Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Subtyping Delimited Continuations

Marek Materzok · Dariusz Biernacki

the date of receipt and acceptance should be inserted later

Abstract We present a type system with subtyping for first-class delimited continuations that generalizes Danvy and Filinski's type system for shift and reset by maintaining explicit information about the types of contexts in the metacontext. We exploit this generalization by considering the control operators known as shift_0 and reset_0 that can access contexts arbitrarily deep in the metacontext. We use subtyping to control the level of information about the metacontext the expression actually requires and in particular to coerce pure expressions into effectful ones. For this type system we prove strong type soundness and termination of evaluation and we present a provably correct type reconstruction algorithm. We also introduce two CPS translations for shift_0 and reset_0: one targeting the untyped lambda calculus, and another—type-directed—targeting the simply typed lambda calculus. The latter translation preserves typability and is selective in that it leaves pure expressions in direct style.

Keywords delimited continuation, continuation-passing style, type system, subtyping

1 Introduction

1.1 Delimited-control operators shift and reset

Control operators for first-class delimited continuations, introduced independently by Felleisen [17] and by Danvy and Filinski [13,14], serve to abstract control in functional programming languages by reifying the current continuation as a first-class value. However, in contrast to the well-known abortive control operator callcc present

This is a revised and expanded version of "Subtyping delimited continuations" presented at the 16th ACM SIGPLAN International Conference on Functional Programming, Tokyo, Japan, September 2011 [28].

Marek Materzok
Institute of Computer Science, University of Wrocław, Wrocław, Poland
E-mail: tilk@tilk.eu

Dariusz Biernacki
Institute of Computer Science, University of Wrocław, Wrocław, Poland
E-mail: dabi@cs.uni.wroc.pl

in Scheme [24] and SML/NJ [21], delimited-control operators model composition of delimited (composable, partial) continuations rather than jumps to undelimited continuations. In particular, while callcc offers the programmer the full power of the continuation-passing style (CPS) in direct style (DS), Danvy and Filinski's shift and reset make it possible to express the continuation-composing style (CCS) patterns in DS that are characteristic of the success-failure continuation model of backtracking [14].

It has been shown by Filinski that shift and reset play an important role among the most fundamental concepts in the landscape of eager functional programming—a language equipped with shift and reset is monadically complete, i.e., any computational monad can be represented in direct style with the aid of these control operators [19, 20]. A long list of non-trivial applications of delimited continuations includes normalization by evaluation and partial evaluation [3, 12, 15], mobile computing [38], linguistics [4], and operating systems [26] to name but a few.

The presence of a control delimiter, such as reset, in a programming language is reflected in the structure of the continuation that represents the rest of the computation at a given point of program execution. Namely, the continuation is split into the current delimited continuation that extends up to the nearest dynamically enclosing reset, and a metacontinuation that represents the rest of the program beyond this delimiter. Originally, the semantics of shift and reset was given in terms of a metacontinuation-passing evaluator [14]. Defunctionalizing this evaluator led Biernacka et al. to an abstract machine that operates on first-order representations of the continuation and metacontinuation that are called a context and a metacontext, respectively [7]. It is convenient to view the two layers of contexts as stacks, where the context is a stack of elementary contexts [18], and the metacontext is a stack of contexts.

Then, the operational semantics of the control operators shift ($\mathcal{S}$) and reset ($\langle\rangle$) can be explained in terms of the abstract machine as follows. Evaluation of the expression $\mathcal{S}k.e$ consists in:

1. binding the variable k to the current delimited context;
2. evaluating the expression e in the empty delimited context and with the newly created binding for k.

Resuming a captured context consists in:

1. pushing the current delimited context on top of the metacontext;
2. installing the captured context as the current delimited context.

Evaluation of the expression $\langle e\rangle$ consists in:

1. pushing the current delimited context on top of the metacontext;
2. evaluating the expression e in the empty delimited context.

When a value has been computed and the current delimited context is empty, the metacontext is popped and the top-most context becomes the current delimited context for the computed value.

For example, the following expression (where $++$ is string concatenation):

$$\textit{“Alice”} \mathbin{++} \langle \textit{“ has ”} \mathbin{++} (\mathcal{S}k.(k\ \textit{“a dog ”}) \mathbin{++} \textit{“and the dog”} \mathbin{++} (k\ \textit{“a cat.”}))\rangle$$

evaluates to the string *“Alice has a dog and the dog has a cat.”* Here, the captured context represents the expression $\langle \textit{“ has ”} ++ [\ ]\rangle$, where the hole $[\ ]$ is filled with *“a dog ”* and *“a cat.”*, upon the two applications of the captured continuation.

The CPS structure underlying shift and reset induces a type-and-effect system due to Danvy and Filinski that is the most expressive of the known monomorphic type systems for shift and reset [13]. Judgments in Danvy and Filinski's type system have the form $\Gamma; B \vdash e : A; C$, with the interpretation that expression e can be plugged into a context expecting a value of type A and producing a value of type B, and a metacontext expecting a value of type C, where B and C can be different types. Hence, expressions are allowed to change the answer type of the context in which they are embedded. For example, the expression $\langle 1 + \mathcal{S}k.\ true \rangle$ is well typed in this type system.

1.2 From shift to shift_0

In their pioneering article on delimited continuations [13], Danvy and Filinski briefly described a variant of shift and reset, known as shift_0 ($\mathcal{S}_0$) and reset_0 (also noted $\langle\rangle$). The operational semantics of the control delimiter reset_0 coincides with that of reset. However, evaluation of the expression $\mathcal{S}_0 k.e$ differs from evaluation of the expression $\mathcal{S}k.e$ in that the expression e is not evaluated in the empty delimited context, but in the top-most delimited context pending on the metacontext. In other words, in contrast to shift, shift_0 removes the control delimiter along with the captured context, which makes it possible to use shift_0 to access delimited contexts arbitrarily deep in the metacontext by repeatedly shifting contexts.

For example, the following expression:

$$\langle \text{“}Alice\text{”} +\!\!+ \langle \text{“ }has\text{ ”} +\!\!+ (\mathcal{S}_0 k_1.\mathcal{S}_0 k_2.\text{“}A\ cat\text{”} +\!\!+ (k_1(k_2\ \text{“.”})))\rangle\rangle$$

evaluates to the string *“A cat has Alice.”*. Here, k_1 is bound to a context representing the expression $\langle \text{“ }has\text{ ”} +\!\!+ [\,]\rangle$, and k_2 is bound to the context representing the expression $\langle \text{“}Alice\text{”} +\!\!+ [\,]\rangle$.

The ability to access delimited contexts arbitrarily deep in the metacontext makes shift_0 a particularly interesting and powerful control operator. Surprisingly, compared to shift, it has not received much attention in the literature so far.

In the untyped setting, shift_0 and reset_0 have been investigated by Shan [37], who presented a CPS translation for shift_0 and reset_0 that conservatively extends Plotkin's call-by-value CPS translation [33] and that leads to a simulation of shift_0 and reset_0 in terms of shift and reset. This CPS translation and simulation hinge on the requirement that the continuations be represented by recursive functions taking a list of continuations as one of their arguments. (The converse simulation is trivial—one simply resets the body of each shift_0 expression.)

In a typed setting, Kiselyov and Shan [27] introduced a substructural type system for shift_0 and reset_0 that abstractly interprets (in the sense of abstract interpretation of Cousot and Cousot [11]) small-step reduction semantics of the language. Their type system allows for continuation answer type modifications and, in a sense, extends Danvy and Filinski's type system.

1.3 Contributions of this work

In this article we present a new type-and-effect system for shift_0 and reset_0 that generalizes Danvy and Filinski's original type system for shift and reset. To this end, we

first introduce two new CPS translations for shift_0 and reset_0: a curried one that conservatively extends Plotkin's call-by-value CPS translation and an uncurried one that requires list structure and list operations in the target language. From the uncurried CPS translation, we derive a new abstract machine that along with the standard reduction semantics forms an operational foundation for shift_0. From the curried CPS translation, we derive a type-and-effect system à la Danvy and Filinski which we refine by allowing for subtyping of types and effect annotations. We show that the type system with subtyping we present satisfies several crucial properties such as strong type soundness [39] and termination of evaluation, and we describe and prove correct a type inference algorithm for this type system.

Compared to Kiselyov and Shan's type system [27], the one presented here is more conventional; it corresponds directly to the CPS translation it has been obtained from, and it heavily relies on subtyping built into the system. Furthermore, we take advantage of the fact that our type system distinguishes between pure and effectful expressions and we present a selective type-directed CPS translation from the language with shift_0 and reset_0 to the simply typed lambda calculus that transforms to CPS only effectful expressions, leaving pure ones in direct style.

We also show that Danvy and Filinski's original type system for shift and reset can naturally be embedded into our type system, and that when restricted, our type system gives rise to a type system à la Danvy and Filinski with subtyping for shift and reset. Similarly, the untyped CPS translation for shift_0 and reset_0 that we present induces a new CPS translation for shift and reset and a new simulation of shift_0 and reset_0 in terms of shift and reset.

It is worth stressing that the subtyping mechanism of our type system addresses one of the subtle limitations of Danvy and Filinski's type system concerning the typing rules for shift and λ-abstraction:

$$\frac{\Gamma, f : A\ {}_F\!\!\rightarrow_F B; E \vdash e : E; C}{\Gamma; B \vdash \mathcal{S} f.e : A; C}$$

$$\frac{\Gamma, x : A; C \vdash e : B; D}{\Gamma; E \vdash \lambda x.e : A\ {}_C\!\!\rightarrow_D B; E}$$

According to the above rule for shift, the type system assigns the captured continuation a functional type with a single, chosen up-front answer type (F), which prevents it from being resumed in contexts with a different answer type. But the continuations captured by shift do not modify the context of their resumption, and therefore they should be applicable in any context. One way to deal with this shortcoming is to introduce polymorphism into the language, as in Asai and Kameyama's work [2]. In their type system, the continuation captured by shift is assigned a functional type with polymorphic answer type. Another approach is to distinguish between continuations and functions in such a way that a captured continuation is given a type which does not mention the answer type [6]. In this approach, additional syntactic construct—a continuation application, distinct from function application—has to be added to the language. Unfortunately, in this type system, functions without control effects (when C and D are the same arbitrary type in the rule for λ-abstraction) still have an answer type chosen up-front and are subject to the same restrictions as described above.

In a way, one can treat the continuation types studied by Biernacka and Biernacki [6] as more general than function types—continuations can be applied in any context, whereas functions can only be applied in contexts with matching answer types.

The type system of this article removes the syntactic distinction between continuations and functions by means of subtyping effect annotations. The subtyping relation allows functions to be called when more is known about the types of the contexts in the metacontext than the function actually requires. In particular, a pure function, possibly representing a captured continuation, can be arbitrarily coerced into an effectful one.

1.4 Outline of the rest of the article

The rest of this article is structured as follows. In Section 2, we introduce the syntax and reduction semantics for the call-by-value λ-calculus with shift_0 and reset_0. We then present two CPS translations for the language and we relate them to the reduction semantics. We also present an abstract machine corresponding to the uncurried CPS translation. Next, we derive a type-and-effect system à la Danvy and Filinski from the curried CPS translation. In Section 3, we introduce subtyping to the type system and we prove several standard properties for the system, including strong type soundness. Then, we use a context-based method of reducibility predicates to prove termination of evaluation of well-typed programs. We subsequently present and prove correct a type inference algorithm for our type system. Finally, we present a selective type-directed CPS translation for the language. In Section 4, we consider some practical applications of the presented language. In Section 5, we show how the type system for shift_0 and reset_0 can be restricted to Danvy and Filinski's type system for shift and reset, and we discuss the CPS translations for shift and reset induced by the presented CPS translations for shift_0 and reset_0. We also describe a new static simulation of shift_0 in terms of shift. In Section 6, we discuss related work and we conclude in Section 7.

Most of the theorems presented in this paper were formalized in the Twelf system [32]. The only exception is the normalization theorem that hinges on the concept of logical relation that is not supported by Twelf.

2 The language $\lambda^{\mathcal{S}_0}$

2.1 Syntax and semantics

In this section, we define the language we will be working with. The language, which we call $\lambda^{\mathcal{S}_0}$, is the call-by-value λ-calculus extended with the two control operators shift_0 ($\mathcal{S}_0$) and reset_0 ($\langle\dot{\rangle}$).

We introduce three syntactic categories of terms, evaluation contexts and trails (i.e., stacks of evaluation contexts that form a particular prefix of the metacontext, as defined later in this section), where term variables are ranged over by x and f:

$$\begin{array}{lll} \textit{values} & v & ::= \lambda x.e \mid x \\ \textit{terms} & e & ::= v \mid e\,e \mid \langle e \rangle \mid \mathcal{S}_0 f.e \\ \textit{contexts} & K & ::= \bullet \mid K\,e \mid v\,K \\ \textit{trails} & T & ::= \Box \mid K \cdot T \end{array}$$

We use the notation $T \cdot T'$ for trail concatenation, defined as follows:

$$\Box \cdot T = T \qquad (K \cdot T) \cdot T' = K \cdot (T \cdot T')$$

Contexts and trails are represented inside-out, which is formalized by the following definition of the plugging of a term into a context and a trail:

$$\begin{aligned} \bullet[e] &= e & \Box[e] &= e \\ (K\, e')[e] &= K[e\, e'] & (K \cdot T)[e] &= T[\langle K[e] \rangle] \\ (v\, K)[e] &= K[v\, e] \end{aligned}$$

There are three contraction rules:

$$\begin{aligned} (\lambda x.e)v &\rightsquigarrow e\{v/x\} \\ \langle v \rangle &\rightsquigarrow v \\ \langle K[\mathcal{S}_0 f.e] \rangle &\rightsquigarrow e\{\lambda x.\langle K[x] \rangle / f\} \end{aligned}$$

where $e\{v/x\}$ stands for the usual capture-avoiding substitution of v for x in e. A term matching the left-hand side of a contraction rule is called a redex.

The reduction rule for shift_0 differs from the following rule for shift:

$$\langle K[\mathcal{S} f.e] \rangle \rightsquigarrow \langle e\{\lambda x.\langle K[x] \rangle / f\} \rangle$$

in that it removes the reset_0 at the root of the redex. This small difference enables the shift_0 operator to inspect the entire context stack, where shift has access only to the topmost context. In Section 4.3, we exhibit an interesting programming example that takes advantage of this difference in practice. Furthermore, it has a big impact on the type systems that we present later in the article.

