A Typed and Unified Reflection for Shift and Shift0*

Yui Tamura[✉][0009−0008−1974−011X] and Kenichi Asai[0000−0001−8040−0394]

Ochanomizu University, Tokyo, Japan
{tamura.yui, asai}@is.ocha.ac.jp

Abstract. A reflection, a formal relationship between source and target calculi, strongly guarantees the correctness of program translations. Although reflection has been shown for delimited-control operators such as shift and shift0, existing reflections primarily focused on untyped calculi with a single control operator and their translations. As previous proofs for shift0 relied on formulations distinct from those for shift, a unified typed approach for both operators has yet to be formulated. In this paper, we present a unified reflection for typed shift and shift0. Recently, our group established a typed reflection for shift and reset. Based on this result, we demonstrate that the framework can be naturally extended with shift0. Our approach starts from a 2CPS interpreter, where both continuations and meta-continuations are explicitly passed. From this interpreter, we systematically derive a colon translation and a target language that avoids creating administrative redexes. Following previous work, we then establish the reflection. A crucial point in the proof is to precisely identify when a continuation is stored in the meta-continuation. The entire proof is formalized in Agda, assuming functional extensionality and postulating a certain structurally trivial rule required by the PHOAS representation. Our results clarify the operational semantics of delimited-control operators on a common foundation and would provide a basis for developing reflection for algebraic effects and handlers.

Keywords: reflection · CPS translation · delimited control · shift · shift0 · Agda

1 Introduction

Reflection [21] establishes a strict correspondence of reductions between source and target calculi, thereby providing a strong mathematical guarantee for the correctness of program translations. This ensures that the behavior and optimizations of a program in the target calculus operate identically in the source calculus. That is, we can safely apply optimizations to the source calculus in the same way as we do for the target one.

* We would like to thank Youyou Cong and the anonymous reviewers for their insightful and constructive comments on an earlier version of this paper.This work is partly supported by JSPS KAKENHI under Grant No. JP25K14986.

In calculi equipped with delimited continuations, reflection takes on an even greater significance. A widely used method to execute programs containing delimited-control operators such as `shift/reset` [9] and `shift0/reset0` [8] is to translate them into continuation-passing style (CPS). By demonstrating reflection, sophisticated optimizations on the complex control structure in CPS can be applied to the source language without performing CPS translation.

Our long-term goal is to establish reflection for algebraic effects and handlers (AEH) proposed by Plotkin and Pretnar [19]. AEH is a powerful language mechanism that isolates computational effects and allows them to be flexibly combined, significantly influencing recent language designs. Reflection for AEH strongly guarantees the correctness of effectful program transformations, leading to safer and more rigorous language implementations. Effect handlers can be classified into deep and shallow handlers, which are macro-expressible in terms of `shift0` and `control0`, respectively. Given this relation, Cong and Asai [6] made significant progress toward a reflection for effect handlers based on `shift0/dollar`[4]; however, complete reflection has not yet been established.

Deep and shallow handlers have their own practical use cases [14], and modern languages like OCaml 5 support both [23]. In environments where both handlers are available, a unified theory to guarantee the correctness of their compilation is essential. Asai and Fujii [2] recently showed that AEH can be defined through an interpreter within a common framework for the four delimited-control operators (i.e., `shift`, `shift0`, `control` [12], and `control0` [13]); investigating these four operators on a common foundation is expected to yield a unified reflection for them, which will naturally extend to both deep and shallow handlers. As a first step, this work focuses on the operators `shift` and `shift0`.

Several studies have investigated reflection for delimited-control operators, presenting proofs for `shift/reset` [3] and `shift0/dollar` [4]. Although `shift` and `shift0` are known to be macro-expressible by each other [22], expressing `shift0` in terms of `shift` requires complex encodings that rely on recursive continuations. Consequently, Biernacki et al. [4] noted that "results established for `shift` do not carry over directly to `shift0`", indicating that `shift` and `shift0` would require independent consideration.

To bridge this gap, we employ a 2CPS interpreter—where both continuations and meta-continuations (outer continuations) are explicitly passed—as a definitional interpreter to express both operators within a unified framework [17]. This common foundation allows us to directly inherit the overall reflection framework for `shift/reset` established in previous work by our group [16]. We can properly formulate the target and intermediate calculi, and directly apply the insights and techniques from the reflection proof to `shift0`.

Previous studies on reflection have primarily focused on untyped calculi. Considering the adoption of AEH in modern typed languages, such as OCaml, we expect that the typed reflection would be indispensable. Although our previous work [16] established a typed reflection for `shift`, a typed reflection for `shift0` has not yet been achieved. In this paper, we establish a typed and unified reflection for `shift` and `shift0`. Importantly, the reflection proof itself does not rely

on types—we have separately formalized the untyped variants in Agda. Rather, the key to the proof lies in the structural annotations (i.e., Δ and Θ), which are required in both typed and untyped settings to precisely track the states of continuations and meta-continuations.

The main contributions of this work are summarized as follows:

- The key to enabling our proof lies in the appropriate definition of the calculi and translations. We systematically derive a CPS translation that avoids administrative redexes by scaling the established derivation method [10] to a 2CPS interpreter. This translation guides the design of a target calculus suitable for the reflection, while incorporating meta-continuation structures in the source calculus enables the inverse translation. Through this systematic design, we extend the proof techniques developed for `shift` to establish reflection for both `shift` and `shift0` within the same framework.
- A crucial design choice is to equip our target calculus with annotations Δ and Θ. While Δ has been utilized in prior work, Θ is newly introduced in this work. These annotations facilitate two essential tasks for the proof: precisely tracking the target k during continuation substitution, and accurately recovering the `reset` during the inverse translation. This precise state management enables us to establish the reflection.
- We fully formalize all proofs in the Agda proof assistant, using intrinsically typed term representation [1] and parameterized higher-order abstract syntax (PHOAS) [5]. Our PHOAS-based implementation leverages the axiom of functional extensionality and a specific postulate for identity substitution; by utilizing these postulates, we complete the entire formalization.

The remainder of this paper is structured as follows. Section 2 provides the technical background and an overview of our approach. Section 3 introduces the exact syntax and semantics of the three distinct calculi employed in our framework. Section 4 derives the CPS translation. Section 5 establishes the reflection. Finally, we discuss related work in Section 6 and conclude in Section 7.

While our unified framework is formally typed, we do not delve deeply into the details of the type systems in this paper to focus on the core mechanisms of the proof. Readers interested in the details are referred to the formalized proof in Agda. The code, including both the typed and untyped setting, is available at:

`http://pllab.is.ocha.ac.jp/~asai/papers/ppdp2026/`

2 Background and Overview

This section first recalls the formal definition of reflection and the behavioral differences between the delimited-control operators `shift` and `shift0`. We then outline our approach for handling both operators.

2.1 Reflection

A *reflection* [21] is a strong form of a Galois connection between a source calculus and a target calculus. In this paper, we write $\longrightarrow$ for a single-step reduction, $\longrightarrow\!\!\!\!\rightarrow$ for its reflexive and transitive closure, and $\equiv$ for syntactic identity up to renaming. Consider a source calculus S with a reduction relation $\longrightarrow_S$ and a target calculus T with a reduction relation $\longrightarrow_T$, where M, M' range over terms of S and N, N' range over terms of T. Given a compiling translation $*\colon S \to T$ and a decompiling translation $\#\colon T \to S$, these translations form a reflection in S of T if the following four conditions hold:

(1) (Left post-inverse) $M \longrightarrow\!\!\!\!\rightarrow_S M^{*\#}$
(2) (Right inverse) $N^{\#*} \equiv_T N$
(3) (Soundness) $M \longrightarrow_S M'$ implies $M^* \longrightarrow\!\!\!\!\rightarrow_T M'^*$
(4) (Completeness) $N \longrightarrow_T N'$ implies $N^\# \longrightarrow\!\!\!\!\rightarrow_S N'^\#$

These conditions guarantee that the translations are operationally step-preserving. Condition (3) ensures soundness by simulating source reductions in the target, while Condition (4) guarantees completeness by verifying that every target reduction corresponds to a valid source reduction.

Condition (2) distinguishes a reflection from a standard Galois connection. While a Galois connection only requires $N^{\#*} \longrightarrow\!\!\!\!\rightarrow_T N$, a reflection demands strict syntactic identity $N^{\#*} \equiv_T N$ (up to renaming of variables). This equivalence implies that the target calculus is isomorphic to a specific subset of the source calculus, known as the *kernel* [21]. Condition (1) states that any source term M reduces to a normalized term $M^{*\#}$ within the kernel.

When we restrict our source calculus to this kernel, Condition (1) becomes a strict syntactic identity $M \equiv_S M^{*\#}$. This relation between the kernel and the target calculus is an *order isomorphism*. Based on this property, prior work [21, 3, 4] has established that every reflection cleanly factors into an inclusion into the kernel and an order isomorphism between the kernel and the target calculus.

2.2 Behaviors of Shift and Shift0

The reduction rules for the delimited-control operators `shift` (written as $Sk.e$) and `shift0` ($S_0k.e$) are as follows, where `reset` ($\langle e \rangle$) acts as a delimiter and $J[...]$ is a pure evaluation context containing no `reset` around the hole.

$$\langle J[Sk.e] \rangle \longrightarrow \langle e[\lambda x.\langle J[x] \rangle / k] \rangle \qquad \langle J[S_0k.e] \rangle \longrightarrow e[\lambda x.\langle J[x] \rangle / k]$$

Both operators capture the current context up to the surrounding `reset`, clear it, bind it to k, and execute the body e. The crucial difference between them is whether the `reset` remains around the execution of e. Unlike `shift`, `shift0` removes this `reset`, which allows it to capture the continuation up to the next outer `reset`. This outer continuation is called a *meta-continuation*.

Below, we present two examples that differ only in their choice of control operator to demonstrate how `shift0` interacts with meta-continuations.

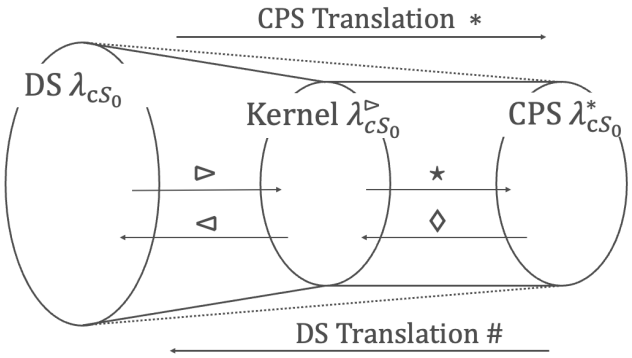


Fig. 1. The structure of the reflection in our formalization.

$$
\begin{aligned}
&\langle(\lambda x.\,1)\mathbin{@}\langle(\lambda y.\,2)\mathbin{@}Sk_1.\,Sk_2.\,(k_1\mathbin{@}0)\rangle\rangle\\
\longrightarrow\ &\langle(\lambda x.\,1)\mathbin{@}\langle Sk_2.\,(k_1\mathbin{@}0)[\lambda z.\,\langle(\lambda y.\,2)\mathbin{@}z\rangle/k_1]\rangle\rangle\\
\longrightarrow\ &\langle(\lambda x.\,1)\mathbin{@}\langle(k_1\mathbin{@}0)[\lambda z.\,\langle(\lambda y.\,2)\mathbin{@}z\rangle/k_1,\lambda z.\,\langle z\rangle/k_2]\rangle\rangle\\
\longrightarrow\ &1\\
&\langle(\lambda x.\,1)\mathbin{@}\langle(\lambda y.\,2)\mathbin{@}S_0k_1.\,S_0k_2.\,(k_1\mathbin{@}0)\rangle\rangle\\
\longrightarrow\ &\langle(\lambda x.\,1)\mathbin{@}(S_0k_2.\,(k_1\mathbin{@}0)[\lambda z.\,\langle(\lambda y.\,2)\mathbin{@}z\rangle/k_1])\rangle\\
\longrightarrow\ &(k_1\mathbin{@}0)[\lambda z.\,\langle(\lambda y.\,2)\mathbin{@}z\rangle/k_1,\lambda z.\,\langle(\lambda x.\,1)\mathbin{@}z\rangle/k_2]\\
\longrightarrow\ &2
\end{aligned}
$$

These examples clearly illustrate the operational difference between `shift` and `shift0`. In the second step, `shift` retains the enclosing `reset`, whereas `shift0` eliminates it. Consequently, in the third line, `shift` binds k_2 to $\lambda z.\,\langle z\rangle$, while `shift0` binds it to $\lambda z.\,\langle(\lambda x.\,1)\mathbin{@}z\rangle$. By capturing the outer evaluation context (i.e., "passing a value to $\lambda x.\,1$"), `shift0` yields a different result. This ability to capture meta-continuations is the key distinction between both operators.

