A Dynamic Interpretation of the CPS Hierarchy

Marek Materzok and Dariusz Biernacki

University of Wrocław, Wrocław, Poland

Abstract. The CPS hierarchy of control operators *shift*$_i$/*reset*$_i$ of Danvy and Filinski is a natural generalization of the *shift* and *reset* static control operators that allow for abstracting delimited control in a structured and CPS-guided manner. In this article we show that a dynamic variant of *shift/reset*, known as *shift*$_0$/*reset*$_0$, where the discipline of static access to the stack of delimited continuations is relaxed, can fully express the CPS hierarchy. This result demonstrates the expressive power of *shift*$_0$/*reset*$_0$ and it offers a new perspective on practical applications of the CPS hierarchy.

1 Introduction

In the recent years delimited continuations have been recognized as an important concept in the landscape of eager functional programming, with new theoretical [1, 12], practical [10, 13], and implementational [11, 14] advances in the field. Of the numerous control operators for delimited continuations, the so-called static control operators *shift* and *reset* introduced by Danvy and Filinski in their seminal work [8] occupy a special position, primarily due to the fact that their definition has been based on the well-known concept of the continuation-passing style (CPS). As such, *shift* and *reset* have solid semantic foundations [2, 8], they are fundamentally related to other computational effects [9] and their use is guided by CPS [2, 8].

The hierarchy of control operators *shift*$_i$/*reset*$_i$ [8] is a generalization of *shift/reset* that has been defined in terms of the CPS hierarchy which in turn is obtained by iterated CPS translations. The idea is that the operator *shift*$_i$ can access and abstract the context up to the dynamically nearest enclosing control delimiter reset$_j$ with $i \leq j$. The primary goal for considering the hierarchy is layering nested computational effects [8, 7] as well as expressing computations in hierarchical structures [2]. Recently, Biernacka et al. have proposed a framework for studying typed control operators in the CPS hierarchy [3], where more flexible than *shift*$_i$/*reset*$_i$ hierarchical control operators have been considered.

While the static delimited-control operators are often the choice for theoreticians and practitioners, it has been observed already in Danvy and Filinski's pioneering article [8], that there is an interesting dynamic variant of *shift* and *reset*, nowadays known as *shift*$_0$ and *reset*$_0$ [16], that allows to inspect the stack of delimited contexts arbitrarily deep. In our previous work [15] we have presented a study of *shift*$_0$ and *reset*$_0$ in which we employed a type-and-effect system with subtyping in order to faithfully describe the interaction between terms and layered contexts. Interestingly enough, simple and elegant CPS translations for *shift*$_0$ and *reset*$_0$, in both untyped and typed version, have been given.

Since *shift*$_0$ and *reset*$_0$ can explore and manipulate the stack of contexts quite arbitrarily, the natural question of their relation to the CPS hierarchy arises. In this work we answer this question by showing that *shift*$_0$ and *reset*$_0$ can fully express the CPS hierarchy. To this end, we formally relate their operational semantics, CPS translations and type systems. Furthermore, we show some typical examples of programming in the CPS hierarchy implemented in terms of *shift*$_0$/*reset*$_0$ and an example that goes beyond the CPS hierarchy.

On one hand, the results we present exhibit a considerable expressive power of *shift*$_0$ and *reset*$_0$. On the other hand, they provide a new perspective on the practice and theory of the CPS hierarchy, with possible reasoning principles and implementation techniques for the hierarchy in terms of *shift*$_0$ and *reset*$_0$.

The rest of the article is structured as follows. In Section 2, we present the syntax, operational semantics, CPS translation and type system for the calculus $\lambda_{\$}$ that is a variant of $\lambda_{\mathcal{S}_0}$—a calculus for *shift*$_0$/*reset*$_0$ [15]. In Section 3, we recall the syntax, operational semantics, CPS translation and type system for the calculus $\lambda^{\mathrm{H}}_{\hookleftarrow}$—a calculus for the CPS hierarchy, as defined in [3]. In Section 4, we provide a translation from $\lambda^{\mathrm{H}}_{\hookleftarrow}$ to $\lambda_{\$}$ and we prove soundness of this translation w.r.t. CPS translations, type systems, reduction semantics, and abstract machines. In Section 5, we consider some programming examples that illustrate the use of *shift*$_0$ and *reset*$_0$ in typical applications of the CPS hierarchy. Finally, in Section 6 we conclude.

Theorems presented in this paper are machine-checked with the Twelf theorem prover. The proofs can be accessed at `http://www.tilk.eu/shift0/`.

2 The calculus $\lambda_{\mathcal{S}_0}$ and its variant $\lambda_{\$}$

In this section, we present the syntax, reduction semantics, abstract machine, CPS translation and type system of the calculus $\lambda_{\$}$ that is a variant of $\lambda_{\mathcal{S}_0}$ introduced in [15].

2.1 Introducing *shift*$_0$/*dollar*

The calculus $\lambda_{\mathcal{S}_0}$ is an extension of the call-by-value lambda calculus with a control delimiter *reset*$_0$ ($\langle\rangle$) that delimits contexts and a control operator *shift*$_0$ ($\mathcal{S}_0$) that captures delimited contexts. In this work we introduce a new control operator, called *dollar* (\$). This operator is a generalization of the *reset*$_0$ operator and its variant has been discussed by Kiselyov and Shan [12].[1] It generalizes *reset*$_0$ in the sense that, while $\langle e\rangle$ intuitively means 'run e with a new, empty, context on the context stack', the expression $e_1\,\$\,e_2$ means 'evaluate e_1, then run e_2 with the result of evaluating e_1 pushed on the context stack'. Despite this difference, *dollar* is equally expressive as *reset*$_0$, in both typed and untyped settings. The former one can obviously macro-express the latter as follows:

$$\overline{\langle e\rangle} = (\lambda x.x)\,\$\,\overline{e}$$

[1] The *dollar* of [12] syntactically allows only coterms (which describe contexts) to be pushed on the context stack. In our language, we do not distinguish between terms and coterms and thus allow any function to be pushed as a context on the context stack.

The latter macro-expresses the former in a more complicated way:

$$\overline{e_1 \,\$\, e_2} = (\lambda k.\langle(\lambda x.\mathcal{S}_0 z.k\, x)\, \overline{e_2}\rangle)\, \overline{e_1}$$

We can read the expression above as follows. First, evaluate e_1 and bind the resulting value to the variable k (we are using the call-by-value semantics.) Then run e_2 in a new context—and after the evaluation finishes, remove the delimiter and call k with the resulting value.

Even though *reset*$_0$ and *dollar* are equally expressive, we believe that using the little-studied *dollar* operator, rather than the well-known *reset*$_0$, is justified. The reason is that the translations which express the *shift*$_k$/*reset*$_k$ operators of the CPS hierarchy using *reset*$_0$/*dollar* rely on the semantics of *dollar*; replacing *dollar* with *reset*$_0$ would result in a clumsy and awkward translation. This will become clear in the main part of the article. The *dollar* operator also has several interesting properties which make it worth studying; in particular, it is, in a sense, the inverse of the *shift*$_0$ operator.

2.2 Syntax and semantics of $\lambda_\$$

In this section we formally describe the syntax and reduction semantics of the $\lambda_\$$ language. We introduce the following syntactic categories of terms, values, evaluation contexts and context stacks (called *trails* in this work):

terms	$e ::= x \mid \lambda x.e \mid e\, e \mid e \,\$\, e \mid \mathcal{S}_0 x.e$
values	$v ::= \lambda x.e$
contexts	$K ::= \bullet \mid K\, e \mid v\, K \mid K \,\$\, e$
closed contexts	$\hat{K} ::= v\$K$
trails	$T ::= \Box \mid \hat{K} \cdot T$

Additionally, we use $\langle e \rangle$ as a shorthand for $(\lambda x.x) \,\$\, e$. Metavariables $x, y, f, g, \ldots$ range over variables.

Closed contexts are evaluation contexts terminated at the bottom with a *dollar*. For the purpose of easier manipulation of closed contexts, we introduce the following shorthands:

$$(v\$K)\, e = v\$(K\, e) \qquad v'\, (v\$K) = v\$(v'\, K) \qquad (v\$K) \,\$\, e = v\$(K \,\$\, e)$$

We also define $\hat{\bullet}$ to mean $(\lambda x.x)\$\bullet$.