The unique decomposition lemma is stated as follows:

Lemma 1 (Unique decomposition) *Any closed $\lambda^{\mathcal{S}_0}$ term either is a value, or it is a stuck term of the form $K[\mathcal{S}_0 f.e]$, or it can be uniquely represented as $K[T[e]]$, where e is a redex.*

Finally, we define the reduction relation on terms representing programs as follows:

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

The context K (at the bottom of the stack) is the only one not delimited by a reset_0. As we shall see later, having this last context distinguished from the others is important from the viewpoint of the type system, because it can be assigned a simpler type than the other contexts. A trail T along with the distinguished evaluation context K will be called a metacontext in the rest of this article. Notice that in this approach the current delimited context is the top-most element of the metacontext, and therefore the meaning of the metacontext here is slightly different from the standard one, introduced for shift and reset [7].

$$
\begin{aligned}
[\![x]\!] &= \lambda k.k\,x \\
[\![\lambda x.e]\!] &= \lambda k.k\,(\lambda x.[\![e]\!]) \\
[\![e_1 e_2]\!] &= \lambda k.[\![e_1]\!](\lambda f.[\![e_2]\!](\lambda x.f\,x\,k)) \\
[\![\langle e \rangle]\!] &= [\![e]\!](\lambda x.\lambda k.k\,x) \\
[\![\mathcal{S}_0 f.e]\!] &= \lambda f.[\![e]\!]
\end{aligned}
$$

Fig. 1 Untyped CPS translation for $\lambda^{\mathcal{S}_0}$

$$
\begin{aligned}
[\![x]\!]^{@} &= \lambda^l(k:ks).k\,x\,@\,ks \\
[\![\lambda x.e]\!]^{@} &= \lambda^l(k:ks).k\,(\lambda x.[\![e]\!]^{@})\,@\,ks \\
[\![e_1 e_2]\!]^{@} &= \lambda^l(k:ks).[\![e_1]\!]^{@}\,@\,((\lambda f.\lambda^l ks'.[\![e_2]\!]^{@}\,@\,((\lambda x.\lambda^l ks''.f\,x\,@\,(k:ks''))\,:\,ks'))\,:\,ks) \\
[\![\langle e \rangle]\!]^{@} &= \lambda^l(k:ks).[\![e]\!]^{@}\,@\,((\lambda x.\lambda^l(k':ks').k'\,x\,@\,ks')\,:\,k:ks) \\
[\![\mathcal{S}_0 f.e]\!]^{@} &= \lambda^l(f:k:ks).[\![e]\!]^{@}\,@\,(k:ks)
\end{aligned}
$$

Fig. 2 Untyped uncurried CPS translation for $\lambda^{\mathcal{S}_0}$

2.2 CPS translations

The shift_0 control operator allows access to every context in the trail. What is more, unlike with shift and reset whose CPS translation reads as follows [14]:

$$
\begin{aligned}
[\![\mathcal{S} f.e]\!] &= \lambda k.[\![e]\!]\{\lambda x.\lambda k'.k'(k\,x)/f\}(\lambda x.x) \\
[\![\langle e \rangle]\!] &= \lambda k.k([\![e]\!](\lambda x.x))
\end{aligned}
$$

control effects in a captured context are not isolated—the captured context, when applied, can capture contexts present at the invocation site. This ability must be reflected in the CPS translation for the shift_0 operator. In the translation presented by Shan [37], the trail is represented explicitly as a list of contexts which has to be passed to every continuation. In our approach, the contexts below the current one are captured by additional lambda abstractions. The translation for the shift_0 operator captures a context using a lambda abstraction, and the translation for reset_0 introduces a new context represented by the CPS-translated identity continuation. Hence, just like in Shan's work, our CPS accounting for shift_0 relies on continuations accepting continuations as parameters. The entire CPS translation is shown in Figure 1. Evaluating a CPS-translated program requires an initial continuation $\lambda x.x$. The translation preserves the semantics:

Theorem 1 *If* $e_1 \to^* e_2$ *in* $\lambda^{\mathcal{S}_0}$*, then* $[\![e_1]\!] =_{\beta\eta} [\![e_2]\!]$ *in* λ*.*

We note the striking simplicity of the translation and the fact that the control operator shift_0 is trivially turned into a lambda abstraction, expecting a continuation as an argument. This CPS translation can be modified so that every translated term takes all its continuation parameters in one list parameter. Figure 2 presents such an uncurried CPS translation that targets the untyped lambda-calculus extended with a separate syntactic category of lists (with the standard list constructors [ ] and :) and two new kinds of expressions: list abstraction λ^l and list application @. Evaluating a CPS-translated program requires a list whose only element is the continuation $\lambda x.\lambda^l[\,].x$.

$$
\begin{array}{rl}
\text{Environments:} & \rho ::= \rho_{\text{init}} \mid \rho[x \mapsto v] \\
\text{Values:} & v ::= [x, e, \rho] \mid E \\
\text{Contexts:} & E ::= \mathsf{End} \mid \mathsf{Arg}(e, \rho, E) \mid \mathsf{Fun}(v, E) \\
\text{Metacontexts:} & F ::= \Box \mid E : F
\end{array}
$$

$$
\begin{array}{rl}
\text{Initial configurations:} & \langle e, \rho_{\text{init}}, \mathsf{End} : \Box \rangle_{\text{eval}} \\
\text{Final configurations:} & \langle \mathsf{End}, v, \Box \rangle_{\text{cont}}
\end{array}
$$

$$
\begin{array}{rcl}
\langle x, \rho, E : F \rangle_{\text{eval}} & \Rightarrow & \langle E, \rho(x), F \rangle_{\text{cont}} \\
\langle \lambda x.e, \rho, E : F \rangle_{\text{eval}} & \Rightarrow & \langle E, [x, e, \rho], F \rangle_{\text{cont}} \\
\langle e_1 e_2, \rho, E : F \rangle_{\text{eval}} & \Rightarrow & \langle e_1, \rho, \mathsf{Arg}(e_2, \rho, E) : F \rangle_{\text{eval}} \\
\langle \mathcal{S}_0 f.e, \rho, E : E' : F \rangle_{\text{eval}} & \Rightarrow & \langle e, \rho[f \mapsto E], E' : F \rangle_{\text{eval}} \\
\langle \langle e \rangle, \rho, E : F \rangle_{\text{eval}} & \Rightarrow & \langle e, \rho, \mathsf{End} : E : F \rangle_{\text{eval}} \\
\langle \mathsf{End}, v, E : F \rangle_{\text{cont}} & \Rightarrow & \langle E, v, F \rangle_{\text{cont}} \\
\langle \mathsf{Arg}(e, \rho, E), v, F \rangle_{\text{cont}} & \Rightarrow & \langle e, \rho, \mathsf{Fun}(v, E) : F \rangle_{\text{eval}} \\
\langle \mathsf{Fun}(E, E'), v, F \rangle_{\text{cont}} & \Rightarrow & \langle E, v, E' : F \rangle_{\text{cont}} \\
\langle \mathsf{Fun}([x, e, \rho], E), v, F \rangle_{\text{cont}} & \Rightarrow & \langle e, \rho[x \mapsto v], E : F \rangle_{\text{eval}}
\end{array}
$$

Fig. 3 Abstract machine for $\lambda^{\mathcal{S}_0}$

2.3 Abstract machine

The uncurried CPS translation of the previous section underlies a continuation-passing style evaluator that, when defunctionalized [34,1], gives rise to the abstract machine of Figure 3. This abstract machine extends Felleisen and Friedman's CEK abstract machine [18] and can be seen to be equivalent with the abstract machine for $\mathsf{shift}_0/\mathsf{reset}_0$ described by Biernacki et al. [10]. (An alternative way to obtain an abstract machine for the language we consider would be to follow Biernacka and Danvy's syntactic correspondence between context-based reduction semantics with explicit substitutions and abstract machines [8,9].) Had we decided to translate terms in such a way that the head and tail of the continuation list are passed as separate arguments, we would have obtained a CPS translation that coincides with Shan's [37] and which corresponds exactly to the abstract machine of Biernacki et al. [10]. (The intuitive semantics of shift_0 presented in Section 1 is actually phrased in terms of such an abstract machine.)

The abstract machine makes use of the environment mapping variables to values (closures and captured contexts) and it implements the eval/continue model, where the transitions from the eval-configurations interpret the expressions of the language and the transitions from the cont-configurations interpret the stack of the abstract machine. The contexts E are the standard call-by-value evaluation contexts represented as stacks of elementary evaluation contexts. The metacontexts F are stacks of contexts.

Notice that the transition for shift_0 requires the rest of the metacontext below the current delimited context to be non-empty. This requirement coincides with the observation of Section 2.1 that the context at the bottom of the metacontext has a special status, in particular it is not allowed to be captured. For this reason, in the grammar of contexts we could actually replace the empty context End with two such contexts Stop and Pop. Then Stop would be used only as a termination mark of the bottom-most context, namely in the initial and final configurations when the tail of the metacontext is always empty, whereas Pop would be introduced and eliminated

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

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

$$\frac{\Gamma, f : \tau_1 \xrightarrow{\sigma_1} \tau_2 \vdash e : \tau_3\ \sigma_2}{\Gamma \vdash \mathcal{S}_0 f.e : \tau_1\ [\tau_2\ \sigma_1]\ \tau_3\ \sigma_2}\ \text{SHIFT0} \qquad \frac{\Gamma \vdash e : \tau'\ [\tau'\ [\tau''\ \sigma'']\ \tau''\ \sigma'']\ \tau\ \sigma}{\Gamma \vdash \langle e \rangle : \tau\ \sigma}\ \text{RESET0}$$

Fig. 4 Type system for $\lambda^{\mathcal{S}_0}$ without subtyping

in the transitions for reset_0 and End, respectively, when the tail of the metacontext is never empty. Indeed, in the uncurried CPS translation there are two different 'initial' continuations: one in the clause for reset_0, corresponding to Pop and one at the top level, corresponding to Stop.

2.4 From CPS translation to a type system

Using the approach of Danvy and Filinski [13], we build a type system based on the curried CPS translation. Let us thus limit our attention to those terms of $\lambda^{\mathcal{S}_0}$, whose translations are well-typed terms in $\lambda_{\to}$. One can then easily see that the types for variables, lambda abstractions and application must have the same form as in the type system of Danvy and Filinski. And what types should we assign to shift_0 and reset_0? Suppose that $[\![e]\!]$ has type τ, then $[\![\mathcal{S}_0 f.e]\!] = \lambda f.[\![e]\!]$ must have the type $\tau' \to \tau$, where τ' is the type of the captured continuation. For reset_0, for $[\![\langle e \rangle]\!] = [\![e]\!]\,(\lambda x.\lambda k.kx)$ to have type τ, $[\![e]\!]$ has to have a type of the form $(\tau' \to (\tau' \to \tau'') \to \tau'') \to \tau$. Thus every use of shift_0 introduces a new function arrow into the type, and every use of reset_0 removes one. This leads us to a recursive definition of effect annotations, which in a sense "remembers" the number of nested uses of shift_0 inside a term.

We now introduce the syntactic categories of types and effect annotations, where α ranges over type variables and ε stands for the empty annotation:

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

The resulting type system for $\lambda^{\mathcal{S}_0}$ is shown in Figure 4. We introduce a typing judgment of the form $\Gamma \vdash e : \tau\ \sigma$, which means that e evaluates to a value of type τ, possibly executing control effects described by the type annotation σ. The type environment Γ associates a pure type τ (without any annotation) with every variable mentioned.

The type annotations require some explanation. Their meaning is best understood together with types they annotate. The typing judgment $\Gamma \vdash e : \tau_1'[\tau_1\sigma_1] \ldots \tau_n'[\tau_n\sigma_n]\tau\varepsilon$ can be read as follows: "the term e in a typing context Γ may evaluate to a value of type τ when plugged in a trail of contexts of types $\tau_1' \xrightarrow{\sigma_1} \tau_1, \ldots, \tau_n' \xrightarrow{\sigma_n} \tau_n$." As a special case, $\Gamma \vdash e : \tau$ means: "the term e in a typing context Γ may evaluate to a value of type τ without observable control effects." Function types also have effect annotations, which can be thought of as associated with the return type.

In order to see the difference with shift and reset, we rephrase the original typing rules for these control operators using the notation of the type system for shift_0 and reset_0:

$$\frac{\Gamma, f : \tau_1 \xrightarrow{[\tau'\ \varepsilon]\ \tau'\ \varepsilon} \tau_2 \vdash e : \tau\ [\tau\ \varepsilon]\ \tau_3\ \varepsilon}{\Gamma \vdash \mathcal{S} f.e : \tau_1\ [\tau_2\ \varepsilon]\ \tau_3\ \varepsilon}\ \textsc{shift} \qquad \frac{\Gamma \vdash e : \tau''\ [\tau''\ \varepsilon]\ \tau\ \varepsilon}{\Gamma \vdash \langle e \rangle : \tau\ [\tau'\ \varepsilon]\ \tau'\ \varepsilon}\ \textsc{reset}$$

For brevity and improved readability, we omit the empty annotations ε in the rest of the paper. For example, we will write the type $\tau_1\ [\tau_2\ \varepsilon]\ \tau_3\ \varepsilon$ from the rules above as $\tau_1\ [\tau_2]\ \tau_3$. This introduces no syntactic ambiguity.

This type system preserves types as stated in Theorem 2 below, where translations on types and annotated types are defined as follows:

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![\tau_1 \xrightarrow{\sigma} \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2\ \sigma]\!] \\ [\![\tau\ \varepsilon]\!] &= [\![\tau]\!] \\ [\![\tau\ [\tau_1\ \sigma_1]\ \tau_2\ \sigma_2]\!] &= ([\![\tau]\!] \to [\![\tau_1\ \sigma_1]\!]) \to [\![\tau_2\ \sigma_2]\!] \end{aligned}$$

Theorem 2 *If $\Gamma \vdash e : \tau\ \sigma$, then $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau\ \sigma]\!]$ in $\lambda_{\to}$.*

While interesting, this type system is very restrictive. For example, the term

$$\lambda x.\mathcal{S}_0 f.f\ \langle f\ x \rangle$$

is translated (with administrative redexes reduced away for clarity) to

$$\lambda k.k\ (\lambda x.\lambda f.\lambda k.f\ x\ (\lambda y.\lambda k'.k'\ y)\ (\lambda y.f\ y\ k))$$

In this term, the two occurrences of f constrain its type to the form $\tau_1 \to \tau_2 \to \tau_f$, where $\tau_f = (\tau_1 \to \tau_f) \to \tau_3$—so the term cannot be given a type in the simply typed lambda calculus. The reason is that the type of functions restricts their use to only those locations in a term, where the number and types of contexts match exactly. This is an unnecessary restriction—a function which requires the top n contexts to be of some type can obviously be run safely when more contexts are present, if the other contexts can be composed together and accept the function's return value. This observation leads us to another, much more expressive type system, which we present in the next section.