2.3 Overview of Our Approach

Fig. 1 illustrates the structure of the reflection proof in this work. The source calculus, referred to as the DS calculus $\lambda_{c\mathcal{S}_0}$, is a simply-typed λ-calculus extended with let-expressions and the delimited-control operators `shift`, `shift0`, and `reset`. The target calculus, referred to as the CPS calculus $\lambda^{*}_{c\mathcal{S}_0}$, is the calculus obtained via the continuation-passing style (CPS) translation. In our framework, the kernel calculus $\lambda^{\triangleright}_{c\mathcal{S}_0}$ is defined as a subset of the DS calculus, consisting exclusively of terms in A-normal form (ANF). Translations are established between these calculi: the transformation from $\lambda_{c\mathcal{S}_0}$ to $\lambda^{*}_{c\mathcal{S}_0}$ is denoted by the CPS translation $*$, and its inverse by the DS translation $\#$. The CPS translation $*$ is decomposed into an ANF translation $\triangleright$ followed by a kernel CPS translation $\star$, while the DS translation $\#$ is factored into a kernel DS translation $\Diamond$ followed by an embedding $\triangleleft$.

In prior work [21, 3, 4], the relationship between the DS calculus $\lambda_{c\mathcal{S}_0}$ and the kernel calculus $\lambda^{\triangleright}_{c\mathcal{S}_0}$ has typically been treated as a trivial inclusion. Although our kernel calculus is similarly a subset of the DS calculus, we formulate it as a distinct language to clarify its structural relationship with the CPS calculus. Honda et al. [16] defined the kernel calculus to strictly correspond one-to-one with the CPS calculus and thus decomposed the overall proof into two steps: a reflection between $\lambda_{c\mathcal{S}_0}$ and $\lambda^{\triangleright}_{c\mathcal{S}_0}$, and an order isomorphism between $\lambda^{\triangleright}_{c\mathcal{S}_0}$ and $\lambda^{*}_{c\mathcal{S}_0}$. We employ this approach to construct our proof.

In the proof for `shift/reset`, the target calculus was a pure simply-typed λ-calculus, where the state of continuations was managed using an annotation Δ. However, as discussed in Section 2.2, the operator `shift0` captures and manipulates meta-continuations. Consequently, standard CPS frameworks that only account for continuations are insufficient for `shift0`, and the reflection techniques developed for `shift/reset` cannot be directly applied. If we were only concerned with `shift0`, we could adopt the formulation by Biernacki et al. [4], which effectively handles stacked continuations. However, their approach is specialized for `shift0` and cannot directly accommodate both `shift` and `shift0`.

To address this limitation, we employ an interpreter that handles both continuations and meta-continuations (i.e., a stack of outer continuations). This style of passing two continuations is called *2CPS* in this paper. We systematically derive a 2CPS *colon translation* by applying the method of Danvy and Nielsen [10] to this interpreter. A colon translation is a CPS translation designed to avoid administrative redexes during translation; because its output is compact, it is commonly used in establishing reflection. Basing our target calculus on this translation naturally induces a list structure to accurately express the meta-continuation stack. By managing the states of both continuations using the existing annotation Δ and a newly introduced Θ for the shape of the meta-continuation, our framework uniformly accommodates both `shift` and `shift0`.

3 Source, Target, and Kernel Calculi

In this section, we formally define the three calculi used in this paper.

3.1 Direct Style Calculus (DS) $\lambda_{c\mathcal{S}_0}$

A direct-style calculus is based on a simply-typed λ-calculus with let-expressions. Our formulation extends the calculus for `shift` and `reset` by Biernacki et al. [3] with the `shift0` operator. Its syntax is presented in Fig. 2.

A term is either a value V or a non-value P. Values consist of variables, λ-abstractions, and the control operators `shift` $\underline{\mathcal{S}}$ and `shift0` $\underline{\mathcal{S}_0}$. Non-values comprise function applications, let-expressions, and the delimiter `reset` $\langle \underline{M} \rangle$.

Following previous work [3, 4], we define `shift` and `shift0` as combinators (i.e., values) rather than dedicated syntactic constructs, as this approach is technically convenient for our formalization. Consequently, an expression traditionally written as $S_0 k.M$ is expressed as the application of the combinator to a λ-abstraction, $\underline{\mathcal{S}_0}\,\underline{@}\,(\underline{\lambda} k.\, M)$. The same applies to $\underline{\mathcal{S}}$.

$$
\begin{array}{rll}
\text{terms } L, M, N & ::= & V \mid P \\
\text{values } V, W & ::= & x \mid \underline{\lambda} x.\, M \mid \underline{\mathcal{S}} \mid \underline{\mathcal{S}_0} \\
\text{non-values } P, Q & ::= & M \mathbin{\underline{@}} N \mid \underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N \mid \underline{\langle} M \underline{\rangle} \\
\text{pure contexts } J, K & ::= & [\,] \mid K[[\,] \mathbin{\underline{@}} M] \mid K[V \mathbin{\underline{@}} [\,]] \mid K[\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ M] \\
\text{contexts } G & ::= & [\,] \mid G[K[\underline{\langle}[\,]\underline{\rangle}]]
\end{array}
$$

$$
\begin{array}{lllll}
(\beta.v) & (\underline{\lambda} x.\, M) \mathbin{\underline{@}} V & \longrightarrow M[V/x] & \\
(\eta.v) & \underline{\lambda} x.\, (V \mathbin{\underline{@}} x) & \longrightarrow V & \text{if } x \notin fv(V) \\
(\beta.let) & \underline{\mathsf{let}}\ x = V\ \underline{\mathsf{in}}\ M & \longrightarrow M[V/x] & \\
(\eta.let) & \underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ x & \longrightarrow M & \\
(assoc) & \underline{\mathsf{let}}\ y = (\underline{\mathsf{let}}\ x = L\ \underline{\mathsf{in}}\ M)\ \underline{\mathsf{in}}\ N & \longrightarrow \underline{\mathsf{let}}\ x = L\ \underline{\mathsf{in}}\ (\underline{\mathsf{let}}\ y = M\ \underline{\mathsf{in}}\ N) & \text{if } x \notin fv(N) \\
(let.1) & P \mathbin{\underline{@}} N & \longrightarrow \underline{\mathsf{let}}\ x = P\ \underline{\mathsf{in}}\ x \mathbin{\underline{@}} N & \text{if } x \notin fv(N) \\
(let.2) & V \mathbin{\underline{@}} Q & \longrightarrow \underline{\mathsf{let}}\ y = Q\ \underline{\mathsf{in}}\ V \mathbin{\underline{@}} y & \text{if } y \notin fv(V) \\
(\beta.\mathcal{S}) & \underline{\langle} J[\underline{\mathcal{S}} \mathbin{\underline{@}} W] \underline{\rangle} & \longrightarrow \underline{\langle} W \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\langle} J[y] \underline{\rangle}) \underline{\rangle} & \text{if } y \notin fv(J) \\
(\beta.\mathcal{S}_0) & \underline{\langle} J[\underline{\mathcal{S}_0} \mathbin{\underline{@}} W] \underline{\rangle} & \longrightarrow W \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\langle} J[y] \underline{\rangle}) & \text{if } y \notin fv(J) \\
(\beta.\mathcal{R}) & \underline{\langle} V \underline{\rangle} & \longrightarrow V &
\end{array}
$$

Fig. 2. Syntax of $\lambda_{c\mathcal{S}_0}$

In Fig. 2, we define two structural levels of evaluation contexts: pure contexts and general contexts. The pure contexts J, K adopt an inside-out representation. While an outside-in formulation is more conventional in the literature, we deliberately choose the inside-out form to align the structural order of contexts with the kernel calculus. The general context G then represents the outer continuation structure that nests these pure contexts within `reset`, which directly corresponds to the meta-continuation in our framework. As with the pure contexts, its definition is systematically derived from the structure of the kernel calculus. This design is required to straightforwardly define the translation back to the direct style.

The reduction relations are also given in Fig. 2. The first four rules are the standard β- and η-reductions for functions and let-expressions. $(let.1)$ and $(let.2)$ are introduced to name subexpressions for translating terms into A-normal form[1] and $(assoc)$ flattens nested let-expressions. Exhaustive application of these three rules reduces any serious term to an A-normal form. For example, applying these rules to the term $f \mathbin{\underline{@}} ((h \mathbin{\underline{@}} a) \mathbin{\underline{@}} b)$ yields the following:

$$
\begin{array}{lll}
& f \mathbin{\underline{@}} ((h \mathbin{\underline{@}} a) \mathbin{\underline{@}} b) & \\
\longrightarrow & \underline{\mathsf{let}}\ y = (h \mathbin{\underline{@}} a) \mathbin{\underline{@}} b\ \underline{\mathsf{in}}\ f \mathbin{\underline{@}} y & (let.2) \\
\longrightarrow & \underline{\mathsf{let}}\ y = (\underline{\mathsf{let}}\ x = h \mathbin{\underline{@}} a\ \underline{\mathsf{in}}\ x \mathbin{\underline{@}} b)\ \underline{\mathsf{in}}\ f \mathbin{\underline{@}} y & (let.1) \\
\longrightarrow & \underline{\mathsf{let}}\ x = h \mathbin{\underline{@}} a\ \underline{\mathsf{in}}\ (\underline{\mathsf{let}}\ y = x \mathbin{\underline{@}} b\ \underline{\mathsf{in}}\ f \mathbin{\underline{@}} y) & (assoc)
\end{array}
$$

The remaining rules describe the behavior of the delimited-control operators. $(\beta.\mathcal{S})$ captures the nearest pure context J up to the enclosing delimiter and reifies it as a function $\underline{\lambda} y.\, \underline{\langle} J[y] \underline{\rangle}$. This function is then passed to W, enabling the use of the composable continuation within its body. $(\beta.\mathcal{S}_0)$ similarly captures

[1] Biernacki et al. [4] use a single rule $(name)$ that generalizes both $(let.1)$ and $(let.2)$.

$$
\begin{array}{rll}
\text{root } R & ::= & \underline{\lambda}k.\,\underline{\lambda}g.\,M_k \\
\text{terms } M_{(\Delta{+}{+}\Theta)} & ::= & K_\Delta \mathbin{\underline{@}} V \mathbin{\underline{@}} G_\Theta \mid V \mathbin{\underline{@}} W \mathbin{\underline{@}} K_\Delta \mathbin{\underline{@}} G_\Theta \\
\text{values } V, W & ::= & x \mid \underline{\lambda}x.\,\underline{\lambda}k.\,\underline{\lambda}g.\,M_k \mid S \mid S_0 \\
\text{shift } S & ::= & \underline{\lambda}w.\,\underline{\lambda}j.\,\underline{\lambda}g.\,w \mathbin{\underline{@}} (\underline{\lambda}y.\,\underline{\lambda}k.\,\underline{\lambda}g.\,j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :\!\!: g)) \mathbin{\underline{@}} k_{id} \mathbin{\underline{@}} g \\
\text{shift0 } S_0 & ::= & \underline{\lambda}w.\,\underline{\lambda}j.\,\underline{\lambda}(k_0 :\!\!: g).\,w \mathbin{\underline{@}} (\underline{\lambda}y.\,\underline{\lambda}k.\,\underline{\lambda}g.\,j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :\!\!: g)) \mathbin{\underline{@}} k_0 \mathbin{\underline{@}} g \\
\text{continuations } J_\Delta, K_\Delta & ::= & {}_{(\Delta=k)}\,k \mid {}_{(\Delta=\bullet)}k_{id} \mid \underline{\lambda}x.\,\underline{\lambda}g.\,M_\Delta \\
\text{meta-continuations } G_\Theta & ::= & {}_{(\Theta=g)}\,g \mid {}_{(\Theta=\Delta)}\,G_\Delta \\
\text{active meta-conts } G_{(\Delta{+}{+}\Theta)} & ::= & K_\Delta :\!\!: G_\Theta
\end{array}
$$