Evaluation contexts and trails are represented inside-out. This is formalized with the following definition of plugging terms inside contexts and trails:

$$\begin{array}{rr} \bullet[e] = e & (v\$K)[e] = v \,\$\, K[e] \\ (K\, e')[e] = K[e\, e'] & \\ (v\, K)[e] = K[v\, e] & \Box[e] = e \\ (K \,\$\, e')[e] = K[e \,\$\, e'] & (\hat{K} \cdot T)[e] = T[\hat{K}[e]] \end{array}$$

We define the operation of appending two evaluation contexts as follows:

$$\begin{array}{rr} K@\bullet = K & K@(v\, K') = v\, (K@K') \\ K@(K'\, e) = (K@K')\, e & K@(K' \,\$\, e) = (K@K') \,\$\, e \end{array}$$

$$\begin{array}{rlrl}
\langle \lambda x.e, \hat{K} \cdot T\rangle_{\mathrm{e}} &\Rightarrow \langle \hat{K}, \lambda x.e, T\rangle_{\mathrm{a}} & \langle \hat{K}\,\$\,e, v, T\rangle_{\mathrm{a}} &\Rightarrow \langle e, (v\$\bullet)\cdot\hat{K}\cdot T\rangle_{\mathrm{e}}\\
\langle \hat{K}', \hat{K}\cdot T\rangle_{\mathrm{e}} &\Rightarrow \langle \hat{K}, \hat{K}', T\rangle_{\mathrm{a}} & \langle \hat{K}\,e, v, T\rangle_{\mathrm{a}} &\Rightarrow \langle e, (v\,\hat{K})\cdot T\rangle_{\mathrm{e}}\\
\langle e_1\,e_2, \hat{K}\cdot T\rangle_{\mathrm{e}} &\Rightarrow \langle e_1, (\hat{K}\,e_2)\cdot T\rangle_{\mathrm{e}} & \langle \hat{K}'\,\hat{K}, v, T\rangle_{\mathrm{a}} &\Rightarrow \langle \hat{K}', v, \hat{K}\cdot T\rangle_{\mathrm{a}}\\
\langle \mathcal{S}_0 f.e, \hat{K}\cdot T\rangle_{\mathrm{e}} &\Rightarrow \langle e[\hat{K}/f], T\rangle_{\mathrm{e}} & \langle (\lambda x.e)\,\hat{K}, v, T\rangle_{\mathrm{a}} &\Rightarrow \langle e[v/x], \hat{K}\cdot T\rangle_{\mathrm{e}}\\
\langle e_1\,\$\,e_2, \hat{K}\cdot T\rangle_{\mathrm{e}} &\Rightarrow \langle e_1, (\hat{K}\,\$\,e_2)\cdot T\rangle_{\mathrm{e}} & \langle (\lambda x.e)\$\bullet, v, T\rangle_{\mathrm{a}} &\Rightarrow \langle e[v/x], T\rangle_{\mathrm{e}}\\
& & \langle \hat{K}\$\bullet, v, T\rangle_{\mathrm{a}} &\Rightarrow \langle \hat{K}, v, T\rangle_{\mathrm{a}}
\end{array}$$

Fig. 1. Abstract machine for $\lambda_{\$}$

The operation of appending a closed context to an evaluation context is defined as $(v\$K)@K' = v\$(K@K')$.

We have three contraction rules:

$$\begin{array}{rll}
(\lambda x.e)\,v &\rightsquigarrow e[v/x] & (\beta_{\mathrm{v}})\\
v'\,\$\,v &\rightsquigarrow v'\,v & (\$_{\mathrm{v}})\\
\hat{K}[\mathcal{S}_0 f.e] &\rightsquigarrow e[\lambda x.\hat{K}[x]/f] & (\$/\mathcal{S}_0)
\end{array}$$

The first one is the familiar call-by-value beta-reduction. The second one is a generalization of the $\langle v\rangle \rightsquigarrow v$ rule used in $\lambda_{\mathcal{S}_0}$. Indeed, when we interpret $\langle v\rangle$ in $\lambda_{\$}$ as a shorthand for $(\lambda x.x)\,\$\,v$, we have $\langle v\rangle = (\lambda x.x)\,\$\,v \rightsquigarrow (\lambda x.x)\,v \rightsquigarrow v$. The last rule describes how the *shift*$_0$ operator captures the nearest enclosing context (with the delimiting *dollar*) and reifies it as a function.

Finally, we define the reduction relation:

$$K[T[e]] \to K[T[e']] \text{ iff } e \rightsquigarrow e'$$

The semantics defined above satisfies the following unique-decomposition property: for every closed $\lambda_{\$}$ term e, either it is a value, or it is a stuck term of the form $K[\mathcal{S}_0 f.e]$, or it can be uniquely decomposed into a context K, a trail T and a term e' such that $e = K[T[e']]$ and there exists a term e'' such that $e' \rightsquigarrow e''$.

2.3 The abstract machine for $\lambda_{\$}$

The abstract machine for $\lambda_{\$}$ (with values including closed contexts) is defined in Fig. 1. It is very similar to the abstract machine for $\lambda_{\mathcal{S}_0}$, as defined in [5] and [15]. The difference comes from the fact that the (closed) evaluation contexts of $\lambda_{\$}$ are delimited with a *dollar*. This means that empty contexts are not really empty, but contain a terminating value, which needs to be called. That is why instead of the following transition from the abstract machine for $\lambda_{\mathcal{S}_0}$:

$$\langle \bullet, v, K\cdot T\rangle_{\mathrm{a}} \Rightarrow \langle K, v, T\rangle_{\mathrm{a}}$$

we have the following two for $\lambda_{\$}$:

$$\begin{array}{c}
\langle (\lambda x.e)\$\bullet, v, T\rangle_{\mathrm{a}} \Rightarrow \langle e[v/x], T\rangle_{\mathrm{e}}\\
\langle \hat{K}\$\bullet, v, T\rangle_{\mathrm{a}} \Rightarrow \langle \hat{K}, v, T\rangle_{\mathrm{a}}
\end{array}$$

$$\overline{x} = \lambda k.k\, x \qquad \overline{\mathcal{S}_0 x.e} = \lambda x.\overline{e}$$
$$\overline{\lambda x.e} = \lambda k.k\, (\lambda x.\overline{e}) \qquad \overline{e_1 \,\$\, e_2} = \lambda k.\overline{e_1}\, (\lambda v_1.\overline{e_2}\, v_1\, k)$$
$$\overline{e_1\, e_2} = \lambda k.\overline{e_1}\, (\lambda v_1.\overline{e_2}\, (\lambda v_2.v_1\, v_2\, k))$$

Fig. 2. CPS translation for $\lambda_{\$}$ (untyped)

$$\frac{\tau \leq \tau' \qquad \sigma \leq \sigma'}{\tau\ \sigma \leq \tau'\ \sigma'} \qquad \frac{}{\alpha \leq \alpha} \qquad \frac{\tau_1' \leq \tau_1 \qquad \tau_2\ \sigma \leq \tau_2'\ \sigma'}{\tau_1 \xrightarrow{\sigma} \tau_2 \leq \tau_1' \xrightarrow{\sigma'} \tau_2'}$$

$$\frac{}{\epsilon \leq \epsilon} \qquad \frac{\tau\ \sigma \leq \tau'\ \sigma'}{\epsilon \leq [\tau\ \sigma]\ \tau'\ \sigma'} \qquad \frac{\tau_1'\ \sigma_1' \leq \tau_1\ \sigma_1 \qquad \tau_2\ \sigma_2 \leq \tau_2'\ \sigma_2'}{[\tau_1\ \sigma_1]\ \tau_2\ \sigma_2 \leq [\tau_1'\ \sigma_1']\ \tau_2'\ \sigma_2'}$$

$$\frac{}{\Gamma, x : \tau_1 \vdash x : \tau_1}\ \text{VAR} \qquad \frac{\Gamma \vdash e : \tau\ \sigma \qquad \tau\ \sigma \leq \tau'\ \sigma'}{\Gamma \vdash e : \tau'\ \sigma'}\ \text{SUB} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2\ \sigma}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\sigma} \tau_2}\ \text{ABS}$$

$$\frac{\Gamma, x : \tau_1 \xrightarrow{\sigma} \tau_2 \vdash e : \tau_3\ \sigma'}{\Gamma \vdash \mathcal{S}_0 x.e : \tau_1\ [\tau_2\ \sigma]\ \tau_3\ \sigma'}\ \text{SFT} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\sigma} \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2\ \sigma}\ \text{PAPP}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{[\tau_4'\ \sigma_4']\ \tau_3'\ \sigma_3'} \tau_2\ [\tau_2'\ \sigma_2']\ \tau_1'\ \sigma_1' \qquad \Gamma \vdash e_2 : \tau_1\ [\tau_3'\ \sigma_3']\ \tau_2'\ \sigma_2'}{\Gamma \vdash e_1\, e_2 : \tau_2\ [\tau_4'\ \sigma_4']\ \tau_1'\ \sigma_1'}\ \text{APP}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\sigma} \tau_2 \qquad \Gamma \vdash e_2 : \tau_1\ [\tau_2\ \sigma]\ \tau_3\ \sigma'}{\Gamma \vdash e_1 \,\$\, e_2 : \tau_3\ \sigma'}\ \text{PDOL}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\sigma} \tau_2\ [\tau_2'\ \sigma_2']\ \tau_1'\ \sigma_1' \qquad \Gamma \vdash e_2 : \tau_1\ [\tau_2\ \sigma]\ \tau_3\ [\tau_3'\ \sigma_3']\ \tau_2'\ \sigma_2'}{\Gamma \vdash e_1 \,\$\, e_2 : \tau_3\ [\tau_3'\ \sigma_3']\ \tau_1'\ \sigma_1'}\ \text{DOL}$$