3 The type system with subtyping

3.1 The type system $\lambda^{\mathcal{S}_0}_{\leq}$

The type system with subtyping for $\lambda^{\mathcal{S}_0}$, called $\lambda^{\mathcal{S}_0}_{\leq}$, is shown in Figure 5. In this type system, the term $\lambda x.\mathcal{S}_0 f.f\ \langle f\ x \rangle$, which could not be typed before, is well typed and can be assigned the type $\alpha \xrightarrow{[\alpha]\ \alpha} \alpha$:

$$\frac{}{\varepsilon \le \varepsilon}\ \text{S-EPS} \qquad \frac{\tau\ \sigma \le \tau'\ \sigma'}{\varepsilon \le [\tau\ \sigma]\ \tau'\ \sigma'}\ \text{S-PURE} \qquad \frac{\tau_2\ \sigma_2 \le \tau_1\ \sigma_1 \qquad \tau_1'\ \sigma_1' \le \tau_2'\ \sigma_2'}{[\tau_1\ \sigma_1]\ \tau_1'\ \sigma_1' \le [\tau_2\ \sigma_2]\ \tau_2'\ \sigma_2'}\ \text{S-NPURE}$$

$$\frac{\tau \le \tau' \qquad \sigma \le \sigma'}{\tau\ \sigma \le \tau'\ \sigma'}\ \text{S-STR} \qquad \frac{}{\alpha \le \alpha}\ \text{S-VAR} \qquad \frac{\tau_2' \le \tau_1' \qquad \tau_1\ \sigma_1 \le \tau_2\ \sigma_2}{\tau_1' \xrightarrow{\sigma_1} \tau_1 \le \tau_2' \xrightarrow{\sigma_2} \tau_2}\ \text{S-FUN}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma \vdash e : \tau\ \sigma \qquad \tau\ \sigma \le \tau'\ \sigma'}{\Gamma \vdash e : \tau'\ \sigma'}\ \text{SUB}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2\ \sigma}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\sigma} \tau_2}\ \text{ABS} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ \text{APP-PURE}$$

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

$$\frac{\Gamma, f : \tau_1 \xrightarrow{\sigma_1} \tau_2 \vdash e : \tau_3\ \sigma_2}{\Gamma \vdash \mathcal{S}_0 f.e : \tau_1\ [\tau_2\ \sigma_1]\ \tau_3\ \sigma_2}\ \text{SHIFT0} \qquad \frac{\Gamma \vdash e : \tau'\ [\tau']\ \tau\ \sigma}{\Gamma \vdash \langle e \rangle : \tau\ \sigma}\ \text{RESET0}$$

Fig. 5 Type system for $\lambda^{\mathcal{S}_0}$ with subtyping ($\lambda^{\mathcal{S}_0}_{\le}$)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \alpha \to \alpha}\ \text{VAR} \qquad \dfrac{\dfrac{}{\Gamma \vdash f : \alpha \to \alpha}\ \text{VAR} \qquad \dfrac{}{\Gamma \vdash x : \alpha}\ \text{VAR}}{\Gamma \vdash f\ x : \alpha}\ \text{APP}}{\Gamma = x : \alpha, f : \alpha \to \alpha \vdash f\ \langle f\ x \rangle : \alpha}\ \text{SHIFT0}}{x : \alpha \vdash \mathcal{S}_0 f.f\ \langle f\ x \rangle : \alpha\ [\alpha]\ \alpha}\ \text{ABS}}{\vdash \lambda x.\mathcal{S}_0 f.f\ \langle f\ x \rangle : \alpha \xrightarrow{[\alpha]\ \alpha} \alpha}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \alpha \to \alpha}\ \text{VAR} \qquad \dfrac{\dfrac{}{\Gamma \vdash f : \alpha \to \alpha}\ \text{VAR} \qquad \dfrac{}{\Gamma \vdash x : \alpha}\ \text{VAR}}{\Gamma \vdash f\ x : \alpha}\ \text{APP} \qquad \cdots}{\Gamma \vdash f\ x : \alpha\ [\alpha]\ \alpha}\ \text{SUB}}{\Gamma \vdash \langle f\ x \rangle : \alpha}\ \text{RESET0}}{\Gamma = x : \alpha, f : \alpha \to \alpha \vdash f\ \langle f\ x \rangle : \alpha}\ \text{APP}$$

We define three subtyping relations: one defined on types, one on effect annotations, and (for convenience) one on pairs of types and effect annotations.

Lemma 2 *The subtyping relations are partial orders—they are reflexive, transitive and weakly antisymmetric.*

All subtyping rules are rather straightforward, except the rule S-PURE, which may seem non-intuitive. To get some understanding of what this rule means, let us consider an effect-annotated type where the contexts are effect-free (type annotations inside the square brackets are equal to ε and are thus omitted): $\tau_1\ [\tau_1']\ \ldots \tau_n\ [\tau_n']\ \tau_f$. This effect-annotated type means that we have n evaluation contexts on the trail with types $\tau_1 \to \tau_1'$, …, $\tau_n \to \tau_n'$, and the final answer type is τ_f. For a pure τ to be a subtype of this type, the derivation of the subtyping needs to look like this:

$$\dfrac{\tau \le \tau_1 \qquad \dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{\tau'_n \le \tau_f \qquad \dfrac{}{\varepsilon \le \varepsilon}\,\text{S-EPS}}{\tau'_n \le \tau_f}\,\text{S-STR}}{\varepsilon \le [\tau'_n]\,\tau_f}\,\text{S-PURE}}{\tau'_{n-1} \le \tau_n \quad \varepsilon \le [\tau'_n]\,\tau_f}\,\text{S-STR}}{\tau'_{n-1} \le \tau_n\,[\tau'_n]\,\tau_f}\ \dots}{\tau'_1 \le \tau_2\,[\tau'_2]\,\tau_3\ \dots \tau_n\,[\tau'_n]\,\tau_f \quad \varepsilon \le [\tau'_1]\,\tau_2\ \dots \tau_n\,[\tau'_n]\,\tau_f}\,\text{S-PURE}}{\tau \le \tau_1\,[\tau'_1]\,\tau_2\ \dots \tau_n\,[\tau'_n]\,\tau_f}\,\text{S-STR}$$

So, it is required that $\tau \le \tau_1$, $\tau'_k \le \tau_{k+1}$ for all k, and $\tau'_n \le \tau_f$; in other words, that a value of type τ can be passed to the first context, the contexts can be composed together, and the result of the final context can be used as the final answer.

One can also define the subtyping relations by first defining the successor relations $\prec$ for types, effect annotations and annotated types as the smallest congruences containing $\varepsilon \prec [\tau\,\sigma]\,\tau\,\sigma$ for every τ and σ.

Theorem 3 *The reflexive and transitive closure of the successor relation $\prec$ is the subtyping relation $\le$.*

The type system has two distinct rules for function application: one for pure expressions (i.e., those without control effects), and one for impure expressions. The rule for pure application seems not strictly necessary, but its removal reduces the expressiveness of the type system—for example, the following term

$$(\lambda f.\lambda y.(\lambda z.\langle(\lambda v.\lambda w.y)\,(f\,y)\rangle)\,\langle f\,y\rangle)\,(\lambda x.(\lambda x.x)\,x)$$

can no longer be typed without this rule. The reason is that the function application inside the subterm $\lambda x.(\lambda x.x)\,x$ would then force it into "impure" typing, which makes the typing fail in the left part of the term because of answer type incompatibility. Some other terms would still be typeable, but the set of possible types would be smaller. For example, the program presented in the beginning of this section can be typed without the pure application rule, but with a different, more complicated type: $\alpha \xrightarrow{[\alpha]\ \alpha\ [\beta]\ \beta} \alpha$ instead of $\alpha \xrightarrow{[\alpha]\ \alpha} \alpha$.

The type system satisfies the expected key properties such as subject reduction, progress and unique decomposition, which is demonstrated next.

Theorem 4 (Subject reduction) *If $\vdash e : \tau\,\sigma$ and $e \to e'$ for some e', then $\vdash e' : \tau\,\sigma$.*

Proof We prove the theorem for $\rightsquigarrow$ and then the thesis for $\to$ follows by trivial induction. Using induction on the derivation of the typing judgment, we can assume that it does not have the subtyping rule SUB at the root and we proceed by case analysis. The proofs of the cases require using the subtyping relation in a nontrivial way, and are an instructional demonstration on how subtyping works in the language.

We have several cases to consider:

- $\langle v\rangle \rightsquigarrow v$. Then for some type τ' we have $\vdash v : \tau'\,[\tau']\,\tau\,\sigma$. Because v is a value, we have $\vdash v : \tau'$ and $\varepsilon \le [\tau']\,\tau\,\sigma$, which implies $\tau' \le \tau\,\sigma$. Thus we get $\vdash v : \tau\,\sigma$.
- $(\lambda x.e)\,v \rightsquigarrow e\{v/x\}$ and $\sigma = \varepsilon$. Suppose that $\vdash v : \tau'$. We have $x : \tau'' \vdash e : \tau'''$, $\tau' \le \tau''$ and $\tau''' \le \tau$. We get $\vdash v : \tau''$, and $\vdash e\{v/x\} : \tau$ follows from a standard substitution lemma (that we omit).

– $(\lambda x.e)\, v \rightsquigarrow e\{v/x\}$ and $\sigma = [\tau_4\ \sigma_4]\ \tau_1\ \sigma_1$. Suppose that $\vdash v : \tau'\ [\tau_3\ \sigma_3]\ \tau_2\ \sigma_2$ and $\vdash \lambda x.e : \tau' \xrightarrow{[\tau_4\ \sigma_4]\ \tau_3\ \sigma_3} \tau\ [\tau_2\ \sigma_2]\ \tau_1\ \sigma_1$. Because v is a value, we have $\vdash v : \tau'$ and $\tau_3\ \sigma_3 \leq \tau_2\ \sigma_2$. Because $\lambda x.e$ is a value, we have $\vdash \lambda x.e : \tau' \xrightarrow{[\tau_4\ \sigma_4]\ \tau_3\ \sigma_3} \tau$ and $\tau_2\, \sigma_2 \leq \tau_1\, \sigma_1$; by transitivity, $\tau_3\, \sigma_3 \leq \tau_1\, \sigma_1$ follows. We also have $x : \tau'' \vdash e : \tau'''\ \sigma'$, $\tau' \leq \tau''$, $\tau'''\ \sigma' \leq \tau\ [\tau_4\ \sigma_4]\ \tau_3\ \sigma_3$. By transitivity we get $\tau'''\ \sigma' \leq \tau\ [\tau_4\ \sigma_4]\ \tau_1\ \sigma_1$. We get $\vdash v : \tau''$, and $\vdash e\{v/x\} : \tau\ [\tau_4\ \sigma_4]\ \tau_1\ \sigma_1$ follows from the substitution lemma.
– $\langle K[\mathcal{S}_0 f.e]\rangle \rightsquigarrow e\{\lambda x.\langle K[x]\rangle/f\}$. Then for some type τ' we have $\vdash K[\mathcal{S}_0 f.e] : \tau'[\tau']\tau\sigma$. So (by induction on K) we have $\vdash \mathcal{S}_0 f.e : \tau''\,[\tau'''\,\sigma''']\,\tau\,\sigma$ (and thus $f : \tau'' \xrightarrow{\sigma'''} \tau''' \vdash e : \tau\ \sigma$) and $x : \tau'' \vdash K[x] : \tau'\ [\tau']\ \tau'''\ \sigma'''$. Then $\vdash \lambda x.\langle K[x]\rangle : \tau'' \xrightarrow{\sigma'''} \tau'''$, so using the substitution lemma we get $\vdash e\{\lambda x.\langle K[x]\rangle/f\} : \tau\ \sigma$.

Theorem 5 (Progress) *If $\vdash e : \tau\ \sigma$, then e is a value, $e \to e'$ for some e', or $e = K[\mathcal{S}_0 f.e]$. If $\sigma = \varepsilon$, the third case cannot occur.*

3.2 Typing contexts and trails

Even though it is not strictly necessary, it is useful to have separate typing rules for contexts and trails as a proof device and for improving the understanding of the system.

We first have to introduce a different way of looking at the types of expressions. In the type system shown in Figure 5, we put emphasis on the argument type of the first context on the trail (or of the bottom context, if the trail is empty). In the current, alternative approach, the final answer type will be emphasized.

Let us introduce the syntactic category of trail types (which are just sequences of function types):

$$\pi ::= (\tau \xrightarrow{\sigma} \tau)^*$$

We use ι to denote the empty trail type, and we use juxtaposition for trail type concatenation.

We define three functions: $\overrightarrow{\tau\ \sigma}$, which extracts the trail type, $\downarrow(\tau\ \sigma)$ which extracts the answer type, and the composition $\pi\ (\tau\ \sigma)$, which appends a trail type to an effect-annotated type. These functions are defined as follows:

$$\begin{aligned}
\overrightarrow{\tau} &= \iota \\
\overrightarrow{\tau_1'\ [\tau_1\ \sigma_1]\ \tau\ \sigma} &= (\tau_1' \xrightarrow{\sigma_1} \tau_1)\ \overrightarrow{\tau\sigma} \\
\downarrow\tau &= \tau \\
\downarrow(\tau_1'\ [\tau_1\ \sigma_1]\ \tau\ \sigma) &= \downarrow(\tau\ \sigma) \\
\iota\ (\tau\ \sigma) &= \tau\ \sigma \\
\pi\ (\tau_1' \xrightarrow{\sigma_1} \tau_1)\ (\tau\ \sigma) &= \pi\ (\tau_1'\ [\tau_1\ \sigma_1]\ \tau\ \sigma)
\end{aligned}$$

We will omit the parentheses when there is no confusion about the meaning of the given expression.

Property 1 For any τ and σ, we have $\tau\ \sigma = \overrightarrow{\tau\ \sigma}\downarrow\tau\ \sigma$.

Property 2 For any π, π', τ and σ we have $(\pi'\ \pi)\ \tau\ \sigma = \pi'\ (\pi\ \tau\ \sigma)$.

Contexts

$$\frac{}{\Gamma \vdash \bullet : \tau \to \tau}\ \text{EMPTY} \qquad \frac{\Gamma \vdash K : \tau \xrightarrow{\sigma_1} \tau_1 \qquad \Gamma \vdash e : \tau' \ [\tau_2\ \sigma_2]\ \tau_3\ \sigma_3}{\Gamma \vdash Ke : (\tau' \xrightarrow{[\tau_1\ \sigma_1]\ \tau_2\ \sigma_2} \tau) \xrightarrow{\sigma_3} \tau_3}\ \text{APPL}$$

$$\frac{\Gamma \vdash K : \tau \xrightarrow{\sigma_1} \tau_1 \qquad \Gamma \vdash v : \tau' \xrightarrow{[\tau_1\ \sigma_1]\ \tau_2\ \sigma_2} \tau}{\Gamma \vdash vK : \tau' \xrightarrow{\sigma_2} \tau_2}\ \text{APPR}$$

$$\frac{\Gamma \vdash K : \tau_1 \xrightarrow{\sigma} \tau_2 \qquad \tau_1' \leq \tau_1 \qquad \tau_2\ \sigma \leq \tau_2'\ \sigma'}{\Gamma \vdash K : \tau_1' \xrightarrow{\sigma'} \tau_2'}\ \text{SUB}$$

Pure contexts

$$\frac{}{\Gamma \vdash_P \bullet : \tau \to \tau}\ \text{EMPTY} \qquad \frac{\Gamma \vdash_P K : \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_3}{\Gamma \vdash_P Ke : (\tau_3 \to \tau_1) \to \tau_2}\ \text{APPL}$$

$$\frac{\Gamma \vdash_P K : \tau_1 \to \tau_2 \qquad \Gamma \vdash v : \tau_3 \to \tau_1}{\Gamma \vdash_P vK : \tau_3 \to \tau_2}\ \text{APPR} \qquad \frac{\Gamma \vdash_P K : \tau_1 \to \tau_2 \qquad \tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\Gamma \vdash_P K : \tau_1' \to \tau_2'}\ \text{SUB}$$

Trails

$$\frac{}{\Gamma \vdash \square : \iota}\ \text{EMPTY} \qquad \frac{\Gamma \vdash K : \tau' \xrightarrow{\sigma} \tau \qquad \Gamma \vdash M : \pi}{\Gamma \vdash K \cdot M : (\tau' \xrightarrow{\sigma} \tau)\ \pi}\ \text{CONS}$$

Subtyping for trail types

$$\frac{}{\iota \leq \iota}\ \text{T-NIL} \qquad \frac{\tau_1 \xrightarrow{\sigma} \tau_2 \leq \tau_1' \xrightarrow{\sigma'} \tau_2' \qquad \pi \leq \pi'}{(\tau_1 \xrightarrow{\sigma} \tau_2)\ \pi \leq (\tau_1' \xrightarrow{\sigma'} \tau_2')\ \pi'}\ \text{T-CONS}$$