$$
\begin{array}{lll}
(\beta.v) & (\underline{\lambda}x.\,\underline{\lambda}k.\,\underline{\lambda}g.\,M_k) \mathbin{\underline{@}} V \mathbin{\underline{@}} K_\Delta \mathbin{\underline{@}} G_\Theta \longrightarrow M_k[V/x, K_\Delta/k, G_\Theta/g] & \\
(\eta.v) & \underline{\lambda}x.\,\underline{\lambda}k.\,\underline{\lambda}g.\,V \mathbin{\underline{@}} x \mathbin{\underline{@}} k \mathbin{\underline{@}} g \longrightarrow V & \text{if } x, k, g \notin fv(V) \\
(\beta.let) & (\underline{\lambda}x.\,\underline{\lambda}g.\,M_\Delta) \mathbin{\underline{@}} V \mathbin{\underline{@}} G_\Theta \longrightarrow M_\Delta[V/x, G_\Theta/g] & \\
(\eta.let) & \underline{\lambda}x.\,\underline{\lambda}g.\,K_\Delta \mathbin{\underline{@}} x \mathbin{\underline{@}} g \longrightarrow K_\Delta & \text{if } x, g \notin fv(K) \\
(\beta.\mathcal{S}) & S \mathbin{\underline{@}} W \mathbin{\underline{@}} J_\bullet \mathbin{\underline{@}} G_\Delta \longrightarrow W \mathbin{\underline{@}} (\underline{\lambda}y.\,\underline{\lambda}k.\,\underline{\lambda}g.\,J_\bullet \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :\!\!: g)) \mathbin{\underline{@}} k_{id} \mathbin{\underline{@}} G_\Delta & \\
(\beta.\mathcal{S}_0) & S_0 \mathbin{\underline{@}} W \mathbin{\underline{@}} J_\bullet \mathbin{\underline{@}} (K_\Delta :\!\!: G_\Theta) \longrightarrow W \mathbin{\underline{@}} (\underline{\lambda}y.\,\underline{\lambda}k.\,\underline{\lambda}g.\,J_\bullet \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :\!\!: g)) \mathbin{\underline{@}} K_\Delta \mathbin{\underline{@}} G_\Theta & \\
(\beta.\mathcal{R}) & k_{id} \mathbin{\underline{@}} V \mathbin{\underline{@}} (K_\Delta :\!\!: G_\Theta) \longrightarrow K_\Delta \mathbin{\underline{@}} V \mathbin{\underline{@}} G_\Theta &
\end{array}
$$

Fig. 3. Syntax of $\lambda^*_{c\mathcal{S}_0}$

J, but discards the enclosing `reset`, allowing `shift0` to access outer contexts. Finally, $(\beta.\mathcal{R})$ removes a `reset` when its body is a value.

The type system of $\lambda_{c\mathcal{S}_0}$ is based on the type framework with trails and meta-continuations proposed by Ishio and Asai [17]. Specifically, we omit the trails and adjust the notation accordingly, because trails are required only for `control` and `control0` operators, which are not included in our calculus.

While Materzok and Biernacki [18] propose a type system with subtyping for `shift0` and `reset0`, we employ a monomorphic type system. It is an open problem whether a reflection holds for a typed calculus with subtyping.

3.2 Continuation-Passing Style Calculus (CPS) $\boldsymbol{\lambda^*_{c\mathcal{S}_0}}$

The syntax of the CPS calculus, which we dub $\lambda^*_{c\mathcal{S}_0}$, is presented in Fig. 3. To capture the behavior of S_0, which accesses outer evaluation contexts, this target calculus is formulated as a simply-typed λ-calculus extended with a separate syntactic category of lists and two new kinds of expressions: list abstraction and list application. In our syntax, the list abstraction is realized by the explicit binder for a meta-continuation (e.g., $\underline{\lambda}g$), possibly with pattern matching ($\underline{\lambda}(k_0 :\!\!: g)$). The list application corresponds to passing a meta-continuation ($\underline{@}\,G_\Theta$).

Terms explicitly take continuations and meta-continuations as arguments. The construct $K_\Delta \mathbin{\underline{@}} V \mathbin{\underline{@}} G_\Theta$ represents returning a value V to the current continuation K_Δ with the meta-continuation G_Θ. For instance, the CPS translation of a simple variable x yields $\underline{\lambda}k.\,\underline{\lambda}g.\,k \mathbin{\underline{@}} x \mathbin{\underline{@}} g$, where its body $k \mathbin{\underline{@}} x \mathbin{\underline{@}} g$ directly instantiates this construct. In the standard λ-calculus, this expression could be η-reduced to the well-known form $\underline{\lambda}x.\,\underline{\lambda}k.\,k \mathbin{\underline{@}} x$. However, to keep the syntactic structure uniform, our calculus excludes such η-reduced forms and requires the

explicit passing of a meta-continuation. The construct $V \mathbin{\underline{@}} W \mathbin{\underline{@}} K_\Delta \mathbin{\underline{@}} G_\Theta$ denotes the application of a function V to an argument W along with the current continuations. For example, the CPS translation of an application $f \mathbin{\underline{@}} x$ yields $\underline{\lambda} k.\, \underline{\lambda} g.\, f \mathbin{\underline{@}} x \mathbin{\underline{@}} k \mathbin{\underline{@}} g$, where its body $f \mathbin{\underline{@}} x \mathbin{\underline{@}} k \mathbin{\underline{@}} g$ exemplifies this functional application form. Values consist of variables, λ-abstractions, and the combinators S and S_0, which represent the CPS-transformed forms of the direct-style operators $\underline{\mathcal{S}}$ and $\underline{\mathcal{S}_0}$, respectively. We treat S and S_0 as shorthands for their explicit λ-term definitions, and they are expanded exclusively by the dedicated reduction rules $(\beta.\mathcal{S})$ and $(\beta.\mathcal{S}_0)$.

A continuation K_Δ consists of a continuation constant k, an identity continuation k_{id}, or a continuation $\underline{\lambda} x.\, \underline{\lambda} g.\, M_\Delta$ arising from a let-expression and nested function applications. The annotation Δ can be either k or $\bullet$. Following Biernacki et al. [3], this annotation serves to distinguish the "tail-recursive" parts of a term that return a value to the current continuation k, from "returning" calls within a `reset` that return a value to the identity continuation k_{id}. Intuitively, it specifies to which continuation the evaluation ultimately passes its result. When $\Delta = k$, the computation returns to the default k; conversely, when $\Delta = \bullet$, it returns to k_{id}, i.e., the expression is surrounded by a `reset`. Expanding continuations K_Δ with Δ yields:

$$\begin{aligned} \text{continuations } J_k, K_k &::= k \mid \underline{\lambda} x.\, \underline{\lambda} g.\, M_k \\ J_\bullet, K_\bullet &::= k_{id} \mid \underline{\lambda} x.\, \underline{\lambda} g.\, M_\bullet \end{aligned}$$

For K_k, the base case is the continuation constant k, which represents the initial continuation of the program. In contrast, for $K_\bullet$, it is the identity continuation k_{id}, indicating that the continuation is initialized by a surrounding `reset`.

The continuation variables k and g serve as primitive constants with fixed names; their λ-bindings in Fig. 3 merely specify the lexical scope. In the pure λ-calculus, it is well-known that fixing continuation names in a proper CPS notation is a syntactically sound approach [7, 11]. One might wonder, however, whether this discipline holds in our setting because delimited-control operators allow captured continuations to be invoked in non-tail positions. This concern is resolved by our annotations Δ and Θ, which explicitly track the constant k. The meta-continuation g is safe because it always appears in tail positions. For the continuation k, a potential name clash could occur if the captured continuation j in S and S_0 were substituted with a continuation that contained the constant k. However, as specified by the reduction rules $(\beta.S)$ and $(\beta.S_0)$, the continuation substituted for j is always a delimited continuation of the form $J_\bullet$. The annotation $\Delta = \bullet$ indicates that the continuation contains no occurrence of the constant k. Since j does not contain the constant k, it is never captured, ensuring the safety of fixing the continuation names.

A meta-continuation G_Θ is modeled as a stack of continuations. It is parameterized by the annotation Θ, which can be either g or Δ. When $\Theta = g$, the meta-continuation is the constant g, serving as the default initial meta-continuation. Conversely, when $\Theta = \Delta$, it specifies an *active meta-continuation* G_Δ formed by prepending a continuation to the remaining meta-continuation, where Δ is the annotation of the prepended continuation. Since a meta-continuation represents

the evaluation context outside the innermost enclosing `reset`, an active meta-continuation operationally implies that the current computation is enclosed by one or more resets.

These annotations are essential for two reasons. First, they track the precise position of the continuation constant k during substitution. For example, considering the rule $(\beta.v)$, k within the term M_k can appear either in the continuation position (e.g., $k \mathbin{\underline{@}} x \mathbin{\underline{@}} g$) or within the meta-continuation (e.g., $k_{id} \mathbin{\underline{@}} x \mathbin{\underline{@}} (k :: g)$). The annotations explicitly distinguish these cases, ensuring the correct substitution of K_Δ for k. Second, they encode the behavior of `reset` delimiters, which disappear during the CPS translation. Evaluating a `reset` captures the current continuation, pushes it onto the meta-continuation ($\Theta = \Delta$), and initializes the continuation to the identity ($\Delta = \bullet$). By tracking this structural state with Δ and Θ, the inverse translation can accurately recover the original `reset` boundaries. Since terms and active meta-continuations each incorporate both a continuation and a meta-continuation, their overall state is precisely given by the composition $\Delta{+}{+}\Theta$, which can be either k or $\bullet$. We explain each case with examples.

Case 1: $\Delta = k$**,** $\Theta = g$**,** $k{+}{+}g = k$
The term is not enclosed by `reset`. Considering the term 1, its CPS translation is $\underline{\lambda}k.\, \underline{\lambda}g.\, k \mathbin{\underline{@}} 1 \mathbin{\underline{@}} g$, which has the form $\underline{\lambda}k.\, \underline{\lambda}g.\, K_k \mathbin{\underline{@}} V \mathbin{\underline{@}} G_g$. The body $K_k \mathbin{\underline{@}} V \mathbin{\underline{@}} G_g$ as a whole has an annotation k, because it returns a result $V(= 1)$ to the default continuation $K_k(= k)$. We indicate this relationship by the equation $k{+}{+}g = k$.

Case 2: $\Delta = \bullet$**,** $\Theta = g$**,** $\bullet{+}{+}g = \bullet$
The term returns to the identity continuation k_{id} after evaluation, and the meta-continuation is initial. Such terms appear within continuations arising from a let-expression. Considering the term $\underline{\langle}\underline{\mathsf{let}}\ x = 2\ \underline{\mathsf{in}}\ x\underline{\rangle}$, its CPS translation is $\underline{\lambda}k.\, \underline{\lambda}g.\, (\underline{\lambda}x.\, \underline{\lambda}g.\, k_{id} \mathbin{\underline{@}} x \mathbin{\underline{@}} g) \mathbin{\underline{@}} 2 \mathbin{\underline{@}} (k \mathbin{\underline{::}} g)$, which has the form $\underline{\lambda}k.\, \underline{\lambda}g.\, K_\bullet \mathbin{\underline{@}} V \mathbin{\underline{@}} G_k$. The body of the continuation part $\underline{\lambda}x.\, \underline{\lambda}g.\, k_{id} \mathbin{\underline{@}} x \mathbin{\underline{@}} g$ has the form $K_\bullet \mathbin{\underline{@}} V \mathbin{\underline{@}} G_g$. The annotation of the body becomes $\bullet$, because it returns a result to k_{id}.

Case 3: $\Delta = \bullet$**,** $\Theta = \Delta$
The term is delimited, and the meta-continuation is active. In this case, there are two further cases depending on the annotion of the meta-continuation, whether it is k or $\bullet$.