Fig. 3. The type system for $\lambda_{\$}$ (with subtyping)

2.4 The CPS translation for $\lambda_{\$}$

The CPS translation is defined in Fig. 2. It is very similar to the untyped translation for $\lambda_{\mathcal{S}_0}$, defined in [15]. The translation for the *reset*$_0$ operator in the original translation is as follows:

$$\overline{\langle e \rangle} = \overline{e}\, (\lambda x.\lambda k.k\, x)$$

In $\lambda_{\$}$, the translation of $\langle e \rangle = (\lambda x.x) \,\$\, e$ is as follows:

$$\overline{(\lambda x.x) \,\$\, e} = \lambda k.(\lambda k.k\, (\lambda x.\lambda k.k\, x))\, (\lambda v.\overline{e}\, v\, k) =_{\beta\eta} \overline{e}\, (\lambda x.\lambda k.k\, x)$$

2.5 The type system for $\lambda_{\$}$

We introduce the syntactic categories of types and effect annotations:

$$\begin{array}{ll} \textit{types} & \tau ::= \alpha \mid \tau \xrightarrow{\sigma} \tau \\ \textit{annotations} & \sigma ::= \epsilon \mid [\tau\ \sigma]\ \tau\ \sigma \end{array}$$

The type system is shown in Fig. 3. Effect annotations have no meaning by themselves; they should be understood together with types they annotate. The typing judgment $\Gamma \vdash e : \tau_1' \ [\tau_1 \ \sigma_1] \ \tau_2' \ldots \tau_n' \ [\tau_n \ \sigma_n] \ \tau$ (we omit ϵ for brevity) can be read as follows: 'the term e in a typing context Γ evaluates to a value of type τ when plugged in a trail of contexts of types $\tau_1' \xrightarrow{\sigma_1} \tau_1, \ldots, \tau_n' \xrightarrow{\sigma_n} \tau_n$'. The judgment $\Gamma \vdash e : \tau$ means 'the term e in a typing context Γ evaluates to a value of type τ without observable control effects'. Function types also have effect annotations, which can be thought to be associated with the return type.

We made two changes to the type system presented in [15]:

1. The rule for pure application has been generalized. This modification does not change the theory, because both cases of the new rule (with σ empty or non-empty) are derivable using the original rules. Yet some proofs are easier with the new rule.
2. The typing rule for *reset*$_0$ has been replaced with two rules for $\$$. When we take $\langle e \rangle = (\lambda x.x) \, \$ \, e$, the original rule for *reset*$_0$ can be derived from them.

This means that the type system in Fig. 3 is a conservative extension of the original type system for *shift*$_0$/*reset*$_0$.

3 The CPS hierarchy—the calculus $\lambda^{\mathrm{H}}_{\hookleftarrow}$

For the purpose of formalizing the connection between *shift*$_0$/*reset*$_0$ and the CPS hierarchy, for our second language we use the hierarchy of flexible delimited-control operators, as defined in the article by Biernacka et al. [3]. The main reason is that it has a very expressive type system, defined in the same article, which we relate to the type system of $\lambda_{\$}$. The language expresses the original CPS hierarchy, as defined in [8], so the results still hold for the original hierarchy.

In this section, we define the syntax, reduction semantics, abstract machine, CPS translation and type system of $\lambda^{\mathrm{H}}_{\hookleftarrow}$. We try to keep the description succinct; more detailed description can be found in [3].

3.1 Syntax and semantics of $\lambda^{\mathrm{H}}_{\hookleftarrow}$

We define the syntactic categories of terms, values, coterms (which represent evaluation contexts), evaluation contexts and programs:

terms	$e ::= x \mid \lambda x.e \mid e \, e \mid \langle e \rangle_i \mid \mathcal{S}_i k_1 \ldots k_i.e \mid (h_1, \ldots, h_i) \hookleftarrow_i e$
values	$v ::= \lambda x.e$
coterms	$h_i ::= k_i \mid E_i$
level 1 *contexts*	$E_1 ::= \bullet_1 \mid E_1 \, e \mid v \, E_1 \mid (E_1, \ldots, E_i) \hookleftarrow_i E_1$
level > 1 *contexts*	$E_i ::= \bullet_i \mid E_i.E_{i-1}$
level n *programs*	$p ::= \langle e, E_1, \ldots, E_{n+1} \rangle$

$$
\begin{aligned}
\langle \lambda x.e, E_1, \ldots, E_{n+1} \rangle_{\mathrm{e}} &\Rightarrow \langle \lambda x.e, E_1, \ldots, E_{n+1} \rangle_{\mathrm{a}} \\
\langle e_1\, e_2, E_1, \ldots, E_{n+1} \rangle_{\mathrm{e}} &\Rightarrow \langle e_1, E_1\, e_2, E_2, \ldots, E_{n+1} \rangle_{\mathrm{e}} \\
\langle (E'_1, \ldots, E'_i) \hookleftarrow_i e, E_1, \ldots, E_{n+1} \rangle_{\mathrm{e}} &\Rightarrow \langle e, (E'_1, \ldots, E'_i) \hookleftarrow_i E_1, E_2, \ldots, E_{n+1} \rangle_{\mathrm{e}} \\
\langle \langle e \rangle_i, E_1, \ldots, E_{n+1} \rangle_{\mathrm{e}} &\Rightarrow \langle e, \bullet_1, \ldots, \bullet_i, E_{i+1}.(E_i. \ldots .(E_2.E_1)), \\
&\qquad E_{i+2}, \ldots, E_{n+1} \rangle_{\mathrm{e}} \\
\langle \mathcal{S}_i k_1, \ldots, k_i.e, E_1, \ldots, E_{n+1} \rangle_{\mathrm{e}} &\Rightarrow \langle e[E_1/k_1] \ldots [E_i/k_i], \bullet_1, \ldots, \bullet_i, \\
&\qquad E_{i+1}, \ldots, E_{n+1} \rangle_{\mathrm{e}} \\
\langle v, E_1\, e_2, E_2, \ldots, E_{n+1} \rangle_{\mathrm{a}} &\Rightarrow \langle e_2, v\, E_1, E_2, \ldots, E_{n+1} \rangle_{\mathrm{e}} \\
\langle v, (\lambda x.e)\, E_1, E_2, \ldots, E_{n+1} \rangle_{\mathrm{a}} &\Rightarrow \langle e[v/x], E_1, E_2, \ldots, E_{n+1} \rangle_{\mathrm{e}} \\
\langle v, (E'_1, \ldots, E'_i) \hookleftarrow_i E_1, E_2, \ldots, E_{n+1} \rangle_{\mathrm{a}} &\Rightarrow \langle v, E'_1, \ldots, E'_i, E_{i+1}.(E_i. \ldots .(E_2.E_1)), \\
&\qquad E_{i+2}, \ldots, E_{n+1} \rangle_{\mathrm{a}} \\
\langle v, \bullet_i, E_{i+1}, \ldots, E_{n+1} \rangle_{\mathrm{a}} &\Rightarrow \langle v, E_{i+1}, \ldots, E_{n+1} \rangle_{\mathrm{a}} \\
\langle v, E_i.(E_{i-1}. \ldots .(E_2.E_1)), E_{i+1}, \ldots, E_{n+1} \rangle_{\mathrm{a}} &\Rightarrow \langle v, E_1, E_2, \ldots, E_{n+1} \rangle_{\mathrm{a}}
\end{aligned}
$$

Fig. 4. Abstract machine for $\lambda^{\mathrm{H}}_{\hookleftarrow}$

Plugging terms inside evaluation contexts is defined as follows:

$$
\begin{aligned}
\bullet_i[e] &= e \\
(E_1\, e')[e] &= E_1[e\, e'] \\
(v\, E_1)[e] &= E_1[v\, e] \\
((E'_1, \ldots, E'_i) \hookleftarrow_i E_1)[e] &= E_1[(E_1, \ldots, E_i) \hookleftarrow_i e] \\
(E_i.E_{i-1})[e] &= E_i[\langle E_{i-1}[e] \rangle_{i-1}]
\end{aligned}
$$