Metacontexts, programs

$$\frac{\Gamma \vdash T_M : \pi \qquad \Gamma \vdash_P K : \tau'' \to \tau' \qquad \tau \leq \pi\ \tau''}{\Gamma \vdash \langle T_M, K \rangle : \tau \to \tau'}\ \text{META}$$

$$\frac{\Gamma \vdash e : \tau\ \sigma \qquad \Gamma \vdash T : \overrightarrow{\tau\ \sigma} \qquad \Gamma \vdash_P K : \downarrow\tau\ \sigma \to \tau'}{\Gamma \vdash \langle e, T, K \rangle : \tau'}\ \text{PROG}$$

Fig. 6 Typing rules for contexts and trails in $\lambda^{\mathcal{S}_0}_{\leq}$

The typing rules for contexts and trails are shown in Figure 6. Contexts are assigned function types, and the types of trails are just lists of context types. We can relate these typings to the typing relation for terms:

Lemma 3 (Typing contexts) *$\Gamma \vdash K : \tau' \xrightarrow{\sigma} \tau$ if and only if $\Gamma \vdash \lambda x.\langle K[x] \rangle : \tau' \xrightarrow{\sigma} \tau$.*

Lemma 4 (Typing trails) *$\Gamma \vdash T : \pi$ is derivable if and only if for every e, τ, σ such that $\Gamma \vdash e : \pi\ \tau\ \sigma$ one can derive $\Gamma \vdash T[e] : \tau\ \sigma$.*

Additionally, we introduce the notion of pure contexts, which are not allowed to have occurrences of the shift_0 operator not surrounded by a reset_0. For this typing relation the following lemma holds:

Lemma 5 (Typing pure contexts) *$\Gamma \vdash_P K : \tau' \to \tau$ if and only if $\Gamma \vdash \lambda x.K[x] : \tau' \to \tau$.*

Pure typing of contexts is a restricted version of the impure typing, in the sense that even when considering only contexts with a pure type (i.e., of the form $\tau \to \tau'$), there are strictly fewer pure contexts than impure ones. For example, take the context $K = \bullet\,(\mathcal{S}_0 f.y)$ in the type environment $\Gamma = y : \alpha$. We have

$$y : \alpha \vdash \lambda x.\langle x\,(\mathcal{S}_0 f.y)\rangle : (\alpha \xrightarrow{[\alpha]\ \alpha} \alpha) \to \alpha$$

so from Lemma 3 follows that $y : \alpha \vdash K : (\alpha \xrightarrow{[\alpha]\ \alpha} \alpha) \to \alpha$. But any type of the term $\lambda x.x\,(\mathcal{S}_0 f.y)$ ($\lambda x.K[x]$ from Lemma 5) must have the function arrow annotated with an impure annotation, therefore $y : \alpha \not\vdash_P K : (\alpha \xrightarrow{[\alpha]\ \alpha} \alpha) \to \alpha$.

However, every pure context is an impure one:

Lemma 6 (Impure typing of pure contexts) *If $\Gamma \vdash_P K : \tau' \to \tau$, then $\Gamma \vdash K : \tau' \to \tau$.*

We can now introduce the typing relation for programs. As witnessed by the unique-decomposition theorem, we can view a program as composed of three parts: an expression, a trail, and a pure context at the bottom. The following theorem establishes the correctness of the rule PROG:

Lemma 7 (Typing programs) *$\Gamma \vdash \langle e, T, K\rangle : \tau$ if and only if $\Gamma \vdash K[T[e]] : \tau$.*

Next, we formally introduce the notion of metacontexts, which will play a role in proving termination. A metacontext in our system is a pair of a trail T_M, typed $\Gamma \vdash T_M : \pi$ for some π, and a pure context K, typed $\Gamma \vdash_P K : \tau'' \to \tau'$ for some τ' and τ'', and where for some τ we have $\tau \le \pi\,\tau''$. The intuitive meaning of this restriction is that the contexts in the trail cannot access contexts outside of it and they can be composed, so that when given a value of type τ, they produce a value of type τ'', which can be put into the pure context at the bottom.

Lemma 8 (Typing metacontexts) *$\Gamma \vdash \langle T_M, K\rangle : \tau \to \tau'$ if and only if $\Gamma, x : \tau \vdash K[T_M[x]] : \tau'$.*

Finally, we introduce a different notion of program decomposition. In a well-typed program, the trail can be split into two parts: the top part, which is accessible by the expression inside, and the bottom part, which is guaranteed by the type of the expression to be inaccessible. This bottom part of the trail, together with the pure context at the very bottom, form a metacontext. This is made formal by the following theorem:

Theorem 6 *If $\Gamma \vdash e : \tau\ \sigma$ and $\Gamma \vdash \langle e, T, K\rangle : \tau'$, then there exist T_1 and T_M such that $T = T_1 \cdot T_M$, $\Gamma \vdash T_1 : \overrightarrow{\tau\,\sigma}$, and $\Gamma \vdash \langle T_M, K\rangle : {\downarrow}\tau\ \sigma \to \tau'$.*

3.3 Termination

We now prove termination of evaluation of closed well-typed expressions using a variant of the method described in [5,6]. We believe that it is instructive to see the proof in full detail. First, the proof reveals otherwise hidden nuances of how the reduction semantics and the type system interact. Second, it contains explicit information on how well-typed terms in our language are evaluated: each case of the proof of Lemma 13 corresponds exactly to a clause of an evaluator which can be extracted from this proof [5,6].

Let us start by defining three families of mutually inductive predicates. The families are defined by induction on types: $\mathcal{R}_\tau$ is defined on well-typed values and is indexed by their types, $\mathcal{T}_\pi$ is defined on well-typed trails and is indexed by trail types, and finally $\mathcal{M}_\tau$ is defined on metacontexts and is indexed by their argument types. We also define a predicate $\mathcal{N}(e, T, K)$ defined on well-typed programs, which means that the evaluation of this program terminates, i.e., that $K[T[e]] \to^* v$ for some value v.

$$\begin{aligned}
\mathcal{R}_\alpha(v) &:= \top \\
\mathcal{R}_{\tau' \xrightarrow{\sigma} \tau}(v) &:= \forall v', T, T_M, K. \\
&\qquad \mathcal{R}_{\tau'}(v') \Rightarrow \\
&\qquad \mathcal{T}_{\overrightarrow{\tau\,\sigma}}(T) \Rightarrow \\
&\qquad \mathcal{M}_{\downarrow \tau\ \sigma}(T_M, K) \Rightarrow \\
&\qquad \mathcal{N}(v\, v', T \cdot T_M, K) \\
\mathcal{T}_\iota(\Box) &:= \top \\
\mathcal{T}_{(\tau' \xrightarrow{\sigma} \tau)\ \pi}(K \cdot T) &:= \mathcal{R}_{\tau' \xrightarrow{\sigma} \tau}(\lambda x.\langle K[x] \rangle) \wedge \mathcal{T}_\pi(T) \\
\mathcal{M}_\tau(T_M, K) &:= \forall v. \mathcal{R}_\tau(v) \Rightarrow \mathcal{N}(v, T_M, K)
\end{aligned}$$

For the proof, we need several lemmas about the subtyping relation and its connection to the typing rules and the above predicates.

Lemma 9 *If $\tau \le \pi\ \tau_1$, $\Gamma \vdash T : \pi$, $\Gamma \vdash \langle T_M, K \rangle : \tau_1 \to \tau_2$, then $\Gamma \vdash \langle T \cdot T_M, K \rangle : \tau \to \tau_2$.*

Proof From $\Gamma \vdash \langle T_M, K \rangle : \tau_1 \to \tau_2$ for some τ', π' we have $\Gamma \vdash T_M : \pi'$, $\Gamma \vdash_P K : \tau' \to \tau_2$ and $\tau_1 \le \pi'\ \tau'$. So $\Gamma \vdash T \cdot T_M : \pi\ \pi'$. From $\tau_1 \le \pi'\ \tau'$ we have $\pi\ \tau_1 \le \pi\ \pi'\ \tau'$, by transitivity from the assumption $\tau \le \pi\ \tau_1$ we have $\tau \le \pi\ \pi'\ \tau'$.

Lemma 10 *1. The predicates respect the subtyping relations:*
- *R. If $\tau \le \tau'$ and $\mathcal{R}_\tau(v)$, then $\mathcal{R}_{\tau'}(v)$.*
- *T. If $\pi \le \pi'$ and $\mathcal{T}_\pi(T)$, then $\mathcal{T}_{\pi'}(T)$.*
- *M. If $\tau' \le \tau$ and $\mathcal{M}_\tau(T_M, K)$, then $\mathcal{M}_{\tau'}(T_M, K)$.*

2. If $\tau'\ \sigma' \le \tau\ \sigma$, $\mathcal{T}_{\overrightarrow{\tau\,\sigma}}(T)$ and $\mathcal{M}_{\downarrow \tau\ \sigma}(T_M, K)$, then for some T', T'_M we have $\mathcal{T}_{\overrightarrow{\tau'\,\sigma'}}(T')$, $\mathcal{M}_{\downarrow \tau'\ \sigma'}(T'_M, K)$ and $T \cdot T_M = T' \cdot T'_M$.

Proof Proof by simultaneous induction. The proofs refer to other cases of the lemma with smaller subtyping derivations. The only exception is a reference from (1M) to (1R), but since it does not introduce a cycle, the induction is justified.

1R. $\tau = \tau_1 \xrightarrow{\sigma} \tau_2$, $\tau' = \tau'_1 \xrightarrow{\sigma'} \tau'_2$. We have $\tau'_1 \le \tau_1$ and $\tau_2\ \sigma \le \tau'_2\ \sigma'$. Let v, T, T_M, K be such that $\mathcal{R}_{\tau'_1}(v)$, $\mathcal{T}_{\overrightarrow{\tau'_2\,\sigma'}}(T)$, $\mathcal{M}_{\downarrow \tau'_2\ \sigma'}(T_M, K)$. By induction hypothesis for case (2) we have T' and T'_M such that $\mathcal{T}_{\overrightarrow{\tau_2\,\sigma}}(T')$, $\mathcal{M}_{\downarrow \tau_2\ \sigma}(T'_M, K)$ and $T \cdot T_M = T' \cdot T'_M$. Thus from the assumption $\mathcal{R}_\tau(v)$ follows the thesis.

1T. The case $\pi = \pi' = \iota$ is trivial. Let $\pi = \tau_1 \xrightarrow{\sigma} \tau_2\ \pi_1$ and $\pi' = \tau'_1 \xrightarrow{\sigma'} \tau'_2\ \pi'_1$. Then we have $\tau_1 \xrightarrow{\sigma} \tau_2 \le \tau'_1 \xrightarrow{\sigma'} \tau'_2$, $\pi_1 \le \pi'_1$, $T = K \cdot T_1$, $\mathcal{R}_{\tau_1 \xrightarrow{\sigma} \tau_2}(\lambda x.\langle K[x] \rangle)$ and $\mathcal{T}_{\pi_1}(T_1)$. By induction hypothesis for case (1R) and (1T) we get $\mathcal{R}_{\tau'_1 \xrightarrow{\sigma'} \tau'_2}(\lambda x.\langle K[x] \rangle)$ and $\mathcal{T}_{\pi'_1}(T_1)$, which give us the thesis.

1M. Let v be such that $\mathcal{R}_{\tau'}(v)$. By case (1R) we have $\mathcal{R}_\tau(v)$, with $\mathcal{M}_\tau(T_M, K)$ this gives us $\mathcal{N}(v, T_M, K)$.

2. Induction on the subtyping derivation.

- $\sigma' = \varepsilon$, $\sigma = \varepsilon$. Then $T = \Box$, $\tau' \leq \tau$, $\mathcal{M}_\tau(T_M, K)$. Let $T' = \Box$, $T'_M = T_M$. We thus need to show that $\mathcal{M}_{\tau'}(T_M, K)$, but that follows from case (1M).
- $\sigma' = \varepsilon$, $\sigma = [\tau_1\,\sigma_1]\,\tau_2\,\sigma_2$. Then $\tau' \leq \tau$, $\tau_1\,\sigma_1 \leq \tau_2\sigma_2$, $T = K_1 \cdot T'$, $\mathcal{T}_\tau \xrightarrow{\sigma_1} \tau_1(K_1)$, $\mathcal{T}_{\overrightarrow{\tau_2\,\sigma_2}}(T')$, $\mathcal{M}_{\downarrow\tau_2\,\sigma_2}(T_M, K)$. Let $T' = \Box$, $T'_M = T \cdot T_M$. Trivially we have $\mathcal{T}_\iota(\Box)$, we need to show that $\mathcal{M}_{\tau'}(T \cdot T_M, K)$. The metacontext $\langle T \cdot T_M, K\rangle$ is well-typed by Lemma 9. Let v be such that $\mathcal{R}_{\tau'}(v)$, I will show that $\mathcal{N}(v, T \cdot T_M, K)$. From case (1T) we have $\mathcal{T}_{\tau'} \xrightarrow{\sigma_2} \tau_2(K_1)$, by definition of $\mathcal{T}_{\tau'} \xrightarrow{\sigma_2} \tau_2$ we have $\mathcal{R}_{\tau'\ \xrightarrow{\sigma_2}\ \tau_2}(\lambda x.\langle K_1[x]\rangle)$. With $\mathcal{R}_{\tau'}(v)$, $\mathcal{T}_{\overrightarrow{\tau_2\,\sigma_2}}(T')$ and $\mathcal{M}_{\downarrow\tau_2\,\sigma_2}(T_M, K)$ we have $\mathcal{N}((\lambda x.\langle K_1[x]\rangle)\,v, T' \cdot T_M, K)$. Because $(\lambda x.\langle K_1[x]\rangle)\,v$ reduces in single step to $\langle K_1[v]\rangle$, we have $\mathcal{N}(\langle K_1[v]\rangle, T' \cdot T_M, K)$, which is equivalent to $\mathcal{N}(v, K_1 \cdot T' \cdot T_M, K)$, which we wanted to prove.
- $\sigma' = [\tau'_1\,\sigma'_1]\,\tau'_2\,\sigma'_2$, $\sigma = [\tau_1\,\sigma_1]\,\tau_2\,\sigma_2$. Then $T = K_1 \cdot T_1$, $\mathcal{T}_\tau \xrightarrow{\sigma_1} \tau_1(K_1)$, $\mathcal{T}_{\overrightarrow{\tau_2\,\sigma_2}}(T_1)$, $\mathcal{M}_{\downarrow\tau_2\,\sigma_2}(T_M, K)$, $\tau' \leq \tau$ and $\tau'_2\,\sigma'_2 \leq \tau_2\,\sigma_2$. By induction hypothesis we have T'_1, T'_M such that $\mathcal{T}_{\overrightarrow{\tau'_2\,\sigma'_2}}(T'_1)$, $\mathcal{M}_{\downarrow\tau'_2\,\sigma'_2}(T'_M, K)$ and $T_1 \cdot T_M = T'_1 \cdot T'_M$. From $\mathcal{M}_{\downarrow\tau'_2\,\sigma'_2}(T'_M, K)$ we have $\mathcal{M}_{\downarrow\tau'\,\sigma'}(T'_M, , K)$. Let $T' = K_1 \cdot T'_1$. From $\tau_1\,\sigma_1 \leq \tau'_1\,\sigma'_1$, $\tau' \leq \tau$, $\mathcal{T}_\tau \xrightarrow{\sigma_1} \tau_1(K_1)$ and the induction hypothesis for case (1T) we have $\mathcal{T}_{\tau'} \xrightarrow{\sigma'_1} \tau'_1(K_1)$. With $\mathcal{T}_{\overrightarrow{\tau'_2\,\sigma'_2}}(T'_1)$ we have $\mathcal{T}_{\overrightarrow{\tau'\,\sigma'}}(K_1 \cdot T'_1)$.