- If $\Theta = k$ ($\bullet{+}{+}k = k$), the term returns to the initial continuation k after evaluation. Considering the term $\underline{\langle}1\underline{\rangle}$, its CPS translation is $\underline{\lambda}k.\, \underline{\lambda}g.\, k_{id} \mathbin{\underline{@}} 1 \mathbin{\underline{@}} (k :: g)$, which has the form $\underline{\lambda}k.\, \underline{\lambda}g.\, K_\bullet \mathbin{\underline{@}} V \mathbin{\underline{@}} G_k$. In this case, the annotation of the body becomes k, because even if V is passed to a delimited continuation $K_\bullet$, it ultimately passes a result to its meta-continuation G_k that forwards the result to the default k.
- If $\Theta = \bullet$ ($\bullet{+}{+}\bullet = \bullet$), the term returns to the identity continuation k_{id} after evaluation. Such terms appear within continuations arising from a let-expression. Considering the term $\underline{\langle}\underline{\mathsf{let}}\ x = 2\ \underline{\mathsf{in}}\ \underline{\langle}x\underline{\rangle}\underline{\rangle}$, its CPS translation is $\underline{\lambda}k.\, \underline{\lambda}g.\, (\underline{\lambda}x.\, \underline{\lambda}g.\, k_{id} \mathbin{\underline{@}} x \mathbin{\underline{@}} (k_{id} \mathbin{\underline{::}} g)) \mathbin{\underline{@}} 2 \mathbin{\underline{@}} (k \mathbin{\underline{::}} g)$, which has the form $\underline{\lambda}k.\, \underline{\lambda}g.\, K_\bullet \mathbin{\underline{@}} V \mathbin{\underline{@}} G_k$. The body of the continuation part

$\underline{\lambda} x.\, \underline{\lambda} g.\, k_{id} \mathbin{\underline{@}} x \mathbin{\underline{@}} (k_{id} :: g)$ has the form $K_\bullet \mathbin{\underline{@}} V \mathbin{\underline{@}} G_\bullet$. The annotation of the body becomes $\bullet$, because the result is passed to $K_\bullet$ and then to $G_\bullet$ which are both surrounded by `reset`.

Case 4: $\Delta = k$**,** $\Theta = \Delta$

This configuration is unreachable. By definition, a meta-continuation represents the evaluation context outside the innermost enclosing `reset`. Therefore, the meta-continuation cannot be active without an enclosing `reset`.

To summarize these cases, we define the composition Δ++Θ as follows:

$$\Delta \mathbin{++} g = \Delta$$
$$\bullet \mathbin{++} \Delta = \Delta$$

We also present the reduction rules in Fig. 3. The $(\beta.v)$ and $(\beta.let)$ rules are defined to perform simultaneous substitution of arguments, continuations, and meta-continuations. While these rules generally correspond to those in the direct style calculus (Fig. 2), the rules $(let.1)$, $(let.2)$, and $(assoc)$ are omitted because CPS terms are already in a form where subexpressions are explicitly named. For example, converting the term $f \mathbin{\underline{@}} ((h \mathbin{\underline{@}} a) \mathbin{\underline{@}} b)$ to CPS yields the following term:

$$\underline{\lambda} k.\, \underline{\lambda} g.\, h \mathbin{\underline{@}} a \mathbin{\underline{@}} (\underline{\lambda} x.\, \underline{\lambda} g.\, x \mathbin{\underline{@}} b \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} g.\, f \mathbin{\underline{@}} y \mathbin{\underline{@}} k \mathbin{\underline{@}} g) \mathbin{\underline{@}} g) \mathbin{\underline{@}} g$$

The variable x receives the result of $h \mathbin{\underline{@}} a$ to evaluate $x \mathbin{\underline{@}} b$, and then, y receives that result to evaluate $f \mathbin{\underline{@}} y$. This term is also obtained by translating any term in its reduction sequence to A-normal form in Section 3.1.

The type system of $\lambda^*_{cS_0}$ directly corresponds to our Agda formalization using PHOAS [5]. In previous work [16], the continuation constant k and its state annotation Δ were excluded from the standard PHOAS scope and maintained within a dedicated context. A key to our successful mechanization in Agda is extending this technique: we treat the newly introduced meta-continuation constant g and its structural annotation Θ in the exact same way. This customized context resolves the structural mismatch of variable bindings, enabling us to achieve the proof in Agda.

3.3 Kernel Calculus $\boldsymbol{\lambda^{\triangleright}_{c\mathcal{S}_0}}$

The syntax of the kernel calculus, which we dub $\lambda^{\triangleright}_{c\mathcal{S}_0}$, is presented in Fig. 4. We formulate this kernel as a syntactically distinct calculus rather than a mere subset of the DS calculus to make the one-to-one structural correspondence with the CPS calculus $\lambda^*_{c\mathcal{S}_0}$ explicit. The CPS terms $K_\Delta \mathbin{\underline{@}} V \mathbin{\underline{@}} G_\Theta$ and $V \mathbin{\underline{@}} W \mathbin{\underline{@}} K_\Delta \mathbin{\underline{@}} G_\Theta$ correspond to the kernel terms $G_\Theta[K_\Delta[V]]$ and $G_\Theta[W[K_\Delta \mathbin{\underline{@}} V]]$, respectively. In $G_\Theta[K_\Delta[V]]$, if the context K_Δ takes the form $\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ M_\Delta$, this term expands to $G_\Theta[\underline{\mathsf{let}}\ x = V\ \underline{\mathsf{in}}\ M_\Delta]$. Values V, contexts K_Δ, and meta-contexts G_Θ follow this same correspondence. However, active meta-contexts $G_{(\Delta \mathbin{++} \Theta)}$ might initially seem different. As discussed in Section 3.2, evaluating a `reset` pushes the current continuation K_Δ onto the meta-continuation $(K_\Delta :: G_\Theta)$ and initializes the new continuation to the identity. The active meta-context $G_\Theta[K_\Delta[\underline{\langle}[\,]\underline{\rangle}]]$

$$
\begin{array}{rll}
\text{terms } M_{(\Delta +\!+ \Theta)} & ::= & G_\Theta[K_\Delta[V]] \mid G_\Theta[K_\Delta[V \mathbin{\underline{@}} W]] \\
\text{values } V, W & ::= & x \mid \underline{\lambda} x.\, M_k \mid \underline{\mathcal{S}} \mid \underline{\mathcal{S}_0} \\
\text{contexts } J_\Delta, K_\Delta & ::= & {}_{(\Delta = k)}\ [\,]_k \mid {}_{(\Delta = \bullet)}\ [\,]_\bullet \mid \underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ M_\Delta \\
\text{meta-contexts } G_\Theta & ::= & {}_{(\Theta = g)}\ [\,]_g \mid {}_{(\Theta = \Delta)}\ G_\Delta \\
\text{active meta-contexts } G_{(\Delta +\!+ \Theta)} & ::= & G_\Theta[K_\Delta[\underline{\langle}[\,]\underline{\rangle}]]
\end{array}
$$

$$
\begin{array}{lllll}
(\beta.v) & G_\Theta[K_\Delta[(\underline{\lambda} x.\, M_k) \mathbin{\underline{@}} V]] & \longrightarrow M_k[V/x, K_\Delta/[\,]_k, G_\Theta/[\,]_g] & & \\
(\eta.v) & \underline{\lambda} x.\, (V \mathbin{\underline{@}} x) & \longrightarrow V & & \text{if } x \notin fv(V) \\
(\beta.let) & G_\Theta[\underline{\mathsf{let}}\ x = V\ \underline{\mathsf{in}}\ M_\Delta] & \longrightarrow M_\Delta[V/x, G_\Theta/[\,]_g] & & \\
(\eta.let) & \underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ K_\Delta[x] & \longrightarrow K_\Delta & & \text{if } x \notin fv(K_\Delta) \\
(\beta.\mathcal{S}) & G_\Delta[J_\bullet[\underline{\mathcal{S}} \mathbin{\underline{@}} W]] & \longrightarrow G_\Delta[[W \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\langle} J_\bullet[y] \underline{\rangle})]_\bullet] & & \text{if } y \notin fv(J_\Delta) \\
(\beta.\mathcal{S}_0) & G_\Theta[K_\Delta[\underline{\langle} J_\bullet[\underline{\mathcal{S}_0} \mathbin{\underline{@}} W] \underline{\rangle}]] & \longrightarrow G_\Theta[K_\Delta[W \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\langle} J_\bullet[y] \underline{\rangle})]] & & \text{if } y \notin fv(J_\Delta) \\
(\beta.\mathcal{R}) & G_\Theta[K_\Delta[\underline{\langle} V \underline{\rangle}]] & \longrightarrow G_\Theta[K_\Delta[V]] & &
\end{array}
$$

Fig. 4. Syntax of $\lambda^{\triangleright}_{c\mathcal{S}_0}$

reconstructs this stacked continuation: the context K_Δ represents the pushed continuation, enclosing the `reset` boundary which marks the start of the new identity continuation.

The reduction rules of $\lambda^{\triangleright}_{c\mathcal{S}_0}$ maintain a one-to-one correspondence with those of $\lambda^{*}_{c\mathcal{S}_0}$. Similar to the CPS calculus, the rules $(let.1)$, $(let.2)$, and $(assoc)$ are omitted because kernel terms are in A-normal form. For example, converting the term $f \mathbin{\underline{@}} ((h \mathbin{\underline{@}} a) \mathbin{\underline{@}} b)$ to the kernel calculus yields the following term, which directly corresponds its CPS counterpart shown in Section 3.2:

$$
\begin{array}{ll}
\text{Kernel:} & [\underline{\mathsf{let}}\ x = h \mathbin{\underline{@}} a\ \underline{\mathsf{in}}\ ([\underline{\mathsf{let}}\ y = x \mathbin{\underline{@}} b\ \underline{\mathsf{in}}\ ([[f \mathbin{\underline{@}} y]_k]_g)]_g)]_g \\
\text{CPS:} & \underline{\lambda} k.\, \underline{\lambda} g.\, h \mathbin{\underline{@}} a \mathbin{\underline{@}} (\underline{\lambda} x.\, \underline{\lambda} g.\, x \mathbin{\underline{@}} b \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} g.\, f \mathbin{\underline{@}} y \mathbin{\underline{@}} k \mathbin{\underline{@}} g) \mathbin{\underline{@}} g) \mathbin{\underline{@}} g
\end{array}
$$

In $(\beta.\mathcal{S})$, the active meta-context G_Δ implicitly subsumes the delimiting `reset`, naturally preserving it during reduction without requiring it to be explicitly written in the rule. Conversely, $(\beta.\mathcal{S}_0)$ structurally decomposes the overall meta-context into its outer component G_Θ and the innermost continuation K_Δ to explicitly expose the delimiting `reset`, which is subsequently discarded so that the captured continuation and the value W can compose directly without an intervening delimiter.

In the original kernel calculus by Sabry and Wadler [21], the $(\beta.v)$ rule requires K_Δ to be explicitly *maximal*. Consider the term $\underline{\mathsf{let}}\ x = (\underline{\lambda} y.\, M_k) \mathbin{\underline{@}} 1\ \underline{\mathsf{in}}\ x$ as an example. Its subterm $(\underline{\lambda} y.\, M_k) \mathbin{\underline{@}} 1$ must be evaluated under the maximal context $\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ x$, rather than the empty context $[\,]_\Delta$. In our framework, however, this additional condition is unnecessary. Because we treat our terms and contexts strictly as abstract syntax trees (ASTs), evaluating the subterm under the empty context would inherently break the syntax of the outer part.

The typing system of $\lambda^{\triangleright}_{c\mathcal{S}_0}$ is a direct adaptation of the annotated typing system used in the CPS calculus to the kernel constructs.

$$
\begin{aligned}
M^* &= \underline{\lambda} k.\, \underline{\lambda} g.\, (M : k : g) \quad (k, g \notin fv(M)) \\
x^\dagger &= x \\
(\underline{\lambda} x.\, M)^\dagger &= \underline{\lambda} x.\, M^* \\
\underline{\mathcal{S}}^\dagger &= \underline{\lambda} w.\, \underline{\lambda} j.\, \underline{\lambda} g.\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k.\, \underline{\lambda} g.\, j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :: g)) \mathbin{\underline{@}} k_{id} \mathbin{\underline{@}} g \\
\underline{\mathcal{S}_0}^\dagger &= \underline{\lambda} w.\, \underline{\lambda} j.\, \underline{\lambda} (k_0 :: g).\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k.\, \underline{\lambda} g.\, j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :: g)) \mathbin{\underline{@}} k_0 \mathbin{\underline{@}} g \\
V : K : G &= K \mathbin{\underline{@}} V^\dagger \mathbin{\underline{@}} G \\
(V \mathbin{\underline{@}} W) : K : G &= V^\dagger \mathbin{\underline{@}} W^\dagger \mathbin{\underline{@}} K \mathbin{\underline{@}} G \\
(V \mathbin{\underline{@}} Q) : K : G &= Q : (\underline{\lambda} y.\, \underline{\lambda} g.\, V^\dagger \mathbin{\underline{@}} y \mathbin{\underline{@}} K \mathbin{\underline{@}} g) : G \quad \text{if } y \notin (fv(V) \cup fv(K)) \\
(P \mathbin{\underline{@}} W) : K : G &= P : (\underline{\lambda} x.\, \underline{\lambda} g.\, x \mathbin{\underline{@}} W^\dagger \mathbin{\underline{@}} K \mathbin{\underline{@}} g) : G \quad \text{if } x \notin (fv(W) \cup fv(K)) \\
(P \mathbin{\underline{@}} Q) : K : G &= P : (\underline{\lambda} x.\, \underline{\lambda} g.\, (Q : (\underline{\lambda} y.\, \underline{\lambda} g.\, x \mathbin{\underline{@}} y \mathbin{\underline{@}} K \mathbin{\underline{@}} g) : g)) : G \\
&\qquad \text{if } x \notin (fv(Q) \cup fv(K)),\ y \notin fv(K) \\
(\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N) : K : G &= M : (\underline{\lambda} x.\, \underline{\lambda} g.\, (N : K : g)) : G \qquad \text{if } x \notin fv(K) \\
\underline{\langle} M \underline{\rangle} : K : G &= M : k_{id} : (K \mathbin{\underline{::}} G)
\end{aligned}
$$

Fig. 5. Colon Translation

4 Derivation of the Colon Translation

In this section, we briefly describe the derivation of a colon translation [20], which is suitable for proving reflection due to its compact output, from a naive, interpreter-based CPS translation by following the systematic derivation method proposed by Danvy and Nielsen [10]. Notably, we demonstrate that it scales to a 2CPS setting with meta-continuations. Full details and figures of the derivation are in Appendix A.