The syntactic category of programs exists for the purpose of defining the reduction semantics. The *plug* function defined below reconstructs the term represented by a given program tuple:

$$
\begin{aligned}
\mathit{plug}\, \langle e, E_1\, e', E_2, \ldots, E_{n+1} \rangle &= \mathit{plug}\, \langle e\, e', E_1, E_2, \ldots, E_{n+1} \rangle \\
\mathit{plug}\, \langle e, v\, E_1, E_2, \ldots, E_{n+1} \rangle &= \mathit{plug}\, \langle v\, e, E_1, E_2, \ldots, E_{n+1} \rangle \\
\mathit{plug}\, \langle e, (E'_1, \ldots, E'_i) \hookleftarrow_i E_1, E_2, \ldots, E_{n+1} \rangle &= \mathit{plug}\, \langle (E'_1, \ldots, E'_i) \hookleftarrow_i e, \\
&\qquad E_1, E_2, \ldots, E_{n+1} \rangle \\
\mathit{plug}\, \langle e, \bullet_1, \ldots, \bullet_i, E_{i+1}.(E_i. \ldots .(E_2.E_1)), \\
\qquad E_{i+2}, \ldots, E_{n+1} \rangle &= \mathit{plug}\, \langle \langle e \rangle_i, E_1, \ldots, E_{n+1} \rangle \\
\mathit{plug}\, \langle e, \bullet_1, \ldots, \bullet_{n+1} \rangle &= \langle e \rangle_n
\end{aligned}
$$

Finally, we define the reduction semantics as follows:

$$
\begin{aligned}
\langle (\lambda x.e)\, v, E_1, \ldots, E_{n+1} \rangle &\rightarrow \langle e[v/x], E_1, \ldots, E_{n+1} \rangle \\
\langle \mathcal{S}_i k_1 \ldots k_i.e, E_1, \ldots, E_{n+1} \rangle &\rightarrow \langle e[E_1/k_1] \ldots [E_i/k_i], \bullet_1, \ldots, \bullet_i, \\
&\qquad E_{i+1}, \ldots E_{n+1} \rangle \\
\langle \langle v \rangle_i, E_1, \ldots, E_{n+1} \rangle &\rightarrow \langle v, E_1, \ldots, E_{n+1} \rangle \\
\langle (E'_1, \ldots, E'_i) \hookleftarrow_i v, E_1, \ldots, E_{n+1} \rangle &\rightarrow \langle v, E'_1, \ldots, E'_i, E_{i+1}.(E_i. \ldots .(E_2.E_1)), \\
&\qquad E_{i+2}, \ldots E_{n+1} \rangle
\end{aligned}
$$

3.2 The abstract machine for $\boldsymbol{\lambda^{\mathrm{H}}_{\hookleftarrow}}$

The abstract machine is defined in Fig. 4.

$$
\begin{aligned}
\overline{x} &= \lambda k.k\,x \\
\overline{\lambda x.e} &= \lambda k.k\,(\lambda x.\overline{e}) \\
\overline{\langle e \rangle_i} &= \lambda k_1 \ldots . \lambda k_{i+1}.\overline{e}\,\theta\,.^i.\,\theta \\
&\qquad (\lambda v.k_1\,v\,k_2\ \ldots\ k_{i+1}) \\
\overline{\mathcal{S}_i k_1 \ldots k_i.e} &= \lambda k_1 \ldots . \lambda k_i.\overline{e}\,\theta\,.^i.\,\theta \\
\overline{(h_1,\ldots,h_i) \hookleftarrow_i e} &= \lambda k.\overline{e}\,(\lambda v.\lambda k_2 \ldots . \lambda k_{i+1}.\overline{h_1}\,v\,\overline{h_2}\ \ldots\ \overline{h_i} \\
&\qquad (\lambda w.k\,w\,k_2\ \ldots\ k_{i+1})) \\
\overline{k} &= k \\
\overline{\bullet_i} &= \theta \\
\overline{E_1\,e} &= \lambda v.\overline{e}\,(\lambda w.v\,w\,\overline{E_1}) \\
\overline{v\,E_1} &= \lambda w.v^*\,w\,\overline{E_1} \\
\overline{(E_1,\ldots,E_i) \hookleftarrow_i E_1'} &= \lambda v.\lambda k_2 \ldots . \lambda k_{i+1}.\overline{E_1}\,v\,\overline{E_2}\ \ldots\ \overline{E_i} \\
&\qquad (\lambda w.\overline{E_1'}\,w\,k_2\ \ldots\ k_{i+1}) \\
\overline{E_i.E_{i-1}} &= \lambda v.\overline{E_{i-1}}\,v\,\overline{E_i} \\
\\
(\lambda x.e)^* &= \lambda x.\overline{e} \\
\text{where } \theta &= \lambda x.\lambda k.k\,x
\end{aligned}
$$

Fig. 5. CPS translation for $\lambda^{\mathrm{H}}_{\hookleftarrow}$ (eta-reduced)

3.3 The CPS translation for $\boldsymbol{\lambda^{\mathbf{H}}_{\hookleftarrow}}$

The CPS translation is defined in Fig. 5. It is presented in eta-reduced form. The translated terms can be eta-expanded back to the form presented in [3].

3.4 The type system for $\boldsymbol{\lambda^{\mathbf{H}}_{\hookleftarrow}}$

We introduce the syntactic categories of types and context types:

$$
\begin{array}{ll}
\textit{types} & \tau ::= \alpha \mid \tau \to [\delta_1,\ldots,\delta_{n+1}] \\
\textit{level} \le n \textit{ context types} & \delta_i ::= \tau \rhd \delta_{i+1} \rhd \ldots \rhd \delta_{n+1} \\
\textit{level } n+1 \textit{ context types} & \delta_{n+1} ::= \neg\tau
\end{array}
$$

The type system for terms is defined in Fig. 6. For lack of space, we omit the typing rules for coterms (they do not introduce any new essential concepts). The entire type system can be found in [3].

4 Relating $\boldsymbol{\lambda_{\$}}$ to $\boldsymbol{\lambda^{\mathbf{H}}_{\hookleftarrow}}$

In this section we relate the CPS translations, type systems, abstract machines, and reduction semantics of $\lambda_{\$}$ and $\lambda^{\mathrm{H}}_{\hookleftarrow}$. This is the main section of the article.

4.1 CPS translations

We present in Fig. 7 a translation from $\lambda^{\mathrm{H}}_{\hookleftarrow}$ to $\lambda_{\$}$. The translation represents the control operators of the CPS hierarchy with *shift*$_0$/*dollar*, and refunctionalizes the evaluation contexts present in the reduction semantics of $\lambda^{\mathrm{H}}_{\hookleftarrow}$. We arrived at this translation by trying to match the CPS translations of $\lambda^{\mathrm{H}}_{\hookleftarrow}$ and $\lambda_{\$}$ and to capture the operational meaning of the CPS hierarchy's control operators:

$$\frac{}{\Gamma,\, x:\tau;\Delta \;\vdash_n\; x:\tau \rhd \delta_2 \ldots \rhd \delta_{n+1}, \delta_2 \ldots \delta_{n+1}}$$

$$\frac{\Gamma,\, x:\tau;\Delta \;\vdash_n\; e:\delta'_1, \ldots \delta'_{n+1}}{\Gamma;\Delta \;\vdash_n\; \lambda x.e : (\tau \to [\delta'_1 \ldots \delta'_{n+1}]) \rhd \delta_2 \ldots \rhd \delta_{n+1}, \delta_2 \ldots \delta_{n+1}}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \;\vdash_n\; e_0 : (\tau \to [\delta_1 \ldots \delta_{n+1}]) \rhd \delta''_2 \ldots \rhd \delta''_{n+1}, \delta'_2 \ldots \delta'_{n+1} \\ \Gamma;\Delta \;\vdash_n\; e_1 : \tau \rhd \delta_2 \ldots \rhd \delta_{n+1}, \delta''_2 \ldots \delta''_{n+1}\end{array}}{\Gamma;\Delta \;\vdash_n\; e_0\, e_1 : \delta_1, \delta'_2 \ldots \delta'_{n+1}}$$

$$\frac{\mathcal{I}_1(\delta'_1) \quad \ldots \quad \mathcal{I}_i(\delta'_i) \quad \Gamma;\Delta \;\vdash_n\; e : \delta'_1 \ldots \delta'_i, (\tau \rhd \delta_{i+2} \ldots \rhd \delta_{n+1}), \delta'_{i+2} \ldots \delta'_{n+1}}{\Gamma;\Delta \;\vdash_n\; \langle e \rangle_i : \tau \rhd \delta_2 \ldots \rhd \delta_{n+1}, \delta_2 \ldots \delta_{i+1}, \delta'_{i+2} \ldots \delta'_{n+1}}$$

$$\frac{\mathcal{I}_1(\delta'_1) \quad \ldots \quad \mathcal{I}_i(\delta'_i) \quad \Gamma;\Delta, k_1:\delta_1, \ldots, k_i:\delta_i \;\vdash_n\; e : \delta'_1 \ldots \delta'_i, \delta_{i+1} \ldots \delta_{n+1}}{\Gamma;\Delta \;\vdash_n\; \mathcal{S}_i k_1 \ldots k_i.e : \delta_1, \delta_2 \ldots \delta_{n+1}}$$

