calcagno, C.: A polymorphic modal type system for Lisp-like multi-staged languages. In: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL06), Charleston, SC (January 2006) 257–268
6. Chen, C., Xi, H.: Meta-programming through typeful code representation. In: Proceedings of ACM International Conference on Functional Programming (ICFP'03), Uppsala, Sweden (August 2003) 275–286
7. Mainland, G.: Explicitly heterogeneous metaprogramming with MetaHaskell. In: Proceedings of ACM International Conference on Functional Programming (ICFP'12), Copenhagen, Denmark (September 2012) 311–322
8. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03). (2003) 26–37
9. Calcagno, C., Moggi, E., Taha, W.: ML-like inference for classifiers. In Schmidt, D., ed.: Proceedings of European Symposium on Programming (ESOP'04). Volume 2986 of Lecture Notes on Computer Science., Barcelona, Spain, Springer Verlag (March/April 2004) 79–93
10. Tsukada, T., Igarashi, A.: A logical foundation for environment classifiers. Logical Methods in Computer Science **6**(4:8) (December 2010) 1–43
11. Benaissa, Z.E.A., Moggi, E., Taha, W., Sheard, T.: Logical modalities and multi-stage programming. In: Proceedings of Workshop on Intuitionstic Modal Logics and Applications (IMLA'99). (1999)
12. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
13. Takahashi, M.: Parallel reductions in lambda-calculus. Inf. Comput. **118**(1) (1995) 120–127

14. Davies, R.: A temporal-logic approach to binding-time analysis. In: Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS 1996), IEEE Computer Society Press (July 1996) 184–195
15. Glück, R., Jørgensen, J.: Efficient multi-level generating extensions for program specialization. In: Proceedings of Programming Languages, Implementations, Logics and Programs (PLILP'95). Volume 982 of Lecture Notes on Computer Science. (1995) 259–278
16. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM **48**(3) (2001) 555–604
17. Yuse, Y., Igarashi, A.: A modal type system for multi-level generating extensions with persistent code. In: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'06), Venice, Italy (2006) 201–212
18. Taha, W., Benaissa, Z.E.A., Sheard, T.: Multi-stage programming: Axiomatization and type-safety. In: Proceedings of 25th International Colloquium on Automata, Languages and Programming (ICALP'98). Volume 1443 of Lecture Notes on Computer Science., Aalborg, Denmark, Springer Verlag (July 1998) 918–929
19. Moggi, E., Taha, W., Benaissa, Z.E.A., Sheard, T.: An idealized MetaML: Simpler, and more expressive. In: Proceedings of European Symposium on Programming (ESOP'99). Volume 1576 of Lecture Notes on Computer Science. (1999) 193–207