Our derivation begins with a naive CPS translation based on the operational interpreter for four delimited-control operators proposed by Ishio and Asai [17]. It explicitly passes a continuation k and a meta-continuation g to accommodate both `shift` and `shift0`. However, the generated terms are highly complex due to numerous administrative redexes.

To resolve this issue, we adopt a refined CPS translation by Danvy and Filinski [9]. This translation introduces static constructs (evaluated at translation time) and dynamic constructs (which remain in the output). By evaluating static terms at translation time, we can avoid generating administrative redexes.

While this two-level translation is clear, it relies on mutual recursion and requires simultaneously considering the translation and static execution, which complicates inductive proofs. Therefore, we refine it into the colon translation [20] through two steps: (1) we separate the translation into distinct functions for trivial terms (values) and serious terms (non-values), and (2) we eliminate the static execution via case analysis when translating applications, based on whether subterms are values or non-values.

This final formulation (Fig. 5) corresponds to the well-known colon translation, but supports both `shift` and `shift0` and incorporates meta-continuations. Hereafter, we write the term translation as $M : K : G$ and the value translation as $V^\dagger$.

5 Reflection

In this section, we establish the reflection between the DS and CPS calculi. Following the architecture outlined in Fig. 1, our proof factors through the intermediate kernel calculus: we first establish an order isomorphism between the kernel and CPS calculi, then a reflection between the DS and kernel calculi, and finally compose them to yield the main result.

While the conditions in Section 2.1 formulate soundness (3) and completeness (4) using single-step reductions, we state the following theorems using multi-step reductions (i.e., reflexive and transitive closures) to align directly with our mechanization—a straightforward generalization since multi-step preservation naturally follows from single-step preservation.

All proofs are fully mechanized in Agda. Since most of them proceed straightforwardly by mutual induction on either the structure of terms, values, contexts, and meta-contexts, or the reduction, we omit the routine steps. Instead, we focus on highlighting the crucial lemmas and non-trivial steps essential to the proof.

5.1 Order Isomorphism between kernel and CPS calculi

The order isomorphism is established through a kernel CPS translation $\star : \lambda^{\triangleright}_{c\mathcal{S}_0} \to \lambda^{*}_{c\mathcal{S}_0}$ and its inverse $\Diamond : \lambda^{*}_{c\mathcal{S}_0} \to \lambda^{\triangleright}_{c\mathcal{S}_0}$. Because the kernel calculus is carefully designed to be in one-to-one correspondence to the CPS calculus, the two translations simply adapt the syntax between the two calculi (See Appendix B). We present the order isomorphism in the following theorem.

Theorem 1 (Order Isomorphism between kernel and CPS calculi). *The following properties hold:*

1. *For all terms* $M_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$, $M_\Delta^{\star\Diamond} \equiv_{\lambda^{\triangleright}_{c\mathcal{S}_0}} M_\Delta$
2. *For all roots* $R \in \lambda^{*}_{c\mathcal{S}_0}$, $R^{\Diamond\star} \equiv_{\lambda^{*}_{c\mathcal{S}_0}} R$
3. *For all terms* $M_\Delta, M'_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$, $M_\Delta \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{\triangleright}_{c\mathcal{S}_0}} M'_\Delta$ *implies* $M_\Delta^{\star} \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{*}_{c\mathcal{S}_0}} (M'_\Delta)^{\star}$
4. *For all roots* $R, R' \in \lambda^{*}_{c\mathcal{S}_0}$, $R \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{*}_{c\mathcal{S}_0}} R'$ *implies* $R^{\Diamond} \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{\triangleright}_{c\mathcal{S}_0}} (R')^{\Diamond}$

While the proof naturally succeeds due to the strict one-to-one correspondence between the calculi, its formalization in Agda is non-trivial owing to variable bindings. The kernel abstraction $\underline{\lambda}x.\, M_k$ binds only x, whereas the corresponding CPS abstraction $\underline{\lambda}x.\, \underline{\lambda}k.\, \underline{\lambda}g.\, M_k$ binds the variables x, k, and g. This discrepancy in bound variables leads to a failure in relating the two M_k terms if we naively represent all variables using PHOAS. To resolve this structural mismatch, we treat the variables k and g in the CPS terms as distinguished constants rather than locally bound variables, which allows us to align the binding structures of both calculi.

Furthermore, the mechanized proof requires the axiom of functional extensionality. Under the PHOAS encoding, functions are represented as meta-level functions in Agda. We need to ensure the equivalence of a λ-abstraction before and after CPS and DS translations. Hence, we cannot directly show their equivalence, and we need to use the axiom of extensionality, which states that if $f(x) = g(x)$ for all arguments x, then $f = g$.

5.2 Reflection between DS and kernel calculi

The reflection is proven between the A-normalization translation $\triangleright : \lambda_{c\mathcal{S}_0} \to \lambda^{\triangleright}_{c\mathcal{S}_0}$ and the embedding $\triangleleft : \lambda^{\triangleright}_{c\mathcal{S}_0} \to \lambda_{c\mathcal{S}_0}$. The formal rules are given in Appendix C. The translation $\triangleright$ is derived by replacing the CPS terms in our colon translation (Fig. 5) with the corresponding kernel terms. The embedding $\triangleleft$ might be expected to be straightforward because the kernel calculus is a syntactic subset of the DS calculus. However, it is not an identity function, since the kernel term, unlike the DS term, is constructed by context and meta-context. The crucial insight is that the DS contexts K and G are designed to naturally embed the explicit contexts K_Δ and meta-contexts G_Θ of the kernel calculus.

We state the reflection between the DS and kernel calculi as follows.

Theorem 2 (Reflection between DS and kernel calculi). *The following properties hold:*

1. *For all term* $M \in \lambda_{c\mathcal{S}_0}$, $M \longrightarrow\!\!\!\!\!\rightarrow_{\lambda_{c\mathcal{S}_0}} M^{\triangleright\triangleleft}$
2. *For all term* $N_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$, $N_\Delta^{\triangleleft\triangleright} \equiv_{\lambda^{\triangleright}_{c\mathcal{S}_0}} N_\Delta$
3. *For all terms* $M, M' \in \lambda_{c\mathcal{S}_0}$, $M \longrightarrow\!\!\!\!\!\rightarrow_{\lambda_{c\mathcal{S}_0}} M'$ *implies* $M^{\triangleright} \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{\triangleright}_{c\mathcal{S}_0}} (M')^{\triangleright}$
4. *For all terms* $N_\Delta, N'_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$, $N_\Delta \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{\triangleright}_{c\mathcal{S}_0}} N'_\Delta$ *implies* $N_\Delta^{\triangleleft} \longrightarrow\!\!\!\!\!\rightarrow_{\lambda_{c\mathcal{S}_0}} (N'_\Delta)^{\triangleleft}$

In the following, we highlight two crucial lemmas required to establish reduction preservation (soundness) and discuss the issues arising from using PHOAS.

First, we show the existence lemma of evaluation contexts for describing the behavior of `shift` and `shift0`.

Lemma 1 (Existence of a $\lambda^{\triangleright}_{c\mathcal{S}_0}$ context for each $\lambda_{c\mathcal{S}_0}$ context). *For all context* $J \in \lambda_{c\mathcal{S}_0}$ *and context* $K_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$, *there exists a context* $\hat{J}_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$ *such that the following two hold.*

- *For all nonvalue* $P \in \lambda_{c\mathcal{S}_0}$ *and meta-context* $G_\Theta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$,
  $J[P] : K_\Delta : G_\Theta = P : \hat{J}_\Delta : G_\Theta$
- *For all value* $V \in \lambda_{c\mathcal{S}_0}$ *and meta-context* $G_\Theta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$,
  $V : \hat{J}_\Delta : G_\Theta \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{\triangleright}_{c\mathcal{S}_0}} J[V] : K_\Delta : G_\Theta$

This lemma extends the one from the previous work [3] by incorporating meta-contexts into the framework. Let $J[P]$ denote a pure context J enclosing an operator P, where P represents an application of either `shift` or `shift0`. The first part of the lemma states that the pure context J can be captured and pushed into the evaluation context environment as $\hat{J}_\Delta$. The second part ensures that evaluating a value V under this captured continuation $\hat{J}_\Delta$ coincides with the evaluation of V under the original context J. Since the meta-context remains passive throughout this interaction, this property can be established using an arbitrary meta-context.

Second, we introduce the substitution lemma for contexts.

Lemma 2 (Substitution Lemma for Contexts). *The following properties hold:*

- *For all terms* $M \in \lambda_{c\mathcal{S}_0}$*, contexts* $K_\bullet, J_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$ *and meta-context* $G_k \in \lambda^{\triangleright}_{c\mathcal{S}_0}$*,* $M : K_\bullet : (G_k[J_\Delta/k]) \equiv_{\lambda^{\triangleright}_{c\mathcal{S}_0}} (M : K_\bullet : G_k)[J_\Delta/k]$
- *For all terms* $M \in \lambda_{c\mathcal{S}_0}$*, contexts* $K_k, J_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$ *and meta-context* $G_g \in \lambda^{\triangleright}_{c\mathcal{S}_0}$*,* $M : (K_k[J_\Delta/k)] : G_g \equiv_{\lambda^{\triangleright}_{c\mathcal{S}_0}} (M : K_k : G_g)[J_\Delta/k]$

In our framework, the context variable k can appear in either the context or the meta-context. Therefore, the proof requires a case analysis on the valid configurations of Δ and Θ satisfying $\Delta{+}{+}\Theta = k$, which naturally restricts the possibilities to two cases: $(\Delta = \bullet, \Theta = k)$ and $(\Delta = k, \Theta = g)$. The proof proceeds by structural induction on M. When proving the second proposition, a naive induction encounters a difficulty in the case for $\underline{\langle M \rangle}$, because the colon translation for $\underline{\langle M \rangle}$ transfers the pure context into the meta-context. We overcome this by combining both propositions into a single lemma, allowing us to use the first proposition—which handles occurrences of k in the meta-context—as an inductive hypothesis for the $\underline{\langle M \rangle}$ case.

Finally, we discuss the issues arising from using PHOAS. Similar to the previous section, the axiom of functional extensionality is required. Moreover, the proof of 4 (completeness) relies on the following postulate.

Postulate 1 *For all terms* $M_\Delta \in \lambda^{\triangleright}_{c\mathcal{S}_0}$*,* $M_\Delta[x/x] \equiv_{\lambda^{\triangleright}_{c\mathcal{S}_0}} M_\Delta$

In a traditional setting, this property would be trivially true, as it simply states that replacing the variable x in M_Δ with x itself results in M_Δ. Under the PHOAS encoding, however, this proposition cannot be proved directly. This issue is also noted by Chlipala himself, who proposed PHOAS [5]. Nevertheless, when a term M_Δ is given concretely, this property can be verified straightforwardly in Agda, making our postulate a reasonable and practical choice.

5.3 Main Results via Composition

Finally, we establish the direct reflection between DS and CPS calculi in the following corollary.