$$\frac{\begin{array}{c}\delta_1 = \tau \rhd \delta_2 \ldots \rhd \delta_{n+1} \qquad \delta_{i+1} = \tau' \rhd \delta'_{i+2} \ldots \rhd \delta'_{n+1} \\ \Gamma;\Delta \;\vdash_n\; h_1 : \delta_1 \quad \ldots \quad \Gamma;\Delta \;\vdash_n\; h_i : \delta_i \\ \Gamma;\Delta \;\vdash_n\; e : \tau \rhd \delta''_2 \rhd \ldots \rhd \delta''_{i+1} \rhd \delta_{i+2} \rhd \ldots \rhd \delta_{n+1}, \delta'_2, \ldots \delta'_{n+1}\end{array}}{\Gamma;\Delta \;\vdash_n\; (h_1 \ldots h_i) \hookleftarrow_i e : \tau' \rhd \delta''_2 \rhd \ldots \rhd \delta''_{i+1} \rhd \delta'_{i+2} \rhd \ldots \rhd \delta'_{n+1}, \delta'_2 \ldots \delta'_{n+1}}$$

$$\mathcal{I}_i(\delta_i) := \exists \tau, \delta_{i+2}, \ldots, \delta_{n+1}.\delta_i = \tau \rhd (\tau \rhd \delta_{i+2} \rhd \ldots \rhd \delta_{n+1}) \rhd \delta_{i+2} \rhd \ldots \rhd \delta_{n+1}$$

Fig. 6. The type system for level n $\lambda^{\mathrm{H}}_{\hookleftarrow}$ terms

- The operator $\langle e \rangle_n$ affects only the top $n+1$ contexts. It resets the top n contexts, and pushes the original contexts down to the $n+1$-th context. We can achieve that in $\lambda_\$$ by first capturing the $n+1$ contexts by using the *shift*$_0$ operator $n+1$ times, pushing the extended $n+1$-th context using \$, and then pushing the empty context using *reset*$_0$ n times.
- The operator $(k_1, \ldots, k_n) \hookleftarrow_n e$ works very similar to $\langle e \rangle_n$, only instead of resetting the top n contexts, it replaces them with the given contexts $k_1, \ldots, k_n$. In $\lambda_\$$, instead of pushing empty contexts, we push the given contexts using \$.
- The operator $\mathcal{S}_n k_1 \ldots k_n.e$ affects only the top n contexts. It captures them and replaces them by empty contexts. We can achieve that in $\lambda_\$$ by using the *shift*$_0$ operator n times, and then using *reset*$_0$ n times.

We have the following theorem:

Theorem 1. *For every term e in $\lambda^{H}_{\hookleftarrow}$, we have $\overline{e} =_{\beta\eta} \overline{[\![e]\!]}$.*

Proof. By induction and simple calculation. It is easy to see that following hold:

$$\overline{(\lambda x.e)\,\$\,e'} =_{\beta\eta} \overline{e'}\,(\lambda x.\overline{e}) \qquad \overline{x\,\$\,e} =_{\beta\eta} \overline{e}\,x$$
$$\overline{\langle e \rangle} =_{\beta\eta} \overline{e}\,\theta \qquad \overline{k_n\,\$\,\ldots\,\$\,k_1\,\$\,x} =_{\beta\eta} k_1\,x\,k_2\,\ldots\,k_n$$

With these, showing the theorem is straightforward.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![e_1\ e_2]\!] &= [\![e_1]\!]\ [\![e_2]\!] \\
[\![\lambda x.e]\!] &= \lambda x.[\![e]\!] \\
[\![\langle e\rangle_i]\!] &= \mathcal{S}_0 f_1.\ldots.\mathcal{S}_0 f_{i+1}.(\lambda x.f_{i+1}\,\$\,\ldots\,\$\,f_1\,\$\,x)\,\$\,\langle .\overset{i}{.}.\langle [\![e]\!]\rangle\ldots\rangle \\
[\![(h_1,\ldots,h_i)\leftarrow_i e]\!] &= \mathcal{S}_0 f_1.\ldots.\mathcal{S}_0 f_{i+1}.(\lambda x.f_{i+1}\,\$\,\ldots\,\$\,f_1\,\$\,x)\,\$ \\
&\qquad [\![h_i]\!]_i\,\$\,\ldots\,\$\,[\![h_1]\!]_1\,\$\,[\![e]\!] \\
[\![\mathcal{S}_i k_1\ldots k_i.e]\!] &= \mathcal{S}_0 k_1.\ldots.\mathcal{S}_0 k_i.\langle .\overset{i}{.}.\langle [\![e]\!]\rangle\ldots\rangle \\
[\![k_i]\!]_i &= k_i \\
[\![E_i]\!]_i &= \lambda x.(\lambda y.[\![E_i]\!]^{\mathrm{c}}_i(y))\,\$\,x \\
[\![\bullet_i]\!]^{\mathrm{c}}_i(x) &= \langle x\rangle \\
[\![E_1\ e]\!]^{\mathrm{c}}_1(x) &= [\![E_1]\!]^{\mathrm{c}}_1(x\ [\![e]\!]) \\
[\![v\ E_1]\!]^{\mathrm{c}}_1(x) &= [\![E_1]\!]^{\mathrm{c}}_1([\![v]\!]\ x) \\
[\![(E_1,\ldots,E_i)\leftarrow_i E_1']\!]^{\mathrm{c}}_1(x) &= [\![E_1']\!]^{\mathrm{c}}_1((\lambda y.\mathcal{S}_0 f_1.\ldots.\mathcal{S}_0 f_{i+1}.(\lambda z.f_{i+1}\,\$\,\ldots\,\$\,f_1\,\$\,z)\,\$ \\
&\qquad [\![E_i]\!]_n\,\$\,\ldots\,\$\,[\![E_1]\!]_1\,\$\,y)\,x) \\
[\![E_i.E_{i-1}]\!]^{\mathrm{c}}_i(x) &= (\lambda y.[\![E_i]\!]^{\mathrm{c}}_i(y))\,\$\,[\![E_{i-1}]\!]^{\mathrm{c}}_{i-1}(x) \\
[\![\langle e, E_1, E_2, \ldots, E_{n+1}\rangle]\!] &= (\lambda x.[\![E_n]\!]^{\mathrm{c}}_{n+1}(x))\,\$\,\ldots\,\$\,(\lambda x.[\![E_1]\!]^{\mathrm{c}}_1(x))\,\$\,[\![e]\!]
\end{aligned}
$$

Fig. 7. Translation of $\lambda^{\mathrm{H}}_{\hookleftarrow}$ to $\lambda_{\$}$

4.2 Type systems

We use the following convention:

$$\tau' \to (\tau\ \sigma) = \tau' \xrightarrow{\sigma} \tau$$

I.e. annotating the type on the right side of the function arrow means the same as annotating the function arrow. Also, we define the operation of appending a function type and an annotated type as follows:

$$(\tau_1 \xrightarrow{\sigma} \tau_2)\mathbin{@}\tau'\ \sigma' = \tau_1\ [\tau_2\ \sigma]\ \tau'\ \sigma'$$

The meaning of this definition comes from the fact that annotated types describe types of the individual contexts on a trail.

We define the translation from $\lambda^{\mathrm{H}}_{\hookleftarrow}$ types to $\lambda_{\$}$ types as follows:

$$
\begin{aligned}
[\![\alpha]\!] &= \alpha \\
[\![S \to [C_1,\ldots,C_{n+1}]]\!] &= [\![S]\!] \to [\![C_1,\ldots,C_{n+1}]\!] \\
[\![C_i,\ldots,C_n,\neg S]\!] &= [\![C_i]\!]\mathbin{@}\ldots\mathbin{@}[\![C_n]\!]\mathbin{@}[\![S]\!] \\
[\![S \rhd C_{i+1} \rhd \ldots \rhd C_{n+1}]\!] &= [\![S]\!] \to [\![C_{i+1},\ldots,C_{n+1}]\!]
\end{aligned}
$$

Theorem 2. *If $\Gamma \vdash_n e : C_1,\ldots,C_{n+1}$ in $\lambda^{H}_{\hookleftarrow}$, then $[\![\Gamma]\!] \vdash [\![e]\!] : [\![C_1,\ldots,C_{n+1}]\!]$ in $\lambda_{\$}$.*

Proof. Cases for variables and lambda abstractions are trivial and follow from the fact that for every τ, τ' and σ, $\tau' \le \tau'\ [\tau\ \sigma]\ \tau\ \sigma$.