Lemma 11 *For every type τ, $\mathcal{T}_{\tau \to \tau}(\bullet \cdot \Box)$ holds.*

Proof By the definition of $\mathcal{T}_{\tau \to \tau}$ it is sufficient to prove that $\mathcal{R}_{\tau \to \tau}(\lambda x.\langle x\rangle)$. Let v, T_M, and K be such that $\mathcal{R}_\tau(v)$ and $\mathcal{M}_\tau(T_M, K)$ hold. We have to show that $\mathcal{N}((\lambda x.\langle x\rangle)\,v, T_M, K)$. The expression $(\lambda x.\langle x\rangle)\,v$ reduces in two steps to v, so it suffices to prove $\mathcal{N}(v, T_M, K)$, which follows from $\mathcal{R}_\tau(v)$.

Lemma 12 *For every type τ, $\mathcal{M}_\tau(\Box, \bullet)$ holds.*

Proof Let v be a value of type τ, then $\mathcal{N}(v, \Box, \bullet)$ is trivially true.

We now state the main lemma:

Lemma 13 *Let $\Gamma \vdash e : \tau\ \sigma$, where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. Assume that $\vec{v}$ is a sequence of values of length n such that for every i, $\mathcal{R}_{\tau_i}(v_i)$ holds. Then for every trail T such that $\mathcal{T}_{\overrightarrow{\tau\,\sigma}}(T)$ holds, and for every metacontext $\langle T_M, K\rangle$ such that $\mathcal{M}_{\downarrow\tau\,\sigma}(T_M, K)$ holds, $\mathcal{N}(e\{\vec{v}/\vec{x}\}, T \cdot T_M, K)$ holds.*

Proof Induction on the structure of the typing derivation D of e.

- $D = \textsc{sub}(D', \tau'\sigma' \leq \tau\sigma)$. From Lemma 10 (2) we have T', T'_M such that $\mathcal{T}_{\overrightarrow{\tau'\,\sigma'}}(T')$, $\mathcal{M}_{\downarrow\tau'\,\sigma'}(T'_M, K)$ and $T \cdot T_M = T' \cdot T'_M$. The conclusion follows from the induction hypothesis.
- $D = \textsc{var}$, $e = x_i$. So we have $e\{\vec{v}/\vec{x}\} = v_i$, $\tau = \tau_i$, $\sigma = \varepsilon$,$T = \Box$, $\mathcal{M}_{\tau_i}(T_M, K)$. By assumption $\mathcal{R}_{\tau_i}(v_i)$ and the definition of $\mathcal{M}_{\tau_i}$ we have $\mathcal{N}(v_i, T_M, K)$.
- $D = \textsc{abs}(D')$, $e = \lambda x.e'$. So $\tau = \tau'' \to \sigma'\tau'$, $\sigma = \varepsilon$, $T = \Box$, $\mathcal{M}_\tau(T_M, K)$. As in the previous case, to show that $\mathcal{N}((\lambda x.e')\{\vec{v}/\vec{x}\}, T_M, K)$ we need $\mathcal{R}_\tau(\lambda x.e'\{\vec{v}/\vec{x}\})$.
  Let v, T, T_M, K be such that we have $\mathcal{R}_{\tau''}(v)$, $\mathcal{T}_{\overrightarrow{\tau'\,\sigma'}}(T)$ and $\mathcal{M}_{\downarrow\tau'\,\sigma'}(T_M, K)$. We need to show that $\mathcal{N}((\lambda x.e'\{\vec{v}/\vec{x}\})\,v, T \cdot T_M, K)$. Because $(\lambda x.e'\{\vec{v}/\vec{x}\})\,v$ reduces in one step to $e'\{\vec{v}/\vec{x}\}\{v/x\}$, it suffices to show $\mathcal{N}(e'\{\vec{v}/\vec{x}\}\{v/x\}, T \cdot T_M, K)$, which follows from the induction hypothesis.

- $D = \textsc{app}(D_1, D_2)$, $e = e_1\, e_2$. Then $\sigma = [\tau_A\, \sigma_A]\, \tau_D\, \sigma_D$, $T = K_1 \cdot T'$, $\mathcal{T}_\tau \xrightarrow{\sigma_A} \tau_A(K_1)$, $\mathcal{T}_{\overrightarrow{\tau_B\, \sigma_B}}(T')$, $\Gamma \vdash e_1 : \tau' \xrightarrow{[\tau_A\, \sigma_A]\, \tau_B\, \sigma_B} \tau\, [\tau_C\, \sigma_C]\, \tau_D\, \sigma_D$, $\Gamma \vdash e_2 : \tau'\, [\tau_B\, \sigma_B]\, \tau_C\, \sigma_C$. Let $e_1' = e_1\{\vec{v}/\vec{x}\}$ and $e_2' = e_2\{\vec{v}/\vec{x}\}$. To show that $\mathcal{N}(e_1'\, e_2', K_1 \cdot T' \cdot T_M, K)$, we can show instead that $\mathcal{N}(e_1', K_1\, e_2' \cdot T' \cdot T_M, K)$ using the induction hypothesis for D_1, providing that $\mathcal{T}_{(\tau' \xrightarrow{[\tau_A\, \sigma_A]\, \tau_B\, \sigma_B} \tau)} \xrightarrow{\sigma_C} \tau_C(K_1\, e_2')$. To show that, let us unfold the definitions. Let v_L, T_L, T_{KL} and M_L satisfy the appropriate predicates, i.e., $\mathcal{R}_{\tau'} \xrightarrow{[\tau_A\, \sigma_A]\, \tau_B\, \sigma_B} \tau(v_L)$, $\mathcal{T}_{\overrightarrow{\tau_C\, \sigma_C}}(T_L)$ and $\mathcal{M}_{\downarrow \tau_C\, \sigma_C}(T_{KL}, M_L)$. Then we have to show that $\mathcal{N}((\lambda x.\langle K_1[x\, e_2]\rangle)\, v_L, T_L \cdot T_{KL}, M_L)$. Because $(\lambda x.\langle K_1[x\, e_2]\rangle)\, v_L$ reduces in one step to $\langle K_1[v_L\, e_2]\rangle$, we can show instead that $\mathcal{N}(\langle K_1[v_L\, e_2]\rangle, T_L \cdot T_{KL}, M_L)$. We will show instead that $\mathcal{N}(e_2, v_L\, K_1 \cdot T_L \cdot T_{KL}, M_L)$. This follows for the induction hypothesis for D_2, providing that $\mathcal{T}_{\tau'} \xrightarrow{\sigma_B} \tau_B(v_L\, K_1)$. To show that, let us unfold the definitions again. Let v_R, T_R, T_{KR} and M_R be such that $\mathcal{R}_{\tau'}(v_R)$, $\mathcal{T}\overrightarrow{{}_{\tau_B\, \sigma_B}}(T_R)$ and $\mathcal{M}_{\downarrow \tau_B\, \sigma_B}(T_{KR}, M_R)$ hold. We then have to show that $\mathcal{N}((\lambda x.\langle K_1[v_L\, x]\rangle)\, v_R, T_R \cdot T_{KR}, M_R)$ hold. Using analogous reasoning as before, it is sufficient to show that $\mathcal{N}(v_L\, v_R, K_1 \cdot T_R \cdot T_{KR}, M_R)$, which follows from $\mathcal{R}_{\tau'} \xrightarrow{[\tau_A\, \sigma_A]\, \tau_B\, \sigma_B} \tau(v_L)$.
- $D = \textsc{app-pure}(D_1, D_2)$, $e = e_1\, e_2$. The proof is similar to the case for app.
- $D = \textsc{shift0}(D')$, $e = \mathcal{S}_0 f.e'$. So, in this case $\sigma = [\tau'\, \sigma']\, \tau''\, \sigma''$, $T = K_1 \cdot T'$, $\mathcal{R}_\tau \xrightarrow{\sigma'} \tau'(\lambda x.\langle K_1[x]\rangle)$, $\mathcal{T}_{\overrightarrow{\tau''\sigma''}}(T')$ and $\mathcal{M}_{\downarrow \tau''\sigma''}(T_M, K)$. We want to show that $\mathcal{N}(\mathcal{S}_0 f.e\{\vec{v}/\vec{x}\}, K_1 \cdot T' \cdot T_M, K)$, which is equivalent to $\mathcal{N}(\langle K_1[\mathcal{S}_0 f.e\{\vec{v}/\vec{x}\}]\rangle, T' \cdot T_M, K)$. Because $\langle K_1[\mathcal{S}_0 f.e\{\vec{v}/\vec{x}\}]\rangle$ reduces to $e\{\vec{v}/\vec{x}\}\{\lambda x.\langle K[x]\rangle/f\}$, it suffices to show that $\mathcal{N}(e\{\vec{v}/\vec{x}\}\{\lambda x.\langle K[x]\rangle/f\}, T' \cdot T_M, K)$, which follows from the induction hypothesis.
- $D = \textsc{reset0}(D')$, $e = \langle e'\rangle$. Then for some τ' we have $\Gamma \vdash e' : \tau'\, [\tau']\, \tau\, \sigma$. From Lemma 11 have $\mathcal{T}_{\tau'} \to \tau'(\bullet \cdot \Box)$, and thus $\mathcal{T}_{\overrightarrow{\tau'\, [\tau']\, \tau\, \sigma}}(\bullet \cdot T)$. The induction hypothesis gives us $\mathcal{N}(e'\{\vec{v}/\vec{x}\}, \bullet \cdot T \cdot T_M, K)$, which is equivalent to the thesis, $\mathcal{N}(\langle e'\rangle\{\vec{v}/\vec{x}\}, \cdot T \cdot T_M, K)$.

Theorem 7 (Termination) *If $\vdash e : \tau$, then the evaluation of e terminates, i.e., $\mathcal{N}(e, \Box, \bullet)$ holds.*

Proof We have $\mathcal{T}_\iota(\Box)$ and from Lemma 12 we have $\mathcal{M}_\tau(\Box, \bullet)$, thus the theorem follows from Lemma 13.

3.4 Type inference

The type inference algorithm should reconstruct types for a given expression and find its principal type. So let us direct our attention first to the principal types. It would be nice if the principal type for any expression e was an annotated type $\tau\, \sigma$ such that $\Gamma \vdash e : \tau\sigma$ and every other type could be obtained via substitution and subtyping—but that is not the case. Let us take a look at the expression $\lambda f.\lambda x.f\, x; f\, x$ (the *semicolon* operator can be defined as $e_1; e_2 = (\lambda x.e_2)\, e_1$, where $x \notin e_2$). It has three distinct typings:

- $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$ (pure argument, pure result)
- $(\tau_1 \to \tau_2) \to \tau_1 \xrightarrow{[\tau\, \sigma]\, \tau\, \sigma} \tau_2$ (pure argument, impure result)
- $(\tau_1 \xrightarrow{[\tau\, \sigma]\, \tau\, \sigma} \tau_2) \to \tau_1 \xrightarrow{[\tau\, \sigma]\, \tau\, \sigma} \tau_2$ (impure argument, impure result)

There is no type which would be the supertype of the three types above. But notice that these types are all of the form $(\tau_1 \xrightarrow{\sigma_1} \tau_2) \to \tau_1 \xrightarrow{\sigma_2} \tau_2$, where $\sigma_1 \leq \sigma_2$ and σ_2 is

either ε, or is equal to $[\tau\sigma]\tau\sigma$ for some τ and σ. So the notion of principal type (and the type inference algorithm) for $\lambda^{\mathcal{S}_0}$ has to include the possibility of these constraints appearing. This observation agrees with the classical results for the simply typed lambda calculus with subtyping [30].

First, we replace the rules APP and APP-PURE with the following (equivalent) rule:

$$\frac{\Gamma \vdash f : \tau' \xrightarrow{\sigma_3} \tau\,\sigma_1 \qquad \Gamma \vdash e : \tau'\,\sigma_2 \qquad \sigma \ll \sigma_1\,\sigma_2\,\sigma_3}{\Gamma \vdash fe : \tau\,\sigma}\ \text{APP-COMP},$$

where the relation $\ll$, which relates an annotation to a sequence of annotations, is defined as follows:

$$\frac{}{\varepsilon \ll} \qquad\qquad \frac{}{[\tau\,\sigma]\,\tau\,\sigma \ll}$$

$$\frac{\varepsilon \ll \vec{\sigma}}{\varepsilon \ll \varepsilon\,\vec{\sigma}} \qquad\qquad \frac{[\tau_f\,\sigma_f]\,\tau_2\,\sigma_2 \ll \vec{\sigma}}{[\tau_f\,\sigma_f]\,\tau_1\,\sigma_1 \ll [\tau_2\,\sigma_2]\,\tau_1\,\sigma_1\,\vec{\sigma}}$$

The meaning of the relation $\ll$ is that when we have $\sigma \ll \sigma_1 \ldots \sigma_n$, then any n computations annotated with $\sigma_1 \ldots \sigma_n$ can be safely run sequentially, and the entire sequence can be annotated with σ. Please take notice that (as displayed by the CPS translation) call-by-value function application consists of three sequential steps: evaluating the function, evaluating the argument, and executing the function applied to the argument. This is reflected in the APP-COMP typing rule.

Expressions of the form $\tau \leq^? \tau$ and $\sigma \leq^? \sigma$ will be called *subtyping constraints*, and expressions of the form $\sigma \ll^? \vec{\sigma}$ will be called *sequence constraints*. Inside the constraints, the syntax of effect annotations is extended by annotation variables δ.

We will be using substitutions defined on both the type variables and annotation variables. We say that a substitution s σ-grounds a set of (subtyping or sequence) constraints S if $s(S)$ do not contain any annotation variables. We say that a substitution s solves a constraint $\sigma_1 \leq^? \sigma_2$ ($\sigma_1 \ll^? \vec{\sigma}$) if and only if $s(\sigma_1) \leq s(\sigma_2)$ ($s(\sigma_1) \ll s(\vec{\sigma})$) is derivable. This implies that the substitution σ-grounds the constraint, because the definition of $\leq$ and $\ll$ does not include annotation variables.

Theorem 8 (Principal types) *For every term e and typing environment Γ there exist: α, δ, a set of subtyping constraints S, and a set of sequence constraints C, such that:*

1. *For every substitution s which solves S and C we have $\Gamma \vdash e : s(\alpha)\,s(\delta)$.*
2. *For every τ, σ satisfying $\Gamma \vdash e : \tau\,\sigma$ there exists a substitution s such that $s(\alpha) = \tau$, $s(\delta) = \sigma$ and s solves S and C.*

The principal type for the expression $\lambda f.\lambda x.f\,x; f\,x$, which we considered at the beginning of this subsection, is $(\alpha_1 \xrightarrow{\delta_1} \alpha_2) \xrightarrow{\delta_2} \alpha_3 \xrightarrow{\delta_3} \alpha_4\delta_4$ with constraints: $\alpha_3 \leq \alpha_1$, $\alpha_2 \leq \alpha_4$, $\delta_1 \leq \delta_5$, $\delta_1 \leq \delta_6$, $\delta_7 \ll \delta_5\delta_6$, $\delta_7 \leq \delta_3$, $\varepsilon \leq \delta_2$, $\varepsilon \leq \delta_4$. (A simpler type for this expression, which is an instance of the principal type and works well in practice, is $(\alpha_1 \xrightarrow{\delta} \alpha_2) \to \alpha_1 \xrightarrow{\delta} \alpha_2$ with constraint $\delta \ll \delta\delta$.)