Corollary 1 (Reflection between DS and CPS Calculi). *The following properties hold:*

1. *For all terms* $M \in \lambda_{c\mathcal{S}_0}$*,* $M \longrightarrow\!\!\!\!\!\rightarrow_{\lambda_{c\mathcal{S}_0}} M^{*\#}$
2. *For all roots* $R \in \lambda^{*}_{c\mathcal{S}_0}$*,* $R^{\#*} \equiv_{\lambda^{*}_{c\mathcal{S}_0}} R$
3. *For all terms* $M, M' \in \lambda_{c\mathcal{S}_0}$*,* $M \longrightarrow\!\!\!\!\!\rightarrow_{\lambda_{c\mathcal{S}_0}} M'$ *implies* $M^* \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{*}_{c\mathcal{S}_0}} (M')^*$
4. *For all roots* $R, R' \in \lambda^{*}_{c\mathcal{S}_0}$*,* $R \longrightarrow\!\!\!\!\!\rightarrow_{\lambda^{*}_{c\mathcal{S}_0}} R'$ *implies* $R^{\#} \longrightarrow\!\!\!\!\!\rightarrow_{\lambda_{c\mathcal{S}_0}} (R')^{\#}$

6 Related Work

Colon Translation. The colon translation for the λ-calculus was first proposed by Plotkin [20], and later systematically derived from evaluation contexts by Danvy and Nielsen [10]. For calculi with `shift0`, Biernacki et al. [3] defined a

different colon translation for `shift0/dollar`. The derivation of this translation, however, was not explicitly discussed in their work. In this paper, we show that the colon translation for `shift0` can be systematically derived from a 2CPS interpreter. This reveals that even complex translation rules involving meta-continuations naturally emerge from the structure of evaluation contexts.

Reflection. Sabry and Wadler [21] proved a reflection for the CPS translation of λ-calculus, and Biernacki et al. [3] extended this to establish a reflection for a system with `shift/reset`. Recently, our group (Honda et al. [16]) advanced this line of work to a typed setting and formalized the proof in Agda.

However, integrating the `shift0` operator into these existing frameworks has remained a challenge. Biernacki et al. [4] noted that the approach for `shift` could not be directly applied to `shift0`, leading them to develop a distinct framework specifically for `shift0`. Furthermore, while we inherited the fundamental approach from Honda et al., their framework relies on a standard CPS translation, which is structurally insufficient to accommodate `shift0`. Since the behavioral difference between `shift` and `shift0` lies in the ability to capture meta-continuations, we expanded the foundation to include meta-continuations. This allowed us to prove the reflection for both operators within the exact same framework.

For algebraic effect and handlers, inspired by the reflection for `shift0/dollar`, Cong and Asai [6] attempted to prove a reflection for algebraic effects and deep handlers, but encountered a technical problem in completing the proof. They point out that the reason for this is that the CPS translation they adopted from Hillerström et al. [15] loses the distinction between pure evaluation contexts and the return clause of handlers. We expect to overcome this limitation by applying the derivation methodology developed in this paper.

7 Conclusion

In this paper, we have established the reflection for a typed direct-style language featuring the delimited control operators `shift`, `shift0`, and `reset`, and formalized the entire development in the Agda proof assistant. For the target CPS calculus, we followed the previous work by introducing Δ to classify continuations, and additionally introduced Θ to categorize meta-continuations. This dual annotation captures how the syntactic structures of terms vary depending on the combination of both types of continuations. In defining our CPS transformation, we employed the systematic derivation of a colon translation from a reference interpreter, demonstrating its effectiveness even for complex control structures involving meta-continuations.

Future work includes establishing reflection for algebraic effects and handlers. Asai and Fujii proposed an interpreter for them via meta-continuations [2]. We expect that deriving a colon translation from their interpreter using our approach would be a promising step toward establishing reflection for deep handlers.

Appendix

A Derivation of the Colon Translation

In this section, we present the detailed derivation of the colon translation.

A.1 A Naive Interpreter-based CPS translation

$$
\begin{aligned}
[\![x]\!] &= \underline{\lambda}k.\,\underline{\lambda}g.\,k\,\underline{@}\,x\,\underline{@}\,g \\
[\![\underline{\lambda}x.\,M]\!] &= \underline{\lambda}k.\,\underline{\lambda}g.\,k\,\underline{@}\,(\underline{\lambda}x.\,\underline{\lambda}k'.\,\underline{\lambda}g'.\,[\![M]\!]\,\underline{@}\,k'\,\underline{@}\,g')\,\underline{@}\,g \\
[\![M_1\,\underline{@}\,M_2]\!] &= \underline{\lambda}k.\,\underline{\lambda}g.\,[\![M_1]\!]\,\underline{@}\,(\underline{\lambda}v_1.\,\underline{\lambda}g_1.\,[\![M_2]\!]\,\underline{@}\,(\underline{\lambda}v_2.\,\underline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\underline{@}\,g_1)\,\underline{@}\,g \\
[\![\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N]\!] &= \underline{\lambda}k.\,\underline{\lambda}g.\,[\![M]\!]\,\underline{@}\,(\underline{\lambda}x.\,\underline{\lambda}g'.\,[\![N]\!]\,\underline{@}\,k\,\underline{@}\,g')\,\underline{@}\,g \\
[\![\underline{\mathcal{S}}]\!] &= \underline{\lambda}k.\,\underline{\lambda}g.\,k\,\underline{@}\,(\underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}g'.\,w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_{id}\,\underline{@}\,g')\,\underline{@}\,g \\
[\![\underline{\mathcal{S}_0}]\!] &= \underline{\lambda}k.\,\underline{\lambda}g.\,k\,\underline{@}\,(\underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}(k_0\,\underline{::}\,g').\\
&\qquad\qquad w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_0\,\underline{@}\,g')\,\underline{@}\,g \\
[\![\underline{\langle}M\underline{\rangle}]\!] &= \underline{\lambda}k.\,\underline{\lambda}g.\,[\![M]\!]\,\underline{@}\,k_{id}\,\underline{@}\,(k\,\underline{::}\,g)
\end{aligned}
$$

Fig. 6. An Interpreter-based CPS Translation

A naive CPS translation (Fig. 6) is derived from the operational interpreter for four delimited-control operators proposed by Ishio and Asai [17]. It is constructed by eliminating trails and retaining only meta-continuations, taking a continuation k and a meta-continuation g as arguments. All generated constructs are decorated with underlines ($\underline{\lambda}, \underline{@}$), which represent that the expressions are dynamic, namely, they remain in the output.

A.2 A refined CPS Translation

Since the naive CPS translation produces complex terms with many administrative redexes, we can perform these administrative resuction at translation time.

Here, the newly introduced overlined symbols ($\overline{\lambda}, \overline{@}$) represent that these static operations are evaluated during the translation. In Fig. 7, several dynamic constructs ($\underline{\lambda}, \underline{@}$) appearing in Fig. 6 are replaced with static ones ($\overline{\lambda}, \overline{@}$). In addition, some continuations are turned to their two-level η-expanded forms [9]. This translation is defined via mutual recursion using two distinct functions: a static function $[\![\cdot]\!]$ and a dynamic function $[\![\cdot]\!]'$. In this way, we can separate the static and dynamic continuations, which allows us to avoid administrative redexes in the output.

A.3 Colon Translation

While the two-level translation serves as a clear, interpreter-based specification, it still has two main problems. First, it requires simultaneously considering

$$
\begin{aligned}
k_{id} \mathbin{\underline{@}} v \mathbin{\underline{@}} [\,] &= v \\
k_{id} \mathbin{\underline{@}} v \mathbin{\underline{@}} (\kappa \mathbin{::} g) &= \kappa \mathbin{\underline{@}} v \mathbin{\underline{@}} g
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!] &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\overline{@}} x \mathbin{\overline{@}} g \\
[\![\underline{\lambda} x.\, M]\!] &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\overline{@}} (\underline{\lambda} x.\, \underline{\lambda} k.\, \underline{\lambda} g.\, [\![M]\!]' \mathbin{\overline{@}} k \mathbin{\overline{@}} g) \mathbin{\overline{@}} g \\
[\![M_1 \mathbin{\underline{@}} M_2]\!] &= \overline{\lambda} k.\, \overline{\lambda} g.\, [\![M_1]\!] \mathbin{\overline{@}} (\overline{\lambda} v_1.\, \overline{\lambda} g_1.\, \\
&\qquad [\![M_2]\!] \mathbin{\overline{@}} (\overline{\lambda} v_2.\, \overline{\lambda} g_2.\, v_1 \mathbin{\underline{@}} v_2 \mathbin{\underline{@}} (\underline{\lambda} a.\, \underline{\lambda} g'.\, k \mathbin{\overline{@}} a \mathbin{\overline{@}} g') \mathbin{\underline{@}} g_2) \mathbin{\overline{@}} g_1) \mathbin{\overline{@}} g \\
[\![\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N]\!] &= \overline{\lambda} k.\, \overline{\lambda} g.\, [\![M]\!]' \mathbin{\overline{@}} (\underline{\lambda} x.\, \underline{\lambda} g'.\, [\![N]\!]' \mathbin{\overline{@}} (\underline{\lambda} a.\, \underline{\lambda} g''.\, k \mathbin{\overline{@}} a \mathbin{\overline{@}} g'') \mathbin{\overline{@}} g') \mathbin{\overline{@}} g \\
[\![\underline{\mathcal{S}}]\!] &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\overline{@}} (\underline{\lambda} w.\, \underline{\lambda} k'.\, \underline{\lambda} g'.\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k''.\, \underline{\lambda} g''.\, k' \mathbin{\underline{@}} y \mathbin{\underline{@}} (k'' \mathbin{::} g'')) \mathbin{\underline{@}} k_{id} \mathbin{\underline{@}} g') \mathbin{\overline{@}} g \\
[\![\underline{\mathcal{S}_0}]\!] &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\overline{@}} (\underline{\lambda} w.\, \underline{\lambda} k'.\, \underline{\lambda} (k_0 \mathbin{::} g').\, \\
&\qquad w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k''.\, \underline{\lambda} g''.\, k' \mathbin{\underline{@}} y \mathbin{\underline{@}} (k'' \mathbin{::} g'')) \mathbin{\underline{@}} k_0 \mathbin{\underline{@}} g') \mathbin{\overline{@}} g \\
[\![\underline{\langle} M \underline{\rangle}]\!] &= \overline{\lambda} k.\, \overline{\lambda} g.\, [\![M]\!]' \mathbin{\overline{@}} k_{id} \mathbin{\overline{@}} ((\underline{\lambda} a.\, \underline{\lambda} g'.\, k \mathbin{\overline{@}} a \mathbin{\overline{@}} g') \mathbin{::} g)
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]' &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\underline{@}} x \mathbin{\underline{@}} g \\
[\![\underline{\lambda} x.\, M]\!]' &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\underline{@}} (\underline{\lambda} x.\, \underline{\lambda} k.\, \underline{\lambda} g.\, [\![M]\!]' \mathbin{\overline{@}} k \mathbin{\overline{@}} g) \mathbin{\underline{@}} g \\
[\![M_1 \mathbin{\underline{@}} M_2]\!]' &= \overline{\lambda} k.\, \overline{\lambda} g.\, [\![M_1]\!] \mathbin{\overline{@}} (\overline{\lambda} v_1.\, \overline{\lambda} g_1.\, [\![M_2]\!] \mathbin{\overline{@}} (\overline{\lambda} v_2.\, \overline{\lambda} g_2.\, v_1 \mathbin{\underline{@}} v_2 \mathbin{\underline{@}} k \mathbin{\underline{@}} g_2) \mathbin{\overline{@}} g_1) \mathbin{\overline{@}} g \\
[\![\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N]\!]' &= \overline{\lambda} k.\, \overline{\lambda} g.\, [\![M]\!]' \mathbin{\overline{@}} (\underline{\lambda} x.\, \underline{\lambda} g'.\, [\![N]\!]' \mathbin{\overline{@}} k \mathbin{\overline{@}} g') \mathbin{\overline{@}} g \\
[\![\underline{\mathcal{S}}]\!]' &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\underline{@}} (\underline{\lambda} w.\, \underline{\lambda} k'.\, \underline{\lambda} g'.\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k''.\, \underline{\lambda} g''.\, k' \mathbin{\underline{@}} y \mathbin{\underline{@}} (k'' \mathbin{::} g'')) \mathbin{\underline{@}} k_{id} \mathbin{\underline{@}} g') \mathbin{\underline{@}} g \\
[\![\underline{\mathcal{S}_0}]\!]' &= \overline{\lambda} k.\, \overline{\lambda} g.\, k \mathbin{\underline{@}} (\underline{\lambda} w.\, \underline{\lambda} k'.\, \underline{\lambda} (k_0 \mathbin{::} g').\, \\
&\qquad w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k''.\, \underline{\lambda} g''.\, k' \mathbin{\underline{@}} y \mathbin{\underline{@}} (k'' \mathbin{::} g'')) \mathbin{\underline{@}} k_0 \mathbin{\underline{@}} g') \mathbin{\underline{@}} g \\
[\![\underline{\langle} M \underline{\rangle}]\!]' &= \overline{\lambda} k.\, \overline{\lambda} g.\, [\![M]\!]' \mathbin{\overline{@}} k_{id} \mathbin{\overline{@}} (k \mathbin{::} g)
\end{aligned}
$$

Fig. 7. Two-Level CPS Translation

the translation and static execution. Second, the mutual recursion unnecessarily complicates inductive proofs.