4.3 Reduction semantics

The reduction semantics of $\lambda^{\mathrm{H}}_{\hookleftarrow}$ and $\lambda_{\$}$ cannot be related directly. The reason is that in the $\lambda^{\mathrm{H}}_{\hookleftarrow}$ language, evaluation 'goes straight through' the $\langle e\rangle_i$ operators, but their translation to $\lambda_{\$}$ has operational meaning. E.g., the term $\langle(\lambda x.x)\ (\lambda x.x)\rangle_1$ reduces in one step

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![e_1\, e_2]\!] &= [\![e_1]\!]\, [\![e_2]\!] \\
[\![\lambda x.e]\!] &= \lambda x.[\![e]\!] \\
[\![\langle e \rangle_i]\!] &= \mathcal{S}_0 f_1 \ldots \mathcal{S}_0 f_{i+1}.(\lambda x.f_{i+1} \,\$\, \ldots \,\$\, f_1 \,\$\, x) \,\$\, \langle . \overset{i}{.} \langle [\![e]\!] \rangle \ldots \rangle \\
[\![(k_1, \ldots, k_n) \hookleftarrow_i e]\!] &= \mathcal{S}_0 f_1 \ldots \mathcal{S}_0 f_{i+1}.(\lambda x.f_{i+1} \,\$\, \ldots \,\$\, f_1 \,\$\, x) \,\$\, \\
&\qquad\qquad [\![k_i]\!]_i \,\$\, \ldots \,\$\, [\![k_1]\!]_1 \,\$\, [\![e]\!] \\
[\![\mathcal{S}_i k_1 \ldots k_i.e]\!] &= \mathcal{S}_0 k_1 \ldots \mathcal{S}_0 k_i.\langle . \overset{i}{.} \langle [\![e]\!] \rangle \ldots \rangle \\
\\
[\![k]\!]_i &= k \\
[\![E_i]\!]_i &= [\![E_i]\!]_i^{\mathrm{c}} \\
\\
[\![\bullet_i]\!]_i^{\mathrm{c}} &= \hat{\bullet} \\
[\![E_1\, e]\!]_1^{\mathrm{c}} &= [\![E_1]\!]_1^{\mathrm{c}}\, [\![e]\!] \\
[\![v\, E_1]\!]_1^{\mathrm{c}} &= [\![v]\!]\, [\![E_1]\!]_1^{\mathrm{c}} \\
[\![(E_1, \ldots, E_i) \hookleftarrow_i E_1']\!]_1^{\mathrm{c}} &= (\lambda y.\mathcal{S}_0 f_1 \ldots \mathcal{S}_0 f_{i+1}.(\lambda z.f_{i+1} \,\$\, \ldots \,\$\, f_1 \,\$\, z) \,\$\, \\
&\qquad\qquad [\![E_i]\!]_i \,\$\, \ldots \,\$\, [\![E_1]\!]_1 \,\$\, y)\, [\![E_1']\!]_1^{\mathrm{c}} \\
[\![E_i.(E_{i-1}.(\ldots E_{j+1}.E_j))]\!]_i^{\mathrm{c}} &= (\lambda x.[\![E_i]\!]_i^{\mathrm{c}} \,\$\, [\![E_{i-1}]\!]_{i-1}^{\mathrm{c}} \,\$\, \ldots \,\$\, [\![E_{j+1}]\!]_{j+1}^{\mathrm{c}} \,\$\, [\![E_j]\!]_j^{\mathrm{c}} \,\$\, x) \$ \bullet \\
\\
[\![\langle e, E_k, E_{k+1}, \ldots, E_{n+1} \rangle_{\mathrm{e}}]\!] &= \langle [\![e]\!], [\![E_k]\!]_k^{\mathrm{c}} \cdot [\![E_{k+1}]\!]_{k+1}^{\mathrm{c}} \cdot \ldots \cdot [\![E_{n+1}]\!]_n^{\mathrm{c}} \cdot \hat{\bullet} \rangle_{\mathrm{e}} \\
[\![\langle v, E_k, E_{k+1}, \ldots, E_{n+1} \rangle_{\mathrm{a}}]\!] &= \langle [\![v]\!], [\![E_k]\!]_k^{\mathrm{c}} \cdot [\![E_{k+1}]\!]_{k+1}^{\mathrm{c}} \cdot \ldots \cdot [\![E_{n+1}]\!]_n^{\mathrm{c}} \cdot \hat{\bullet} \rangle_{\mathrm{a}}
\end{aligned}
$$

Fig. 8. Translation of $\lambda^{\mathrm{H}}_{\hookleftarrow}$ to $\lambda_{\$}$ (abstract machines)

to $\langle \lambda x.x \rangle_1$, whereas evaluation of the translated program $\langle \langle (\lambda x.x)\, (\lambda x.x) \rangle_1, \bullet_1, \bullet_2 \rangle$ proceeds as follows:

$$
\begin{aligned}
&(\lambda x.\langle x \rangle) \,\$\, (\lambda x.\langle x \rangle) \,\$\, \mathcal{S}_0 f_1.\mathcal{S}_0 f_2.(\lambda x.f_2 \,\$\, f_1 \,\$\, x) \,\$\, \langle (\lambda x.x)\, (\lambda x.x) \rangle \\
\to^* \;& (\lambda x.(\lambda x.(\lambda x.\langle x \rangle) \,\$\, x) \,\$\, (\lambda x.(\lambda x.\langle x \rangle) \,\$\, x) \,\$\, x) \,\$\, \langle \lambda x.x \rangle \\
\neq \;& (\lambda x.\langle x \rangle) \,\$\, (\lambda x.\langle x \rangle) \,\$\, \mathcal{S}_0 f_1.\mathcal{S}_0 f_2.(\lambda x.f_2 \,\$\, f_1 \,\$\, x) \,\$\, \langle \lambda x.x \rangle
\end{aligned}
$$

Interestingly, using a refocused [4] version of the $\lambda^{\mathrm{H}}_{\hookleftarrow}$ reduction semantics eliminates this discrepancy.

We present below how another $\lambda^{\mathrm{H}}_{\hookleftarrow}$ term evaluates:

$$\langle \mathcal{S}_1 k.(k) \hookleftarrow_1 \lambda x.x \rangle_1 \to \langle (\bullet_1) \hookleftarrow_1 \lambda x.x \rangle_1 \to \langle \langle \lambda x.x \rangle_1 \rangle_1$$

Its translation evaluates as follows:

$$
\begin{aligned}
&(\lambda x.\langle x \rangle) \,\$\, (\lambda x.\langle x \rangle) \,\$\, \mathcal{S}_0 k.\langle \mathcal{S}_0 k_1.\mathcal{S}_0 k_2.(\lambda x.k_2 \,\$\, k_1 \,\$\, x) \,\$\, k \,\$\, \lambda x.x \rangle \\
\to^* \;& (\lambda x.\langle x \rangle) \,\$\, \langle \mathcal{S}_0 k_1.\mathcal{S}_0 k_2.(\lambda x.k_2 \,\$\, k_1 \,\$\, x) \,\$\, (\lambda x.(\lambda x.\langle x \rangle) \,\$\, x) \,\$\, \lambda x.x \rangle \\
\to^* \;& ((\lambda x.(\lambda x.\langle x \rangle) \,\$\, x) \,\$\, (\lambda x.\langle x \rangle) \,\$\, x) \,\$\, (\lambda x.(\lambda x.\langle x \rangle) \,\$\, x) \,\$\, \lambda x.x
\end{aligned}
$$

We see that the terms representing the evaluation contexts get more and more complicated. This is because in the reduction semantics for $\lambda_{\$}$, removing a context from the trail and then pushing it back is not the same as just leaving it there:

$$v \,\$\, \mathcal{S}_0 k.k \,\$\, e \to (\lambda x.v \,\$\, x) \,\$\, e \neq v \,\$\, e$$

However, it is easy to see that $\overline{(\lambda x.v \,\$\, x) \,\$\, e} =_{\beta\eta} \overline{v \,\$\, e}$. What is more, we have:

$$\overline{(\lambda x.\hat{K}[x]) \,\$\, e} =_{\beta\eta} \overline{\hat{K}[e]}.$$

Let us define $\approx$ as the smallest congruence containing $(\lambda x.\hat{E}[x])\,\$\,e \approx \hat{E}[e]$. Then, if we define the relation of reduction modulo this congruence:

$$e_1 \to^{\approx} e_2 \text{ iff exists } e_1', e_2' \text{ such that } e_1 \approx e_1' \to e_2' \approx e_2,$$

we obtain the following theorem:

Theorem 3. *Suppose that* $\langle e, E_1, \ldots, E_{n+1}\rangle \rightsquigarrow \langle e', E_1', \ldots, E_{n+1}'\rangle$. *Then we have* $[\![\langle e, E_1, \ldots, E_{n+1}\rangle]\!] \to^{\approx *} [\![\langle e', E_1', \ldots, E_{n+1}'\rangle]\!]$.