The type inference algorithm (written in pseudocode) is shown in Figure 7. The function *infer_final* defined there takes two parameters: the type environment (which maps variable names to types) and the expression considered. It finds the principal

```
infer Γ e =
  match e with
    x → (Γ(x), ε, VAR)
    λx.e′ →
      let α = clean_tvar()
      let (τ, σ, D) = infer (Γ ∪ {x ↦ α}) e′
      (α →^σ τ, ε, ABS(D))
    e1 e2 →
      let (τ1, σ1, D1) = infer Γ e1
      let (τ2, σ2, D2) = infer Γ e2
      let (α1, α2) = (clean_tvar(), clean_tvar())
      let (δ1, δ2, δ3, δ) = (clean_avar(), clean_avar(), clean_avar(), clean_avar())
      let C1 = emit_constraint (τ1 σ1 ≤? α1 →^δ3 α2 δ1)
      let C2 = emit_constraint (τ2 σ2 ≤? α1 δ2)
      let C = emit_constraint (δ ≪? δ1δ2δ3)
      (α2, δ, APP(SUB(D1, C1), SUB(D2, C2) , C))
    S0 f.e′ →
      let (α1, α2, δ) = (clean_tvar(), clean_tvar(), clean_avar())
      let (τ, σ, D) = infer (Γ ∪ {f ↦ α1 →^δ α2}) e′
      (α1, [α2 δ] τ σ, SHIFT0(D))
    ⟨e⟩ →
      let (τ, σ, D) = infer Γ e
      let (α′, α, δ) = (clean_tvar(), clean_tvar(), clean_avar())
      emit_constraint (τ σ ≤? α′ [α′] α δ)
      (α, δ, RESET0(D))
infer_final Γ e =
  let (τ, σ, D) = infer Γ e
  let (α, δ) = (clean_tvar(), clean_avar())
  let C = emit_constraint (τ σ ≤? α δ)
  (α, δ, SUB(D, C))
```

Fig. 7 Type inference algorithm

type (the variables α and δ from Theorem 8 along with the unsolved constraints) in polynomial time and it also generates a typing derivation skeleton, where subtyping derivations have to be filled in. It works the same way as the standard type inference algorithm for the simply typed lambda calculus, only that instead of emitting type equality constraints (which can be solved immediately with the unification algorithm), it emits type inequality and annotation sequencing constraints (as a side effect).

In order to check if the term can actually be well-typed (and to obtain a concrete typing derivation), one has to find a substitution which solves the inequalities. It is easy to see that we can do the following simplification steps on the constraints:

- $\delta \leq^? \varepsilon$—substitute $\delta = \varepsilon$;
- $[\tau_1\, \sigma_1]\, \tau_2\, \sigma_2 \leq^? \delta$— substitute $\delta = [\alpha_1\, \delta_1]\, \alpha_2\, \delta_2$ and add new constraints $\alpha_1 \leq^? \tau_1$, $\delta_1 \leq^? \sigma_1$, $\tau_2 \leq^? \alpha_2$, $\sigma_2 \leq^? \delta_2$, where α_1, α_2, δ_1, δ_2 are fresh variables;
- $\alpha \leq^? \tau_1 \xrightarrow{\sigma} \tau_2$— substitute $\alpha = \alpha_1 \xrightarrow{\delta} \alpha_2$ and add new constraints $\tau_1 \leq^? \alpha_1$, $\alpha_2 \leq^? \tau_2$, $\delta \leq^? \sigma$, where α_1, α_2, δ are fresh variables;
- $\tau_1 \xrightarrow{\sigma} \tau_2 \leq^? \alpha$— substitute $\alpha = \alpha_1 \xrightarrow{\delta} \alpha_2$ and add new constraints $\alpha_1 \leq^? \tau_1$, $\tau_2 \leq^? \alpha_2$, $\sigma \leq^? \delta$, where α_1, α_2, δ are fresh variables;
- $\sigma_0 \ll^? \sigma_1 \ldots \sigma_n$ when for some i from 0 to n we have $\sigma_i = \varepsilon$— unify every σ_i with ε;

$$
\begin{aligned}
[\![\alpha]\!] &= \alpha \\
[\![\tau_1 \xrightarrow{\sigma} \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2\ \sigma]\!] \\
[\![\tau\ \varepsilon]\!] &= [\![\tau]\!] \\
[\![\tau\ [\tau_1\ \sigma_1]\ \tau_2\ \sigma_2]\!] &= ([\![\tau]\!] \to [\![\tau_1\ \sigma_1]\!]) \to [\![\tau_2\ \sigma_2]\!] \\
[\![\alpha \le \alpha]\!] &= \lambda x.x \\
[\![\tau_1' \xrightarrow{\sigma_1} \tau_1 \le \tau_2' \xrightarrow{\sigma_2} \tau_2]\!] &= \lambda f.\lambda x.[\![\tau_1\ \sigma_1 \le \tau_2\ \sigma_2]\!](f([\![\tau_2' \le \tau_1']\!]x)) \\
[\![\tau\ \varepsilon \le \tau'\ \varepsilon]\!] &= [\![\tau \le \tau']\!] \\
[\![\tau\ \varepsilon \le \tau'\ [\tau_1\ \sigma_1]\ \tau_2\ \sigma_2]\!] &= \lambda x.\lambda f.[\![\tau_1\ \sigma_1 \le \tau_2\ \sigma_2]\!](f([\![\tau \le \tau']\!]x)) \\
[\![\tau\ [\tau_1\ \sigma_1]\ \tau_1'\ \sigma_1' \le \tau'\ [\tau_2\ \sigma_2]\ \tau_2'\ \sigma_2']\!] &= \lambda f.\lambda g.[\![\tau_1'\ \sigma_1' \le \tau_2'\ \sigma_2']\!] \\
&\qquad (f(\lambda x.[\![\tau_2\ \sigma_2 \le \tau_1\ \sigma_1]\!](g([\![\tau \le \tau']\!]x))))
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]_{\text{VAR}} &= x \\
[\![e]\!]_{\text{SUB}(D,\tau\ \sigma \le \tau'\ \sigma')} &= [\![\tau\ \sigma \le \tau'\ \sigma']\!][\![e]\!]_D \\
[\![\lambda x.e]\!]_{\text{ABS}(D)} &= \lambda x.[\![e]\!]_D \\
[\![e_1 e_2]\!]_{\text{APP-PURE}(D_1,D_2)} &= [\![e_1]\!]_{D_1}[\![e_2]\!]_{D_2} \\
[\![e_1 e_2]\!]_{\text{APP}(D_1,D_2)} &= \lambda k.[\![e_1]\!]_{D_1}(\lambda f.[\![e_2]\!]_{D_2}(\lambda x.f x\, k)) \\
[\![\mathcal{S}_0 f.e]\!]_{\text{SHIFT0}(D)} &= \lambda f.[\![e]\!]_D \\
[\![\langle e \rangle]\!]_{\text{RESET0}(D)} &= [\![e]\!]_D(\lambda x.x)
\end{aligned}
$$

Fig. 8 Type-directed CPS translation for $\lambda^{\mathcal{S}_0}_{\le}$

– $\sigma_0 \ll^? \sigma_1 \dots \sigma_n$ when for some i from 0 to n we have $\sigma_i = [\_]\ \_$ — get $n+1$ pairs of fresh variables $\alpha_0\ \delta_0, \dots, \alpha_n\ \delta_n$, unify σ_0 with $[\alpha_n\ \delta_n]\ \alpha_0\ \delta_0$, for every i from 1 to n unify σ_i with $[\alpha_i\ \delta_i]\ \alpha_{i-1}\ \delta_{i-1}$.

The only constraints remaining unsolved after the simplifications are of the form $\alpha_1 \le^? \alpha_2$, $\delta_1 \le^? \delta_2$, $\varepsilon \le^? \delta$, $\delta \le^? [\tau_1\ \sigma_1]\ \tau_2\ \sigma_2$, $\delta_0 \ll^? \delta_1 \dots \delta_n$. They can be dealt with, e.g., using backtracking. The solution can be used to instantiate the type derivation skeleton returned from the algorithm in Figure 7.

3.5 Type-directed CPS translation

Having extended the type system with subtyping, we can derive from it a type-directed CPS translation, which takes into account the additional information given by the subtyping. This translation, defined in Figure 8, targets the simply typed lambda calculus. The derivations of the subtyping relation are translated to coercion functions, which are typed as expected:

Lemma 14 *If $\tau\ \sigma \le \tau'\ \sigma'$, then $\vdash [\![\tau\ \sigma \le \tau'\ \sigma']\!] : [\![\tau\ \sigma]\!] \to [\![\tau'\ \sigma']\!]$ in $\lambda_\to$. If $\tau \le \tau'$, then $\vdash [\![\tau \le \tau']\!] : [\![\tau]\!] \to [\![\tau']\!]$ in $\lambda_\to$.*

The translation preserves types:

Theorem 9 (Type preservation) *If D is a typing derivation of $\Gamma \vdash e : \tau\ \sigma$ in $\lambda^{\mathcal{S}_0}_{\le}$, then $[\![\Gamma]\!] \vdash [\![e]\!]_D : [\![\tau\ \sigma]\!]$ in $\lambda_\to$.*

An interesting property of the translation is that it keeps terms annotated as pure in direct style. This is similar to the selective translations presented in [25], [31], and recently in [36], which also use effect annotations to distinguish pure terms from effectful ones.

Unfortunately, the translation does not preserve reductions. For example, let us consider the term $\langle \mathcal{S}_0 f.f \rangle$. We have $\vdash \langle \mathcal{S}_0 f.f \rangle : \tau \to \tau$ and $\langle \mathcal{S}_0 f.f \rangle \to \lambda x.\langle x \rangle$. For the simplest (which means: the number of inference rules used in it is the smallest) derivation D, we have

$$[\![\langle \mathcal{S}_0 f.f \rangle]\!]_D = (\lambda f.f)\,(\lambda x.x)$$

and for the simplest derivation D', we have

$$[\![\lambda x.\langle x \rangle]\!]_{D'} = \lambda x.(\lambda x.\lambda k.(\lambda x.x)\,(k\,((\lambda x.x)\,x)))\,x\,(\lambda x.x).$$

So we cannot β-reduce $[\![\langle \mathcal{S}_0 f.f \rangle]\!]_D$ to $[\![\lambda x.\langle x \rangle]\!]_{D'}$, although they are β-equal. For this reason, this CPS translation could not be used as-is for proving termination of evaluation of well-typed terms in $\lambda^{\mathcal{S}_0}_{\leq}$.

4 Programming examples

4.1 Prototype implementation

In order to study the applications of the type system $\lambda^{\mathcal{S}_0}_{\leq}$, presented in this article, to practical programming, we have developed its prototype implementation. The implementation adds some standard syntactic sugar to the language and extends it with integers, algebraic data types and recursion.

We extend the syntax as follows:

$$\begin{aligned} e ::= \ldots &\mid \mathrm{true} \mid \mathrm{false} \mid \mathrm{if}\ e\ \mathrm{then}\ e\ \mathrm{else}\ e \\ &\mid \mathrm{nil} \mid \mathrm{cons}(e,e) \mid \mathrm{case}\ e\ \{\mathrm{nil} \to e; \mathrm{cons}(x_1,x_2) \to e\} \\ &\mid n \mid e \oplus e \mid e \gtreqless e \mid \mathrm{fix}x.e \\ \tau ::= \ldots &\mid \mathrm{int} \mid \mathrm{bool} \mid \mathrm{list}(\tau) \end{aligned}$$

The typing rules for new expressions introduced above are given in Figure 9.

In the implementation language, the string `@f.e` means $\mathcal{S}_0 f.e$, and `<e>` means $\langle e \rangle$. The rest of the syntax used is ML-like.

4.2 Example—*prefixes*

The *prefixes* example shown in Figure 10 has been considered in [2] and in [6]. The function `prefixes` lists all of the prefixes of a given list, e.g., `prefixes [1,2,3]` yields `[[1],[1,2],[1,2,3]]`. The idea behind the program is that the continuation represents the currently considered prefix of the given list, and the function uses it to emit the result below the delimiter.

This example cannot be typed in the original type system of Danvy and Filinski, because the captured continuation is applied in contexts with different answer types. In our type system, this problem is taken care of by the subtyping relation. In the first

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}\ \text{TRUE} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}\ \text{FALSE}$$

$$\frac{\Gamma \vdash b : \text{bool}\ \sigma_1 \qquad \Gamma \vdash e_1 : \tau\ \sigma_2 \qquad \Gamma \vdash e_2 : \tau\ \sigma_2 \qquad \sigma \ll \sigma_1\ \sigma_2}{\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \tau\ \sigma}\ \text{IF} \qquad \frac{}{\Gamma \vdash n : \text{int}}\ \text{INT}$$

$$\frac{\Gamma \vdash n_1 : \text{int}\ \sigma_1 \qquad \Gamma \vdash n_2 : \text{int}\ \sigma_2 \qquad \sigma \ll \sigma_1\ \sigma_2}{\Gamma \vdash n_1 \oplus n_2 : \text{int}\ \sigma}\ \text{ARITH}$$

$$\frac{\Gamma \vdash n_1 : \text{int}\ \sigma_1 \qquad \Gamma \vdash n_2 : \text{int}\ \sigma_2 \qquad \sigma \ll \sigma_1\ \sigma_2}{\Gamma \vdash n_1 \gtreqless n_2 : \text{bool}\ \sigma}\ \text{REL} \qquad \frac{}{\Gamma \vdash \text{nil} : \text{list}(\tau)}\ \text{NIL}$$

$$\frac{\Gamma \vdash h : \tau\ \sigma_1 \qquad \Gamma \vdash t : \text{list}(\tau)\ \sigma_2 \qquad \sigma \ll \sigma_1\ \sigma_2}{\Gamma \vdash \text{cons}(h, t) : \text{list}(\tau)\ \sigma}\ \text{CONS}$$

$$\frac{\Gamma \vdash e : \text{list}(\tau_l)\ \sigma_1 \qquad \Gamma \vdash e_1 : \tau\ \sigma_2 \qquad \Gamma, h : \tau_l, t : \text{list}(\tau_l) \vdash e_2 : \tau\ \sigma_2 \qquad \sigma \ll \sigma_1\ \sigma_2}{\Gamma \vdash \text{case } e\{\text{nil} \to e_1; \text{cons}(h, t) \to e_2\} : \tau\ \sigma}\ \text{CASE}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix } x.e : \tau}\ \text{FIX}$$

Fig. 9 Typing rules for booleans, integers, lists and recursion

occurrence (`k []`) the continuation is simply given the pure type list(int) → list(int). In the second (`k (w xs)`), it is given a more complex type list(int) $\xrightarrow{\text{[list(int)] list(int)}}$ list(int).

Another interesting thing in this example is that it still works correctly when the first reset inside the shift_0 in the `Cons` branch is removed—i.e. the original shift is replaced by shift_0.