We thus introduce the colon translation [20]. The systematic derivation method by Danvy and Nielsen [10] was originally designed for standard CPS. However, we can extend this framework to a 2CPS setting with meta-continuations. Following their framework, our colon translation is obtained from Fig. 7 through the following two steps: (1) we separate the translation into two distinct functions, one for trivial terms ($[\![\cdot]\!]_v, [\![\cdot]\!]'_v$) and another for serious terms ($[\![\cdot]\!]_s, [\![\cdot]\!]'_s$), and (2) we eliminate the static execution part ($[\![\cdot]\!]_v, [\![\cdot]\!]_s$) so that we only use the dynamic translation part ($[\![\cdot]\!]'_v, [\![\cdot]\!]'_s$).

First, we classify the terms into trivial terms (values) and serious terms (non-values). This yields the translation in Fig. 8. Note that the definitions of $[\![\cdot]\!]_v$ and $[\![\cdot]\!]'_v$ are completely identical.

Next, we eliminate the static function $[\![\cdot]\!]$, which is used in function applications. We perform a case analysis based on whether the subterms are values or non-values, as detailed in Fig. 9. When both subterms are values, the static translation layer can be immediately eliminated by exploiting the identity $[\![\cdot]\!]_v = [\![\cdot]\!]'_v$ obtained in the first step. For the other cases involving non-values, however, the static translation functions still remain in the intermediate terms. We resolve this issue by eta-expanding the static continuations into dynamic lambda abstractions. This enables us to completely replace the remaining static functions with their dynamic counterparts, thereby eliminating the static execution layer. The resulting translation is shown in Fig. 10. This translation is commonly referred to as the colon translation. As shown in Fig. 5, we write $[\![M]\!]'_s \mathbin{\overline{@}} K \mathbin{\overline{@}} G$ as $M : K : G$ and $[\![V]\!]'_v$ as $V^\dagger$.

$$
\begin{aligned}
[\![v]\!] &= \overline{\lambda}k.\,\overline{\lambda}g.\,k\,\overline{@}\,[\![v]\!]_v\,\overline{@}\,g \\
[\![s]\!] &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![s]\!]_s\,\overline{@}\,k\,\overline{@}\,g \\
[\![x]\!]_v &= x \\
[\![\underline{\lambda}x.\,M]\!]_v &= \underline{\lambda}x.\,\underline{\lambda}k.\,\underline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,k\,\overline{@}\,g \\
[\![M_1\,\underline{@}\,M_2]\!]_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M_1]\!]\,\overline{@}\,(\overline{\lambda}v_1.\,\overline{\lambda}g_1.\\
&\qquad [\![M_2]\!]\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,(\underline{\lambda}a.\,\underline{\lambda}g'.\,k\,\overline{@}\,a\,\overline{@}\,g')\,\underline{@}\,g_2)\,\overline{@}\,g_1)\,\overline{@}\,g \\
[\![\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N]\!]_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,(\underline{\lambda}x.\,\underline{\lambda}g'.\,[\![N]\!]'\,\overline{@}\,(\underline{\lambda}a.\,\underline{\lambda}g''.\,k\,\overline{@}\,a\,\overline{@}\,g'')\,\overline{@}\,g')\,\overline{@}\,g \\
[\![\underline{\mathcal{S}}]\!]_v &= \underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}g'.\,w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_{id}\,\underline{@}\,g' \\
[\![\underline{\mathcal{S}_0}]\!]_v &= \underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}(k_0\,\underline{::}\,g').\,w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_0\,\underline{@}\,g' \\
[\![\underline{\langle M \rangle}]\!]_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,k_{id}\,\overline{@}\,((\underline{\lambda}a.\,\underline{\lambda}g'.\,k\,\overline{@}\,a\,\overline{@}\,g')\,\underline{::}\,g) \\
[\![v]\!]' &= \overline{\lambda}k.\,\overline{\lambda}g.\,k\,\underline{@}\,[\![v]\!]'_v\,\underline{@}\,g \\
[\![s]\!]' &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![s]\!]'_s\,\overline{@}\,k\,\overline{@}\,g \\
[\![x]\!]'_v &= x \\
[\![\underline{\lambda}x.\,M]\!]'_v &= \underline{\lambda}x.\,\underline{\lambda}k.\,\underline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,k\,\overline{@}\,g \\
[\![M_1\,\underline{@}\,M_2]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M_1]\!]\,\overline{@}\,(\overline{\lambda}v_1.\,\overline{\lambda}g_1.\,[\![M_2]\!]\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g_1)\,\overline{@}\,g \\
[\![\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,(\underline{\lambda}x.\,\underline{\lambda}g'.\,[\![N]\!]'\,\overline{@}\,k\,\overline{@}\,g')\,\overline{@}\,g \\
[\![\underline{\mathcal{S}}]\!]'_v &= \underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}g'.\,w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_{id}\,\underline{@}\,g' \\
[\![\underline{\mathcal{S}_0}]\!]'_v &= \underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}(k_0\,\underline{::}\,g').\,w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_0\,\underline{@}\,g' \\
[\![\underline{\langle M \rangle}]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,k_{id}\,\overline{@}\,(k\,\underline{::}\,g)
\end{aligned}
$$

Fig. 8. Separation of Trivial and Serious Terms

$$
\begin{aligned}
[\![V\,\underline{@}\,W]\!]'_s\,\overline{@}\,k\,\overline{@}\,g &= [\![V]\!]\,\overline{@}\,(\overline{\lambda}v_1.\,\overline{\lambda}g_1.\,[\![W]\!]\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g_1)\,\overline{@}\,g \\
&= (\overline{\lambda}k.\,\overline{\lambda}g.\,k\,\overline{@}\,[\![V]\!]_v\,\overline{@}\,g)\,\overline{@}\,(\overline{\lambda}v_1.\,\overline{\lambda}g_1.\\
&\qquad [\![W]\!]\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g_1)\,\overline{@}\,g \\
&\longrightarrow (\overline{\lambda}v_1.\,\overline{\lambda}g_1.\,[\![W]\!]\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g_1)\,\overline{@}\,[\![V]\!]_v\,\overline{@}\,g \\
&\longrightarrow [\![W]\!]\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,[\![V]\!]_v\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g \\
&= (\overline{\lambda}k.\,\overline{\lambda}g.\,k\,\overline{@}\,[\![W]\!]_v\,\overline{@}\,g)\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,[\![V]\!]_v\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g \\
&\longrightarrow (\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,[\![V]\!]_v\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,[\![W]\!]_v\,\overline{@}\,g \\
&\longrightarrow [\![V]\!]_v\,\underline{@}\,[\![W]\!]_v\,\underline{@}\,k\,\underline{@}\,g \\
&= [\![V]\!]'_v\,\underline{@}\,[\![W]\!]'_v\,\underline{@}\,k\,\underline{@}\,g \\
[\![V\,\underline{@}\,Q]\!]'_s\,\overline{@}\,k\,\overline{@}\,g &= [\![Q]\!]_s\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,[\![V]\!]'_v\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g \\
[\![P\,\underline{@}\,W]\!]'_s\,\overline{@}\,k\,\overline{@}\,g &= [\![P]\!]_s\,\overline{@}\,(\overline{\lambda}v_1.\,\overline{\lambda}g_1.\,v_1\,\underline{@}\,[\![W]\!]'_v\,\underline{@}\,k\,\underline{@}\,g_1)\,\overline{@}\,g \\
[\![P\,\underline{@}\,Q]\!]'_s\,\overline{@}\,k\,\overline{@}\,g &= [\![P]\!]_s\,\overline{@}\,(\overline{\lambda}v_1.\,\overline{\lambda}g_1.\,[\![Q]\!]_s\,\overline{@}\,(\overline{\lambda}v_2.\,\overline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g_1)\,\overline{@}\,g
\end{aligned}
$$

Fig. 9. Case analysis on function application

$$
\begin{aligned}
[\![v]\!]' &= \overline{\lambda}k.\,\overline{\lambda}g.\,k\,\underline{@}\,[\![v]\!]'_v\,\underline{@}\,g \\
[\![s]\!]' &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![s]\!]'_s\,\overline{@}\,k\,\overline{@}\,g \\
[\![x]\!]'_v &= x \\
[\![\underline{\lambda}x.\,M]\!]'_v &= \underline{\lambda}x.\,\underline{\lambda}k.\,\underline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,k\,\overline{@}\,g \\
[\![\underline{\mathcal{S}}]\!]'_v &= \underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}g'.\,w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_{id}\,\underline{@}\,g' \\
[\![\underline{\mathcal{S}_0}]\!]'_v &= \underline{\lambda}w.\,\underline{\lambda}k'.\,\underline{\lambda}(k_0\,\underline{::}\,g').\,w\,\underline{@}\,(\underline{\lambda}y.\,\underline{\lambda}k''.\,\underline{\lambda}g''.\,k'\,\underline{@}\,y\,\underline{@}\,(k''\,\underline{::}\,g''))\,\underline{@}\,k_0\,\underline{@}\,g' \\
[\![V\,\underline{@}\,W]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![V]\!]'_v\,\underline{@}\,[\![W]\!]'_v\,\underline{@}\,k\,\underline{@}\,g \\
[\![V\,\underline{@}\,Q]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![Q]\!]'_s\,\overline{@}\,(\underline{\lambda}v_2.\,\underline{\lambda}g_2.\,[\![V]\!]'_v\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g \\
[\![P\,\underline{@}\,W]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![P]\!]'_s\,\overline{@}\,(\underline{\lambda}v_1.\,\underline{\lambda}g_1.\,v_1\,\underline{@}\,[\![W]\!]'_v\,\underline{@}\,k\,\underline{@}\,g_1)\,\overline{@}\,g \\
[\![P\,\underline{@}\,Q]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![P]\!]'_s\,\overline{@}\,(\underline{\lambda}v_1.\,\underline{\lambda}g_1.\,[\![Q]\!]'_s\,\overline{@}\,(\underline{\lambda}v_2.\,\underline{\lambda}g_2.\,v_1\,\underline{@}\,v_2\,\underline{@}\,k\,\underline{@}\,g_2)\,\overline{@}\,g_1)\,\overline{@}\,g \\
[\![\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,(\underline{\lambda}v.\,\underline{\lambda}g'.\,[\![N]\!]'\,\overline{@}\,k\,\overline{@}\,g')\,\overline{@}\,g \\
[\![\underline{\langle M \rangle}]\!]'_s &= \overline{\lambda}k.\,\overline{\lambda}g.\,[\![M]\!]'\,\overline{@}\,k_{id}\,\overline{@}\,(k\,\underline{::}\,g)
\end{aligned}
$$

Fig. 10. Derived CPS Translation

B Translations between kernel and CPS

We present the translation functions between kernel and CPS calculi.