Another way to resolve this discrepancy is to change the reduction semantics so as not to refunctionalize the closed contexts on capture, and expand them when on the left side of the *dollar*:

$$\hat{K}\,v \rightsquigarrow \hat{K}[v] \qquad \hat{K}[\mathcal{S}_0 f.e] \rightsquigarrow e[\hat{K}/f] \qquad \hat{K}\,\$\,e \rightsquigarrow \hat{K}[e]$$

4.4 Abstract machines

Relating the abstract machines for the two languages requires a modified translation from $\lambda^{\mathrm{H}}_{\hookleftarrow}$ to $\lambda_{\$}$. The idea behind the translation is that we want to 'emulate' the behavior of the control operators of $\lambda^{\mathrm{H}}_{\hookleftarrow}$ (as most clearly described by the abstract machine in Fig. 4) by step-by-step manipulation of the trail. The translation, as presented in Fig. 8, differs from the previous one in that it does not refunctionalize the evaluation contexts, translating them to closed contexts of $\lambda_{\$}$.

There is a minor mismatch between the abstract machines of $\lambda^{\mathrm{H}}_{\hookleftarrow}$ and $\lambda_{\$}$. The reason is that shifting a context from the trail and then pushing it back with $\$$ alters the trail to a different, but observationally indistinguishable state:

$$\langle \mathcal{S}_0 x.x\,\$\,e, \hat{K}\cdot\hat{K}'\cdot T\rangle_{\mathrm{e}} \Rightarrow \langle \hat{K}\,\$\,e, \hat{K}'\cdot T\rangle_{\mathrm{e}} \Rightarrow^* \langle \hat{K}'\,\$\,e, \hat{K}, T\rangle_{\mathrm{a}} \Rightarrow \langle e, (\hat{K}\$\bullet)\cdot\hat{K}'\cdot T\rangle_{\mathrm{e}}$$

To abstract this detail away, let us define $\approx$ to be a family of congruences defined for $\lambda_{\$}$ expressions, contexts, closed contexts and abstract machines, containing $\hat{K}\$K' \approx \hat{K}@K'$. We get the following theorem:

Theorem 4. *For every two* $\lambda^{H}_{\hookleftarrow}$ *abstract machine configurations* $m_1 \Rightarrow^* m_2$, *then there exists a* $\lambda_{\$}$ *machine configuration* m *such that* $[\![m_1]\!] \Rightarrow^* m \approx [\![m_2]\!]$ *in* $\lambda_{\$}$.

Because the abstract machine for $\lambda^{\mathrm{H}}_{\hookleftarrow}$ uses $n+1$ evaluation contexts for the n-th level of the hierarchy, we need to enclose the translated program in $n+1$ *resets* to match the initial configurations of the machines: $[\![e]\!]_n = \langle \overset{n+1}{\ldots}\langle [\![e]\!]\rangle \ldots\rangle$. Thus, we have:

$$\langle [\![e]\!]_n, \hat{\bullet}\rangle_{\mathrm{e}} \Rightarrow^* \langle [\![e]\!], \hat{\bullet}\cdot\overset{n+2}{\ldots}\cdot\hat{\bullet}\rangle_{\mathrm{e}} = [\![\langle e, \bullet_1, \ldots, \bullet_{n+1}\rangle_{\mathrm{e}}]\!]$$

5 Examples

In this section we show example programs to display the correspondence between $\lambda^{\mathrm{H}}_{\hookleftarrow}$ and $\lambda_{\$}$. The programming languages used are straightforward extensions of $\lambda^{\mathrm{H}}_{\hookleftarrow}$ and $\lambda_{\$}$ to an ML-like language.

$\text{fail}\ () = \mathcal{S}_1 k_s.()$
$\text{amb}\ a\ b = \mathcal{S}_1 k_s.(k_s) \hookleftarrow_1 a; (k_s) \hookleftarrow_1 b$
$\text{emit}\ a = \mathcal{S}_2 k_s k_f.a :: (k_s, k_f) \hookleftarrow_2 ()$
$\text{run}\ f = \langle\langle \text{emit}\ (f\ ())\rangle_1; []\rangle_2$

$\text{fail}\ () = \mathcal{S}_0 k_s.()$
$\text{amb}\ a\ b = \mathcal{S}_0 k_s.k_s\ a; k_s\ b$
$\text{emit}\ a = \mathcal{S}_0 k_s.\mathcal{S}_0 k_f.a :: (k_f\ \$\ k_s\ ())$
$\text{run}\ f = \langle\langle \text{emit}\ (f\ ())\rangle; []\rangle$

Fig. 9. Implementation of *amb* and *emit* in $\lambda^{\text{H}}_{\hookleftarrow}$ (left) and $\lambda_{\$}$ (right)

5.1 Nondeterminism

One of the classic applications of the CPS hierarchy is an implementation of McCarthy's *amb* non-deterministic choice operator, together with a function for returning results of the non-deterministic computation [8]. The implementation uses two levels of the CPS hierarchy, where the two continuations operate as the success and failure continuations. It is shown in Fig. 9, on the left.

- The *fail* function shifts the success continuation, leaving only the failure continuation to be evaluated.
- The ambiguous choice *amb* operator first shifts the success continuation, and evaluates it twice with the two possible return values. The sequence gets pushed to the failure continuation.
- The *emit* function shifts both the success and failure continuations, adds a new value to the result list, and restores the shifted continuations. The emitted value is pushed to the third continuation.
- The *run* function resets two levels of continuations, freeing them to be used as the success and failure continuations. It then initializes the failure continuation to return the empty list, and the success continuation to emit the final value. Then it calls the body of the backtracking computation.

A program using these functions to solve the n-queens problem is shown in Fig. 10. The functions can be translated to $\lambda_{\$}$ using the translation of Fig. 7:

$$
\begin{aligned}
&\text{fail}\ () = \mathcal{S}_0 k_s.\langle()\rangle \\
&\text{amb}\ a\ b = \mathcal{S}_0 k_s.\langle \mathcal{S}_0 k_1.\mathcal{S}_0 k_2.(\lambda x.k_2\ \$\ k_1\ \$\ x)\ \$\ k_s\ \$\ a\ ; \\
&\qquad\qquad \mathcal{S}_0 k_1.\mathcal{S}_0 k_2.(\lambda x.k_2\ \$\ k_1\ \$\ x)\ \$\ k_s\ \$\ b\rangle \\
&\text{emit}\ a = \mathcal{S}_0 k_s.\mathcal{S}_0 k_f.\langle\langle a :: \\
&\quad \mathcal{S}_0 k_1.\mathcal{S}_0 k_2.\mathcal{S}_0 k_3.(\lambda x.k_3\ \$\ k_2\ \$\ k_1\ \$\ x)\ \$\ k_f\ \$\ k_s\ \$\ ()\rangle\rangle \\
&\text{run}\ f = \mathcal{S}_0 k_1.\mathcal{S}_0 k_2.\mathcal{S}_0 k_3.(\lambda x.k_3\ \$\ k_2\ \$\ x)\ \$ \\
&\quad \langle\langle \mathcal{S}_0 k_1.\mathcal{S}_0 k_2.(\lambda x.k_2\ \$\ k_1\ \$\ x)\ \$\ \langle \text{emit}\ (f\ ())\rangle; []\rangle\rangle
\end{aligned}
$$

The translation follows exactly the semantics of the original functions, but are needlessly complicated by the code ensuring that the appropriate contexts end up on the right positions on the context stack. If we relax those requirements, we can reimplement the four functions in a very similar form to the original definitions (Fig. 9, right).

Even though the definitions are similar, the semantics of this new implementation is very different. We demonstrate this below using abstract machine runs of the $\lambda^{\text{H}}_{\hookleftarrow}$ and $\lambda_{\$}$ versions of the *fail* and *emit* functions. It's easy to check the other two.

```
ambList [] = fail ()
ambList (h :: t) = amb h (ambList t)

nQueens n = let f 0 l = l
                f k l = let ok _ _ [] = true
                            ok v x (h :: t) =
                              v ≠ h and k + v − 1 ≠ x + h and
                                k − v − 1 ≠ x − h and ok v (x + 1) t
                        in let v = ambList [0..n − 1]
                           in if ok v k l
                              then f (k − 1) (v :: l)
                              else fail ()
            in run (λ().f n [])
```

Fig. 10. Example: solving the n-queens problem

– The *fail* function in $\lambda^{\mathrm{H}}_{\hookleftarrow}$ replaces the success continuation with the empty context, which is then activated before the failure continuation:

$$\langle \mathcal{S}_1 k_s.(), E_\mathrm{s}, E_\mathrm{f}, E_3\rangle_\mathrm{e} \Rightarrow \langle (), \bullet_1, E_\mathrm{f}, E_3\rangle_\mathrm{e} \Rightarrow \langle (), \bullet_1, E_\mathrm{f}, E_3\rangle_\mathrm{a} \Rightarrow \langle (), E_\mathrm{f}, E_3\rangle_\mathrm{a}$$

In $\lambda_\$$, the success continuation is removed, the failure continuation is run directly:

$$\langle \mathcal{S}_0 k_s.(), \hat{K}_\mathrm{s} \cdot \hat{K}_\mathrm{f} \cdot T\rangle_\mathrm{e} \Rightarrow \langle (), \hat{K}_\mathrm{f} \cdot T\rangle_\mathrm{e} \Rightarrow \langle \hat{K}_\mathrm{f}, (), T\rangle_\mathrm{a}$$