4.3 Example—*partition*

The *prefixes* example does not take advantage of the semantics of shift_0, because it can be typed with flat effect annotations. This is not the case for *partition*, shown in Figure 11. This program partitions a list of integers into three parts: the ones less than, equal, and greater than a number given as a parameter, maintaining the original order between the integers in each group, e.g., `partition 3 [4,1,3,5,2,3]` yields `[1,2,3,3,4,5]`. The idea behind this example is that the two reset_0's are used as "markers" for the positions in the result where we insert new elements, and shift_0 is used to reach those markers.

The inner function `part` is given the following type:

$$\text{list(int)} \xrightarrow{\text{[list(int)] list(int) [list(int)] list(int)}} \text{list(int)}$$

Hence, it requires two contexts, both of the type list(int) → list(int), and returns a final value of type list(int). These contexts are the "accumulators" for the elements less than and equal to the given value.

The subtyping relation is crucial for typing this example. In particular, in the non-recursive `Nil` case, the following subtyping is used:

$$\text{list(int)} \leq \text{list(int) [list(int)] list(int) [list(int)] list(int)},$$

```
let prefixes xs =
   let w l = case l {
      Nil -> @k. Nil
    | Cons(x, xs) ->
        Cons(x, @k. <Cons(k [], <k (w xs)>)>)
   } in <w xs>
```

Fig. 10 Example program—*prefixes*

```
let partition a l =
   let part l = case l {
      Nil -> []
    | Cons(h,t) ->
        if h > a
        then Cons(h, part t)
        else if h == a
           then @f. Cons(h, <f (part t)>)
           else @f g. Cons(h, <g <f (part t)>>)
   } in <<part l>>
```

Fig. 11 Example program—*partition*

where the subtyping derivation for the effect annotations is as follows:

$$\dfrac{\dfrac{}{\text{list(int)} \le \text{list(int)}} \qquad \dfrac{\dfrac{}{\text{list(int)} \le \text{list(int)}} \qquad \dfrac{}{\varepsilon \le \varepsilon}}{\varepsilon \le [\text{list(int)}]\ \text{list(int)}}}{\varepsilon \le [\text{list(int)}]\ \text{list(int)}\ [\text{list(int)}]\ \text{list(int)}}$$

Using the translation from Figure 8 we obtain the following coercion function (reduced to normal form for clarity):

$$\lambda x.\lambda k_1.\lambda k_2.k_2\,(k_1\,x)$$

We can see that in the translated term the composition of the "accumulator" contexts after traversing the entire input list is done by the coercion function.

5 Relation to shift/reset

In this section, we discuss the influence of the presented theory for shift_0 and reset_0 on their relation with shift and reset.

5.1 Embedding of Danvy and Filinski's type system into $\lambda^{\mathcal{S}_0}_{\le}$

The operational semantics of shift_0 suggests that the shift operator can be defined with shift_0 and reset_0 by putting a reset_0 inside a shift_0: $\mathcal{S}f.e = \mathcal{S}_0 f.\langle e\rangle$; reset_0 then behaves just like reset. Using this definition, one can check how the type system presented here relates to the type system for $\mathsf{shift}/\mathsf{reset}$ introduced by Danvy and Filinski [14] (let us call their system $\lambda^{\mathcal{S}}_{\to}$). First, we define a translation from $\lambda^{\mathcal{S}}_{\to}$ to $\lambda^{\mathcal{S}_0}_{\le}$ on terms and types (the syntactic forms not mentioned below translate homomorphically):

$$\frac{}{\Gamma, x : \tau; \tau' \vdash x : \tau; \tau'}\ \text{VAR}$$

$\Longrightarrow$

$$\frac{\dfrac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{VAR} \qquad \dfrac{\dots}{\tau \leq \tau\ [\tau']\ \tau'}}{\Gamma, x : \tau \vdash x : \tau\ [\tau']\ \tau'}\ \text{SUB}$$

$$\frac{\Gamma, x : \tau; \tau_1 \vdash e : \tau'; \tau_2}{\Gamma; \tau'' \vdash \lambda x.e : (\tau_{\tau_1} \rightarrow_{\tau_2} \tau'); \tau''}\ \text{ABS}$$

$\Longrightarrow$

$$\frac{\dfrac{\Gamma, x : \tau \vdash e : \tau'\ [\tau_1]\ \tau_2}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{[\tau_1]\ \tau_2} \tau'}\ \text{ABS} \qquad \dfrac{\dots}{\tau \xrightarrow{[\tau_1]\ \tau_2} \tau' \leq \tau \xrightarrow{[\tau_1]\ \tau_2} \tau'[\tau'']\ \tau''}}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{[\tau_1]\ \tau_2} \tau'[\tau'']\ \tau''}\ \text{SUB}$$

$$\frac{\Gamma; \tau_4' \vdash e_1 : (\tau_{1\,\tau_1'} \rightarrow_{\tau_3'} \tau_2); \tau_2' \qquad \Gamma; \tau_3' \vdash e_2 : \tau_2; \tau_4'}{\Gamma; \tau_1' \vdash e_1 e_2 : \tau_2; \tau_2'}\ \text{APP}$$

$\Longrightarrow$

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

$$\frac{\Gamma, f : (\tau_{\tau'} \rightarrow_{\tau'} \tau_1); \tau_3 \vdash e : \tau_3; \tau_2}{\Gamma; \tau_1 \vdash \mathcal{S} f.e : \tau; \tau_2}\ \text{SHIFT}$$

$\Longrightarrow$

$$\frac{\dfrac{\dfrac{\Gamma, f : \tau \xrightarrow{[\tau']\ \tau'} \tau_1 \vdash e : \tau_3\ [\tau_3]\ \tau_2}{\Gamma, f : \tau \xrightarrow{[\tau']\ \tau'} \tau_1 \vdash \langle e \rangle : \tau_2}\ \text{RESET0}}{\Gamma \vdash \mathcal{S}_0 f.\langle e \rangle : \tau\ [\tau_1\ [\tau']\ \tau']\ \tau_2}\ \text{SHIFT0} \qquad \dfrac{\dots}{\tau\ [\tau_1\ [\tau']\ \tau']\ \tau_2 \leq \tau\ [\tau_1]\ \tau_2}}{\Gamma \vdash \mathcal{S}_0 f.\langle e \rangle : \tau\ [\tau_1]\ \tau_2}\ \text{SUB}$$

$$\frac{\Gamma; \tau_1 \vdash e : \tau_1; \tau}{\Gamma; \tau' \vdash \langle e \rangle : \tau; \tau'}\ \text{RESET}$$

$\Longrightarrow$

$$\frac{\dfrac{\Gamma \vdash e : \tau_1\ [\tau_1]\ \tau}{\Gamma \vdash \langle e \rangle : \tau}\ \text{RESET0} \qquad \dfrac{\dots}{\tau \leq \tau\ [\tau']\ \tau'}}{\Gamma \vdash \langle e \rangle : \tau\ [\tau']\ \tau'}\ \text{SUB}$$

Fig. 12 Embedding of Danvy and Filinski's type system $\lambda^{\mathcal{S}}_{\rightarrow}$ into $\lambda^{\mathcal{S}_0}_{\leq}$

$$\begin{aligned}\overline{\tau} &= \tau\\ \overline{\tau_1\ \tau_3 \to_{\tau_4} \tau_2} &= \overline{\tau_1} \xrightarrow{[\overline{\tau_3}]\ \overline{\tau_4}} \overline{\tau_2}\\ \overline{\mathcal{S}f.e} &= \mathcal{S}_0 f.\langle \overline{e} \rangle\end{aligned}$$

Using this translation, one can prove the following result:

Theorem 10 *If $\Gamma; \tau_1 \vdash e : \tau; \tau_2$ in $\lambda^{\mathcal{S}}_{\to}$, then $\overline{\Gamma} \vdash \overline{e} : \overline{\tau}\ [\overline{\tau_1}]\ \overline{\tau_2}$ in $\lambda^{\mathcal{S}_0}_{\leq}$.*

One can also consider the CPS translations in both systems $\lambda^{\mathcal{S}}_{\to}$ and $\lambda^{\mathcal{S}_0}_{\leq}$. If we introduce the translation on typing derivations as shown in Figure 12, we notice that the CPS translations of the typing derivations resulting from this translation are β-equal to the standard CPS translations for shift and reset [14]. In order to formalize this observation we introduce the following notation. If D is a typing derivation in $\lambda^{\mathcal{S}}_{\to}$, then $\overline{D}$ is the corresponding typing derivation in $\lambda^{\mathcal{S}_0}_{\leq}$, obtained using the translation inductively defined in Figure 12.

Lemma 15 *Let D_e be a typing derivation for the term e in $\lambda^{\mathcal{S}}_{\to}$. Then the following equalities hold:*

- $[\![\overline{\mathcal{S}f.e}]\!]_{\overline{D_{\mathcal{S}f.e}}} =_\beta \lambda k.[\![\overline{e}]\!]_{\overline{D_e}}\{\lambda x.\lambda k'.k'\,(k\,x)/f\}(\lambda x.x)$
- $[\![\overline{\langle e \rangle}]\!]_{\overline{D_{\langle e \rangle}}} =_\beta \lambda k.k([\![\overline{e}]\!]_{\overline{D_e}}(\lambda x.x))$

As a corollary, we can prove by straightforward induction the following theorem that coupled with Theorem 10 materializes the folklore relation of shift and shift_0 in the typed setting:

Theorem 11 *Let e be a well-typed term in $\lambda^{\mathcal{S}}_{\to}$, and let D be its typing derivation. Then $[\![e]\!] =_\beta [\![\overline{e}]\!]_{\overline{D}}$.*

5.2 Embedding of Biernacka and Biernacki's type system into $\lambda^{\mathcal{S}_0}_{\leq}$

We can also consider the type system of [6], which we call $\lambda^{\mathcal{S}}_{\rhd}$. Again, we first define a translation on terms, types and context types (as before, syntactic forms not mentioned below are translated homomorphically):

$$\begin{aligned}\overline{b} &= \tau_b\\ \overline{\tau_1\ \tau_3 \to_{\tau_4} \tau_2} &= \overline{\tau_1} \xrightarrow{[\overline{\tau_3}]\ \overline{\tau_4}} \overline{\tau_2}\\ \overline{\tau_1 \rhd \tau_2} &= \overline{\tau_1} \to \overline{\tau_2}\\ \overline{k \hookleftarrow e} &= k\,\overline{e}\\ \overline{\mathcal{S}k.e} &= \mathcal{S}_0 k.\langle \overline{e} \rangle\end{aligned}$$

Then, the following theorem holds:

Theorem 12 *If $\Gamma, \Delta; \tau_1 \vdash e : \tau; \tau_2$ in $\lambda^{\mathcal{S}}_{\rhd}$, then $\overline{\Gamma} \cup \overline{\Delta} \vdash \overline{e} : \overline{\tau}\ [\overline{\tau_1}]\ \overline{\tau_2}$ in $\lambda^{\mathcal{S}_0}_{\leq}$.*

$$\frac{}{\varepsilon \leq \varepsilon}\ \text{S-EPS} \qquad \frac{\tau \leq \tau'}{\varepsilon \leq [\tau]\,\tau'}\ \text{S-PURE} \qquad \frac{\tau_2 \leq \tau_1 \qquad \tau_1' \leq \tau_2'}{[\tau_1]\,\tau_1' \leq [\tau_2]\,\tau_2'}\ \text{S-NPURE}$$

$$\frac{\tau \leq \tau' \qquad \sigma \leq \sigma'}{\tau\ \sigma \leq \tau'\ \sigma'}\ \text{S-STR} \qquad \frac{}{\alpha \leq \alpha}\ \text{S-VAR} \qquad \frac{\tau_2' \leq \tau_1' \qquad \tau_1\ \sigma_1 \leq \tau_2\ \sigma_2}{\tau_1' \xrightarrow{\sigma_1} \tau_1 \leq \tau_2' \xrightarrow{\sigma_2} \tau_2}\ \text{S-FUN}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma \vdash e : \tau\ \sigma \qquad \tau\ \sigma \leq \tau'\ \sigma'}{\Gamma \vdash e : \tau'\ \sigma'}\ \text{SUB}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2\ \sigma}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\sigma} \tau_2}\ \text{ABS} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ \text{APP-PURE}$$

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

$$\frac{\Gamma, f : \tau_1 \to \tau_2 \vdash e : \tau_3}{\Gamma \vdash \mathcal{S}_0 f.e : \tau_1\ [\tau_2]\,\tau_3}\ \text{SHIFT0} \qquad \frac{\Gamma \vdash e : \tau'\ [\tau']\,\tau}{\Gamma \vdash \langle e \rangle : \tau}\ \text{RESET0}$$

Fig. 13 Type system for $\lambda^{\mathcal{S}_0}$ with flat effect annotations ($\lambda^{\mathcal{S}}_{\leq}$)

For both type systems for shift considered here, there exist terms which are not well-typed in those systems, but their translation to $\lambda^{\mathcal{S}_0}_{\leq}$ is well-typed. In particular, the following term:

$$(\lambda f.\lambda y.(\lambda z.\langle(\lambda v.\lambda w.y)\,(f\,y)\rangle)\,\langle f\,y\rangle)\,(\lambda x.x)$$

cannot be typed in either of the two systems. However, it translates without changes to $\lambda^{\mathcal{S}_0}_{\leq}$, where it is well typed (for example, it can be assigned the type $\alpha \to (\beta \to \alpha)$). In this sense, the type system presented here is strictly more expressive than the other systems.

5.3 A type system for shift_0 and reset_0 with flat effect annotations

An interesting feature of the translations of Sections 5.1 and 5.2 is that a non-empty type annotation never occurs inside another type annotation—the annotations are "flat." Thus one can consider a variant of the $\lambda^{\mathcal{S}_0}_{\leq}$ type system, which enforces this restriction syntactically:

$$\tau ::= \alpha \mid \tau \xrightarrow{\sigma} \tau$$
$$\sigma ::= \varepsilon \mid [\tau]\,\tau$$

The typing rules for this system (called $\lambda^{\mathcal{S}}_{\leq}$) are shown in Figure 13. For this type system Theorem 10 and 11 still hold.

In this type system shift_0 is forbidden from being used inside another shift_0 without a corresponding control delimiter and, therefore, only the top context of the stack can be accessed by control operations. It follows that in this type system shift_0 has the operational semantics of shift. To see this, we observe that if D is a typing derivation of $\Gamma \vdash \mathcal{S}_0 f.e : \tau_1\ [\tau_2]\,\tau_3$, then by using a coercion $\tau_3 \leq \tau_3\ [\tau_3]\,\tau_3$, we obtain a typing

derivation D' of $\Gamma \vdash \mathcal{S}_0 f.\langle e\rangle : \tau_1\ [\tau_2]\ \tau_3$ such that $[\![\mathcal{S}_0 f.e]\!]_D =_\beta [\![\mathcal{S}_0 f.\langle e\rangle]\!]_{D'}$. We can thus view this type system as an exotic type system for the shift operator (that is why we call it $\lambda^{\mathcal{S}}_{\leq}$).