$$
\begin{aligned}
M^{\star}_{\Delta} &= \underline{\lambda} k.\, \underline{\lambda} g.\, M^{\circ}_{k} \\
x^{\dagger} &= x \\
(\underline{\lambda} x.\, M_k)^{\dagger} &= \underline{\lambda} x.\, \underline{\lambda} k.\, \underline{\lambda} g.\, M^{\circ}_{k} \\
\underline{\mathcal{S}}^{\dagger} &= \underline{\lambda} w.\, \underline{\lambda} j.\, \underline{\lambda} g.\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k.\, \underline{\lambda} g.\, j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :: g)) \mathbin{\underline{@}} k_{id} \mathbin{\underline{@}} g \\
\underline{\mathcal{S}_0}^{\dagger} &= \underline{\lambda} w.\, \underline{\lambda} j.\, \underline{\lambda} (k_0 :: g).\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k.\, \underline{\lambda} g.\, j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :: g)) \mathbin{\underline{@}} k_0 \mathbin{\underline{@}} g \\
G_{\Theta}[K_{\Delta}[V]]^{\circ} &= K^{\ddagger}_{\Delta} \mathbin{\underline{@}} V^{\dagger} \mathbin{\underline{@}} G^{\ddagger\ddagger}_{\Theta} \\
G_{\Theta}[K_{\Delta}[V \mathbin{\underline{@}} W]]^{\circ} &= V^{\dagger} \mathbin{\underline{@}} W^{\dagger} \mathbin{\underline{@}} K^{\ddagger}_{\Delta} \mathbin{\underline{@}} G^{\ddagger\ddagger}_{\Theta} \\
[\,]^{\ddagger}_{k} &= k \\
[\,]^{\ddagger}_{\bullet} &= k_{id} \\
(\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ M_{\Delta})^{\ddagger} &= \underline{\lambda} x.\, \underline{\lambda} g.\, M^{\circ}_{\Delta} \\
[\,]^{\ddagger\ddagger}_{g} &= g \\
G_{\Theta}[K_{\Delta}[\underline{\langle}[\,]\underline{\rangle}]]^{\ddagger\ddagger} &= K^{\ddagger}_{\Delta} :: G^{\ddagger\ddagger}_{\Theta}
\end{aligned}
$$

Fig. 11. CPS Translation ($\star : \lambda^{\triangleright}_{c\mathcal{S}_0} \longrightarrow \lambda^{*}_{c\mathcal{S}_0}$)

Fig. 11 defines the forward translation from $\lambda^{\triangleright}_{c\mathcal{S}_0}$ to $\lambda^{*}_{c\mathcal{S}_0}$, which is formalized in `DSK-CPS.agda`. It relies on four mutually recursive translations (†, ∘, ‡, ‡‡) operating by structural recursion on values, terms, continuations, and meta-continuations, respectively.

$$
\begin{aligned}
(\underline{\lambda} k.\, \underline{\lambda} g.\, M_{\Delta})^{\Diamond} &= M^{\sharp}_{\Delta} \\
x^{\natural} &= x \\
(\underline{\lambda} x.\, \underline{\lambda} k.\, \underline{\lambda} g.\, M_k)^{\natural} &= \underline{\lambda} x.\, M^{\sharp}_{k} \\
(\underline{\lambda} w.\, \underline{\lambda} j.\, \underline{\lambda} g.\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k.\, \underline{\lambda} g.\, j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :: g)) \mathbin{\underline{@}} k_{id} \mathbin{\underline{@}} g)^{\natural} &= \underline{\mathcal{S}} \\
(\underline{\lambda} w.\, \underline{\lambda} j.\, \underline{\lambda} (k_0 :: g).\, w \mathbin{\underline{@}} (\underline{\lambda} y.\, \underline{\lambda} k.\, \underline{\lambda} g.\, j \mathbin{\underline{@}} y \mathbin{\underline{@}} (k :: g)) \mathbin{\underline{@}} k_0 \mathbin{\underline{@}} g)^{\natural} &= \underline{\mathcal{S}_0} \\
(K_{\Delta} \mathbin{\underline{@}} V \mathbin{\underline{@}} G_{\Theta})^{\sharp} &= G^{\flat\flat}_{\Theta}[K^{\flat}_{\Delta}[V^{\natural}]] \\
(V \mathbin{\underline{@}} W \mathbin{\underline{@}} K_{\Delta} \mathbin{\underline{@}} G_{\Theta})^{\sharp} &= G^{\flat\flat}_{\Theta}[K^{\flat}_{\Delta}[V^{\natural} \mathbin{\underline{@}} W^{\natural}]] \\
k^{\flat} &= [\,]_{k} \\
k^{\flat}_{id} &= [\,]_{\bullet} \\
(\underline{\lambda} x.\, \underline{\lambda} g.\, M_{\Delta})^{\flat} &= \underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ M^{\sharp}_{\Delta} \\
g^{\flat\flat} &= [\,]_{g} \\
(K_{\Delta} :: G_{\Theta})^{\flat\flat} &= G^{\flat\flat}_{\Theta}[K^{\flat}_{\Delta}[\underline{\langle}[\,]\underline{\rangle}]]
\end{aligned}
$$

Fig. 12. DS Translation ($\Diamond : \lambda^{*}_{c\mathcal{S}_0} \longrightarrow \lambda^{\triangleright}_{c\mathcal{S}_0}$)

Conversely, Fig. 12 defines the inverse translation, which is formalized in `CPS-DSK.agda`. It uses similar mutually recursive translations ($\natural$, $\sharp$, $\flat$, $\flat\flat$) over the same syntactic categories.

C Translations between DS and Kernel calculi

We present the translation functions between DS and kernel calculi.

$$
\begin{aligned}
M^{\triangleright} &= (M : [\,]_k : [\,]_g) \quad (k, g \notin fv(M)) \\
x^{\blacktriangleright} &= x \\
(\underline{\lambda} x.\, M)^{\blacktriangleright} &= \underline{\lambda} x.\, M^{\triangleright} \\
\underline{\mathcal{S}}^{\blacktriangleright} &= \underline{\mathcal{S}} \\
\underline{\mathcal{S}_0}^{\blacktriangleright} &= \underline{\mathcal{S}_0} \\
V : K : G &= G[K[V^{\blacktriangleright}]] \\
(V \mathbin{\underline{@}} W) : K : G &= G[K[V^{\blacktriangleright} \mathbin{\underline{@}} W^{\blacktriangleright}]] \\
(V \mathbin{\underline{@}} Q) : K : G &= Q : (\underline{\mathsf{let}}\ y = [\,]\ \underline{\mathsf{in}}\ [K[V^{\blacktriangleright} \mathbin{\underline{@}} y]]_g) : G \\
(P \mathbin{\underline{@}} W) : K : G &= P : (\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ [K[x \mathbin{\underline{@}} W^{\blacktriangleright}]]_g) : G \\
(P \mathbin{\underline{@}} Q) : K : G &= P : (\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ (Q : (\underline{\mathsf{let}}\ y = [\,]\ \underline{\mathsf{in}}\ [K[x \mathbin{\underline{@}} y]]_g) : [\,]_g)) : G \\
(\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N) : K : G &= M : (\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ (N : K : [\,]_g)) : G \\
\underline{\langle} M \underline{\rangle} : K : G &= M : [\,]_{\bullet} : (G[K[\underline{\langle}[\,]\underline{\rangle}]])
\end{aligned}
$$

Fig. 13. A-Normalization Translation. ($\triangleright : \lambda_{c\mathcal{S}_0} \longrightarrow \lambda^{\triangleright}_{c\mathcal{S}_0}$)

Fig. 13 defines the A-normalization translation from $\lambda_{c\mathcal{S}_0}$ to $\lambda^{\triangleright}_{c\mathcal{S}_0}$, which is formalized in `DS-DSK.agda`. This translation ($\triangleright$) and its two mutually recursive translations ($\blacktriangleright$ and $- : - : -$) are defined by structural recursion on roots, values, and terms, respectively.

$$
\begin{aligned}
x^{\blacktriangleleft} &= x \\
(\underline{\lambda} x.\, M_k)^{\blacktriangleleft} &= \underline{\lambda} x.\, M_k^{\triangleleft} \\
\underline{\mathcal{S}}^{\blacktriangleleft} &= \underline{\mathcal{S}} \\
\underline{\mathcal{S}_0}^{\blacktriangleleft} &= \underline{\mathcal{S}_0} \\
(G_{\Theta}[K_{\Delta}[V]])^{\triangleleft} &= G^{\trianglelefteq}_{\Theta}[K^{\triangleleft}_{\Delta}[V^{\blacktriangleleft}]] \\
(G_{\Theta}[K_{\Delta}[V \mathbin{\underline{@}} W]])^{\triangleleft} &= G^{\trianglelefteq}_{\Theta}[K^{\triangleleft}_{\Delta}[V^{\blacktriangleleft} \mathbin{\underline{@}} W^{\blacktriangleleft}]] \\
[\,]^{\triangleleft}_k &= [\,] \\
[\,]^{\triangleleft}_{\bullet} &= [\,] \\
(\underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ M_{\Delta})^{\triangleleft} &= \underline{\mathsf{let}}\ x = [\,]\ \underline{\mathsf{in}}\ M^{\triangleleft}_{\Delta} \\
[\,]^{\trianglelefteq}_g &= [\,] \\
(G_{\Theta}[K_{\Delta}[\underline{\langle}[\,]\underline{\rangle}]])^{\trianglelefteq} &= G^{\trianglelefteq}_{\Theta}[K^{\triangleleft}_{\Delta}[\underline{\langle}[\,]\underline{\rangle}]]
\end{aligned}
$$

Fig. 14. Embedding. ($\triangleleft : \lambda^{\triangleright}_{c\mathcal{S}_0} \longrightarrow \lambda_{c\mathcal{S}_0}$)

Conversely, Fig. 14 defines the embedding from $\lambda^{\triangleright}_{c\mathcal{S}_0}$ to $\lambda_{c\mathcal{S}_0}$, which is formalized in `DSK-DS.agda`. It is defined by four mutually recursive translations, $\blacktriangleleft$, $\triangleleft$, $\trianglelefteq$, and $\trianglelefteq\!\!\triangleleft$, via structural recursion on values, terms, continuations, and meta-continuations, respectively.

References

1. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: International Workshop on Computer Science Logic. pp. 453–468. Springer (1999)
2. Asai, K., Fujii, M.: Defining algebraic effects and handlers via trails and metacontinuations. In: Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday. pp. 26–39 (2025)
3. Biernacki, D., Pyzik, M., Sieczkowski, F.: A Reflection on Continuation-Composing Style. In: 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020). pp. 18:1–18:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2020)
4. Biernacki, D., Pyzik, M., Sieczkowski, F.: Reflecting stacked continuations in a fine-grained direct-style reduction theory. In: Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming. pp. 1–13 (2021)
5. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional programming. pp. 143–156 (2008)
6. Cong, Y., Asai, K.: Towards a reflection for effect handlers. In: Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation. pp. 55–65 (2023)
7. Danvy, O.: Back to direct style. Science of Computer Programming **22**(3), 183–195 (1994)
8. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. pp. 151–160 (1990)
9. Danvy, O., Filinski, A.: Representing control: A study of the CPS transformation. Mathematical Structures in Computer Science **2**(4), 361–391 (1992)
10. Danvy, O., Nielsen, L.R.: A first-order one-pass CPS transformation. Theoretical computer science **308**(1), 239–257 (2003)
11. Danvy, O., Pfenning, F.: The occurrence of continuation parameters in CPS terms. Tech. Rep. CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (Feb 1995)
12. Felleisen, M.: The theory and practice of first-class prompts. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 180–190 (1988)
13. Gunter, C.A., Rémy, D., Riecke, J.G.: A generalization of exceptions and control in ML-like languages. In: Proceedings of the seventh international conference on functional programming languages and computer architecture. pp. 12–23 (1995)
14. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Ryu, S. (ed.) Programming Languages and Systems. Lecture Notes in Computer Science, vol. 11275, pp. 415–435. Springer (2018)
15. Hillerström, D., Lindley, S., Atkey, R.: Effect handlers via generalised continuations. Journal of Functional Programming **30**(e5), 1–69 (2020)
16. Honda, K., Yamamoto, J., Asai, K.: A Reflection Proof for a Typed Language with shift/reset. In: Proceedings of the 25th Workshop on Programming and Programming Languages (PPL 2023). pp. 1–20 (2023), (in Japanese)
17. Ishio, C., Asai, K.: Type system for four delimited control operators. In: Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 45–58 (2022)

18. Materzok, M., Biernacki, D.: Subtyping delimited continuations. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional programming. pp. 81–93 (2011)
19. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: European Symposium on Programming. pp. 80–94. Springer (2009)
20. Plotkin, G.D.: Call-by-name, call-by-value and the λ-calculus. Theoretical computer science **1**(2), 125–159 (1975)
21. Sabry, A., Wadler, P.: A reflection on call-by-value. ACM transactions on programming languages and systems (TOPLAS) **19**(6), 916–941 (1997)
22. Shan, C.c.: A static simulation of dynamic delimited control. Higher-order and symbolic computation **20**(4), 371–401 (2007)
23. Sivaramakrishnan, K., Dolan, S., White, L., Kelly, T., Jaffer, S., Madhavapeddy, A.: Retrofitting effect handlers onto OCaml. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 206–221. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454039, `https://doi.org/10.1145/3453483.3454039`