– The *emit* function in $\lambda^{\mathrm{H}}_{\hookleftarrow}$ resets the top two contexts when capturing the success and failure continuations, and these empty contexts are pushed to the third context, along with the emitted value, when restoring the success and failure continuations:

$$\begin{aligned}&\langle \mathcal{S}_2 k_s k_f.a :: (k_s, k_f) \hookleftarrow_2 (), E_\mathrm{s}, E_\mathrm{f}, E_3\rangle_\mathrm{e} \Rightarrow \langle a :: (E_\mathrm{s}, E_\mathrm{f}) \hookleftarrow_2 (), \bullet_1, \bullet_2, E_3\rangle_\mathrm{e}\\ &\Rightarrow \langle (E_\mathrm{s}, E_\mathrm{f}) \hookleftarrow_2 (), a :: \bullet_1, \bullet_2, E_3\rangle_\mathrm{e} \Rightarrow^* \langle (), E_\mathrm{s}, E_\mathrm{f}, E_3.(\bullet_2.(a :: \bullet_1))\rangle_\mathrm{a}\end{aligned}$$

In $\lambda_\$$, the emitted value is placed directly on the third context:

$$\begin{aligned}&\langle \mathcal{S}_0 k_s.\mathcal{S}_0 k_f.a :: (k_f \,\$\, k_s\,()), \hat{K}_\mathrm{s} \cdot \hat{K}_\mathrm{f} \cdot \hat{K} \cdot T\rangle_\mathrm{e}\\ &\Rightarrow^* \langle a :: (\hat{K}_\mathrm{f} \,\$\, \hat{K}_\mathrm{s}\,()), \hat{K} \cdot T\rangle_\mathrm{e} \Rightarrow \langle \hat{K}_\mathrm{f} \,\$\, \hat{K}_\mathrm{s}\,(), (a :: \hat{K}) \cdot T\rangle_\mathrm{e}\\ &\Rightarrow \langle \hat{K}_\mathrm{s}\,(), (\hat{K}_\mathrm{f}\$\bullet) \cdot (a :: \hat{K}) \cdot T\rangle_\mathrm{e} \Rightarrow^* \langle \hat{K}_\mathrm{s}, (), (\hat{K}_\mathrm{f}\$\bullet) \cdot (a :: \hat{K}) \cdot T\rangle_\mathrm{a}\end{aligned}$$

5.2 Sorting

In this section we show that there are programs in $\lambda_\$$ which can use an arbitrary number of contexts on the context stack, and therefore an analogous program with direct correspondence between the trail and the layered contexts cannot be written in the CPS hierarchy directly.[2] This would hold even for a hierarchy with no maximum level, because a program in the CPS hierarchy can access only a finite number of layers.

[2] The $\lambda^{\mathrm{H}}_{\hookleftarrow}$ language can macro-express $\lambda_\$$, because it contains the ordinary *shift*/*reset* operators, which were shown to be able to macro-express $\lambda_{\mathcal{S}_0}$ [15, 16]. However, such a simulation would not lead to a program of a structure resembling the one presented in Fig. 11.

```
csort l = let insert a =
              let cinsert k = S_0 f.case f [] of
                  [] → ⟨⟨a :: k ()⟩⟩
                  [b] → if a < b then cinsert (λx. f $ k x) else f $ ⟨a :: k ()⟩
              in S_0 k. cinsert k
          in ⟨⟨ foldr (λx.λl'. insert x; l' ) [] l ⟩⟩
```

Fig. 11. Example: insertion sort on the context stack

The example program, shown in Fig. 11, implements the insertion sort algorithm which uses the context stack to store and manipulate the output list. The main idea is that each element of the output list is contained inside a separate context on the context stack. The empty context marks the end of the output list. The algorithm runs the *insert* helper function for each element of the input list, and then returns [] to finally construct the result. The *insert* function shifts contexts from the context stack until it finds the right place for the element being inserted. Then it puts the element there in a new context and puts back the contexts it shifted.

The program uses $n + 2$ contexts on the context stack, where n is the size of the input list. So it cannot be typed using the type system in Fig. 3.

6 Conclusion and future work

We have made and formalized an observation that the delimited-control operators *shift*$_0$ and *reset*$_0$ are expressive enough to encode the control operators in the original CPS hierarchy. We have also shown that in programming practice, *shift*$_0$ and *reset*$_0$ can do the job of the CPS hierarchy and more. These results demonstrate a considerable expressive power of *shift*$_0$ and *reset*$_0$ on one hand. On the other hand, they open a possibility of using the presented translations, e.g., for implementing the hierarchy in terms of any implementation of *shift*$_0$ and *reset*$_0$.

Another step in the programme of investigating *shift*$_0$ and *reset*$_0$ is a realistic implementation on top of an existing language such as Scala, using the selective CPS translation we have devised in our previous work [15] along with Scala type annotations. Of more theoretical importance, we plan to build a behavioral theory for *shift*$_0$ and *reset*$_0$ following recent developments of the second author and Lenglet [6]. We are also in the course of constructing axiomatizations of *shift*$_0$ and *reset*$_0$ that are sound and complete with respect to the CPS translations (typed and untyped).

Acknowledgments. We thank Małgorzata Biernacka and the anonymous reviewers for valuable comments. This work has been funded by the Polish National Science Center, DEC-2011/03/B/ST6/00348.

References

1. Biernacka, M., Biernacki, D.: Context-based proofs of termination for typed delimited-control operators. In López-Fraguas, F.J., ed.: Proceedings of the 11th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09), Coimbra, Portugal, ACM Press (September 2009)

2. Biernacka, M., Biernacki, D., Danvy, O.: An operational foundation for delimited continuations in the CPS hierarchy. Logical Methods in Computer Science **1**(2:5) (November 2005) 1–39
3. Biernacka, M., Biernacki, D., Lenglet, S.: Typing control operators in the CPS hierarchy. In Hanus, M., ed.: Proceedings of the 13th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'11), Odense, Denmark, ACM Press (July 2011)
4. Biernacka, M., Danvy, O.: A syntactic correspondence between context-sensitive calculi and abstract machines. Theoretical Computer Science **375**(1-3) (2007) 76–108
5. Biernacki, D., Danvy, O., Millikin, K.: A dynamic continuation-passing style for dynamic delimited continuations. Technical Report BRICS RS-06-15, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark (October 2006)
6. Biernacki, D., Lenglet, S.: Applicative bisimulations for delimited-control operators. In Birkedal, L., ed.: Foundations of Software Science and Computation Structures, 15th International Conference, FOSSACS 2012. Number 7213 in Lecture Notes in Computer Science, Tallinn, Estonia, Springer-Verlag (March 2012) 119–134
7. Calvès, C.: Complexity and Implementation of Nominal Algorithms. PhD thesis, King's College London, London, England (2010)
8. Danvy, O., Filinski, A.: Abstracting control. In Wand, M., ed.: Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, France, ACM Press (June 1990) 151–160
9. Filinski, A.: Representing monads. In Boehm, H.J., ed.: Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, Oregon, ACM Press (January 1994) 446–457
10. Kameyama, Y., Kiselyov, O., Shan, C.: Shifting the stage: Staging with delimited control. In Puebla, G., Vidal, G., eds.: Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2009), Savannah, GA, ACM Press (January 2009) 111–120
11. Kiselyov, O.: Delimited control in OCaml, abstractly and concretely: System description. In Blume, M., Vidal, G., eds.: Functional and Logic Programming, 10th International Symposium, FLOPS 2010. Number 6009 in Lecture Notes in Computer Science, Sendai, Japan, Springer (April 2010) 304–320
12. Kiselyov, O., Shan, C.: A substructural type system for delimited continuations. In Rocca, S.R.D., ed.: Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007. Number 4583 in Lecture Notes in Computer Science, Paris, France, Springer-Verlag (June 2007) 223–239
13. Kiselyov, O., Shan, C.: Embedded probabilistic programming. In Taha, W., ed.: Domain-Specific Languages, DSL 2009. Number 5658 in Lecture Notes in Computer Science, Oxford, UK, Springer (July 2009) 360–384
14. Masuko, M., Asai, K.: Direct implementation of shift and reset in the MinCaml compiler. In Rossberg, A., ed.: Proceedings of the ACM SIGPLAN Workshop on ML, ML'09, Edinburgh, UK (August 2009) 49–60
15. Materzok, M., Biernacki, D.: Subtyping delimited continuations. In Danvy, O., ed.: Proceedings of the 2011 ACM SIGPLAN International Conference on Functional Programming (ICFP'11), Tokyo, Japan, ACM Press (September 2011) 81–93
16. Shan, C.: A static simulation of dynamic delimited control. Higher-Order and Symbolic Computation **20**(4) (2007) 371–401