5.4 A new untyped CPS translation for shift and reset

The untyped CPS translations of Section 2.2 also induce CPS translations for shift and reset. For instance, for $\mathcal{S}f.e$ expressed as $\mathcal{S}f.\langle e\rangle$ the curried CPS translation of Figure 1 yields:

$$[\![\mathcal{S}f.e]\!] = \lambda f.[\![e]\!]\,(\lambda x.\lambda k.k\,x)$$
$$[\![\langle e\rangle]\!] = [\![e]\!](\lambda x.\lambda k.k\,x)$$

It is interesting to observe that although this CPS translation validates the standard reduction semantics of shift and reset:

$$\langle v\rangle \rightsquigarrow v$$
$$\langle K[\mathcal{S}f.e]\rangle \rightsquigarrow \langle e\{\lambda x.\langle K[x]\rangle/f\}]\rangle$$

it generates a different equational theory than the one of the standard CPS translation for shift and reset [23]. For example, the equation

$$\langle\langle e\rangle\rangle = \langle e\rangle$$

is not sound with respect to the presented CPS translation.

5.5 Static simulations of shift_0 and reset_0

The untyped uncurried CPS translation of Figure 2 can be modified slightly so that the head of the list of contexts is captured by a separate lambda abstraction. This modified translation gives rise to definitions of shift_0 and reset_0 operators in the continuation monad with the following (recursive) answer type (where τ is an arbitrary type):

$$\mathrm{Ans} = \mathrm{List}\,(\tau \to \mathrm{Ans}) \to \tau$$

Therefore, shift_0 and reset_0 can be represented using the monadic reflection (μ) and reification ($[\![\cdot]\!]$) operators that, as shown by Filinski [19], can be defined using shift and reset. When specialized for the continuation monad, they read as follows:

$$\mu(e) := \mathcal{S}k.\lambda c.e\,(\lambda x.k\,x\,c)$$
$$[\![e]\!] = \langle(\lambda x.\lambda k.k\,x)\,e\rangle$$

Then we obtain the following simulation of shift_0 and reset_0:

$$\begin{aligned}
\mathrm{stop} &:= \lambda x.\lambda[\,].x\\
\mathrm{pop} &:= \lambda x.\lambda(k : ks).k\,x\,ks\\
\mathrm{cont} &:= \lambda f.\lambda x.\mu(\lambda k.\lambda ks.f\,x\,(k : ks))\\
\\
\mathcal{S}_0 f.e &:= \mu(\lambda k.\lambda(k' : ks).(\lambda f.[\![e]\!])\,(\mathrm{cont}\,k)\,k'\,ks)\\
\langle e\rangle_0 &:= \mu(\lambda k.\lambda ks.[\![e]\!]\ \mathrm{pop}\,(k : ks))\\
\mathrm{run}\mathcal{S}_0\ e &:= [\![e]\!]\ \mathrm{stop}\,[\,]
\end{aligned}$$

In the above, stop defines the initial continuation, pop is the empty continuation pushed by reset_0, and run$\mathcal{S}_0$ starts the enclosed computation by providing the initial continuation. This simulation coincides with the one presented by Shan in [37].

Using the CPS translation in Figure 1, one can define a new static simulation of shift_0 and reset_0. As before, let us first define the answer type for the continuation monad:

$$\mathrm{Ans} = \tau + ((\tau \to \mathrm{Ans}) \to \mathrm{Ans})$$

That is, the answer type is expressed as a sum type, where the left summand represents the answer type of the top-level continuation and the right summand represents the answer type of the remaining continuations. The simulation is then defined as follows (where the sum-type specific operations are left implicit):

$$\begin{aligned}
\mathrm{stop} &:= \lambda x.x \\
\mathrm{pop} &:= \lambda x.\lambda k.k\,x \\
\mathrm{cont} &:= \lambda f.\lambda x.\mu(f\,x) \\
\\
\mathcal{S}_0 f.e &:= \mu(\lambda k.(\lambda f.[\![e]\!])\,(\mathrm{cont}\,k)) \\
\langle e\rangle_0 &:= \mu([\![e]\!]\ \mathrm{pop}) \\
\mathrm{run}\mathcal{S}_0\ e &:= [\![e]\!]\ \mathrm{stop}
\end{aligned}$$

Inlining stop, pop, cont and the monadic reflection and reification operators we get the following simulation using shift and reset:

$$\begin{aligned}
\mathcal{S}_0 f.e &:= \mathcal{S}k.\lambda c.(\lambda f.\langle(\lambda x.\lambda k.k\,x)\,e\rangle)(\lambda x.\mathcal{S}k'.\lambda c'.k\,x\,c\,(\lambda x'.k'\,x'\,c')) \\
\langle e\rangle_0 &:= \mathcal{S}k.\lambda c.\langle(\lambda x.\lambda k.k\,x)\,e\rangle\,(\lambda x.\lambda k.k\,x)\,(\lambda x.k\,x\,c) \\
\mathrm{run}\mathcal{S}_0\ e &:= \langle(\lambda x.\lambda k.k\,x)\,e\rangle\,(\lambda x.x)
\end{aligned}$$

6 Related work

6.1 Relation to the Scala implementation

The type system and the selective CPS-translation presented here resemble strongly those presented by Rompf et al. in [36] for the programming language Scala. The presentation in [36] is based on Scala and is somewhat informal, so the exact comparison is not possible, but still one can see the following similarities:

- Our annotations (from the type system $\lambda^{\mathcal{S}}_{\leq}$) correspond to `@cps` annotations. For example, our type $\alpha\,[\beta]\,\gamma$ corresponds to `A @cps[B, C]` in Scala.
- The type system of Scala distinguishes pure and impure expressions as our type system does. In particular, as in our restricted system with flat annotations, the expression inside a shift must have a pure type, the entire expression has an impure type, and the captured continuation has a pure function type.
- Scala's `Shift` class used for implementing delimited control, which implements the continuation monad, is constructed by shift and consumed by reset, in the same way that in our translation shift_0 introduces a lambda abstraction and reset_0 eliminates it.
- Our annotation composition relation $\ll$ introduced in Section 3.4 corresponds to the *comp* control effect composition operation.

The type system in [36] does not allow shift to be used inside another shift—it has to be put inside a new reset. This is what happens with the shift_0 operator with flat effect annotations in our type system, which suggests that Scala actually implements the restricted $\mathsf{shift}_0/\mathsf{reset}_0$ operators that only happen to operationally coincide with shift/reset.

6.2 Substructural type system of Kiselyov and Shan

The type system for $\mathsf{shift}_0/\mathsf{reset}_0$ introduced by Kiselyov and Shan [27] shares many properties with the type system presented in this work—in particular, strong type soundness and termination. In their work, expressions and evaluation contexts are treated as structure in linear logic, where structural rules play a vital role. Moreover, their type system abstractly interprets (in the sense of abstract interpretation of Cousot and Cousot [11]) small-step reduction semantics of the language and it does not require effect annotations in judgments and arrow types. Instead these annotations occur in types and cotypes assigned to terms and coterms, respectively. In terms of presentation, our system is very close to the well-known type system by Danvy and Filinski and seems more conventional than Kiselyov and Shan's. Both type systems allow for answer type modifications and for exploring the stack of contexts beyond the nearest control delimiter. While in our type system subtyping is built in and plays a major role, leading to an interesting CPS translation, in Kiselyov and Shan's work it is just an entailment of the type system.

Our type system is equivalent in expressive power to the type system of Kiselyov and Shan. To embed our type system into theirs, we first need to replace all occurrences of the reset_0 operator $\langle e \rangle$ by $\# \, \$ \, e$. We take the following translations for types and annotated types:

$$\begin{aligned} \overline{\alpha} &= \alpha \\ \overline{\tau \xrightarrow{\sigma'} \tau'} &= \overline{\tau} \to \overline{\tau' \; \sigma'} \\ \overline{\tau \; \varepsilon} &= \overline{\tau} \\ \overline{\tau \; [\tau' \; \sigma'] \; \tau'' \; \sigma''} &= (\overline{\tau} \uparrow \overline{\tau' \; \sigma'}) \downarrow \overline{\tau'' \; \sigma''} \end{aligned}$$

Taking these translations, the proof is straightforward.

In the other direction, we need to represent the \$ operator of Kiselyov and Shan in our type system. We can do it using the following translation:

$$\overline{C \, \$ \, E} = \langle (\lambda x. \mathcal{S}_0 z. \overline{C} \; x) \; \overline{E} \rangle$$

We also need to represent coterms, but they can be converted to equivalent terms easily. The proof is more involved than the other direction, but still pretty straightforward.

We have studied some properties of the \$ operator formed by the translation above (slightly modified to account for possible control effects in the expression on the left of the \$) in [29]. In particular, we have shown that this operator can be used (and is particularly well suited) for expressing the CPS hierarchy of Danvy and Filinski [14] inside $\mathsf{shift}_0/\mathsf{reset}_0$.

6.3 Monadic Framework for Delimited Continuations of Dybvig et al.

Dybvig, Peyton Jones and Sabry designed and implemented (in Scheme and Haskell) a general framework for delimited continuations [16]. The paper defines, among others, a CPS translation and an abstract machine. Many kinds of delimited control operators can be defined within the framework, including the four variations described in [37]: shift/reset, $\mathsf{shift}_0/\mathsf{reset}_0$, Felleisen's control/prompt and its variant, $\mathsf{control}_0/\mathsf{prompt}_0$. The framework also allows to use many kinds of delimiters (called prompts in the paper) and dynamically create new ones. It is achieved by representing the metacontext as a list of contexts interspersed with identifiers of prompts.

It is an interesting question how the CPS translation and abstract machine for the framework of Dybvig et al. relates to the ones defined in this paper. It turns out the relationship is rather complex, mostly owing to the generality of the framework. The definitions of the CPS translation and abstract machine in [16] can be specialized to define a language with $\mathsf{shift}_0/\mathsf{reset}_0$ operators and only one prompt (for the reset_0 operator). The resulting definitions place many unnecessary empty contexts on the metacontexts; these can be removed manually from the definitions. Then one can simplify the definitions further by changing the representation of the metacontext to just a list of contexts. Finally, the current context can be moved to the top of the metacontext. The resulting abstract machine is the same (except for inessential details) as the one presented in Figure 3, and the resulting CPS translation corresponds to the uncurried CPS translation in Figure 2.

7 Conclusion

We have presented a new type system for the delimited-control operators shift_0 and reset_0 that has been derived from a CPS translation for these control operators and that generalizes Danvy and Filinski's type system for shift and reset. The type system is equipped with two subtyping relations: one on types and the other one on effect annotations describing a stack of contexts in which a given expression can be embedded. The subtyping mechanism makes it possible to coerce pure expressions into effectful ones.

The effect annotations and the corresponding subtyping relation are used to guide a selective CPS translation that precisely takes into account the information about the contexts surrounding a translated expression. In particular, it leaves pure expressions in direct style. Such a translation seems promising, if one would like to implement full-fledged shift_0 and reset_0, e.g., in Scala (instead of their flat version that accidentally coincides with shift and reset [36]). Our type annotations translate to Scala by nesting the `@cps` annotations already used by the current implementation of delimited control. For example, the effect-annotated type $\alpha\ [\alpha]\ \alpha\ [\alpha]\ \alpha$ would translate to Scala's type `A @cps[A, A @cps[A, A]]`.

The present article also demonstrates the effectiveness of the context-based method of reducibility predicates [5,6] that has turned out to be the right way to tackle the non-trivial problem of termination of evaluation for the case of the type-and-effect system with subtyping considered here. It can be observed (and remains to be verified, e.g., in Coq) that the computational content of this constructive proof takes the form of a type-directed evaluator in CPS that corresponds to the type-directed CPS translation presented in this work.

The type system presented in this work is monomorphic, but it forms a foundation for an extension to a full-fledged type-and-effect system with parametric polymorphism and subtyping. We expect that, as in the polymorphic type system of Asai and Kameyama presented in [2], it should be safe to generalize the types of pure terms. Also, because of the presence of subtyping, one might need to store constraints about generalized variables in the polymorphic types, as in [22].

Another aspect of this work that deserves attention are equational theories of shift_0 and reset_0 with respect to the presented CPS translations, both in the untyped and typed settings. In particular, it seems vital to understand the role of the subtyping relation in the equational theories of the typed control operators.

Acknowledgements First of all, we would like to thank Małgorzata Biernacka for numerous comments on the presentation of this work as well as for helping us to get the proof of termination of evaluation of Section 3.3 under control. Hugo Herbelin made the observation about the difference between the standard equational theory for shift and reset and the one induced by the curried untyped CPS translation of Section 2.2.

We also thank the anonymous referees of HOSC as well as the anonymous referees and the PC members of ICFP'11 for their valuable comments on the presentation and technical contents of this article. In particular, we are grateful to Kenichi Asai for his careful proofreading of the journal version of the article and for his friendly assistance during the publishing process.

This work has been supported by Polish NCN grant number DEC-011/03/B/ST6/00348.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.
2. Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In Zhong Shao, editor, *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems, APLAS'07*, number 4807 in Lecture Notes in Computer Science, pages 239–254, Singapore, December 2007.
3. Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.
4. Chris Barker. Continuations in natural language. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 1–11, Venice, Italy, January 2004.
5. Małgorzata Biernacka and Dariusz Biernacki. A context-based approach to proving termination of evaluation. In *Proceedings of the 25th Annual Conference on Mathematical Foundations of Programming Semantics(MFPS XXV)*, Oxford, UK, April 2009.
6. Małgorzata Biernacka and Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. In Francisco J. López-Fraguas, editor, *Proceedings of the 11th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, Coimbra, Portugal, September 2009. ACM Press.
7. Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005.
8. Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007.
9. Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.

10. Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Technical Report BRICS RS-05-16, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, May 2005.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977. ACM Press.
12. Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.
13. Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.
14. Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.
15. Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.
16. R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.
17. Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.
18. Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.
19. Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.
20. Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.
21. Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.
22. Fritz Henglein. Syntactic properties of polymorphic subtyping. Technical Report Semantics Report D-293, DIKU, Computer Science Department, University of Copenhagen, May 1996.
23. Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, SIGPLAN Notices, Vol. 38, No. 9, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.
24. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.
25. Jung-taek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In Greg Morrisett, editor, *Record of the 1998 ACM SIGPLAN Workshop on ML and its Applications*, Baltimore, Maryland, September 1998.
26. Oleg Kiselyov and Chung-chieh Shan. Delimited continuations in operating systems. In Boicho Kokinov, Daniel C. Richardson, Thomas R. Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007*, number 4635 in Lecture Notes in Artificial Intelligence, pages 291–302, Roskilde, Denmark, August 2007. Springer.
27. Oleg Kiselyov and Chung-chieh Shan. A substructural type system for delimited continuations. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007*, number 4583 in Lecture Notes in Computer Science, pages 223–239, Paris, France, June 2007. Springer-Verlag.
28. Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In Olivier Danvy, editor, *Proceedings of the 2011 ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, pages 81–93, Tokyo, Japan, September 2011. ACM Press.

29. Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. In Ranjit Jhala and Atsushi Igarashi, editors, *Proceedings of the Tenth Asian Symposium on Programming Languages and Systems, APLAS'12*, number 7705 in Lecture Notes in Computer Science, pages 296–311, Kyoto, Japan, December 2012.
30. John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.
31. Lasse R. Nielsen. A selective CPS transformation. In Stephen Brookes and Michael Mislove, editors, *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 201–222, Aarhus, Denmark, May 2001. Elsevier Science Publishers.
32. Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, pages 202–206. Springer-Verlag, 1999.
33. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
34. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [35].
35. John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.
36. Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In Andrew Tolmach, editor, *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 317–328, Edinburgh, UK, August 2009. ACM Press.
37. Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.
38. Eijiro Sumii. An implementation of transparent migration on standard Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Technical Report 00-368, Rice University, pages 61–64, Montréal, Canada, September 2000.
39. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.